Towards Comprehensive Web Search

Erik Warren Selberg

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

Doctor of Philosophy

University of Washington

1999

Program Authorized to Offer Degree: Computer Science & Engineering

© Copyright 1999
Erik Warren Selberg

University of Washington

Graduate School

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

Erik Warren Selberg

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.

Chair of Supervisory Committee:

____

Oren Etzioni

Reading Committee:

____

Steve Tanimoto

____

Efthimis Efthimiadis

Date: __________________________

In presenting this dissertation in partial fulfillment of the requirements for the Doctorial 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 this thesis is allowable only for scholary purposes, consistant with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature________________________

Date________________________

University of Washington

Abstract

Towards Comprehensive Web Search

by Erik Warren Selberg

Chair of Supervisory Committee

Associate Professor Oren Etzioni
Department of Computer Science & Engineering

The World Wide Web has rapidly become a key medium for information dissemination to all members of society. However, its disorganized nature and sheer size can make it difficult for people to find information. Web search services have made a significant contribution towards enabling people to quickly find information on the Web. Unfortunately, as of this writing, no Web search service can conduct a comprehensive search of the Web for any topic.

In addition, many major Web search services are unable to return a stable set of results. An intuitive assumption about the behavior of any search service is that the results of a given query will be unchanged unless either the documents referred to in the results change and become irrelevant or better documents become available. Unfortunately, due to a variety of real-world constraints and design choices, many search services are unstable, intermittently omitting relevant documents from search results even though the documents are contained within their indices.

One technique that could enable both a more comprehensive search of the Web as well as a more stable search is *meta-search*. Meta-search is conducting a single search using multiple search resources. This thesis examines the application of meta-search

to the World Wide Web. It answers the following questions: can meta-search provide a more comprehensive search than traditional Web searching? Is meta-searching necessary now, and will it be necessary in the future? Can meta-searching be implemented in a practical manner? And can meta-search enable a more stable search?

We present MetaCrawler, a meta-search service, as a means of obtaining a more comprehensive search than existing Web search services. MetaCrawler addresses some of the issues with Web search service through forwarding a user query and combining the results from multiple search services into a single list.

To summarize the results of this thesis, we conclude that MetaCrawler demonstrates that meta-search can be implemented in a manner such that average Web users will take advantage of the benefits of meta-search. We also conclude that MetaCrawler demonstrates that meta-search can be provided and maintained with limited resources. Through our experiments using Inference of User Value through Real-world data, a new methodology to evaluate search services, we conclude that MetaCrawler provides a significantly more comprehensive search than any single search service. Furthermore, the growth trends of the Web and search service indices lead us to conclude that meta-search will be necessary to provide a comprehensive search in the future. Our experiments also demonstrate that most major search services are unstable, and may omit relevant documents even if those documents are present in their indices. Finally, we conclude that search results can not only be made stable, but can be improved through Collaborative Index Enhancement, a novel model for enhancing a searchable index based on the experience of previous users.

TABLE OF CONTENTS

List of Figures **vi**

List of Tables **viii**

Chapter 1: Introduction **1**

1.1 Motivation 1

1.2 Overview 2

1.2.1 Practical implementation 4

1.2.2 More comprehensive Web search through meta-search 4

1.2.3 Continued likelihood of meta-search benefits 5

1.2.4 Stable search 6

1.3 Scientific contributions 7

1.4 Organization 8

1.5 Conventions used in this thesis 12

1.5.1 Typographic convention for functions and libraries 12

1.5.2 Search engines vs. search services 12

Chapter 2: Contemporary Web Search Techniques **13**

2.1 The Web Search Problem 14

2.2 Web browsing agents 14

2.3 Traditional Information Retrieval search engines 18

2.3.1 HTML 18

2.3.2 Indexing 19

2.3.3 Retrieval . . . 19
2.4 The Web Discovery Problem . . . 22
2.4.1 Collective user histories . . . 22
2.4.2 Distributed search . . . 24
2.4.3 Spiders . . . 25
2.5 The Coherence Problem . . . 27
2.6 Real-world constraints and Web search engines . . . 28
2.7 Summary . . . 31

Chapter 3: MetaCrawler Architecture **32**
3.1 MetaCrawler overview . . . 33
3.1.1 Understanding query and output formats . . . 35
3.1.2 Collation and duplicate detection . . . 36
3.1.3 Highest common denominator of services . . . 37
3.1.4 Highest quality results and fastest information location time . . . 38
3.2 Design goals . . . 38
3.3 Architectural layout . . . 39
3.3.1 Client Layer . . . 41
3.3.2 Server Layer . . . 42
3.3.3 I/O Layer . . . 44
3.4 Integrating user-defined search resources . . . 44
3.4.1 Wrapper libraries and templates . . . 45
3.4.2 Dynamic Service Protocol . . . 46
3.5 Architecture evaluation . . . 49
3.5.1 Expandable . . . 49
3.5.2 Low maintenance cost . . . 50
3.5.3 Fast response time . . . 51

3.5.4 Scalability 51
3.5.5 Portability 52
3.6 Summary 53

Chapter 4: MetaCrawler Implementation **54**
4.1 MetaCrawler user interface 55
4.1.1 Search form 55
4.1.2 Feedback with server-push and Java 57
4.1.3 Result page with click logging 60
4.2 Data fusion algorithm 61
4.2.1 Biased interleave methods 62
4.2.2 Normalize-Distribute-Sum algorithm 63
4.3 Heuristics for duplicate detection 64
4.3.1 Default duplicate detection 65
4.3.2 Redirect Heuristic 66
4.3.3 Mirror Heuristic 67
4.4 Parallel I/O 69
4.4.1 Naive process or thread-based approaches 70
4.4.2 Event-based paradigm 71
4.4.3 Event-based paradigm with DNS 75
4.4.4 Handling other protocols 76
4.4.5 Evaluation of event-based paradigm 76
4.4.6 Results of event-based model 83
4.5 Further extensions to MetaCrawler 83
4.5.1 Ahoy! The HomePage Finder 83
4.5.2 HuskySearch 86
4.6 Summary 89

Chapter 5: Empirical Evaluation **90**

5.1 Comprehensiveness of Web search services 91

5.1.1 Inference of User Value through Real-world Data 93

5.1.2 The 1995 search service evaluation 95

5.1.3 The 1999 search service evaluation 100

5.1.4 Independent confirmation 103

5.2 Longevity of meta-search 110

5.2.1 Calculating the size of the Web 111

5.2.2 Search service and Web growth trends 113

5.3 Instability of Web search service 116

5.3.1 Repeated query study 116

5.3.2 Experimental results 118

5.3.3 Analysis 124

5.3.4 Observations about individual services 126

5.3.5 Impact of unstable results 126

5.4 Summary 127

Chapter 6: Collaborative Index Enhancement **129**

6.1 Using collaboration to improve retrieval performance 130

6.1.1 Information retrieval through direct collaboration 130

6.1.2 Information retrieval through indirect collaboration 131

6.1.3 A general model of indirect collaboration 132

6.2 Collaborative Index Enhancement design 133

6.2.1 Additional information and improved ranking 135

6.2.2 Convenient query expansion 136

6.2.3 Scaling CIE 140

6.2.4 Hardware and software 141

6.3 Experimental validation 141
6.3.1 User study 142
6.3.2 Log analysis 151
6.4 Summary 161

Chapter 7: Conclusions and Future Work **164**
7.1 Summary 164
7.1.1 Practical implementation of meta-search 165
7.1.2 Significant contribution of meta-search 166
7.1.3 Longevity of meta-search 166
7.1.4 Stable search 167
7.2 Future work 168
7.2.1 Qualitative analysis 168
7.2.2 Improving inference of user value 169
7.2.3 Query routing 169
7.2.4 Alternative instances of CIE 170
7.2.5 Beyond HTML 171
7.2.6 Information integration 172
7.2.7 Outside the box 173

Bibliography **175**

Appendix A: Queries from Lawrence and Giles study **189**

Appendix B: CIE User Survey Form **191**

LIST OF FIGURES

2.1 Formal description of the Web Search Problem. 15
2.2 A sample HTML document. 20
2.3 Creating an inverted index. 21
2.4 Formal description of the Web Discovery Problem. 23
2.5 A simple spider algorithm. 26
2.6 Sizes of search services' indices (in millions of pages). 30
3.1 MetaCrawler control flow. 34
3.2 MetaCrawler architecture. 40
3.3 Clio and MetaCrawler integration. 48
4.1 MetaCrawler v1.0 Homepage screenshot, Jan. 1996. 58
4.2 MetaCrawler v1.5 Homepage screenshot, Aug. 1996. 59
4.3 The components of a URL. 65
4.4 FSM for a general HTTP connection. 72
4.5 FSM network for general HTTP connections. 74
4.6 FSM for HTTP with DNS. 77
4.7 Overhead of threads and processes for busywait benchmark. 80
4.8 Overhead of threads and processes for file I/O benchmark. 81
4.9 HuskySearch Homepage screenshot, Aug. 1999. 88
5.1 An example of search service overlap. 92
5.2 Unique Document Percentage, 1995. 97

5.3 Viewed Document Share, 1995. 99
5.4 Unique Document Percentage, 1995 and 1999. 101
5.5 Viewed Document Share, 1995 and 1999. 102
5.6 Cumulative Document Percentage. 104
5.7 Cumulative Viewed Percentage. 105
5.8 Percentage of viewed documents returned by one, two, and three or more services. 106
5.9 Number of Web servers on the Internet. 114
5.10 Trends of the size of search service indices. 115
5.11 Top 200 and Top 10 results using default options. 119
5.12 Average change over time for Top 10 URLs. 121
5.13 Percentage of URLs removed temporarily. 123

6.1 CIE architecture. 134
6.2 Sample HuskySearch results from the query "Utah Jazz." 138
6.3 Accuracy of result judgment. 147
6.4 Time to answer each question (Hours:Min). 148
6.5 View Rate of each CIE auxiliary. 153
6.6 View Rate for ClickedURLs and 3 representative Web search services. 156
6.7 Cumulative Overlap Percentage through 7.2 million documents. . . . 162

LIST OF TABLES

4.1 Redirect Heuristic example 1. 68

4.2 Redirect Heuristic example 2. 68

4.3 Mirror Heuristic example 1. 69

4.4 Default, adjusted, and actual maximum number of processes, threads, and open connections. 82

4.5 Top ranked and found percentages for Ahoy! evaluation. 85

5.1 Coverage of search services. 108

5.2 Unique Document Percentages for Bharat and Broder. 109

5.3 Change over one month for Default, Phrase, and AllPlus options, averaged across all services for Top 10 and Top 200. 120

5.4 Histogram of viewed document ranks. 124

6.1 Percentage of users accurately answering each question. 144

6.2 Uniqueness of queries. 145

6.3 Percentage of users making secondary and tertiary queries. 150

6.4 Additional viewed documents contributed by ClickedURLs and three representative services. 158

6.5 Average Rank, Median Rank, and standard deviation for viewed documents. 160

ACKNOWLEDGMENTS

This thesis would not be possible were it not for the love and support of a great many people. First and foremost, I would like to acknowledge Mary Kaye Rodgers, who has not only made the past five years the best five years of my life, but has helped me make the best of these past few years.

I would not be where I am today without the sage advice of my advisor, Oren Etzioni. Oren taught me how to be a good researcher and ask the right questions when investigating a problem. Many thanks also go to the members of my reading committee, Efthimis Efthimiadis and Steve Tanimoto, who helped make this thesis a reality. I would also like to thank Raya Fidel, who gave me many valuable insights into the world of Information Retrieval, and Steve Hanks, who made me type "`mkdir thesis`." The user evaluation of CIE would not be possible without the efforts of Sam Oh and Allyson Carlyle, both of whom donated their classes. I would also like to thank my many colleagues who proofread this thesis under rather harsh time constraints: Lauren Bricker, Rich Segal, Marc Friedman, Greg Barnes, and Elizabeth Walkup. Also, thanks to Ed Lazowska for providing me with lots of insight into how things are done.

Thanks go out to the people who made MetaCrawler and HuskySearch operational for the past five years. The support people in our department, Erik Lundberg, Nancy Johnson Burr, Warren Jessop, and Voradesh Yenbut, as well as the many folks in Computers & Communications, especially Terry Gray, Melody Winkle, Steve Corbato, Art Dong, Ali Marashi, Rick White, and Tom Profit all helped tremendously in keeping MetaCrawler and HuskySearch running smoothly, allowing me to work on

my research. Thanks also to the other graduate students and undergrads who worked on MetaCrawler and HuskySearch: Oren Zamir, Melissa Johnson, Zhenya Sigal, Greg Lauckhart, Christin Boyd, Darren Schack, and Tim Bradley.

This thesis would not have been completed without the support of so many friends who made my time at the University of Washington so exciting and memorable. Thanks to Ruben Ortega and Lauren Bricker, Rich and Joanna Segal, Sean Sandys, Denise Pinnel, Andy and Debbie Berman, Lara Lewis, Dave "Pardo" Keppel, Shuichi Koga, Keith Golden and Ellen Spertus, Dawn Werner, Steve Wolfman, Rachel Pottinger, and Zach Ives. I'd also like to thank the members of the Spuds Softball Team and Disc Drives Ultimate Team for many enjoyable games, and the members of Fifth Element, especially those who stuck with me through the great unpleasantness: Lst, DarkTroll, Joust, Fallon, Jooky, HugeGuy, Monk, and Phatty, for helping me blow off a great deal of steam these past two years.

Finally, I would like to thank my family for all their support these many years. I would especially like to thank my great-aunt and uncle Junie and John Marcinkevich for their love and support while I've been in Seattle, my mother Joy, my father Ryan, and brother Scott everything they have done for me.

DEDICATION

In memory of John Peter Selberg.

Chapter 1

INTRODUCTION

1.1 Motivation

While the World Wide Web began as a convenient method for scientists to disseminate information to one another, the Web has rapidly become a key medium for information dissemination for everyone. Indeed, the Web has been likened to a "digital library," and to a searchable 15-billion word encyclopedia [6]. As pointed out by Lawrence and Giles, "Web search engines have made the large and growing body of scientific literature and other information resources accessible within seconds" [66]. However, the Web is unstructured, with no consistent organization. A common analogy is that the Web is like the world's largest library, but without a catalog. Therein lies the heart of the *Web Search Problem*: How can a user find all relevant information about a topic that exists on the Web?

Currently, Web search services such as Excite [37] or Yahoo! [40] are used to locate information on the Web. In the 10th Web Survey done by the Georgia Institute of Technology's Graphics, Visualization, and Usability Center, 84.8% out of 16,891 respondents used search services to find Web pages [56]. Considering that in December 1998, Excite reported an average of 58 million page-views per day with 20 million registered users [38], and Yahoo! reported 167 million page-views per day and 35 million registered users [111], it is clear that these services have a substantial impact on millions of people daily.

A desired feature of these general search services, often assumed to exist by naive

users, is that these search services are able to locate any piece of information available on the Web. However, numerous studies have shown that no single search service has indexed the entire Web [92, 66, 12]. In fact, the largest percentage of the Web any search service has been estimated to index is approximately 50% [12]. Therefore, currently no single service can knowingly provide all the information on the Web germane to a particular query. However, if no *single* Web search service can provide all of the germane information, perhaps *multiple* Web search services can.

Another desired feature of these search services, assumed by both naive and experienced users alike, is that the results returned by these search services are *stable*. Stable in this context means that the results returned by submitting a given query do not change unless either the documents referred to in the results change or more appropriate documents become available. We show in Section 5.3 that most of the major Web search services are unstable, which directly impacts the quality of the results obtained by a user.

This thesis explores the use of *meta-search* to provide both a more comprehensive Web search than any single Web search service as well as a more stable search. Meta-search, sometimes called federated search, is defined as conducting a single search using multiple heterogeneous search resources and combining the results. Our claim is that meta-search provides both a significantly more comprehensive Web search than existing Web search services and can enable a stable search even if the underlying search services are unstable.

1.2 Overview

We address four key questions regarding comprehensive search of the Web in this thesis. First we examine whether meta-search can be implemented in a practical manner for users. Second, we examine if meta-search contributes significantly to comprehensive Web search. Then we examine whether it is likely that meta-search

will continue to provide improved comprehensiveness relative to single Web search services, and finally if meta-search can provide a stable search even if the underlying search services are unstable.

Our approach is to design and build a prototype Web meta-search engine and deploy it on the Web. Implementing a publicly available meta-search engine allows us to demonstrate that meta-search is practical. We then evaluate the four questions using the prototype by performing various experiments and user studies.

We designed two prototype meta-search systems. The original is called MetaCrawler [44], and was operational as a public meta-search service at the University of Washington from June 1995 through November 1996. In November 1996, MetaCrawler was licensed to Go2Net, Inc., which took over daily operation. In January 1997, we deployed HuskySearch [93], a more advanced research meta-engine derived from the original MetaCrawler code base. As part of the HuskySearch system, we developed Collaborative Index Enhancement, a general model for improving search results through user interaction.

The HuskySearch service remains in operation by the author as of the writing of this thesis. This thesis only describes work done on MetaCrawler while MetaCrawler was in operation by the author at the University of Washington. Work done on the commercial version of MetaCrawler will not be discussed.

MetaCrawler was designed to determine if meta-search could be done in a practical manner, and to evaluate how much more comprehensive meta-search is than a single search service. Collaborative Index Enhancement is a general model that was designed to enable HuskySearch to provide a stable search, as well as improve the quality of search results over time. Ensuring that MetaCrawler and HuskySearch were able to address the questions introduced many challenges in their design. The following sections detail these challenges.

1.2.1 Practical implementation

Comprehensive meta-search provides little value unless it can be used by people who desire it. Therefore, our first goal was to design and implement a meta-search system that anyone on the Web could and would use. Users interact with most contemporary Web search services by entering a query and receiving results in under a few seconds. Users have become accustomed to receiving results quickly. Therefore, in order to attract and retain users, a meta-engine must also return results quickly. Users are also attuned to the quality of results. Most search services return results in a *ranked relevancy list*, where documents are ranked according to the service's confidence of their relevance. If results from heterogeneous services are to be collated into a single ranked relevancy list, users will expect there to be no duplicate documents, and that the ranking does make sense to them.

Meta-search services must make several external Web queries. These queries consume a non-trivial number of resources. As more and more users interact with the service, more and more queries are issued. Resources are limited, therefore we have optimized MetaCrawler and HuskySearch to use as few resources as possible.

An actively used meta-search service allows for a great deal of empirical testing. In addition to determining if any particular search service can provide a comprehensive search, a number of related hypotheses can be evaluated. In particular, we are able to determine if the utilized search services each return results not returned by others, and we can determine if the results returned by those search services are useful to users.

1.2.2 More comprehensive Web search through meta-search

If meta-search can be practically implemented, the next question is whether it adds any significant value to the user. Our hypothesis is that meta-search provides a significantly more comprehensive search than any single search service. First, we

must establish whether the various search services are providing some information that other search services are not. There is little to be gained by using a search service whose information is subsumed by others. To determine if the search services we use are all providing results that other search services do not provide, we analyze the documents returned by the Web search services from user queries as well as the documents viewed by the user. We show that each Web search services returns a largely unique set of documents, and each Web search service returns documents that the user views.

After we show that each search service is providing some inherent contribution, we then address how much benefit meta-searching provides over any subset of those services. We use the documents returned by each search service as an upper bound of the performance of any single search service. We then show the percentage of documents made available through the results of only one service, then through the combined results of two, then three, and so on. These percentages show that each search service contributes a significant number of available documents. In a similar fashion, we also show that each search service contributes a significant number of viewed documents.

1.2.3 Continued likelihood of meta-search benefits

It might be argued that traditional Web search services will eventually index the entire Web, and thus the contributions of meta-searching toward comprehensive Web search, however impressive today, will soon be obviated. We cannot predict what leaps in technology and surprising discoveries may come, and thus this conjecture may be true. Meanwhile, we can extrapolate when a Web search service will index the entire Web, or if a leap in technology will be required.

Four quantities are necessary to determine if and when a single search service will be able to provide comprehensive Web search: the size of the Web, how fast the Web is growing, the size of the search service's index, and how fast the index is growing.

Growth in this context is an amalgam of new Web pages being created, existing Web pages being removed, and existing Web pages being changed. Estimates on the size of the Web can be made based on determining how many documents two search service indices have in common. The rate of Web growth can be extrapolated from multiple Web size estimates. Another estimate of Web growth is just the growth rate of Web servers. Historical data on the sizes of Web search service indices are readily available, and future trends can be extrapolated from those sizes.

We present the data on the size of Web search service indices and predict their growth over nine months. We then present two estimates of the size of the Web. We present two estimates on the growth of the Web, one based on the size estimates, and the other based on the growth rate of Web servers. From these trends, we show that no search service will index the entire Web. Thus, meta-searching will continue to provide substantial benefit towards comprehensive search.

1.2.4 Stable search

There is no guarantee that meta-search provides a complete solution to obtaining comprehensive Web search. Meta-search is largely dependent on the quality of the results from the underlying search services. One of the difficulties with using large public Web services is that they undergo constant internal change. Some changes, such as modifications to their output format, are readily apparent. Other changes, such as the contents of their index or their ranking algorithm, are more subtle.

An intuitive assumption about the behavior of a search service is that the results of a given query will be stable. A result that is relevant should not be ranked lower unless it is no longer relevant or a better result becomes available. We challenge this assumption, and show that it does not hold for a majority of the major contemporary search services. Furthermore, we show that through Collaborative Index Enhancement, a general method for improving search results by collecting information about user sessions, we are not only able to ensure a consistent set of results, but improve

the quality of the results overall.

1.3 Scientific contributions

We now summarize the main scientific contributions presented in this thesis:

The MetaCrawler architecture

A fundamental question regarding meta-search is how to build a meta-search engine. We present the MetaCrawler architecture, an architecture for a scalable Web meta-search engine.

Inference of User Value through Real-world Data

Evaluation of the Web is difficult due to the lack of standard test data sets. This thesis introduces Inference of User Value through Real-world Data, a new method to evaluate Web search services based on observing of a large number of user sessions. We take advantage of the large number of users and user queries and our ability to query multiple services in order to provide meaningful evaluation of Web search services.

A better understanding of the behavior of Web search services

Using our methodology and meta-search engine, we present a number of empirical evaluations of contemporary Web search services. Through these evaluations, we are able to make some generalizations about their characteristics and behavior.

A scalable model for fast parallel Web page retrieval

Contemporary parallel Web page retrieval methods use either a multiple process or multiple thread model, which are inherently limited to about a thousand simultaneous page retrievals. This thesis describes a model that enables retrieving over four thousand Web pages simultaneously using a single process.

The Normalize-Distribute-Sum algorithm

We present *Normalize-Distribute-Sum*, a novel algorithm that collates results from ranked relevancy lists that use different ranking methods. This algorithm takes advantage of available ranking information in the results.

Collaborative Index Enhancement

This thesis presents Collaborative Index Enhancement, a novel model for capturing and incorporating information about existing Web pages through the observation of user accesses. We use this method to explore four different instantiations of the Collaborative Index Enhancement model.

1.4 Organization

The remainder of this thesis is organized as follows:

Chapter 2: Contemporary Web Search Techniques

The Web Search Problem is formally defined in Chapter 2. We then discuss various automatic browsing techniques that can provide relevant information, but do not guarantee to provide that information in a timely manner. We also describe how traditional Information Retrieval engines can address the Web Search Problem. There are two difficulties with using traditional engines. The first is the Web Discovery Problem: How can all Web pages be located? The second is the Coherence Problem: How can the index be kept up to date with the Web? We highlight the major techniques for addressing the Web Discovery Problem: collective user histories, distributed search, and spiders. Using spiders, a traditional Information Retrieval engine can provide a comprehensive search. The Coherence Problem can also be addressed through polling documents and reindexing when necessary.

Unfortunately, there are numerous real-world constraints that limit the ability of a spider-based search service to provide a fully comprehensive search. We describe

how the cost of providing and maintaining a comprehensive index can be prohibitive. We show the size of the largest search service indices, which demonstrates that there is potential improvement in combining them through meta-search.

Chapter 3: MetaCrawler Architecture

Chapter 3 presents MetaCrawler, a meta-search engine, as a means of obtaining a more comprehensive search than existing Web search services. MetaCrawler addresses some of the issues with spider-based search service through forwarding a user query and combining the results from multiple search services into a single list. We detail MetaCrawler's architecture, which is designed to promote expandability, low maintenance, performance, scalability, and portability, and we describe how the architecture are able to accommodate those goals.

Chapter 4: MetaCrawler Implementation

Even though it appears to be straightforward to create a meta-search engine, implementing one that is fast, efficient, and scalable is not trivial. Chapter 4 goes into further technical depth of MetaCrawler. We show that average Web users can take advantage of comprehensive search through a clean Web interface. A problem with combining results from different search services is that the results may be biased in some fashion. We present the Normalize-Distribute-Sum algorithm that collates results from heterogeneous search services that takes into account the potential biases of various services. Another problem with combining the results from different services is that the same URL may be described in different ways. We present the Redirect Heuristic and Mirror Heuristic that classify common cases of duplication from redirects and mirroring.

Parallel retrieval of Web pages consumes a significant amount of system resources. We show how using an event-based paradigm enables an application to download over four thousand pages at once, compared to the common alternative of threads

which allows for slightly over a thousand simultaneous retrievals on workstation-class computers. Finally, we present two applications that build on MetaCrawler: Ahoy! and HuskySearch, which implement various features requested by users such as known item searching, persistent options, and query logic for locating people.

Chapter 5: Empirical Evaluation

In Chapter 5, we demonstrate the value of meta-search through a number of experiments. Using Inference of User Value through Real-world Data, a new methodology that enables us to measure the performance of a search service, we show that no single search service was comprehensive in 1995 and in 1999. While the available search services have changed, the advantages gained through meta-search are just as valuable in 1999 as they were in 1995.

Using various estimates of the sizes of the Web and search services, we extrapolate the rate of growth of the Web as well as the rate of growth of three largest search services available. We predict that not only will Web search services be unable to fully index the Web by the end of 2000, but that even if the indices of the three largest search services were completely disjoint, combining them will not be enough to provide a comprehensive Web search.

Finally, through direct observation we demonstrate that not only do search services provide an incomplete search of the Web, but that the results are often unstable. We show that a third of the documents returned in the Top 10 results by five of the nine major search services are removed from the search results over the course of one month, only to be returned to the results of subsequent queries.

Chapter 6: Collaborative Index Enhancement

In Chapter 6 we present Collaborative Index Enhancement, or CIE, a model for enhancing a searchable index based on the experience of previous users. The key idea behind CIE is to take the results document from a query and feed it back into

the source index or indices in some manner. We demonstrate a prototype system based on the HuskySearch search service that implements CIE using auxiliary search indices. This implementation allows us to use and experiment with several different CIE methods at once, without the need to modify or even control the original Web indices we use.

We present a series of experiments using both a user study as well as our Inference of User Value through Real-world Data methodology. Our results show that CIE contributes approximately 2% of the documents viewed by users. While a modest figure, we demonstrate that on a large scale system, after a short period of time over 50% of the documents viewed had been viewed in a previous session. This shows that there is potential for some CIE system to be of benefit in a large scale system. Our results suggest that CIE is a useful addition to HuskySearch, and promises to provide additional benefit over a long duration. To our knowledge, this is the first experimental evaluation of any Collaborative Index Enhancement method applied to the Web.

Chapter 7: Conclusions and Future Work

Finally, Chapter 7 summarizes the results presented and gives directions for future work. We conclude that MetaCrawler demonstrates that meta-search can be implemented in a manner such that average Web users will take advantage of the benefits of meta-search. We also conclude that MetaCrawler demonstrates that meta-search can be provided and maintained with limited resources. Through our experiments in Chapter 5, we conclude that MetaCrawler provides a significantly more comprehensive search than any single search service. Furthermore, the growth trends of the Web and search service indices lead us to conclude that meta-search will be necessary to provide a comprehensive search in the future. Our experiments also demonstrate that most major search services are unstable, and may omit relevant documents even if those documents are present in their indices. Finally, we conclude that search re-

sults can not only be made stable, but can be improved through Collaborative Index Enhancement.

1.5 Conventions used in this thesis

1.5.1 Typographic convention for functions and libraries

The algorithms and techniques presented in this thesis are not dependent on a given operating system or programming language. However, the prototype was implemented under a POSIX-compliant UNIX operating system using C++ and Perl. Occasionally, we will reference key libraries and system calls that affect the implementation. These libraries and functions will be presented in monospace type with the appropriate UNIX manual section appended in parenthesis, such as the `gethostbyname(3N)` system call or the `lib:LWP(3)` Perl library.

1.5.2 Search engines vs. search services

Contemporary literature often uses the term “search engine” to refer to different aspects of a Web site that provides access to a searchable index. Common usage of “search engine” is to refer to general popular Web sites with large, general purpose indices, such as AltaVista [30] or Excite [37]. In Information Retrieval literature, “search engine” typically refers to the part of a system that accepts a query and performs the search over an index. For the sake of clarity, we will use the term “Web search service” or just “service” to refer to the entire Web site. We will use the term “search engine” or “engine” to refer to the program that accepts a query and performs the lookup over a searchable index.

Chapter 2

CONTEMPORARY WEB SEARCH TECHNIQUES

The *Web Search Problem* is the problem of finding all information on the Web relevant to a given query. This chapter highlights work done to address the Web Search Problem. This chapter is organized as follows:

- We begin this chapter with a formal definition of the Web Search Problem.
- A common approach to addressing the Web Search Problem is through the use of agents. We highlight a selection of the key agent work in Section 2.2.
- Another common approach to addressing the Web Search Problem is through traditional Information Retrieval search engines. We highlight how traditional Information Retrieval engines are used to address the Web Search Problem in Section 2.3.
- The major drawback to using these engines is the *Web Discovery Problem*: How can all Web pages be located? We describe three major techniques to address the Web Discovery Problem in Section 2.4.
- Another problem with retrieval engines is the *Coherence Problem*: How can the contents of a searchable index be kept up to date with the Web? We present two commonly used techniques for addressing this problem in Section 2.5.
- Finally, we present some real-world constraints on Web search services in Section 2.6. These constraints limit how comprehensive Web search services can

be. This thesis addresses how to provide a more comprehensive search within those constraints.

2.1 The Web Search Problem

The Web Search Problem is a instance of the Information Retrieval Problem. The Information Retrieval Problem is defined as: given a set of documents and a query, determine the subset of documents relevant to the query. The Web Search Problem, outlined in Figure 2.1, is the problem of finding the set of documents on the Web relevant to a given user query. This problem accepts as input a set of Web documents and a query. Note that this set is not necessarily the entire Web. The output is a subset of all Web documents such that every document is relevant to the query. Like the Information Retrieval Problem, the Web Search Problem is difficult to resolve because relevance is a notion in each user's mind which may change during the course of a search, and thus must be approximated. The Web Search Problem is further compounded because there is no direct way to obtain all Web pages. Therefore, only a subset of available Web documents are given as input. Thus, relevant documents must somehow be located from given set of Web documents.

2.2 Web browsing agents

There are a wide variety of methods and techniques to address the Web Search Problem. The most straightforward is simple manual browsing. However, due to both the unstructured organization of the Web and its sheer size, manual browsing is often ineffective. Automatic browsing, on the other hand, might be an effective solution. A substantial quantity of work has been done exploring the creating of Web browsing agents, or programs that browse for a user and attempt to find relevant information. These agents treat the Web as a graph, with documents representing nodes and hyperlinks representing edges. We now highlight three representative techniques that

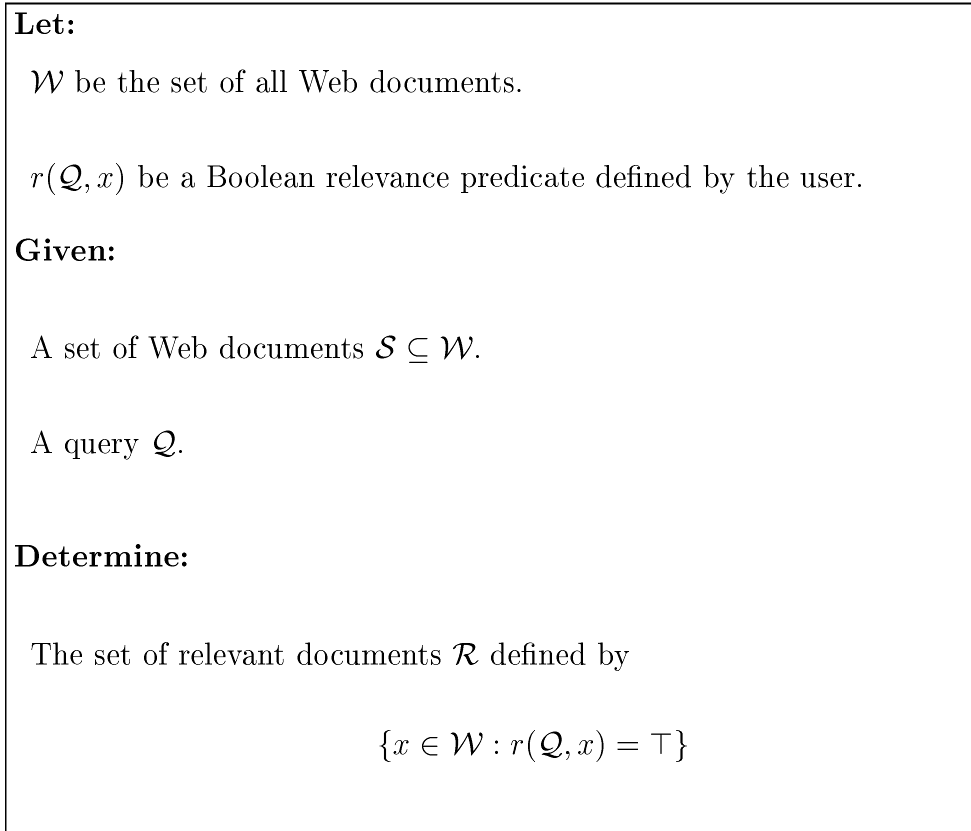


Figure 2.1: Formal description of the Web Search Problem.

use agents to find information through some kind of graph traversal.

De Bra and Post's *fish-search* explored automatic Web browsing by enhancing the Mosaic Web browser to automatically browse starting from the current search page to a certain depth [28]. When a user requested a search, fish-search would conduct an exhaustive depth-first search from the current page. The search could be limited to either a fixed period of time or until a certain number of relevant pages have been found. Although they provided a number of heuristics to optimize the search [27], Web server operators noted that fish-search was not far removed from exhaustive search. If all users were to employ a fish-search for every search, the overall demands would impose a substantial burden on Web servers.

A similar agent to fish-search is Letizia [67]. Letizia is a user interface agent that observes user behavior and attempts to locate useful items. Letizia doesn't use any heuristics to guide its search, but rather employs resource constraints, such as exploring only so many pages per minute, to restrict its search. Even though Letizia adheres to more resource constraints than fish-search, it has the potential of using more resources than fish-search. In a similar manner to fish-search, Letizia conducts a depth-first search through pages that the user is currently browsing. However, rather than searching on demand, Letizia constantly conducts and refines its search, and only returns results to the user on demand. Thus, Letizia consumes as many resources as possible as long as the user is browsing.

Another approach is Lamacchia's Internet Fish Construction Kit [62]. Users construct Internet Fish, or IFish, using the tools available from the kit. IFish are semi-autonomous agents that automatically browse the Web searching for relevant information. Unlike fish-search, which attempted to locate all available information at once, IFish use a casual model where relevant information is given to the user when it is found. There is no inherent time limit on how quickly the information should be retrieved, nor on how long any search can take. While a single IFish does not impose as large a load on Web servers as the fish-search model, users could and were encour-

aged to make multiple IFish for their various information needs. Thus if IFish were to catch on, all users would have multiple IFish running at the same time, causing a huge load on network resources.

The main advantage to using Web agents is how they handle relevancy. While users may not be able to perfectly communicate what it means for a document to be relevant to a given query, agents typically provide a very expressive language for users to define what they mean by relevant. For example, on a given query, one user may find a particular document relevant, and another may find it irrelevant. If both users used their own agents, they could each define their relevancy criteria for their own agent, and thus only the user who found the document relevant would see it.

The main drawback to using Web agents is that they consume a large amount of resources. A few well-designed agents do not pose a large problem for the Web. However, in large numbers even well-designed agents can overload a server. Even if overloading a server with requests was not a problem, the entire family of automatic browsing agents also suffer from one of two problems relating to the scale of the Web. The first problem relates to automatic browsing agents that require a promising path of Web pages to information that may be relevant. If there is no path available to that information, then the agent will not be able to discover that information. The second problem relates to agents that do not require a promising path. These agents typically use some kind of graph traversal algorithm, such as a depth-first or breadth-first traversal. Graph traversal algorithms may be a reasonable method to search through a few million pages. However, with a Web that consists of several hundred million pages, it is entirely possible, and likely, that the agent wanders in a sea of useless information for quite some time before finding any relevant information.

2.3 Traditional Information Retrieval search engines

An alternative approach to solving the Web Search Problem using Web browsing agents is to use traditional Information Retrieval search engines. These search engines locate information in a *corpus*, or set of documents, through a two step process. The first step is *indexing*, which converts the corpus into some kind of index that maps words to documents. The second step is *retrieval*, where a user query is used to lookup documents. The program that conducts the indexing is the *indexer*, and the program that facilitates the retrieval via the index is the *search engine*. Search engines can be used to locate information on the Web by using a set of Web documents as the corpus. They can then be indexed and searched like any other corpus of documents.

To illustrate the indexing and retrieval process, we overview in the following sections how Web documents are indexed using a simple inverted index, and how those documents are then retrieved. Indexing and retrieval are two of the fundamental research areas in the field of Information Retrieval, and thus readers who desire more information should consult some of the authoritative texts on the subject, such as the texts by Salton and McGill or van Rijsbergen [88, 108].

2.3.1 HTML

The Web is comprised of text files written in HyperText Markup Language, or HTML. A HTML file is simply a text file embedded with *markup tags* that describe various parts of the document. Markup tags are text delimited by `<` and `>`. Most markup tags appear in pairs around a region of text, with the opening tag being of the form `<word args>` and the closing tag being simply `</word>`, although some markup tags do not require a closing tag. One particular tag is the *anchor tag*, which allows an author to create a *hyperlink* to another document on the Web. A hyperlink combines a Uniform Resource Locator, or URL, with a piece of text that can be *clicked on*, or selected, by a user. If the user clicks on the text, the user's Web browser loads the

URL contained within the hyperlink. Figure 2.2 shows a sample HTML document. The HTML standard specifies that each document has a *Title*. The title is found between the `<title>` and `</title>` tags. The title of the document in Figure 2.2 is "A Sample Document." An option for HTML pages are META tags. Two pertinent META tags are the *keywords* tag and the *description* tag. The keywords meta tag contains a comma-separated list of key words and phrases for the document, and the description tag contains a short description of the page. These two tags are designed to aid the indexing of a page.

2.3.2 Indexing

HTML documents must be *cleaned* before they can be indexed. Cleaning refers to the removal of the document header and markup tags in the body of the text. In some cases, data contained within markup tags, such as the title, keywords, and description found in the head of the document, are kept and used as part of the indexing.

An inverted index is created from a document corpus by creating a mapping between each term in the document corpus to the documents that contain that word. For clarity, we will liken a term to a word, although a term can refer to other concepts such as a word stem, bigram, or synonym list. Figure 2.3 illustrates this technique. Common adaptations of this technique include term weighting and positional information with each document entry. Each entry in the inverted index is called a *key*.

2.3.3 Retrieval

The search engine accepts a query comprised of one or more keywords. The engine simply looks up which documents contain the keywords via the inverted index, and obtains a set of documents for each of the keywords. The sets of documents are then merged. The process of merging the lists for each word depends on the query semantics of each individual engine. For example, a Boolean engine requires the query be formulated using Boolean logic between keywords. Merging is then straightforward as

```
<html>
  <head>
    <title>A Sample Document</title>
    <meta name="keywords" content="sample document, html">
    <meta name="description" content="Just a sample document">
  </head>

  <body>

    <h1>This is a large header</h1>

    This is the text of a sample HTML document.

    <a href="http://www.cnn.com">This is a hyperlink to CNN</a>

  </body>
</html>
```

Figure 2.2: A sample HTML document.

A sample HTML document. The indentation is only for clarity.

Doc1:

Jazz defeat Portland...

Doc2:

Portland steamrolls slumping Lakers...

Doc3:

Jazz coach upset even though Jazz defeat Sonics...

Inverted Index:

coach	→	{Doc3, 2}
defeat	→	{Doc1, 2}, {Doc3, 7}
even	→	{Doc3, 4}
Jazz	→	{Doc1, 1}, {Doc3, 1}, {Doc3, 6}
Lakers	→	{Doc2, 4}
Portland	→	{Doc1, 3}, {Doc2, 1}
slumping	→	{Doc2, 3}
Sonics	→	{Doc3, 8}
steamrolls	→	{Doc2, 2}
though	→	{Doc3, 5}
upset	→	{Doc3, 3}

Figure 2.3: Creating an inverted index.

An inverted index is created from Doc1, Doc2, and Doc3. In this example, the position of the word is included in the index. This facilitates searching using positional logic, such as phrases.

the documents sets are combined using the appropriate Boolean logic. Another common method is to simply take the intersection or use a threshold, e.g. all documents that appear in 66% of the document sets.

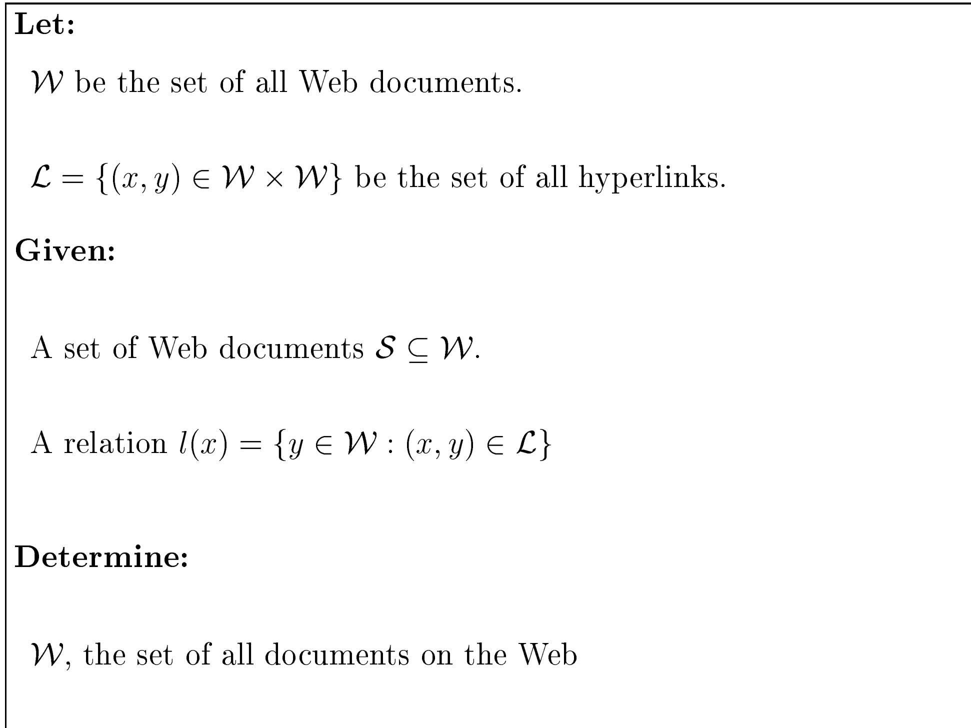
2.4 The Web Discovery Problem

The problem with using a traditional Information Retrieval approach for Web searching is that the indexer needs to have access to the documents in order to index them. However, there is no direct way to obtain all of the available documents on the Web. The *Web Discovery Problem* is the problem of obtaining all available documents on the Web. It takes as input a starting set of Web documents and a method of obtaining documents through hyperlinks, and returns as output the set of all available Web documents. This is highlighted in Figure 2.4. Assuming all Web documents have been found, a traditional Information Retrieval approach can then be used to solve the Web Search Problem. We now turn to work that has attempted to address this problem.

2.4.1 Collective user histories

One approach to the Web Discovery Problem is to make use of the browsing done by the user. One method is to collect and gather users' browsing histories. Lim's Coollist system is an early example of this technique [68]. The histories of users are sent to a Coollist Repository at the end of each browsing session. Lim's system defines a session by a fixed amount of idle time. A Coollist Library then merges the histories from the Coollist Repository into a single index. In addition, other indices can be created, such as indices that are comprised of users within a certain group.

A technique like this could potentially create a comprehensive index of the Web. It is reasonable to assume that the author of a Web page will visit it at least once to ensure that there are no errors in the HTML. Thus, all pages would be part of some



Let:

$\mathcal{W}$ be the set of all Web documents.

$\mathcal{L} = \{(x, y) \in \mathcal{W} \times \mathcal{W}\}$ be the set of all hyperlinks.

Given:

A set of Web documents $\mathcal{S} \subseteq \mathcal{W}$.

A relation $l(x) = \{y \in \mathcal{W} : (x, y) \in \mathcal{L}\}$

Determine:

$\mathcal{W}$, the set of all documents on the Web

Figure 2.4: Formal description of the Web Discovery Problem.

Note that solving this problem is predicated on the existence of a path to every element in $\mathcal{W}$ *from at least one element in* $\mathcal{S}$*. Thus, a poor choice of* $\mathcal{S}$ *will make this problem unsolvable.*

user's Coollist, and thus would be added to a global index. Unfortunately, there are two non-technical issues that make this approach impractical. The first problem is that users loath to divulge their complete browsing history due to privacy concerns. In some cases, such as users browsing from a corporate site, they may be legally restricted from divulging their browsing history. The second problem is that this solution requires users to have some kind of client program that sends the histories to a repository. Ensuring that all of the users on the Web have such a client is a huge deployment problem, and unlikely to be solved unless the client program is embedded in the browser or operating system.

2.4.2 Distributed search

Another way of solving the Web Discovery Problem is to index pages locally rather than move them to a central repository for indexing. This is known as *distributed search*. In a distributed search model, each Web server creates a searchable index of its local pages. Presumably, each Web server has, or can obtain, a list of files it provides. The Web server then does one of two things: it either transmits the index to a central repository to create a global index, or it transmits routing data to a central repository so that appropriate queries can be forwarded to the local server. We discuss each of these models in turn.

In the Harvest model, the index is sent to a central repository and merged with other indices to create a global index [91]. Naturally this assumes all Web servers are using the same indexer to create their local indices. Users are then able to search the entire Web through the comprehensive index. An advantage of this model is that resources required for retrieval can be combined in one central location, rather than spread out over myriads of Web servers. This makes it easier to operate and maintain the search service.

In the Query Routing model, only the keys of the index are sent to a repository [46, 98]. When a user issues a query at the repository, the query is forwarded to indices

that may have relevant information, and the results are merged. This spreads the cost of search resources across all Web servers. There are issues regarding the proper way to merge documents into a single list [7], but these are not insurmountable.

While distributed search appears to be a straightforward method to obtain a comprehensive Web index, it is a technology that is difficult to use. Distributed search requires that Web server operators index their site using a particular indexer. This puts a substantial burden on the operator. Even if that was not an issue, there is an assumption that a local indexer is able to create a comprehensive local index. The common assumption is that a local index can be created by traversing the file system on local or networked disks. However, files on a local disk do not necessarily correspond directly to files on the Web, and thus indexing a local disk may index files that are not available through the Web. Finally, the distributed search model assumes that the Web server operator is knowledgeable enough about his or her Web site to properly configure a local indexer. While this may be true of a small site, operators of large sites may be completely overwhelmed by the number and variety of pages at their site. Because of these issues, distributed search has not yet become a viable option for comprehensive Web search.

2.4.3 Spiders

A Web browsing agent is able to locate documents, but the time it may take to locate all relevant documents is undefined. A search engine is able to retrieve documents quickly, but it cannot locate documents directly. An obvious solution to the Web Search Problem is to use a Web browsing agent to gather all available documents, and then use a traditional indexer and search engine to search through those documents. Agents that gather documents are known as *spiders*, sometimes called Web robots or crawlers [79, 71]. A spider is an agent that traverses hyperlinks in an attempt of finding all available Web pages. A spider takes as input a starting pool of URLs and an document index, which is typically some kind of inverted index. Until the pool of

```
Spider(URL pool url_pool, Document index index)
    while ( url_pool not empty)
        url ← pick URL from url_pool
        doc ← download url
        new_urls ← extract urls from doc
        insert doc into index
        insert url into indexed_urls
        for each u ∈ new_urls
            if u ∉ indexed_urls
                append u to url_pool
```

Figure 2.5: A simple spider algorithm.

URLs is empty, the spider removes a URL from the pool and downloads the document to which it refers. It then extracts all URLs from the document, inserts those into the pool, and inserts the document into the index. This is encapsulated by the pseudo-code depicted in Figure 2.5. This spider algorithm is equivalent to a graph-covering algorithm where nodes represent documents and edges represent hyperlinks.

Although a spider provides a sound theoretical model by which to locate all available Web pages, there are technical limitations. Spiders typically operate from fast machines linked to high-speed data connections. However, the sites that host Web servers are often not so fortunate. If a spider requests multiple pages from a low-powered server, then that server may be unable to satisfy those requests, or may be

unable to satisfy requests from real users.

While there is no formal standard on how often a spider can request a page or how many pages a spider can request from a server at any one time, there is an informal guideline that limits how many requests a spider can issue to a server in a set time period [59]. Most spider operators use the guideline of no more than one page per minute. With several million URLs to choose from, this would not seem to be much of an issue even when requesting a thousand URLs per minute. However, this limitation can impact how long it takes to index an entire site. For example, at the time of this writing the University of Washington Computer Science and Engineering Web site provided nearly 50,000 distinct HTML pages in one week. At a rate of one page per minute, it would take a spider thirty-four days to index every page.

2.5 The Coherence Problem

Web documents are constantly modified, added to the Web, and removed from the Web. Thus, Web indices are constantly going out of date. The *Coherence Problem* is the problem of keeping the index consistent with the contents of the documents that are indexed. It takes as input a searchable index, and returns as output a searchable index whose content is consistent with the original source data. There are two approaches to solving the Coherence Problem. The first approach involves some system whereby index maintainers are notified that a particular page has changed. They can then reindex the page immediately and ensure a coherent index. The second approach periodically polls the pages, and reindexes them when they change. If the polling interval is frequent enough, then the index will not be incoherent for very long.

Although there have been many proposals suggested to allow notification to interested parties [26, 86, 23], none of them are widely used in practice. The main reason is that there is no widely deployed software to handle the notification. Server operators

are therefore required to locate, install, and maintain one which may not support the appropriate notification protocols. Not surprisingly, most server operators do not support any notification system, and therefore index maintainers do not rely on one. If a single notification standard emerged, then a notification method may become a feasible solution. However, that is not likely in the near future [57].

Polling can also be used to keep large indices up to date. Documents are polled at some rate, and reindexed when they change. Further spidering may also be necessary if new pages are discovered. A key question is when and how often a document should be polled. Some documents change daily, and others change rarely, if at all. One common method is to learn how frequently a page changes using a method similar to binary search [85]. Initially, pages are revisited after a set time. If the page changes, the interval is halved. If the page remains the same, the interval is doubled. This process continues until an appropriate frequency is found, or global maximums or minimums are reached. Typical maximums and minimums are revisiting a page no more than once per day, and at least once in four months.

Polling is analogous to repeated spidering when the URLs to be polled are given to the spider as starting documents. Most polling is in fact done using a spider. As a result the resource requirements as well as the server guidelines of polling are the same as for spidering. Continuing our example above, if it takes thirty-four days to fully index a site with 50,000 pages, then it will take an additional thirty-four days to completely reindex it. Even using the binary search method to determine the polling frequency, starting at thirty-four days it would take over two months to determine that a page on this site changed daily.

2.6 Real-world constraints and Web search engines

One could argue that a spider will eventually visit every page on the Web within a reasonable time. The Excite spider visits 10 million pages per week [81], and high

estimates for Web growth are 25 million pages per month [12]. Since the Excite spider is visiting pages faster than new pages are being created, Excite should be able to index the entire Web given enough time.

One might further argue that polling could be used to keep an index up to date. Even though it may take two to three months to determine the proper polling frequency, it would eventually be determined. And one could claim that most pages do not change frequently, therefore the guidelines on how often documents can be polled from a single server will not be violated if the proper polling frequencies are used.

In an ideal world, a spider-based search engine that updates its index using polling could maintain a comprehensive index of the entire Web. However, Web search services are also bound by several real-world constraints: how much storage is available, how many CPU cycles are available, how quickly results must be returned to the user, and how many users will access the system. While storage is relatively inexpensive, fast and reliable storage comes at a premium. Even if the cost of storage is not an issue, the larger an index, the more CPU cycles may be spent in performing retrievals. This directly impacts the speed by which a search engine can return results to the user. Furthermore, search servers can only do so much work. As CPU utilization for a single query increases, the number of queries that can be processed by a single server decreases. Thus, in order to satisfy the same user base, additional servers may need to be installed.

Figure 2.6 shows the size of the major Web search service indices from December 1995 through March, 1999. Clearly, if the size of the Web is 150 million documents then only AltaVista provides a comprehensive Web search. However, as we show in Section 5.1.4, the size of the Web is on the order of 600 million documents. A more interesting trend is that the index sizes for many of the search services have remained surprisingly unchanged, even though the Web has grown consistently during the time shown. In particular, HotBot, Excite, InfoSeek, and WebCrawler have used the same size index for the past year, and Lycos' index remained at the same size for nearly

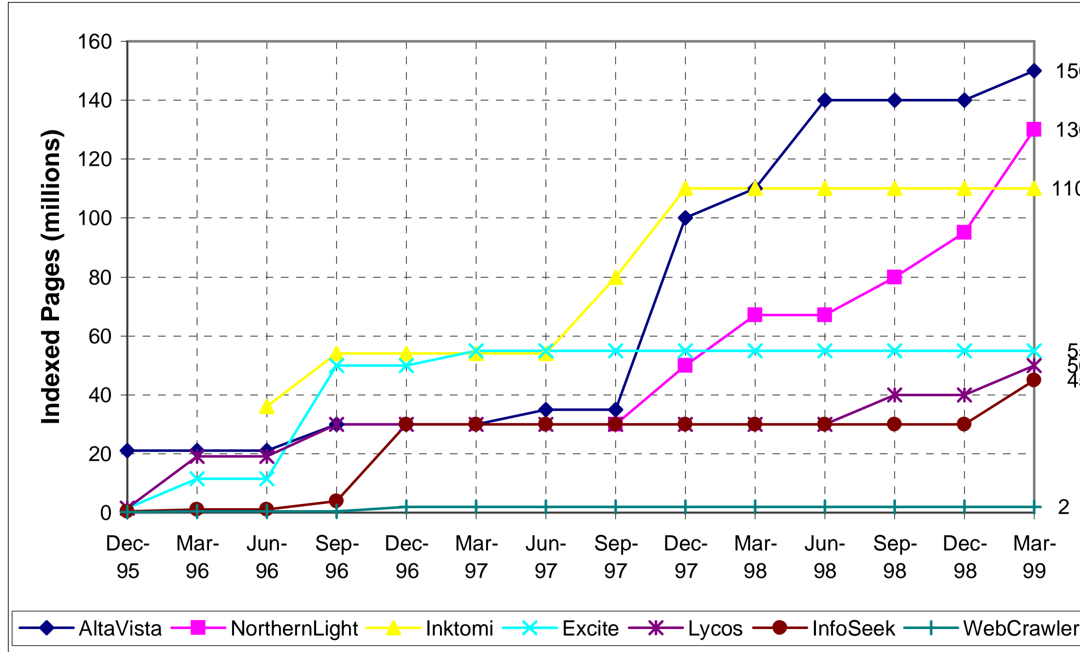


Figure 2.6: Sizes of search services' indices (in millions of pages).

This figure shows the size of the indices of the most popular spider-based search services in millions of pages from December, 1995 through March, 1999. This data comes from `www.searchenginewatch.com`*, an online site that monitors Web search services [105]. This data was collected from press releases and other public information about the engines, and thus should be considered to be an upper bound.*

two years before increasing in June of 1998. These trends support the hypothesis that indexes grow not when new pages are discovered, but rather when substantial resources are added to the service. While slightly dated, a 1997 article by Brake gives a good discussion as to why this is the case [16], indicating that search service companies are focusing on improving finding quality matches within their index rather than expanding their index.

2.7 Summary

In this chapter, we presented the Web Search Problem: How can a user find all relevant information about a topic on the Web? We highlighted various automatic browsing techniques which can provide relevant information, but which do not guarantee to provide that information in a timely manner. We then described how Information Retrieval engines could address the Web Search Problem. However, using Information Retrieval engines introduces the Web Discovery Problem: How can all Web pages be located? We highlighted the major techniques for addressing the Web Discovery Problem: collective user histories, distributed search, and spiders. We have shown how an Information Retrieval engine can use a spider-based index to provide a comprehensive search. We then presented the Coherence Problem: How can the contents of a Web index be kept up to date with the Web? We have also shown how polling can address this problem to a reasonable degree.

Finally, we have shown that spider-based search services have real-world constraints that limit their ability to provide a comprehensive search. We now turn to our solution to providing a more comprehensive search than using any major Web search service: MetaCrawler.

Chapter 3

METACRAWLER ARCHITECTURE

In the previous chapter we highlighted the key techniques to solving the Web Search Problem. The most promising of those techniques, spider-based search services, are prevented from providing a comprehensive index of the Web by real-world constraints. An obvious question is whether a comprehensive search can be accomplished by querying multiple spider-based search services and combining the results.

To answer this question, we designed and implemented a *meta-search* engine. A meta-search engine is one that does not use its own index. Instead, it selects other search services to query, forwards the query to those sources, collates and post-processes the results, and then returns those results to the user. The motivating belief of a meta-engine is that it is as good as the sum of its parts. Our claim is that our meta-engine, MetaCrawler, is greater than the sum of its parts, which in this case are some of the best Web search services available.

The remainder of this chapter is organized as follows:

- We claim that a solution to the Web Search Problem is the MetaCrawler meta-engine. Therefore, it is necessary to define what the MetaCrawler meta-engine is and does. We begin this chapter with a high-level overview of the MetaCrawler meta-engine. This will set the foundation for the rest of the thesis.

- Implementing a meta-search engine is trivial, but a trivial design is likely to be very limited. We designed MetaCrawler with much broader goals, which we present in Section 3.2.

- Satisfying our design goals is not straightforward. In Section 3.3 we present an overview of the architecture we designed to address our design goals. We also present the three main components to our architecture: the Client Layer, the Server Layer, and the I/O Layer.

- An assumption in designing a meta-engine is that the only input will be a user query. An extension to the MetaCrawler architecture allows search service information to be submitted as well to dynamically extend the number of services MetaCrawler can query. This extension is described in Section 3.4.

- Finally, we evaluate how well our architecture satisfied our design goals in Section 3.5.

3.1 MetaCrawler overview

MetaCrawler is a search service that uses a meta-engine to conduct the search. MetaCrawler and other search services that use meta-engines are often called *meta-search services*. MetaCrawler has certain requirements in order to be a competent meta-search service. First, it must take user queries and format them appropriately for each search service. Next, it needs to correctly parse and aggregate the results from the other sources. Finally, it has to analyze the results to eliminate duplicates and perform other checks to ensure quality. Figure 3.1 details MetaCrawler's control flow.

In addition, there are several additional features needed to make MetaCrawler usable by average Web users. These features strongly impact how MetaCrawler is designed. First and foremost, MetaCrawler needs a user interface that the average person can understand. Second, MetaCrawler needs to perform its tasks for the user as quickly as possible. Finally, MetaCrawler needs to be able to adapt to a rapidly changing environment.

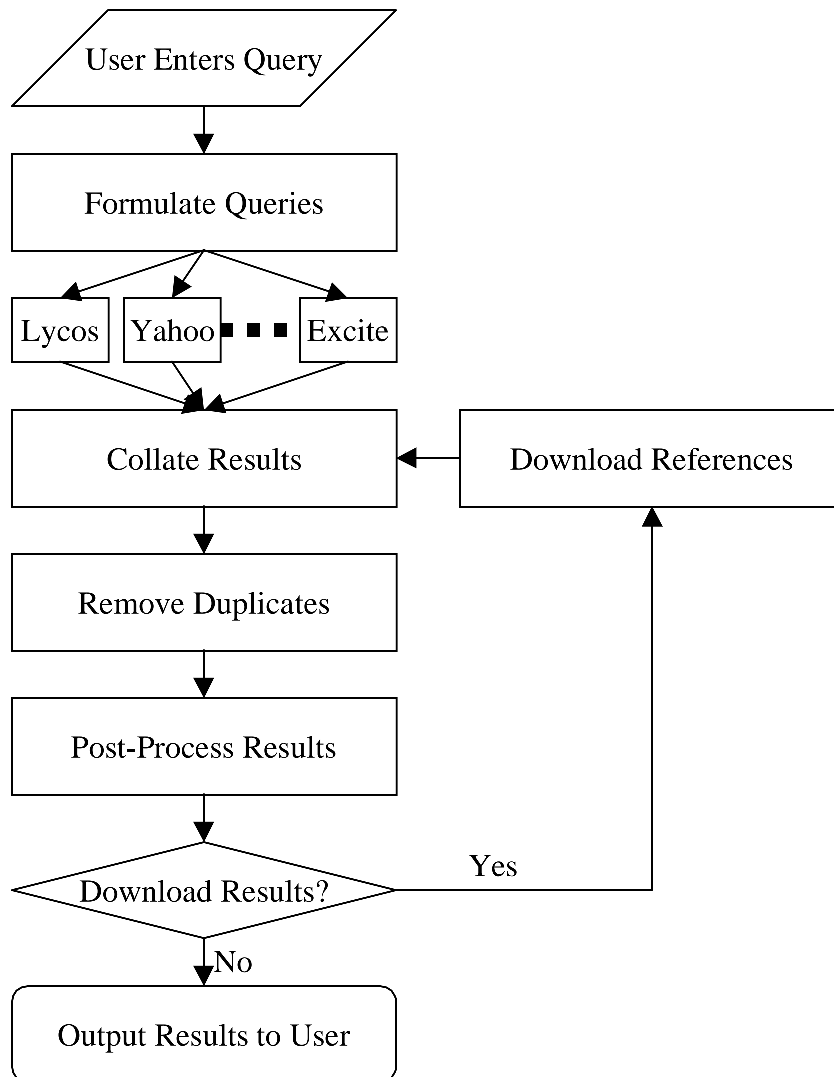


Figure 3.1: MetaCrawler control flow.

The control flow for MetaCrawler. The user enters a query, which is then translated and forwarded to the sundry Web information sources, such as Lycos and Yahoo!. The results from these services are then collated, duplicate entries are removed, and post-processing is done. If the results are downloaded, they will again need to be collated, have duplicates detected, etc. Finally, the results are sent to the user.

3.1.1 *Understanding query and output formats*

Once the user enters the search terms and various parameters, MetaCrawler translates the terms into queries that each of the underlying Web search services can process. It then submits those queries and parses the results that are returned into a canonical form for further processing. The code that does this is encapsulated in a *wrapper.* A wrapper takes as input a query with appropriate parameters, and returns as output a list of *tuples.* A tuple is a data structure that contains the information for each individual result. The tuples used by MetaCrawler describe *document references.* A document reference is the URL, title, snippet, and confidence score for the given document.

Wrappers handle the problem of dealing with different search services that have different user interfaces and query languages. Wrappers provide a common programming interface to each search service. They handle the translation of the query into the search services' particular formats, and handle the parsing of the results into a canonical format. Wrappers typically do not submit the translated query directly, but instead rely on hooks to the appropriate I/O interface.

Typically, each search service requires its own wrapper. Each wrapper requires its own unique code to handle query translation. This translation is typically not difficult, but there are some details involved. For example, Alta Vista prefers searching for phrases by enclosing the phrase in double quotes, e.g. "Utah Jazz," whereas HotBot uses a menu option that selects phrase searching. Unfortunately, it is these details that typically require some degree of manual coding for each search resource. There has been some work at alleviating the need for manual coding [78], but there are still no general solutions that handle feature sets as rich as the search services. Fortunately, Web search services do not change their query format frequently, and therefore manual coding is an acceptable option.

Translating a user query into the proper format for each search service only ad-

dresses half of the problem. The wrapper must also parse the results of the query into a canonical format in order to collate the results. Again, it is not difficult to write a parser for the results, but one must write a unique parser for each search resource. Fortunately, even though Web search services do change their output format somewhat frequently, there have been some advances in automatically generating and maintaining result parsers [61].

3.1.2 Collation and duplicate detection

Once the wrappers have returned the results in a canonical format, it is necessary to identify any duplicates. Detection of duplicate references is important for three reasons. First, users do not want to spend time looking at duplicate results. Second, it is often desirable to increase a reference's ranking if multiple search resources return that reference. Third, removing duplicates will save any cost associated with processing the same document twice. Detecting duplicate references is difficult without the full contents of a particular page, due to host name aliases, symbolic links, redirections, and other obfuscating factors. Even with the full contents of a page, duplicate detection is not always trivial as there are often insignificant differences in two pages' text. For example, copies of the manual often contain a pointer to the local maintainer at the bottom of the text. MetaCrawler uses several heuristics to detect duplicates; these are described in detail in Section 4.3.

Once duplicates have been detected, MetaCrawler needs to collate the results from each search service into a single list. This task is difficult because each service uses a different ranking criterion to order their results, and some services don't report any information about the ranking besides the order. MetaCrawler uses a novel collation algorithm called *Normalize-Distribute-Sum*. This algorithm interleaves results based on a number of features, including the document ordering, number of services returning the document, and service-specific confidence score. This algorithm will be described in detail in Section 4.2.

3.1.3 Highest common denominator of services

Supporting a rich query feature set can be problematic. Some services support all the query features that MetaCrawler provides, whereas others are lacking in particular areas such as phrase searching or weighing words differently. Rather than providing "lowest common denominator" service and support features only found in all engines, MetaCrawler implements features not available from some or any service. One way it accomplishes this is through downloading the pages referred by each service and analyzing them directly. The time required to download all pages in parallel adds no more than a couple of minutes to the search time, which is a reasonable quality-for-speed tradeoff for many users. In addition, MetaCrawler is able to process results that have arrived while waiting for others. Therefore, minimal computation time is required after all the references have been retrieved. Most users prefer to have the results displayed quickly. Thus, even though downloading and post-processing documents does improve quality substantially, users must explicitly activate this feature.

Downloading and analyzing references locally has proven a powerful tool for implementing features not found in some services. For example, when Lycos was first deployed, it did not handle phrase searching. MetaCrawler simulated phrase searching by sending Lycos a query for documents containing all the query words, downloading the resulting pages, and extracting those pages that actually contain the appropriate phrase rather than just the words somewhere on the page. Using similar methods, we are able to handle other features commonly found in some, but not all, search services, such as requiring words to be either present or absent in pages.

In addition to simulating features on a particular search service, we are able to implement new features. First, by downloading a reference in real time, we verify that it exists. This has proven to be a popular feature in itself. We are also able to apply sophisticated ranking and clustering algorithms to the documents, as well as extract words from the documents that may be useful for refining the query [112, 15].

3.1.4 Highest quality results and fastest information location time

A claim that we could make of MetaCrawler is that it returns the highest quality results available for a given query. MetaCrawler requests the top k documents from each service and collates them. The top k documents that MetaCrawler returns are thus the highest quality documents as determined by the various search services. Thus, the top k documents returned by MetaCrawler should be of higher quality than the top k documents returned by any single search service.

Another claim we could make is that MetaCrawler provides fastest overall method of locating relevant information. Without meta-search, in order to find relevant information a user needs to visit a search service, enter the query, and analyze the results that are returned. If the proper information is not present, the user continues to the next search service and repeats the process. In contrast, using MetaCrawler a user enters the query once, waits slightly longer than the slowest service, and then analyzes all the results at once. If relevant information is likely to be present on all search services, then on average users will find relevant information much faster than by using traditional Web search services in a serial manner.

Our focus on MetaCrawler is providing a comprehensive search, not necessarily the best ranking of the results nor providing the fastest time to locate information. As we show in Section 5.1, evaluating these claims would also require a substantial user study, which we were unable to accomplish. Therefore, even though our practical experience indicates these claims are true, we did not evaluate these claims scientifically.

3.2 Design goals

MetaCrawler was designed, not simply as a meta-search engine, but as a general meta-search tool that can be applied to a variety of searching problems. We designed MetaCrawler with the following goals:

Expandable: We designed MetaCrawler as a research testbed. Therefore, it was critical that the author as well as other researchers could add to MetaCrawler or build on top of MetaCrawler to further their own research.

Low maintenance cost: While we designed MetaCrawler to be a public service, MetaCrawler was operated by graduate students with limited time. Therefore, it was important that they could maintain MetaCrawler with as little effort as possible.

Fast response time: Fast response time is critical for two reasons. First, we designed Metacrawler to be a publicly available Web service. Therefore, to retain users it had to have good performance relative to the other search services. Second, there were limited computing resources available to operate MetaCrawler. Therefore, the faster it was able to satisfy a user query, the more users MetaCrawler would be able to help.

Scalability: When we first designed MetaCrawler, new search services were becoming available in rapid succession. Therefore, we designed MetaCrawler to incorporate a plethora of search services.

Portability: We designed MetaCrawler to exist not only as a Web-based search service handling millions of users, but also as client application that would service just a single user. As a stand-alone client application, the resources available to it are likely to change, and thus our design needs to accommodate likely changes.

3.3 Architectural layout

To facilitate the design goals, we designed MetaCrawler in a modular fashion as depicted in Figure 3.2. There are three main components: the Client Layer, the Server

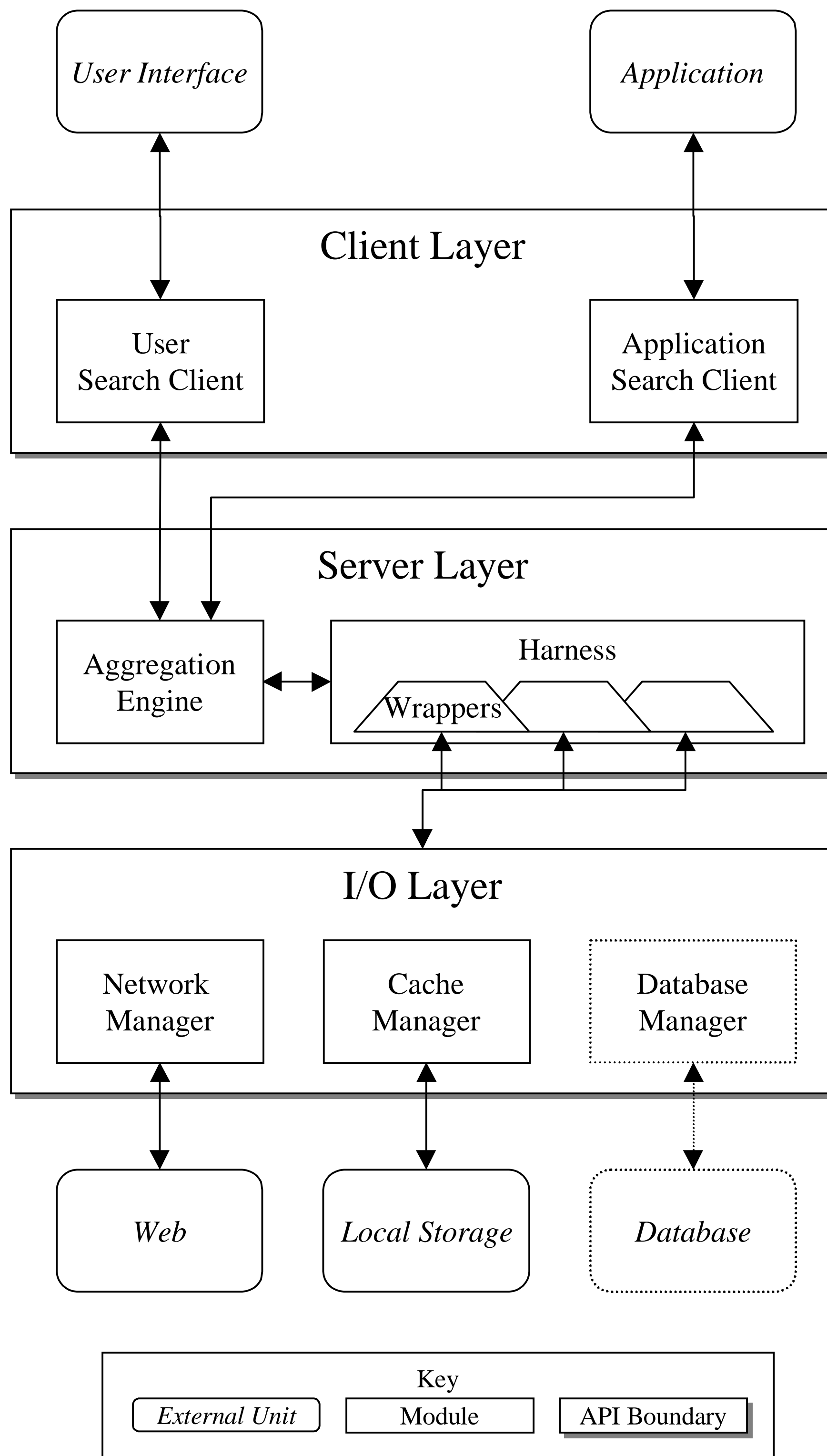


Figure 3.2: MetaCrawler architecture.

Layer, and the I/O Layer. The Client Layer contains the MetaCrawler interface code to external entities. Users interact with MetaCrawler through an external user interface module, such as a CGI translation program, and applications interact with MetaCrawler through custom interface modules. The Server Layer is the bulk of the MetaCrawler meta-search engine, and is responsible for selecting search services to use, submitting queries, parsing results, collating documents, and detecting duplicates. The I/O Layer is the low-level code that performs the queries and retrieves Web documents. These units are described in the following sections.

3.3.1 Client Layer

MetaCrawler was designed to be accessed by both users and controlling applications. In order to accommodate different user interfaces as well as different application demands, MetaCrawler uses a number of *Search Client* modules, which are contained within the Client Layer. Each Search Client module contains interface code that bridges the Server Layer to the controlling entity. The appropriate Search Client module is loaded when the MetaCrawler receives a query, and persists until the query is satisfied. The Search Client receives as input the *user query*, which is comprised of search terms, query logic, and other appropriate parameters. The output of the Search Client is the *final results* to be displayed by the controlling entity. Optionally, the Search Client may also output status information as the query is executing.

There are two main Search Clients available in the MetaCrawler code: a CGI (Common Gateway Interface) Client, which bridges the Server Layer with a traditional Web-based form interface, and a Java Client, which bridges the Server Layer with a controlling Java applet. This applet is used to provide real-time feedback and browsing as described in Section 4.1.2. However, it is not difficult to add additional Search Clients should a new external entity require one. For example, a new external application may require a new Application Client Layer, as shown in Figure 3.2.

3.3.2 Server Layer

The Server Layer contains the two core modules for MetaCrawler: the *Aggregation Engine* and the *Harness*. The Aggregation Engine is the control unit of this architecture. It accepts as input a *user query*. The Search Client module may have modified the original user query slightly, but typically there is little deviation. As output, it returns a collated list of document references to be formatted appropriately by the Search Client. The Aggregation Engine is responsible for the final selection of search services to use. It selects the appropriate search services and passes this information as well as the user query to the Harness.

The Harness is the unit which contains the information about available search services. The Harness is simply a collection of wrappers, and thus forwards the user query to the appropriate wrappers and returns a concatenation of their results. The Harness accepts as input the user query, given to the Aggregation Engine, as well as which search services to query and returns as output the raw document references obtained from the wrappers used.

The Aggregation Engine receives the document references from the Harness. It is then responsible for the collation of those references, which it returns to the Search Client. The Aggregation Engine contains the main event loop of the meta-engine. Therefore, it is responsible for evaluating the halting criteria. Typically, this is either when all wrappers have finished or timed out, or when the connection to the controlling entity has been severed.

Types of wrappers

Wrappers can be partitioned into three categories:

Hard Coded: A hard coded wrapper is simply a wrapper with all the pursuant information encoded into the wrapper at compile time. These are the easiest to create, but are often brittle as changes in the search service may cause the

wrapper to fail.

Parameterized: A parameterized wrapper is a wrapper that requires extra parameters either at run or query time. For example, a source may require a user name and password for access, which need to be passed in as parameters to the wrapper.

Dynamic: A dynamic wrapper is a wrapper whose specification is not determined until run or query time. For example, a service which uses Apple's Sherlock specifications downloads the specification at query time [2].

MetaCrawler uses hard coded wrappers for most sources, and some Parameterized wrappers for extensions that will be covered in Section 3.4. While hard coded wrappers are in fact very susceptible to breaking due to minor changes in the search service, most Web sources do not change very often. Further, there has been recent work investigating improved ways to automatically adjust wrappers to some changes [60].

Query translation and result parsing

Wrappers are responsible for translating the user query into the appropriate format for the search services. For some search services, this is simply creating an appropriate information request to obtain a single document containing the results. However, for many search services, multiple information requests must be issued to obtain the desired information. This may involve some extra processing of the previously returned document, as it may have information required to create the next request.

The other primary task for a wrapper is to parse the document returned by a search service into tuples, which in our case are the elements of document references. This is typically solved by simple regular expression parsing. However, some documents may

require more sophisticated techniques. Typically, when a wrapper breaks because of a change in the search service, it is the parsing component of the wrapper that fails.

3.3.3 I/O Layer

The key to high-performance in a meta-search engine is a good I/O Layer. For input, the I/O Layer takes information requests and returns either the requested information from a search service or an error code. The I/O Layer contains the modules that support interaction between different types of storage. MetaCrawler uses a Network Manager to interact with the World Wide Web and a Cache Manager to interact with files cached on local storage. Other managers could be easily added, such as a Database Manager to interact with a database.

The key to MetaCrawler's high-performance I/O Layer is that I/O requests are handled in parallel, meaning that the latency of any request is not affected significantly by other requests in the system. Parallelism is also what makes implementing the I/O Layer non-trivial; how it is implemented is discussed in Section 4.4.

3.4 Integrating user-defined search resources

Most meta-engines are only able to query a fixed set of search services, decided in advance by the authors of the meta-engine. An improvement to this paradigm is to allow users to define and use new search services, such as a local Intranet search service or an index of their browser history. It is an open question on how to extend the set of search services available to the user. We highlight two methods that allow users to add search services to a meta-engine, wrapper templates and wrapper libraries, and then detail the technique used by MetaCrawler.

3.4.1 Wrapper libraries and templates

One method of allowing users to add search services to a meta-engine is to use a *wrapper library*, which is a collection of publicly available wrappers. For example, both Apple Inc. and Apple Donuts maintain libraries of wrappers for the Sherlock meta-engine [2, 3]. These wrappers contain the information necessary to make a query as well as extract the URL, anchor text, snippet, and confidence score from the results. However, there are two substantial drawbacks with using wrapper libraries. The first is that because these wrappers are written for general use, they assume that search services are in fixed locations, e.g. Excite is available at `www.excite.com`. Thus they cannot be used for search services with dynamic locations. The second is that these wrappers have no inherent method of authorizing the user for a particular source. Users would be required to enter authorization information at query time. Cookies, which are small pieces of information stored on a browser, cannot be used to shortcut this authorization step. Users who share browsers will also share cookies, and thus if cookies are used one user could use the authorization of another. While potentially tedious, entering authorization information for each session may work for a client-side meta-engine. However, giving authorization information to a publicly available meta-search service is suspect, as the meta-search service could record the authorization information for illicit use. Thus, only publicly available sources can be used.

Another option is to use a *wrapper template*, which is simply a general purpose wrapper that requires a user to identify certain fields, such as the query field. For example, users use this technique to add information sources to the commercial client-side meta-engine WebCompass [83]. To fill out the template, WebCompass instructs the user to issue a query using particular terms to the search service. WebCompass extrapolates how to issue a general query by simply replacing the query terms with user terms. WebCompass then uses a primitive parser to extract URLs. It makes the

assumption that URLs on a results page that point to remote sites are the results, and that URLs that point to internal pages are help pages, navigational aids, or other local information not relevant to the search. The parser simply extracts all URLs that are non-local. Unlike the wrappers available from the Sherlock wrapper libraries, WebCompass' wrappers are unable to extract the confidence score and snippet.

Wrapper templates are able to incorporate search services with dynamic locations. However, the location needs to be entered each time it changes, which may be a burden for the user. In addition, a properly expressive wrapper template can allow for user authentication. This belies one of the limitations of wrapper templates. While wrapper templates allow the integration of numerous user-defined sources, they are only as expressive as the actual template. For example, the WebCompass template is only able to generalize search services that accept a single set of query terms; search services that accept two or more sets, such as one that accepts a book title in one set and an author in another cannot be integrated into this scheme. The WebCompass template only allows the query terms to be modified. Other options, such as setting the query logic or maximum number of results, are immutable. Finally, the results must be non-local to the search service. A search service that returns local documents, such as the search page for an Internet directory like Yahoo!, will not be correctly parsed.

3.4.2 Dynamic Service Protocol

Users can add a number of search services using wrapper libraries and templates. However, they are still unable to add a restricted access source with a dynamic location without continually entering the access and location information. To address this problem, we created the *Dynamic Service Protocol.* The Dynamic Service Protocol is a protocol whereby a controlling application informs MetaCrawler of the location of a search resource as well as how it should access that resource. To demonstrate this protocol, we incorporated the Clio system into MetaCrawler.

The Clio system

Clio allows a user to search his or her Web history [63]. To use Clio, a user's Web browser accesses the Web through a Clio proxy. In addition to providing Web access, the proxy transparently inserts documents into a private, searchable index. Clio is part of CollabClio, a project that enables users to search through other users' histories. The focus of the Clio research was on the privacy aspects of the system. One aspect of the system is that users had to provide proper authentication in order to search any index, including their own.

MetaCrawler interface to Clio

Rather than having users use a Clio interface to search their own history and then a separate MetaCrawler interface to search the Web, we integrated Clio into MetaCrawler. In order for MetaCrawler to query Clio, it needed the location of the user's Clio index and the authentication information to access it. The Clio proxy was enhanced to provide this information when it accessed MetaCrawler. MetaCrawler then instantiated a parameterized Clio wrapper with this information. The user was given the option of searching just his or her history. MetaCrawler then queried the user's Clio index along with the other sources if appropriate. Clio's results were then collated with the rest for a uniform appearance. Figure 3.3 provides a graphical overview of this integration.

One deficiency with the prototype implementation is that it used the user's authentication information. While the testers of Clio were local to the University of Washington and trusted the authors of MetaCrawler not to record this authentication information, in a deployed system more security is needed. A simple but somewhat insecure idea would be for Clio to create a temporary one-use password that would be valid for one query from the MetaCrawler. Better ideas would be to use some kind of secure transmission via the standard Secure Sockets Layer (SSL) [76] or

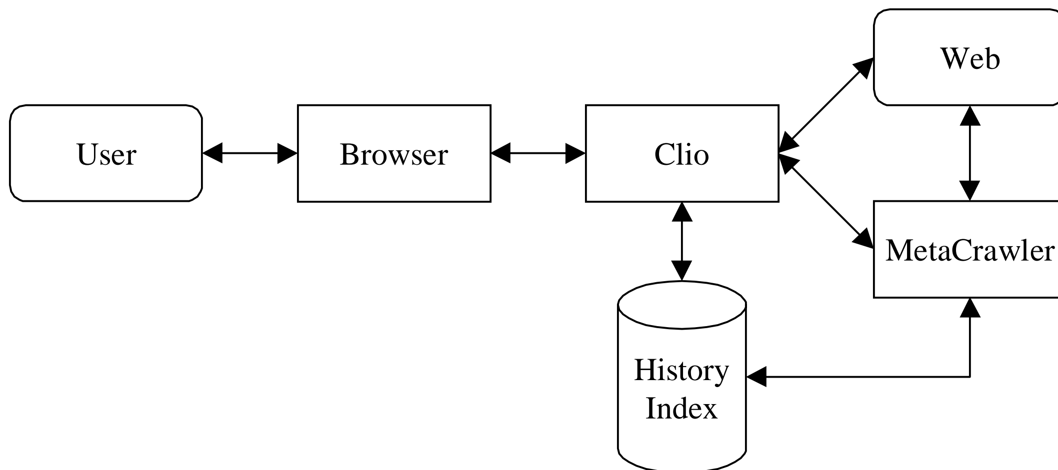


Figure 3.3: Clio and MetaCrawler integration.

A user connects to the Web via a Browser. The Browser connects to the Web via Clio, which copies all pages browsed into a History Index. When a user connects to MetaCrawler, Clio sends additional information that allows MetaCrawler to query the History Index. This information is then merged with results from other search services, and transmitted back to the user.

authentication using a public key or capability-based system [87, 11].

The integration of Clio into MetaCrawler via the Dynamic Service Protocol requires no explicit user interaction. Rather the Clio system handles that interaction directly. In this manner, the Clio system is able to impart to MetaCrawler all the information necessary to search a particular search service. This technique extends easily to other kinds of search services, such as pay-per-query sources or corporate intranet search resources available via a dynamic connection through a firewall.

3.5 Architecture evaluation

MetaCrawler's architecture was largely successful in attaining its goals. We now show how well MetaCrawler met each of its design goals in the following sections.

3.5.1 Expandable

MetaCrawler's architecture was successful at enabling a wide variety of enhancements and additions to the system. By separating the Client Layer from the Server Layer, we were able to experiment with a number of user interface designs beyond just the simple HTML forms shown in Section 4.1. For example, Christin Boyd designed and developed a query refinement system as part of her senior thesis [15]. This system attempted to aid the user in improving a given query as well as provide us with failure data on poor queries. Greg Lauckhart added a "View by Site" option to the Search Client that sorted the results not based on traditional relevance ranking but in alphabetical ordering by URL [65]. Separating the I/O Layer from the Server Layer gave us similar flexibility. For his senior thesis, Darren Schack developed a caching mechanism that would store pages downloaded from the Web to benefit repeated or similar queries [90].

A more sophisticated extension is Grouper [112] by Oren Zamir and Oren Etzioni. Grouper adds a *clustered* output to MetaCrawler, which displays URLs grouped into

clusters based on common terms and phrases. Rather than creating clusters after all references are received, Grouper creates the clusters as references are received. MetaCrawler is thus able to return the results immediately once all the references have been received. To do this, Zamir and Etzioni created a superclass of the Aggregation Module which handles the clustering.

Two large-scale applications using MetaCrawler will be described later. Ahoy! The Homepage Finder by Shakes, Langheinrich, and Etzioni is a complete application that interfaces directly with MetaCrawler [96]. The second is HuskySearch [95], a meta-engine based on MetaCrawler after MetaCrawler was licensed to Go2Net Inc. in 1996. These will both be discussed in detail in Section 4.5.

3.5.2 Low maintenance cost

Change in the output of Web search services is the primary cause of maintenance in MetaCrawler. Addressing these changes requires updating the appropriate wrapper to handle the new output format. These are typically trivial changes and do not amount to a great deal of time.

The secondary cause for maintenance is when a new project is added to MetaCrawler, such as the senior theses. By dividing MetaCrawler into the various functional units, most projects are able to fit inside a particular unit. Thus a minimum of coding is necessary to incorporate a new enhancement to the rest of the system.

One way to implement MetaCrawler would have been to use a single MetaCrawler process to handle multiple queries, where presumably each query session would be contained in its own thread. Instead, we used a single MetaCrawler process for each query. This decision was initially made for convenience, but was continued to keep the overall MetaCrawler system robust. Since MetaCrawler was a research testbed, there were often errors in the deployed code. If any one query session encountered an error, only that query would be terminated in case of a crash, while the other queries would be able to continue. The one process per query model also made debugging

more straightforward than a multi-threaded model.

3.5.3 Fast response time

The key module that affects the wall-clock performance of MetaCrawler is the Network Manager in the I/O Layer. By separating the I/O Layer from the Server Layer, we were able to experiment with a number of different techniques and enhancements without rewriting other modules. Even after we finally settled on the event-driven model, described in Section 4.4, we were able to continue to make improvements, such as with the Cache Manager, again without affecting other modules.

The other aspect of the architecture that added to improving overall performance is the linear data traffic pattern. Most modules exist in a chain, communicating with adjacent modules. This makes bottlenecks straightforward to identify via profiling.

3.5.4 Scalability

Scaling the number of search services MetaCrawler accessed is a challenging problem. We decided to use two modules to address this problem. The Harness module presents each search service through a uniform interface. The Aggregation Engine then uses the appropriate search services to collect information and conducts the appropriate collation, duplicate detection, and post-processing. The Harness is separate from the Aggregation Engine so that creation and maintenance of wrappers can be completed without the need to modify the Aggregation Engine. There is no limit on the number of wrappers the Harness could contain; the difficulty is in writing and maintaining them.

One area that we did not explore is *Query Routing*. Query Routing is the process of selecting appropriate search resources from a large set of resources. Proper Query Routing is the key requirement for a meta-engine to scale to a large number of search services. There are a number of Query Routing schemes available [98, 46]. However,

since the actual implementation of MetaCrawler used under ten search resources, MetaCrawler defaults to broadcasting to all available general Web search services.

3.5.5 Portability

We have had great success in compiling MetaCrawler on different architectures, including DEC OSF, Linux, and Windows NT. The only port that required significant effort was the Windows NT port, which required a different I/O Layer implementation. However, because of the architectural modularity, the rewriting of the I/O Layer did not require other units to be modified.

MetaCrawler is an intelligent interface to powerful remote services. It does not require large databases or large amounts of memory. Therefore, MetaCrawler is not limited to residing on high-end servers. While MetaCrawler was initially implemented as a universally accessible service at the University of Washington, we have created prototype implementations that reside on the user's machine. Although this prototype provided no new functionality over the existing service, an individualized MetaCrawler *client* that accesses multiple Web search services has a number of potential advantages. First, the user's machine bears the load of the post-processing and analysis of the returned references. Given extra time, post-processing can be quite sophisticated. For example, MetaCrawler could use slower but potentially better clustering methods than Grouper, or it could engage in *secondary search* by following references to related pages to determine potential interest. Second, the processing can be customized to the user's tastes and needs. For example, the user may choose to filter advertisements or parents may try to block X-rated pages. Third, MetaCrawler could support scheduled queries, e.g., What's new today about the Seattle Mariners? By storing the results of previous queries on the user's machine, MetaCrawler can focus its output on new or updated pages. Finally, for pay-per-query services, MetaCrawler can be programmed with selective query policies, e.g., "go to the cheapest service first" or even "compute the optimal service querying sequence."

3.6 Summary

In this chapter we have presented MetaCrawler, a meta-engine for information retrieval on the Web. MetaCrawler addresses some of the issues inherent with spider-based Web search services. We have detailed MetaCrawler's architecture, which was designed to promote expandability, low maintenance, performance, scalability, and portability, and we have described how the architecture was able to accommodate those goals. We now turn to some of the technical details of MetaCrawler.

Chapter 4

METACRAWLER IMPLEMENTATION

In the previous chapter, we have presented our meta-engine architecture. While it may seem straightforward to implement a meta-search engine based on that architecture, there are several issues that arise in order to make a practical and useful meta-search service for the World Wide Web. We address the main technical challenges to MetaCrawler:

- Comprehensive Web search provides little value if users cannot utilize its functionality. How can average Web users take advantage of the features of MetaCrawler?

- Each search service ranks documents in a different manner. With the limited information about documents that is returned, can results from heterogeneous search engines be combined in a way that is not biased towards any service?

- As mentioned above, the results that are returned from search services contain limited information about a document. Can duplicate documents be detected with just this limited amount of information?

- Querying search services and downloading the Web pages they return in real time takes a significant amount of system resources. This is compounded by multiple queries on a single machine. What is the best parallel technique to handle this load?

- Users provide a large amount of both positive and negative feedback through e-mail and other forms of direct communication. What are the enhancements to MetaCrawler that most benefit users?

4.1 MetaCrawler user interface

The motivating design principle behind MetaCrawler is that the user should say *what* he or she wants, and MetaCrawler should determine automatically *where* to search and *how* to search. This was inspired by the Internet Softbot project [36]. In addition, the user should be allowed to specify what he or she wants without learning a complex syntax or feature set. With this in mind, it is important that MetaCrawler have a good user interface that allows the user to communicate what he or she is searching for.

While giving the user a Web form with added expressive power was straightforward, presenting the user with a form that would facilitate using novel features of our meta-search engines proved to be a challenge. We strove for a balance between a simple search form and an expressive one, keeping in mind interface issues described by service providers [79]. Our MetaCrawler version 1.0 interface design was based on the common schemes of contemporary Web search forms, which were at the time very Spartan: a text entry box for the search terms, a search button, and occasionally some extra controls for various parameters.

4.1.1 Search form

The default query syntax used by most contemporary search services was simple keywords separated by spaces. Without any query modifiers, documents matched the query based on how many of the query terms the document contained. For example, the query `John Cleese` would return any document that contained the term "John" and "Cleese." However, there was no implicit ordering to the terms, thus a document

containing "John Grisham" and "Hector Cleese" could also be returned. There is also no implicit requirement that all terms be present in documents, thus documents containing just "John" or "Cleese" could also be returned.

In our version 1.0 design, we focused on improving the query syntax. We added query modifiers similar to InfoSeek's search syntax: quotes or parentheses were used to define phrases, a plus sign designated a required word, and a minus designated a non-desired word [49]. For example, to search for "John Cleese," requiring that both "John" and "Cleese" appear together, the syntax required is `(+John +Cleese)`. In addition, we used a three button design for describing the query logic of the search:

Search for words as a phrase: Treat the search terms as a single phrase, and attempt to match the phrase in pages retrieved

Search for all of these words: Attempt to find each word in the search terms somewhere in the retrieved pages. This is the equivalent of logical "and" for all terms.

Search for any of these words: Attempt to find any word in the search terms in the retrieved pages. This is the equivalent of logical "or" for all terms.

We adopted this design from Open Text's Web search service, which ceased operation in 1996 [77]. The query logic and query syntax options have some overlapping functionality. The query logic options are targeted to the naive user and are not very expressive. The query syntax is targeted to the more experienced user who would write more sophisticated queries. We considered a fully Boolean interface to MetaCrawler. However, feedback we received from users suggested that they often confused Boolean expressions. For example, they used AND when they meant OR, or thought that OR meant EXCLUSIVE OR. Therefore, we did not make a fully Boolean interface to MetaCrawler.

In addition to the search term entry box, we provided various advanced options which could be activated via two menus. These allowed the user to indicate where the desired information should be located. The first menu described a coarse grain locality, with options for the user's continent, country, and Internet domain, as well as options to select a specific continent. The second menu described the various Internet domain types, e.g. `.edu`, `.com`, etc. Users could use these options to focus their search results to particular logical or geographical domains. Figure 4.1 shows a screenshot of the MetaCrawler version 1.0 search form.

Even with explicit examples presented on the search form, users often introduced syntactical errors causing the resulting search to produce an entirely irrelevant set of hits. For example, users would often omit spaces between words when using either '-' or '+,' turning the search text "+Monty +Python -snake" into the single term "+Monty+Python-snake." Since terms with '+' and '-' were acceptable, such as "C++" or "Berners-Lee," queries such as this were treated as one large word and unsurprisingly returned no results. In addition, the combination of the radio buttons describing the query logic in conjunction with search term syntax added some extra confusion.

Figure 4.2 shows the MetaCrawler version 1.5 design. We reduced the need for extra syntax, and instead ask the user to select the type of search. The older syntax is still supported, although it is not advertised prominently on the main search page. After we changed the search page to the version 1.5 design, the number of malformed requests dropped significantly, although user education may have also contributed to that phenomenon.

4.1.2 Feedback with server-push and Java

Even though we strove to make MetaCrawler as fast as possible, it was still slower than other Web search engines. As a result, users would become impatient and cancel the query, presumably to visit another search service. Therefore, we provided

Figure 4.1: MetaCrawler v1.0 Homepage screenshot, Jan. 1996.

This shows the user interface of MetaCrawler version 1.0. Visually, there is a single text entry box, radio buttons whose options describe the logic of the query (e.g. "All these words," "As a phrase"), various options describing what results are desirable (e.g. results from the user's country, .edu sites, etc), and various performance metrics (e.g. how long to wait).

Figure 4.2: MetaCrawler v1.5 Homepage screenshot, Aug. 1996.

The help text has mostly been removed in favor of a cleaner interface. There are also various pointers to the Java Beta, at the time an experimental interface that allowed users to browse results while MetaCrawler continued to search. In addition, a new "Fast Search" button was added to give users a better way to dictate the length of the search rather than adjusting the "Max Wait" performance parameter.

the user with feedback as the query progressed so the user knew MetaCrawler was still working on the query, how much of it had accomplished, and how much was left. We used Netscape's server-push technology [75] to report how many results a Web search engine had returned as soon as that information became available. However, because most non-Netscape browsers do not implement server-push properly, if at all, this feature is limited to Netscape users.

While reporting to the user which service had returned and with how many results was useful, often one or two services would not return. As a result, MetaCrawler would not display its results until the timeout for its search was reached. MetaCrawler's default timeout was thirty seconds, but was as high as three minutes if the user requested MetaCrawler to download the results. This was often longer than users wanted to wait. To address this problem, we created a Java interface that allowed users to request the results that had already come in and browse those results while MetaCrawler continued with the search.

The Java solution does have one serious drawback. The Java applet opened a separate connection with the server for the applet to get status updates and for the server to know when the user was finished with the search. Unfortunately, most firewalls do not allow these kinds of connections, and thus many users, mostly corporate, could not take advantage of the Java solution.

4.1.3 Result page with click logging

In a format similar to other search services, MetaCrawler displays the document references returned from the underlying Web search services on a single page in a *ranked relevance list*. A ranked relevance list orders document references by their *confidence scores*. A confidence score is just a number indicating how confident the engine is that the document is relevant to the query. Confidence scores range from 0 to 1,000 with 1,000 representing the "most confident." Each document is displayed using the document title as a hyperlink to the document. Following the title is

the snippet of text describing the document returned by the source, with multiple snippets if the document was returned by multiple sources. Following this information is the confidence score, the actual URL of the document, and the list of services that returned the document.

Logging capability is one of the key novel enhancements added to MetaCrawler. Rather than just include a hyperlink to the document in the results, MetaCrawler instead returned a hyperlink to a CGI program that logged various information about the click and returned to the browser a redirect to the proper URL. The browser then loaded the proper document automatically. This logging mechanism is key to evaluating MetaCrawler as well as underlying search services, as it gives us some information about how the information returned by MetaCrawler is used. We will base a large portion of the evaluation in Chapter 5 on this logging information.

4.2 Data fusion algorithm

Once duplicate URLs are identified, MetaCrawler collates the results obtained from the sundry search services into a single ranked list. This is often called *data fusion* or just fusion in Information Retrieval literature. When MetaCrawler is instructed to download the references returned by the search services, fusion is straightforward. MetaCrawler assigns a confidence score to each document using one of two different ranking methods. For queries using the “All these words” or “Any of these words” logic, it uses a TF-IDF based ranking algorithm. TF-IDF stands for *Term Frequency – Inverse Document Frequency*. This comprises a family of weighting originally studied as part of the SMART project [89, 88] that score documents based on how many query terms appear in that document and how often the query terms appear in other documents. When using “Phrase” searching, MetaCrawler scores documents based on how close the phrase appears in the document. The presence of extra terms or missing terms is factored into the scoring. Once MetaCrawler has scored all documents using

a uniform function, it simply sorts all documents by their confidence scores.

MetaCrawler does not download documents to satisfy most queries. Therefore, MetaCrawler must fuse the results using only the limited information available. There are actually a number of techniques for fusion provided by the Information Retrieval literature. Unfortunately, most rely on either known characteristics of the text corpus or having information that makes fusion straightforward [7, 8].

4.2.1 Biased interleave methods

An obvious method one might use is to strictly interleave the results, ordering the top-ranked result returned by each of the search services first, followed by all of the second-ranked results, and so on. This method is ill-defined. For example, consider the following three input sequences, A, B, and C, and three possible fusings, R_1, R_2, and R_3:

$$\left.\begin{array}{rcl} A & = & [a_1, a_2, a_3] \\ B & = & [b_1, b_2, b_3] \\ C & = & [c_1, c_2, c_3] \end{array}\right\} \Rightarrow \left\{\begin{array}{ll} [a_1, b_1, c_1, a_2, b_2, c_2, a_3, b_3, c_3] & (R_1) \\ [b_1, a_1, c_1, b_2, a_2, c_2, b_3, a_3, c_3] & (R_2) \\ [a_1, b_1, c_1, c_2, a_2, b_2, b_3, c_3, a_3] & (R_3) \end{array}\right.$$

The difficulty is that there is no ordering between documents ranked equivalently by different sources. One could select an arbitrary ordering based on some global preference, such as R_1, but that biases the results to the search services ordered first which may not always be appropriate. One could also use a first-come first-ranked ordering that incorporates the speed by which results were retrieved, such as R_2. However, that biases the results to whichever source returns the fastest, not necessarily which returns the best results. One could also use a random ordering such as R_3, which will remove bias, but will likely produce inferior results.

One way of removing artificial bias is to somehow incorporate the confidence score returned originally. This assumes two things: first, that all services return some kind of confidence score, which is not true. The second is that the sundry search services

score pages in generally the same way. A problem with this is that often search services rank several pages equivalently. This is especially problematic when the top pages are all ranked with the search service's top score. This has the affect of biasing the results towards the search services that use scoring functions that rank many pages equally.

A final issue which has not been addressed is how to incorporate information about a single URL from multiple sources. One manner would be to rank a URL returned by multiple sources in the average position, or take the average confidence score returned by all sources. However, this assumes that confidence scores from two different search services can be averaged. Also, it is unclear if giving a URL an average rank is what the user desires. Certainly, if a URL is returned by two or more sources, then it is much more likely to be relevant, and thus should be ordered somewhere above the URLs returned by only one source.

4.2.2 Normalize-Distribute-Sum algorithm

In order to address the problem of biasing results towards a search service that ranked a number of results equally, MetaCrawler uses a three stage algorithm called *Normalize-Distribute-Sum* (NDS) to collate documents. The relevance scores from each service are normalized to $[0..1,000]$. The scores are then redistributed via the following formula:

$$s'_i = \frac{N - h_i + 1}{N} * s_i \tag{4.1}$$

where N is the number of documents returned by the service, h_i is the rank of document i, ranging from $[1..N]$ with 1 being the top rank, and s_i being the original relevance score of i given by the reference source. The redistribution is done to prevent services that return multiple "perfectly" relevant results – i.e. many results that all score 1,000 – from being listed before the results from other services. We then sum the redistributed scores from duplicate entries, and then re-normalize the scores

to $[0..1,000]$ for end-user presentation, keeping with a style similar to other search sites. One important, and intended, consequence of this algorithm is that references returned by two or more sites tend to be ranked higher than references returned by only one.

If we believed that each ranking algorithm is accurate or we knew that all services used the same ranking algorithm, the distribution step would not be needed. In fact, other fusion algorithms, such as those used to merge TREC results [21], would likely perform much better than the NDS algorithm were this the case. We do not yet have either a formal or experimental justification for this algorithm, but it does work well in practice. Some comparison with other Web fusion algorithms [43, 47] as well as ones used in the TREC environment is definitely warranted.

4.3 Heuristics for duplicate detection

When MetaCrawler is instructed to download documents, it is able to use many sophisticated methods to determine duplicates or near duplicates, such as the techniques by Broder or Shivakumar [20, 99]. However, users usually preferred to receive results quickly and did not have MetaCrawler download documents. Therefore, MetaCrawler must use alternative means to detect duplicates. MetaCrawler uses two heuristics to detect duplicate documents. The first, the *Redirect Heuristic*, is designed to detect redirects, symbolic links, and other forms of referencing the same document within the same domain. The second is the *Mirror Heuristic* which is designed to identify identical documents located at different sites. Before we describe the heuristics, it is necessary to review the components of a URL. A URL is comprised of five components: the *Protocol*, the *Hostname*, the *Port*, the *Path*, and the *Filename*. The Hostname can be broken down further into *Machine Name*, *Sub-domain(s)*, and *Domain*. An example is shown in Figure 4.3.

In every index of Web documents, a number of fields are kept concerning each

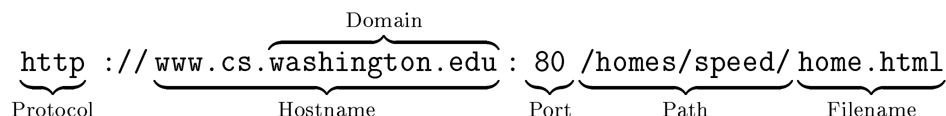


Figure 4.3: The components of a URL.

A URL is comprised of five parts: the protocol, hostname, port, path, *and* filename. *The Hostname is comprised of the machine name, sub-domain(s), and domain. In this example,* `www` *is the machine name,* `cs` *the sub-domain, and* `washington.edu` *the domain.*

document. The URL used to access the document and the title of the document are almost always two of those fields. While two different URLs may be used to access the same document, the title of the document will be the same in both cases.

4.3.1 Default duplicate detection

Before any heuristic is involved, all URLs received by MetaCrawler are put into canonical form. This involves inserting the default port identifier of 80 into the URL if one is not already present, and appending "/index.html" if the URL does not specify a file. For duplicate detection purposes only, file extensions of ".htm" are renamed to ".html." For example, the canonical form of `http://www.washington.edu` is `http://www.cs.washington.edu:80/index.html`. One enhancement which is currently not used is mapping the hostname to the IP address. One could use this technique to detect hostname aliasing, e.g. discovering that `www.washington.edu` and `www3.cac.washington.edu` were equivalent. The reason this technique is not used is that the resource requirements on the DNS service for translating these was substantial and most duplicates of this nature were detected via the Redirect Heuris-

tic. Once URLs are put into canonical form, string comparison detects the obvious duplicates.

4.3.2 Redirect Heuristic

Often, a document will be referenced multiple times on the same site. A common case is a *redirect*, which is when the URL for a particular document changes and the previous URL redirects the user to the new one. To determine if two URLs A and B refer to the same document via a redirect, symlink, or other method, MetaCrawler uses the following heuristic:

Redirect Heuristic:

IF	A and B share the same filename, AND
	A and B share the same non-empty title, AND
	A and B share the same domain
THEN	A and B refer to the same file.

The assumptions made by this heuristic are that any URL to a particular file will use its actual filename, and that within a certain domain, the filename and title are enough to designate a unique document. There are some notable exceptions to this heuristic. In particular, while specifying a title is encouraged, titles are often not specified. The occurrence of an "index.html" file, the default filename, without a title is common enough that we added the requirement of non-empty titles. Another issue with this heuristic is that generic or copied titles can be incorrectly assessed as duplicates. Often, people create HTML pages from previous HTML pages or use a set template, but neglect to change the title. However, practical experience has shown that this does not often occur.

Tables 4.1 and 4.2 show two examples of how the Redirect Heuristic works. Table 1 shows three URLs, all of which refer to the same file. The Redirect Heuristic will correctly identify URLs 1 and 2 as the same file; however, it will incorrectly classify URL 3 as a distinct page. The reason why is made clearer in Table 4.2. Here, we have three URLs, all of which refer three distinct documents in a mailing list archive. All have the same title and path, and only differ in the filename. The Redirect Heuristic correctly classifies all three documents as distinct documents.

4.3.3 Mirror Heuristic

Another common case of duplicates are *mirrors*. A mirror is a copy of a document or set of documents on a different server. Typically, mirrors are used to provide fast access time in geographically remote areas. MetaCrawler uses the following heuristic to determine if two URLs A and B refer to mirrored copies of the same document:

Mirror Heuristic:

IF	A and B share the same filename, AND
	A and B share the same non-empty title, AND
	A and B have different domains, AND
	A and B share a non-trivial path suffix
THEN	A and B refer to two copies of the same file.

Like the Redirect Heuristic, we use a combination of filename and non-empty title to identify potentially equivalent documents. However, in the case of mirrored copies, we are also able to assume that some portion of the path is equivalent. The full paths of two mirrored documents are rarely equal. Sites often host multiple mirrors, and thus each mirror is placed in a subdirectory under a common “mirrors” directory. It is an open question as to how much of the path to use. MetaCrawler uses two-thirds

Table 4.1: Redirect Heuristic example 1.

	URL	Title
1.	http://www.cs.uw.edu/homes/speed/home.html	E's Home Page
2.	http://zhadum.cs.uw.edu/~speed/home.html	E's Home Page
3.	http://bauhaus.cs.uw.edu/homes/selberg/index.html	E's Home Page

The Redirect Heuristic will determine URLs 1 and 2 refer to the same file. Even though URL 3 also refers to the same file, it will not be considered a distinct URL.

Table 4.2: Redirect Heuristic example 2.

	URL	Title
1.	http://info.wc.com/lst/rbots/0274.html	Re: New Robot Announce
2.	http://info.wc.com/lst/rbots/0275.html	Re: New Robot Announce
3.	http://info.wc.com/lst/rbots/0277.html	Re: New Robot Announce

Three URLs from a mailing list archive. All three URLs refer to distinct documents, even though only their filename is different. The Redirect Heuristic will classify all three as distinct documents.

Table 4.3: Mirror Heuristic example 1.

URL
`http://www.acm.org/sigmod/dblp/db/indices/a-tree/s/Selb:E.html`
`http://sunsite.info.rwth-aachen.de/dblp/db/indices/a-tree/s/Selb:E.html`
`http://www.info.uni-trier.de/~ley/db/indices/a-tree/s/Selb:E.html`

Three URLs referring to mirrored copies of the same document. The Mirror Heuristic will classify all three as mirrors.

of the directories in the shortest path of candidate URLs. Again, this has proven to work well in practice.

Table 4.3 shows three URLs, all of which refer to the same document mirrored at three different locations. MetaCrawler first selects candidate mirrors based on filename, title, and domain. It then selects two-thirds of the shortest candidate path, which in this case would be `/db/indices/a-tree/s/`. The Mirror Heuristic would then correctly classify all three URLs as mirrors.

4.4 Parallel I/O

A poor implementation of a meta-search engine would be to query each Web search service serially, waiting for each search service to return its results before proceeding to the next one. The lower bound on the time this approach would require is the sum of the time required for each Web search service to return or timeout. If the requests are made in parallel, then the lower bound on the time it takes to return results to the user is just the time of the slowest Web search service to return or timeout. To have actual performance approach that lower bound requires a sophisticated parallel engine in order to keep overhead to a minimum and enable MetaCrawler to process as much as it is able while waiting for services to return.

4.4.1 Naive process or thread-based approaches

One of the earliest Web libraries was `lib::LWP(3)`, a Perl library for WWW access. Because Perl also had a number of features that enabled ready parsing of pages, Perl was a natural fit for constructing a meta-engine. SavvySearch [47], which predated MetaCrawler's public deployment by roughly a month, was implemented using Perl and `lib::LWP(3)`.

Unfortunately, Perl did not have an easy way to incorporate parallel I/O, so SavvySearch was implemented using a Perl process for each query submission, and a parent Perl process that handled the distribution and collation. This method has two severe drawbacks. The first is that each Perl process requires a large amount of system resources. On a Sun UltraSparc 1, the platform originally used by SavvySearch, a Perl process requires at least 1.7 metabytes of virtual memory to run. A meta-engine using this paradigm for ten sources would require at least 17 metabytes of virtual memory per query just for the Perl interpreter. This problem can be addressed by simply adding virtual memory to a system. The second drawback is that most operating systems have a hard limit on the number of processes per user. The POSIX limit is 64. For a meta-engine with 10 engines, this means there can be at most 6 simultaneous queries. This number is typically an adjustable constant. However, on most operating systems adjusting this constant requires a recompilation of the operating system kernel, a task many users may be unable to undertake.

An obvious solution to using an entire process to handle a query is to use a multithreaded design, such as Java threads, where each query is handled by a single thread. This approach reduces the amount of system resources a single query consumes. MetaCrawler was initially implemented using `pthreads(3)` under DEC OSF v3.2D on the Alpha platform. This worked well initially; unfortunately, when functionality to download pages was added to MetaCrawler, the overall system performance suffered greatly. In addition to using a thread for each Web search service

query, MetaCrawler also used a thread for each Web page it attempted to download. Each machine hosting the MetaCrawler service processed between 5 and 15 queries at a time during peak loads. Each query would attempt to retrieve between 10 and 150 documents. Thus, during peak loads, the various MetaCrawler processes could request over 2,000 threads at once. Unfortunately, DEC OSF v3.2D had a default limit of 256 threads per user. In addition to causing queries to wait in order for resources to become available, new queries were unable to start, which caused many users to receive "Service Unavailable" errors when they attempted to execute their search.

While it was possible to increase the user thread limit, it was still a hard limit which may have been reached under load. In addition, the DEC OSF v3.2D kernel would crash routinely if over 1,500 threads were created. Another approach to address the thread limit problem would have been to use a system similar to Chores [33] which limits the number of threads any process could use, and queues the requests if there are not enough threads available. However, we were experiencing substantial growth in the number of queries being submitted daily, and eventually either solution would likely hit the thread limit. Furthermore, the `pthreads(3)` libraries were not universally available across multiple platforms at the time, especially on Linux, which necessitated exploring an alternative approach.

4.4.2 Event-based paradigm

Our only use for multiple threads in MetaCrawler was to implement parallel I/O. Therefore, the question arises: Is there a way to implement parallel I/O without threads? The answer to this question is, "Yes" and the means to this end is via the `select(2)` system call. However, before we delve into the details of this implementation, it is necessary to examine how the HTTP protocol works.

Retrieving a document from the Web via the HTTP protocol can be represented by a Finite State Machine (FSM). Figure 4.4 shows a simple HTTP FSM. As shown,

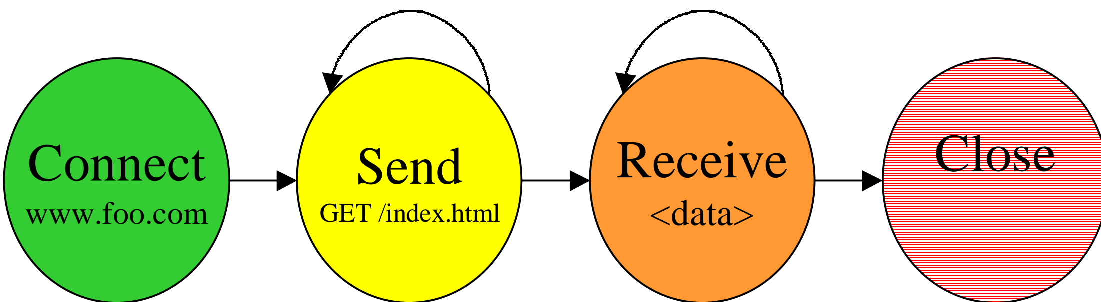


Figure 4.4: FSM for a general HTTP connection.

A FSM for the most simple HTTP session. A connection is opened to a HTTP server, a request for a document is sent to that server, the data is received, and the connection is closed. Notice the self-referential arrows from the Send and Receive states; this is because it may take multiple send and receive calls to transmit all the data.

a connection is opened to the HTTP server, in this example `www.foo.com`. Then, a request is sent, in this case `GET /index.html` which retrieves the site's default Web page. While the data sent in this example is trivially small, typically more data is sent in compliance with the HTTP/1.1 protocol [48]. Due to I/O buffering limits in the operating system, all of the data may not be sent at once, and thus must be sent in smaller segments. The data is then received. Again, due to I/O buffering and potential latency, the data may not be received all at once but in several segments, and thus each segment must be read in turn. The connection is then closed. While the HTTP/1.1 protocol does include a framework for multiple requests being satisfied over the same connection, this functionality was not used in MetaCrawler and will not be described here, although it is not difficult to extend the FSM to include this functionality.

Typically, HTTP libraries use *blocking* send and receive calls. When used, these calls block the program from doing anything else until some data is either sent or received. In addition to the delay that may come with network latency, the operating system may add some delay due to its own resource management. When parallelism is accomplished via multiple processes or threads, the underlying HTTP library uses these blocking calls. Both the Perl and Java libraries are written in this fashion. The alternative to using blocking calls for sending and receiving data is to use non-blocking calls. Non-blocking send and receive calls either send or receive as much data as possible at the instant they are called, and then return control back to the program.

One way of implementing parallel HTTP requests using non-blocking calls is to have a list of FSMs, as depicted in Figure 4.5. The program iterates over the list, causing each FSM to undergo a single state transition until all FSMs reach the *Close* state. When there are many connections, unnecessary delay can occur as the program iterates over each FSM to find those that are able to actually send or receive data. Most send and receive implementations require some system utilization when they

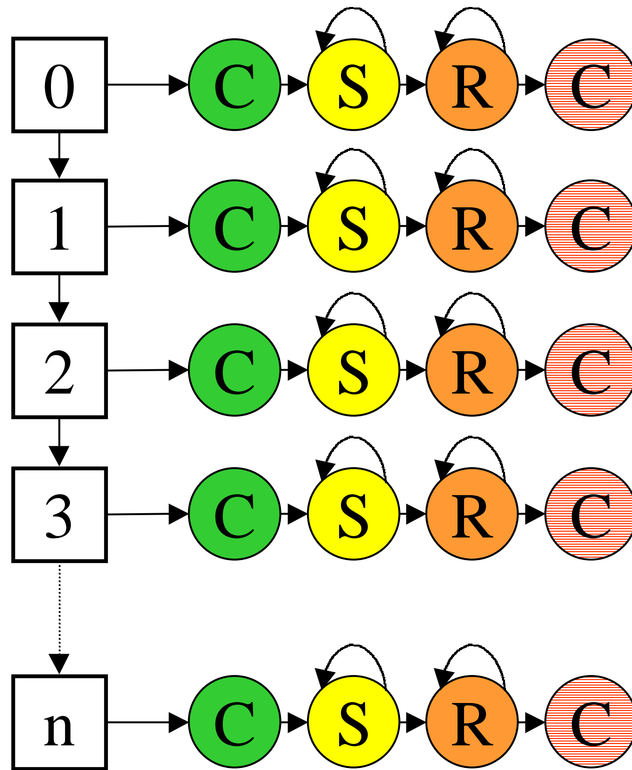


Figure 4.5: FSM network for general HTTP connections.

A network of HTTP Finite State Machines, used to implement parallel Web requests. Each request is realized by its own FSM. The program iterates over this network, causing state transitions in FSMs. When a FSM reaches the Close *state (C), the data has been received or an error state has been reached, and the resources can be reclaimed by the program for another connection. The only inherent limit to this design is the number of open connections in a single process, which is typically 4,093 for UNIX-based systems.*

are called, even when no data is available. Thus, it is wasteful to issue these calls when no data is available.

There is an OS call that informs the calling program when there is data available for send and receive calls: the `select(2)` call. The `select(2)` call accepts as input descriptors to connections that have I/O events pending, and returns the descriptors that have I/O events ready to be processed. Colloquially, programs that use `select(2)` to multiplex I/O are described as "event-based" systems. In order to multiplex HTTP connections as in Figure 4.5, there needs to be a mapping between descriptors and FSMs. Descriptors in UNIX are simple integers, and thus an array using descriptors as the index value is appropriate data structure to use for this mapping.

4.4.3 Event-based paradigm with DNS

One of the deficiencies with the event-based model is that it is not pre-emptive, meaning that the processing of any one state transition will not be interrupted, regardless of how long it takes. In particular, the starting *Connect* state can be quite problematic in this model. Remote sites are typically described using alpha-numeric names, such as `www.cs.washington.edu`. In order to make a connection, these names must be translated into Internet IP addresses. For example, `www.cs.washington.edu` would be translated into `129.95.4.112`. The service that provides this translation is the Domain Name Service, or DNS. Most operating systems have a call which, given a host name, automatically contacts the DNS to perform the translation; in UNIX, the call is `gethostbyname(3N)`.

The `gethostbyname(3N)` is a blocking call. Normally it returns rapidly, so there is not an issue. However, under heavy load, the DNS daemon that provides the translation can become loaded, and thus the `gethostbyname(3N)` call may block the process for a significant amount of time. The default timeout is five minutes. Because the event-based system is not pre-emptive, no work can be done until the

`gethostbyname(3N)` call finishes. This may cause the unwanted timeout of other connections that are otherwise operating correctly.

DNS servers can be accessed using send and receive calls in a similar way to HTTP servers. Therefore, the HTTP FSM model can be extended to include a non-blocking DNS call. This extended FSM is described in Figure 4.6.

Because opening and closing connections to the DNS server is resource intensive, it is advantageous to open only one connection to the DNS server per query and have each HTTP connection use that global connection. This does serialize DNS requests; however, requests that are already connected to remote servers are able to continue unimpeded. This approach has worked well for MetaCrawler practically.

4.4.4 Handling other protocols

One of the features of MetaCrawler was the ability to download the pages contained within the search results to provide extra functionality. While most of the results were pointers to documents available via HTTP servers, a number of them used other protocols: FTP, WAIS, and Gopher being the most common. While these protocols were somewhat more complex to implement than HTTP, it was still straightforward to implement FSMs to realize the retrieval of a remote document. In addition, at the time MetaCrawler was developed the HTTP protocol was undergoing significant change as well. The FSM for HTTP was enhanced with most of the HTTP additions, including the ability to handle redirects and the various supported authentication methods seamlessly.

4.4.5 Evaluation of event-based paradigm

It is difficult to evaluate the performance of the event-based paradigm compared to using threads or processes. Certainly one method would be to run a benchmark that attempted to retrieve a certain number of documents available remotely using all three methods. However, this introduces network latency, which may severely inhibit

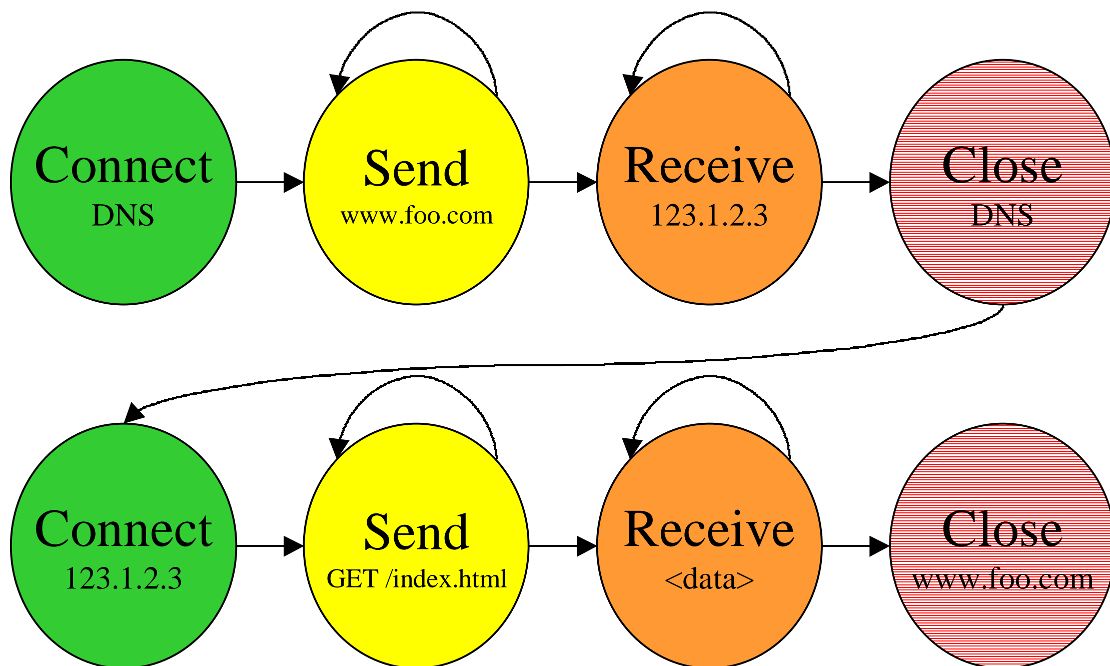


Figure 4.6: FSM for HTTP with DNS.

A Finite State Machine, expanding the Open *state from Figure 4.4 to include states for a non-blocking call to the DNS service.*

accurate measurement. Instead, we conducted two experiments. In the first experiment we examined what the overhead is for processes and threads, and whether that overhead increases as the number of threads and processes increase. In the second experiment we measured how many simultaneous processes, threads, and open connections were attainable. Both experiments were conducted on Digital AlphaStation 500/333, running at 333Mhz with 290M of main memory and an additional 670M of virtual memory, using the DEC OSF v3.2D kernel. The kernel had been recompiled to handle a maximum of 2,068 simultaneous processes and 4,136 simultaneous threads.

Benchmark Results

We measured the overhead of using multiple processes or threads. Presumably, if the overhead is simply a constant factor, then while multiple processes or threads may be slightly slower, they should not limit the number of queries. We measured the overhead by running a benchmark serially for N iterations, and then compared it to using N processes or N threads. To determine if the overhead remained constant or increased, we measured the overhead using N of 100 to 2,000 in increments of 100. We used two benchmarks. The first was a simple busywait loop. The second opened a file, wrote 10K of random data to the file, closed the file, and deleted the file. The second benchmark was used to simulate the I/O dependent nature of MetaCrawler, and files were used instead of Web pages to remove any potential network latency. Figure 4.7 shows the results for the busywait benchmark taking the average time to complete a busywait loop serially, using threads, or using processes. As shown, while there is some overhead, the overhead remains constant for threads. There is some volatility in the overhead of multiple processes, but this does not appear to be tied to the number of processes. Figure 4.8 paints a slightly different picture. It shows that the overhead for processes does in fact increase as the number of processes increases. On the other hand, the overhead for threads does not appear to increase with the number of threads. However, it is worth noting that the overhead is substantially

larger – almost 200% – than the overhead in using the busywait benchmark. While conducting this evaluation, we observed that we were not able to generate over 600 simultaneous processes nor over 1,100 simultaneous threads. After the benchmark program reached those limits, it refrained from creating a new process or thread until an existing one had completed its task. This in effect serialized the program after a certain number of processes and threads had been reached. We were also unable to generate results for 1,600 or more simultaneous processes. Even with the serialization when system resources became unavailable, the operating system halted after creating between 1,500 and 1,600 processes in short succession from a single program.

Maximum open simultaneous connection

While it appeared that the overhead for threads was constant, we observed that the number of simultaneous processes and threads we were able to generate was much less than the system's advertised maximum. Therefore, we attempted to see how many simultaneous threads and simultaneous processes we were able to attain. In addition, we attempted to determine how many simultaneous connections we were able to make. Table 4.4 shows our findings.

We were only able to spawn at most 509 processes on the AlphaStation before running out of system resources. We were able to create 1,027 threads, a little over double our maximum number of processes, before we ran out of system resources. We were able to run 24 simultaneous processes each of which opened 4,093 connections without difficulty.

Although it does not appear that the overhead of threads is significantly affected by the number of threads in a system, there is a fixed limit as to how many threads are available. Since MetaCrawler required more than 1,027 simultaneous connections during peak access hours, neither multiple processes nor multiple threads were an option on the given hardware.

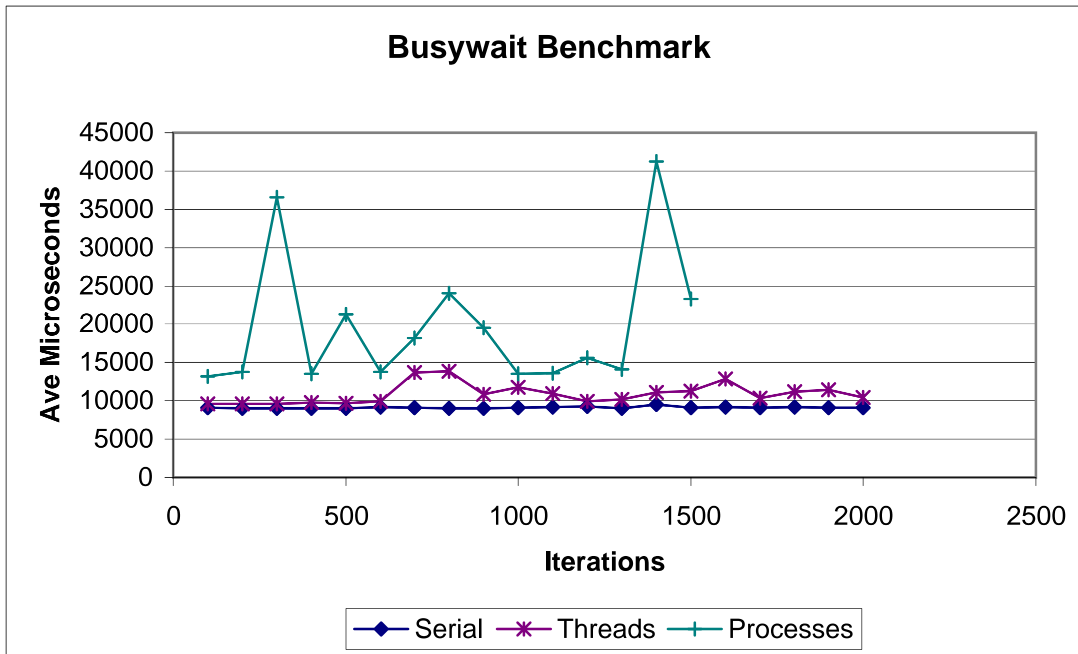


Figure 4.7: Overhead of threads and processes for busywait benchmark.

As shown, the overhead of threads was neither significant nor did it increase as the number of threads increased. The overhead of processes was noticeable and somewhat more volatile, but did not appear to be tied to the number of processes. 1,600 or more multiple processes were not evaluated due to system limitations.

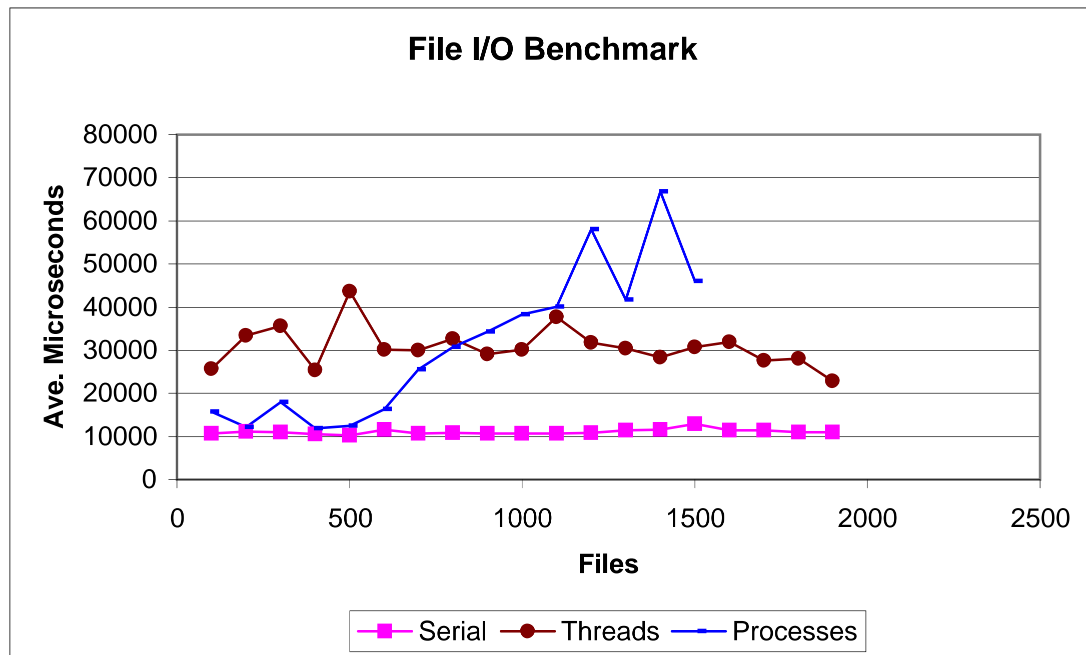


Figure 4.8: Overhead of threads and processes for file I/O benchmark.

The overhead of multiple processes was both significant as well as increasing with the number of processes. The overhead of threads was also significant, but did not increase with the number of threads. 1,600 or more multiple processes were not evaluated due to system limitations.

Table 4.4: Default, adjusted, and actual maximum number of processes, threads, and open connections.

	Processes	Threads	Connections
Default	64	256	31,744
Adjusted	2,068	4,136	31,744
Actual	509	1,027	4,093

This table shows the default number of simultaneous processes, threads, and open connections, the maximum on an adjusted DEC OSF v3.2D kernel, and the actual number attained before reaching resource limitations. While the number of processes and threads depict the maximum number for the entire system, the number of open connections is per process. We were successfully able to achieve 98,232 open connections across 24 processes. Note that these connections were not connected to an available port, of which there are only 31,744 available (32,768 total, with the first 1,024 reserved for the operating system). These tests were conducted on an AlphaStation 500/333, running at 333Mhz with 290M of main memory and an additional 670M of virtual memory.

4.4.6 Results of event-based model

MetaCrawler has used the event-based I/O model with great success. Originally based on the World Wide Web Consortium's `libwww` v4.0d C library, written by Henrik Frystyk Neilsen [73], MetaCrawler's event-based I/O library enabled MetaCrawler to handle in excess of 100 queries at a single time on a single processor. Certainly, there are further improvements that can be made. In particular, an improved implementation of the `select(2)` call, available in more recent versions of DEC OSF, could enhance things dramatically [5].

In addition to the performance improvements, the event-based model used calls that were standard in all POSIX-compliant variants of UNIX. Thus porting the MetaCrawler code to other POSIX-compliant platforms, in particular Linux, was a trivial matter. At the time MetaCrawler was first created Linux did not have a standard threads package. Therefore porting a threads-based MetaCrawler would have necessitated creating our own threads package or using a non-standard threads package.

4.5 Further extensions to MetaCrawler

In addition to MetaCrawler's architecture allowing for a plethora of small-scale enhancements and improvements, it also allows for some larger-scale extensions. We describe two such applications built on MetaCrawler: Ahoy! The HomePage Finder [96] and HuskySearch [93].

4.5.1 Ahoy! The HomePage Finder

A *known item search* is a search for a single piece of information that is known to exist. Optimally, the user will receive only a single result which contains the proper information. The difficulty in handling known item searches is that it is often difficult to properly express the information, thus users typically use an underspecified query and sift through the results for the proper item. One of the most common complaints

in user feedback of MetaCrawler is the number of irrelevant results that is returned.

One domain that demonstrates this problem is a query for a person's home page. Jonathan Shakes, Marc Langheinrich, and Oren Etzioni developed the Ahoy! service as a specialized search service designed to find individual's home pages. Users entered a first and last name, and optionally an organization, e-mail address, and country. Ahoy! then formatted a query for MetaCrawler. Ahoy! uses a specialized Search Client in MetaCrawler that is designed to output the raw results, rather than the formatted results. In addition to querying MetaCrawler, Ahoy! queried various e-mail indices with the person's name in order to discern the person's e-mail address if it was not provided by the user. Ahoy! then used the e-mail address as part of a suite of heuristics to extract home pages from MetaCrawler's output. For its output, Ahoy! would return only pages that met its strict criteria resulting in a typical result of only one or two URLs.

To evaluate Ahoy, Shakes, Langheinrich, and Etzioni compared the performance of Ahoy! to that of submitting just the name of the desired user to MetaCrawler, HotBot, AltaVista, and Yahoo!. Two home page lists available on the Web were chosen: David Aha's list of Machine Learning and Case-Based Reasoning Home Pages [1], a list of 582 home pages, and the netAddress Book of Transportation Professionals [107], a list of 53 home pages. These were respectively referred to as the Researchers Sample and the Transportation Sample. The desired users' names were submitted as queries to MetaCrawler, HotBot, AltaVista, and Yahoo!. At the time of the experiment, HotBot, AltaVista, and Yahoo! all had special syntax for names, whereas MetaCrawler did not. Instead, MetaCrawler used phrase searching when forwarding queries to HotBot, AltaVista, and Yahoo!. Thus the direct query to the three search services could produce better results than the results that are returned from MetaCrawler's query to those services.

The experiments measured whether the home pages were returned and if they were returned as the top ranked document. The results, summarized in Table 4.5, show that

Table 4.5: Top ranked and found percentages for Ahoy! evaluation.

Researchers Sample

	Top Ranked	Found
Ahoy!	74.38%	84.57%
MetaCrawler	22.22%	75.62%
HotBot	26.85%	64.20%
AltaVista	22.53%	58.02%
Yahoo!	0.62%	0.93%

Transportation Sample

	Top Ranked	Found
Ahoy!	65.38%	67.31%
MetaCrawler	19.23%	65.38%
HotBot	11.54%	42.31%
AltaVista	19.23%	34.62%
Yahoo!	9.62%	11.54%

Ahoy! used two sets of Home Pages: A 582 entry Researchers Sample, comprised of academic and industrial researchers, and a 53 entry Transportation Sample of transportation professionals. Both were independently created sets. As shown, Ahoy is able to achieve significant improvement in both ranking accuracy as well as retrieval. MetaCrawler is also able to retrieve more home pages than the other Web search services, showing that meta-search in this domain is advantageous.

in this domain Ahoy! was able to retrieve more home pages than any other service as well as returning them as the top ranked result significantly more often than any other service. The results are also favorable to MetaCrawler, showing that MetaCrawler is second best at retrieving the appropriate home pages. MetaCrawler is also comparable to the rest in terms of ranking the appropriate home pages first, even considering MetaCrawler was using sub-optimal syntax. This demonstrates that extensions to MetaCrawler can lead to significant improvements, and that these improvements are above and beyond improvements made by MetaCrawler to base services.

4.5.2 HuskySearch

Based on our experience with MetaCrawler and our desire to highlight research enhancements, we implemented MetaCrawler version 2.0. Due to contractual issues relating to the commercial operation of MetaCrawler, MetaCrawler version 2.0 was renamed HuskySearch in January 1997.

User interface enhancements

A number of enhancements were made to the User Interface based on feedback from users. Experienced users frequently changed options after every query. To make their experience less tedious, we enhanced the search form so that the query options from the previous query were already selected and the text of the previous query was already entered. HuskySearch was also enhanced to provide superior searching for University of Washington faculty, staff, and students. In addition to searching global Web indices, options were added to allow the user to search local University of Washington intranet indices as well as *The Daily*, the local student newspaper [106].

Previous implementations of the Java version used two different search forms. This was confusing to users, so the two forms were merged and an option was added that selected the output format. With the persistent query option enhancement described above, users only needed to choose once for a peculiar output format for any number of queries. Users were often unclear when documents would be downloaded. The MetaCrawler version 1.0 would only download documents if the user requested phrase searching. Once more services began supporting phrase searching, we made downloading documents an option for all logics. However, this option conflicted with the "Maximum Wait" option. Users could request MetaCrawler to download documents, and return results in 15 seconds. This was not possible in practice. Therefore, we merged the document download option and the maximum wait option into three submit buttons: "Fast Search," which returned whatever results were available within

5 seconds, "Default Search," which returned results within 30 seconds, and "Quality Search," which downloaded the documents and returned results within 3 minutes.

A number of users requested various advanced features. While adding these was not difficult, we were unable to develop a search form that accommodated both advanced as well as novice users. Therefore, we created a "Power Users" page that contained all of the performance parameters, URL filtering options, and other miscellaneous options previously unavailable. We then removed the advanced options from the default search page. Figure 4.9 shows the current search form for HuskySearch.

Engine enhancements

In addition to enhancements made to the User Interface, we made several significant enhancements to the underlying engine. The first, which will be described in further detail in Chapter 6, was the addition of a URL statistics database. This database keeps count of how many times a URL is returned and how many times it is viewed by a user. HuskySearch adjusts a URL's ranking based on this data.

Another enhancement is the addition of two new query options: "Search for a Person" and "Reverse URL Search." Many of the underlying services HuskySearch used had added a "Person" query syntax, and our own informal log analysis showed a substantial amount of queries appeared to contain proper names. Another query option gaining in popularity is the "Reverse URL Search," in which the query is a URL, and the results are pages that have a hyperlink to that URL.

The addition of these two query logics to the user interface was trivial; the three radio buttons that represented the query logic were replaced by a single menu. Adding the functionality to the wrappers whose services did not provide the appropriate logic was also trivial. We enhanced the wrappers of these services with approximations of the logic and added filters to ensure quality results. For example, the approximation for the "Person" logic for services that did not support it was to submit the query `(``First Last'' OR ``Last, First'')`.

Figure 4.9: HuskySearch Homepage screenshot, Aug. 1999.

Some of the options added to HuskySearch were not obvious to naive users. Therefore, brief help that explained the options available were included.

However, enhancing the wrappers of services that did natively support the query logics, while straightforward, was quite tedious. Each wrapper had to be enhanced to support the new syntax in the service's native format. It is unclear if there is a solution to the problem of recoding wrappers to add additional query options. However, as the number of search services increases and search services get more sophisticated and complicated, even solutions that lessen the manual coding requirements will be of great benefit.

4.6 Summary

To summarize this chapter, we have described our implementation of MetaCrawler. We have shown that average Web users can take advantage of comprehensive search through a clean Web interface. A problem with combining results from different search services is that the results may be biased in some fashion. We have presented the Normalize-Distribute-Sum algorithm that collates results from heterogeneous search services that takes into account the potential biases of various services. Another problem with combining the results from different services is that the same URL may be described in different ways. We have presented the Redirect Heuristic and Mirror Heuristic that classify common cases of duplication from redirects and mirroring.

Parallel retrieval of Web pages consumes a significant amount of system resources. We have shown how the event-based paradigm enables an application to download over four thousand pages at once, compared to the common alternative of threads which allows for slightly over a thousand simultaneous retrievals. Finally, we have presented two applications that build on MetaCrawler: Ahoy! and HuskySearch, which implement various features requested by users such as known item searching, persistent options, and query logic for locating people.

Now that we have fully described the MetaCrawler system, we turn to the evaluation of MetaCrawler and of its underlying search services.

Chapter 5

EMPIRICAL EVALUATION

The previous chapters described the general architecture of MetaCrawler and detailed some of its technical aspects. In this chapter we will explore its evaluation, focusing on these key questions relating to MetaCrawler:

- Contemporary Web search services such as AltaVista and Excite are supported by copious resources. For example, the AltaVista Query Engine is powered by more than sixteen Alpha Server 8400 5/440s, each with twelve 440Mhz Alpha processors, 8Gb of main memory, and 300Gb of disk storage [24]. The query interface, spider, and indexer operate on additional hardware. With these resources powering a single search service, is meta-search really necessary in order to obtain a comprehensive search of the Web?

- If no single search service provides a comprehensive search, what is the advantage of combining multiple search services? Do all search services contribute a substantial amount of information, or are the main advantages of meta-search provided by combining only a few select services?

- Even if search services do not provide a comprehensive search of the Web today, they may eventually be able to. Given the current rate of growth of the Web and of the search services, will an individual search service eventually be able to provide a comprehensive search of the Web?

- While the Web is a dynamic entity, many of the documents contained within it are static. A reasonable assumption regarding the Web is that if a document

can be found through a search service, and if it does not change, then it can be found through that search service again. Is this assumption accurate?

5.1 Comprehensiveness of Web search services

We claim that MetaCrawler provides a significantly more comprehensive search of the Web than any single search service. To substantiate our claim, we show that each search service MetaCrawler uses provides a substantial number of relevant results to user queries. We show this by analyzing data obtained through the public use of MetaCrawler. We first observe that the results of the search services are largely unique. We then show that users follow results returned from all of the search services, indicating that all of the services are returning useful results. Finally we detail the actual contributions of each search service in turn.

A more direct method of determining whether MetaCrawler provides a more comprehensive search than any individual search service is to compare the number of documents indexed by the search services to the total number of documents on the Web. While the search services publish the number of documents they index, there is no direct method of counting the number of documents on the Web. Even comparing the size of search service indices to estimates of the size of the Web provides only limited insight into the potential contribution of MetaCrawler. Figure 5.1 shows a possible arrangements of four search service indices covering portions of the Web, A, B, C, and D. While comparing the size of the indices to the size of the Web might indicate that combining the indices would be of great benefit, because indices A, C, and D largely overlap index B, combining the results would have little benefit. Thus, we need to evaluate the overlap among search services in order to determine what the benefit of combining them truly is. Since we are unable to examine the indices of the search services directly, we determine the overlap between search services based on the search results from user queries.

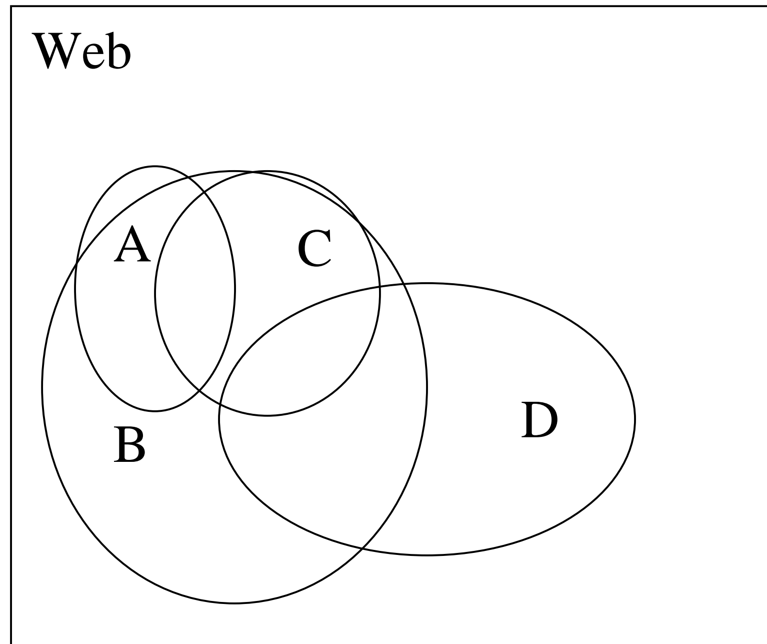


Figure 5.1: An example of search service overlap.

This Venn diagram illustrates a problem in determining the benefit of combining search services by comparing the sizes of their indices. If the indices are largely overlapping, then there will be little benefit to combining them.

A potential problem with using results from user queries to determine overlap between two search services' indices is that the way the search services match documents may adversely bias the findings. As an extreme example, consider two search services that use the same index. One search service attempts to return the most relevant documents to a given query, the other returns a random selection of documents. Based on the overlap between the results, it would seem that the two search services had indexed different documents, and thus combining the two would provide substantial benefit. However, based on the relevant documents returned, combining

the two search services would only increase the number of irrelevant documents in the final results. Thus, it is necessary to not only analyze the number of overlapping documents between search services, but also the number of overlapping *relevant* documents.

Unfortunately, determining which documents on the Web are relevant to a query is impractical since there is no direct way of obtaining all relevant documents without using the tools we are evaluating. Another way to address this problem is to assume all relevant documents were returned through one or more search results. Human assessors can then manually evaluate the results and form the set of relevant results. This is known as the *pooling method* [101], but this approach is extremely labor intensive. For example, human assessors had to evaluate an average of 1,326 out of a possible 3,100 documents for 50 queries in the TREC-6 *ad hoc* query track [110], for a rough total of 66,300 documents. Assuming it takes one minute for an assessor to read and evaluate a single document, and assuming that only one assessor evaluates each document, then it will take 1,105 hours to evaluate all 66,300 documents, or 138 8-hour days for a single person. A common complaint in the TREC series is that 50 queries is not enough to fully evaluate any search engine [110]. Therefore, even 1,105 hours may not be enough to evaluate the documents required for a significant study.

5.1.1 Inference of User Value through Real-world Data

Although there is no standard set of Web queries where all available relevant documents are known, there are millions of users issuing queries to Web search services every day. Therefore, we will use a new methodology to measure the performance of the search services called *Inference of User Value through Real-world Data.* Using real-world data, we will infer the value to the user provided by a search service. This methodology requires both a significant number of queries as well as some measurable quantity by which we can infer value.

To obtain a significant number of queries, we deployed MetaCrawler to the public

in June 1995. Our goal was to have users search the Web using MetaCrawler. After a significant number of users issued queries to MetaCrawler, the information concerning each query could be analyzed. Users will only use a search service if it provides them with some tangible benefit. Therefore, to obtain enough user queries to perform meaningful experiments, it was necessary to make MetaCrawler a high-performance service.

Measuring value through user clicks

While the Web provides an attractive medium for obtaining a large number of user queries, it does not provide any inherent mechanism to measure how many results of a search are relevant to the user. Most search services return a list of document references in response to a given query. The document references contain hyperlinks which lead directly to the appropriate document. When a user clicks on a hyperlink, the user's Web browser contacts the server that provides the referenced page and retrieves the page. Note that the search service that returned the document reference is not contacted during this process, and therefore does not know if a user followed none, one, or all of the results returned.

The most direct method to determine the relevance of a document is to ask the user. However, users are often unwilling to fully evaluate the results of a search. Another method to determine which documents are relevant is to observe which documents the user chooses to view.

As mentioned in Section 4.1.3, the document references which MetaCrawler returns do not contain hyperlinks that lead directly to the referenced documents. Instead, the hyperlinks lead to a program at the MetaCrawler site. This program logs pertinent information about the document being viewed, such as what query it is associated with, its ranking, the time it was followed, and so on. The program then redirects the user's browser to the proper document. Redirects are handled seamlessly by most browsers; therefore, users are not affected by the logging except for a slight

delay caused by following the redirect.

We are able to determine documents of interest by logging the resulting links that are followed. Documents of interest indicate whether or not a search service is returning useful information. Thus, we will infer user value from user clicks on the references returned by a search service.

Viewed documents as an upper bound

We do not make the claim that following a hyperlink implies that the referenced document is relevant. The document viewed may have looked relevant at first glance, but turned out to be irrelevant. The document may have been irrelevant to the query, but was interesting to the user for some other reason. The document may have been irrelevant by itself, but hyperlinks from that document may have lead to relevant documents.

However, we do know that the documents that are *not* viewed are, for all practical purposes, irrelevant. Therefore, the number of viewed documents is an upper bound on the number of relevant documents returned for a given query. We use it as such for the remainder of the thesis.

5.1.2 The 1995 search service evaluation

For our first evaluation of Web search services we observed 20,906 queries from November 26 through December 2, 1995. The first hypothesis we tested was that no single search service provided a comprehensive search of the Web. To satisfy this hypothesis, we calculated the Unique Document Percentage, UDP, for each Web search service MetaCrawler used. Let D_s be the set of documents returned by search service s. Note that D_s is dependent on how many documents are requested from search service s, not how many documents are available. The Unique Document Percentage for a search service is the percentage of document references returned exclusively by that search service, defined by the following equation:

$$UDP_s = \frac{\left|D_s - \bigcup_{i \neq s} D_i\right|}{|D_s|} \tag{5.1}$$

Unique documents were calculated using the heuristics described in Section 4.3. At the time of the experiment, MetaCrawler used Excite [37], Galaxy [35], InfoSeek [49], Inktomi [51], Lycos [69], Open Text [77], WebCrawler [80], and Yahoo! [40]. All of these services except Yahoo! use spider-based indices. Yahoo! uses a manually-generated directory.

MetaCrawler requested ten documents from each service, which at the time was the default number of documents returned by a Web search service. Although some search services were able to return more than ten documents on a given query, each search service s contributed at most ten documents to D_s for a given query.

Figure 5.2 shows the *UDP* for the nine search services MetaCrawler used. In eight of the search services used, over 88% of the documents returned by each service were returned exclusively by that service. It is surprising that Yahoo! did not have a much lower *UDP*. Because Yahoo! is a directory, spiders are able to gather all of the documents available from Yahoo!. At the time these experiments were run, many of the search services used the Yahoo! directory as the initial seed for their spiders, and thus should have had all documents available through Yahoo!.

Even though the results returned by the search services are disjoint, it is unclear whether the services are returning relevant information. In order to make this determination we calculated the Viewed Document Share, *VDS*, for each search service. Let V_s be the set of viewed documents from search service s. The Viewed Document Share is calculated by the following equation:

$$VDS_s = \frac{|V_s|}{|\bigcup_i V_i|} \tag{5.2}$$

The *VDS* of a search service measures the percentage of viewed documents that the search service provided. A viewed document is a document that the user sees

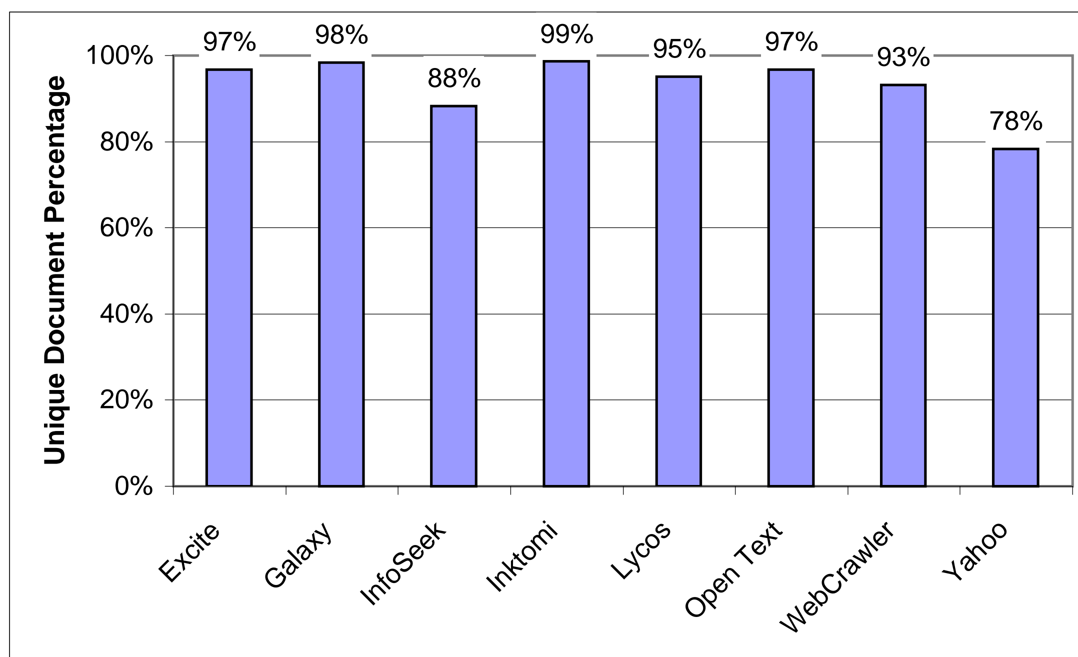


Figure 5.2: Unique Document Percentage, 1995.

This chart shows the Unique Document Percentage for each service. The percentage is calculated by dividing the number of documents returned exclusively from a service by the number of documents returned from that service. These figures are derived from analysis of 20,906 queries from November 26 through December 2, 1995. Ten documents were requested from each service. As shown, over 90% of the documents in all but two services were returned by only a single service.

by clicking on a hyperlink, as described in Section 5.1.1. Documents returned by two or more services are counted for each service. Figure 5.3 shows the *VDS* for the nine search services. Clearly, no service returns a majority of the documents that are viewed. Since the search services all return a large percentage of unique documents, and all search services return documents that users find of interest, clearly none these services is solely able to provide a comprehensive search of the Web.

Implications of only requesting ten documents per service

Because we only compared the top ten results for each query, it is possible that some service provided a comprehensive search, but that its ranking algorithm did not rank the appropriate documents high enough. Therefore, if all of the documents were requested, the results may be different. Unfortunately, requesting all available documents for the 20,906 queries would put an undue burden on the search services. To determine if a search service indexed a followed reference but ranked it lower than ten, we selected a sample of followed references that were returned by other search services. We then submitted the query that returned the original reference to the search service in question and obtained all of the available results. We then scanned those results for the reference that was followed.

Due to difficulties obtaining more than ten results from most of the services, we were only able to complete the test for Lycos. However, as shown in Figure 5.3, Lycos would be the most likely candidate for providing a comprehensive search. We collected a sample of 250 followed references that were not returned by Lycos. Each reference was the only followed reference in the corresponding query results. After issuing the queries that generated the followed references to Lycos, we found 36 references out of the 250, or 14.4%. Therefore, while there may be more overlap if more documents were requested, it would not change the overall conclusion that no search service could provide a comprehensive search.

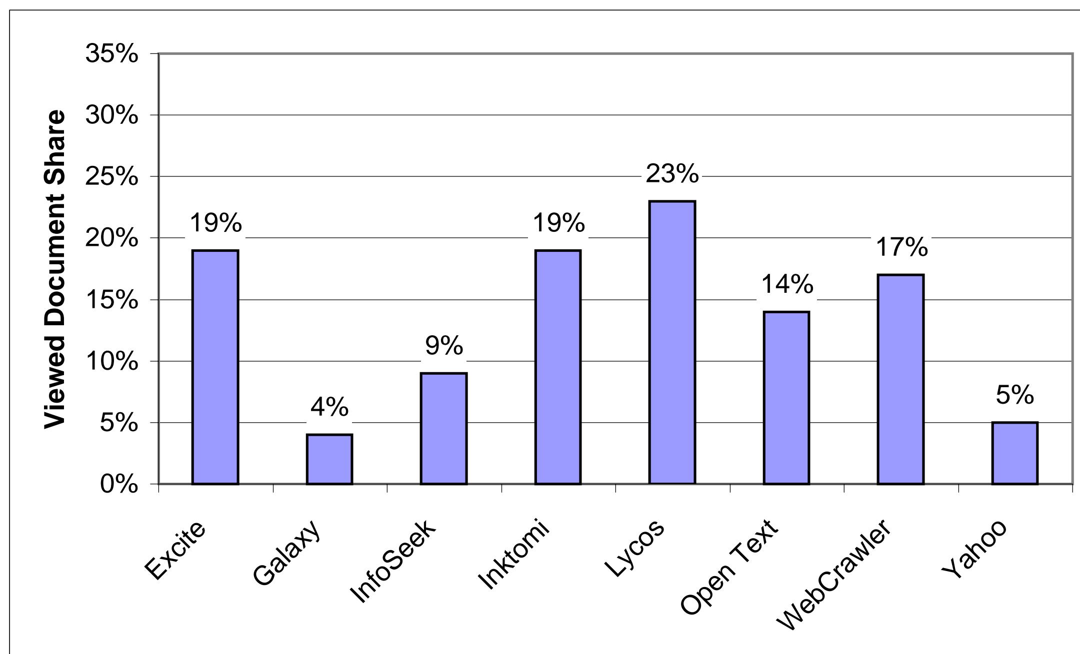


Figure 5.3: Viewed Document Share, 1995.

This chart shows the Viewed Document Share, VDS, of each service. VDS is the number of viewed documents returned by a service divided by the total number of viewed documents. As shown, no service returns a majority of viewed documents, thus no search service provides a comprehensive search of the Web.

5.1.3 The 1999 search service evaluation

While the 1995 findings indicate that no search service provided a comprehensive index of the Web in 1995, it was not clear that this was still the case over three years later. Therefore, we conducted a followup study that analyzed 224,195 queries issued from January 1, 1999 through April 30, 1999. This study used our HuskySearch service, which is a more advanced version of MetaCrawler. We were unable to collect results from Galaxy, Open Text, and InfoSeek. Open Text ceased operation in 1996, and Galaxy and InfoSeek requested that HuskySearch refrain from querying its service in 1997. However, we did collect results from three search services that emerged after our 1995 experiment: AltaVista [30], Google [45], and PlanetSearch [82]. In 1996 the public Inktomi search service became the HotBot search service, although the underlying engine was still provided by Inktomi, Inc. Inktomi also provides the backend search service for Yahoo!. Our figures for Inktomi reflect those of the HotBot service; the graph labels are unchanged for clarity. Instead of requesting ten documents from each service as in 1995 study, HuskySearch requested thirty. The results of our findings are presented in Figures 5.4 and 5.5.

As shown, the *UDP* for all of the services evaluated in 1999 is above 80%. Furthermore, while AltaVista returned the most viewed documents, it returned under a third of all documents that were viewed. Clearly, none of these services were able to provide a comprehensive Web search.

Even though the search services HuskySearch uses do not provide a comprehensive search, it is unclear how advantageous it is to combine them. In particular, it is unclear if the benefit of combining the eight search services could also be obtained by combining seven or fewer. To determine the contribution of each search service in providing documents, we define the Cumulative Document Percentage, *CDP*. Let π be a permutation of n search services such that $\pi_s \in [0 \ldots n-1]$ and for all search

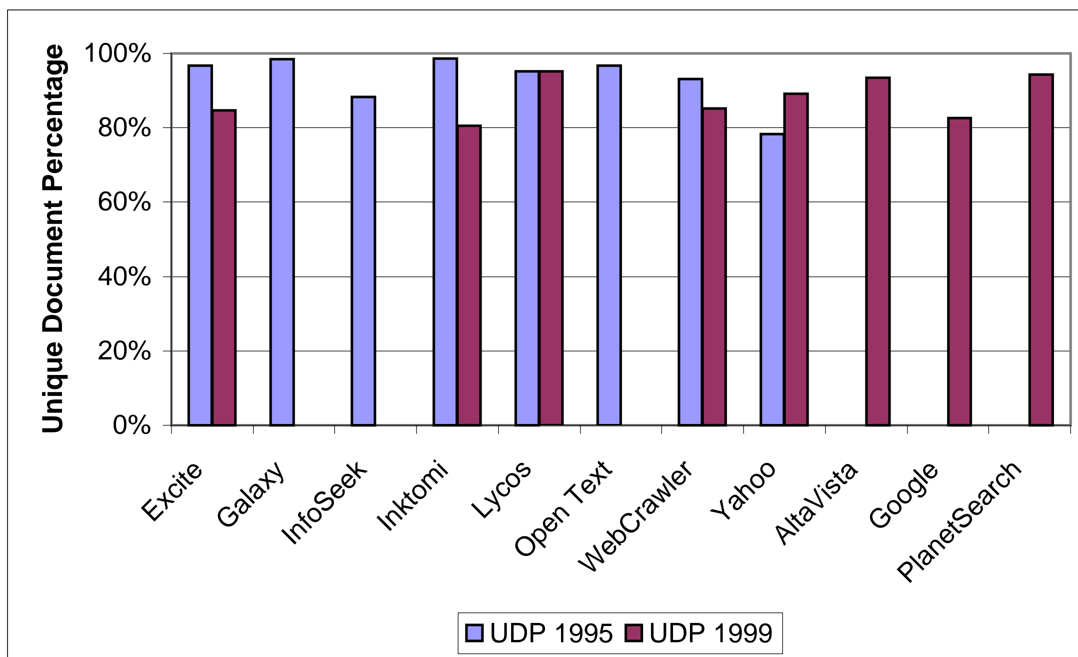


Figure 5.4: Unique Document Percentage, 1995 and 1999.

This chart shows the Unique Document Percentage for November 1995 and January through April 1999. Results from Galaxy, Open Text, and InfoSeek are unavailable. The 1999 results are based on 224,195 queries. Thirty results were requested from each service.

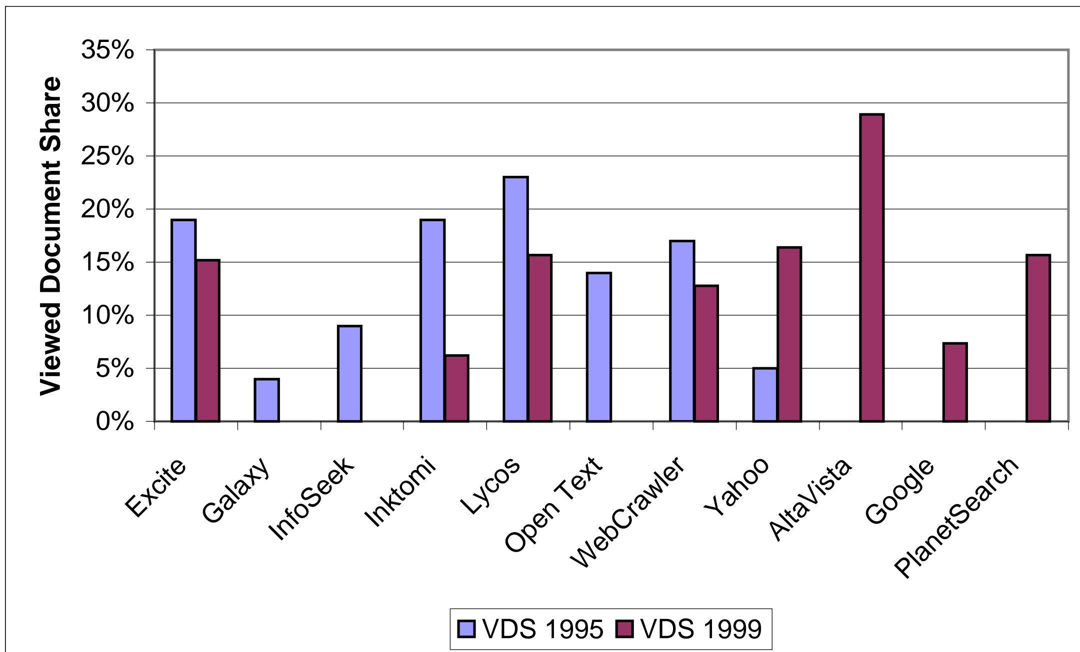


Figure 5.5: Viewed Document Share, 1995 and 1999.

This chart shows the Viewed Document Share for each service, comparing results from the 1995 study to the 1999 study.

services i and j, $\pi_i = \pi_j$ if and only if $i = j$. The *CDP* of search service is defined as:

$$CDP_s = \frac{\left|\bigcup_{i=0}^{\pi_s} D_{\pi_i}\right|}{\left|\bigcup_{i=0}^{\pi_{n-1}} D_{\pi_i}\right|} \tag{5.3}$$

Figure 5.6 plots the *CDP* for the eight search services HuskySearch uses. While the *CDP* can be affected by the ordering of the search services, alternative orderings in this case produce little difference. As shown, each search service does contribute a significant number of distinct documents.

However, just because search services return a large number of distinct documents, that does not mean that they return a substantial number of the documents that are viewed. To determine the contribution of each search service to the number of viewed documents, we define the Cumulative Viewed Percentage, *CVS*, of search service as:

$$CVS_s = \frac{\left|\bigcup_{i=0}^{\pi_s} V_{\pi_i}\right|}{\left|\bigcup_{i=0}^{\pi_{n-1}} V_{\pi_i}\right|} \tag{5.4}$$

Figure 5.7 plots the *CVS* of the eight services in the same order as in Figure 5.6. As shown, each service not only contributes a significant number of documents, but also returns a significant number of documents that are viewed. The reason for this is that most of the documents that are viewed are each returned by only a single search service. Figure 5.8 shows a pie chart that partitions the documents viewed by the number of services that returned them. As shown, 80% of the documents viewed by users were returned by only one search service.

5.1.4 Independent confirmation

Two independent studies confirmed our findings that no search service provides a comprehensive search of the Web. In one, Lawrence and Giles selected 575 queries issued by scientists to the Inquiris meta-search engine [66]. They then retrieved all

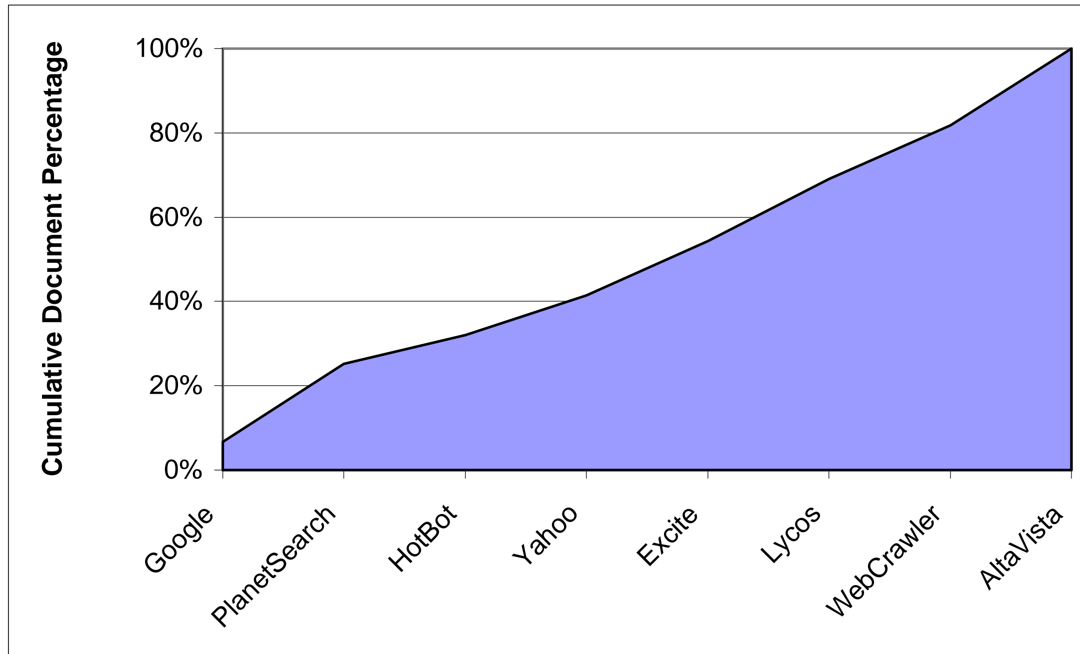


Figure 5.6: Cumulative Document Percentage.

This chart shows the Cumulative Document Percentage, CDP, for each search service. The figures show that that each search service contributes a significant number of the documents returned. There were a total of 6,757,671 documents returned over 224,195 queries. Thirty documents were requested from each search service.

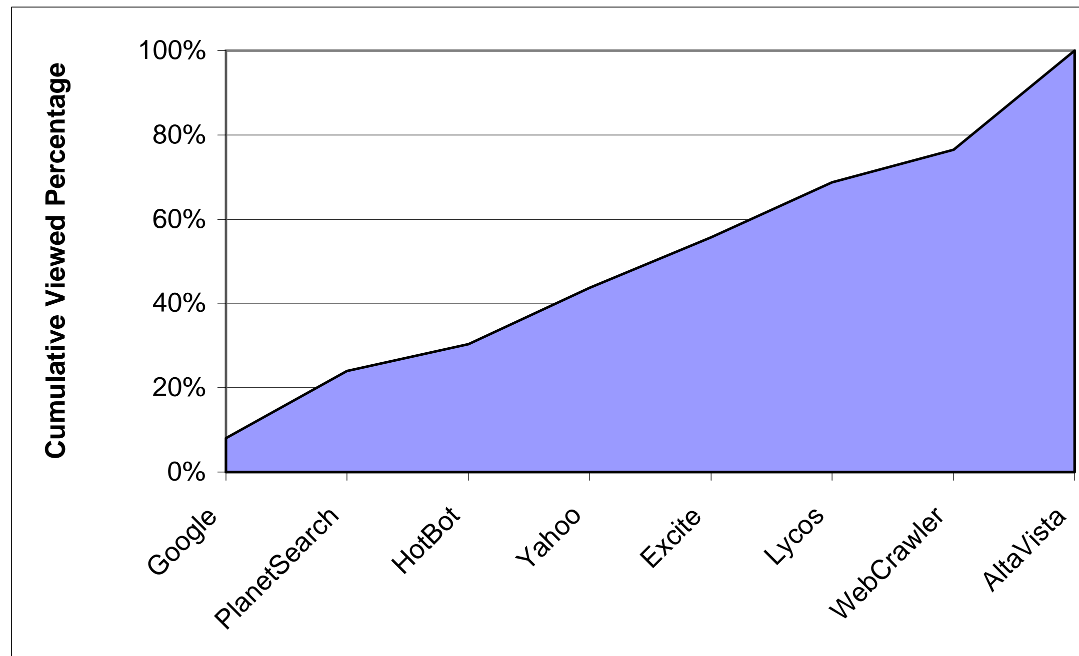


Figure 5.7: Cumulative Viewed Percentage.

This chart shows the Cumulative Viewed Percentage, CVS, for each search service. The values illustrate that each search service contributes a significant number of the documents viewed by users. These numbers strongly correlate with the numbers presented in Figure 5.6. There were a total of 211,850 documents viewed out of 6,757,671 documents returned.

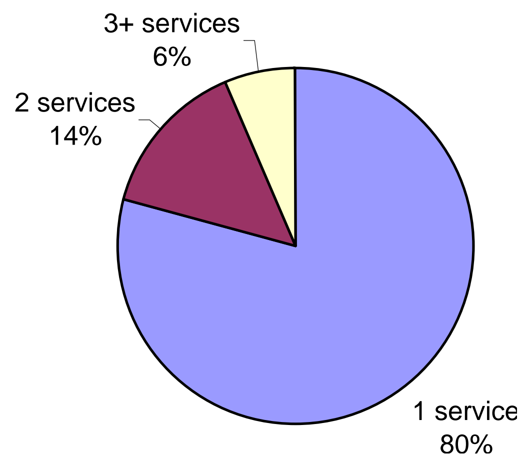


Figure 5.8: Percentage of viewed documents returned by one, two, and three or more services.

This chart shows the percentage of documents viewed that were returned by one, two, and three or more services. These figures are from the 224,195 queries evaluated in the 1999 study.

of the results available from every service and compared the overlap directly. They qualified the 575 queries by ensuring that none of the queries retrieved more than 200 results from any search service. Table 5.1 shows their calculations of *coverage* for six search services. Coverage is defined by the number of references returned by a search service divided by the total number of distinct references returned, or more formally:

$$\text{Coverage}_s = \frac{|D_s|}{|\bigcup_i D_i|} \tag{5.5}$$

For their study, HotBot contained 57.5% of the results available.

While coverage is useful in determining the contribution made by a single service, it does not help differentiate one service from another. For example, while HotBot had coverage of 57.5% and Lycos had coverage of 4.4%, it is not clear what the benefit is of combining the two. The combined coverage could be as high as 61.9% if the two services provide unique results, or could remain at 57.5% if HotBot subsumes Lycos. Thus, we chose to use UDP and CDP in our experiments.

Bharat and Broder conducted a similar study [12], evaluating AltaVista, Excite, InfoSeek, and HotBot. Rather than determining the overlap of search results for a number of queries, they instead selected a number of URLs at random and determined if those URLs were contained within the indices. To find a random URL, they created randomly generated queries based on a lexicon of roughly 400,000 words derived from 300,000 documents available in the Yahoo! directory. They then submitted this query to one of four search services, and randomly picked one of the top 100 references that were returned.

In order to determine if any of the other three search services had also indexed the selected URL, they created a *strong query* for that URL. The strong query is a query meant to uniquely identify a page. Bharat and Broder constructed strong queries by selecting the "k (say 8) most significant terms on the page. Significance is taken to be inversely proportional to frequency in the lexicon" [12]. They then queried the three remaining search services with the strong query and examined whether the selected

Table 5.1: Coverage of search services.

Service	Coverage
HotBot	57.5%
AltaVista	46.5%
Northern Light	32.9%
Excite	23.1%
InfoSeek	16.5%
Lycos	4.4%

Coverage calculations from Lawrence and Giles experiments [66]. Coverage is the number of references returned by a service divided by the total number of references returned by all services. These figures are from 575 queries.

URL was contained within the results.

Bharat and Broder conducted their experiment using four trials, each of which utilized a randomly generated set of roughly 10,000 queries except the third trial which used roughly 5,000. Trials 1 and 2 used disjunctive queries, and trials 3 and 4 used conjunctive queries. Table 5.2 shows their findings.

These figures are broadly consistent with our own and show that no search service provides a fully comprehensive search of the Web. However, our findings for the services' *UDP* were higher than Bharat and Broder's. There are two likely causes of this discrepancy. the first is that in our study we only requested thirty documents from each service. Presumably, requesting more would result in fewer unique documents. The second cause is due to the method in which Bharat and Broder established whether a search service indexed a particular document. A given document may be returned from a query by one service, but due to differences in the indexing and

Table 5.2: Unique Document Percentages for Bharat and Broder.

	AltaVista	Excite	HotBot	InfoSeek
Trials 1 & 2	61%	68%	53%	82%
Trials 3 & 4	38%	80%	52%	83%
1999 Evaluation	93%	85%	81%	n/a

The Unique Document Percentages calculated for trials 1 & 2 and trials 3 & 4 of the Bharat and Broder study. Trials 1 & 2 used randomly generated disjunctive queries, trials 3 & 4 used conjunctive queries. Also presented for comparison are the UDP results from Section 5.1.3 for AltaVista, Excite, and HotBot. Results from InfoSeek were not available. These figures confirm our findings that search services return a largely disjoint set of results for a given query. However, they also indicate that our figures for UDP may be exaggerated because we only request thirty documents from each service.

retrieval algorithms, that document may not be returned by the same query to another service, no matter how many results are obtained. By constructing strong queries, Bharat and Broder were able to determine whether a search service contained a particular document independent of the query used. However, even though they were able to determine that a search service's index contained a particular document, it is unclear if that's of any benefit to the user if the document is not returned given the appropriate query.

5.2 Longevity of meta-search

We have shown that combining the results from several search services provided a significant advantage in 1995 and continues to do so in 1999. However, there is an argument to be made that the real-world constraints described in Section 2.6 are only temporary in nature. Costs of disk storage are decreasing, CPU speeds are increasing, and greater bandwidth is becoming more widespread. Therefore, it could just be a matter of time until a single search service is able to provide comprehensive Web search, obviating the benefits of meta-search.

Two rates are necessary to determine if a particular Web search service will index the entire Web: the rate of growth of the search service's index, and the rate of growth of the Web. Furthermore, if the size of the Web and the size of the index are known, we can predict when the search service will index the entire Web. In Figure 2.6 we presented Sullivan's historical data on the size of Web search service indices. From this data, we are able to calculate trends on the sizes of the indices.

Calculating the growth of the Web is not as straightforward. Since there is no direct means of retrieving all available Web pages, there is no way to retrieve them at two different time points and calculate the difference. However, if an estimate of the size of the Web could be calculated indirectly, then the rate of change can be determined by extrapolating a trend from estimates taken at different times.

5.2.1 Calculating the size of the Web

One technique we can use to estimate the size of the Web is probability theory [66, 12]. Probability theory provides a mechanism to determine the size of a set if two random overlapping subsets are known. The size of the set is derived in the following manner: Let $\mathcal{W}$ be a set, and let A and B be two independently created random subsets of $\mathcal{W}$. Let $P(A)$ represent the probability that an element is a member of A. Since A is a random subset, we know that:

$$\begin{aligned} P(A) &= |A|/|\mathcal{W}| \\ |\mathcal{W}| &= |A|/P(A) \end{aligned}$$

Thus, if we know $P(A)$, we can calculate the size of $\mathcal{W}$. Since A and B are independent, we also know the following:

$$\begin{aligned} P(A \cap B) &= P(A) \cdot P(B) \\ P(A) &= P(A \cap B)/P(B) \end{aligned}$$

By Bayes' Theorem, we can then define $P(A)$ as:

$$P(A) = P(A \cap B|B)$$

Thus, the size of $\mathcal{W}$ is calculated by:

$$\begin{aligned} |\mathcal{W}| &= |A|/P(A) \\ &= |A|/P(A \cap B|B) \end{aligned} \tag{5.6}$$

$P(A \cap B|B)$ can be estimated by selecting a random sample of B and testing whether they are elements of A.

The two studies described in Section 5.1.4 used Equation 5.6 to estimate the size of the Web. The studies chose two search services, A and B, to represent the randomly

created subsets of the Web, $\mathcal{W}$. They then calculated $P(A \cap B|B)$ for one of the services, and used the published size of the index of A to calculate $|A|/P(A \cap B|B)$. However, the method for estimating $P(A \cap B|B)$ differed in the two studies, producing different results.

Lawrence and Giles estimated $P(A \cap B|B)$ by selecting 575 user queries. They issued each query to A and B and retrieved all available documents. They then calculated $P(A \cap B|B)$ for each document in B. Using AltaVista and HotBot as A and B respectively, Lawrence and Giles calculated the Web as 320 million documents in December 1997 [66].

Bharat and Broder calculated $P(A \cap B|B)$ in a more direct fashion, as described in Section 5.1.4. They created a random query, issued it to A and B, selected a random result from B, and checked if it was a member of A. They calculated the size of the Web in November 1997 as 200 million documents [12].

The disparity between the two estimates is most likely caused by various forms of bias introduced in the sampling and checking mechanisms. Lawrence and Giles used queries from scientists at NEC. Since the queries may be esoteric compared to average Web queries, the result of these queries may be equally esoteric. Spiders may be biased towards finding information more applicable to the general public, and thus the search services may only index a small portion of the available relevant documents. Since the inherent likelihood of two search services indexing the same document may be lower than average, the overlap between two search services may be overestimated. On the other hand, the randomly generated queries used by Bharat and Broder may also introduce bias into the system. Since the queries are randomly generated, documents that match the queries are likely to be long, content rich documents. Spiders may have a bias towards locating these documents because of their content. Thus, the Bharat and Broder study may underestimate the size of the Web because the documents they choose are inherently more likely to be contained in multiple search services.

5.2.2 *Search service and Web growth trends*

In a follow-up to their 1997 study, Bharat and Broder calculated their estimate on the size of the Web in June 1997 as 125 million documents, and in March 1998 as 275 million documents [13]. These figures indicate that the Web is doubling every nine months. Another estimate on Web growth is that the Web doubles every year. This estimate is based on the Netcraft Web Server Survey, which counts the number and type of Web servers on the Internet [74]. Figure 5.9 shows the total number of Web servers for each month starting with August 1995. As shown, the number of Web servers doubles every year.

With estimates on the size and growth of the Web, as well as historical data on the size and growth of Web search services, we are able to extrapolate whether any search service will be comprehensive in the near future. Figure 5.10 charts the historical size of the largest Web search services: AltaVista, HotBot, and and Northern Light. The chart then plots an upper bound on their growth until March 2001. The upper bound is the percentage increase from March 1998 through March 1999. In addition, two trends are shown indicating the growth of the Web. The first is based on Lawrence and Giles' estimate of 320 million Web documents in December 1997, using the Web growth estimate of doubling every year. The second is based on Bharat and Broder's estimates of 125 million documents in June 1997, 200 million documents in November 1997, and 275 million documents in March 1998.

Projecting forward, the disparity between the size of the Web and the size of the Web search services only increases. Simply put, Web search services are not doubling in size every year, whereas the Web itself is. Even more apparent is that the advantage of using meta-search will soon become a necessity. At the time of this writing, the major Web search services have each indexed 150 million documents. The estimates of the size of the Web are between 600 and 700 million documents. Even if the indices of the top three Web search services were completely disjoint, they would still be unable

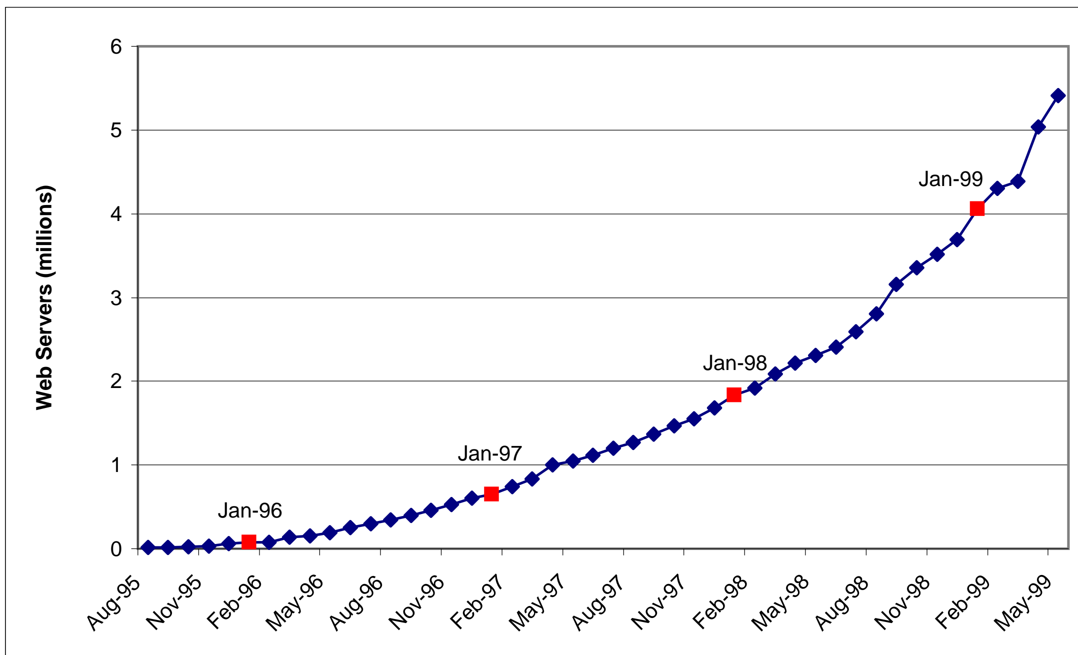


Figure 5.9: Number of Web servers on the Internet.

This chart shows the total number of Web servers on the Internet. As shown, the number of Web servers doubles roughly every year. This data was extracted from the publicly available monthly reports on the Netcraft web site [74].

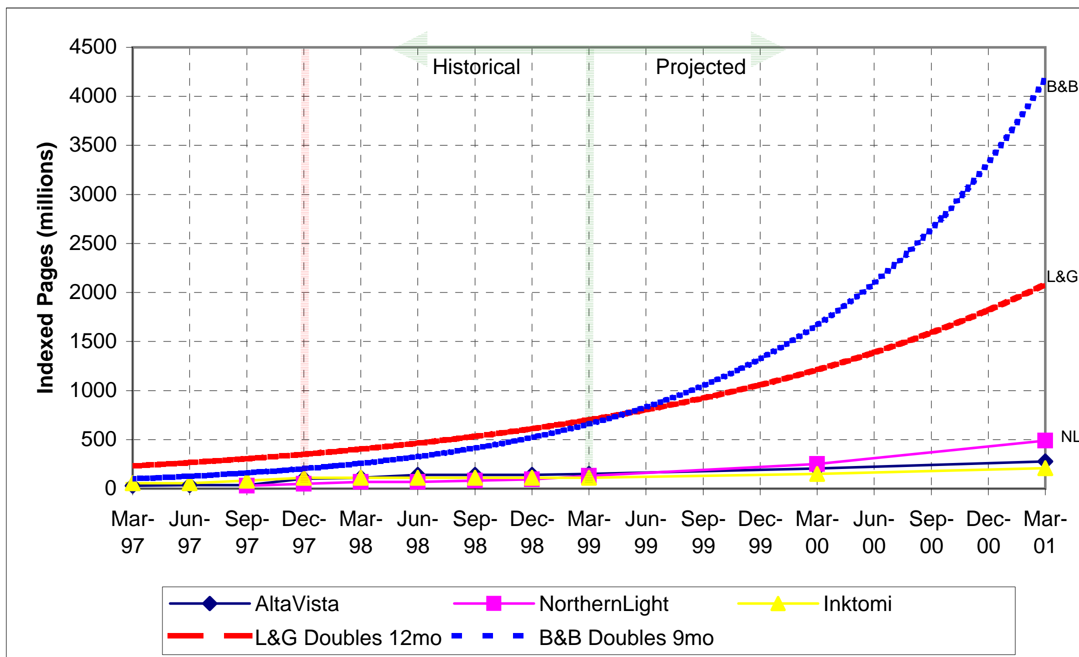


Figure 5.10: Trends of the size of search service indices.

This chart shows the historical data from Figure 2.6 for AltaVista, NorthernLight, and Inktomi (HotBot), and predicts the growth of those search services using the yearly percentage increase from March 1998 through March 1999. It also plots two estimates on the growth of the Web. It shows an estimate based on Lawrence and Giles Web size estimate with the size of the Web doubling every year, and another estimate based on the Bharat and Broder estimate with the size of the Web doubling every nine months. As shown, the size of the Web is rapidly outpacing the size of the search services.

to provide a comprehensive Web search. Thus, the need for continued work regarding meta-search is clear.

5.3 Instability of Web search service

The basic functionality of any meta-engine is to submit a query to many underlying sources and collate the results that are returned. Since the quality of the documents in the collated results are only as good as quality of the documents in the original results, the performance of the underlying search services merits detailed scrutiny. Our experiments, presented in Section 5.1, demonstrate that each of the major search services returned only a fraction of the URLs of interest to users, and that the overlap of the results returned by different search services was surprisingly small.

Another key question is how the sundry search services behave. If we view the Web as a digital library, then the search services could be likened to online card catalogs. All available information is indexed somewhere in the search services, and for a given search, while the results may change slightly over time due to the addition of new titles and the removal of other titles, they will remain largely stable. However, this does not appear to be the case.

5.3.1 Repeated query study

As a means of evaluating the stability of the services, we conducted an experiment to measure the change in the search services' output and compared that change with what was expected. We issued a set of twenty-five queries to nine major search services: AltaVista, Excite, Lycos, HotBot, InfoSeek, Lycos, Northern Light, Planet-Search, WebCrawler, and Yahoo!, and performed analysis on the documents that were returned. The queries are an independently generated set that were used as part of the Lawrence and Giles study. The particular queries are presented in Appendix A. For our study, we modified the queries by removing any syntactic or logical modi-

fiers in the query and treated them as pure keyword queries. All queries except one consisted of two or more words.

Methodology

We define the "instability" of a search service as the difference between two sets of results, calculated from the same query submitted at two different times. Because the Web is a dynamic environment, there will naturally be some degree of instability as documents are added and removed from an index. However, the question remains as to whether or not a search service exhibits instability that is in line with the rate of Web change and growth. To determine the stability of a search service over a period of time, we repeatedly issued a set of queries to the nine major search services. The URLs were then extracted from the results, and were compared with the URLs extracted from a previous submission of the query to the search service.

The twenty-five queries were issued twenty-five times over a one month period, between December 7, 1998 and January 8, 1999. Some search services have reported using a form of query result caching, mostly to improve performance for users who have their browser reload the page, causing a reissue of the query [19]. To avoid measuring the effects of query result caching mechanisms, the interval between issuing the set of queries increased exponentially, with the initial interval being 15 minutes, then increasing to 30 minutes, 60 minutes, and so forth.

Metrics

The URLs in the document references returned by the search services were extracted into a URL set. Only URL sets that came from the same service and same query were compared. As each service is consistent in the way it reports a particular URL, simple string comparison was used to detect duplicates. The difference between the

two URL sets A and B was measured using the bi-directional set difference:

$$\text{URL set difference} \quad = \quad \frac{|(A - B) \cup (B - A)|}{|A \cup B|} \tag{5.7}$$

This measure does not consider changes in the position of URLs because while a change in position can make it more or less convenient to locate information, that information can still be found via the query.

5.3.2 Experimental results

A reasonable assumption to make is that the results of a search service do not change substantially over a month. In the initial query submission run, 25,386 URLs were returned from the nine services. A run done a month later contained 25,756 URLs. However, 8,222, or 32.39%, of those URLs were not present in the initial queries' results. Clearly, substantial change had occurred. Further analysis, broken down by the individual search services as shown in Figure 5.11, shows that change is not isolated to a few services. The output from eight of the nine services changed by over 40%. The only exception is AltaVista, which changed by slightly over 20%. If the results were restricted to the Top 10 results, most services report more change, with InfoSeek, the highest, at 64%.

These initial results were retrieved using the Default search syntax and options of each service, which typically involves some kind of "best match" ranking. "Best match" ranking uses a function that ranks documents based on how many of the query words they contain. Two common alternatives were also examined: submitting the entire query as a phrase, and "AllPlus" syntax, which is to preface each term with a "+" sign. Prefacing a term with a "+" has become standard on Web search services to designate a term that is required to be present in the referenced document. One might assume that the default search syntax produces general results that would change substantially over time, and that more specific syntax would produce results that are more stable. Table 5.3 summarizes the average difference across all services

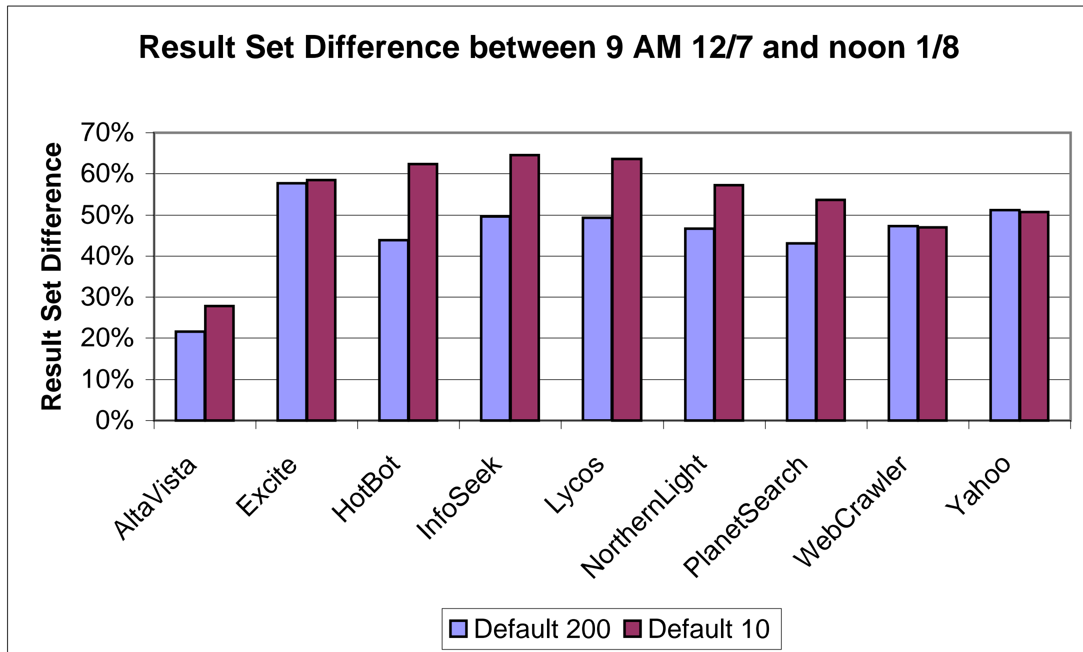


Figure 5.11: Top 200 and Top 10 results using default options.

As shown, all the Top 200 results except AltaVista changed by over 40% difference over the course of 1 month. If only the Top 10 are examined, then most differ between 50% and 60%, indicating that some URLs are being re-ordered. These figures compare the URLs from the first run to the second to last run which exhibited the most change.

Table 5.3: Change over one month for Default, Phrase, and AllPlus options, averaged across all services for Top 10 and Top 200.

	Default	AllPlus	Phrase
Top 10	54.38%	33.34%	30.77%
Top 200	45.60%	22.92%	19.84%

Note that the "Default" syntax and Top 10, which is the common case for most queries [100], changes by over 50%. The best case, using the Phrase option and getting 200 results, is still substantial at just under 20% difference.

for Default, AllPlus, and Phrase syntax. Averaged across all services, the change for the Top 10 was 54.38% and 45.60% for the Top 200. Although there is less change when using different query syntax, the change is still substantial. Furthermore, numerous studies have shown that most users use the default syntax [92, 100]. Thus, our observations regarding queries using the default syntax are likely more applicable to average Web users.

One might argue that this high rate of change in search service output is due to growth and change in the Web. To determine if this is the case, we plotted the average rate of change for the search service output throughout the month for the Default, AllPlus, and Phrase syntax, as shown in Figure 5.12. In addition, we plotted a flat 25% per month growth and change rate for the Web. This rate is an upper bound of the estimated growth rate by Bharat and Broder as presented in Section 5.2.2. The output changes at a roughly 40% linear rate for Default and 30% for the AllPlus and Phrase options. Clearly, while Web change and growth may explain some of the change in the results, it does not explain all of the change.

Another argument for the high rate of change in search service output is that

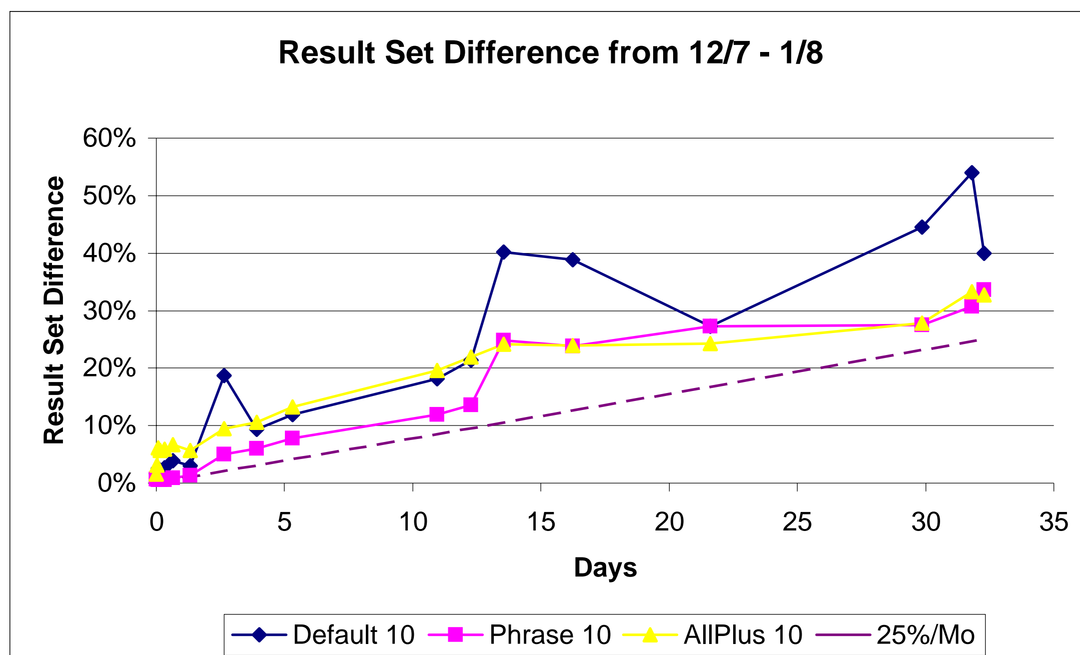


Figure 5.12: Average change over time for Top 10 URLs.

The AllPlus results grow roughly linearly. Phrase results are also fairly linear, with a notable jump near the 20,000 minute mark. Default results are much more errant, with a corresponding jump to the phrase graph as well as two others. However, all results grew faster than the estimate for Web growth and change of 25% per month. The interval was reset to a hour after ten days in order to obtain more data points, and at the termination of the experiment the query sets were run two more times manually, once at noon and once at midnight, to ensure that there were no errors in the automatic submission script.

because the Web is much larger than any one search service, search services may be growing at a faster rate than the Web in order to stay up-to-date and competitive with one another. As we reported in Section 2.6, Sullivan has reported that most services did not report any substantial growth in the testing time period except for NorthernLight, although PlanetSearch was not part of his study [105].

Even though the queries used in this experiment were independently generated, one concern may be they were simply susceptible to returning results that had a high degree of change. To determine if this was in fact the case, we examined how many URLs were removed in subsequent queries only to reappear later. In order to remove the effect of URLs falling outside either the Top 10 or Top 200 window, we looked at which of the Top 10 did not appear in the Top 200 of a subsequent query, only to reappear again in the Top 10 of later query results. Figure 5.13 shows our results. As shown, in five out of the nine services over a third of the URLs returned were temporarily removed from the search results. While not as pronounced, three of the remaining four services also exhibit some of this phenomenon. WebCrawler, which has the lowest reported index of the nine, was the only service that did not display this phenomenon at all.

Since most services show a higher degree of stability when 200 documents are retrieved as compared to 10, and most services reported that they have thousands of matches for most of the twenty-five queries, it is a valid conjecture that by retrieving all available results, there might be substantially less instability in the results over a month's time. This may in fact be true. Unfortunately, some services have a hard limit on how many documents are retrievable regardless of how many are advertised. Therefore we were unable to retrieve all available documents. At the time of these experiments, AltaVista had a hard limit of 200, which was the smallest limit of all nine services. In addition, a histogram of the documents viewed during April 1998 on the HuskySearch service [93] shown in Table 5.4 demonstrates that 99.69% of the documents viewed are ranked 200 or better. A similar study by Silverstein *et al.*

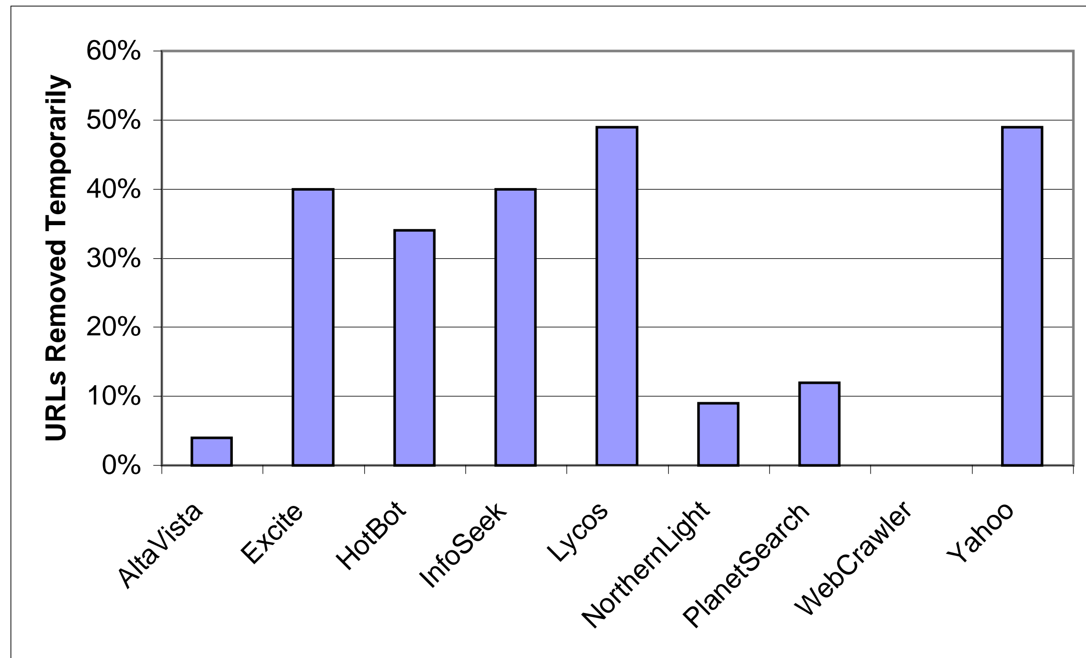


Figure 5.13: Percentage of URLs removed temporarily.

This chart shows, for each service, the percentage of URLs returned in the Top 10 that were not present in the Top 200 of a subsequent result set, only to reappear in the Top 10 of another subsequent result set. These results used the Default query settings; results using AllPlus and Phrase were extremely similar except for InfoSeek, where the results decreased by roughly 10% and 20% respectively.

Table 5.4: Histogram of viewed document ranks.

Rank Range	Pct. Covered
1-10	52.20%
1-30	79.98%
1-50	88.49%
1-100	96.97%
1-200	99.69%

This table shows shows four ranges of URL ranks in a ranked relevancy list, where a rank of 1 is the top of list, along with the percentage of URLs that were viewed were contained within that range. These numbers are calculated from 38,849 URLs that were viewed during April 1998 on the HuskySearch parallel web search service.

on a million queries submitted to AltaVista showed that 95.7% of all users did not look beyond 30 results [100]. These findings may be exaggerated because AltaVista presents ten documents to the user at a time, as compared to HuskySearch, which presents all available documents on a single page. For all practical purposes, the Top 200 documents are sufficient to compare instability.

5.3.3 Analysis

This experiment demonstrates that there is substantial change in Web search service results, and we show that there is substantial change that is unexplained by the Web's dynamic nature. Our hypothesis for this high rate of change in search service results is that the search services trade off quality for speed, and we saw the impact of those tradeoffs. One common technique among search services is to use a limit

on the resources available for each query. Thus, if a system is heavily loaded, the results may not be of as high a quality as when the system is lightly loaded. For our experiment, “lower quality” translates to a greater difference between those URLs and the originals. For example, the spikes in Figure 5.12 are consistent with this technique. The large positive jump near the 12 day mark began with a run that started on Saturday at 11 PM PST, which is a light load time for the search services. The apex of the jump was a run submitted on a Monday at 7:30 AM PST, which is a heavy load time, according to the search services. The negative jump at the end of the experiment started with a run submitted on a Friday around noon, still peak use hours, and ended with a run submitted on a Friday near midnight, which is decidedly off-peak. Both of these jumps occurred when one run was done during “peak” traffic time, and the other during “off-peak” time. The spike that occurred near the beginning of the experiment was the only other occurrence of two consecutive points where one run was during “peak” time and the other “off-peak” traffic times.

Another common technique is to “threshold” results. This technique involves finding a set number of results that match the query within a certain threshold, rather than finding all matches and placing them in an absolute order. This technique is typically used on a portion of the search service’s index that is placed in main memory. Searching a memory-based database is significantly faster than searching a disk-based one; therefore, if an appropriate number of “good” results can be found in main memory, those will be returned under the thresholding scheme. If the requisite number of results are not found in main memory, then the search service will swap a portion of the memory index with a portion of the disk index that does have the appropriate results. Either way these results may not be the best results that could be obtained by searching through the entire index contained on disk. Furthermore, depending on when the query was issued, URLs may be removed and then reappear depending on the contents of the memory-based index. The high rate of change, especially the type of change exhibited by Figure 5.13, is consistent with the application of this

technique.

While these two techniques do address most of the change, each service has a number of proprietary mechanisms which may better explain their behavior. For example, Brian Pinkerton, Chief Scientist at Excite, Inc., explained that every week Excite indexes 10 million pages on the Web at random, and uses these to replace the 10 million oldest pages in its index, regardless of the quality of those pages [81]. Thus, after four weeks, up to 80% of Excite's index may have changed.

5.3.4 Observations about individual services

There were some interesting notes concerning individual services during this experiment. HotBot's ranking implied that many pages were ranked equivalently and that sorting among equivalent ranks was random. For each query option, it had a high rate of change at the Top 10, but much lower rate of change for Top 200. AltaVista did exceptionally well with phrases, with change under 5% for both Top 10 and Top 200. WebCrawler's results had comparable change to other search services with Default syntax; however, it had near-zero change for AllPlus and Phrase syntax. Finally, Yahoo! was exceptionally high in every category. At the time of these experiments, Yahoo! had recently switched its backup search service for queries that contained terms not found in its hand-created directory from AltaVista to a service by Inktomi Inc., which provides the same underlying technology for HotBot. We believe that Yahoo!'s exceptional performance in this experiment is more closely related to Yahoo! bringing their backup search service up to speed rather than an intrinsic behavior of that service.

5.3.5 Impact of unstable results

The output of search services changes at a surprisingly high rate over time, with the rate of change as high as 64%, depending on the search service used and the time between query submissions. Furthermore, as many as 49% of the URLs that appear

in the Top 10 of some result set disappear in subsequent results, only to reappear again later. We hypothesize that the reason for the observed instability in search results is a quality-for-speed tradeoff made by the search services, not the addition of new documents that push older ones outside the 200 document window. The results presented in Section 5.2 show that the most search services are not increasing the size of their indices dramatically, indicating that not only are the search services unable to return information contained within their own indices, but are covering an decreasing portion of the Web.

Unstable search service results are counter-intuitive for the average user, leading to potential confusion and frustration when trying to reproduce the results of previous searches. Scientists who use the Web to locate up-to-date research information run into the same problems. Scientists using search service results as part of their experimental research, such as Lawrence and Giles, need to consider whether instability affects the results of their experiments. Educators who make use of search services in assignments may find those assignments to be unfair because their results cannot be replicated. And people who use the Web for dissemination of information may find that even though a search service has indexed their information, it may still not be retrievable.

In addition, unstable searching does make large-scale, albeit slower search resources more attractive. For example, the Internet Archive [52] is attempting to archive the entire Web. While it will unlikely be as fast as modern search services, it may be both more stable and more comprehensive.

5.4 Summary

In this chapter, we demonstrated the value of meta-search through a number of experiments. We have shown no single search service was comprehensive in 1995, thus necessitating a meta-search service such as MetaCrawler. While the available search

services have changed, the advantages gained through meta-search are just as valuable in 1999 as they were in 1995. Furthermore, we extrapolated the rate of growth of the Web as well as the rate of growth of three largest search services available. We predicted that not only will Web search services be unable to fully index the Web by the end of 1999, but that the three largest search services combined are still not enough to provide a comprehensive Web search. Finally, we demonstrated that not only do search services provide an incomplete search of the Web, but that the results are often inconsistent.

We now turn to Collaborative Index Enhancement, an extension to HuskySearch that not only addresses some of the issues related to inconsistency of search services, but also provides a method of improving search results through user access patterns.

Chapter 6

COLLABORATIVE INDEX ENHANCEMENT

In the previous chapter, we demonstrated that MetaCrawler provides a significant improvement over searching using traditional spider-based and directory-based search services. We also showed current trends indicating that meta-search will continue to provide a substantial improvement over traditional search. However, while MetaCrawler does provide a significant improvement over using single Web search services, MetaCrawler does not provide a complete comprehensive search, nor does it provide a stable search. The Web is larger than the combined size of the search service indices, and the results of contemporary search services are currently unstable. Search services return documents, then sometimes omit them from results, then return them again at a later time.

An approach we took to address some of these difficulties is to incorporate data obtained from previous user interactions into the current search. In this chapter we present Collaborative Index Enhancement, or CIE, a general model for incorporating data from previous user queries into the MetaCrawler architecture. We show how certain instantiations of CIE are able to address the problem of unstable results. In addition, we demonstrate that the overall performance of search is improved through the use of prior query data.

The remainder of this chapter is organized as follows:

- We begin with an overview of the two types of collaboration, direct and indirect, and highlight some representative techniques. We then present a general model for indirect collaboration.

- We describe the design and implementation of our CIE prototypes in Section 6.2. We show that its incorporation into MetaCrawler does not require modifications or additions to the overall architecture. We showcase four enhancement techniques that use CIE. Two techniques use the entire results of a search as a document in its own right, and two use individual results as augmenting information to enhance relevancy ranking criteria. We also examine how the degree of user interactivity affects the enhancement.

- Having described the CIE system, Section 6.3 presents an evaluation of these systems via a controlled user study and a 12 week log analysis.

6.1 Using collaboration to improve retrieval performance

Collaboration is a general technique that allows users to help one another locate relevant information. Collaboration can be used in one of two ways: directly and indirectly. Direct collaboration occurs when a user locates information directly from another user. Recommending a doctor to a friend and forwarding an interesting article to a colleague are two real-world examples of direct collaboration. Indirect collaboration occurs when a user obtains information from other users indirectly. The New York Times Bestseller lists and The Billboard Hot 100 are two real-world examples of locating information through indirect collaboration.

On the Internet, there is a large potential for finding information through collaboration with several million people. We now explore work done using both direct and indirect collaboration online.

6.1.1 Information retrieval through direct collaboration

One of the most compelling instances of direct collaboration is collaborative filtering [97]. Collaborative filtering is based on the assumption that if person A likes items X, Y, and Z and person B likes X and Y but has not seen Z, then person B

would probably like Z. People utilize a collaborative filtering system by giving the system a personal profile. The system then retrieves potentially useful information by locating users with similar profiles and retrieving the information that those users found relevant. This has been successfully implemented by HOMR, the Helpful Online Music Recommendation Service [72], now part of the Firefly network [41]. Users gave HOMR a list of their favorite Compact Discs (CDs). HOMR then located people with similar CD lists and recommended the CDs from their lists that the user had not seen. A similar system is the GroupLens project, which uses collaborative filtering to recommend USENET news articles based on whether or not other users with similar profiles read a particular article [58].

6.1.2 Information retrieval through indirect collaboration

Collaboration can also be used to locate information indirectly. For example, a system may enable all users to enhance a single, centralized information source. Thus, rather than obtaining information from users who share similar profiles, the results which are obtained are enhanced by the tastes and preferences of all users.

Enhancing a searchable index based on user interaction has been experimented with as early as 1971, when Brauen reported on experiments using interaction to modify a searchable index [17]. He used the SMART [89] Information Retrieval search engine as his test bed. In the SMART index, a document is represented by a vector of degree n, where n is the number of distinct words found in all available documents. Each element of the vector represents the *weight*, or importance, of the word in the document. A common formula for weight is the number of times the word appears in the document divided by the length of the document. Brauen enhanced the index by directly modifying the document vectors in the index. He obtained user feedback on relevant and irrelevant documents. He increased the weights of terms that appeared in relevant documents in every document vector, and likewise decreased the weights of terms that appeared in irrelevant documents. He found that this type of enhancement

led to higher *precision*, the percentage of relevant documents returned in a query, and *recall*, the percentage of all available relevant documents returned in a query.

Optimizing database queries can be expensive. Raghavan [84] described an elegant method of locating stored optimized queries previously submitted by users. He compared the results from the current, unoptimized query with those of the optimized queries. His work focused on the tradeoffs of obtaining possibly suboptimal results from prior queries to the computation time of optimizing the current query.

Traditional Information Retrieval focused on providing documents based on rich, well-formed queries. In contrast, in the 1990s users tend to submit queries containing only two or three words [79, 100]. Not surprisingly, traditional Information Retrieval engines often perform very poorly on the Web. A method to improve the performance of traditional search engines on the Web is to automatically expand the query prior to submitting it. Fitzpatrick and Dent explored using prior queries to add terms to a query automatically, and demonstrated performance improvements on TREC benchmarks [42].

Indirect collaboration can also be used to enhance search results. Direct Hit is using collaboration to augment the ranked relevancy lists returned by Web search services [31]. The Direct Hit system logs which documents were viewed in a similar fashion to HuskySearch. Direct Hit then uses that information to rank popular documents higher.

6.1.3 A general model of indirect collaboration

Collaborative Index Enhancement, or CIE, is a general model for enhancing a searchable index through indirect collaboration. Let I be a searchable index, and let d be the data extracted from a user query. Let $enhance()$ be a function that enhances a searchable index I given d. CIE can then be defined as the process of obtaining an

enhanced index I' from I and d in the following manner:

$$I' = enhance(I, d) \tag{6.1}$$

CIE can be instantiated in a number of ways through different definitions of d, the user data, and the $enhance()$ function. We now describe how we designed and implemented a CIE prototype in the HuskySearch system, a publicly available Web metasearch service derived from MetaCrawler [92, 94], and showcase four instantiations of CIE.

6.2 Collaborative Index Enhancement design

A CIE instantiation is incorporated into HuskySearch by creating a separate searchable index that contains the enhancements derived from previous queries. A wrapper for this index is then added to HuskySearch. When a user makes a query, HuskySearch searches the CIE index in parallel with the other services, and the CIE results are merged in with the documents from traditional indices. Additional CIE instantiations can be integrated by adding a new index for each new CIE instantiation. This is depicted graphically in Figure 6.1. By using this design, we are able to implement a CIE system without modification or control of the original Web indices used for searching. To distinguish Web indices from CIE indices, we will often refer to CIE indices as *auxiliary indices* or just auxiliaries.

This design fits seamlessly into the MetaCrawler architecture described in Section 3.3. Additional wrappers are added to the Harness, and a URL Database Manager module is added to the I/O Layer. The Database Manager will be described in Section 6.2.3.

Four CIE indices are described in this chapter:

ReturnedURLs: ReturnedURLs contains all the documents referred to by results documents of previous queries; i.e. for all queries, the referenced documents are downloaded and inserted into this index.

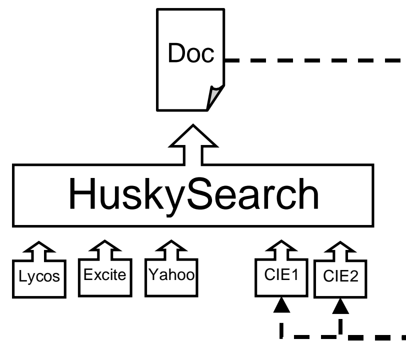


Figure 6.1: CIE architecture.

Each index feeds a list of documents into HuskySearch, which collates these documents into a single results document containing a list of references. This document is then fed back into the CIE auxiliaries which process and index it.

ClickedURLs: ClickedURLs contains all the documents referred to by results document of previous queries that were viewed; i.e. each document in ClickedURLs was viewed by some user in the course of a search.

ResultsPages: Each document in ResultsPages is the *results document* from a particular previous query; i.e. it is the HTML page containing URLs, titles, and snippets to other pages.

SuccessfulResultsPages: Each document in SuccessfulResultsPages is a results document where at least one of the results was clicked on.

We will describe each of these in terms of the model presented in Section 6.1.3. Before we proceed, we define HuskySearch in terms of the CIE model. Although HuskySearch is a meta-engine and contains no index of its own, its index can be considered to be a union of the indices of the services it queries. Let I_s represent the

index of search service s. HuskySearch's index is defined as:

$$I_{HS} = \bigcup_{i} I_i \tag{6.2}$$

Let q be a user query. We define the document $d_q = HS(q)$ as the document obtained by issuing query q to HuskySearch. This document is the *results page*, which is simply a list of document references. We define D_q to be the set of documents referenced by d_q. Note that $D_q \subseteq I_{HS}$. We define V_q to be the set of documents in D_q viewed by the user.

6.2.1 Additional information and improved ranking

The ReturnedURLs and ClickedURLs indices are used in two fashions: to return useful documents not returned by any other search service in response to a user query and to increase the ranking of useful documents returned by other search services. Although both indices are populated by documents obtained from the other search services, there are queries where these auxiliaries will return certain documents while the other search services will not.

In particular, these indices address the problem of unstable search, as presented in Section 5.3. All of the documents retrieved via a prior query will be returned by issuing the query to the ReturnedURLs index. The ClickedURLs index provides similar functionality but only for documents that are viewed by a user. Thus, if a user issues a particular query and follows a document, the user will always be able to obtain that document, even if the search service that originally returned that document fails to provide it in a subsequent search.

The two indices increase the ranking of documents returned by other search services through HuskySearch's scoring algorithm. When two or more services return the same document, the document's confidence score is the sum of each service's score. Thus, when ReturnedURLs or ClickedURLs returns a result that another service also returns, the result's confidence score is automatically increased by some amount rela-

tive to the scores returned by each service. Using a scoring function similar to the one described has been shown to generate better results when using the TREC corpus in traditional Information Retrieval experiments [7] as well as on the Web [92].

The ReturnedURLs auxiliary is comprised of all the documents contained within HuskySearch results. The user data d and enhancement function $enhance_{RU}$ are defined as:

$$\begin{aligned} d &= D_q \\ enhance_{RU}(I, d) &= I \cup d \end{aligned}$$

and thus ReturnedURLs is defined as:

$$\begin{aligned} I'_{RU} &= enhance_{RU}(I, d) \\ &= I \cup D_q \end{aligned} \tag{6.3}$$

The ClickedURLs auxiliary is a subset of the ReturnedURLs auxiliary. Rather than include all documents returned from a query, ClickedURLs only includes the documents viewed by a user. The user data d and enhancement function $enhance_{CU}$ are defined as:

$$\begin{aligned} d &= V_q \\ enhance_{CU}(I, d) &= I \cup d \end{aligned}$$

and thus ClickedURLs is defined as:

$$\begin{aligned} I'_{CU} &= enhance_{CU}(I, d) \\ &= I \cup V_q \end{aligned} \tag{6.4}$$

6.2.2 Convenient query expansion

Query expansion has been shown to produce better results in traditional Information Retrieval [9]. However, numerous studies have shown that Web users do not modify

or expand their queries [100, 64]. This is consistent with our own findings which we will present in Section 6.3.1. It is unclear why users do not attempt to modify their query. Our hypothesis is that if users were given a more convenient method of using query expansion, they might take advantage of it.

The ResultsPages and SuccessfulResultsPages indices are used to implement a form of convenient query expansion. Results documents from previous similar queries are interspersed among the normal results of a query. Users are able to view the results of a previous query in the same manner as viewing a document. Figure 6.2 illustrates this by showing four document references returned by HuskySearch in response to the query "Utah Jazz."

The rational behind using results documents for query expansion is the fact that the snippets describing a document tend to highlight the important terms in that document. While some of these terms are the query terms from the previous queries, others may be related terms. This conjecture is based in part on work by Spink, who demonstrated that the majority of terms users selected for refinement came from document title and descriptor fields [102]. By treating a results page as a normal document, the terms contained within the snippets form a description of the documents referenced using the most descriptive terms available. If a query then matches a results page, presumably the documents referenced on that page are relevant to the query in some manner.

One important distinction between this form of query expansion and other methods is that there is no mode shift in the interface — users are never required to enter a "query refinement" stage, such as selecting relevant documents for feedback or by entering in new terms manually. Because each former results document is available in a ranked list with other potentially relevant documents, users are able to treat previous results documents as just another document with potentially relevant links. This integrated listing also provides the user with an indication as to how relevant a particular query refinement is as compared to other actual documents that proceed

Utah Jazz

WebCrawler: The Official Site of the Utah Jazz Home Court Today's Headline...

Yahoo: official site of the Jazz, featuring news, schedule and scores, players, stats, ...

`601 (1/27) http://www.nba.com/jazz/ (WebCrawler: 2 Yahoo: 2)`

HuskySearch Query: Jerry Sloan

57 references collated. Jerry Bannon at an Icelandic Party on Long Island, 1973.

`101 (0/16) http://huskysearch.cs.washington.edu/...`

`(SuccessResPgs: 37)`

HuskySearch Query: (NBA History)

169 documents collated. NBA at 50: TOP TEN COACHES IN NBA HISTORY. NBA History: Wilt Chamberlain.

`98 (0/9) http://huskysearch.cs.washington.edu/...`

`(SuccessResPgs: 40)`

U of Utah BM Jazz Performance Degree

`52 (0 / 1) http://www.music.utah.edu/ugrads/programs/BMjazz.html`

`(Lycos: 26)`

Figure 6.2: Sample HuskySearch results from the query "Utah Jazz."

Shown are four sample document references obtained from the query "Utah Jazz." The middle two references are actually references to the previous queries "Jerry Sloan," the coach of the Utah Jazz, and the phrase "NBA History." These previous queries are interspaced among normal Web documents to give the user an indication when query expansion may produce better results than further examining the available document references.

it, so that the user does not need to read through a preset number of references before determining that refinement is necessary.

One interesting phenomenon we observed while building this system is that the ResultsPages index keeps an implicit query history for each user. Thus, if a user is looking for a site she previously found via HuskySearch and has trouble remembering the query she used, it is possible for her to search for the previous query explicitly by entering query terms.

The ResultsPages auxiliary is comprised of all the results documents returned by HuskySearch. These documents are lists of document references. The user data d and enhancement function $enhance_{RP}$ are defined as:

$$\begin{aligned} d &= d_q \\ enhance_{RP}(I, d) &= I \cup \{d\} \end{aligned}$$

and thus ResultsPages is defined as:

$$\begin{aligned} I'_{RP} &= enhance_{RP}(I, d) \\ &= I \cup \{d_q\} \end{aligned} \tag{6.5}$$

The SuccessfulResultsPages auxiliary is a subset of the ResultsPages auxiliary. Rather than include the results document from all queries, SuccessfulResultsPages only includes results documents where at least one of the documents was viewed by a user. Recall that V_q is the set of viewed documents, and thus if $V_q \neq \emptyset$ then at least one document was viewed. The user data d and enhancement function $enhance_{CU}$ are defined as:

$$\begin{aligned} d &= \begin{cases} \{d_q\} & \text{if } V_q \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\ enhance_{SRP}(I, d) &= I \cup d \end{aligned}$$

and thus SuccessfulResultsPages is defined as:

$$
\begin{aligned}
I'_{SRP} &= enhance_{RSP}(I, d) \\
&= I \cup \begin{cases} \{d_q\} & \text{if } V_q \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned} \tag{6.6}
$$

6.2.3 Scaling CIE

Given enough time, every document on the Web might be returned by some query, and every document on the Web might be viewed. Thus, ReturnedURLs and ClickedURLs would simply be indices of the entire Web. Real-world constraints aside, if these auxiliaries were comprehensive Web indices, they would add little value to a search because they would simply return the exact same set of results the other Web search services returned, modulo the minor differences in how the ReturnedURLs and ClickedURLs indices match documents.

"Enough time" could be as little as a single day. HuskySearch requests 30 documents from eight search services. If we assume HuskySearch returns 100 documents on average to a query, after ten million queries ReturnedURLs could consist of up to one billion documents. At the time of this writing, the Web is not estimated to be that large. However, ReturnedURLs could likely contain a large percentage of the Web documents indexed by the the Web search services. All of the Web search services used by HuskySearch report they receive over ten million queries per day. Thus, if HuskySearch became as popular as the underlying Web search services, it could take as little as a single day to populate ReturnedURLs with most documents available on the Web.

A single CIE auxiliary needs to be relatively small compared to the index it is enhancing. Therefore, as part of our CIE system we maintain a separate database that contains statistics on documents, such as when a particular document was last viewed and the number of times it was retrieved. Using this information, we can ensure that the sizes of the auxiliaries are reasonable by removing documents which

do not appear to add value to current searches. The database is accessed through HuskySearch by a URL Database Manager, which is a module added to the I/O Layer as shown in Figure 3.2.

In addition to providing information which keeps the size of CIE auxiliaries reasonable, the URL database also enables us to augment the ranking of URLs directly based on prior user interaction. For example, documents that are viewed often can be ranked higher, and documents that are frequently returned but never viewed can be ranked lower.

While HuskySearch actively uses the URL database to augment its results, the CIE auxiliaries have not grown large enough to require removal of documents. We conjecture in Section 6.3.2 that we are not even close to the number of documents where scaling would be necessary. Thus, we do not present any data on the effectiveness of this scaling mechanism at this time.

6.2.4 Hardware and software

The CIE indices used by HuskySearch use the Verity Search '97 search engine v2.0, running on a DEC AlphaStation under DEC UNIX 3.2. HuskySearch also uses AltaVista, Excite, HotBot, Lycos, PlanetSearch, WebCrawler, and Yahoo!, as well as two University of Washington intranet search services: The Daily [106], the local student newspaper; and UWSearch, an index specific to HuskySearch using the Verity engine.

6.3 Experimental validation

We designed our system to find relevant documents on the Web. Since there is not yet an available standard test corpus of Web documents, we chose to conduct experiments using the Web. Another approach would have been to use existing static collections of non-Web documents, such as the TREC collections [109]. However, due to the

limited number of available queries, we did not feel that these collections would fully illustrate the benefits of CIE.

We present two sets of experiments. The first is a user study that evaluates all four CIE auxiliaries where all of the users involved were attempting to answer the same five questions. This test was performed to see which, if any, of the auxiliaries looked promising. The second experiment is based on analysis of 12 weeks of log entries from public use of HuskySearch. In this case, we did not know if a particular user would be looking for information that was at all related to what previous users were searching for.

6.3.1 User study

We conducted a user study to confirm that the results returned by the CIE indices are in fact useful to a group of searchers trying to answer five different questions. For this study, we were interested the following questions:

- *Are users able to answer more questions correctly with CIE?*
- *Are users better able to determine they have the correct answer with CIE?*
- *Are users able to answer questions faster with CIE?*

There were two groups in this user study: a control group that did not use the CIE indices, and a CIE group which had full access to the four CIE indices. The users for this study were volunteers from a second-year graduate Library and Information Science class, nearly all of whom had some experience with searching the Internet. A few had prior experience with HuskySearch and MetaCrawler. Users were given a brief tutorial on the system, but because of previous search experience they were not given an in-depth tutorial on the basics of Web searching.

The students were asked to answer five questions to the best of their ability, and were told they did not have to spend more than 10 minutes per question. On

forms they received, they were to write down the query terms, start and stop time, how confident they felt in their answer, and any comments or problems they had using the system. The study was conducted by distributing instructions and forms to volunteers, who then answered the questions on their own time. The instructions and forms given to the users is reprinted in Appendix B. Although the volunteers were divided into two equal groups, not everyone completed and returned the response form, and some did not provide enough information to be useful for the study. Twenty-five volunteers completed valid forms, fifteen in the control group and ten in the CIE group.

The five questions were:

Kevin: *What are the three most recent roles Kevin Spacey has played?*

Utah: *Which Utah ski resort has the highest elevation, and what is it?*

Tree: *Find a picture of a Fraser fir. Please give the URL.*

Can: *How many members are there in the Canadian parliament?*

MS: *What was Microsoft's IPO price?*

These questions were created so that each would have a single correct answer that was not immediately searchable from the terms in the question. Three of the questions involved finding a numeric answer, and one involved finding a graphic image. In addition, the last two questions involved a temporal aspect, although we did not initially intend for the *Can* question to have one.

We attempted to order these questions from easiest to answer to most difficult based on our own experience in finding the answers. This was done to mitigate any potential bias caused by users learning to use HuskySearch as they worked through the questions. In addition, the users were all given an initial sample question with

Table 6.1: Percentage of users accurately answering each question.

	Kevin	Utah	Tree	Can	MS
Control	100% (0)	87% (2)	80% (3)	40% (6)	7% (14)
CIE	100% (0)	80% (2)	90% (1)	60% (4)	30% (7)

Numbers in parentheses indicate the number of users that were unable to answer the question. Note that starting with the Tree *question, more users in the Control group had difficulty answering the question correctly or at all.*

which to learn the system. For each question in the study, users were asked to write down the answer, starting and stopping time accurate to the minute, their belief in the correctness of the answer, and the search terms for each query they made to HuskySearch. Users described their belief in correctness by circling one of "Very sure," "Pretty sure," "Not sure," or "Didn't finish."

Accuracy

To determine whether the CIE users were better able to answer questions correctly, we compared the percentage of accurate responses in the control and CIE groups. These percentages are shown in Table 6.1. The two easier questions, *Kevin* and *Utah*, were answered well enough by the base system that the CIE auxiliaries didn't have much of an impact either way. The accuracy rate of Group 2 begins to surpass the control group starting with the *Tree* question as more people in the Control group were unable to answer the question correctly or at all. Of note is the *MS* query. Because "Microsoft" and "IPO" are such common terms, users were unable to make much headway using the obvious keywords. However, a few users in the CIE group commented that they were able to take advantage of previous queries that directed

Table 6.2: Uniqueness of queries.

	Kevin	Utah	Tree	Can	MS
Queries	38	34	31	34	54
Terms / Query	2.95	3.94	2.91	2.76	3.04
Pct. Unique	81.6%	82.4%	80.7%	85.3%	92.6%

This table lists the number of queries submitted across both the CIE and Control groups, the average terms per query, and the percentage of queries that are unique. Note that even though there were a large number of queries which tended to use about three terms, over 80% of the queries were unique.

them to the proper site. In particular, a user commented that the inclusion of the term "1986" from a previous query was instrumental in finding the data. Thus, from this data, it appeared that CIE aided in answering questions not easily answered using the obvious keyword searches. For the *Can* question, 20% more users answered the question correctly, and in the MS question, 27% more users answered the question correctly.

This example highlights an important aspect of using results documents from previous queries as indexed documents. In a small group searching for a similar data, members of the group are able to collaborate by using the queries of others, even if their own query is unique. Table 6.2 shows the total number of queries, the percent of unique queries, and the average number of terms per query. Uniqueness is determined by case insensitive string comparison over the query. Since Web search engines treat the query "x AND y" differently than the query "y AND x" we consider those queries unique, even though they would not be in a true Boolean environment. Even with users trying to answer the same question using between two and four terms,

the percentage of unique queries is staggering, at over 80% for each question. This underscores the benefit for this kind of collaboration in guiding users to queries that provide useful information.

Confidence

Previous user studies have noticed discrepancies between the number of relevant documents users thought they had found compared with the actual number of relevant documents available [34, 103]. We conjectured that since our CIE system would highlight documents previously found, users would have a more realistic picture of the correctness of the answer they found. For each question, we asked the user to enter their confidence in the answer as one of "Very sure," "Pretty sure," "Not sure," or "Didn't finish." For those who were able to answer the question, we converted their confidence to c having possible values of 0, 0.5, or 1 (corresponding to "Not sure" through "Very sure") and compared their confidence in their answer to the actual correctness of it, a, which was either 0 or 1, via the formula:

$$\text{Judgment Accuracy} = 1 - |a - c| \tag{6.7}$$

This formula evaluates to 1 if they were correct and were "Very sure" or if they were incorrect and "Not sure," which is the phenomenon we wanted to measure. These results are summarized in Figure 6.3.

The data shown indicates that for the first three questions, both control and CIE users think that they are able to accurately judge their results to the same degree. However, the *Can* and *MS* questions suggest that for the harder questions, CIE users are both more sure of themselves and are more likely to have a realistic belief in their results.

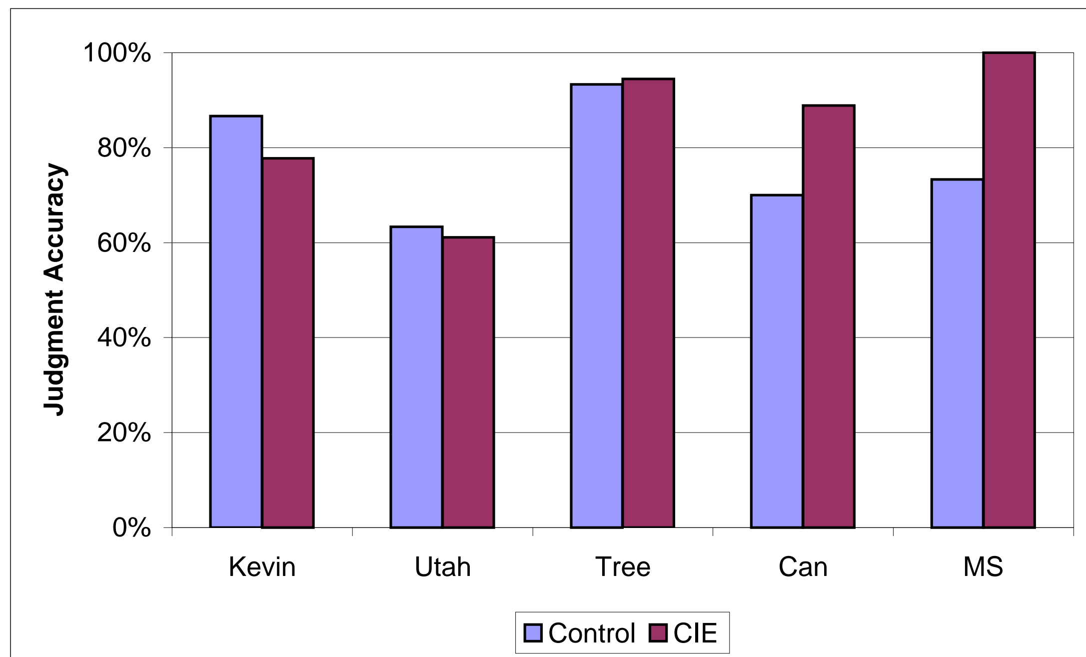


Figure 6.3: Accuracy of result judgment.

This figure shows the percentage of time users correctly judged whether their answer was accurate or not. As shown, CIE users were better able to judge the correctness of their answer on harder questions.

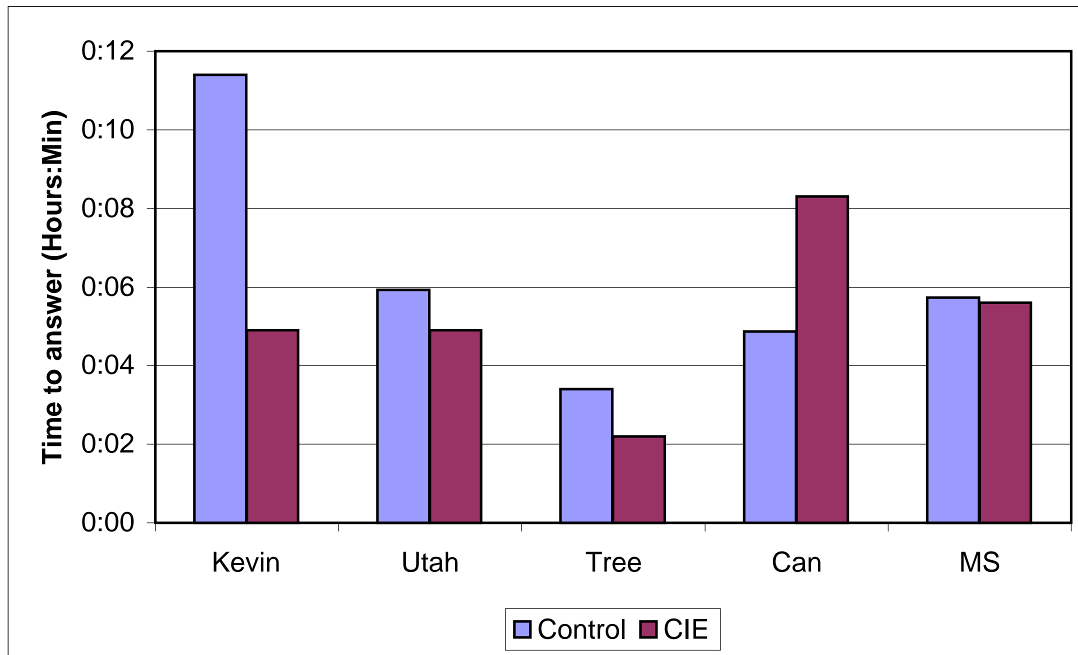


Figure 6.4: Time to answer each question (Hours:Min).

As noted, CIE users were able to answer the question faster except for the Can *question. This is due to there being two "official" answers. CIE users spent additional time to find the correct one.*

Speed

In addition to accuracy and confidence, we also measured whether users were able to answer the questions faster with the CIE system. Users were asked to time themselves, accurate to the nearest minute, and they were asked to write down comments and their searches as they went along, so these numbers should be thought of as approximations. Our findings are summarized in Figure 6.4.

The *Kevin* and *Can* questions stand out. The rest suggest a trend that CIE users are able to answer faster, but the differences aren't statistically significant. The *Kevin*

question shows that the CIE users were able to find the answer roughly 50% faster than the control group. This is surprising, as we feel that this is the easiest question and that CIE won't contribute that much to its success. However, we observe from the user comments that many in the CIE group found a Yahoo! Filmography page in the top few selections; this was caused in part by the ReturnedURLs and ClickedURLs groups boosting that page's ranking.

The *Can* question, regarding the number of members in Canada's parliament, was also surprising. Exploring the user's answers also revealed some confusion as to the "proper" answer to the question. Several users responded with the correct answer: 104 in the Senate, 301 in the House of Commons. However, many responded with 104 in the Senate and 295 in the House, which was the correct number for the previous year's Parliament. After a brief search, it became clear that several official Canadian government pages had not been updated with the new figure. From the user comments, it appears that several users in the CIE group apparently spent extra time trying to determine which number was the current and correct number. In this case, the extra information returned by the CIE system was actually a hindrance, as it often contained wrong or out-of-date information.

The *Can* question illustrates a potential downside to our CIE system. If in fact users find data that is inaccurate, the enhancement method may make it more likely for future users to also find the inaccurate data. Thus, care must be taken so that inaccurate or irrelevant index enhancements can be detected and removed.

Multiple queries

On the user form, there were spaces listed for three queries, and users were instructed to write down information about further queries on the back. Table 6.3 lists the percentage of time users entered secondary and tertiary queries. Only the *MS* query, which went unanswered by the students, had a significant amount of secondary and tertiary queries. This gives the indication that users make only a single query over

Table 6.3: Percentage of users making secondary and tertiary queries.

	Secondary	Tertiary
Kevin	34.6%	11.5%
Utah	23.1%	7.7%
Tree	15.4%	3.8%
Can	23.1%	7.7%
MS	65.4%	42.3%
Silverstein	22.4%	8.9%

This table shows that users did not often make secondary or tertiary queries. There were only 4 queries in the entire study that were beyond the third. This data includes queries from both the Control and CIE groups. For comparison, we included the findings from Silverstein et al.*'s study on the number of secondary and tertiary queries.*

50% of the time and try to find the information they require by visiting subsequent hyperlinks in the returned results. Indeed, many comments from the users indicated that they spent a fair amount of time “digging” through links. While this phenomenon is consistent with other findings [100, 64], what is surprising is that the user form had an inherent bias towards having the users modify their query two or three times, and yet they still issued only one query.

Significance of user study

While we were pleased with the results of our user study, it should be noted that the results were not statistically significant. The study comprised only of twenty-five users split into two groups, and only five questions. It has been argued in the TREC community that fifty questions are not enough to fully examine a system [110]. Certainly, the questions were created by the authors, potentially introducing some bias, and there were other aspects of the study that may have also introduced some bias, such as the ordering of the questions. However, while flawed, the user study did provide insight as to how the CIE system would be used by users and how it could be improved. For a more in-depth study, we now turn to our log analysis.

6.3.2 Log analysis

We demonstrated that CIE is useful in situations where a small group of users are trying to answer the same set of questions. Next, we evaluate which of the CIE auxiliaries are useful on a general purpose Web search service, involving a much larger group of users searching for a wider range of information.

In the ideal case, the CIE auxiliary should only return useful documents. If a CIE auxiliary returns a document that no other Web search service returned, it should nevertheless be a useful document. If a CIE auxiliary returns a document that some Web search service also returns, then it should augment that document’s rank properly. To determine how well CIE performs these two tasks, we evaluate

our CIE auxiliaries again using our Inference of User Value through Real-world Data methodology described in Section 5.1. However, we need to use slightly different metrics to determine CIE's performance.

Open public evaluation of CIE auxiliaries

We incorporated all four CIE auxiliaries into the publicly accessible HuskySearch Web Search Service [93]. To determine if the auxiliaries were returning useful documents, we calculated the *View Rate* for each auxiliary. Let n_s be the number of documents returned by search service s. Let v_s be the number of documents returned by search service s viewed by the user. View Rate is the number of documents viewed by a user divided by the number of documents returned by a search service, defined by the following equation:

$$\text{View Rate of } s = v_s/n_s \tag{6.8}$$

The View Rates were calculated from logs spanning a twelve-week period starting Monday, June 1, 1998 through Monday, August 17, 1998. This log is comprised of 92,072 queries and 84,240 viewed documents. Figure 6.5 shows that ClickedURLs is the most promising auxiliary, while ReturnedURLs is rather poor, with SuccessfulResultsPages being somewhat worse, and ResultsPages having almost no impact whatsoever.

Over the course of the twelve-week evaluation, we received user feedback indicating that they disliked the way SuccessfulResultsPages and ResultsPages were incorporated into the HuskySearch result list. Many users were confused because clicking on these documents did not lead to a page, but instead executed another search. This resulted in many users actively avoiding these documents as they did not want to incur the delay of another search. While there is still potential promise in using SuccessfulResultsPages and ResultsPages as an alternative method of query expansion, interspersing links to new queries among documents to documents did not appear

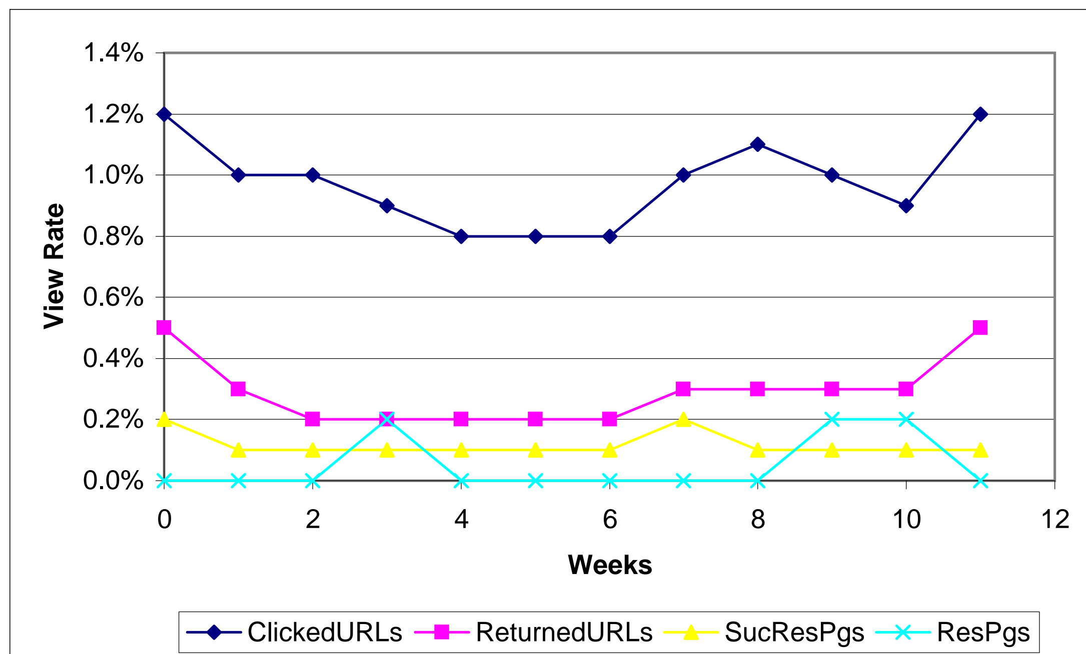


Figure 6.5: View Rate of each CIE auxiliary.

As shown, ClickedURLs is significantly better overall than the other three CIE auxiliaries.

to be a good design choice.

Evaluation of ClickedURLs

Having established that ClickedURLs appeared to be the most promising of the four auxiliaries, we focused our efforts on evaluating its performance over time. We sought to answer the following questions:

- *Does the View Rate of ClickedURLs improve over time?*
- *Do the results from the ClickedURLs auxiliary give the user additional documents the user will view?*
- *Does the re-ranking effect of ClickedURLs rank viewed documents higher on the list?*

Optimally, to answer these questions we would like to study ClickedURLs in a large-scale setting, over several months, on a search service that received over a million queries per day. However, this was not feasible, so we focused on studying the twelve weeks of logs. In Section 6.3.2 we extrapolate how our results might scale on a large-scale system.

Given that HuskySearch receives about 1,000 queries per day, we felt that 12 weeks was an adequate period of time to study the effects of ClickedURLs, and that an additional month or two would not change the results. Manual inspection of the logs from August 17 through September 30, 1998 supports this decision.

ClickedURLs performance over time

We first sought to determine the ClickedURLs auxiliary's performance over time. We calculated the View Rate of ClickedURLs as well as the View Rates for the major services HuskySearch accessed. Figure 6.6 shows our findings for ClickedURLs as well

as three representative Web search services: AltaVista, Excite, and Lycos. The other services used by HuskySearch all had similar graphs; for readability we present just these three.

ClickedURLs typically has a lower View Rate than the three search services. However, while the three services in question vary quite noticeably in their View Rate, ClickedURLs is relatively constant. It does not increase in performance over time, but neither does it decrease. Further analysis showed that ClickedURLs returned 0 documents 39.45% of the time and 1-5 documents 20.54% of the time. This indicates that most queries are diverse enough not to have many matches from the documents contained within ClickedURLs.

Contributions of ClickedURLs

Each document in ClickedURLs was previously returned from some other search service. In more formal terms, if a service E is given query Q_1 and returns a document U that is then viewed, it is added to ClickedURLs. However, a key point is that if ClickedURLs is given a query Q_2 and returns U, it does not hold that E will also return U if given query Q_2. We were interested in measuring how often ClickedURLs returned a viewed document when no other service did, and how much of a contribution those viewed documents made to the overall system. To measure this, we evaluated the frequency and number of unique documents returned by ClickedURLs and viewed by a user. We then compared it with data for the same three representative services used previously.

To evaluate the effectiveness of ClickedURLs at returning documents viewed by the user that no other service returned, we evaluate the *Unique Contribution* of ClickedURLs and the three representative services. The Unique Contribution of a service is defined as the number of unique documents returned by a service *that were viewed* divided by the total number of documents that were viewed. Recall from Section 5.1.2 that D_s is the set of documents returned by search service s and V_s is the set of viewed

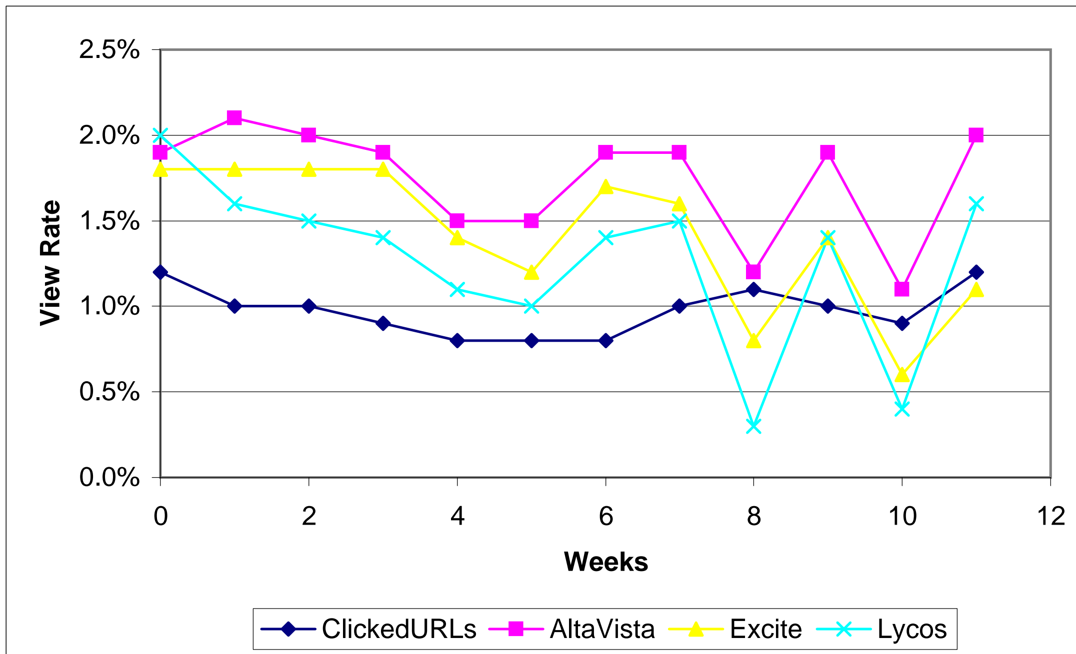


Figure 6.6: View Rate for ClickedURLs and 3 representative Web search services.

This graph shows the volatility of global Web search services in terms of their ability to produce documents that users follow over time, and the relative consistency of ClickedURLs. It also indicates that even with the inherent volatility, the other services do have a greater View Rate, indicating that ClickedURLs is unable to produce results that users follow on many queries.

documents from search service s. The Unique Contribution of search service s can be defined as:

$$UC_s = \frac{\left|V_s - \bigcup_{i \neq s} V_i\right|}{\left|\bigcup_i V_i\right|} \tag{6.9}$$

The Unique Contribution strongly correlates with the number of documents returned by each service, so we also measured the *Document Contribution* of a service. The Document Contribution is the number of documents a service returned divided by the total number of documents returned, as defined by the following equation:

$$DC_s = \frac{|D_s|}{\left|\bigcup_i D_i\right|} \tag{6.10}$$

Finally, we measured the Unique Document Percentage, defined by Equation 5.1 in Section 5.1.2, for each service. For this experiment, there were 84,240 documents viewed and 10,303,553 documents returned. Table 6.3.2 summarizes our data.

AltaVista had the greatest single Unique Contribution of all global web search services used. It also had the greatest Document Contribution and *UDP*. Excite and Lycos were more representative of the other search services.

ClickedURLs had a very modest unique contribution. However, it also had a very small Document Contribution, indicating that it is not yet returning many results on all queries. Roughly half the viewed results returned by ClickedURLs were unique, indicating that many of the results ClickedURLs contributed were unobtainable using HuskySearch without the CIE auxiliaries.

Re-ranking effect of CIE

We also evaluated ClickedURLs for its ability to increase the rank of documents that were viewed by the user. We were especially interested in the non-unique viewed documents returned by ClickedURLs to determine if their ranking improved.

To determine the re-ranking effect, we measured the *Average Rank* and *Median Rank* of documents viewed by the user. Recall that n_s is the number of documents

Table 6.4: Additional viewed documents contributed by ClickedURLs and three representative services.

Service	Unique Contrib	Document Contrib	UDP
AltaVista	22.68%	13.07%	73.22%
Excite	8.33%	10.38%	43.61%
Lycos	10.40%	12.94%	52.56%
ClickedURLs	1.89%	3.87%	48.75%

As shown, the Unique Contribution, or number of documents viewed by users contributed only by the one service, is very modest from ClickedURLs as compared to the rest. However, this correlates to its Document Contribution, or total number of documents returned by the service. The Unique Document Percentage, or UDP, was unique, was in line with the other services. This suggests that ClickedURLs is providing additional information of interest to users, and while its numbers are proportionally in line with the other services, it is still relatively small.

returned on a given query by s, and v_s the number of documents viewed. Let $C_s = [c_1, c_2, ...c_{v_s}]$ be the list of documents returned by search service s viewed by the user. Let $h_s(d)$ be the rank of document d in the results of search service s, ranging from $[0 \ldots n_s]$ with 0 being the top rank. The Average Rank of a search service s is then defined as the average rank of viewed documents:

$$\text{Average Rank of } s = \sum_{i=1}^{v_s} \frac{h(c_i)}{n_s} \tag{6.11}$$

The Median Rank is similarly defined as the median rank of viewed documents:

$$\text{Median Rank of } s = h(c_{\frac{v_s}{2}}) \tag{6.12}$$

We compared the Average Rank and Median Rank of all viewed documents contributed by ClickedURLs in addition to the representative search services. Table 6.3.2 shows our findings.

There is little difference in the Average Rank among ClickedURLs and the three services. However, there is a significant distinction in the Median Rank. ClickedURLs' Median Rank is half that of the next best service, Excite, and a little more than half of Lycos. There is a slight bias against AltaVista in that AltaVista returns a more unique results than Excite, Lycos, and ClickedURLs. Viewed results returned either uniquely or in part by ClickedURLs do suggest that ClickedURLs re-ranking effect does present previously viewed information higher on the list. Half of the documents returned by ClickedURLs that were viewed by users were ranked 0, 1, or 2. In comparison, half of the documents returned by Excite that were viewed were ranked from 0 to 4, the documents from Lycos were ranked from 0 to 5, and the documents from AltaVista were ranked from 0 to 7. This indicates that at least half of the time users are not looking through as many results before clicking on references returned by ClickedURLs.

Table 6.5: Average Rank, Median Rank, and standard deviation for viewed documents.

Service	Average Rank	Median Rank	Std Dev
AltaVista	16.147	**7**	24.652
Excite	15.599	**4**	26.740
Lycos	15.333	**5**	25.804
ClickedURLs	15.682	**2**	32.180

This table shows the Average Rank and Median Rank, as well as the standard deviation, for viewed documents, with 0 being the rank of the first entry in a ranked relevancy list. ClickedURLs has a significantly lower Median Rank than the other three, indicating users are not looking through as many results before clicking on references returned by ClickedURLs.

Performance over time for CIE systems on a large scale

One question concerning a CIE system on a general Web service is whether or not enough people are looking for similar information to make it worthwhile. Our current numbers, especially our figures regarding the View Rate of ClickedURLs, indicate that 84,240 viewed URLs over twelve weeks does not make a significant contribution to the results viewed by the user.

Since it was not feasible for us to run the CIE system on a large-scale web system, we estimated an upper bound on how well an ideal CIE system might perform over time. We define an ideal CIE system as one that for every document a user views, the ideal CIE system returns that document if a previous user viewed it. Naturally, this ideal CIE system will only be able to provide benefit to a user when the user viewed documents that other users previously viewed. Thus, an upper bound on an

ideal CIE system's performance is the probability that a URL that a user viewed had been seen previously.

To calculate this estimate, we obtained a week's worth of logs from the commercial MetaCrawler service [44] containing URLs that users had followed as a result of a search. This came to roughly 7.2 million documents. To estimate the upper bound, we sorted the 7.2 million documents according to the time they were viewed by a user, and then divided them into 72 partitions $p_0 \ldots p_{71}$, each containing 100,000 documents We then calculated the Cumulative Overlap Percentage for each partition p_i, or COP_i, which is the percent of documents in p_i that were also in $p_0 \ldots p_{i-1}$, defined by:

$$COP_i = \frac{\left| p_i \cap \bigcup_{j=0}^{i-1} p_j \right|}{|p_i|} \tag{6.13}$$

The plot of COP for each partition is found in Figure 6.7. As shown, after 1 million documents have been seen, roughly 30% of the URLs in the following 100,000 will have previously been seen. After 3 million it rises to about 40%, and after 6 million 50%. The log analysis presented in this chapter contained only 84,240 viewed results, which means we should have been able to enhance fewer than than 5% of the incoming queries. In reality, only 0.22% of the final 1,000 documents were present in the initial 83,240, which is consistent with our numbers.

Although we were unable to test if ClickedURLs or any of the CIE prototypes we developed would be effective at a large scale, this experiment does indicate that some CIE system would be applicable for over half the queries of a general purpose Web search system. Thus, we believe CIE is not just applicable to small domains, but could provide benefit to a large scale, million-user Web search service.

6.4 Summary

We presented Collaborative Index Enhancement, a model for enhancing a searchable index based on the experience of previous users. The key idea behind CIE is to take

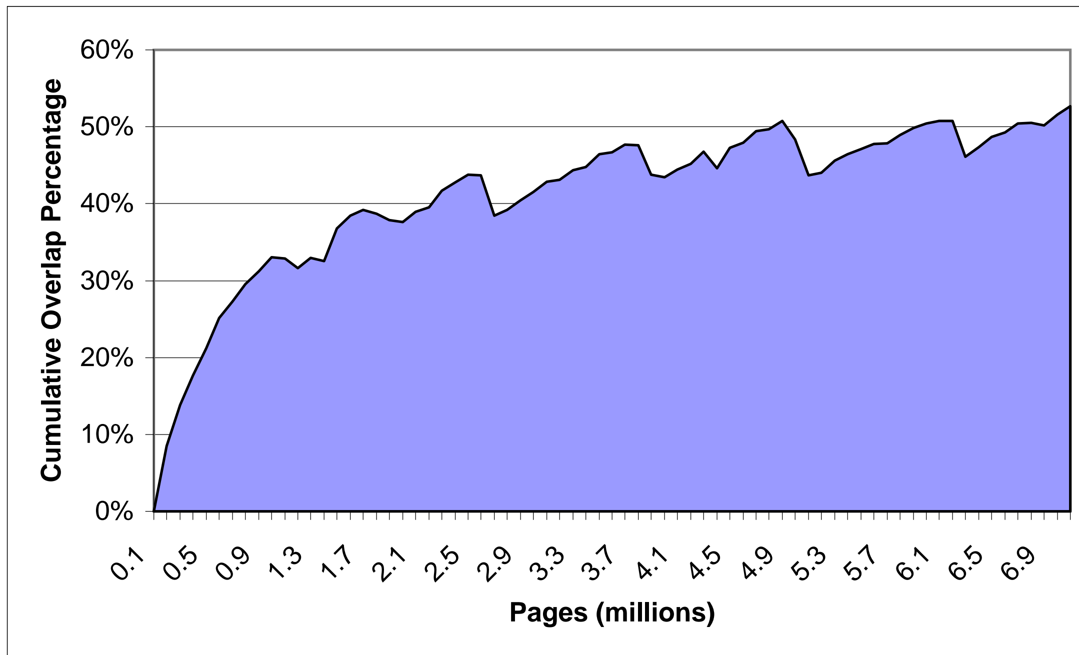


Figure 6.7: Cumulative Overlap Percentage through 7.2 million documents.

This chart shows that after 1 million documents have been seen, about 30% of the next 100,000 will have been seen before. After 6 million, about 50% of the next 100,000 will have been seen previously.

the results document from a query and feed it back into the source index or indices in some manner. We demonstrated a prototype system based on the HuskySearch search service that implements CIE using auxiliary search indices. This implementation allows us to use and experiment with several different CIE methods at once, without the need to modify or even control the original Web indices we use.

In order to validate that CIE is a useful addition to HuskySearch, we conducted a series of experiments based on a user study and log analyses. Our user study indicates that CIE is beneficial and suggests that CIE aided users in answering harder questions as well as in speeding up their answering of easier questions. In one case the time

required to answer a question was decreased by over 50%, and in another case 27% more users were able to answer a hard question correctly. The results also suggest that CIE users were also better able to correctly judge the quality of their answer for harder questions.

In our log analyses, we focused on ClickedURLs, the most promising of the four auxiliaries. We showed that it had a consistent View Rate over time, rather than a volatile one similar to traditional Web search services, though ClickedURLs' View Rate was generally lower than the traditional Web search services. We showed that roughly half of the viewed documents returned by ClickedURLs were unique. We showed that previously viewed documents returned by ClickedURLs had an improved Median Rank over those not returned by ClickedURLs. Finally, even though the overall contribution of CIE on HuskySearch was rather modest, we demonstrated that on a large scale system, after a short period of time over 50% of the documents viewed had been viewed in a previous session. This shows that there is potential for some CIE system to be of benefit in a large scale system.

These results suggest that CIE is a useful addition to HuskySearch, and promises to provide additional benefit over a long duration. To our knowledge, this is the first experimental evaluation of any indirect collaboration method applied to the Web.

Selecting documents for an index is now almost entirely automatic for large-scale systems. As large-scale systems continue to grow, it becomes more difficult to separate relevant documents from the irrelevant ones. CIE is one method that can help alleviate this problem by letting users transparently contribute information that can help other users find relevant documents quickly.

Chapter 7

CONCLUSIONS AND FUTURE WORK

7.1 Summary

The Web Search Problem is the problem of locating all available Web documents which are relevant to a given query. Web search services have addressed this problem through the collection of Web documents via a spider and the subsequent indexing of those documents. This technique effectively enables millions of users to find information on the Web. However, due to the real-world constraints, spider-based search services are unable to provide a complete solution to the Web Search Problem. In our experiments, as well as experiments performed by other groups, no single Web search service provides a comprehensive Web search. Meta-search provides a better solution to the Web Search Problem by combining multiple search services and thus providing a more comprehensive search.

This thesis investigated meta-search to answer four questions: can meta-search be implemented in a practical manner? Does meta-search contribute significantly to comprehensive Web search? Is it likely that meta-search will continue to contribute significantly to comprehensive Web search? Can results provided by meta-search improve over time through user interaction? We answered these questions by designing and implementing two meta-search services, MetaCrawler and HuskySearch, and conducting experiments to ascertain their capabilities.

7.1.1 Practical implementation of meta-search

We designed and implemented MetaCrawler with two goals in mind. The first goal was to demonstrate that meta-search could be implemented in a manner such that average Web users would take advantage of meta-search's benefits. The second goal was to demonstrate that a meta-search service could be provided and maintained with limited resources. To address these goals, we developed an architecture for implementing a meta-engine that is expandable, has a low maintenance cost, has a fast response time, is scalable, and is portable.

In implementing MetaCrawler using our architecture, we were faced with several challenges. First and foremost, MetaCrawler needed to return results to users quickly. MetaCrawler also needed to consume as few system resources as possible in order to maximize the utilization of a single server handling multiple queries. Therefore, we implemented MetaCrawler's retrieval engine using non-blocking I/O and a network of finite state machines. This enabled a *single* MetaCrawler process to simultaneously retrieve 4,093 documents from the Web, which is a limit imposed by the operating system. In contrast, the most common implementation of parallel Web retrievals is using threads. Using threads, we were only able to retrieve a maximum of 1,027 Web documents simultaneously across *all* processes.

Users were also attuned to the quality of results that contemporary Web search services returned. Therefore, MetaCrawler needed to collate the documents from heterogeneous services in a reasonable manner, ensuring that there were no obvious errors, such as the inclusion of duplicate documents. Collating results from heterogeneous search services is not trivial. We developed the Normalize-Distribute-Sum algorithm to collate documents from heterogeneous services given limited information about each document. We also developed two heuristics, the Redirect Heuristic and the Mirror Heuristic, to identify duplicate documents.

MetaCrawler's architecture facilitated both easy expansion and maintenance. This

was necessary because MetaCrawler was maintained by graduate students with limited time. The architecture's modular nature allows the straightforward incorporation of extensions to MetaCrawler, as well as allowing applications built on top of MetaCrawler, such as Grouper and Ahoy!. Furthermore, by isolating the functionalities require the most maintenance into their own modules, maintenance is both straightforward and quickly accomplished.

7.1.2 Significant contribution of meta-search

In Chapter 5 we determined the contribution of meta-search through the evaluation of MetaCrawler and the Web search services it uses through Inference of User Value through Real-world Data. We did this by observing user queries and by logging the documents that each search service returned, as well as which documents users followed. We then analyzed this information.

We demonstrated in two separate studies that each Web search service produces a largely unique set of results for a given query. Furthermore, each of the Web search services provide documents that users view. This was also independently confirmed through several other studies. We then demonstrated that each Web search service contributed a significant percentage of the total results returned by all search services. Furthermore, we demonstrated that each Web search service contributed a significant percentage of the total viewed documents. Therefore, combining multiple search services does provide a significantly more comprehensive search of the Web.

7.1.3 Longevity of meta-search

Having determined that combining multiple Web search services provides a significantly more comprehensive Web search, we then demonstrated that meta-search will likely continue to provide a significant benefit in the future. We first extrapolated an upper bound on the growth of the largest Web search services. Then we constructed two trends based on statistical estimates of the size and growth of the Web. We

observed that even the lower estimate of Web growth is larger than the upper bound on growth of the Web search services.

The upper bound on the growth of the Web search services is a flat rate equal to their largest single-month improvement. Historical data shows that Web search services increase the size of their indices at irregular intervals. This is because the indices are at the limit of the available resources, and quickly consume new resources as they become available. Since the cost of adding storage and servers is not trivial, resources are added intermittently.

One trend showing Web growth is based on the interpolation of three estimates on the size of the Web. An estimate of the size of the Web is calculated by estimating the probability that a document in index B was in index A and B, e.g. $P(A \cap B|B)$. The size of the Web is then estimated by dividing the size of index A by this probability. A trend based on three estimates calculated in this manner indicates the Web doubles every nine months. The other trend showing Web growth is based on the growth of Web servers. The number of available Web servers has been doubling every twelve months.

The lower estimate of Web growth is that the Web doubles every twelve months. This is greater than the upper bound on the growth of Web search service indices. Furthermore, based on the estimates of the size of the Web, even if the top three Web search service indices were completely disjoint, their combined total would not equal that of the size of the Web at the time of this writing. Thus, meta-search will continue to provide significant benefit in the future.

7.1.4 Stable search

We observed that eight of the nine major Web search services produced unstable results. We observed that over a third of the documents returned in the Top 10 results of five out of nine services were absent from the results of the same query submitted at a later time, only to reappear in the results of a subsequent submission.

Three of the remaining four services also exhibited this behavior to a lesser degree.

We presented Collaborative Index Enhancement, a model that generalizes enhancement of an index through indirect collaboration. We showcased four different instantiations of CIE. We evaluated the merits of these four instantiations using both a user study and a log analysis conducted in a manner similar to our evaluation of Web search services. Two of these instantiations, ReturnedURLs and ClickedURLs, provide a means of addressing the problem of instability in the Web search services. The other two methods provide a query history and a convenient form of query expansion.

While ClickedURLs provided modest improvement in HuskySearch, we observed through logs of the commercial MetaCrawler service that after roughly six million documents have been viewed by users, half of the documents viewed have been viewed previously. This indicates that at large scale there is a strong potential for user interaction to improve the results of searching over time.

7.2 Future work

7.2.1 Qualitative analysis

In Section 3.1.4 we described two claims that we could make of MetaCrawler. The first claim was that because MetaCrawler retrieved the top k documents from each search service and then collated them, the top k documents MetaCrawler returned would contain higher quality documents than the top k from any single search service. The second claim was that on average a user could locate relevant information faster by using MetaCrawler instead of using each search service directly. While practical experience indicates that these claims are true, we did not evaluate these claims scientifically. Obvious future work would be to evaluate these claims, determining how much better, if at all, results from meta-search are to any search service, and to determine how much faster, if at all, users can locate information using meta-search. Such work would also likely involve significant exploration into different collation and

ranking algorithms, similar to the work done on traditional Information Retrieval search engines [34].

7.2.2 Improving inference of user value

Our methodology to measure the performance of search services, Inference of User Value through Real-world Data, depends on some measurable quality from which we can infer user value. As we discussed in Section 5.1.1, using viewed documents as a measure of usefulness is only upper bound. A tighter measurement would allow a much more insightful evaluation, both in terms of evaluating the usefulness of each search service as well as in constructing and evaluating Collaborative Index Enhancement auxiliaries. One method of obtaining a better measure would be to ask users after they have read a document whether it is relevant to their query. Asking users in the right way is an issue in itself, but that has been reasonably well studied in Information Retrieval literature [39]. Assuming users give actual relevance information, various models could be constructed to give a better measure on whether a document is actually relevant to a query. In particular, various document attributes may affect its likelihood of being relevant, such as its rank, the amount of time a user views the document, the number of documents the user has viewed in the results prior to the current document, and so on.

7.2.3 Query routing

One area of meta-search which we did not explore in any great depth was scaling MetaCrawler and HuskySearch to handle a large number of Web search services. Most Web search services are either topic-specific or region-specific, such as the CBS SportsLine search [22] or Jubii, a search service for Denmark [54]. While integrating additional search services into the Harness module is not difficult, MetaCrawler's broadcast mechanism could put an undue load on the specialty services it accessed by querying them with off-topic queries.

One technology that addresses the selection of search services is query routing. There has already been significant work in the field of query routing. Sheldon *et al.* experimented with query routing using Wide Area Information Servers, or WAIS [55] search platform [98]. Their system, Discover, creates content labels for each WAIS site. A feature of the WAIS system is that the keys of the searchable index are retrievable; thus the content labels are based on the keys of the remote index. To route queries, Discover simply determines which content labels contain the query terms via an inverted index.

The GLOSS system [46] works in a similar fashion to Discover, although it uses a probabilistic model to route queries. Rather than describing an index by its keys, GLOSS creates a histogram of the keys in an index. GLOSS then uses a probabilistic model with the histogram to rank the indices based on how many documents they are likely to return given the appropriate query.

Atsushi Sugiura is currently conducting an exploration into query routing using the CIE framework [104]. Several documents retrieved from the available search services are inserted into a CIE auxiliary. To route queries, the query is first issued to the CIE auxiliary. The search services that originally returned the documents found in the auxiliary are then queried directly.

7.2.4 Alternative instances of CIE

Our CIE system is still in early development, and there is a great deal of future work that can be done. More sophisticated CIE indices should be explored and evaluated. One direction worth pursuing is to expand the ReturnedURLs and ClickedURLs to include "close pages," such as all pages three links away from a page a user clicked on. Also, an improved user interface might make the SuccessfulResultsPages and ResultsPages auxiliaries more appealing and useful to users.

While we have the infrastructure to address the scaling issues that will arise after a million or more queries have been submitted, evaluation on the actual scaling

mechanisms and parameters still needs to be done.

The question of the portability of CIE is still open. It is unclear if CIE would be of benefit in a traditional IR environment. Effort should be made to explore using non-Web search systems as the basis for CIE.

Finally, more exhaustive testing of CIE auxiliaries are also in order. The results presented in this chapter are promising, but the systems do need further evaluation. Of particular note would be to investigate whether the CIE documents *not* viewed by users significantly impede users from finding information by cluttering the results with irrelevant documents.

7.2.5 Beyond HTML

This thesis focused on searching for relevant Web HTML documents. However, while the majority of Web content is comprised of HTML documents, there are numerous other document types available. In particular, there are a wide variety of text documents written in formats besides HTML, such as Postscript, Portable Document Format (PDF), Rich Text Format (RTF), and so on. There has been some work on including alternative formats. For example, the Harvest system includes a component that converts known document types into a Summary Object Interchange Format, or SOIF, document [14]. However, properly parsing text documents written in other formats than HTML is not that challenging.

A task that is challenging is integrating non-text formats into a single searchable interface. For example, Lycos has incorporated a search feature which allows users to search for audio files encoded in the MP3 audio file format. However, this feature requires its own separate search form [70]. Other search services also have indices for other media, such as pictures and videos. An additional user interface challenge is to allow for the user to query using more than just keywords. For example, many image search engines locate images similar to other images [10].

Even within HTML, different types of files exist. Treating these different files types

as different files rather than plain HTML files could lead to significant improvement in the quality of search results. For example, while it would not be difficult to include a USENET search service, such as DejaNews [29], in HuskySearch, it is not clear that the incorporation of USENET news articles and normal Web documents in a single list would be understood by average Web users.

7.2.6 Information integration

The Information Integration Problem is the problem of finding information available in whole or part from a variety of databases. This is a similar problem to the Web Search Problem. However, rather than attempting to find existing Web documents that contain relevant information, the Information Integration Problem focuses on locating information that matches a specific query through multiple databases.

The Tukwila system is one such project exploring the Information Integration Problem [53]. In the Tukwila system, a *mediated schema* is constructed for a given domain. A mediated schema is a list of attributes that describe individual entities in the domain. Users can then issue queries over the mediated schema. Queries in this context are a list of possible values for some or all of the fields in the mediated schema.

The Tukwila system integrates a number of online databases by transforming the output of a database into a format that can be mapped to attributes in the mediated schema. Currently, a human expert needs to construct this mapping, but there is ongoing work to reduce the need for a human expert through the use of machine learning [32]. The main problem in this area is in mapping attributes labeled differently, but that have the same semantic meaning. Tukwila also requires that the online database return structured data, or data presented in a static format. There is also ongoing work to extend Tukwila to use semi-structured data. Many online databases are moving towards outputting their results using a self-describing semi-structured data format, such as the Extensible Markup Language, XML [18]. The

advantage in using self-describing semi-structured data is that information can be easily exchanged without the need to construct custom wrappers. The difficulty with using self-describing data is in understanding the description, which is not guaranteed to adhere to any standard.

7.2.7 Outside the box

The goal of the Internet Softbot was for users to say *what* they wanted, and the Softbot would figure out *where* to get it and *how* to get it [36]. MetaCrawler explored this concept in the Web domain by enabling users to describe the information they wanted based on keywords. However, while retrieving all of the relevant information to a given query can address a user's immediate information need, it is most likely only a *partial* solution to the user's *actual* problem. A keyword query may convey what information a user wants, but does not convey why a user wants that information nor what the user intends to do with it. For example, consider two people, one living in Salt Lake City, the other in Seattle. They may both issue the query "Utah Jazz schedule" looking for a document that contains the game schedule of the Utah Jazz basketball team. However, the person who lives in Salt Lake City may be looking for the next Jazz game in Utah, whereas the person in Seattle may be looking for the date when the Jazz play in Seattle.

There has been some work on enabling users to better describe what they want. Natural language processing can be applied to interpret a natural language question that better describes what the user wants. For example, the commercial service Ask Jeeves! enables users to enter a natural language question, such as "When is the next Utah Jazz home game?" [4]. Unfortunately, it does not immediately return the answer, but rather returns a number of other queries that may lead to the answer. One of the example queries Ask Jeeves! returned in response to the above question is, "Where can I locate current schedules for the NBA team Utah Jazz?" A more subtle approach is the implicit query project [25]. Rather than requiring a user to

use a search service to find information, the application a user is currently interacting with issues an implicit query based on heuristics of when a user requires additional information. The implicit query is constructed from the context of the application. In this way, the application can provide the user with needed information when it is needed.

In addition to the problem of determining what the user wants, there are many open problems that must be addressed in order to construct a general purpose service where a user can describe what he or she wants and have the service return useful and relevant information. The Web Search Problem and Information Integration Problem are just two problems which need to be addressed in order to provide the user with the information they desire. Meta-search certainly provides a partial solution to the Web Search Problem, but the problem is far from solved.

BIBLIOGRAPHY

[1] David Aha. David Aha's List of Machine Learning and Case-Based Reasoning Home Pages, 1996.
http://www.aic.nrl.navy.mil/~aha/people.html.

[2] Apple Computer, Inc. Sherlock, 1998.
http://www.apple.com/sherlock/.

[3] Apple Donuts. Apple Donuts Sherlock Plugins, 1999.
http://www.apple-donuts.com.

[4] Ask Jeeves, Inc. Ask Jeeves!, 1999.
http://www.ask.com.

[5] Gauruv Banga and Jeffrey Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference (NO 98)*, New Orleans, June 1998. USENIX.

[6] J. M. Barrie and D. E. Presti. The World Wide Web as an Instructional Tool. *Science*, 274(5286):371, October 1996.

[7] N. J. Belkin, P. Kantor, C. Cool, and R. Quatrain. Combining Evidence for Information Retrieval. In Donna Harman, editor, *TREC-2, Proceedings of the Second Text REtrieval Conference.* NIST Special Publication 500-215, 1993.

[8] N. J. Belkin, P. Kantor, E. A. Fox, and J. A. Shaw. Combining the Evidence of Multiple Query Representations for Information Retrieval. *Information Processing & Management*, 31(3):431–448, 1995.

[9] Nicholas J. Belkin and W. Bruce Croft. Retrieval Techniques. In Martha E. Williams, editor, *ARIST*, volume 1, chapter 4, pages 109–145. Information Today, Inc., 1987.

[10] Andrew Berman. *Efficient Content-Based Retrieval of Images using Triangle-Inequality-Based Algorithms*. PhD thesis, University of Washington, 1999.

[11] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the 1995 Winter USENIX Conference*, New Orleans, LA, Jan 1995. USENIX. `http://www.cs.washington.edu/homes/speed/papers/tron/tron.ps.gz`.

[12] Krishna Bharat and Andrei Broder. A Technique for Measuring the Relative Size and Overlap of Public Web Search Engines. In *Proceedings of the 7th World Wide Web Conference*, Brisbane, Australia, August 1998.

[13] Krishna Bharat and Andrei Broder. Measuring the Web. `http://www.research.digital.com/SRC/whatsnew/sem.html`, 1998.

[14] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. Harvest: A Scalable, Customizable Discovery and Access System. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, Colorado, March 1995. `http://harvest.cs.colorado.edu/harvest/papers.html`.

[15] Christin Boyd. Interactive query refinement tool for the huskysearch web search service. Senior Honors Thesis. University of Washington, Department of Computer Science and Engineering, 1997.

[16] David Brake. Lost in cyberspace. *New Scientist*, June 1997. `http://www.newscientist.com/keysites/networld/lost.html`.

[17] T. L. Brauen. Document Vector Modifications in the SMART Retrieval System. In *The SMART Retrieval System: Experiments in Automatic Document Processing*. Prentice-Hall, Englewood Cliffs, NJ:, 1971.

[18] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. W3C Recommendation, February 1998. `http://www.w3.org/TR/REC-xml`.

[19] Eric Brewer. The HotBot Search Engine [talk only]. In *Proceedings of the American Library Association 1997 Annual Conference*, San Francisco, CA, June 1997.

[20] Andrei Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1998. `ftp://ftp.digital.com/pub/DEC/SRC/publications/broder/-positano-final-wpnums.pdf`.

[21] James Callan, Zhihong Lu, and Bruce Croft. Searching Distributed Collections with Inference Networks. In *Proceedings of the 1995 ACM SIGIR Conference*, Seattle, WA, July 1995. ACM Press.

[22] CBS Inc. CBS SportsLine Search Center, 1999. `http://cbs.sportsline.com/u/spmenu.htm`.

[23] J. Cohen and S. Aggarwal. General event notification architecture base. Internet Draft, 1998. `http://search.ietf.org/internet-drafts/draft-cohen-gena-p-base-01.txt`.

[24] Compaq Corporation. About AltaVista, 1999.
http://www.altavista.com/av/content/about_our_technology_2.htm-#equipment.

[25] Mary Czerwinski, Susan Dumais, George Robertson, Susan Dziadosz, Scott Tiernan, and Maarten van Dantzich. Visualizing Implicit Queries for Information Management and Retrieval. In *Proceedings of the 1999 ACM SIGCHI Conference on Human Factors in Computing Systems*, Pittsburgh, PA, 1999. ACM Press.

[26] Mark Day. Simple general awareness protocol. Internet Draft, 1998.
http://search.ietf.org/internet-drafts/draft-day-sgap-01.txt.

[27] P. De Bra, G. J. Houben, Y. Kornatzky, and R. Post. Information Retrieval in Distributed Hypertexts. In *RIAO '94: Intelligent Multimedia Retrieval Systems and Management*, New York, NY, October 1994.

[28] Paul De Bra and R. D. J. Post. Searching for Arbitrary Information in the World Wide Web: the Fish-Search for Mosaic. In *Proceedings of the 2nd World Wide Web Conference*, Chicago, IL USA, October 1994.
http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/debra/-article.html.

[29] DejaNews Research Service. DejaNews Home Page, 1996.
http://www.dejanews.com.

[30] Digital Equipment Corporation. Alta Vista Home Page, 1996.
http://www.altavista.digital.com.

[31] Direct Hit Inc. Direct Hit Homepage, 1998.
http://www.directhit.com.

[32] AnHai Doan. Personal Communication, May 1999.

[33] Derek L. Eager and John Zahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Trans. on Computer Sys*, 11(1):1–32, February 1993.

[34] Efthimis Efthimiadis. *Interactive Query Expansion and Relevance Feedback for Document Retrieval Systems*. PhD thesis, City University, London, 1992.

[35] EINet. Galaxy Home Page, 1995.
`http://galaxy.einet.net/galaxy.html`.

[36] O. Etzioni and D. Weld. A softbot-based interface to the internet. *CACM*, 37(7):72–76, July 1994.
`http://www.cs.washington.edu/research/softbots`.

[37] Excite, Inc. Excite Home Page, 1995.
`http://www.excite.com`.

[38] Excite Inc. Excite reports fourth quarter and total year financial results. Press Release, January 1999.
`http://www.corporate-ir.net/ireye/ir_site.zhtml?-`
`ticker=XCIT&script=410&layout=7&item_id=17946`.

[39] Raya Fidel and Michael Crandall. Users' Perception of the Performance of a Filtering System. In *Proceedings of the 1997 ACM SIGIR Conference*, pages 198–205, Philadelphia, PA, July 1997. ACM Press.

[40] David Filo and Jerry Yang. Yahoo Home Page, 1995.
`http://www.yahoo.com`.

[41] FireFly Networks Inc. The FireFly Home Page, 1996.
http://www.firefly.net.

[42] Larry Fitzpatrick and Mei Dent. Automatic Feedback Using Past Queries: Social Searching? In *Proceedings of the 1997 ACM SIGIR Conference*, pages 306–312, Philadelphia, PA, July 1997. ACM Press.

[43] Susan Gauch, Guijun Wang, and Mario Gomez. ProFusion: Intelligent Fusion from Multiple, Distributed Search Engines. *Journal of Universal Computing*, 2(9), Sept 1996.
http://www.ittc.ukans.edu/~sgauch/papers/JUCS96.ps.

[44] Go2Net, Inc. MetaCrawler Home Page, 1997.
http://www.metacrawler.com.

[45] Google, Inc. The Google Home Page, 1999.
http://www.google.com.

[46] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The Effectiveness of GlOSS for the Text Database Discovery Problem. In *Proceedings of the 1994 ACM SIGMOD Conference*, pages 126–137, Minneapolis, MN, May 1994. ACM Press.
ftp://db.stanford.edu/pub/gravano/1994/stan.cs.tn.93.002.sigmod94.ps.

[47] Adele Howe and Daniel Dreilinger. SavvySearch: A Meta-Search Engine that Learns Which Search Engines to Query. *AI Magazine*, 18(2), summer 1997.
http://daniel.www.media.mit.edu/people/daniel/papers/ss-aimag.ps.gz.

[48] HTTP Working Group. Hypertext transfer protocol – http/1.1. Internet Draft, November 1998.
http://www.w3.org/Protocols/History.html#Rev06.

[49] InfoSeek Corporation. InfoSeek Home Page, 1995.
http://www.infoseek.com.

[50] Inktomi, Inc. HotBot Home Page, 1996.
http://www.hotbot.com.

[51] Inktomi, Inc. Inktomi Home Page, 1996.
http://www.inktomi.com.

[52] Internet Archive, Inc. The Internet Archive, 1998.
http://www.archive.org.

[53] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *Proceedings of SIGMOD Conference on Management of Data*, Philadelphia, PA, USA, June 1999.

[54] Jubii Ltd. Jubii Homepage, 1999.
http://www.jubii.dk.

[55] B. Kahle and A. Medlar. An information system for corporate users: Wide Area Information Servers. Technical Report Technical Report TMC-199, Thinking Machines Inc., April 1991. Version 3.

[56] Colleen Kehoe and Jim Pitkow. *GVU's Tenth WWW User Survey Report*. Office of Technology Licensing, Georgia Tech Research Corporation, 1999.
http://www.gvu.gatech.edu/user_surveys/survey-1998-10/.

[57] Rohit Khare and Adam Rifkin. Scenarios for an internet-scale event notification service (isens). Internet Draft, 1998.

http://search.ietf.org/internet-drafts/draft-khare-notify-scenarios--01.txt.

[58] J. Konstan, B. Miller, D. Maltz, J. Herlocker, L. Gordon, and J. Riedl. GroupLens: Applying Collaborative Filtering to Usenet News. *Communications of the ACM*, 40(3):77–87, 1997. http://www.acm.org/pubs/citations/journals/cacm/1997-40-3/p77-konstan.

[59] Martijn Koster. Robots in the Web: threat or treat? *ConneXions*, 9(4), April 1995.

[60] N. Kushmerick. *Wrapper Induction for Information Extraction.* PhD thesis, Univ. of Washington, 1997.

[61] N. Kushmerick. Regression testing for wrapper maintenance. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, Florida, July 1999. Menlo Park, Calif.: AAAI Press.

[62] Brian A. LaMacchia. The Internet Fish Construction Kit. In *Proceedings of the 6th World Wide Web Conference*, pages 277–288, Santa Clara, CA, April 1997.

[63] Tessa Lau, Oren Etzioni, and Daniel S. Weld. Privacy interfaces for information management. Technical Report UW-CSE-98-02-01, University of Washington, March 1998.

[64] Tessa Lau and Eric Horvitz. Patters of search: Analyzing and modeling web query refinement. In *Proceedings of the 1999 User Modelling Conference*, 1999. To appear.

[65] Gregory Lauckhart. Work done as research programmer. University of Washington, Department of Computer Science and Engineering, 1996.

[66] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280:98–100, April 1998.

[67] H. Lieberman. Letizia: An agent that assists web browsing. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 924–929, 1995.

[68] Jong-Gyun Lim. Using Coollists to Index HTML Documents in the Web . In *Proceedings of the 2nd World Wide Web Conference*, Chicago, IL USA, October 1994.
`http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/lim/-coollist.html`.

[69] Lycos Inc. Lycos Home Page, 1999.
`http://www.lycos.com`.

[70] Lycos Inc. Lycos MP3 Search, 1999.
`http://mp3.lycos.com`.

[71] Michael L. Mauldin and John R. R. Leavitt. Web Agent Related Research at the Center for Machine Translation. In *Proceedings of SIGNIDR V*, McLean, Virginia, August 1994.

[72] Max Metral. Helpful Online Music Recommendation Service, 1995.
`http://rg.media.mit.edu/ringo/ringo.html`.

[73] Henrik Frystyk Neilsen. Libwww - the w3c protocol library, 1995.
`http://www.w3.org/Library/`.

[74] Netcraft. Netcraft Web Server Survey, 1999.
`http://www.netcraft.com/survey`.

[75] Netscape, Inc. An Exploration of Dynamic Documents, 1995.
http://home.netscape.com/assist/net_sites/pushpull.html.

[76] Netscape Inc. Introduction to SSL, 1996.
http://developer.netscape.com/docs/manuals/security/sslin/index.htm.

[77] Open Text, Inc. Open Text Web Index Home Page, 1995.
http://www.opentext.com:8080/omw/f-omw.html.

[78] M. Perkowitz and O. Etzioni. Category translation: Learning to understand information on the Internet. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 930–6, 1995.

[79] Brian Pinkerton. Finding What People Want: Experiences with the WebCrawler. In *Proceedings of the 2nd World Wide Web Conference*, Chicago, IL USA, October 1994.
http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/pinkerton/-WebCrawler.html.

[80] Brian Pinkerton. WebCrawler Home Page, 1995.
http://webcrawler.com.

[81] Brian Pinkerton. Personal Communication, March 1999.

[82] PlanetSearch Network Inc. PlanetSearch Home Page, 1998.
http://www.planetsearch.com.

[83] Quarterdeck, Inc. WebCompass Home Page, 1996.
http://www.qdeck.com/qdeck/demosoft/webcompass_live.

[84] Vijay V. Raghavan and Hayri Sever. On the Reuse of Past Optimal Queries. In *Proceedings of the 1995 ACM SIGIR Conference*, pages 344–350, Seattle, WA, July 1995. ACM Press.

[85] Eric J. Ray, Deborah S. Ray, and Richard Seltzer. *The AltaVista Search Revolution*. Osborne McGraw-Hill, second edition, 1998.

[86] Surendra Reddy. Requirements for event notification protocol. Internet Draft, 1998.
http://search.ietf.org/internet-drafts/draft-skreddy-enpreq-00.txt.

[87] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[88] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[89] Gerard Salton, editor. *The SMART Retrieval System: Experiments in Automatic Document Processing*. Prentice Hall, Englewood Cliffs, NJ, 1971.

[90] Darren Schack. Senior honors thesis. University of Washington, Department of Computer Science and Engineering, 1996.

[91] Michael F. Schwartz, C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, and Udi Manber. The Harvest Information Discovery and Access System . In *Proceedings of the 2nd World Wide Web Conference*, Chicago, IL USA, October 1994.
http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/-
schwartz.harvest/schwartz.harvest.html.

[92] Erik Selberg and Oren Etzioni. Multi-Service Search and Comparison Using the MetaCrawler. In *Proceedings of the 4th World Wide Web Conference*, pages 195–208, Boston, MA USA, December 1995.
`http://huskysearch.cs.washington.edu/papers/www4/html/Overview.html`.

[93] Erik Selberg and Oren Etzioni. HuskySearch Home Page, 1997.
`http://huskysearch.cs.washington.edu`.

[94] Erik Selberg and Oren Etzioni. The MetaCrawler Architecture for Resource Aggregation on the Web. *IEEE Expert*, 12(1):8–14, January 1997.

[95] Erik Selberg and Oren Etzioni. Experiments in Collaborative Index Enhancement. Technical Report UW-CSE-98-06-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA, June 1998.
`http://www.cs.washington.edu/homes/speed/papers/cqp/cqp.ps`.

[96] J. Shakes, M. Langheinrich, and O. Etzioni. Dynamic reference sifting: a case study in the homepage domain. In *Proc. 6th World Wide Web Conf.*, 1997. See `http://www.cs.washington.edu/research/ahoy`.

[97] U. Shardanand and Pattie Maes. Social Information Filtering: Algorithms for Automating 'Word of Mouth'. In *Proceedings of the CHI-95 Conference*, Denver, CO, May 1995.

[98] Mark A. Sheldon, Andrzej Duda, Ron Weiss, and David K. Gifford. Discover: A Resource Discovery System based on Content Routing. In *Proceedings of the 3rd World Wide Web Conference*, Elsevier, North Holland, April 1995.
`http://www-psrg.lcs.mit.edu/ftpdir/papers/www95.ps`.

[99] Narayanan Shivakumar and Hector Garcia-Molina. Finding near-replicas of documents on the web. In *Proceedings of Workshop on Web Databases (WebDB'98)*,

March 1998.
http://www-db.stanford.edu/~shiva/Pubs/web.ps.

[100] Craig Silverstein, Monika Henzinger, Hannes Marais, and Michael Moricz. Analysis of a Very Large AltaVista Query Log. Technical Report 1998-014, Compaq Systems Research Center, Palo Alto, CA, October 1998.
http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/-src-tn-1998-014.html.

[101] K. Sparck Jones and C. J. van Rijsbergen. Report on the need for and provision of an "ideal" information retrieval test collection. Technical Report 5266, British Library Research and Development Report, Computer Laboratory, University of Cambridge, 1979.

[102] Amanda Spink. Term Relevance Feedback and Query Expansion: Relation to Design. In *Proceedings of the 1994 ACM SIGIR Conference*, pages 81–90, Dublin, Ireland, July 1994. ACM Press.

[103] Louise T. Su. The relevance of recall and precision in user evaluation. *J. American Society of Information Science*, 45(3):207–217, 1994.

[104] Atsushi Sugiura, 1999. Personal Communication.

[105] Danny Sullivan. Search Engine Watch, 1999.
http://www.searchenginewatch.com.

[106] The Daily of the University of Washington. The Online Daily of the University of Washington, 1997.
http://www.thedaily.washington.edu.

[107] The Intelligent Transportation Systems Program. netAddress Book of Transportation Professionals, 1996.
http://dragon.princeton.edu/~dhb/TRANSPORT_NAB/.

[108] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, second edition, 1979.

[109] Ellen Voorhees and Donna Harman, editors. *Information Technology: The Fifth Text REtrieval Conference*, Gathersburg, MD, nov 1996. NIST Special Publication 500-238.
http://trec.nist.gov.

[110] Ellen Voorhees and Donna Harman. Overview of the Sixth Text REtrieval Conference (TREC-6). In Ellen Voorhees and Donna Harman, editors, *TREC-6, Proceedings of the Second Text REtrieval Conference*. NIST Special Publication 500-240, 1997.

[111] Yahoo Inc. Yahoo! reports fourth quarter and 1998 fiscal year end financial results. Press Release, January 1999.
http://www.yahoo.com/docs/pr/release259.html.

[112] O. Zamir and O. Etzioni. A dynamic clustering interface to web search results. In *Proceedings of the Eighth Int. WWW Conference*, 1999.

Appendix A

QUERIES FROM LAWRENCE AND GILES STUDY

1. adaptive access control
2. neighborhood preservation topographic
3. hamiltonian structures
4. right linear grammar
5. pulse width modulation neural
6. unbalanced prior probabilities
7. ranked assignment method
8. internet explorer favourites importing
9. karvel thornber
10. zili liu
11. softmax activation function
12. bose multidimensional system theory
13. gamma mlp
14. dvi2pdf

15. john oliensis

16. rieke spikes exploring neural

17. video watermarking

18. counterpropagation network

19. fat shattering dimension

20. abelson amorphous computing

21. histogram equalization algorithm

22. mixture distance

23. selective attention memory task sequential

24. universal approximation bounds

25. bayesian interpolation

Appendix B

CIE USER SURVEY FORM

HuskySearch CIE Evaluation

Handed out: Tues., Jan. 13th

Due back: Tues., Jan. 20th

In this study, you'll evaluate HuskySearch, an online search tool by Erik Selberg and Oren Etzioni in the CSE department here at UW. You will be evaluating one of two interfaces to the most recent version of HuskySearch. This evaluation is part of ongoing research with HuskySearch; the goal for this evaluation is to determine how well certain new features are providing the intended functionality.

You will attempt to find answers to five (5) different questions using HuskySearch. You are free to search through whatever Web pages you find on the web, but please do not use other search services you may know of nor use any written material, such as encyclopedias or fact books, even to start your query. Depending on your search strategies, some questions may take longer to answer than others, but you shouldn't feel required to spend more than 10 minutes or so on any one question. In addition, you will write a short journal, which describes your expectations and evaluations of the searching. You will also fill out brief questionnaires before and after you answer all 6 questions.

As you attempt to answer each question, please write down the Start Time, and when you find the answer or give up, the End Time. Please record the times in Hour:Minute format (e.g. 8:34 PM), and try to be as close to the actual minute as you can. We will be attempting to determine how long on average it takes for people to answer each question, so these numbers are critical!

For each query you search with HuskySearch, please write down the terms you used, and

circle what you selected in the **Search for** field (The Phrase, All of These Words, etc.) and the type of search you used (Fast, Default, Quality). (If you just hit the return key after entering your search terms, circle Default.) When you then get the results, please try to briefly write down what you are thinking as you look at the results. Also, if the answer isn't apparent from the results listed, please jot down a reason why you think this may be if one is apparent.

When you find the answer, please jot it down in the space provided, mark down the End Time, and circle the proper value for "Correctness of your answer" and "Evaluation of HuskySearch performance."

Please contact Erik Selberg via e-mail at **selberg@cs.washington.edu** if you have any questions or issues.

Thank you for taking the time to complete this evaluation. Your effort and insight into the current operation of HuskySearch will be used to further enhance its ability to provide quality information retrieval on the World Wide Web.

Sincerely,
Erik Selberg and Oren Etzioni

Group 1 Beginning Questionnaire

1. How would you characterize your familiarity with searching the World Wide Web?

   None Passing Familiar Experienced

2. How frequently do you search the World Wide Web?

   Never Occasionally Daily

3. If you have searched the World Wide Web, how advanced is your use of search syntax?

   Don't use advanced syntax Occasionally use it Routinely use it

4. Have you ever used HuskySearch or before?

   Never Occasionally Daily

5. Please list the Web search services (also called search engines) you use frequently:

You are now ready to begin the evaluation.

Please connect your Web browser to:

`http://huskysearch.cs.washington.edu/eval/group1/tutorial.html`

and take a brief tutorial. Once that is finished, please connect your Web browser to:

`http://huskysearch.cs.washington.edu/eval/group1/index.html`

and begin the evaluation. Please use the above link to HuskySearch for all your work during this evaluation; of note, make sure that "Group 1 Interface" is present on the HTML page. If you accidentally get to the standard HuskySearch page (at http://huskysearch.cs.washington.edu/) or a similar page, please open the above page directly.

Group 2 Beginning Questionnaire

1. How would you characterize your familiarity with searching the World Wide Web?

   None   Passing   Familiar   Experienced

2. How frequently do you search the World Wide Web?

   Never   Occasionally   Daily

3. If you have searched the World Wide Web, how advanced is your use of search syntax?

   Don't use advanced syntax   Occasionally use it   Routinely use it

4. Have you ever used HuskySearch or before?

   Never   Occasionally   Daily

5. Please list the Web search services (also called search engines) you use frequently:

You are now ready to begin the evaluation.

Please connect your Web browser to:

`http://huskysearch.cs.washington.edu/eval/group2/tutorial.html`

and take a brief tutorial. Once that is finished, please connect your Web browser to:

`http://huskysearch.cs.washington.edu/eval/group2/index.html`

and begin the evaluation. Please use the above link to HuskySearch for all your work during this evaluation; of note, make sure that "Group 2 Interface" is present on the HTML page. If you accidentally get to the standard HuskySearch page (at http://huskysearch.cs.washington.edu/) or a similar page, please open the above page directly.

Question: Who were the members of the A-Team from the TV show by the same name?

Start Time (HH:MM): ______________ **End Time:** ______________

Answer:

Correctness of your answer: (circle one)	Very sure	Pretty sure	Not sure	Didn't finish
Evaluation of HuskySearch performance:	Excellent	Very Good	OK	Poor

---

1st Query Term(s): ______________________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

2nd Query Term(s): ______________________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

3rd Query Term(s): ______________________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

Question 1: What are the three most recent roles Kevin Spacey has played? Please give role, movie, and date.

Start Time (HH:MM): ____________ **End Time:** ____________

Answer:

Correctness of your answer: (circle one) Very sure Pretty sure Not sure Didn't finish

Evaluation of HuskySearch performance: Excellent Very Good OK Poor

---

1st Query Term(s): ____________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

2nd Query Term(s): ____________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

3rd Query Term(s): ____________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

Question 2: Which Utah ski resort has the highest elevation, and what is it?

Start Time (HH:MM): ____________ **End Time:** ____________

Answer:

Correctness of your answer: (circle one)	Very sure	Pretty sure	Not sure	Didn't finish
Evaluation of HuskySearch performance:	Excellent	Very Good	OK	Poor

---

1st Query Term(s): ____

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

2nd Query Term(s): ____

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

3rd Query Term(s): ____

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

Question 3: Find a picture of a Fraser fir. This must be an actual detailed picture or photograph of a Fraser fir. Pictures of a generic tree or of a fir in the background setting don't count.

Start Time (HH:MM): ____________ **End Time:** ____________

Answer:

Correctness of your answer: (circle one) Very sure Pretty sure Not sure Didn't finish

Evaluation of HuskySearch performance: Excellent Very Good OK Poor

---

1st Query Term(s): ____________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

2nd Query Term(s): ____________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

3rd Query Term(s): ____________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

Question 4: How many members are there in the Canadian parliament?

Start Time (HH:MM): ____________ **End Time:** ____________

Answer:

Correctness of your answer: (circle one) Very sure Pretty sure Not sure Didn't finish

Evaluation of HuskySearch performance: Excellent Very Good OK Poor

---

1st Query Term(s): ________________________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

2nd Query Term(s): ________________________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

3rd Query Term(s): ________________________________

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

Question 5: What was Microsoft's IPO price? (The price they went public at, not their current stock value).

Start Time (HH:MM): ______________ **End Time:** ______________

Answer:

Correctness of your answer: (circle one)	Very sure	Pretty sure	Not sure	Didn't finish
Evaluation of HuskySearch performance:	Excellent	Very Good	OK	Poor

---

1st Query Term(s): ____

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

2nd Query Term(s): ____

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

3rd Query Term(s): ____

Searched for: Phrase All of these words Any of these words Person Pages pointing to

Type of search: Fast Default Quality

Specific comments:

If you didn't get your answer from this search, why do you think you didn't?

Ending Questionnaire

Thank you for taking the time to complete this study! Results should be published by the end of January. Please fill out this final ending questionnaire with your overall evaluation of the system through all six questions. Feel free to also write down any comment or suggestion at the bottom of this page.

Thanks again, Erik Selberg and Oren Etzioni

1. What features of the search engine helped you find your answer?

2. What features inhibited your searching?

3. HuskySearch tends to return a lot of extraneous results for each search. How did the extraneous results affect your searching?

   Extraneous results not a problem

   Annoying, but manageable

   Difficult to find relevant page

4. Comments on HuskySearch performance and interface:

VITA

Erik W. Selberg

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA. 98195-2350
(206) 543-7798
selberg@cs.washington.edu
http://www.cs.washington.edu/homes/selberg

3516 NE 75th St. #10
Seattle, WA. 98115
(206) 517-3039

Career Objectives

Program or engineering management position in an advanced development group or startup.

Technical Interests

Information retrieval, collaborative systems, large scale systems, retrieval and scalability issues of the World Wide Web, security and cryptography, audio and video systems.

Education

Ph.D., Computer Science and Engineering — June, 1999
University of Washington, Seattle, WA.

M.S., Computer Science and Engineering — June, 1995
University of Washington, Seattle, WA.

B.S., Mathematics / Computer Science and Logic & Computation — May, 1993
Carnegie Mellon University, Pittsburgh, PA.

Awards

The C|Net Awards for Internet Excellence, 1995

MetaCrawler was one of three finalists for the Best Internet Search Engine.

Allen Newell Award for Excellence in Undergraduate Research, 1993

Professional Experience

University of Washington, Seattle, WA — September, 1993 – present

Research assistant for Professor Oren Etzioni. My primary research interests have been the study and integration of World Wide Web resources, with emphasis on information retrieval aspects, and methods of improving searchable indices through user interaction.

I implemented MetaCrawler, one of the first World Wide Web meta search services, and administered MetaCrawler as an Web service for over a year. I was responsible for ensuring the MetaCrawler remained fast and responsive as the number of users grew, given limited hardware resources. I developed many software optimizations and automated server administration tools towards this effort. Before MetaCrawler was licensed in 1996, it was handling almost 100,000 queries per day with room to grow, which was roughly 3-5 times as many as its closest competitor, SavvySearch.

I was also responsible with the continued maintenance and administration of the Ahoy! home page finding service. This service continues to garner roughly 1,000 queries per day.

I created HuskySearch, a second generation meta-search service based on MetaCrawler. HuskySearch improved on the MetaCrawler engine by using an open architecture that allowed users to add previously unknown search services to HuskySearch at run time. A prototype of this technology integrated the Clio inter-user history search application. In addition, I developed a technique called Collaborative Index Enhancement that allowed a searchable index to be modified after query sessions, in order to help improve search results. HuskySearch has also been the basis for other students research, including two undergraduate senior theses, two master's projects, and another doctoral thesis.

While investigating the overlap of search service results, our group observed that the experiments that used search results as data were very unstable. This led to my most recent work on empirical evaluation of major search services. This evaluation measures how rapidly the results of a query change over time, as well as how different the results of a query are when the query is submitted with different query options.

Independent Consulting, 1995 - present

I aided several entrepreneurs by evaluating business plans, creating software development plans and schedules, and providing technical advice and insight.

In addition, I adopted the MetaCrawler code for a custom application designed to issue a large number of queries repeatedly, download the URLs returned, and report on the differences in the documents' text.

Netbot Inc. (acquired by Excite Inc., now @Home Inc.), Seattle, WA Summer 1996

Netbot Inc., a Seattle-based startup, licensed MetaCrawler from our research group. Over the summer of 1996, I worked with Netbot to transfer the MetaCrawler technology and initiate the groundwork for Netbot to host a commercial MetaCrawler service, which would be able to handle upwards of 10 million queries per day at launch. In addition, I was responsible for initial negotiations with the Web search services for commercial use of MetaCrawler with their services, and I was heavily involved with the conception of the business plan for the commercial MetaCrawler service.

Shortly after I returned to my studies, Netbot was acquired by Excite Inc. for $35 million dollars for its Jango software product, a comparison shopping agent that used the MetaCrawler engine, and is now the group responsible for Excite's comparison shopping page. Excite was later acquired by @Home, Inc. in spring of 1999 for 7 billion dollars.

AT&T Bell Laboratories, Murray Hill, NJ Summer, 1994

At AT&T Bell Laboratories, I worked with Bart Selman and Henry Kautz on the Bots project. The Bots project was an exploration into creating personal assistant software agents that would communicate with one another to accomplish various tasks, such as meeting scheduling, expertise referral, and e-mail prioritizing. My tasks were to re-write the Bots in a more manageable way as well as to create a Bot communications protocol so that Bots could effectively transfer information between themselves.

Pittsburgh Science Center, Pittsburgh, PA Summer, 1993

Research intern at the Pittsburgh Science Center, working with Prof. Adam Beguelin on the Parallel Virtual Machine (PVM) project. Designed a monitoring and

debugging tool for applications using PVM as well as designed and integrated a Kerberos-based authentication system for PVM applications.

Teaching Experience

University of Washington, Seattle, WA **Winter – Spring, 1994**

Teaching Assistant for Professors John Zahorjan (Winter Quarter) and Steve Hanks (Spring Quarter) for UW CSE undergraduate second-quarter introduction to computer science course. This course is taken by roughly 400 students per quarter, and gives undergraduates a further understanding of more advanced introductory topics, such as object oriented programming, searching and sorting, pointers, etc. With John Zahorjan and 3 other TAs, we migrated the course from using Ada on UNIX systems to C++ on Windows, Mac, and UNIX systems.

Publications

Submitted For Publication

"On the Instability of Web Search." Erik Selberg and Oren Etzioni. Submitted to Science Magazine.

Fully Refereed Papers

"Multi-Service Search and Comparison using the MetaCrawler." Erik Selberg and Oren Etzioni. In *Proceedings of the 4th International World Wide Web Conference*, Dec., 1995.

"TRON: Process-Specific File Protection for the UNIX Operating System." Andrew Berman, Virgil Bourassa, and Erik Selberg. In *Proceedings of the 1995 Winter USENIX Conference*, Jan., 1995.

Invited Papers

"The MetaCrawler Architecture for Resource Aggregation on the Web." IEEE Expert, Jan. / Feb. 1997, 12(1).

Technical Reports

"Experiments with Collaborative Index Enhancement." Erik Selberg and Oren Etzioni. University of Washington Tech Report UW-CSE-98-06-01, June 1998.

"How to Stop a Cheater: Secret Sharing with Dishonest Participants." Erik Selberg. Carnegie Mellon University Tech Report CMU-CS-93-182, June 1993.

Senior Thesis Advising

As part of the undergraduate honors program, Undergraduate seniors complete a senior project where they work closely with graduate students and professors.

Christin Boyd **Winter and Spring 1996**

Design and implementation of a query refinement system for HuskySearch. This system attempted to aid the user in improving a given query as well as provide us with failure data on poor queries.

Darren Schack **Summer 1996**

Design and implementation of a distributed document caching mechanism for HuskySearch. This system would cache documents downloaded from the Web on demand to benefit repeated or similar queries.

Tim Bradley **Fall 1995 and Winter 1996**

Design and implementation of a system to facilitate MetaCrawler log mining. This data would allow us to discern patterns in what clients were looking for, what they received, and what they subsequently followed.

Invited Talks

Decade of the Web Symposium, University of Iowa **March, 1999**

Presentation on the current state of meta search technology and the World Wide Web, including details on MetaCrawler, HuskySearch, and other Web-related IR projects at the University of Washington.

IBM T.J. Watson Research Center **Oct. 1998**

Boeing Corp. **May 1998**

Presentations on MetaCrawler and HuskySearch, describing both the technical innovations and practical applications for both Internet and Intranet use.

Data Mining Summit **Mar. 1997**

Presentation on meta search technology for data mining professionals, with emphasis on practical applications of current research.

Distributed Indexing and Searching Workshop **May 1996**

IETF Group Meeting **June 1996**

Presentation on MetaCrawler technology emphasizing practical and economic implication of meta search services for leading academics and industry professionals. During the workshop a proposal was reached for scaling meta search services via query routing, which was then presented at the FIND group of the IETF.

Software

HuskySearch. HuskySearch is a second generation meta search service available at the University of Washington. HuskySearch is the primary search service available for searching the University of Washington and is an ongoing research testbed for World Wide Web IR. HuskySearch is available at

http://huskysearch.cs.washington.edu.

MetaCrawler. MetaCrawler is one of the original World Wide Web meta search services. Its early popularity was instrumental in the formation of Netbot, Inc., a startup company founded at the University of Washington and acquired by Excite, Inc. for $35 million. MetaCrawler has since been licensed to Go2Net Inc., a Seattle start-up, and is one of the premiere sites on the Go2Net Network. The Go2Net Network was ranked #25 by Media Metrics in terms of overall web traffic in January 1999. MetaCrawler is available at http://www.metacrawler.com.

Department and University Activities

Graduate Student Orientation Committee, 1994–1995: Supervised orientation presentation for new graduate students in the CSE Department.

References

Professor Oren Etzioni *etzioni@cs.washington.edu*
Department of Computer Science and Engineering, University of Washington

Professor Dan Weld *weld@cs.washington.edu*
Department of Computer Science and Engineering, University of Washington

Professor Ed Lazowska *lazowska@cs.washington.edu*
Department of Computer Science and Engineering, University of Washington

Professor Efthimis Efthimiadis *efthimis@u.washington.edu*
Information School, University of Washington

Advice

If you have tickets to Blondie for the night before your 9 A.M. defense and are wondering whether or not you should go, you should go.