HTML-QS: A QUERY SYSTEM FOR HYPERTEXT MARKUP LANGUAGE DOCUMENTS

A Thesis
Submitted to the Faculty of Graduate Studies and Research
In Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science
University of Regina

By
Yibin Su
Regina, Saskatchewan
April, 2006

Dedication

To my family and my parents

Abstract

The Internet is a huge information centre consisting of billions of web pages. The Hyper Text Markup Language (HTML) is perhaps the most popular specification for constructing web pages on the Internet. Methodologies have been developed to search and query HTML web pages. Search engines such as Google can match given keywords on web pages and identify the most relevant pages. However, search engines can only match keywords and are not able to query web pages in the same manner that one can query databases. Query languages can specify requests for retrieving information by treating web pages as semi-structured data on which to perform queries, but previously existing data models for query languages either cannot capture the hierarchical structure within an HTML web page or can only present the structure at a low level.

In this thesis, we describe HTML-QS, a web-based query service for HTML documents. In previous research, Liu and Ling proposed HTML-CM, a conceptual data model for HTML, and HTML-QL, a rule-based query language. HTML-QS is the first implementation of HTML-CM and HTML-QL. Although HTML-CM has only a few constructs, it is powerful enough to capture the complex hierarchical structure within a HTML web page in a way that is close to the human visualization of the web page. Based on the HTML-CM conceptual data model, HTML-QL is able to query not only the intra-document structure, which is the structure within a single HTML document, but also the inter-document structure, which is the structure among groups of HTML documents. The results of such a query can be restructured and presented to the user in the manner specified in the query. The HTML-QS system provides a high level view of the structures and data in HTML documents. This prototype system demonstrates that a rule-based query language can integrate features of a database query language and a logic programming language.

Acknowledgements

I would like to express my gratitude to Dr. Howard Hamilton, my supervisor, for his consistent support. I would not have been able to finish this thesis without his help. I would like to thank my former supervisor, Dr. Mengchi Liu, for his financial support and constructive guidance, which led to the choice of topic and initial work on this thesis. I would also like to thank the other members of my supervisory committee, Dr. Larry Saxton and Dr. Yiyu Yao, for their time and effort.

Many thanks go to my friends Hongwei Qi, Peijin Zhou, Mahesh Shresthma, and Richard Guo for their invaluable support and encouragement.

I would also like to acknowledge my family and my parents for their selfless support, understanding, and encouragement. I know that they have made significant sacrifices for my education. Without their help, it would have been impossible for me to complete my studies. I will remember everything they have ever done for me forever.

The Faculty of Graduate Studies and Research and the Department of Computer Science of University of Regina and the Natural Science and Engineering Research Council of Canada provided financial support to me while I was working on my M.Sc. program. Their support is much appreciated.

Contents

Abstract ii

Acknowledgements iv

Table of Contents v

List of Tables vii

List of Figures viii

List of Acronyms ix

Chapter 1 Introduction 1

1.1 Problem Statement 1

1.1.1 The Internet and the World Wide Web 1

1.1.2 Data Management 3

1.1.3 Semi-Structured Data 4

1.1.4 Querying Semi-Structured Data 5

1.2 Approach 7

1.2.1 Search Engines 7

1.2.2 Web Query Languages 9

1.3 Contributions and Outline 11

Chapter 2 Background 13

2.1 Google, An Example of a True Search Engine 13

2.1.1 Google System Architecture 16

2.2 Survey of Web Query Languages 18
2.2.1 WebSQL 18
2.2.2 W3QL 20
2.2.3 WebOQL 21
2.2.4 WebLog 23
2.2.5 StruQL 25
2.2.6 NetQL 26
2.2.7 Summary 27

Chapter 3 Approach **29**
3.1 The HTML-CM Conceptual Model for HTML 30
3.2 Converting HTML Documents to HTML-CM Objects 31
3.3 A Rule-based Query Language for HTML Documents 41

Chapter 4 System Architecture and Results **50**
4.1 High Level System Architecture 50
4.2 Implementation 52
4.2.1 Reasons for Selecting Java 53
4.2.2 Modular Components 53
4.3 Evaluation and Comparision 65

Chapter 5 Conclusions and Future Research **78**
5.1 Conclusions 78
5.2 Future Research 79

References **83**

Appendix A Conversion Rules **89**

List of Tables

2.1 PageRank Scores for Five Web Pages 15
2.2 The Document Relation in WebSQL (adapted from [5]) 19
2.3 The Anchor Relation in WebSQL (adapted from [5]) 19
2.4 Summary of Web Query Languages 27
4.1 Predefined Tokens in the Query Parser 56
4.2 List of Object Processors 58
4.3 Elapsed Time for Example Queries (seconds) 76
4.4 Summarization of Testing Example Queries 77

List of Figures

1.1 Ley's DBLP Bibliography 6

1.2 Google Search Space Size 8

2.1 A Set of Interconnected Pages 15

2.2 The High Level System Architecture of Google [11] 17

2.3 A Hypertree in WebOQL 22

3.1 Dr. Robert J. Hilderman's Homepage 42

3.2 Page about Canada in the World Factbook at www.odci.gov 47

3.3 Converted HTML-CM for the Factbook page on Canada 48

4.1 The HTML-QS System Architecture 51

4.2 The Control Flow During Query Processing 54

4.3 The HTML-CM Tree 59

4.4 The Tree for Processing a Table 61

4.5 The Data Structure for the Result Table 63

4.6 The Head Result List 64

List of Acronyms

ANSI	American National Standard Institute
ARPAnet	Advanced Research Project Agency network
DBMS	Database Management System
DHTML	Dynamic HTML
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
HTML-CM	A Conceptual Model for HTML
HTML-QL	A Rule-based Query Language for HTML
HTTP	Hypertext Transfer Protocol
JVM	Java Virtual Machine
LDR	Local Data Repository
OOD	Object Oriented Design
OODB	Object Oriented Database
OQL	Object Query Language
RDBMS	Relational Database Management System
SQL	Structured Query Language
URL	Uniform Resource Locator
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language

Chapter 1

Introduction

In this chapter, an introduction to the thesis is given. The problem addressed in this thesis is discussed in Section 1.1. Section 1.2 describes several approaches to the problem and introduces our approach. Section 1.3 outlines the thesis document and lists its contributions to research.

1.1 Problem Statement

This section describes the problem addressed in this thesis. First, background information on the Internet and World Wide Web is presented in Section 1.1.1, and then the use of databases for database management is described in Section 1.1.2. Next the nature of semi-structured data is explained in Section 1.1.3. Finally, in Section 1.1.4, we present the problem to which this thesis is trying to provide a better solution: querying semi-structured data.

1.1.1 The Internet and the World Wide Web

What is now known as the Internet began in 1969 as a U.S. Defense Department network called the Advanced Research Projects Agency network (ARPAnet). This network was designed for military purposes [25]. The network was opened for public use in the 1970s and flourished in the late 1980s, when most universities and many companies around the world were connected to it. After Berners-Lee proposed the World Wide Web project in 1990, the Internet grew even more rapidly and a new era of computer communications began [3].

Many applications and services are based on the Internet, including Telnet, Secure Shell, File Transfer Protocol (FTP), newsgroups, and E-Mail, as well as the World Wide Web. The *World Wide Web* [1] (also known as *WWW*, the *web*, or *W3*) has been described as "the universe of network-accessible information, the embodiment of human knowledge" [48]. From the above description, we can imagine that there is a huge amount of information on the web. In fact, the web is many things [9].

(1) **A concept:** The World Wide Web was conceived as a seamless world in which all information, from any source, can be accessed in a consistent and simple way.

(2) **A set of protocols:** The protocols that define web communication include the Uniform Resource Locator (URL), the Hypertext Transfer Protocol (HTTP), and the HyperText Markup Language (HTML).

(3) **A collection of software:** The software that enables web connections between client software, such as web browsers, and server software, such as HTTP server applications.

(4) **The universe of online information:** The complete set of information available on the web is stored on many servers of a variety of types. Each type of server provides a number of different services. For example, an FTP server provides archives for download, and an HTTP server provides hypertext information for browsing. Both the archives and the hypertext information are part of the universe of online information.

The number of web sites and web pages on the web has increased incredibly. At the end of 2004, Google (www.google.com), one of the most comprehensive search engines on the Internet, claimed that it had indexed more than 8 billion web pages [19]. In August 2005, Yahoo!, another popular search engine, claimed to have indexed over 19.2 billion web pages [34]. The web has indeed become a world wide information center. It contains abundant information across various areas. The problem of finding useful information efficiently in the billions of web pages has become a major concern.

[1] In this thesis, italics are used to indicate the defining occurrence of a technical term.

HTML is perhaps the most commonly used language used for web presentation on the Internet today. For example, Microsoft and Google both are using HTML in their web sites at the time of the writing of this thesis. Although newer standard web markup languages, such as XHTML, are becoming increasingly popular, they are based on HTML. Another language used on the Internet is XML. XML is a self-describing language and is widely used for data exchange. Some other related languages are XQuery [10], XML-QL [14], and XML-GL [13]. In this thesis, we focus on HTML. Finding an effective way of extracting and analyzing HTML documents remains crucial to the goal of querying the web.

1.1.2 Data Management

Commercial and institutional data is commonly managed using Database Management Systems. A *Database Management System* (DBMS) is a software system that manages one or more databases. DBMSs were first developed in the 1960's [47]. DBMSs were designed to process large amounts of data, typically via a series of transactions, while maintaining the integrity of the underlying database.

The earliest DBMSs did not support high level query languages [47]. After Codd introduced the relational model in 1970 [51], commercial relational database management systems (RDBMSs) developed rapidly. The most important contribution of RDBMSs was the Structured Query Language (SQL), which is the language recommended by the American National Standards Institute (ANSI) for relational database definition and manipulation. SQL is both a data definition language and a data manipulation language. RDBMSs have been applied by large corporations and institutions all over the world to process huge amounts of data.

Intuitively, a RDBMS might provide an effective way of querying web documents if we could treat the web as a huge database. However, traditional RDBMSs require a strict table-based data format and a schema defined in advance. For example, an electronic transaction record for a retail store might include the store number, the cashier name, the transaction number, the total, the date, the time, and a list of items. Each item on the list might then have attributes, such as name, code, and price.

Because the structure of every transaction record at the store is exactly the same, this type of transaction data is well-structured and strictly typed. Therefore, a schema can easily be defined based upon the structure. Although object-oriented database (OODB) management systems allow more types of structures than relational systems, they still require predefined schemas. Now let's take a look at the web. Most web pages are written in HyperText Markup Language (HTML). Can we define a schema for HTML? Unfortunately, the answer is "no", because HTML documents are not fully structured. Neither, however, are they completely formless raw data because there are about 100 elements in HTML that are used to formalize web pages. The characteristics of HTML documents make them fall into a category of data called semi-structured data.

1.1.3 Semi-Structured Data

Roughly speaking, *semi-structured data* are data that are neither raw data nor strictly typed data as would be found in a conventional database system [1]. This is only a rough definition, because the same data that are originally considered unstructured may become highly structured after an analysis has been performed. A weak structuring primitive, called a tag, exists in HTML documents. A *tag* is an HTML keyword enclosed in angle brackets, such as `<title>` or `<table>`. A string of HTML code embedded between a starting tag, such as `<title>`, and a matching ending tag, such as `</title>`, is called an HTML *element*. For example, `<title> My Thesis </title>` is a TITLE HTML element. For some elements, such as the PARAGRAPH element, which starts with `<p>`, the corresponding ending tag `</p>` is optional. The `<a>` and `</a>` tags are used to identify links in HTML. A *link* is some HTML code that indicates a connection either to another part of the same HTML document or to a different document. The *destination* of a link is defined immediately after the href attribute in the starting tag. For example, `<a href="http://www.w3.org/">W3C Web site</a>` is a link, and "http://wwww.w3.org/" is the destination of the link.

HTML documents are not well-structured, because no fixed schema can be defined in advance. For example, let's consider Ley's DBLP Bibliography web page at http://www.informatik.uni-trier.de/∼ley/db. The contents of this page on August 4, 2005 were as shown in Figure 1.1.

This page has five sections, with different types of items in each section. Searching methods such as "Author", "Title", and "Advanced" are listed in the Search section, while types of reference material, such as conferences, journals, and books, are listed in the Bibliographies section. Having these five sections makes the page partially structured, compared to a fully structured table composed of rows and columns. Some other information, such as the list of mirror sites, does not belong to any section, which makes the structure irregular. Section titles, such as Search and Links are presented in boldface in a larger font than the other text, which often implies that the following content provides more detailed information related to the section titles. To summarize, the main characteristics of semi-structured data are:

(1) the structure is irregular,

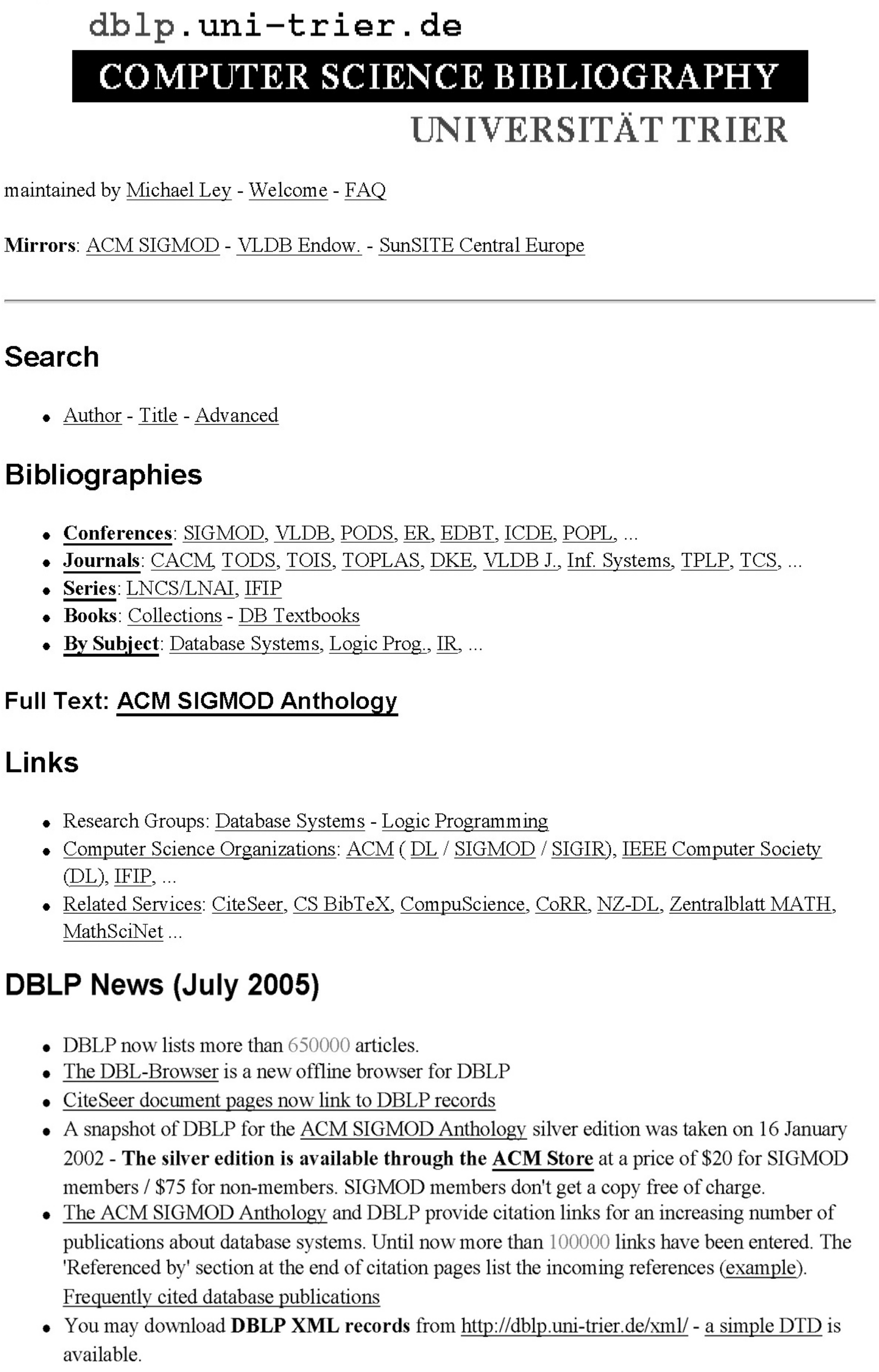
(2) the structure is implicit, and

(3) the structure is partial.

1.1.4 Querying Semi-Structured Data

From the point of view of a database researcher, the web can be regarded as a huge database, and the HTML documents are data in the database. In a conventional database system, we can define schema for data, create a database, and then query the database using SQL or the Object Query Language (OQL) [6]. If the web is treated as a database, conventional database systems cannot be applied, because HTML documents are semi-structured and they do not have a fixed schema. Therefore, we have to find other approaches to query HTML documents.

Liu and Ling suggested a general approach [30, 31, 32] to querying HTML documents in a simple way that they argue is close to human conceptualization. The goal of this thesis is to confirm that this approach can be implemented and the implemented approach functions as expected. To do so, we have implemented a prototype system that demonstrates the distinct features of this approach.



dblp.uni-trier.de

COMPUTER SCIENCE BIBLIOGRAPHY

UNIVERSITÄT TRIER

maintained by Michael Ley - Welcome - FAQ

Mirrors: ACM SIGMOD - VLDB Endow. - SunSITE Central Europe

Search

- Author - Title - Advanced

Bibliographies

- **Conferences**: SIGMOD, VLDB, PODS, ER, EDBT, ICDE, POPL, ...
- **Journals**: CACM, TODS, TOIS, TOPLAS, DKE, VLDB J., Inf. Systems, TPLP, TCS, ...
- **Series**: LNCS/LNAI, IFIP
- **Books**: Collections - DB Textbooks
- **By Subject**: Database Systems, Logic Prog., IR, ...

Full Text: ACM SIGMOD Anthology

Links

- Research Groups: Database Systems - Logic Programming
- Computer Science Organizations: ACM (DL / SIGMOD / SIGIR), IEEE Computer Society (DL), IFIP, ...
- Related Services: CiteSeer, CS BibTeX, CompuScience, CoRR, NZ-DL, Zentralblatt MATH, MathSciNet ...

DBLP News (July 2005)

- DBLP now lists more than 650000 articles.
- The DBL-Browser is a new offline browser for DBLP
- CiteSeer document pages now link to DBLP records
- A snapshot of DBLP for the ACM SIGMOD Anthology silver edition was taken on 16 January 2002 - **The silver edition is available through the ACM Store** at a price of $20 for SIGMOD members / $75 for non-members. SIGMOD members don't get a copy free of charge.
- The ACM SIGMOD Anthology and DBLP provide citation links for an increasing number of publications about database systems. Until now more than 100000 links have been entered. The 'Referenced by' section at the end of citation pages list the incoming references (example). Frequently cited database publications
- You may download **DBLP XML records** from http://dblp.uni-trier.de/xml/ - a simple DTD is available.

Figure 1.1: Ley's DBLP Bibliography

1.2 Approach

Two approaches to the problem of querying HTML documents have been developed. The first approach, which is introduced in Section 1.2.1, uses a search engine to find specific terms in HTML documents. The second approach, which is introduced in Section 1.2.2, defines queries in a web query language.

1.2.1 Search Engines

A *search engine* is a computer program that searches a database and reports information that contains or is related to specified terms [38]. The common user interface to a search engine is a web site where a user can enter keywords, click on any returned link, and browse the linked web pages.

Search engines first appeared on the Internet approximately a decade ago. Two pioneering search engines, the World Wide Web Worm or WWWW (wwww.cs.colorado.edu/wwww) and Yahoo! (www.yahoo.com), both started in 1994. Since then, many other search engines, such as Alta Vista, Infoseek, FastSearch, and Google, have been developed. Over time, some of them have survived, and some have disappeared, which reflects the intense competition and rapid technological development of the Internet.

The scale of search engines, in terms of the number of web pages that they index, has been increasing significantly. Figure 1.2 shows the number of web pages indexed by the Google search engine (www.google.com) between 2001 and 2004.

Search engines can be divided into three types: true search engines, subject directories, and meta-search engines. True search engines and subject directories have their own indexes for users to search while meta-search engines do not have their own indexes. Where indexes are present, the methods that the search engines use to build their indexes are the key to classifying the search engines. One method of building an index is to use an automated program to search the web for new or updated pages. A search engine built this way is called a *true search engine*. We call it a true search engine, because it actually searches the indexed web pages for matches. Another method is to have people build indexes of web pages. A search engine built this way

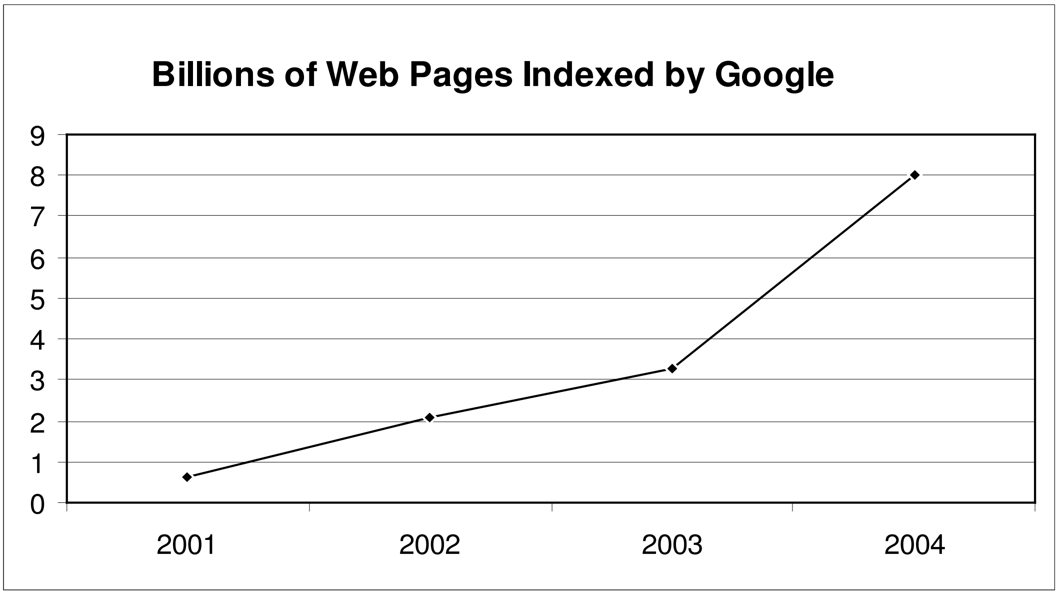


Figure 1.2: Google Search Space Size

is called a *subject directory*.

Generally speaking, a true search engine consists of three major components: a spider, an index, and search engine software. An example of a true search engine is Google. We discuss its architecture in detail in Chapter 2.

A subject directory is a catalog of sites collected and organized by people. An example of a subject directory is Yahoo!, as it was originally implemented. Subject directories are often called *subject trees*, because they start with a few main categories and then branch into subcategories, topics, and subtopics.

Subject directories are helpful for finding information on a topic when a user does not have a precise idea of what he or she wants. Large directories, such as Yahoo!, also provide an option for keyword searching, which can eliminate the need to work through numerous levels of topics and subtopics.

A subject directory can be regarded as a database system that accumulates data by means of manual entry. A web site owner submits a URL along with information required by the directory, such as the title and a short summary of the web site. An editor then writes a description for the site. A search initiated by a user looks for matches only in the submitted descriptions.

The third type of search engine is called a meta-search engine (also known as a *parallel search engine* or a *multiple search engine*). A *meta-search engine* takes the keywords entered by a user, simultaneously sends the query to a number of search engines (true search engines or subject directories), and then presents the search results returned by those search engines to the users. This type of search engine does not have an index or a database other than its list of search engines. An example of a meta-search engine is MetaCrawler (www.metacrawler.com).

1.2.2 Web Query Languages

A *web query language* is a query language designed to query the web. Its purpose is similar to that of SQL, which was designed to query an RDBMS. As discussed in the previous section, a search engine generally seeks only textual matches and does not take into account either the *intra-document structure*, which is the structure within a single HTML document, or the *inter-document structure*, which is the structure among groups of HTML documents. The Google search engine allows one to specify that the search terms must appear in the title or link of the web pages, which enables Google to recognize a few aspects of the internal structure of the web pages. Nonetheless, it does not regard a whole web page as a structured document. As Frank Halasz states [21]:

> "Content search ignores the structure of a hypermedia network. In contrast, structure search specifically examines the hypermedia structure for subnetworks that match a given pattern."

To make use of structure, a number of structural web query languages and systems have been developed over approximately the last decade. Web query languages have been classified into two generations [18]. The first generation includes W3QL [24], WebSQL [5], and WebLog [27]. First generation web query languages allow queries that combine conditions of text patterns appearing within documents with graph patterns describing the link structure. They treat a web page as an indivisible object with two properties. The first property is whether or not the web page contains certain text patterns. The second property is that the web page points to other objects by link attributes. These languages take inter-document structure into account, but they do not consider intra-document structure.

The second generation web query languages are called "web data manipulation languages" [18]. They include WebOQL [4], StruQL [16], NetQL [20, 29], and Florid [22]. These languages have two significant enhancements compared with the first generation languages. First, they provide access to the structure of the web objects that they manipulate. Unlike the first-generation languages, they model the intra-document structure of web documents, as well as the external links that connect web documents. Some of these languages support references, which allow them to model hyperlinks, and some of them support ordered objects, which allow them to represent data in a natural fashion. However, there are many HTML elements that these languages do not support, as will be discussed in Chapter 2. Secondly, except for NetQL, these languages provide the ability to create complex structures to represent the results of queries. Since data on the web are commonly unstructured or semi-structured, the design of these languages emphasizes the ability to support semi-structured features.

Recently, Liu and Ling proposed a conceptual data model and rule-based query language for structuring and querying HTML documents [30, 31, 32]. In general, a *data model* is a logical data structure that describes, in an abstract way, how data are organized and represented. A *conceptual data model* (or more simply a *conceptual model*) is a data model of the conceptual structure within some data. A conceptual model for an HTML document is a visual representation of a HTML document. Liu and Ling's conceptual model is called HTML-CM, which stands for HTML Conceptual Model. HTML-CM has only a few simple constructs, but it is able to represent the complex hierarchical structure within an HTML document at a level which is claimed to be close to the human conceptualization of the document.

Based on this conceptual model, Liu and Ling proposed a rule-based language, called HTML-QL, for querying HTML documents over the Internet [30, 31, 32]. This language provides a simple but effective way to query both intra-document structures and inter-document structures and allows the query results to be restructured. Being rule-based, it naturally supports negation and recursion, and therefore, it is more expressive than the SQL-based web query languages.

1.3 Contributions and Outline

This thesis describes the first implementation of HTML-QS, an approach to web search and inference based on HTML-CM and HTML-QL. The system is intended to provide a high level view of structures and data that are stored in HTML format. One of the key contributions of this prototype system is that it demonstrates how a rule-based query language can integrate features of database query languages and logic programming languages. The system is implemented using the Java programming language and is available online at http://www.scs.carleton.ca/∼mengchi/HTML-QL.

The remainder of the thesis is organized as follows.

Chapter 2 presents background material relevant to search engines and web query languages. A detailed discussion of the system architecture of Google is provided. Other web query languages are surveyed as well, including three first-generation languages, WebSQL, W3QL, and WebLog, and two second-generation languages, WebOQL, StruQL, and NetQL.

Chapter 3 describes our approach by explaining the HTML-CM conceptual model and the rule-based HTML-QL query language. Examples of how to convert HTML documents to HTML-CM web objects, and how to query HTML documents using the HTML-QL query language, are presented.

Chapter 4 presents the high level system architecture of the HTML-QS prototype system, followed by the details of our implementation of HTML-QS. Some components and procedures that presented difficulties during the design and implementation are also discussed. Evidence that HTML-QS correctly implements the approach is given in the form of results for a comprehensive set of test queries.

Chapter 5 concludes the thesis and summarizes the contributions of this research. Open issues, including memory management, robustness, and extensibility, are discussed as well. Appendix A gives the full set of rules for converting HTML documents to HTML-QS web objects.

Chapter 2

Background

This chapter is organized as follows. In Section 2.1 we discuss the architecture of Google, an example of a true search engine. Section 2.2 introduces related research on web query languages.

2.1 Google, An Example of a True Search Engine

As mentioned in Chapter 1, Google is a true search engine. We discuss this type of search engine, because it is closely related to our research on querying the structure of HTML documents. Google has powerful strong searching and ranking abilities, but these are not focuses of our research. We evaluate Google only with respect to its ability to query the structure of HTML documents. Changes made to Google after mid 2005 were not considered in this thesis.

Competitive pressures prevent the developers of commercial true search engines from publishing the details of their system architectures, but an introductory article, describing the original architecture of Google, is available [11]. The article was written when Google was still a research project.

A true search engine utilizes several computer programs in order to obtain web documents and build its database. As mentioned in Chapter 1, a typical true search engine consists of three key components [43], a spider, an index, and the search engine software.

The *spider* (also called the *crawler* or the *robot*) downloads web pages to the search engine, which in turn provides lists of URLs that need to be fetched by the spider. A spider uses HTTP to send requests and receive responses. In 1998, it was reported that a spider could crawl twenty five pages per second [11], which is equivalent to more than two million pages a day. Everything the spider finds goes into the index. The index is a huge catalog, and it can be regarded as a database holding web pages. The search engine software is the program that searches through the millions of pages stored in the index in order to find matches to a query and rank them in order of their apparent relevance to this query.

As a true search engine, Google has the three components just described. Google is the most frequently used search engine on the Internet [44], because, in addition, it utilizes a few novel techniques in order to enhance its search ability. The four key features of Google are the use of the PageRank algorithm, the use of anchor text, the utilization of visual presentation details, and the storage of full, raw HTML for the pages. These features are now discussed.

The single most important feature of Google is the PageRank algorithm. The PageRank algorithm relies on the democratic nature of the Internet by using its massive link structure as an indicator of an individual page's value. In essence, Google interprets a link from page A to page B as a vote, by page A, for page B. Furthermore, Google takes into account not only the number of votes a page receives, but also analyzes the page that casts the vote. Votes cast by pages that are themselves "important" weigh more heavily and help to make other pages "important." [45]. A web page's PageRank is an objective measure of its importance that corresponds well with people's subjective idea of importance. Because of this correspondence, PageRank is the best known way to prioritize search results [11, 39]. The PageRank of a web page, W, is defined as follows. Assuming that pages T_1, T_2, ..., T_n have links pointing to W, that d is a damping factor between 0 and 1, usually set to 0.85 (a detailed discussion of d can be found in [11]), and that $C(T_i)$ is the *outdegree* of page T_i, i.e., the number of links going out of page T_i, the PageRank of W is:

$\text{PageRank}(W) = (1\text{-}d) + d(\text{PageRank}(T_1)/C(T_1) + ... + \text{PageRank}(T_n)/C(T_n))$

For example, consider the small set of interconnected pages shown in Figure 2.1.

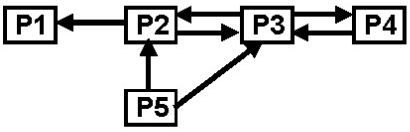


Figure 2.1: A Set of Interconnected Pages

Using the formula defined above, we can calculate the PageRank of all five pages. For example, for page P5, because no links point to it, we obtain:

PageRank(P5) = (1 - 0.85) = 0.15

For page P2, we obtain:

PageRank(P2) = (1 - 0.85) + 0.85(PageRank(P3)/C(P3)
+ PageRank(P5)/C(P5))

The outdegree values for the pages can be obtained by inspection of the graph in Figure 2.1. These values are recorded in the second column of Table 2.1. From Figure 2.1, we obtain:

C(P3) = 2, C(P5) = 2

After substituting values for PageRank(P5), C(P3), and C(P5) into the equation for PageRank(P2) , we obtain:

PageRank(P2) = 0.15 + 0.85(PageRank(P3)/2 + 0.15/2)

To determine PageRank(P2), we need to calculate PageRank(P3) first. After several substitutions, we obtain the results shown in the third column of Table 2.1.

Table 2.1: PageRank Scores for Five Web Pages

Web Page T	C(T)	PageRank(T)
P1	0	0.41121
P2	2 (P1, P3)	0.61460
P3	2 (P2, P4)	0.94318
P4	1 (P3)	0.55085
P5	2 (P2, P3)	0.15000

Page P3 has the highest PageRank, because three pages, namely P2, P4, and P5, point to it, which is the highest number in this small set. This high PageRank is interpreted to mean that this page is the most important. Page P5 has the lowest PageRank, because no page points to it, which is interpreted to mean that it is the least important.

The second key feature of Google is that it considers the visual presentation of the HTML pages when computing the PageRank. Words in a larger font or bold face are weighted higher than other words, because web page designers use these visual effects to emphasize the important parts of their text.

Thirdly, Google makes extensive use of *anchor text*, which is the text or label associated with a link to another page. Most search engines associate the text of a link with the page that the link is on. In addition, Google associates the text of a link with the page the link points to. This approach has two advantages. First, the anchors often provide more accurate descriptions of web pages than do the pages themselves. Secondly, anchors may exist for documents which cannot be indexed by a text-based search engine, such as images and databases. Thus, associating the text of a link with the page being linked to makes it possible to return web pages which have not actually been crawled.

Fourthly, Google has a repository in which full HTML web pages are compressed and stored. The repository is the basis for further indexing. It also makes it possible to support a function for viewing cached pages. Thus, a user can open a cached web page in the repository when the web page cannot be opened from its original server due to reasons such as unexpected server shutdown or the disappearance of the requested page.

2.1.1 Google System Architecture

An overview of Google is shown in Figure 2.2 [11]. The URL Server sends lists of URLs that need to be fetched to several distributed spiders called Crawlers. The Crawlers open connections to the URLs, send requests, read responses, and save web pages to the Store Server. The Store Server compresses web pages and stores them in the Repository. The Repository contains the compressed versions of the full HTML document of every web page, one after the other. Each page has an identifying number called the *document identifier* or the *docID*.

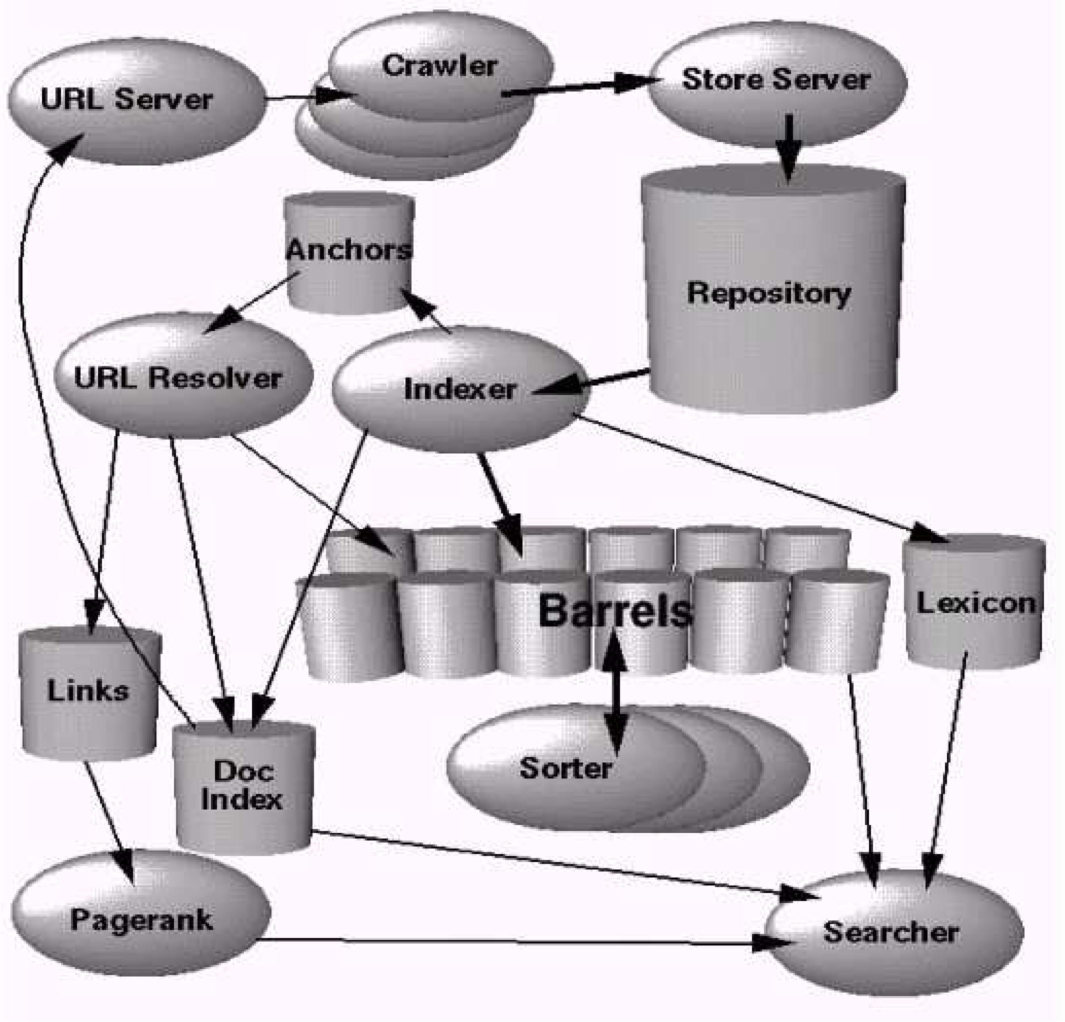


Figure 2.2: The High Level System Architecture of Google [11]

The Indexer reads the repository, decompresses the documents, and parses them. Each document is transformed to sets of word occurrences. For every word, the Indexer finds the corresponding wordID from the Lexicon, and then creates a hit list in which each item, called a *hit*, contains this word's position in the document, and its font size and capitalization. All hits are distributed into a set of Barrels. A *Barrel* is a data container that holds a range a wordIDs. The indexer also parses all links in each web page and stores them in an Anchors file, which contains information, such as the label of a link, which page it is in, and which page it points to.

The Doc Index contains information about every document, such as the document's status, pointers into the Repository, and various statistics. The Doc Index is ordered by docID. The Lexicon contains a list of words. As each document is parsed, every word is converted to a wordID using the Lexicon. The URL Resolver reads the Anchors file and converts URLs to docIDs. It also generates the Links database, each entry of which is a pair of docIDs representing the source and destination page of the link. The Sorter sorts the wordIDs in the Barrels to generate an index that is used to locate the docIDs by searching the wordIDs. The PageRank of each web page, the most important part of Google, is calculated based on the Links database.

2.2 Survey of Web Query Languages

In this section, we discuss several web query languages that have been developed, namely WebSQL [5, 36, 35], W3QL [24], WebOQL [4], WebLog [27], StruQL [16, 15, 17], and NetQL [20, 29]. Our discussion of these languages focuses on their data models, their language styles, and their abilities to query inter-document structure and intra-document structure. Three other web query languages that are not discussed in detail are Florid, Lorel, and UnQL. Florid [22] is an implementation of the deductive and object-oriented database language F-Logic [23]. The web is modeled using two classes, called url and webdoc. Its data model is very similar to that of WebSQL. Lorel [2, 41] and UnQL [12] are general purpose query languages for semi-structured data. They use tree-based and graph-based models to represent the web, but they do not model the intra-document structure in HTML documents. At the end of this section, we present a summary of languages we have studied in detail.

2.2.1 WebSQL

WebSQL is an SQL-like web query language designed to query inter-document structure. In WebSQL [5, 36, 35], the web is viewed as a *virtual graph* where the nodes are web documents and the edges are hypertext links between the documents. WebSQL models the web as a relational database with two relations: Document and

Anchor. The Document relation has one tuple for each HTML web page, as shown in Table 2.2 (adapted from [5]). In the table, the rows represent the pages, and the columns represent attributes of the pages, including the title, textual content, length in bytes, type, and modification date of the pages. Among all the attributes in the Document relation, only the TITLE attribute refers to an HTML element, namely TITLE. The Anchor relation has one tuple for each anchor in each HTML web page, as shown in Table 2.3 (adapted from [5]). In each row of the table, the first column gives the URL of the page where the anchor is located, and the second column gives the label of the anchor. The third column gives the URL of the destination page, i.e., the page where the anchor is pointing to. The LABEL and HREF attributes are derived from the links in the HTML web pages. The remaining attributes in the two relations are attributes of the web pages. Parsing the links in HTML web pages provides full access to the inter-document structure, while parsing the TITLE HTML elements provides only limited access to the intra-document structure.

Table 2.2: The Document Relation in WebSQL (adapted from [5])

URL	TITLE	TEXT	LENGTH	TYPE	MODIF
http://www...	title1	text1	1234	text	1-1-2005
http://www...	title2	text2	2345	text	2-3-2005
http://www...	title3	text3	3456	text	3-4-2005
...	...	...	...	...	...

Table 2.3: The Anchor Relation in WebSQL (adapted from [5])

BASE	LABEL	HREF
http://www...	label1	http://www...
http://www...	label2	http://www...
http://www...	label3	http://www...
...	...	...

The following example WebSQL query returns all pairs of URLs of documents that have the same title and that both contain the words City of Calgary. Using the

MENTIONS keyword triggers the running of a tool to use a search engine to fetch web pages that include the words “City of Calgary”.

```
SELECT d1.url, d2.url
FROM Document d1 SUCH THAT d1 MENTIONS "City of Calgary",
     Document d2 SUCH THAT d2 MENTIONS "City of Calgary"
WHERE d1.title = d2.title AND NOT ( d1.url = d2.url)
```

A shortcoming of WebSQL is that it regards the full text of an HTML page as one value of an attribute in a tuple, which limits its analysis and interpretation to simple text matching. WebSQL only recognizes two HTML elements, TITLE and LINK. Thus, it focuses on the inter-document structure, but it also retrieves the small portion of the intra-document structure that relates to TITLEs. Many other HTML elements that relate to intra-document structures, such as LISTs, SECTIONs, and TABLEs, are not considered by WebSQL.

2.2.2 W3QL

W3QL [24] is similar to WebSQL. It also models the web as a graph in which each URL constitutes a node, and each link in one node pointing to another node constitutes an edge. The most obvious difference between W3QL and WebSQL is that W3QL uses external programs for specifying content conditions on files, while WebSQL implements all the functions in an internal Java package. For example, the following query searches for HTML pages that have a title of Calgary Flames.

```
SELECT cp n2/* result;
FROM n1,l1,n2;
WHERE
n1 in Myindexes.url;
SQLCOND ( n2.format=HTML) AND ( n2.title="Calgary Flames");
```

The query visits every URL listed in the Myindexes.url file, which is denoted n1 in the query, and creates query results n2 in the result folder. The SQLCOND keyword is followed by the required condition for any result for the query. As with WebSQL, the only HTML elements that W3QL recognize are TITLE and LINK, and therefore, the majority of the intra-document structure is unavailable for querying.

2.2.3 WebOQL

WebOQL [4] models the web as a forest of hypertrees. A *hypertree* is an ordered arc-labeled tree. The two types of arcs in a hypertree are internal arcs and external arcs. An *internal arc* represents an object in the document modeled by a hypertree, and an *external arc* represents a hyperlink among documents. Arcs are given labels that identify attributes of the object, such as Tag, Text, Source, and Url.

For example, let's consider Ley's DBLP Bibliography web page at http://www.informatik.uni-trier.de/∼ley/db. The contents of this page on August 4, 2005 were as shown in Figure 1.1. The HTML code of the page is as follows (pieces of HTML code referring to irrelevant aspects, such as color and script, have been removed in order to make this example more concise):

```
<html>
<head><title>DBLP Bibliography</title></head>
<h2>Search</h2>
<ul>
<li><a href="indices/a-tree/index.html">Author</a>
    <a href="indices/t-form.html">Title</a>
    <a href="http://... .../query.html">Advanced</a>
</ul>
<h2>Bibliographies</h2>
<ul>
<li><a href="conf/indexa.html">Conferences</a>:
    <a href="conf/sigmod/index.html">SIGMOD</a>,
    <a href="conf/vldb/index.html">VLDB</a>,
    <a href="conf/pods/index.html">PODS</a>,
```

. . .

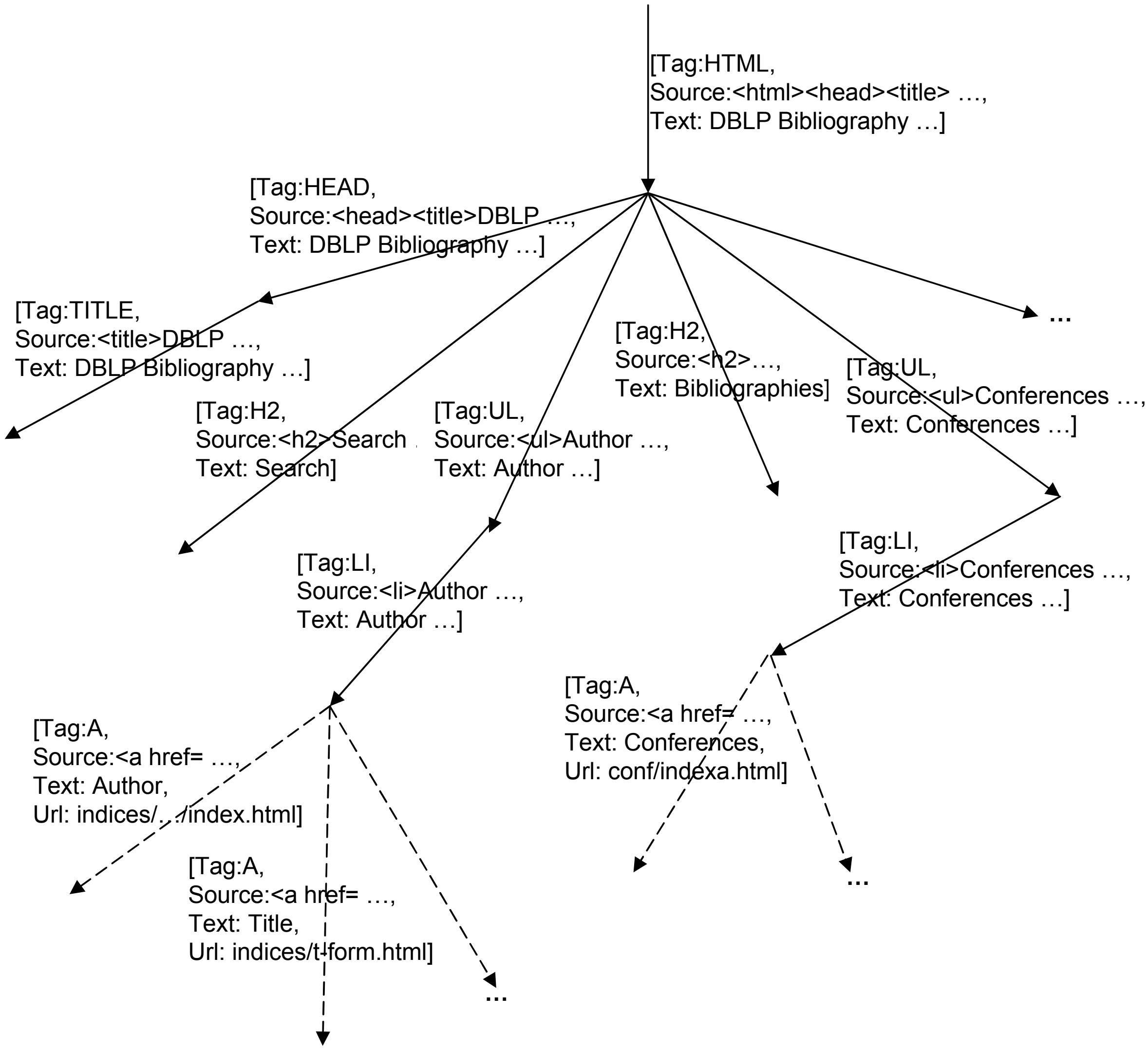


Figure 2.3: A Hypertree in WebOQL

The hypertree created from the above HTML code is shown in Figure 2.3. It has a number of internal arcs and external arcs that are labeled with three or four attribute-value pairs. The value of the Tag attribute is the HTML tag that this arc represents. The value of the Source attribute is the HTML source code between the beginning of the tag and the end of the tag, including the tag itself. The value of the Text attribute is the actual text information that is displayed on the screen. For example, for the arc labeled with the `<title>`, the value of the Source attribute is

"<title> DBLP Bibliography </title>". The value of the Text attribute is "DBLP Bibliography". All internal arcs have the above three labels. External arcs have an extra label, Url, that has the URL as its value.

Like WebSQL and W3QL, WebOQL is an SQL-like web query language. For example, the following query searches for the Text attributes of objects in page index.html with tag H2.

```
SELECT [ n.Text ]
FROM n in "index.html" via ^* [ Tag = "H2" ]
```

In this query, the "^" symbol is a predicate that is true if the arc is internal. The "*" symbol is an operator that applies the logical AND operation to two predicates. A query in WebOQL is a function that maps one hypertree to another hypertree. For example, the query result of the above example query is another hypertree similar to the hypertree shown in Figure 2.3 except that the arcs in the resulting hypertree are all labeled "[Text: ", followed by the actual Text attribute of objects that match the query conditions, followed by "]". Thus, WebOQL can also restructure HTML documents. This feature makes it quite different from WebSQL and W3QL. However, although WebOQL translates all HTML tags to its hypertree data model, it treats the HTML tags as textual content and uses them only for text matching. The intra-document structure is not considered.

2.2.4 WebLog

WebLog [27] is a declarative logic-based query language, which was inspired by SchemaLog [26], a logic language proposed for interoperability in multidatabase systems. WebLog uses deductive rules based on a conceptual model that represents the conceptual structures of HTML documents. WebLog defines "a group of related information" within an HTML page as a unit of related information called a *rel_infon.* Thus, an HTML document is a set of rel_infons. A rel_infon has several attributes. Two major attributes are strings "occurs" and "hlink". Other attributes correspond to HTML elements used to enhance the appearance of a web page, such as the elements associated with the `<title>` and `<b>` tags.

For example, consider the HTML source code for Ley's DBLP Bibliography page, as shown in Section 2.2.3. The text within a `<h2> ... </h2>` segment can be regarded as a rel_infon. The attributes are mapped to string values, except for the HLINK attribute, which is mapped to an *hlink-id*, a unique identifier associated with a hyperlink. For string processing, WebLog also provides built-in predicates and programming predicates, which are implemented by calling external programs. WebLog supports restructuring of the query result by defining the output format in the head part of a rule.

For example, the following WebLog query finds all hyperlinks in Ley's DBLP Bibliography page, as shown in Figure 1.1, and the titles of the web pages that those hyperlinks point to.

```
ans.html [title ⟶ "all links", hlink ⟶↠ L, occurs ⟶↠ T ]
    ⟵ ley_DBLP_url [hlink ⟶↠ L], href(L,U), U[title ⟶ T]
```

In this query, variable L ranges over all links in Ley's DBLP Bibliography page. Variable U is the URL that L points to. $href()$ is a built-in predicate that is used to navigate from page L to page U. Variable T is the title of page U. The query result is stored in an HTML document, called ans.html, with a title of "all links".

The data model of WebLog treats HTML elements other than TITLE and LINK as text for matching purposes only. For example, the following query searches for the years that papers about Coral (a deductive database system) were published in the VLDB Journal.

```
ans(Y) ⟵ ley_DBLP_url(U),
  U[title ⟶ "coral", occurs ⟶↠ S],
  substring(S, "VLDB Journal"),
  substring(S, Y), isa(Y, year)
```

In this query, U is the set of URLs on Ley's DBLP Bibliography page. The query goes through each URL in U to search for pages with titles that include the word "coral".

S is the textual content of the web page. The built-in predicate `substring(S, "VLDB Journal")` is used to search for the string "VLDB Journal" in S. The built-in predicate `isa(Y, year)` is used to determine if Y is an instance of the year type. In this example, `title` is an attribute that can be queried, while the other information S in page U is just a string that can be searched for certain keywords, such as "VLDB Journal" in the above example. The intra-document structure is for the most part not represented.

2.2.5 StruQL

StruQL [16, 15, 17] is the SQL-like web query language of the Strudel web site management system. It provides a data model in the form of a data graph to model all data sources uniformly, and it also provides a query language for both data integration and view definition. The *data graph* describes the logical structure of all information available at a site. It is a labeled directed graph similar to the Object Exchange Model (OEM) [40]. In this graph, nodes represent objects or values, and edges are labeled with attribute names. For example, nodes can represent integer, real, string, or Boolean values, while edges are typically labeled with strings. The Strudel data graph model also supports several atomic types that commonly appear on the web, such as URLs, PostScript files, images, and HTML files. In addition, the Strudel data model provides collections, where a *collection* is a set of nodes (objects).

For example, the following query searches for links with an attribute called "Conferences" in Ley's DBLP Bibliography page.

```
where HomePages(ley_DBLP_url), p → "Conferences" → q
collect ConferenceLinks(q)
```

HomePages is a collection, which in this case contains only one object, namely ley_DBLP_url. `p → "Conferences" → q` is a condition meaning that there exists an edge labeled "Conferences" from `p` to `q`. The query result is a new collection, called ConferenceLinks, which contains all the answers.

In STRUDEL, STRUQL serves two purposes: querying heterogeneous sources to integrate them into a data graph, and querying this data graph to produce a site graph. The limitation of STRUQL is that it only considers links in the HTML documents. The issue of intra-document structure for the most part is not addressed.

2.2.6 NetQL

Guan et al. developed and implemented an approach, called NetQL, for using structure-based querying to find web pages [20, 29]. The NetQL approach has three main features. First, NetQL permits similarity based matching on words contained in text by using natural language processing to find the stems of words, such as "program" from "programs", and WordNet [37] to identify synonyms.

Secondly, NetQL allows restrictions to be placed on the time required to answer a query, the number of results returned, the number of levels of hyperlinks to follow, and the portion of the web to search. Thirdly, NetQL converts HTML pages to hypertrees to allow querying within the pages.

The last of these features is most relevant to this thesis. In more detail, information is extracted by treating each web page as either an index page or a content page or a hybrid mixture of both. Guan et al. do not state how to distinguish the three types of pages, but it may be done based on the number of links present. Each of the three types of pages is converted to a tree, by applying a single conversion. The conversions are not applied recursively.

An index page is converted to a tree using the appearance of any `<h1>`, `<h2>`, ..., `<h6>`, `<menu>`, or `<em>` tag as an indication that a new section is beginning. Each such section is converted to a top-level branch of the tree, and any list following such a tag is converted to a subtree under the main branch. The label associated with the tag is used to label the branch of the tree.

A content page is converted to a tree using the appearance of any `<p>`, `<hr>`, or `<br>` tag as an indication of a new section, which is converted to a top-level branch of the tree. If a paragraph begins with a `<strong>`, `<b>`, or `<i>` tag, the associated text is used to label the branch of the tree, otherwise, apparently, the complete text is used to label the branch.

A hybrid page is first divided into sections, based on the appearance of `<p>` and `<br>` tags. Each section is treated in the same manner as a page by classifying it either as an index section, a content section, or a hybrid section, and processing it as described above.

Some conversions are also performed for HTML tables. In particular, the `<th>`, `<tr>`, and `<td>` tags are used to identify the major components of a table and convert them to a tree with a branch for each row of the table. Embedded tables are not handled.

After converting HTML documents to trees, NetQL can search them for exact or approximate matches. As output, NetQL produces an HTML document beginning with the text "The search result is:" that contains links to pages containing relevant information. NetQL does not extract the relevant information from the pages. Overall, NetQL extracts some intra-document structure, but it does not handle recursive structures such as subsections within sections or tables embedded in other tables.

2.2.7 Summary

Table 2.4: Summary of Web Query Languages

Language	Data Model	Inter-document Structure (LINK) Support	Intra-document Structure Support
WebSQL	relational	Yes	Partial (TITLE Only)
W3QL	graphs	Yes	Partial (TITLE Only)
WebOQL	hypertrees	Yes	No
WebLog	relational	Yes	Partial (TITLE Only)
StruQL	graphs	Yes	No
NetQL	hypertrees	Yes	One level

Table 2.4 presents a summary of the six web query languages discussed in detail in this section. As shown in the table, these web query languages use either a relational data model or a graph data model (hypertrees can be regarded as graphs at a higher

level). All six web query languages we studied in detail recognize links in HTML documents and use them as attributes, which enables them to query the inter-document structure of HTML documents. Three of these six languages recognize the TITLE HTML element and use it as an attribute on which to perform queries. Of these six languages, only WebOQL and NetQL recognize HTML elements other than TITLE. WebOQL uses such elements only for text matching purposes. NetQL extracts some intra-document structure, but it does not handle recursive structures. Thus, with these languages, little of the intra-document structure of HTML documents is taken into account for querying purposes.

The next chapter presents an approach that takes the intra-document structure into account when performing querying.

Chapter 3

Approach

In this chapter, we describe HTML-QL, a rule-based language for querying HTML documents. We also present the structure of HTML-CM, a conceptual model for HTML documents, and give examples of how to convert HTML web pages to HTML-CM web objects. The syntax of HTML-QL is briefly presented as well. Examples of how to utilize this query language to answer high level queries are provided. Based on HTML-CM, HTML-QL is able to query both the inter-document structure and the intra-document structure of HTML documents.

There are two logical components in the HTML-QL language, the language itself and the HTML-CM conceptual model that the language is based on. To use the HTML-QL language for querying, the HTML documents are first converted to web objects that are instances of the HTML-CM conceptual model, and then the queries are performed. In Section 3.1, the HTML-CM conceptual model is described, and in Section 3.2, we present rules for converting HTML documents to web objects defined in HTML-CM. In Section 3.3, we present HTML-QL and provide examples showing how to use HTML-QL to query both the inter-document structure and the intra-document structure of HTML documents.

3.1 The HTML-CM Conceptual Model for HTML

Liu and Ling proposed HTML-CM, a novel conceptual model [30, 31, 32] for HTML. Recall from Chapter 1 that a *conceptual model* is a data model of the conceptual structure within some data. It is a representation that explicitly shows the relationships among the data. A conceptual model for an HTML document explicitly represents the overall structure of the HTML document and provides high-level information about the structure of the data in the HTML document. Liu and Ling argue that HTML-CM allows the information in an HTML document to be queried and viewed in a way that is close to human conceptualization and visualization. HTML-CM is defined by a set of the correspondences between elements of HTML documents and HTML-CM web objects. An instance of the HTML-CM is a list of web objects that together describe the conceptual structure of an HTML document.

Let us first define some terminology adapted from Liu and Ling's work [30, 31, 32]. Let $\mathcal{U}$ be a set of URLs, and let $\mathcal{C}$ be a set of constants.

A *lexical object* is a string of constant text such as `"The City of Calgary"` or `"digital cell phone"`. The contents of the set $\mathcal{C}$ are lexical objects.

A *linking object* consists of two components, a label followed by a URL. The label is the description of the URL, and the URL points to the web page where detailed information of description is provided. Two examples of linking objects are `Contact<contact.html>` and `About Google<http://www.google.ca/intl/en/about.html>`.

An *attributed object* also consists of two components, an attribute followed by the value of the attributed object, with an arrow symbol in between. An example of an attributed object is `Title => Google`.

A *list object* is a list of any conformation of lexical objects, linking objects, or attributed objects, surrounded by braces. The following is an example of a list object:

```
{Contact <contact.html>,
About Google <http://www.google.ca/intl/en/about.html>}
```

3.2 Converting HTML Documents to HTML-CM Objects

By studying HTML documents, one may discover that most of them are constructed in a way that is close to the way people conceptualize the information while looking at the corresponding web page. For example, the order of items in an HTML document closely matches the order in which items are shown when this HTML document is browsed. Also, the hierarchically structured groups of items in an HTML file reflect the hierarchical structure of the information shown on the web page. To represent the underlying conceptualization of the information, we use the HTML-CM model. Liu and Ling devised a set of rules that can be used to convert an HTML document to an instance of the HTML-CM data model. The conversion is based on HTML 4.01 [42].

Since our primary focus is the textual information that HTML documents deliver, the visualization tags in HTML, such as font type and size, styles, and colors, are ignored during conversion. While in general we assume that input HTML documents are syntactically correct, our system is robust enough to handle problems such as the absence of one of a pair of tags.

The definition of the conversion rules for HTML-CM can be found in [30, 31, 32]. For easy reference, they are also given in Appendix B. The following examples cover and explain all the rules.

A typical HTML document starts with an `<html>` tag and ends with an `</html>` tag. Most HTML tags are paired, such as `<html>` `</html>` and `<body>` `</body>`. An HTML document usually consists of a *head*, embedded between `<head>` and `</head>` tags, followed by a *body*, embedded between `<body>` and `</body>` tags. The head usually states the title of the document, which is specified between a pair of `<title>` and `</title>` tags. The body is the content of the document. Detailed information, such as text and tables, appears in the body.

The major HTML elements that provide conceptual structure in HTML documents are titles, sections, paragraphs, links, images, lists, tables, forms, and framesets. In the remainder of this section, we describe how each is converted to an HTML-CM web object.

Titles: The title located in the head of the HTML document is converted to an attributed object in the HTML-CM conceptual model. For example, consider the following HTML code.

```
<title>The City of Calgary</title>
```

The converted attributed object is as follows.

```
Title ⇒ The City of Calgary
```

Sections: A section in an HTML document is converted to an attributed object. The heading of the section is converted to an attribute and the rest of the section is converted to a value for this attribute. Consider the following HTML code:

```
<h2>Mission</h2>
To provide outstanding services to the community.
```

This code is converted to the following attributed object.

```
Mission ⇒ To provide outstanding services to the community
```

Paragraphs: A paragraph is the content after a `<p>` tag up to a `</p>` tag or another `<p>` tag. As mentioned in Chapter 1, the `</p>` tag is optional. If a paragraph has some boldface or italic words or some words preceding a colon at the beginning, then the paragraph is converted to an attributed object, and otherwise it is converted to a list object. A sequence of multiple paragraphs is converted to a list object too. The following is an example.

```
<p>
<b>Location:</b> Northern North America
<p>
<b>Geographic coordinates:</b> 60 00 N, 95 00 W
```

The converted list object is as follows.

$\{$ `Location` $\Rightarrow$ `Northern North America,`
`Geographic coordinates` $\Rightarrow$ `60 00 N, 95 00 W` $\}$

Links: A link in an HTML document is converted to a linking object, which consists of a label and an anchor. The label is the text presented and the anchor is the destination of the link, which can refer to another HTML document, a PDF file, a video/audio clip, etc. The following is an example.

```
<a href="intro.html">Introduction</a>
```

The converted linking object is as follows.

```
Introduction <intro.html>
```

Images: A reference to an image document in an HTML document is converted to an attributed object using the *Image* keyword as the attribute and the destination of the link to the image as the value. The following is an example.

```
<img src=cityhall.jpg alt="Municipal Building">
```

After conversion, the following object is obtained.

`Image` $\Rightarrow$ `Municipal Building <cityhall.jpg>`

Lists: There are three types of lists in HTML: ordered lists, unordered lists and definition lists. An *ordered list* is a list between `<ol>` and `</ol>` tags. An *unordered list* is a list between `<ul>` and `</ul>` tags. Each item in an ordered list or unordered list is specified by a non-paired `<li>` tag. A *definition list* is a list between `<dl>` and `</dl>` tags. Information after a non-paired `<dt>` tag specifies the term that is being defined, and the content after a non-paired `<dd>` tag provides the definition. A list can be nested inside another list.

The following piece of HTML code shows all three types of lists.

```
<ol>
<li>Artificial Intelligence
<ul>
<li>Cognitive Science <li>Linguistics <li>Reasoning
</ul>
<li>Database Systems
<ul>
<li>Query Processing  <li>Data Models <li>Active DB
</ul>
</ol>
<dl>
<dt>General Information
<dd>The department was established in 1970.
<dt>Programs of Study
<dd>It offers M.Sc. and Ph.D. degrees in Computer Science
<dt>Financial Support
<dd>A variety of scholarships are available
<dt>Facilities
<dd>The research labs have all kinds of state-of-the-art equipment
</dl>
```

Items of ordered or unordered lists are converted to constants. A nested list is converted to an attributed object with the outermost items as attributes and the innermost items as values. A definition list is converted to a list object, which consists of attributed objects, by using the information immediately after the `<dt>` tags as attributes and the information immediately after the `<dd>` tags as values. Therefore, for the example HTML code, the following objects are produced.

```
{ Title ⇒ CS Department Research
  Research Areas ⇒ {
    Artificial Intelligence ⇒ {
      Cognitive Science,
      Linguistics,
      Reasoning}
    Database Systems ⇒ {
      Query Processing,
      Data Models,
      Active DB}}
  General Information ⇒ The department was established in 1970,
  Programs of Study ⇒ It offers M.Sc.  and Ph.D. degrees in ...,
  Financial Support ⇒ A variety of scholarships are available,
  Facilities ⇒ The research labs have all kinds of ...
}
```

Tables: Tables in HTML are used either to display tabular information or to provide visitors with a better visual layout. A table begins with a `<table>` tag and ends with a `</table>` tag. A table that has a caption embedded between `<caption>` and `</caption>` tags is converted to an attributed object by using the caption as the attribute and the rest of the table's contents as the value. The content of the table, ignoring the caption, consists of rows and columns of cells. The `<tr>` tag shows the beginning of a new row. The `<td>` tag shows the beginning of a cell. The `</tr>` and `</td>` tags can be used to end rows and cells, respectively, but they are not mandatory. Both rows and columns can have headings that are specified by `<th>` tags. The set of cells in a row are converted to either a single list object if there are no row headings, or an attributed object if there are row headings. If a table has column headings, each cell is converted to an attributed object using the heading of the column as the attribute and the content of the cell as the value. If a table does not have column headings, each cell is converted to a lexical object.

The following example gives an HTML table with captions and headings.

```
<table>
<caption align = top> Video Format </caption>
```

```
<tr>
<td> <br> <th> Resolution <th> Size <th> Codec
<tr> <th> DVD <td> 720x480 <td> 70M/min <td> MPEG-2
<tr> <th> VCD <td> 352x240 <td> 10M/min <td> MPEG-1
<tr> <th> SVCD <td> 480x480 <td> 20M/min <td> MPEG-2
<tr> <th> miniDV <td> 720x480 <td> 215M/min <td> DV
</table>
```

After conversion, this table appears as follows.

```
Video Format ⇒{
  DVD   ⇒{Resolution ⇒720x480, Size ⇒70M/min, Codec ⇒MPEG-2},
  VCD   ⇒{Resolution ⇒352x288, Size ⇒10M/min, Codec ⇒MPEG-1},
  SVCD  ⇒{Resolution ⇒720x480, Size ⇒20M/min, Codec ⇒MPEG-2},
  miniDV⇒{Resolution ⇒720x480, Size ⇒210M/min,Codec ⇒DV    }
  }
```

The following example gives the HTML code for a one row, three column table without captions or headings.

```
<h2 align=center> Research Interests </h2>
<table border="0" cellpadding=3 cellspacing=3 align=center>
<tr>
<td colspan=50% align=left>
<a href="urban.html">Urban Planning</A>
<td colspan=50% align=left>
<a href="3d.html">3D Visualization</A>
<td colspan=50% align=left>
<a href="gis.html">GIS</A>
</table>
```

After conversion, this table appears as follows.

```
Research Interests ⇒{
  Urban Planning <urban.html>,
    3D Visulization <3d.html>,
    GIS <gis.html>
  }
```

Forms: An HTML form is used to display data and receive input from the user. A form begins with `<form>` and ends with `</form>`. An HTML form is converted to a single attributed object. HTML forms have many special elements called *controls*. Some example controls are `text input`, `checkbox`, `button`, `radio button`, and `menu`. Although the visualizations of these controls are quite different, they all have three common attributes, specifically, names, values, and types. The *name* specifies the data being displayed as output, the *value* is the data that are stored in the HTML document, and the *type* determines the type of value being received as input.

The two methods of sending information from a form to the server are `GET` and `POST`. The major difference between them is the way that they organize and send information to the server. The `GET` method sends the information as name-value pairs. It has a limitation on how long the string can be, which makes it good for sending short information. The `POST` method first places the name-value pairs together in a file and then sends the file to the server. There is no limit on the size of the file.

The two special types of buttons in HTML forms are `SUBMIT` and `RESET`. Activating a `SUBMIT` button sends information entered so far on the form, while activating a `RESET` button removes any information entered by the user from the form and prepares the form for new entries.

A `text input` control in an HTML form is specified by the `<input ...>` tag and is converted to an attributed object whose value consists of a list of attributed objects. For example, consider the following portion of a form.

```
<form method=post action="script"><br>
<p>Username:
<input type="Text" name="Name" value="Enter a name here">
</form>
```

From this portion of a form, the following objects are obtained.

```
Form ⇒ {
   Text Field ⇒ {
      Label ⇒ Username,
      Name ⇒ Name,
      Type ⇒ Text,
      Value ⇒ Enter a name here }
}
```

A `radio button` control in an HTML form provides a selection of options from which a visitor can only select one. A group of adjacent radio buttons is converted to an attributed object using the `radio button` keyword as the attribute. The value of the object consists of three attributed objects which have the `Label`, `Name`, and `Option` keywords as their attributes. The title or description of the group of buttons is converted to the value of the Label attribute. The value of the Name attribute is determined by the value associated with the `Name` keyword in the HTML code. The value of the Option attribute consists of another list of pairs of attributed objects, one of which has a Label attribute with the label displayed to a user as the value, and the other has a Value attribute with information specified by the Value keyword in the HTML code as the value.

Consider the following HTML code fragment.

```
<p>Computer:
<input type = "radio" name = "kind" value = "desktop">Desktop
<input type = "radio" name = "kind" value = "laptop">Laptop <br>
```

From this fragment, the following objects are obtained.

```
Radio Button ⇒ {
   Label ⇒ Computer,
   Name ⇒ Kind,
    Options ⇒ {
        {Label ⇒ Desktop, Value ⇒ desktop},
        {Label ⇒ Laptop, Value ⇒ laptop}}
}
```

A checkbox control in an HTML form allow multiple choices. A checkbox is similar to a radio button control, but it allows several options to be selected simultaneously instead of only one. A checkbox is converted to a web object similarly to how a radio button is converted.

A Menu control provides functionality similar to that of a radio button or a checkbox, but it uses much less screen space because it hides its items in a click open menu. In HTML, a Menu is specified as a series of options embedded between <select> and </select> tags.

An example of a Menu is as follows.

```
<p>Age Category:
<select name = "Category">
<option value = "teenager">13-19
<option value = "adult">20-60
<option value = "senior">over 60
</select>
```

This menu is converted to the following objects.

```
Menu ⇒{
    Label ⇒Age Category,
    Name ⇒Category,
    Options ⇒{
        {Label ⇒13-19, Value ⇒teenager},
        {Label ⇒20-60, Value ⇒adult},
        {Label ⇒over 60, Value ⇒senior}
}
```

Documents: We classify HTML documents as either regular or frame-based. A *frame-based HTML document* is a type of HTML document that does not have a body. Instead, it has one or more *framesets*, each of which consists of several *frames* that each link to an HTML document. For the purpose of this discusssion, all other documents are regular documents.

Frame-based documents: A frame-based HTML document is converted to a single list object, containing attributed objects. If a title is present in the frame-based document, it is converted to an attributed object. Each frameset as a whole is converted to an attributed object. Each frame in the frameset is converted to a linking object. For instance, consider the following HTML document.

```
<html><head><title> A Frameset Document </title></head>
<frameset cols="20%,80%">
   <frame name="content1" src="content_of_frame1.html">
   <frame name="content2" src="content_of_frame2.html">
</frameset>
</html>
```

This document is converted to the following list object in an HTML-CM model.

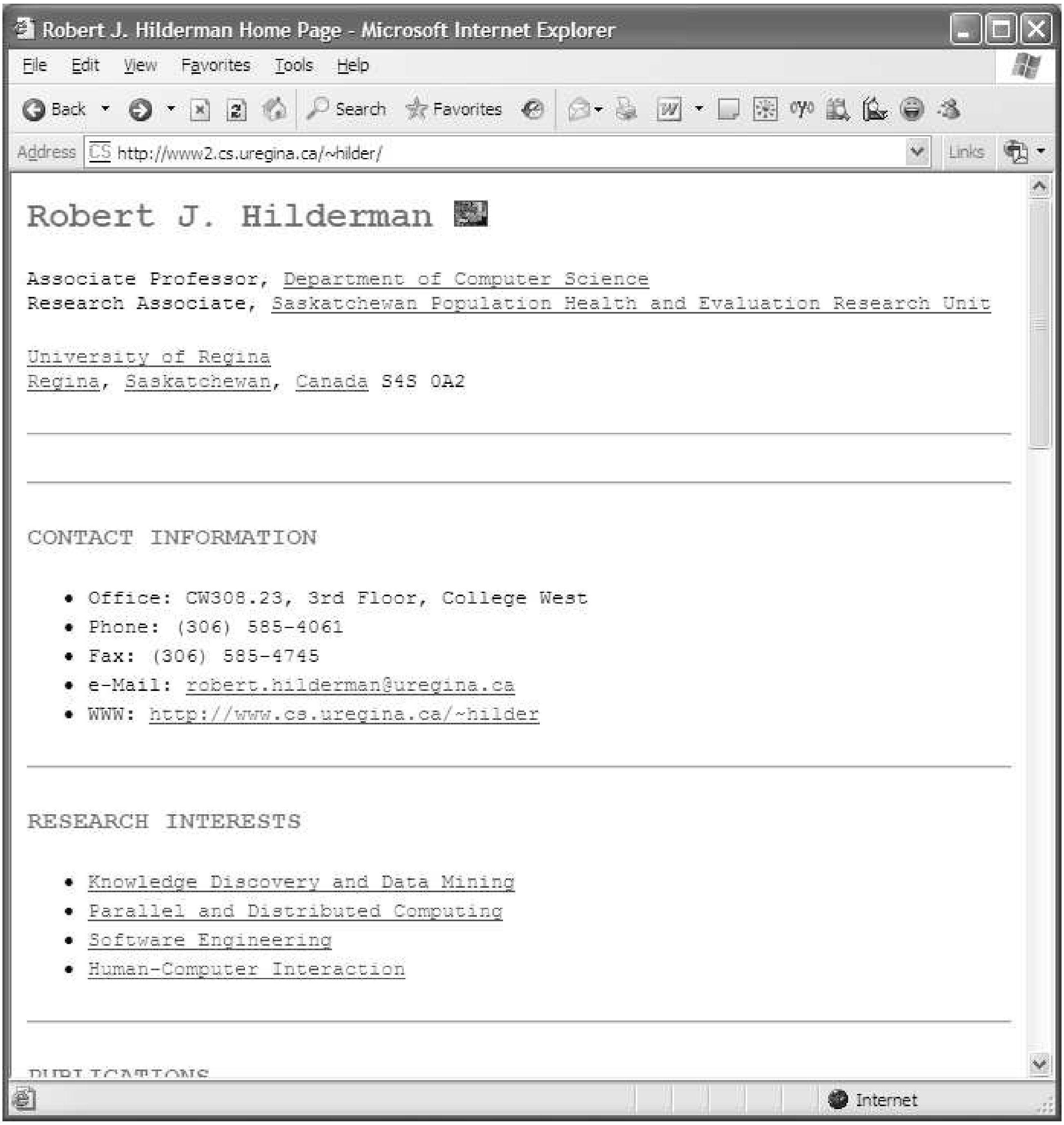
```
{
  Title ⇒ A Frameset Document
  Frameset ⇒
    {content1 <content_of_frame1.html>}
    {content2 <content_of_frame2.html>}
}
```

Regular documents: After all pieces of an HTML document are converted to various web objects, a single list object is created from these web objects. Thus, the overall form of the HTML-CM structure corresponding to a regular HTML document is a list object. Figure 3.1 shows Dr. Robert J. Hilderman's homepage at http://www2.cs.uregina.ca/~hilder. It is converted to the following object.

```
{
  Title ⇒ Robert J. Hilderman Home Page
  Robert J. Hilderman ⇒ {
    photos/me.jpg<http://www2..../photo_page.html>
    ...
  }
  CONTACT INFORMATION ⇒ {
    Office ⇒ CW308.23, 3rd Floor, College West
    Phone ⇒ (306) 585-4061
    Fax ⇒ (306) 585-4745
    e-Mail ⇒ robert.hilderman@uregina.ca <mailto:...>
    WWW ⇒ http://www.cs.uregina.ca/ hilder <http:...>
  }
  RESEARCH INTERESTS ⇒ {
    Knowledge Discovery and Data Mining <http://...>
    Parallel and Distributed Computing <http://...>
    Software Engineering <http://...>
    Human-Computer Interaction <http://...>
  }
  ...
}
```

3.3 A Rule-based Query Language for HTML Documents

HTML-QL is a rule-based query language for HTML documents. It is a declarative language and is similar to logic programming languages such as Prolog [7]. Each query statement is specified as a rule, which consists of two parts, a rule body and a rule head. The *rule body* specifies the source of the information, while the *rule head* specifies the structure of the query result. HTML-QL employs the mechanism of variable binding to specify the result. By binding each variable with the appropriate


Robert J. Hilderman Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://www2.cs.uregina.ca/~hilder/
Links
Internet

Robert J. Hilderman

Associate Professor, Department of Computer Science
Research Associate, Saskatchewan Population Health and Evaluation Research Unit

University of Regina
Regina, Saskatchewan, Canada S4S 0A2

CONTACT INFORMATION

- Office: CW308.23, 3rd Floor, College West
- Phone: (306) 585-4061
- Fax: (306) 585-4745
- e-Mail: robert.hilderman@uregina.ca
- WWW: http://www.cs.uregina.ca/~hilder

RESEARCH INTERESTS

- Knowledge Discovery and Data Mining
- Parallel and Distributed Computing
- Software Engineering
- Human-Computer Interaction

PUBLICATIONS

Figure 3.1: Dr. Robert J. Hilderman's Homepage

value retrieved from an HTML document, HTML-QL can specify a result that is consistent with the structure specified in the rule head.

Considering only data, an HTML document can be regarded as a deductive database, consisting of two parts, an extensional database, which stores the facts, and an intentional database, which is created by applying the rules in the extensional database. A major difference between query processing in HTML-QL and Prolog, which also uses a deductive database, is the mechanism by which variables are bound. In Prolog, after each variable in the body is bound using a fact, a result is obtained instantly in the head. However, with HTML-QL, because reconstructions are needed to create the result, the head of the rule is not decided until all necessary information for building the result is found. The goal of HTML-QL is to specify, in a declarative way, the extracting of structure from HTML documents.

Logical variables are used in HTML-QL for querying and constructing the result. Terms are defined in HTML-QL to represent the various objects in HTML-CM that were specified in the previous section. A *primary variable* in HTML-QL consists of a dollar sign symbol ($) followed by a possibly empty string. Examples are `$X`, `$`, and `$Label`. An anonymous primary variable is denoted by a lone $ symbol or a "*" symbol. One type of HTML-QL term is defined for each type of object in HTML-CM. A *lexical term* represents a lexical object, which is a constant string. For instance, `Faculty` is a lexical term. A *linking term* represents a linking object. Examples of linking terms are `Faculty <fac.html>` and `$Label <$URL>`. The components of terms such as anchors or labels can be either variable or constant. A linking term without a URL is called a *label term.* A linking term without a label is called an *anchor term.* For example, `$X<>` is a label term while `<$Y>` is an anchor term. An *attributed term* represents an attributed object. An example is `Answer⇒$X`. A *set term* represents a list object. `{$X,$Y}` is a set term. A *variable* can be either a primary variable, a lexical term, a linking term, an anchor term, a label term, an attributed term, or a set term.

If `U` is a variable or a URL and `T` is a term, then `U : T` is called a *positive expression.* Two examples of positive expressions are `u :  $X` and `$U : $V`. An expression containing a negated URL, i.e., ¬`U : T`, is called a *negative expression.* An example of a negative expression is ¬`u : Faculty<>`, which means that URL `u` cannot have an anchor term `Faculty<>`. The URL `u` in both the positive expression and negative expression functions is a predicate.

A *rule* consists of two parts, a head and a body, with a :– symbol in between. A *head* is a positive expression, and a *body* is a positive expression, a negative expression, an arithmetic expression, a string, or a set operation expression. A set of rules forms a *query*.

The following are some examples of queries. The DBLP web page located at http://www.informatik.uni-trier.de/∼ley/db is the source HTML document for all these examples (see Figure 1.1). To simplify the query statements, we use u_l to denote the actual long URL and u_o to denote the output URL, which is the name of a local file for these examples.

Example 1: Copy the content of the HTML document at u_l to a local file u_o.

u_o : `$X` :– u_l : `$X`

This query performs a strict data duplication from u_l to u_o, with no other data processing whatsoever. Therefore, the structure of the HTML code is not needed, so it is not retrieved.

To capture the structure of the HTML document at u_l, the following query is used.

Example 2: Retrieve the internal structure of the HTML document at u_l to a local file u_o.

u_o : {`$X`} :– u_l : {`$X`}

This query looks very similar to the previous one, but their meanings are quite different. This query returns the structure of the HTML document at u_l. The {\$X} term in the body of the query forces the retrieval of the structure of the source HTML document and the {\$X} term in the head of the query means that the query result is a copy of the {\$X} term in the query body. The query result is stored in a local file u_o.

Example 3: Obtain web objects under attribute "Title".

u_o : {`Answer` ⇒`$X`} :– u_l : {`Title` ⇒`$X`}

This query goes through every attributed object at u_l until it finds an attributed object with attribute "Title", and then returns the value of the object, which can be either one object or a list of objects. The result then is restructured to an attributed object with an attribute of "Answer" and the returned query result as the value. Attribute "Title" is special because title is also an HTML element. The next example is a more generic query.

Example 4: Obtain web objects under attribute "Search".

u_o : {`Answer` ⇒`$X`} :– u_l : {`Search` ⇒`$X`}

Similar to Example 3, this query seeks an attributed object with attribute "Search". This example shows HTML-QL's capability in extracting generic intra-document structure in addition to the attribute title.

Example 5: Retrieve all URLs under attribute "Search".

u_o : {`Answer`⇒{`$X`}} :– u_l : {`Search` ⇒{`<$X>`}}

This query goes through the HTML document at u_l and looks for an attributed object whose attribute is "Search". Then it looks for all URLs in the value part of the object and returns them.

Example 6: List labels of linking objects under the attribute "Search".

u_o : {`Answer` ⇒{`$X`}} :– u_l : {`Search` ⇒{`$X<>`}}

Similar to Example 5, this query returns only the labels of all linking objects that it find under attribute "Search".

Example 7: List all attributes at the first and second levels.

u_o : {`Answer` ⇒{`$X`}} :– u_l : {`$X` ⇒`$Y`}

u_o : {`Answer` ⇒{`$Y`}} :– u_l : {`$X` ⇒`$Y` ⇒`$Z`}

Detailed description of level can be found in Section 4.3.

Example 8: Search for the label "TODS" and return its URL.

u_o : {`Answer ⇒$X`} :– u_l : {`*TODS<$X>`}

This query examines all linking objects until it finds the first linking object whose label is "TODS". It then returns the URL of the object.

Example 9: Retrieve all URLs.

u_o : {`Answer⇒{$U}`} :– u_l : {`*<$U>`}

Similarly to the last example, this query examines all linking objects and collects all URLs.

Example 10: Obtain all URLs that are reachable from page u_l.

u_r : {`$X`} :– u_l : {`*<$X>`}

u_r : {`$X`} :– u_r : {`$Y`}, `$Y:` {`*<$X>`}

This query is recursive, because the result object specified in the first rule is used in the body of the second rule. It also involves more than one web object. This query returns the URL only. If one wishes to see the labels as well, the following query is required.

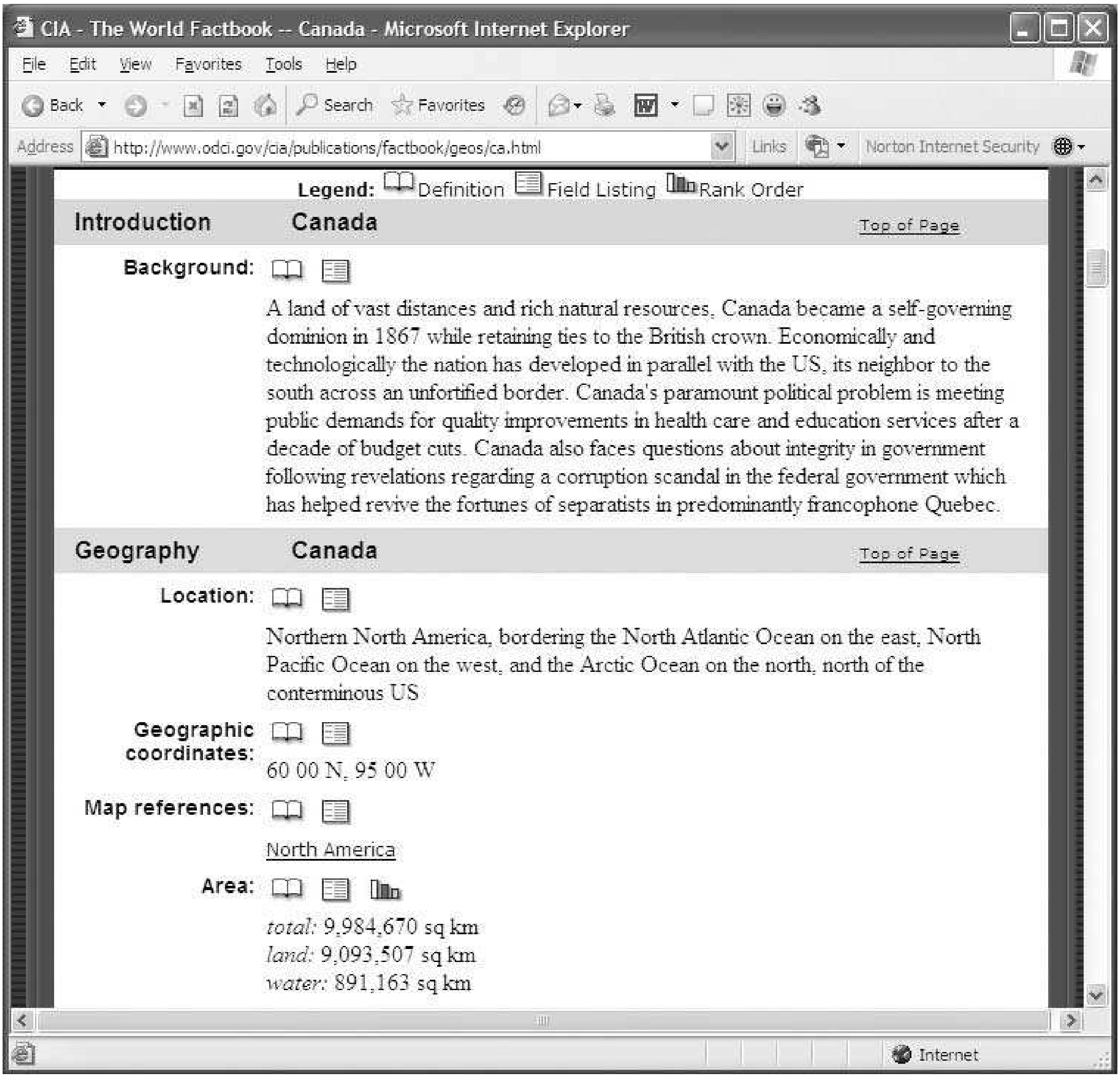
Example 11: Obtain the URLs and the corresponding labels for all URLs that are reachable from page u_l.

u_r : {`$L<$U>`} :– u_l : {`*$L<$U>`}

u_r : {`$L<$U>`} :– u_r : {`*$L<$U>`}, `$U:` {`*$L<$U>`}

The World Factbook at http://www.odci.gov/cia/publications/factbook/index.html provides information about every country in the world, including information, such as location, population, land boundaries, and gross domestic product. This web site is well suited to demonstrating the ability of HTML-QL to obtain useful information. Figure 3.2 shows a screen capture of the page about Canada, as displayed in a web browser. Figure 3.3 shows the converted web objects corresponding to the web page shown in Figure 3.2.

We now present four example queries based on the World Factbook HTML document. For the purpose of these examples, a *country* is defined as an administrative


CIA - The World Factbook -- Canada - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address http://www.odci.gov/cia/publications/factbook/geos/ca.html
Links
Norton Internet Security
Legend: Definition Field Listing Rank Order
Top of Page
Top of Page
Internet

Introduction Canada

Background:

A land of vast distances and rich natural resources, Canada became a self-governing dominion in 1867 while retaining ties to the British crown. Economically and technologically the nation has developed in parallel with the US, its neighbor to the south across an unfortified border. Canada's paramount political problem is meeting public demands for quality improvements in health care and education services after a decade of budget cuts. Canada also faces questions about integrity in government following revelations regarding a corruption scandal in the federal government which has helped revive the fortunes of separatists in predominantly francophone Quebec.

Geography Canada

Location:

Northern North America, bordering the North Atlantic Ocean on the east, North Pacific Ocean on the west, and the Arctic Ocean on the north, north of the conterminous US

Geographic coordinates:

60 00 N, 95 00 W

Map references:

North America

Area:

total: 9,984,670 sq km
land: 9,093,507 sq km
water: 891,163 sq km

Figure 3.2: Page about Canada in the World Factbook at www.odci.gov

```
http://www.odci.gov/cia/publications/factbook/geos/ca.html:  {
  Title ⇒CIA - The World Factbook -- Canada
  Canada ⇒{
    Background ⇒{A land of vast distance and ...}
    Geography ⇒{
      Location ⇒{Northern North America, ...}
      Geographic coordinates ⇒{60 00 N, 95 00 W}
      Map references ⇒{North America}
      ...
      Land boundaries ⇒{
        border countries ⇒{US}
      ...
  }}}}
```

Figure 3.3: Converted HTML-CM for the Factbook page on Canada

unit that has its own page in the World Factbook.

Example 12: Find countries that border both Germany and France. A country X is said to *border* another country Y if Y is one of the border countries listed in the World Factbook page for country X. Roughly, this statement corresponds to cases where two countries share a land border. A country does not border itself.

$$\begin{aligned} u_o : \{\texttt{Answer} \Rightarrow\{\texttt{\$N}\}\} \;\text{:-}\; \texttt{\$U} : \{&\texttt{Title} \Rightarrow\texttt{\$N},\\ &*\texttt{border country} \Rightarrow\{\texttt{Germany, France}\}\} \end{aligned}$$

Example 13: Find countries that border Germany but not France.

$$\begin{aligned} u_o : \{\texttt{Answer} \Rightarrow\{\texttt{\$N}\}\} \;\text{:-}\; \texttt{\$U} : \{&\texttt{Title} \Rightarrow\texttt{\$N},\\ &*\texttt{border country} \Rightarrow\{\texttt{Germany}\},\\ &\neg *\texttt{border country} \Rightarrow\{\texttt{France}\}\} \end{aligned}$$

Note that this query involves negation.

Example 14: Obtain pairs of countries that border exactly the same set of countries.

$$u_o : \{\texttt{Answer}\Rightarrow\{\{\texttt{Country1}\Rightarrow\texttt{\$N1, Country2}\Rightarrow\texttt{\$N2}\}\}\} \;\text{:-}$$

$$\texttt{\$U1} : \{\texttt{Title} \Rightarrow\texttt{\$N1, }*\texttt{border countries} \Rightarrow\texttt{\$Cs}\},$$

$$\texttt{\$U2} : \{\texttt{Title} \Rightarrow\texttt{\$N2, }*\texttt{border countries} \Rightarrow\texttt{\$Cs}\},$$

```
N1 != N2
```

Example 15: Find countries that directly or indirectly border Canada.

This query is recursive. It returns all the countries that directly or indirectly border Canada according to the World Factbook, which roughly means that they can be reached from Canada by ground.

$$\begin{aligned} u_r : \{\texttt{Answer}\Rightarrow\{\texttt{\$C}\}\} :- {} & \texttt{\$U} : \{\texttt{Title } \Rightarrow\texttt{Canada,} \\ & *\texttt{border countries } \Rightarrow\{\texttt{\$C}\}\} \\ u_r : \{\texttt{Answer}\Rightarrow\{\texttt{\$C}\}\} :- {} & u_r : \{\texttt{Answer } \Rightarrow\{\texttt{\$X}\}\}, \\ & \texttt{\$U} : \{\texttt{Title } \Rightarrow\texttt{\$X, } *\texttt{border countries } \Rightarrow\{\texttt{\$C}\}\} \end{aligned}$$

The result is shown in Section 4.3.

Because HTML-QL is a rule-based query language, a typical person may take some time to learn the syntax of the language and understand how to use the language to formulate complex queries. To used HTML-QL to query the intra-document structure of an HTML document, it is helpful to know something about the content and the structure of the document.

To summarize this chapter, the HTML-CM conceptual model provides a solid foundation to capture both the internal and external structure of HTML documents, and the HTML-QL query language takes advantage of the data model and provides functionalities to query both structures.

Chapter 4

System Architecture and Results

Based on the conceptual data model and web query language presented in the previous chapters, we developed a prototype system, called HTML-QS, for performing queries on HTML documents on the web. In this chapter, we present the design and implementation of HTML-QS. We introduce the high level architecture of HTML-QS in Section 4.1, and we discuss the implementation of some of the system's modular components in Section 4.2. In Section 4.3, we describe the results of applying HTML-QS to the example queries described in Section 3.3. HTML-QS fully implements Liu and Ling's model for HTML-CM and HTML-QL, except results for recursive queries are limited to a specified number of levels of recursion to constrain memory usage.

4.1 High Level System Architecture

The high level architecture of the HTML-QS system has four layers, as shown in Figure 4.1. The use of layers in the architecture makes the system easier to design, implement, test, and extend.

The top layer is the User Interface. A command line textual interface was implemented in order to provide a simple yet powerful way for the user to enter queries and display results. Query statements are stored in a text file, which can be edited using any text editing tool. Query results can either be saved to a text file or be displayed on the screen. From the command line, query statements are forwarded to the second layer of the system.

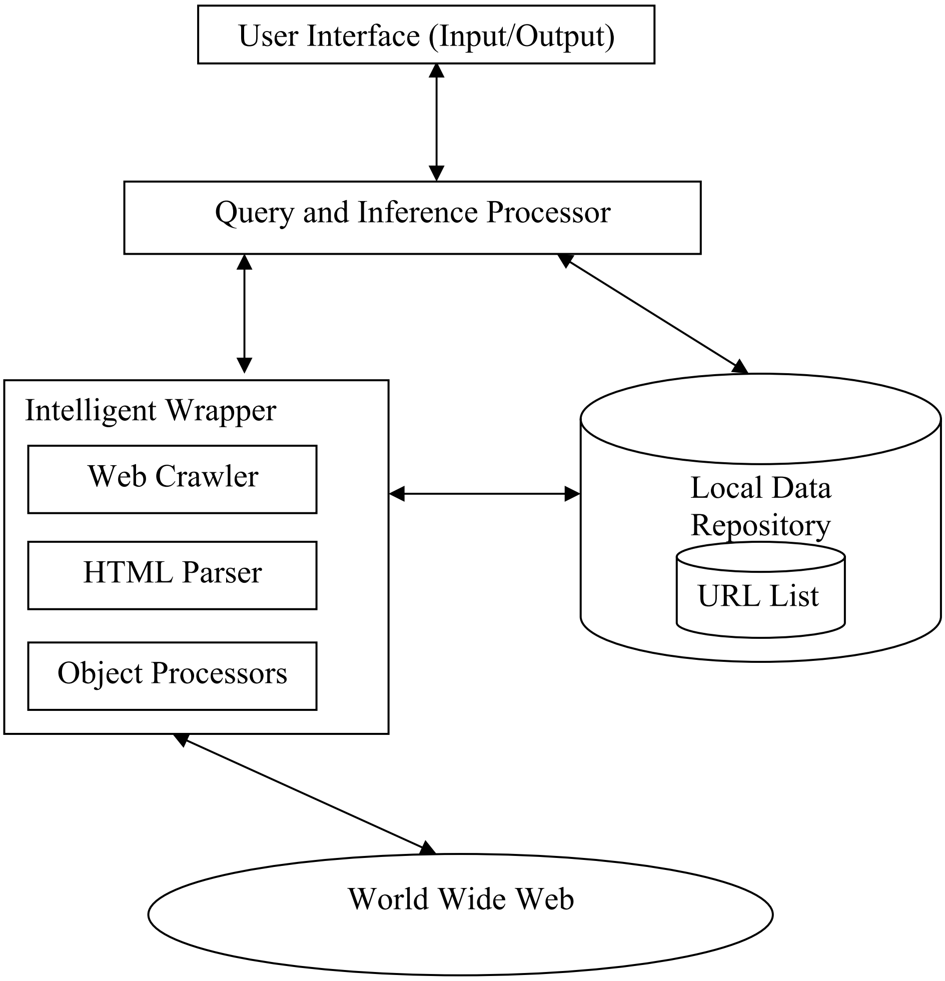


Figure 4.1: The HTML-QS System Architecture

The second highest layer of the system is the Query and Inference Processor. It is a communication layer between the User Interface and the third layer, the Intelligent Wrapper. It communicates with the User Interface layer to accept user inputs, validates query statements, visits the Internet or the Local Data Repository, retrieves web documents, and returns query results to the User Interface layer. For recursive queries, it uses iterative computation with a maximum depth to generate the result, as explained in Section 4.2.2.

The third highest layer consists of the Intelligent Wrapper and the Local Data Repository. It is the most complex component of the system. By design, the Intelligent Wrapper contains the Web Crawler module, the HTML Parsing module, and the Object Processors. The Web Crawler module in the wrapper accesses the World Wide Web and fetches HTML documents. The HTML Parser parses HTML documents, calls Object Processors, and converts HTML documents to HTML-CM web objects.

The Local Data Repository is the disk space where the query system stores data. It contains a URL list. The URL list provides URLs for the Robot to use while navigating the Internet. This list is updated by the Robot whenever it encounters a new URL.

The fourth layer of HTML-QS is the World Wide Web. This layer is the source of data for the system. The system can process standard HTML documents that conform to the HTML 4.01 specifications [42].

4.2 Implementation

In this section, we explain how HTML-QS is implemented using the Java programming language. This section is organized as follows. Section 4.2.1 explains why we chose Java. Section 4.2.2 discusses the implementation, including system procedures, system components, and important data structures.

4.2.1 Reasons for Selecting Java

We chose Java because it is object-oriented, platform independent, and convenient for network programming. We discuss these features in this section.

Object-oriented: Java is a pure object-oriented (OO) programming language. Using object-oriented design (OOD) makes all phases of software development more manageable. OOD has many advantages, such as encapsulation, inheritance, polymorphism, and abstraction. These features make it easier to produce modular and reusable code.

Platform Independent: Java is platform-independent, both at the source level and at the binary level. In other words, it runs on diverse operating systems and processors. Java's foundation class libraries make it easy to write code that can be moved from platform to platform without rewriting it to work with particular platforms. Java binary files are also platform-independent and can run on multiple platforms without the need to recompile the source, provided that the target operating system has available a compatible Java interpreter, known as a Java Virtual Machine (JVM).

Convenient for Network Programming: There are many pre-defined classes of network programming in Java's foundation class libraries, allowing programmers to concentrate only on program control, and not on the construction of basic data structures. Specifically, Java's foundation class libraries make writing networking code much simpler than with other well known programming languages. Java also provides an easy way to support client/server computing. Using a Java Applet greatly simplifies the development of online computing applications.

4.2.2 Modular Components

As described in Section 4.1, the HTML-QS system consists of four layers. The two most important layers are the Intelligent Wrapper and the Query and Inference

Processor. The Intelligent Wrapper consists of the Web Crawler, the HTML Parser, and the Object Processor. These three components work together in order to convert HTML documents to HTML-CM web objects. The Query and Inference Processor consists of the Query Parser, the Query Manager, and the Result Processor.

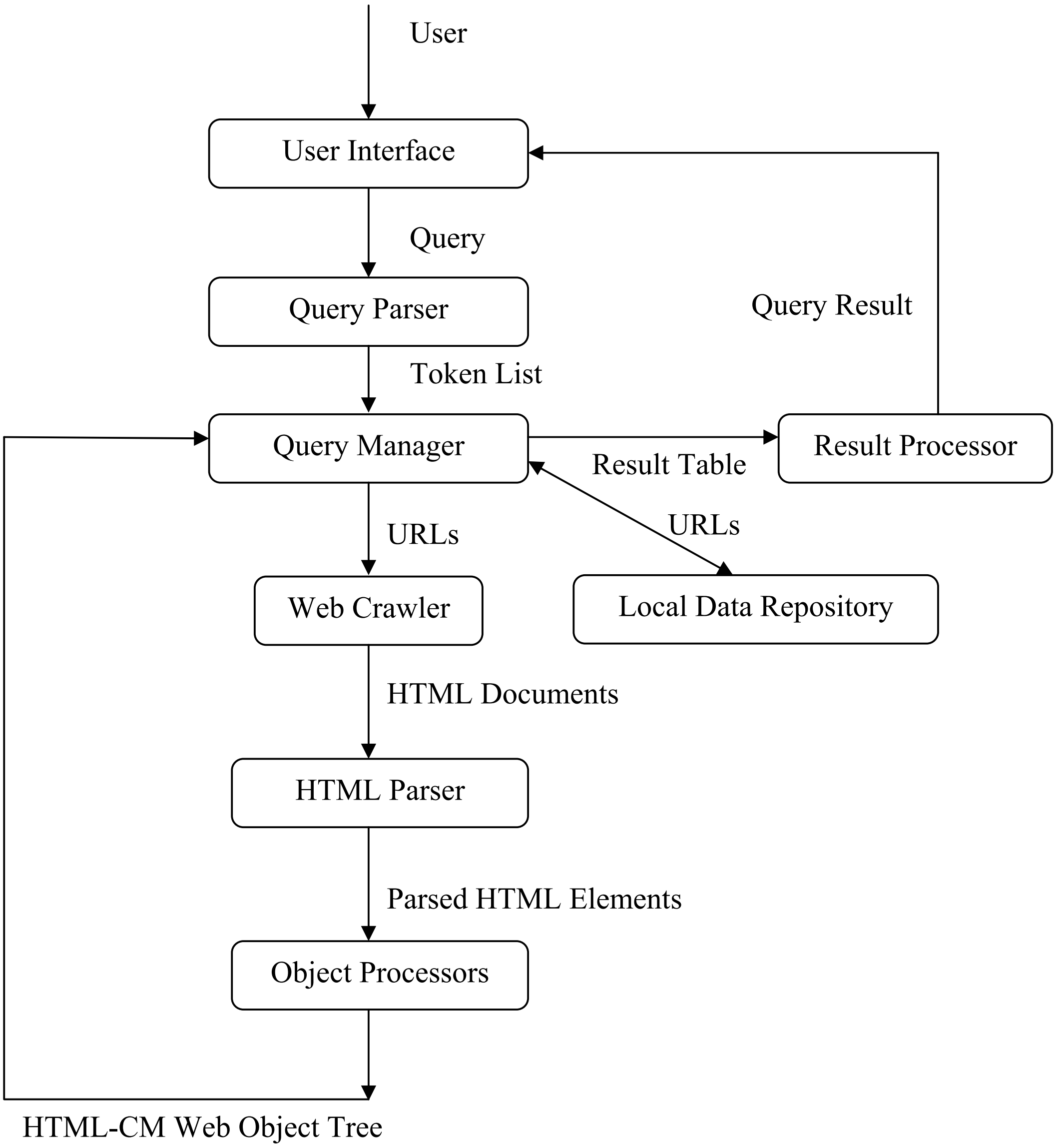


Figure 4.2: The Control Flow During Query Processing

The flow of control among these components is shown in Figure 4.2. First, a user enters a query through the User Interface. Then the query is sent to the Query Parser, which stores the parsed query in a token list. The Query Manager accesses the token list, checks whether the URLs exist in the Local Data Repository (LDR), and updates the LDR if it does not contain the URLs. Although conceptually the LDR is designed to store cached HTML documents, the current implementation has only a text file that stores the URLs that have been visited by the system. The Query Manager then calls the Web Crawler to obtain the HTML documents. Given a URL, a single thread of the Web Crawler visits the web page, retrieves an HTML document, and sends it back to the server. The HTML parser reads through the HTML documents, identifies the HTML elements, and calls the corresponding Object Processors to perform conversions. The converted HTML-CM objects are then sent to the Query Manager. Once the Query Manager has collected the necessary information, it begins to build a Result Table where the raw results of the queries are stored. The Result Table is sent to the Result Processor, which then displays the results to users.

In the remainder of this section, the components mentioned in the preceding paragraph are discussed in detail.

User Interface: As mentioned in Section 4.1, the User Interface layer is a simple command line interface where users can either enter a single query or edit a number of queries in a text batch file.

Query Parser: Given a query, the Query Parser first checks whether any syntax errors exist in the query. If not, it then builds token lists. Each token is a basic element of the query that cannot be divided any further. A list of pre-defined tokens is shown in Table 4.1. As well, a constant string is a token, and a variable is also a token. As mentioned in Chapter 3, a query consists of a set of rules, where each rule consists of a head and a body, and both of these consist of expressions. Therefore, two token lists are built, the head list and the body list. The head list is used to detect if a variable in the body is free or bound and to format the results for display. The body list defines the pattern that should be searched for during the query.

Query Manager Step 1: Collecting Information: The Query Manager has two major roles in HTML-QS. Here we discuss its role in collecting information. The

Table 4.1: Predefined Tokens in the Query Parser

Token Name	Description
CONST	Constant String
VAR	$X
STAR	*
LEFTDKH	{
LEFTJKH	[
RIGHTDKH	}
RIGHTJKH	]
IMPLY	=>
URL_FILE	http://www.........
LOCAL_FILE	Local file
LDR	Local data repository
MH	:
DH	,
TH	!
EQUAL	==
NOTEQUAL	!=
G	>
GE	>=
L	<
LE	<=

Query Manager is the query control center. Its input is the two token lists built by the HTML Parser. It reads through the body token list, obtains the URL token, sends out Web Crawlers to get HTML documents, and calls the HTML Parser, which in turn calls the Object Processors to convert HTML documents to HTML-CM web objects. The head token list is used in the final step when formatting the results for display.

Web Crawler: A Web Crawler receives a URL as input. It builds and opens a network connection with the URL and retrieves web pages. Some URLs may be inaccurate or correspond to web sites that have disappeared or servers that are temporarily or permanently inactive. The Web Crawler is threaded, with a new thread created for each URL. If a thread of the crawler cannot connect to the server within a specific waiting period, then it is terminated.

HTML Parser: The HTML Parser reads the HTML document retrieved by the Web Crawler. Not all HTML pages on the Internet conform to HTML 4.01 specifications. For example, web pages written by people often are missing some HTML tags, resulting in unpaired tags in the document. Secondly, because individual people have different ideas as to how to use HTML tags in order to achieve a desired appearance, some HTML tags may be used only for their visual presentation without regard for their original semantic meaning. The HTML Parser is designed to handle these difficulties. As it reads, the Parser removes comments and scripts, since they do not affect the retrieval of the internal structure of an HTML document, and then starts analyzing the HTML code. When the Parser encounters a beginning HTML tag, it determines which Object Processor should be selected. Table 4.2 gives the list of Object Processors defined in HTML-QS, along with the tasks they perform. The Parser keeps reading the HTML code until it finds the ending tag corresponding to the beginning tag. If the ending tag is not mandatory by the HTML standard, the parser stops at the next new beginning tag of any type. Some examples where the ending tag is not required are the `</p>` tag for paragraphs, the `</li>` tags for list items, and the `</td>`, `</tr>`, and `</th>` tags for tables, so in these cases, the parser looks for any type of tag. After the Parser has found two tags, it sends everything between them to the selected Object Processor to process the object. When the Object Processor has finished its task, the Parser resumes reading the HTML code until another beginning tag is found, and the process repeats. The Parser recognizes and processes one HTML element in each iteration until the end of the HTML document is reached.

Table 4.2: List of Object Processors

Object Processor	Task Description
bold	processes boldface `<b> ... </b>`
href	processes links `<a href=....> ... </a>`
image	processes images `<img ....>`
olist	processes ordered lists `<ol> ... </ol>`
ulist	processes unordered lists `<ul> ... </ul>`
table	processes tables `<table> ... </table>`
select	processes selects `<select> ... </select>`

Overall, step 1 of the Query Manager creates an HTML-CM web object tree for each HTML document. For example, consider the following HTML code:

```
<html><head>
<title> Computer Science Department </title></head>
<h2>People</h2>
<ul><li><a href=fac.html> Faculty </a>
<li><a href=staff.html> Staff </a>
<li><a href=studens.html> Students </a>
</ul>
<h2>Programs</h2>
<ul><li><a href= phd.html> Ph.D. Program </a >
<li><a href= msc.html> M.Sc. Program </a >
<li><a href= bsc.html> B.Sc. Program </a>
</ul>
```

```
<h2><a href=research.html> Research </a></h2>
</html>
```

The HTML-CM tree generated by the Query Manager for this code is shown in Figure 4.3.

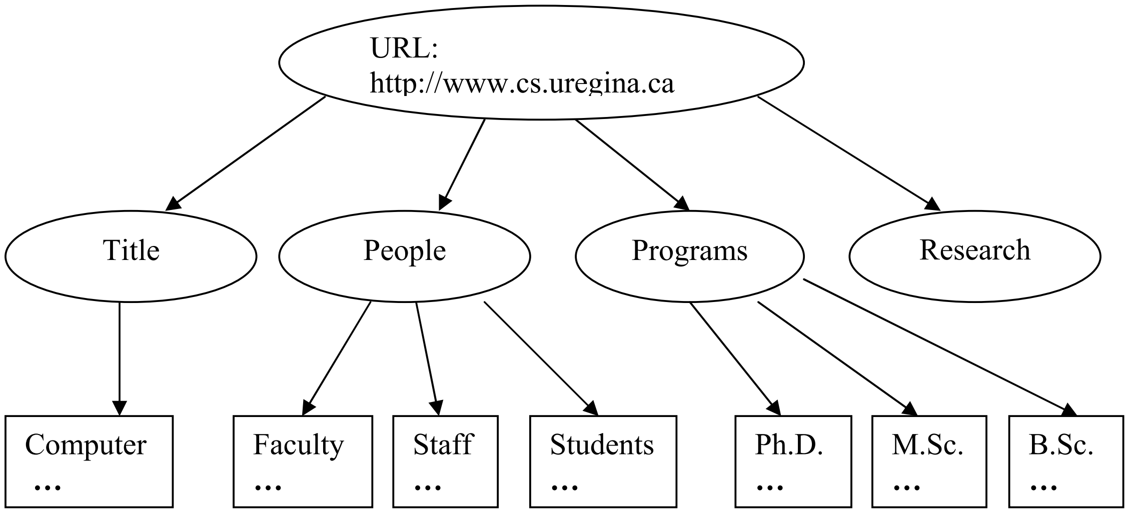


Figure 4.3: The HTML-CM Tree

The root of an HTML-CM tree is the URL of the web page that the tree represents. An attributed object is presented as a sub-tree with the attribute as the sub-root and the value as branches from the sub-root. Other objects are presented as leaf nodes.

Object Processors: The Object Processors implemented in the system are listed in Table 4.2. Most of the Object Processors function similarly. Based on the conversion rules introduced in Chapter 3, a processor converts a part of the input, which is a portion of an HTML document, to the corresponding part of an HTML-CM web object. Object Processors create branches of HTML-CM web objects, which are attached to the HTML-CM tree.

The most difficult part of converting HTML documents to HTML-CM web objects is the processing of HTML code for Tables. Processing tables is complicated, because a table can contain nested tables. The Table Object Processor builds a tree while processing a table, and adds a subtree to this tree for each nested table, as will be explained shortly. The nodes in the tree have the following structure:

```
class table_node {
  int attr;
  String cell_content;
  table_node next_cell;
  table_node next_table;
  table_node next_row;
}
```

The integer variable `attr` in the node structure identifies the type of this node. The types of attributes implemented in HTML-QS are TABLE, TR, TD, TH, and CAPTION. The tree for a two-dimensional table with one nested table is shown in Figure 4.4. The root of the table tree is given by the variable called Table_root. This variable points to a table node Row_1, which is the root of the first row in the table. For each row in the table, there is a root node called Row_j, which contains two pointers. One pointer points to the first cell $Cell_{j,1}$ in Row_j, which points to the next cell $Cell_{j,2}$ in the row, and so on. The other pointer in root node Row_j points to the root node Row_{j+1}, which points to the root node of the next row, and so on. Any two dimensional unnested table can be represented using this type of tree structure. If there are nested tables, a third type of pointer is used. For example, for a nested table contained in $Cell_{j,i}$, a pointer in this cell node points to the root of the first row of the nested table, which starts a subtree similar in structure to the main tree. The tree representing a table can be navigated easily and processed efficiently.

After the table tree is created, an algorithm, called Table_process, is used in order to process the table tree. Table_process is a recursive algorithm. It performs a top down traversal of the table tree, corresponding to processing the table row by row. The Table_process algorithm is given in pseudocode as follows.

```
Table_process(Table_element, Table_root)
{
```

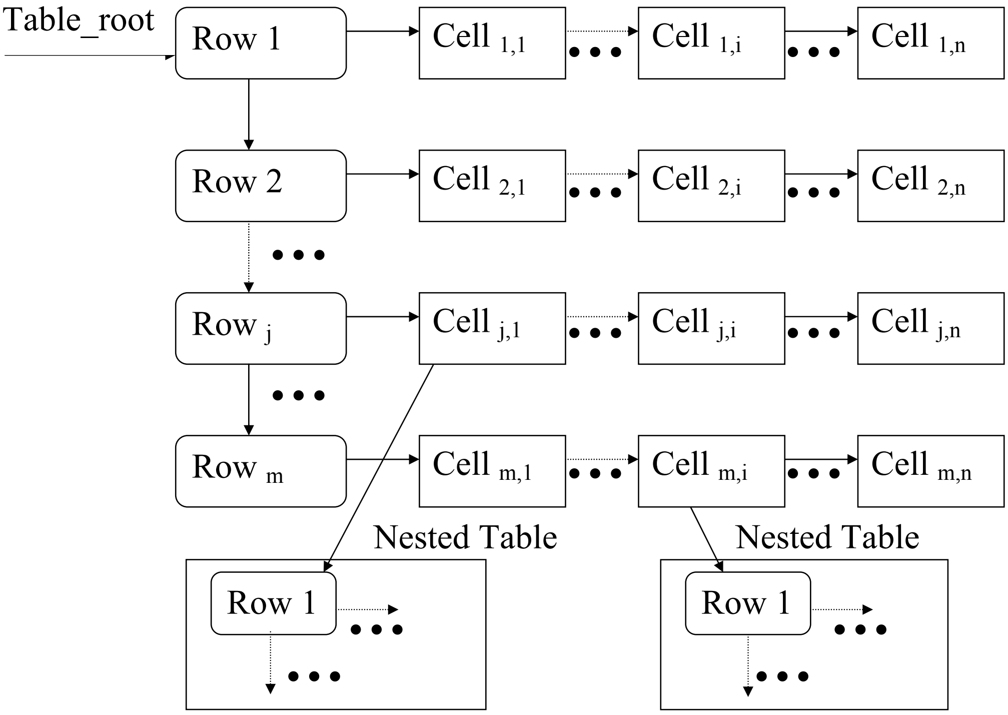


Figure 4.4: The Tree for Processing a Table

```
if (Table_root is NULL)
{
   return;
}
if (Table_element is a data cell)
{
   Process the content of the current cell;
   Table_process(Table_element->next_cell);
}
else if (Table_element is a cell at the beginning of a row)
{
   Table_process(Table_element->next_cell);
   Table_process(Table_element->next_row);
}
else if (Table_element contains a nested table)
```

```
      {
         Table_process(Table_element->next_table);
         Table_process(Table_element->next_cell);
      }
   }
```

An HTML element, such as TABLE, TR, TD, TH, or CAPTION, is stored as the content of a cell and is processed when the cell containing it is encountered during the traversal of the tree.

The tree for the table shown in Figure 4.4 is added as a branch of the overall HTML-CM tree for the HTML document containing the table.

Query Manager Step 2: Building the Result Table Once the HTML-CM tree has been created, the Query Manager starts building the Result Table. The data structure used to represent the Result Table is shown in Figure 4.5.

For every expression in the body of the query, the Query Manager creates an expression root node and an expression node in the Result Table. The expression root nodes form a dynamic linking table. The root pointer of the Result Table points to the first expression root node $Expression_1$. Each expression root node has two pointers, one pointing to the root node of the next expression in the linking table, and the other pointing to an expression node. In each expression node, there is another linking table, which consists of nodes for the variables used in the expression corresponding to that expression node. The variables in a single expression node must be unique, although the same variable may appear in more than one expression node. Each node for a variable contains a string representing the variable and a vector of search results. All the vectors together can be viewed as a table. This table is much like a table in a relational database. The combination of the i^{th} records in all vectors corresponds to a tuple. All of the tuples form a database table, which can be used in further query processing. For normal variables, the element of a vector stores the string result of a query. For constant expressions that are also considered to be special variables, the elements of each vector store a boolean value of `True` or `False`.

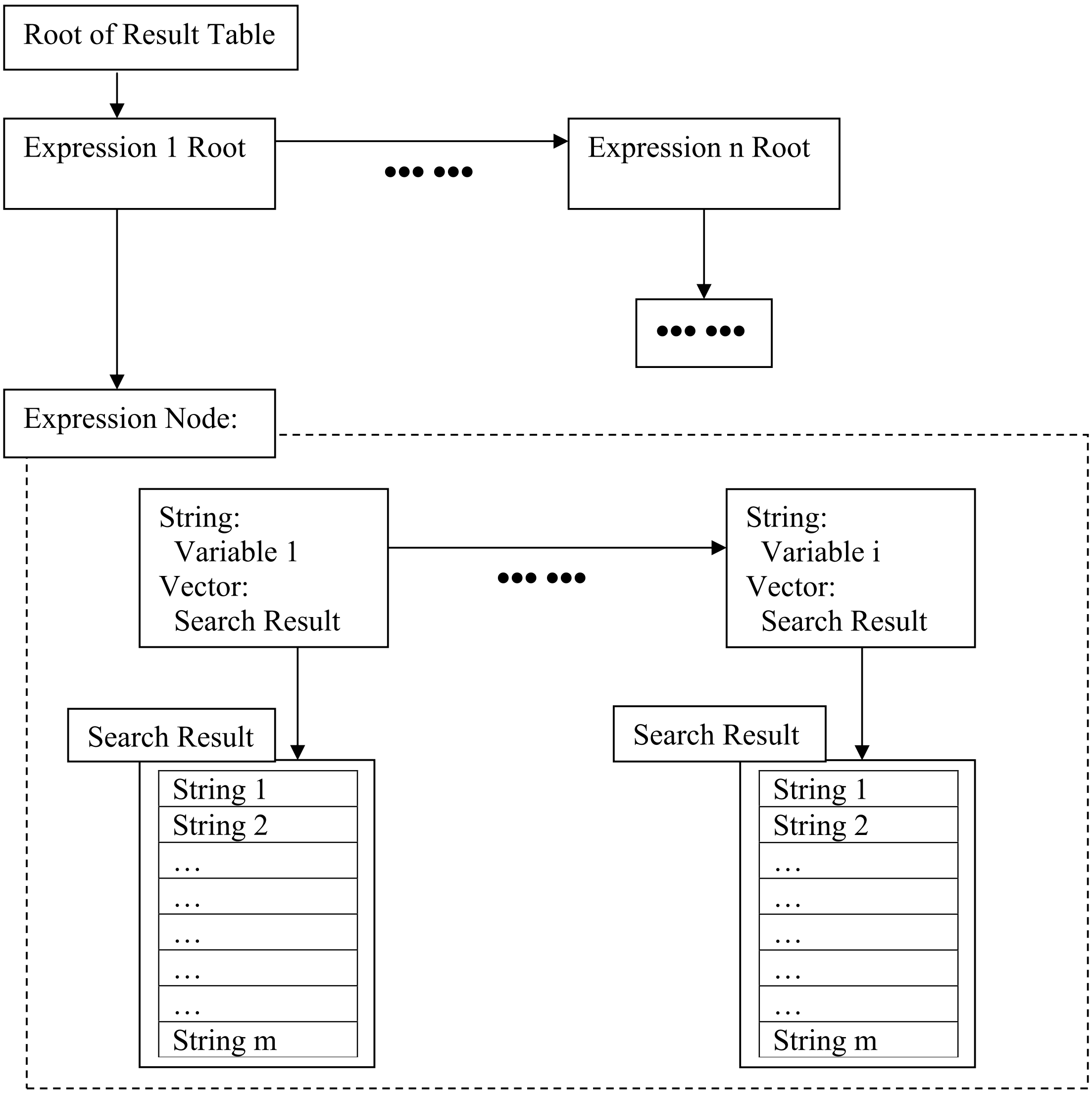


Figure 4.5: The Data Structure for the Result Table

Recursive Query: When a recursive query is performed, HTML-QS applies an iterative method to repeatedly execute the same query on different HTML documents until no more new results can be produced or the the maximum depth is reached. The *maximum depth* is the total number of levels of URL links that the system will try to reach from the current web page. For example, when querying the link objects that can be reached from the current web page, a default maximum depth of 5 is used to stop the system from going any further. We chose 5 for the default vaule, because that was the maximum depth that the HTML-QS system could run on the computer we used for testing before it ran out of memory. The maximum depth can be adjusted according to the hardware resources available on the computer.

Result Processor: Once the result table has been generated, the Query Manager passes it to the Result Processor. The Result Processor first prunes the result table by removing all tuples with False values. The Result Processor then starts building a Head Result List, which is a dynamic linking table that will later be used to construct a visual representation of the query result. The structure of the Head Result List is shown in Figure 4.6.

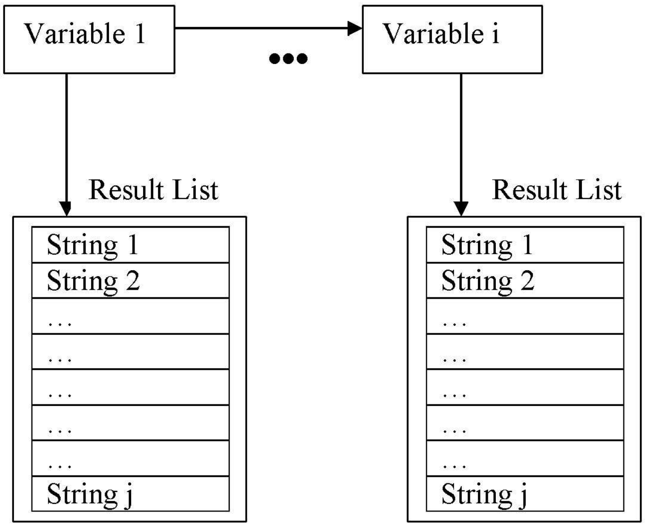


Figure 4.6: The Head Result List

For every variable shown in the query head, the Result Processor adds an item to the Head Result List. It then searches the result table, finds the matching variable, makes a copy of the result list for that variable, and links this list to the newly inserted item. The final step of the Result Processor is to display output from the Head Result List by iterating through all variables and result lists.

4.3 Evaluation and Comparision

In this section, we provide a comprehensive comparison and evaluation of the HTML-QS system. We discussed 15 example queries in Chapter 3 to show the capabilities of HTML-QL. To evaluate the HTML-QS system, we tested it on these 15 queries. The results of this testing are presented in this section. Where possible, we also present a comparison of these results with those obtained by the Google search engine and those that would have been obtained using the five of the web query languages described in Section 2.2, namely WebSQL, W3QL, WebOQL, WebLog, and StruQL.

Example 1: Copy the contents of the HTML document at www2.cs.uregina.ca/ ∼pwlfong to file a1.

```
a1 : $X :- http://www2.cs.uregina.ca/~pwlfong : $X
```

The following is the contents of file a1 after running this HTML-QL query:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
... ... ...
<HEAD><TITLE>Philip W. L. Fong</TITLE>
... ... ...
<BODY>
<H1>Philip W. L. Fong</H1>
<P>
<IMG alt=Photo src="http://www2.cs.uregina.ca/~pwlfong/photo.jpg">
... ... ...
</BODY></HTML>
```

When searching for "http://www2.cs.uregina.ca/∼pwlfong" in Google, Google returns a message indicating no information is available for the URL. However, it also suggests that the user should click on a provided link, pointing to http://www2.cs.uregina.ca/∼pwlfong, to visit the web page if the user thinks the URL is valid. None of the web query languages we described in Chapter 2 can perform this query.

To test HTML-QS on Examples 2 to 10, we use Ley's DBLP bibliography web page at http://www.informatik.uni-trier.de/∼ley/db as an example HTML document for the queries.

Example 2: Retrieve the intra-document structure of Ley's DBLP bibliography web page to file a2.

```
a2 : {$X} :- http://www.informatik.uni-trier.de/~ley/db : {$X}
```

The following is the resulting contents of file a2:

```
{
   Title=>DBLP Bibliography
   ...
   Search=>{
      Author<http://www.informatik.../a-tree/index.html>
      Title<http://www.informatik.../indices/t-form.html>
      Advanced<http://www.informatik.../indices/query.html>
   }
   Bibliographies=>{
      Conferences<http://www.informatik.../conf/indexa.html>=>{
         ...
      }
      Journals<http://www.informatik.../journals/index.html>=>{
         CACM<http://www.informatik.../cacm/index.html>
         TODS<http://www.informatik.../tods/index.html>
         ...
```

```
        }
    ...
}
```

HTML-QS converts the whole HTML document to a list object, but we only show some objects above to keep the results concise. Neither Google nor the five web query languages from Chapter 2 can extract this kind of intra-document structure from an HTML document.

Example 3: Obtain the web objects that occur under the attribute "title" and save the results to file a3.

```
a3 : {Answer=>$X} :-
     http://www.informatik.uni-trier.de/~ley/db : {title=>$X}
```

The following is the resulting contents of file a3:

```
{Answer=>
      DBLP Bibliography
}
```

In HTML-CM, the TITLE HTML element is converted to an attribute object. This query returns the value of the attribute object, which is a text string. In Google, one can search for some keywords that only appear in the title of the web pages. Therefore, if we search "DBLP Bibliography" and indicate that only titles are to be searched, Google returns web page http://www.informatik.uni-trier.de/∼ley/db as the first result. However, this querying with Google is done in the opposite way from what was defined. One needs to know the "value" of the Title first and then Google can find the web page. In HTML-QS, one can start with the web page and then query for the "value" of Title. The WebSQL, W3QL, and WebLog web query languages can perform a query on the Title attribute, while the five web query languages from Chapter 2 either can only search for `<title>` tag or cannot perform this type of query. WebSQL, W3QL, and WebLog can perform this query, because TITLE is a special

HTML element, However, they cannot perform queries on other attributes, as shown by the following example.

Example 4: Obtain the web objects that occur under the attribute "search" and save the results to file a4.

```
a4 : {Answer=>$X} :-
     http://www.informatik.uni-trier.de/~ley/db : {*search=>$X}
```

The symbol "*" before "search" means HTML-QS will search for attributed objects at any level, as will be explained with reference to Example 6. This query is similar to the one in Example 3, but it queries for the "search" attribute instead of the Title attribute. From the results for Example 2, we see there is an attributed object whose attribute is "search", and value of this attributed object is two link objects. The results for Example 4 are as follows:

```
{Answer=>
     Author<http://www.informatik.../a-tree/index.html>
     Title<http://www.informatik.../indices/t-form.html>
     Advanced<http://www.informatik.../indices/query.html>
}
```

Google can search for the "search" keyword from a given site or domain, but it cannot regard "search" as an attribute and therefore it cannot search for the value of the "search" attribute. None of five web query languages from Chapter 2 can do search in this manner either.

Example 5: Retrieve all URLs under attribute "search" to file a5.

```
a5 : {Answer=>$X} :-
     http://www.informatik.uni-trier.de/~/db : {*search=>{<$X>}}
```

As with Example 4, this query searches for the "search" attribute, but only URLs in the value of the attribute are to be returned. This query touches the intra-document structure of an HTML document other than the Title. Neither Google nor any of the five web query languages from Chapter 2 can specify this type of query.

The following is the resulting contents of file a5:

```
{Answer=>{
   http://www.informatik..../a-tree/index.html
   http://www.informatik..../indices/t-form.html
   http://www.informatik..../indices/query.html
}}
```

Comparing file a5 with file a3, we can see that the two anchors in a5 are exactly all the anchors listed in file a3. HTML-QS does exactly what it is supposed to do. Neither Google nor any of the five web query languages from Chapter 2 can do this.

Example 6: List labels of linking objects under the attribute "search".

```
a6 : {Answer=>{$X}} :-
     http://www.informatik.uni-trier.de/~ley/db : {*search=>{$X<>}}
```

As with Example 5, this query only searches for labels of linking objects. The following is the resulting contents of file a6:

```
{Answer=>{
   Author
   Title
   Advanced
}}
```

Example 7: List all attributes at the first and second levels.

```
a7 : {Answer=>$X} :-
     http://www.informatik.uni-trier.de/~ley/db : {$X=>$Y}
a7 : {Answer=>$Y} :-
     http://www.informatik.uni-trier.de/~ley/db : {$X=>$Y=>$Z}
```

The *level* is the depth from the root at which objects are located in an HTML-CM tree. In the HTML-CM tree shown in Figure 4.3, the objects at level 1 are those at the bottom ends of the branches emerging directly from the root, such as Title and People. The following is the resulting contents of file a7:

```
{Answer=>
   Title
   Mirrors
   Search
   Bibliographies
   Full Text
   Links
   DBLP News (February 2006)
   Conferences<http://www.informatik.../conf/indexa.html>
   ...
}
```

Example 8: We know there is a journal called TODS. We want to know the URL of this journal. HTML-QS can search for the label “TODS” and return its URL.

```
a8 : {Answer=>$X} :-
     http://www.informatik.uni-trier.de/~ley/db : {*TODS<$X>}
```

The following is the resulting contents of file a8:

```
{Answer=>
   http://www.informatik.../tods/index.html
}
```

If we look at the query result of Example 2, we can see a linking object under attributed object Journal whose label is TODS and its URL is exactly what HTML-QS returned. This query relates to links in the HTML documents. As discussed in Chapter 2, Google and the five web query languages from Chapter 2 can perform this type of query.

Example 9: Retrieve all URLs in the DBLP Bibliography web page.

```
a9 : {Answer=>{$Y}} :-
     http://www.informatik.uni-trier.de/~ley/db : {*<$Y>}
```

The following is the resulting contents of file a9:

```
{Answer=>{
   http://www.informatik.uni-trier.de/~ley/db/index.html
   http://dblp.uni-trier.de/
   http://www.uni-trier.de/
   http://www.informatik.uni-trier.de/~ley/
   http://www.informatik.uni-trier.de/~ley/db/welcome.html
   http://www.informatik.uni-trier.de/~ley/db/about/faq.html
   http://www.acm.org/sigmod/dblp/db/index.html
   http://www.vldb.org/dblp/db/index.html
   http://sunsite.informatik.rwth-aachen.de/dblp/db/
   http://www.informatik..../a-tree/index.html
   ...
   (72 URLs in total)
}
```

Because this query looks for inter-document structure, all web query languages we studied can handle this query in one way or another. Google can return links to web pages as search results, but it cannot return all links on one page.

Example 10: Obtain all URLs that are reachable from the DBLP Bibliography web page.

```
a10 : {$X} :- http://www.informatik.uni-trier.de/~ley/db : {*<$X>}
a10 : {$X} :- a10 : {<$V>} , $V : {*<$X>}
```

The first query statement is identical to query Example 9. The second query statement uses the output of the first query which is file a10, as an input, and continues retrieving URLs by following the URLs in file a10. This query extends Example 9 and tracks down URLs in the DBLP Bibliography web page. This query is recursive, and stops at a depth of 5. It finds more than 3600 URLs. A detailed discussion of depth can be found in Section 4.2.2.

The following is the resulting contents of file a10:

```
{
   http://www.informatik.uni-trier.de/~ley/db/index.html
   http://dblp.uni-trier.de/
   http://www.uni-trier.de/
   http://www.informatik.uni-trier.de/~ley/
   http://www.informatik.uni-trier.de/~ley/db/welcome.html
   http://www.informatik.uni-trier.de/~ley/db/about/faq.html
   http://www.acm.org/sigmod/dblp/db/index.html
   http://www.vldb.org/dblp/db/index.html
   http://sunsite.informatik.rwth-aachen.de/dblp/db/
   http://www.informatik.../a-tree/index.html
   ...
   (3601 URLs in total)
}
```

Example 11: Obtain the URLs and the corresponding labels for all URLs that are reachable from the DBLP Bibliography web page.

```
a11 : {$X<$Y>} :-
        http://www.informatik.uni-trier.de/~ley/db : {*$X<$Y>}
a11 : {$X<$Y>} :-a11 : {$U<$V>} , $V : {*$X<$Y>}
```

As with Example 10, this query not only searches for URLs but also for the corresponding labels of the URLs. The following is the resulting contents of file a11:

```
  {
     dblp.uni-trier.de<Logo.gif><http://www.informatik.../index.html>
     dblp.uni-trier.de<url.gif><http://dblp.uni-trier.de/>
     UNIVERSITT TRIER<ut.gif><http://www.uni-trier.de/>
     Michael Ley<http://www.informatik.uni-trier.de/˜ley/>
     Welcome<http://www.informatik.uni-trier.de/˜ley/db/welcome.html>
     FAQ<http://www.informatik.uni-trier.de/˜ley/db/about/faq.html>
     ACM SIGMOD<http://www.acm.org/sigmod/dblp/db/index.html>
     VLDB Endow.<http://www.vldb.org/dblp/db/index.html>
     SunSITE Central Europe<http://sunsite.informatik.../dblp/db/>
     Author<http://www.informatik.../a-tree/index.html>
     ...
     (2349 rows in total)
}
```

The total number of label-URL pairs in file a11 is less than the total number of URLs in file a10 for Example 10. A quick investigation shows that some URLs do not have labels. Examples 10 and 11 both search for inter-document structure, and all web query languages we studied can handle this query in one way or another.

For Examples 12 to 15, we use the World Factbook web site at http://www.odci.gov/cia/publications/factbook/index.html as a source of HTML documents. The queries for these examples search the intra-document structure created by converting HTML documents to HTML-CM web objects. Neither Google nor any of the five web query languages from Chapter 2 can perform any of these queries.

Example 12: Find countries that border both Germany and France.

```
a12 : {Answer=>{$N}} :-
       $U : {Title=>$N , *border countries=>{Germany, France}}
```

The following is the resulting contents of file a12:

```
{Answer=>{
   Belgium
   Switzerland
   Luxembourg
}}
```

Example 13: Find countries that border Germany but not France.

```
a13 : {Answer=>{$N}} :-
       $U : {Title=>$N , *border countries=>{Germany},
             !*border countries=>{France}}
```

The following is the resulting contents of file a13:

```
{Answer=>{
   Czech Republic
   Denmark
   France
   Netherlands
   Poland
   Austria
}}
```

Example 14: Obtain pairs of countries that border exactly the same set of countries.

```
a14 : {Answer=>{$A,$B}} :-
       $U : {Title=>$A , *border countries=>$C},
             $V : {Title=>$B , *border countries=>$C},$A!=$B
```

The following is the resulting contents of file a14:

```
{Answer=>{
   Bhutan,Nepal
   Gibraltar,Portugal
   Holy See (Vatican City),San Marino
}}
```

Example 15: Find countries that directly or indirectly border Canada. In other words, find countries that can be reached from Canada by ground.

```
a15 : {Answer=>{$C}} :-
       $U : {Title=>canada , *border countries=>{$C}}
a15 : {Answer=>{$C}} :-
       a15 : {Answer=>{$X}},
              $U : {Title=>$X , *border countries=>{$C}}
```

The following is the resulting contents of file a15:

```
{Answer=>{
   US
   Canada
   Mexico
   Belize
   Guatemala
   El Salvador
   Honduras
   Nicaragua
   Costa Rica
   Panama
   Colombia
   Brazil
   Ecuador
   Peru
   Venezuela
   Argentina
   Bolivia
   French Guiana
```

```
      Guyana
      Paraguay
      Suriname
      Uruguay
      Chile
   }}
```

Twenty-three countries, including Canada itself, were found.

Table 4.3 shows the elapsed times for all 15 example queries. These elapsed times were recorded on Grendel, a twenty processor Unix server in the Department of Computer Science, University of Regina. Only a single processor was used to handle each query. Most of the queries were handled quickly. The retrieval times for Examples 12 through 15 were long. In particular, the elapsed time for Example 15 was more than five hours. These longer times were expected, because many hundreds of web pages had to be retrieved and processed. Building a local data cache to store web pages and the corresponding converted HTML-CM web objects would improve the system performance greatly for such queries, as discussed in Chapter 5.

Table 4.3: Elapsed Time for Example Queries (seconds)

Test	Elapsed Time (second)
1	1.1
2	2.3
3	1.3
4	1.1
5	1.0
6	1.0
7	2.0
8	1.5
9	1.7
10	229.3 (3.8 minutes)
11	185.7 (3 minutes)
12	4478 (1.2 hours)
13	5424 (1.5 hours)
14	6660 (1.9 hours)
15	19176 (5.3 hours)

Table 4.4 summarizes the results described in this section. HTML-QS performed very well in processing all types of queries. The results are correct in every case.

Table 4.4: Summarization of Testing Example Queries

	Example Numbers	HTML-QS Can Do	Google Can Do	Other Web Query Languages Can Do
Inter-document structure query	1, 9, 10, 11	Y	Partly	Y
Intra-document structure query (Title)	3	Y	Partly	Some(3 of 5)
Intra-document structure query (non-Title)	2, 4, 5, 6, 7, 8, 12, 13, 14, 15	Y	N	N

In contrast, Google can perform important aspects of queries related to the inter-document structure. It can also perform important apsects of queries related to the intra-document structure as long as they relate to the Title attribute. It cannot perform queries related to other aspects of the intra-document structure. As described in Chapter 2, the other web query languages can perform queries related to the inter-document structure and some queries related to the intra-document structure. The effectiveness of the HTML-QS system on these 15 queries provide good evidence that it can handle a wide variety of queries. These results also provide evidence for the usefulness of the HTML-CM conceptual model and the HTML-QL query language.

Chapter 5

Conclusions and Future Research

We present the conclusions of the thesis in Section 5.1, with emphasis on the major contributions. We discuss some open issues for future research in Section 5.2.

5.1 Conclusions

The HTML-CM conceptual model and the HTML-QL web query language provide a methodology for finding intra-document structure of HTML documents and performing high level queries on such documents. The other approaches studied in Chapter 2, namely WebSQL, W3QL, WebOQL, WebLog, StruQL, and NetQL, emphasizethe inter-document structure, such as hyperlinks between HTML documents. As described in Chapter 3, the HTML-CM conceptual model captures both the intra-document and inter-document structure relevant to an HTML document. HTML-QL, which is also described in Chapter 3, allows querying based on this conceptual model.

The major original contribution of this thesis is the design and implementation of a prototype system called HTML-QS. The HTML-QS query system supports data extraction, transformation, and integration over HTML documents based on the HTML-CM data model and the HTML-QL query language. All features of HTML-CM and HTML-QL were implemented in HTML-QS, except results for recursive queries were limited to a specified number of levels of recursion. In Chapter 4, we described how the system is designed to perform these functions. The purpose of implementing the HTML-QS query system was twofold: to investigate the feasibility of converting

HTML documents to HTML-CM and performing queries using HTML-QL, and to gain sufficient application experience using the HTML-QL language to identify any needs for improvement.

The second major original contribution of this thesis was a comprehensive evaluation of the HTML-QS system. The implementation of HTML-QS was completed in approximately one year. It performs as expected. The existence of the implementation proves the feasibility of using the HTML-CM data model and the HTML-QL query language. All examples shown in Chapter 3 have been tested in the system and successful results have been obtained as shown in Section 4.3. By comparison with the Google search engine and several other web query languages, we demonstrated the advantages of HTML-QL for the 15 example queries.

5.2 Future Research

This section discusses open issues related to the research described in this thesis. The five aspects considered are data management, memory management, compatibility and extensibility, robustness, script languages, and web query engines.

Data Management: In our prototype system, the *Local Data Repository* consists of flat text files. A better method of data management might be to store the initially retrieved HTML documents and converted web objects in relational tables in a database management system. Our system could then take advantage of database functionality, such as searching and indexing. Because modern database systems are designed to handle huge amounts of data, the performance of the HTML-QS system might be significantly improved.

Memory Management: The current implementation of HTML-QS uses the Java programming language to access and process the HTML documents. Unlike the C programming language, Java does not have a memory allocation function that programmers can call to allocate memory for a variable. Instead, programmers claim and initialize a variable when needed, and Java itself takes care of allocating and releasing memory. Java provides a garbage collector that collects memory that is not being used anymore. This function frees the programmer from spending a great amount of time writing code to manage memory. However, it does have some drawbacks. When a recursive query is being processed, Java's built-in garbage collector does not act soon enough to free memory occupied by queries handled by earlier recursive passes. Therefore, the maximum depth of recursion is sharply limited. A new algorithm or memory management tool might be able to reduce the severity of this problem. One option is to pause the process and save the intermediate result into a file and then resume the process when enough memory has been released.

Compatibility and Extensibility: Although HTML is still the most widely used web construction language, a variety of related standards and recommendations for constructing web sites have emerged in the last decade, such as Extensible Hyper Text Markup Language (XHTML) [49], Dynamic HTML (DHTML) [28], and Extensible Markup Language (XML) [50]. XML's fast spread suggests that more languages will be designed.

Both the XHTML and DHTML languages can be regarded as extensions to HTML. XHTML is based on a standard defined by W3C, while DHTML is defined by Netscape and Microsoft. XHTML and DHTML include all tags present in HTML, but they also include some new tags or scripts that support new features for making the presentation of web pages more attractive. For instance, in DHTML, one can add transition effects between web pages similar to those found between slides in Microsoft PowerPoint presentations. These extensions do not affect the textual information that the web pages convey, so they will not affect HTML-QS's retrieval of the structure of these web pages. However, new filters should be developed and deployed in the Intelligent Wrapper to handle the new tags in XHTML and DHTML.

XML is a widely used language for data representation and exchange. It does not support the HTML tag set but instead it provides a facility to define tags. Liu has already proposed a preliminary outline of a rule-based XML query system [33] that is similar to HTML-QS. A system that generalizes Liu's XML query system and

HTML-QS could be developed.

Robustness: In the prototype system, we used a layered design to make the system relatively robust. This layered design can capture most types of errors at the layer where they occur and return meaningful information for further investigation. The Query Processor captures syntax errors in query statements. The improper use of query constructs is the most frequently seen error. For example, the ">" symbol may be missing from an attributed object. Logic errors occur when users of HTML-QS fail to use the HTML-QL language in the way that it is intended. An error of this type may not cause HTML-QS to crash, but it may instead produce results that differ from what users expect. Logic errors can be complicated and of a great variety. In general, logic errors can be reduced by further study of the syntax and semantics of languages, so further study of HTML-QL might reduce such errors. A comparison of sample queries and results might also reduce the number of logic errors.

The Intelligent Wrapper finds errors in HTML documents and attempts to handle them without quitting the current query. For example, a missing ending tag is added automatically while an unrecognized tag is reported to the user.

Bugs in the code and other unexpected operations can occasionally cause the HTML-QS system to fail. We use the exception mechanism of Java to handle these failures. When an error occurs, the system captures the exception and reports an error message to the user. For example, the following problem was encountered. The same Java source code was compiled on two different platforms, Microsoft Windows XP Professional and Unix. When a query was submitted on the Unix platform, the system failed. However, when the same query was submitted on the Windows XP platform, the system performed as expected. Further investigation showed that these two platforms had two different versions of the Java JDK and JVM installed. Functions that are deprecated in one version but not in another version might cause problems.

In some cases, however, the error messages are difficult to understand neither or misleading about the actual error. In order to use error messages to narrow down the sources of problems, we need to characterize the various types of error-causing situations and identify them with more specific error messages.

Script Languages: In the current implementation, scripts written in Java Script or VB Script, are filtered out. However, given the popularity of using scripts to beautify web pages, an increasing amount of information is being embedded in scripts. For example, a flashing text string may denote something important. A further study of scripting languages is necessary in order to extract the information embedded within scripts.

Web Query Engines: The HTML-QS query system and a search engine could be combined to create a web query engine. The search engine component would focus on searching the Internet, matching keywords, and ranking the relevant HTML documents. The HTML-QS component would focus on querying the structure of the HTML documents. Therefore, such a web query engine could take advantage of the searching and ranking abilities of a search engine, and also be able to perform queries on both the inter-document structure and inter-document structure of HTML documents. Web crawlers and local data repositories in the two components could be consolidated to improve efficiency.

Bibliography

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, 1997.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[3] AOL. http://www.aol.com/nethelp/misc/history.html. Accessed in October 2005.

[4] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring documents, databases, and webs. In *Proceedings of the International Conference on Data Engineering*, pages 24–33, 1998.

[5] G. O. Arocena, A. O. Mendelzon, and G. A. Mihaila. Applications of a Web query language. *Computer Networks and ISDN Systems*, 29(8–13):1305–1315, 1997.

[6] T. Attwood, J. Duhl, G. Ferran, M. Loomis, and D. Wade. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1996.

[7] R. Barták. On-line Guide to Prolog Programming, http://kti.ms.mff.cuni.cz/~bartak/prolog/index.html. Accessed in October 2005.

[8] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set construction in a logic database language. *Journal of Logic Programming*, 10(3,4):181–232, 1991.

[9] T. Berners-Lee and R. Cailliau. WWW Seminar, http://www.w3.org/Talks /General/Transparencies.html. Accessed in October 2005.

[10] S. Boag, D. Chamberlin, M. F. Fernandez, and et al. XQuery 1.0: An XML Query Language. *See at http://www.w3.org/TR/xquery/.*

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

[12] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 505–515, Montréal, Canada, 1996.

[13] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. In *Proceedings of the Eighth International Conference on the World Wide Web*, pages 1171–1187, Toronto, Canada, 1999.

[14] A. Deutsch, M. Fernandez, and et al. XML-QL: A Query Language for XML. *See at http://www.w3.org/TR/NOTE-xml-ql/.*

[15] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Overview of Strudel: A web-site management system. *Networking and Information Systems*, 1(1):115–140, 1998.

[16] M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, 1997.

[17] M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. Reasoning about Website structure. In *Proceedings of Knowledge Representation Meets Databases*, pages 10.1–10.9, 1998.

[18] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[19] Google. http://www.google.com/intl/en/corporate/history.html. Accessed in March 2006.

[20] T. Guan, M. Liu, and L. V. Saxton. Structure-based queries over the World Wide Web. In *Proceedings of the 17th International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 107–120, Singapore, 1998.

[21] F. G. Halasz. Reflections on notecards: Seven issues for the next generation of hypermedia systems. In *Proceedings of Hypertext'87*, pages 345–365, Chapel Hill, North Carolina, USA, 1987.

[22] R. Himmeroder, G. Lausen, B. Ludascher, and C. Schlepphorst. On a declarative semantics for web queries. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 386–398, Switzerland, 1997.

[23] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[24] D. Konopnicki and O. Shmueli. W3QS: A query system for the World-Wide Web. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 54–65, Zurich, Switzerland, 1995.

[25] E. Krol. *The Whole Internet User's Guide & Catalog.* O'Reilly & Associates, 1992.

[26] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the logical foundation of schema integration and evolution in heterogeneous database systems. In *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases*, pages 81–100, Phoenix, Arizona, 1993.

[27] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A declarative language for querying and restructuring the Web. In *Proceedings of the Sixth International Workshop on Research Issues on Data Engineering: Interoperability of Nontraditional Database Systems*, pages 12–21, 1996.

[28] MSDN Library. Dynamic HTML (DHTML), http://msdn.microsoft.com/library /default.asp?url=/workshop/author/dhtml/dhtml.asp. Accessed in December 2005.

[29] M. Liu. *NETQL: A Structured Web Query Language*, Master's thesis, Department of Computer Science, University of Regina, 1998.

[30] M. Liu and T. W. Ling. A conceptual model for the web. In *Proceedings of the International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 225–238, 2000.

[31] M. Liu and T. W. Ling. A data model for semistructured data with partial and inconsistent information. In *Proceedings of the Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology*, pages 317–331, Konstanz, Germany, 2000.

[32] M. Liu and T. W. Ling. A conceptual model and rule-based query language for HTML. *World Wide Web*, 4(1-2):49–77, 2001.

[33] M. Liu and T. W. Ling. Towards declarative XML querying. In *Proceedings of the 3rd International Conference on Web Information System Engineering*, Singapore, Decemeber 2002.

[34] T. Mayer. http://www.ysearchblog.com/archives/000172.html. Accessed in March 2006.

[35] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[36] G. A. Mihaila. *WebSQL – An SQL-like query language for the World Wide Web*, Master's thesis, University of Toronto, 1996.

[37] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography*, 4(3):235–312, 1990.

[38] Editors of The American Heritage Dictionaries. *The American Heritage Dictionary of the English Language, Fourth Edition.* Houghton Mifflin, 2000.

[39] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank Citation Ranking: Bring Order to the Web.* Technical Report, Stanford University, 1998.

[40] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the 11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995.

[41] D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases*, pages 319–344, 1995.

[42] D. Raggett, A. L. Hors, and I. Jacobs. *HTML 4.01 Specification.* W3C Recommendation, 1999.

[43] Citynet Search Engine Services. http://www.citynetsearch.com/basics.htm. Accessed in October 2005.

[44] D. Sullivan. Nielsen NetRatings seach engine ratings, http://searchenginewatch.com/reports/article.php/2156451. Accessed in November 2005.

[45] Google Search Technology. http://www.google.com/technology/index.html. Accessed in October 2005.

[46] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[47] J. D. Ullman and J. Widom. *A First Course in Database Systems.* Prentice Hall, 1997.

[48] W3C. http://www.w3.org/www. Accessed in October 2005.

[49] World Wide Web Consortium (W3C). Extensible Hyper Text Markup Language (XHTML), http://www.w3.org/TR/xhtml1/. Accessed in December 2005.

[50] World Wide Web Consortium (W3C). Extensible Markup Language (XML), http://www.w3.org/XML. Accessed in October 2005.

[51] R. T. Watson. *Data Management: An Organizational Perspective*. John Wiley & Sons, 1996.

Appendix A

Conversion Rules

The following conversion rules are adapted from Liu and Ling's description of HTML-QL [30, 31, 32].

Rule 1 Let D_F be a frame-based document with frames or nested framesets $F_1, ..., F_n$ in its body:

$$D_F = \texttt{<html><head><title>}\ T\ \texttt{</title></head>}$$
$$\texttt{<frameset ...>}\ F_1, ..., F_n\ \texttt{</frameset> </html>}$$

and $\uplus$ be list concatenation operator. Then D_F is converted into a list of two attributed objects using the operator C as follows:

$$C(D_F) = \{Title \Rightarrow T, Frameset \Rightarrow C(F_1) \uplus ... \uplus C(F_n)\}$$

Rule 2 Let F = `<frame name = "`N`" src = "`U`">` be a frame, where N is the name and U is the URL of the frame. Then $C(F) = \{N\langle U\rangle\}$.

Rule 3 Let Fs = `<frameset ...>`$F_1, ..., F_n$`</frameset>` be a frame set, where each F_i for $1 \leq i \leq n$ is a frame or a nested frameset. Then $C(Fs) = C(F_1) \uplus ... \uplus C(F_n)$.

Rule 4 Let D_R be a regular document with sections $S_1, ..., S_n$ in its body:

$$D_R = \texttt{<html><head><title>}\ T\ \texttt{</title></head>}$$
$$\texttt{<body>}\ S_1\ ...\ S_n\ \texttt{</body> </html>}$$

Then $C(D_R) = \{Title \Rightarrow T, C(S_1), ..., C(S_n)\}$.

Rule 5 Let $S =$ `<h`n`>` H `</h`n`>` T be a section with a heading H and contents T and $S' = T'$ a section without a heading. Then $C(S) = C(H) \Rightarrow C(T)$, $C(S') = C(T')$.

Rule 6 Let $P =$ `<`t`>`B`</`t`>` : R or $P =$ `<`t`>`B : `</`t`>`R be a paragraph with an emphasized beginning and $P' = R$ a paragraph without an emphasized beginning, where t is either `b`, `i`, `em`, or `strong`. Then $C(P) = C(B) \Rightarrow C(R)$, and $C(P') = C(R)$. If R has logical parts $R_1, ..., R_n$ with $n \geq 1$, then

$$\begin{array}{ll} C(R) = \{C(R_1), ..., C(R_n)\} & \text{if } n > 1 \\ C(R) = C(R_1) & \text{if } n = 1 \end{array}$$

Each logical part R_i for $1 \leq i \leq n$ is converted as a paragraph recursively.

Rule 7 Let $I =$ `<img src = "`U`" alt="`T`">` be an image link, where U is a URL and T is a string. Then $C(I) = Image \Rightarrow T\langle U\rangle$. When the `alt` field is missing, we treat it as `alt=""`.

Rule 8 Let $H =$ `<a href = "U">` T `</a>` be a hypertext link, where U is a URL and T is a string. Then $C(H) = C(T)\langle U\rangle$.

Rule 9 Let S be a character string. Then $C(S) = S$.

Rule 10 Let $L =$ `<`t`><li>`L_1 ... `<li>`L_n`</`t`>` be an ordered or unordered list, where t is either `ol` or `ul`. Then

$$\begin{array}{ll} C(L) = \{C(L_1), ..., C(L_n)\} & \text{when } n > 1 \\ C(L) = C(L_1) & \text{when } n = 1 \end{array}$$

Rule 11 Let $L_D =$ `<dl><dt>`N_1`<dd>`L_1 ...`<dt>`N_m`<dd>`L_m `</dl>` be a definition list. Then

$$\begin{array}{ll} C(L_D) = \{C(N_1) \Rightarrow C(L_1), ..., C(N_m) \Rightarrow C(L_m)\} & \text{when } n > 1 \\ C(L_D) = C(N_1) \Rightarrow C(L_1) & \text{when } n = 1 \end{array}$$

Rule 12 Let $I = T$`<`t`><li>`T_1 ...`<li>`T_m`</`t`>` be a nested ordered or unordered list item where t is either `ol` or `ul`. Then

$$C(I) = C(T) \Rightarrow \{C(T_1), ..., C(T_m)\}$$

Rule 13 Let $T =$ `<table><caption>`H`</caption>`TC`</table>` be a table with a caption and $T' =$ `<table>`TC`</table>` be a table without a caption, where H is the caption and TC is the table contents. Then

$$C(T) = C(H) \Rightarrow C(TC)$$
$$C(T') = C(TC).$$

Let $R_1, ..., R_n$ be rows other than the row for column headings in the table contents TC. Then $C(TC) = \{C(R_1), ..., C(R_n)\}$. For each row R_i with $1 \leq i \leq n$:

(1) if the table has column headings $H_1, ..., H_n$ and each row R_i has a row heading H: $R_i =$ `<tr><th>`H`<td>`$C_1 \ldots$`<td>`C_n, then
$C(R_i) = C(H) \Rightarrow \{C(H_1) \Rightarrow C(C_1), ..., C(H_n) \Rightarrow C(C_n)\}$

(2) if the table has column headings $H_1, ..., H_n$, but each row R_i has no row heading: $R_i =$ `<tr><td>`$C_1 \ldots$`<td>`C_n, then
$C(R_i) = \{C(H_1) \Rightarrow C(C_1), ..., C(H_n) \Rightarrow C(C_n)\}$

(3) if the table does not have column headings but each row R_i has a row heading: $R_i =$ `<tr><th>`H`<td>`$C_1 \ldots$`<td>`C_n, then
$C(R) = C(H) \Rightarrow \{C(C_1), ..., C(C_n)\}$

(4) if the table has neither column nor row headings, then for each row $R_i =$ `<td>`$C_1 \ldots$`<td>`C_n, $C(R) = \{C(C_1), ..., C(C_n)\}$

Rule 14 Let F be a form as follows where $E_1, ..., E_n$ are the elements in the form.

$F =$
```
<form ...>
E1,...,En
<input type = "submit" ...>
<input type = "reset"...>
</form>
```

Then $C(F) = \mathit{Form} \Rightarrow \{C(E_1), ..., C(E_n)\}$.

Rule 15 Let $T_F = L$:`<input type="`T`" name="`N`" value="`V`">` be a text field, where L is the label visible to the user, T is the type for the input text, N is the name for the text field that is not visible to the user, and V is the default value. Then $C(T_F) = \mathit{Text\ Field} \Rightarrow \{\mathit{Label} \Rightarrow L,\ \mathit{Name} \Rightarrow N,\ \mathit{Type} \Rightarrow T,\ \mathit{Value} \Rightarrow V\}$.

Rule 16 Let R be a group of radio buttons with the same name of the following form, where L is the label for the group, N is the name, V_i is the value, and L_i is the label visible to the user.

$$
\begin{aligned}
R = L :\ & \texttt{<input type = "radio" name = "}N_1\texttt{" value = "}V_1\texttt{">}L_1 \\
& \ \ldots \\
& \texttt{<input type = "radio" name = "}N_m\texttt{" value = "}V_m\texttt{">}L_m
\end{aligned}
$$

Then $C(R) = \mathit{Radio\ Buttons} \Rightarrow \{\mathit{Label} \Rightarrow L,\ \mathit{Name} \Rightarrow N,$

$$
\begin{aligned}
\mathit{Options} \Rightarrow \{ & \{\mathit{Label} \Rightarrow L_1, \mathit{Value} \Rightarrow V_1\}, \\
& \ldots \\
& \{\mathit{Label} \Rightarrow L_n, \mathit{Value} \Rightarrow V_n\}\}
\end{aligned}
$$

Rule 17 Let M be a menu of the following form, where L is the label for the menu visible to the user, N is the name for the menu, V_i is the value, and L_i is the label visible to the user.

$$
\begin{aligned}
M = L\ & \texttt{<select name = "}N\texttt{" ...>} \\
& \texttt{<option value = "}V_1\texttt{">}L_1 \\
& \ \ \ldots \\
& \texttt{<option value = "}V_n\texttt{">}L_n \\
& \texttt{</select>}
\end{aligned}
$$

Then $C(M) = \mathit{Menu} \Rightarrow \{\mathit{Label} \Rightarrow L,\ \mathit{Name} \Rightarrow N$

$$
\begin{aligned}
\mathit{Options} \Rightarrow \{ & \{\mathit{Label} \Rightarrow L_1,\ \mathit{Value} \Rightarrow V_1\}, \\
& \ldots \\
& \{\mathit{Label} \Rightarrow L_n,\ \mathit{Value} \Rightarrow V_n\}\}
\end{aligned}
$$