



Denis Shestakov

Search Interfaces on the Web: Querying and Characterizing

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Dissertations

No 104, May 2008

Search Interfaces on the Web: Querying and Characterizing

Denis Shestakov

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in the Auditorium Lambda, ICT building, on June 12, 2008, at 12 o'clock.

University of Turku
Department of Information Technology
Joukahaisenkatu 3-5, FIN-20520 Turku

2008

Supervisor

Professor Tapio Salakoski
Department of Information Technology
University of Turku
Joukahaisenkatu 3-5, FIN-20520 Turku
Finland

Reviewers

Docent Ari Pirkola
Department of Information Studies
University of Tampere
Kanslerinrinne 1, FIN-33014 Tampere
Finland

Dr. Bin He
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120-6099
USA

Opponent

Associate Professor Mark Sanderson
Department of Information Studies
University of Sheffield
Regent Court, 211 Portobello St, Sheffield, S1 4DP
United Kingdom

ISBN 978-952-12-2099-9
ISSN 1239-1883
Painosalama Oy - Turku, Finland 2008

Abstract

Current-day web search engines (e.g., Google) do not crawl and index a significant portion of the Web and, hence, web users relying on search engines only are unable to discover and access a large amount of information from the non-indexable part of the Web. Specifically, dynamic pages generated based on parameters provided by a user via web search forms (or search interfaces) are not indexed by search engines and cannot be found in searchers' results. Such search interfaces provide web users with an online access to myriads of databases on the Web. In order to obtain some information from a web database of interest, a user issues his/her query by specifying query terms in a search form and receives the query results, a set of dynamic pages that embed required information from a database. At the same time, issuing a query via an arbitrary search interface is an extremely complex task for any kind of automatic agents including web crawlers, which, at least up to the present day, do not even attempt to pass through web forms on a large scale.

In this thesis, our primary and key object of study is a huge portion of the Web (hereafter referred as the deep Web) hidden behind web search interfaces. We concentrate on three classes of problems around the deep Web: characterization of deep Web, finding and classifying deep web resources, and querying web databases.

Characterizing deep Web: Though the term deep Web was coined in 2000, which is sufficiently long ago for any web-related concept/technology, we still do not know many important characteristics of the deep Web. Another matter of concern is that surveys of the deep Web existing so far are predominantly based on study of deep web sites in English. One can then expect that findings from these surveys may be biased, especially owing to a steady increase in non-English web content. In this way, surveying of national segments of the deep Web is of interest not only to national communities but to the whole web community as well. In this thesis, we propose two new methods for estimating the main parameters of deep Web. We use the suggested methods to estimate the scale of one specific national segment of the Web and report our findings. We also build and make publicly available a dataset describing more than 200 web databases from the national segment of the Web.

Finding deep web resources: The deep Web has been growing at a very fast pace. It has been estimated that there are hundred thousands of deep web sites. Due to the huge volume of information in the deep Web, there has been a significant interest to approaches that allow users and computer applications to leverage this information. Most approaches assumed that search interfaces to web databases of interest are already discovered and known to query systems. However, such assumptions do not hold true mostly because of the large scale of the deep Web – indeed, for any given domain of interest there are too many web databases with relevant content. Thus, the ability to locate search interfaces to web databases becomes a key requirement for any application accessing the deep Web. In this thesis, we describe the architecture of the I-Crawler, a system for finding and classifying search interfaces. Specifically, the I-Crawler is intentionally designed to be used in deep Web characterization studies and for constructing directories of deep web resources. Unlike almost all other approaches to the deep Web existing so far, the I-Crawler is able to recognize and analyze JavaScript-rich and non-HTML searchable forms.

Querying web databases: Retrieving information by filling out web search forms is a typical task for a web user. This is all the more so as interfaces of conventional search engines are also web forms. At present, a user needs to manually provide input values to search interfaces and then extract required data from the pages with results. The manual filling out forms is not feasible and cumbersome in cases of complex queries but such kind of queries are essential for many web searches especially in the area of e-commerce. In this way, the automation of querying and retrieving data behind search interfaces is desirable and essential for such tasks as building domain-independent deep web crawlers and automated web agents, searching for domain-specific information (vertical search engines), and for extraction and integration of information from various deep web resources. We present a data model for representing search interfaces and discuss techniques for extracting field labels, client-side scripts and structured data from HTML pages. We also describe a representation of result pages and discuss how to extract and store results of form queries. Besides, we present a user-friendly and expressive form query language that allows one to retrieve information behind search interfaces and extract useful data from the result pages based on specified conditions. We implement a prototype system for querying web databases and describe its architecture and components design.

Acknowledgements

Many people have contributed to this thesis in one way or another. I am happy for the opportunity to sincerely thank all of these people.

First and foremost, I would like to thank my supervisor, Professor Tapio Salakoski for his guidance, advice and support. I thank Tapio specifically for giving me enough flexibility in my research work.

I would also like to express my gratitude to all the members of the TUCS Bioinformatics Laboratory, Further, I would like to thank the entire TUCS staff for both financial and administrative support I was provided with during my studies. Specifically, I am very grateful for the funding granted to me by the Turku Centre for Computer Science. Special thanks goes to Tomi Mäntylä for his help with the preparation of the camera-ready version of this thesis.

I would also like to thank the reviewers of my thesis, Docent Ari Pirkola at the University of Tampere and Dr. Bin He at the IBM Almaden Research Center, for their time and efforts in reviewing the manuscript. Their constructive and valuable comments helped me to improve my thesis.

On the personal level, I want to express my gratitude to my wife's and my parents for their understanding and love. Last, but by no means least, I cannot end without thanking my family, my beautiful wife and wonderful daughter, on whose constant encouragement and love I have relied throughout my PhD-work.

In Turku, May 2008
Denis Shestakov

List of original publications

- I.** Denis Shestakov. A prototype system for retrieving dynamic content. In *Proceedings of Net.ObjectDays 2003*, 2003.
- II.** Denis Shestakov, Sourav S. Bhowmick, and Ee-Peng Lim. DEQUE: querying the deep Web. *Data Knowl. Eng.*, 52(3):273–311, 2005.
- III.** Denis Shestakov and Natalia Vorontsova. Characterization of Russian deep Web. In *Proceedings of Yandex Research Contest 2005*, pages 320–341, 2005. In Russian.
- IV.** Denis Shestakov and Tapio Salakoski. On estimating the scale of national deep Web. In *Proceedings of DEXA'07*, pages 780–789, 2007.
- V.** Denis Shestakov. Deep Web: databases on the Web. Entry in *Encyclopedia of Database Technologies and Applications*, 2nd edition, IGI Global, To appear.
- VI.** Denis Shestakov and Tapio Salakoski. Characterization of national deep Web. Technical Report 892, Turku Centre for Computer Science, May 2008.

Contents

1	Introduction	1
1.1	Non-indexable Web	1
1.2	Deep Web	5
1.2.1	Scale of Deep Web	9
1.3	Motivation	9
1.3.1	Characterizing the Deep Web	10
1.3.2	Automatic Finding of Search Interfaces	10
1.3.3	Querying Web Databases	11
1.4	Challenges	11
1.5	Contributions	13
1.6	Scope and Organization of this thesis	14
2	Related Work	17
2.1	Deep Web	18
2.1.1	Search Engines	18
2.1.2	Deep Web	19
2.1.3	Classification of Web Databases	20
2.2	Querying Search Interfaces	23
2.2.1	W3QL and W3QS	23
2.2.2	ARANEUS	25
2.2.3	Database System for Querying Forms	28
2.2.4	Crawling the Hidden Web	31
2.2.5	Modeling and Querying the World Wide Web	33
2.2.6	Summary: Querying Search Interfaces	36
2.3	Extraction from Data-intensive Web Sites	37
2.3.1	Extracting Semi-structured Information from the Web	37
2.3.2	Automatic Data Extraction from Large Web Sites	38
2.4	Deep Web Characterization	41
2.4.1	Overlap analysis	41
2.4.2	Random Sampling of IP Addresses	42
2.4.3	Virtual Hosting and DNS load balancing	43
2.4.4	National Web Domains	45

2.5	Finding deep web resources	46
3	Deep Web Characterization	47
3.1	Random Sampling of IP Addresses	47
3.2	Stratified Random Sampling of Hosts	50
3.2.1	Subject Distribution of Web Databases	53
3.3	Discussion: results of the 2005 survey	53
3.4	Stratified cluster sampling of IP addresses	55
3.4.1	Dataset preparation and stratification	56
3.4.2	Results of the 2006 survey	57
4	Finding Deep Web Resources	61
4.1	Introduction	61
4.2	Motivation and Challenges	62
4.3	Our approach: Interface Crawler	64
4.4	Architecture of I-Crawler	67
4.5	Experimental Results	70
4.6	Conclusion and Future Work	71
5	Querying the Deep Web	73
5.1	Introduction	73
5.2	Modeling of A Single HTML Form	76
5.2.1	HTML Forms	76
5.2.2	Form Fields	77
5.3	Modeling of Consecutive Forms	78
5.3.1	Form Type and Location	79
5.3.2	Issues in Modeling of Consecutive Forms	80
5.3.3	Submission Data Set of Consecutive Forms	81
5.3.4	Form Unions	82
5.3.5	Super Form	83
5.3.6	Form Extraction	86
5.4	Representation of Result Pages	89
5.4.1	Result Navigation	89
5.4.2	Result Matches	89
5.4.3	DEFINE Operator	91
5.4.4	Result Extraction	93
5.5	Deep Web Query Language (DEQUEL)	94
5.5.1	Value Assignment	95
5.5.2	DEQUEL Syntax	96
5.5.3	Examples of DEQUEL Queries	97
5.5.4	DEQUEL Query Execution	101
5.6	Implementation	102
5.6.1	System Architecture	103

5.6.2	Form and Result Storage	105
5.6.3	Experimental Results	107
5.7	Conclusion	110
6	Conclusions	113
6.1	Summary of Our Contributions	113
6.2	Future Work	114
6.3	Future Trends	115
6.4	Conclusion	116
	Bibliography	117
A	Deep Web	127
A.1	User interaction with web database	127
A.2	Key terms	127
B	DEQUEL Grammar	129
B.1	DEFINE Operator	129
B.1.1	First Type of Syntax	129
B.1.2	Second Type of Syntax	129
B.2	SELECT Operator	129
C	Deep Web Characterization	133
C.1	Detection of Active Web Servers	133
C.2	Reverse IP/Host Lookup tools	133
C.3	Hostgraph fraction of zones	134
C.4	RU-hosts	134

Chapter 1

Introduction

Current web search engines include in their indices only a portion of the Web. There are a number of reasons for this, including inevitable ones, but the most important point here is that the significant part of the Web is unknown to search engines. It means that search results returned by web searchers enumerate only relevant pages from the indexed part of the Web while most web users treat such results as a complete (or almost complete) set of links to all relevant pages on the Web.

In this thesis our primary and key object of study is a huge portion of the Web hidden behind web search interfaces, which are also called web search forms. These interfaces provide web users with an online access to myriads of databases on the Web. At the same time, web forms are formidable barriers for any kind of automatic agents, e.g., web crawlers, which, unlike human beings, have great difficulties in filling out forms and retrieving information from returned pages. Hereafter we refer to all web pages behind search interfaces as the deep Web.

The deep Web is not the only part of the Web, which is badly indexed by search engines. For better understanding the subject we start with a brief description of the non-indexable portion of the Web in general.

1.1 Non-indexable Web

The non-indexable Web (or invisible Web [97]) consists of web documents which are poorly indexed or not indexed at all by current-day search engines. According to the study conducted in 1999 by Lawrence and Giles [72] no engine of that time indexed more than one-sixth of the Web.

A web crawler, an automatic tool used by search engines to collect web pages to be indexed, browses the Web in the following way: it starts with a predefined list of URLs to visit (called the seeds) and, as the crawler processes these URLs, it extracts all hyperlinks from visited pages and adds

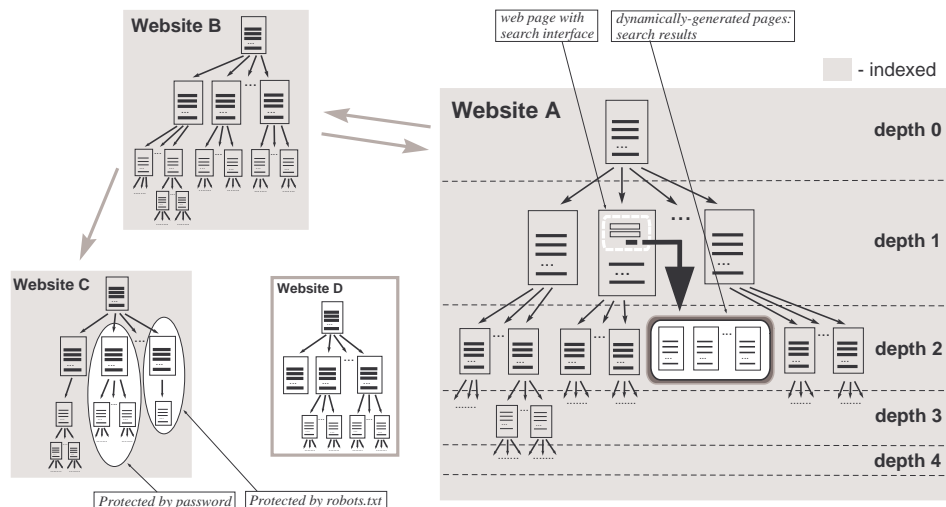


Figure 1.1: Sketchy representation of the Web (for simplicity, an arrow from one website to another website means that there is at least one hyperlink on some page of the former site to some page of the latter one).

them to the list of URLs to visit (called the crawl frontier). URLs from the frontier are recursively visited according to a set of crawler's rules. Clearly, web pages which URLs are neither in the seeds nor in the frontier are not added to search engines' indices. Figure 1.1 gives an schematic illustration of this. While websites *A*, *B*, and *C* are connected to each other via hyperlinks on their pages no incoming links to any page of website *D* exist. Thus, all pages of website *D* remain unindexed until the crawler visits a page with a link to some page on site *D* or some URL of the *D* is appended to the crawler seeds.

Limited resources of web crawlers are a further factor restricting engines' coverage of the Web. Specifically, huge volumes of web data imply that a crawler can only download a fraction of a particular website within a given time. In a like manner, large web documents are not indexed fully by search engines - any text that extends beyond certain size limits is, in fact, disregarded¹. One can expect, however, that these limitations will be gradually overcome over the next few years. Indeed, until 2001, web documents in non-HTML formats such as, for instance, PDF or MS Office (Word, Excel, PowerPoint) were ignored by searchers due to high computational costs of indexing non-HTML files. The situation was changed during last years when search engines began to recognize and index several document formats and,

¹According to Bondar, in 2006, the limits of three leading search engines were 210KB for Yahoo!, 520KB for Google, and 1030KB for Live Search. (<http://www.sitepoint.com/article/indexing-limits-where-bots-stop>)

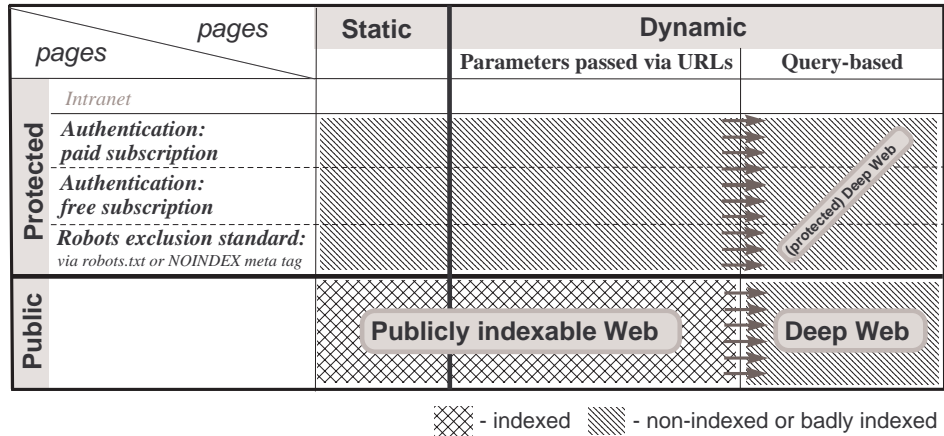


Figure 1.2: Indexable and non-indexable portions of the Web.

hence, the corresponding portion of the Web turned out to be indexed.

Protection of web content is another roadblock for web crawlers. There are generally two types of web page protection (website *C* depicted in Figure 1.1 are protected using both types):

- For a variety of reasons a website owner can specify which parts of a site should not be crawled and indexed by search engines in a file named *robots.txt* [9]. There is a page-specific prevention as well: NOINDEX meta tag in the head of a web page does not allow a searcher to crawl this page.
- Pages can be protected with a login and password, which are obtainable either on a free or a paid basis. Crawlers cannot access such pages. Though little can be done about pages accessible on a paid basis it is a matter of concern that freely available password-protected pages are unknown to search engines. Note that according to our point of view “web” pages which are only available when visited from within a certain network, i.e., a corporate intranet, or which are only accessible by employees of a certain company do not belong to the Web because such pages are not intended for large audience as conventional web pages (even with a paid access) are.

Lastly, dynamic web content poses a challenge for search engines. A dynamic web page is a page which content is generated at run-time, i.e., after a request for this page has been made, by a program executing either on a server or on a client. This is in contrast to a static web page which content already exists on a server, ready to be transmitted to a client whenever a request is received. Dynamic pages can be divided into two classes. First group of pages is those that include parameters used for page generation in

their URLs². Technically these pages are indexable since they are accessible via hyperlinks. But, in reality, crawlers have troubles with such pages and frequently prefer to avoid them. The main threat here is so called “crawler traps” [32], i.e., a dynamically-generated page may have a link to another dynamic page which may lead to next page and so on. Following “*Next month*” link on a web calendar is an example of a “fair”³ endless loop formed by dynamic pages. Second class of dynamic content is pages generated based on parameters provided by a user via search interfaces (also known as search forms). These interfaces provide web users with an online access to myriads of databases on the Web. In order to obtain some information from a web database of interest, a user issues his/her query by specifying query terms in a search form and receives the query results, a set of dynamic pages which embed required information from a database. However, issuing a query via arbitrary search interface is an extremely complex task for any kind of automatic agents including web crawlers, which, at least up to the present day, do not even attempt to pass through web forms on a large scale. To illustrate, website *A* shown in Figure 1.1 has one page containing a search interface and this particular page is known and indexed by search engines. However, dynamically-generated pages with results of a query issued via this search interface are not indexed by web searchers due to their inability to query web forms.

Summarizing picture of the indexable and non-indexable portions of the Web is given in Figure 1.2 (note that the portion sizes shown are not to scale). There are four main types of web data: public, protected, static, and dynamic. Though we do not consider intranet pages as a part of the Web they can still be classified as protected pages. Dynamic web pages can be divided into those with parameters specified within their URLs and those generated based on provided-via-search-interface parameters. The indexable Web or **publicly indexable Web** (PIW) [71] consists of publicly available both static and parameter-known dynamic web pages. All other portions of the Web belong to the non-indexable Web. In this thesis our principal concern is the deep Web, i.e., a portion of the Web formed by intersection of public and query-based dynamic parts of the Web. However, since most problems related to the deep Web do not lie on the level of authorization or registration our study is also relevant to the protected deep Web (and intranet deep Web).

²They can often be detected by URLs with ‘?’ and ‘&’ symbols.

³It may be “unfair” as well. For instance, millions of similar but not quite identical pages can be specifically generated to spam search engines indices.

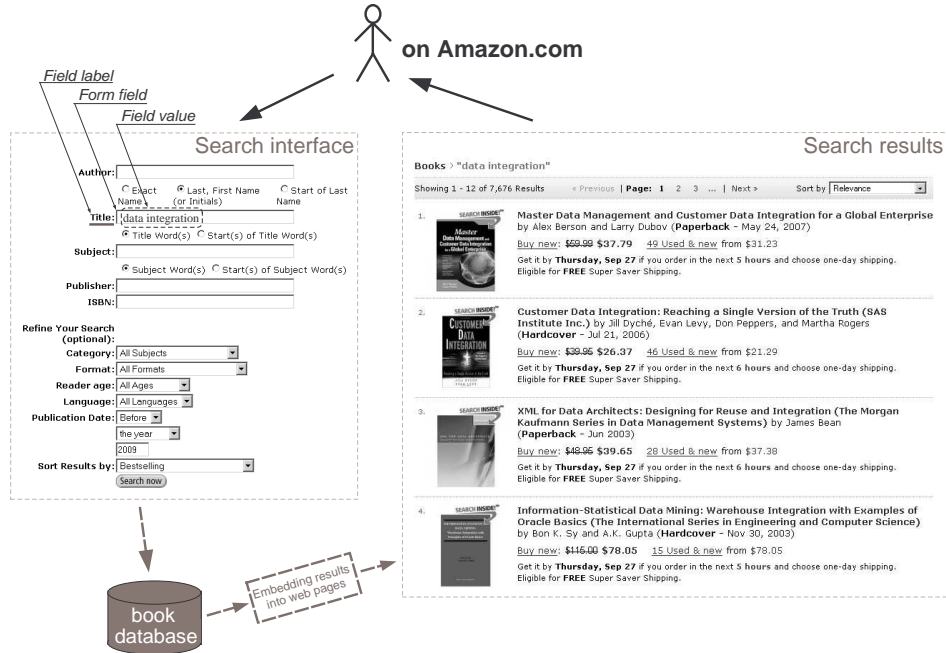


Figure 1.3: User interaction on *Amazon.com*.

1.2 Deep Web

With the advances in web technologies [1, 2, 5, 7, 45, 48, 54, 60, 84], web pages are no longer confined to static HTML files that provide direct content. This leads to more interactivity of web pages and, at the same time, to ignoring a significant part of the Web by search engines due to their inability to analyze and index most dynamic web pages. Particularly, query-based dynamic portion of the Web known as the deep Web is poorly indexed by current-day search engines.

There is a slight uncertainty in the terms defining the part of the Web that is accessible via web search interfaces to databases. In literature, one can observe the following three terms: invisible Web [97], hidden Web [46], and deep Web [25]. The first term, invisible Web, is a superior to latter two terms as it refers to all kind of web pages which are non-indexed or badly indexed by search engines (i.e., non-indexable Web). The terms hidden Web and deep Web are generally interchangeable, and it is only a matter of preference which to choose. In this thesis we use the term deep Web and define it as web pages generated as results of queries issued via search interfaces to databases available online. In this way, the deep Web is a large part but still part of the invisible Web (see Figure 1.2).

We further clarify the concept of the deep Web by three related notions

introduced in [34]: a deep web site, a database, and a search interface (or search form). A deep web site is a web site that provides an access to one or more back-end web databases, each of which is searchable through one or more search interfaces⁴. Appendix A provides additional information regarding to the terms commonly used in deep Web-oriented studies.

The presence of at least one search interface on a web site is a key factor that makes this site a deep web one. For example, website A depicted in Figure 1.1 is a deep web site or, as Figure 1.3 shows, *Amazon.com* is a deep web site too as it allows a web user to search for books in the Amazon’s book database. Figure 1.3 schematically depicts a user interaction, in which a user searches for books via one of the interfaces on *Amazon.com* and gets a web page with search results. The search form shown in Figure 1.3 is not the only one available on **Amazon** – depending on his/her search task a user can use other forms which are depicted schematically in Figure 1.4. For example, books which descriptions are stored in book database can be searched via two different interfaces (called “*simple search*” at the Amazon’s front page, and “*advanced search*”) while search in music database can be performed using four different forms.

As mentioned earlier, current-day search engines do not index and, hence, miss a large number of documents in the deep Web. We illustrate this by the following two examples:

Example 1.1: The *AutoTrader.com* is one of the largest web services for car searching with over 3 millions used vehicles listed for sale by private owners, dealers, and manufacturers. One of AutoTrader’s search interfaces is available at <http://www.autotrader.com> (see Figure 1.5). If we query *AutoTrader.com* for any five-year-old “Ford Taurus”, it returns 1,916 possible vehicles (results are shown in Figure 1.5). However, such major web search engines as Google, Yahoo! and Live Search (search string is “*2002 Ford Taurus*” *site:autotrader.com*) return links to only 245, 262 and 299 pages from the **AutoTrader** correspondingly⁵. Browsing of search engine results further reveal that most of them are “car dealers” pages, each listing several cars offered by a dealer (including one 2002 Ford Taurus), rather than pages with descriptions of 2002 Ford Tauruses exclusively. Moreover, approximately one third of result pages are outdated in the sense that they did contain *2002 Ford Taurus* at the time of crawling but do not contain these terms at the time of our search. Thus, search engines index only a small subset of actual AutoTrader’s data (around 10% according to this example) and even those indexed pages are less relevant than original **AutoTrader** results.

⁴This definition is a slightly changed version of the one given by Chang et al.

⁵Unless otherwise stated all numbers in this section were retrieved at the end of September 2007. Note that the query itself is not ad hoc - it is easy to check that **AutoTrader** pages about other car models are indexed in a similar way.

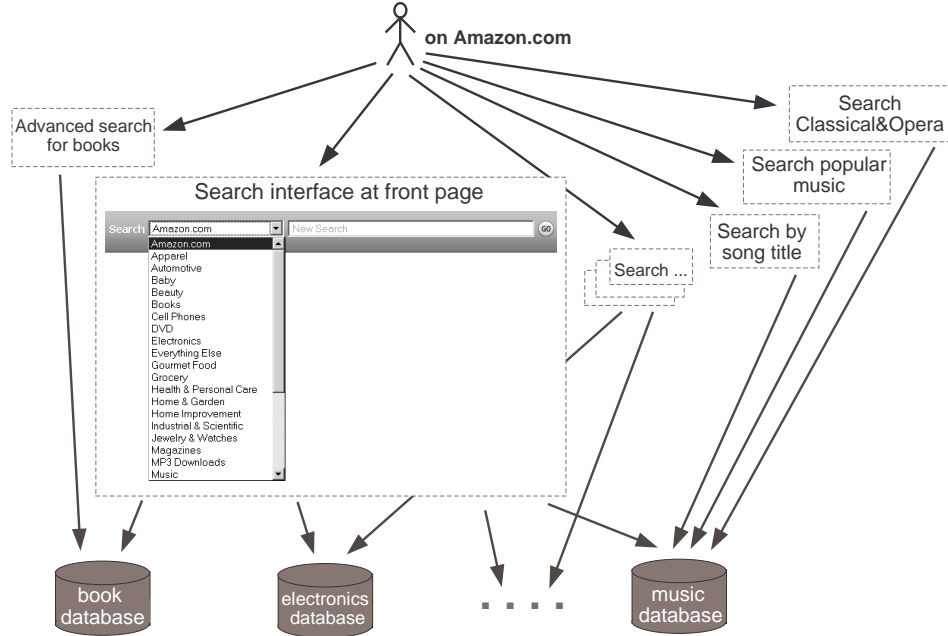


Figure 1.4: A number of search interfaces and databases on *Amazon.com*. “Advanced search for books” interface is shown in Figure 1.3.

Example 1.2: Consider the PubMed database [8]. The PubMed’s search interface and page with search results are depicted in Figure 1.6. Querying PubMed for documents with keywords “*breast cancer*” returns 163,871 matches while search on Google, Yahoo! and Live Search for pages on the PubMed site with “*breast cancer*” return 80,100, 4,440 and 12,200 results correspondingly⁶. Similar to the previous example, around one third of search engine results are links to some pages on the PubMed rather than links to articles in the PubMed database. Here one can see that coverage of deep web data may vary among searchers. Additionally, as we did issue the same queries to the PubMed database and to Google in July 2002 (when the PubMed had 120,665 articles vs. 3,810 results returned by Google), we certainly observe the gradual improvement in indexing of the PubMed data during past five years. This trend improvement in indexing of deep web data, is marked by arrows in Figure 1.2.

In addition, there is another noticeable feature of search engines’ results. Web searchers normally stop at about the 1000th result leaving all other matching links hidden from a user. For instance, as Example 1.2 demon-

⁶These numbers are, in fact, approximate because searchers show only result count estimates [104].

Find Your Car

Search new, used & certified.
Over 3 million listings!

Make Any Make

ZIP *

*Required

Next

Search Results

We found **1,916** Ford listings near New York, NY.

Year	Make / Model	Mileage	Price	
2002 Used	Ford Taurus SEL Sedan Color - Spruce Green Metallic * 3 Day money back guarantee * 30 Day/1,000 mile exchange policy * 3 month/5,000 mile 84 component limited warranty (on most vehicles) * Free 2 year... View Car Details	65,113 Mileage	\$10,997 Price	90 mi from ZIP SPRINGFIELD Springfield Ford Inc. Toll Free: 1-888-666-6318
2002 Used	Ford Taurus SES Color - Gray This vehicle has a 3.0L V6 FFV engine and an automatic transmission. It includes Power Windows, Alloy Wheels, Clock, Trip Odometer, Air... View Car Details	66,809 Mileage	\$10,536 Price	632 mi from ZIP Ray Viles Ray Viles Ford Toll Free: 1-888-221-6063
2002 Used	Ford Taurus SES Color - Blue Air Conditioning, Power Steering, Power Windows, Power Door Locks, 18 Wheel, Cruise Control, AM/FM Stereo, Single Compact Disc, Dual Front Air... View Car Details	88,726 Mileage	\$10,200 Price	1259 mi from ZIP LONG McARTHUR Ford Union Mercury Long McArthur, Inc. Toll Free: 1-888-557-4149

Figure 1.5: Autotrader's search interface and search results.

NCBI

PubMed

A service of the National Library of Medicine and the National Institutes of Health

www.pubmed.gov

All Databases PubMed Nucleotide Protein Genome Structure

Search PubMed for

Go Clear

Limits Preview/Index History Clipboard Details

All: **163871** Review: 19839

Items 1 - 20 of 163871

Page 1 of 8194 Next

- Puig T, Vazquez-Martin A, Relat J, Petriz J, Menendez JA, Porta R, Casals G, Marrero FF, Haro D, Brunet J, Colomer R.**
Fatty acid metabolism in breast cancer cells: differential inhibitory effects of epigallocatechin gallate (EGCG) and C75.
 Breast Cancer Res Treat. 2007 Sep 28; [Epub ahead of print]
 PMID: 17902053 [PubMed - as supplied by publisher]
- Green AR, Burney C, Oranger CJ, Pash EC, El-Sheikh S, Rakha EA, Powe DG, Macmillan RD, Ellis IO, Stylianou E.**
The prognostic significance of steroid receptor co-regulators in breast cancer: co-repressor NCOR2/SMRT is an independent indicator of poor outcome.
 Breast Cancer Res Treat. 2007 Sep 28; [Epub ahead of print]
 PMID: 17902051 [PubMed - as supplied by publisher]
- Grau JJ, Estrach T.**
Old masters as clinical photographers: multifocal breast cancer diagnosed 400 years ago.
 Breast Cancer Res Treat. 2007 Sep 27; [Epub ahead of print]
 PMID: 17902050 [PubMed - as supplied by publisher]
- Knowlden JM, Jones HE, Barrow D, Gee JM, Nicholson RI, Hutcheson IR.**
Insulin receptor substrate-1 involvement in epidermal growth factor receptor and insulin-like growth factor receptor signalling: implication for Gefitinib (Iressa) response and resistance.
 Breast Cancer Res Treat. 2007 Sep 28; [Epub ahead of print]
 PMID: 17902048 [PubMed - as supplied by publisher]
- Wang X, Southard RC, Alfred CD, Talbert DR, Wilson ME, Kilgore MW.**
MAZ drives tumor-specific expression of PPAR gamma 1 in breast cancer cells.
 Breast Cancer Res Treat. 2007 Sep 28; [Epub ahead of print]
 PMID: 17902047 [PubMed - as supplied by publisher]

Figure 1.6: PubMed's search interface and search results.

strated, even Google is likely to be aware of approximately eighty thousands documents with “*breast cancer*” on the PubMed site it actually returns links to only the first thousand of these documents, which is in sharp contrast to fully browsable and achievable PubMed results.

Search interfaces are formidable barriers for web crawlers, which are designed to follow hyperlinks but fail to fill and submit search forms. The examples above have shown that, indeed, information behind search interfaces are poorly covered by search engines. Now we discuss the scale of the problem, i.e., how many deep web sites and databases are on the Web.

1.2.1 Scale of Deep Web

Historically, the first web gateway to a relational database system was created by Arthur Secret at the CERN in September 1992 [11]. Less than eight years after it has been estimated that 43,000–96,000 deep web sites existed in March 2000 [25]. A 3-7 times increase during the period 2000-2004 has been reported in the surveys by Chang et al. [34, 35], where the total numbers for deep web sites in December 2002 and April 2004 were estimated as around 130,000 and 310,000 correspondingly. Moreover, even national- or domain-specific subsets of deep web resources are, in fact, very large collections. For example, in summer 2005 there were around 7,300 deep web sites on the Russian part of the Web only [101]. Or the bioinformatics community alone has been maintaining almost one thousand molecular biology databases freely available on the Web [47]. To sum up, deep Web is, beyond all doubts, a huge, important and largely unexplored frontier of the Web.

1.3 Motivation

Search engines are clearly among those that can benefit from the information behind search interfaces. Current searchers deal well with informational (give me the information I search for) and navigational (give me the URL of the site I want to reach) queries, but transactional (show me sites where I can perform a certain transaction, e.g., shop, access database, download a file, etc.) queries are satisfied only indirectly [28]. It is needless to say that data in the deep Web is indispensable to use when answering transactional queries that constitute about 10% of web search engine queries [63, 64]. The fact that the deep Web is still poorly indexed is another major issue for search engines, which are eager to improve their coverage of the Web.

Besides web search engines, at least two other parties also have a direct interest in the deep Web. First, content providers (e.g., librarians) are realizing that putting content “into Web-accessible databases will not make it easily available, because commonly used search engines do not crawl

databases” [110]. Due to a huge number of databases (estimated in the hundreds of thousands) available on the Web today this is becoming a serious problem. Indeed, a recently-created database with highly relevant information on some topic may remain almost invisible for everyone as people discover new content using web searchers that do a poor job of indexing deep web content [55]. Second, mining deep web resources is invaluable and often a must for specialized (also called vertical) search engines. These search services are intended for a focused audience and designed to find information on a specialized topic. For many topics, deep web sites are the only or the most important data sources – for instance, it is unlikely that search engine for apartment seekers does not aggregate information from at least several deep web resources devoted to apartment search.

In this thesis, we concentrate on three classes of problems around the deep Web: characterization of deep Web, finding and classifying deep web resources, and querying web databases. We describe our motivations for each class of problems in the following subsections.

1.3.1 Characterizing the Deep Web

Though the term deep Web was coined in 2000 [25], which is sufficiently long ago for any web-related concept/technology, we still do not know many important characteristics of the deep Web. Indeed, until now there are only two works (namely, [25, 34]) solely devoted to the deep web characterization and, more than that, one of these works is a white paper, where all findings were obtained by using proprietary methods. As an example, the total size of the deep Web (i.e., how much information can be accessed via search interfaces) may only be guessed.

Another matter of concern is that the surveys mentioned above are predominantly based on study of deep web sites in English. One can then expect that the reported estimates are most likely to be biased, especially owing to a steady increase in non-English web content. In this way, surveying of national segments of the deep Web is of interest not only to national communities but to the whole web community as well.

1.3.2 Automatic Finding of Search Interfaces

It stands to reason that search for cars or for biomedical articles are better performed via services similar to the ones described in Examples 1.1 and 1.2 (see Section 1.2) correspondingly. In other words, for many search tasks a search process should ideally consist of two steps:

1. Finding search interfaces to web databases of interest. It is a transactional search that can be performed using search engines or existing web directories.

2. Searching using interfaces found at the previous step.

The first step is not that naive as it sounds. In reality, finding search interfaces in a particular domain is not an obvious task especially if a user does not know this domain well [73]. A user may prefer to find search forms with help of existing directories of deep web resources rather than conventional search engines but, according to [34], even the largest such directory covers less than one sixth of the total number of web databases. Therefore, there is a need in a relatively complete directory of search interfaces available on the Web. The scale of the deep Web suggests that such directory cannot be built by manual efforts. Automatic detection of search interfaces will also be useful for ranking of search engines' results – web pages containing (detected by the method) search interfaces should have ranking priority when answering transactional queries.

1.3.3 Querying Web Databases

Retrieving information by filling out web search forms is a typical activity for a web user. This is all the more so as interfaces of conventional search engines are also web forms. At present, a user needs to manually provide input values to search interfaces and then extract required data from the pages with results. The manual filling out forms is not feasible and cumbersome in cases of complex queries but such kind of queries are essential for many web searches especially in the area of e-commerce. For example, a marketing analyst may provide lists of products and companies that are of his/her interest, so that when a form query tool encounters an interface requiring that a “company” or a “product” be filled-in, the query tool can automatically fill in many such forms. Of course, an analyst could have filled out the forms manually, but this process would be very laborious.

In this way, the automation of querying and retrieving data behind search interfaces is desirable and essential for the following tasks:

- Building automated web agents searching for domain-specific information (vertical search engines)
- Building deep web crawlers (see [91])
- Wrapping a web site for higher level queries
- Extracting and integrating information from different web sites

1.4 Challenges

Web pages in the deep Web (for example, see those in Figures 1.3, 1.5 or 1.6) are dynamic pages that are constructed by embedding database records into predefined templates. The content of these pages usually have implicit

structures or schemas and, thus, such pages are often referred to as semi-structured web pages [16]. In regard to how structured the data is, the deep web content is much different either from regular web pages that do not have any structure or from structured data as stored in databases. Such intermediate position of semi-structured data particularly means that many challenges of the deep Web are unique both to the database community working with structured data and to the information retrieval community dealing with unstructured content. Specifically, while web search engines do work well with unstructured web pages their general performance over pages in the deep Web (semi-structured data) is open to question. For instance, let us consider Example 1.1 again and let us assume that a web search engine is able to fully index all the content of the AutoTrader’s car database⁷. If so, the same search as in Example 1.1 would return at least 1,916 pages (besides the most relevant 1,916 pages with car descriptions the searcher would also return some number of less relevant pages, e.g., “car dealers” pages and so on). However, for the search engine any of the returned pages would be just unstructured data and, hence, it would not take into account at all such useful parameters as, for example, car price or car mileage. Because of that a web user would still prefer the AutoTrader interface to the search engine’s one since the results of the former are data in a structured form that gives a significant flexibility with managing results like sorting the results by attributes, refining query, etc. while the results of the latter are unstructured pages, which are hard to handle.

Web pages either with semi-structured or unstructured content are designed for human beings. In other words, information on the Web is overwhelmingly provided in a user-friendly way (convenient for human perception) rather than in a program-friendly way (convenient for processing by computer applications). This, in fact, explains the best part of challenges we face with the deep web data.

To begin with, search interfaces are not created and not designed to be used by computer applications. Such simple task (from the position of a user) as filling out search form turns out to be a computationally hard problem. For example, the Amazon’s search interface shown in Figure 1.3 is submitted correctly if a book title is typed into the second text field from above (it should be a title since the field is described by the label “Title”). This fact is trivial for any user⁸ who takes a quick glance at the interface but absolutely non-trivial for an application which, unlike a human being, cannot perceive anything from interface’s visual layout. Instead, an application analyzes the code of web page with the interface to detect bindings between so called field descriptors or labels and corresponding form fields.

⁷One way to do this is to retrieve, by issuing a set of certain queries, and then index all pages, each describing one car object stored in the database.

⁸Who knows English of course.

Recognition of field labels is a challenging task since there are myriads of web coding practices and, thus, no common rules regarding the relative position of two elements, declaring a field label and its corresponding field respectively, in the page code. The navigation complexity is another challenge to be considered. Particularly, navigating some web sites requires repeated filling out of forms many of which themselves are dynamically generated by server-side programs as a result of previous user inputs. For example, the **AutoTrader** web site produces a web page containing results after at least two successful form submissions (see Figures 5.1(a) and 5.1(b)). These forms are collectively called *consecutive forms*.

Automatic recognition of search interfaces is a challenging task too. The task can be formulated in the following way: for a given web page with a form, identify automatically whether a form is searchable (search interface to a database) or non-searchable. It is a tough problem due to a great variety in the structure and vocabulary of forms and even within a well-known domain (e.g., search for car classifieds) there is no common schema that accurately describes most search interfaces of this domain.

In a similar way, data extraction from pages in the deep Web is an arduous task as well. As we explained above, successful filling out form is not enough: pages with query results are not interesting in themselves. To make them useful, we need to extract structured data from these pages. However, since result pages are generated for a human-perception they contain lots of non-informative elements (headers, advertisements, menus, images and so on) that prevent an application from retrieving structured content. In particular, there are typically lots of unnecessary elements at the top or at the left part of a page with query results. Moreover, each deep web site has its own design, i.e., rules how informative and non-informative data are presented.

1.5 Contributions

The thesis contributions are summarized below:

- We propose two new methods for estimating the main parameters of the deep Web.
- We use the suggested methods to estimate the scale of one specific national segment of the Web and report our findings.
- We build and make publicly available a dataset describing more than 200 web databases on the national segment of the Web.
- We describe the architecture of the I-Crawler, a system for finding and classifying search interfaces. Specifically, the I-Crawler is intentionally designed to be used in deep Web characterization studies and for constructing directories of deep web resources.

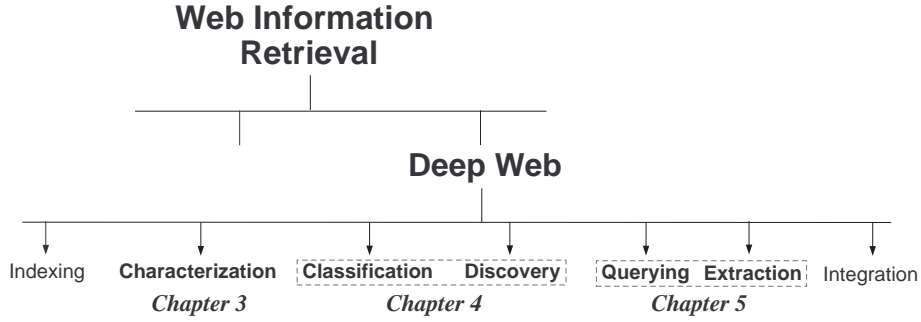


Figure 1.7: Topics addressed in this thesis.

- Unlike almost all other approaches to the deep Web existing so far, the I-Crawler is able to recognize and analyze JavaScript-rich and non-HTML searchable forms.
- We present a data model for representing search interfaces and discuss techniques for extracting field labels, client-side scripts and structured data from HTML pages. We also describe a representation of result pages and discuss how to extract and store results of form queries.
- We present a user-friendly and expressive form query language that allows one to retrieve information behind search interfaces and extract useful data from the result pages based on specified conditions.
- We implement a prototype system for querying web databases and describe its architecture and components design.

1.6 Scope and Organization of this thesis

This thesis focuses on the deep Web, and we study deep Web at several different levels. Main topics addressed are characterizing the deep Web, automatic finding and classifying search interfaces, and issuing queries to web databases in an automatic way. We aim at designing and developing: an efficient system to retrieve data from the deep Web and an intelligent tool to automatically find and classify web databases. Additionally, our characterization efforts give an insight into the structure of the deep Web.

Figure 1.7 depicts the relation of studied topics to the area of web information retrieval in general.

The following is an outline of the contents of this thesis:

- This chapter introduced the concept of deep Web, presented our motivation and challenges, and summarized the contributions of this thesis. Some of the work in this chapter is reported in [99].

- Chapter 2 reviews selected publications related to the topics covered in this thesis.
- Chapter 3 discusses current state of deep web characterization studies. We propose two new methods for estimating the main parameters of deep Web. We describe two consecutive surveys of one national segment of the Web performed based on the proposed techniques in 2005 and in 2006 and report our findings. Some of the work in Chapter 3 is reported in [101, 102, 103].
- Chapter 4 describes the architecture of the system for finding and classifying search interfaces. We discuss how it can be used in deep Web characterization studies and for constructing deep web directories. We also report results of our preliminary experiments.
- Chapter 5 describes our approach to query the deep Web. In particular, we propose a data model for representing and storing HTML forms, and a web form query language for retrieving data from the deep Web and storing them in the format convenient for additional processing. We present a novel approach in modeling of consecutive forms and introduce the concept of the super form. We implement a prototype system based on the discussed methodologies. Some of the work in Chapter 5 is reported in [98, 100].
- Chapter 6 summarizes our contributions and provides some guidelines for future work in this area.

Finally, bibliography includes over 115 references to publications in this area. The next chapter is a survey of the most important works in the context of this thesis.

Chapter 2

Related Work

Today the World Wide Web (WWW) is a huge repository of web resources hosted by a large number of autonomous web sites. It is an extremely important source of information and the role of WWW as the main information source becomes more significant day by day. The impetuous growth of web data and the variety of web technologies require researchers to develop usable web tools for classifying, searching, querying, retrieving, extracting, and characterizing web information. In this literature review we try to cover some works devoted to the indicated problems. We classify the description of related research into five parts:

- **Deep Web:** The formidable part of the Web known as the deep Web is not “crawlable” by traditional search engines [25]. Web pages in the deep Web are dynamically generated in response to queries submitted via search interfaces to web databases. The deep Web provides more relevant and high-quality information in comparison with the “crawlable” part of the Web.
- **Querying search interfaces:** A brief review of web query languages providing some mechanisms to navigate web search forms in the queries is given in this part. We also consider several architectures of form query systems and designate the limitations of these systems.
- **Extraction from data-intensive websites:** Pages in data-intensive sites are automatically generated: data are stored in a back-end DBMS, and HTML pages are produced using server-side web scripts from the content of the database. The **RoadRunner** system [38], which has been specifically designed to automate the data extraction process, is described.
- **Deep Web characterization:** Two key works devoted to the characterization of the deep Web are described in this section. Special

attention is paid to the methods for estimating the principal parameters of the deep Web.

- **Finding deep web resources:** Finding of search interfaces to web databases is a challenging problem. We discuss existing approaches and their role in efficient location of the entry points to deep web resources [22].

2.1 Deep Web

The concept “deep Web” appearing in several works in recent years requires special attention because of its significance to the problem described in this thesis. This section is devoted to the brief description of the most popular form of web query systems, search engines, their shortcomings, the increase of the dynamic content in the Web, and, at last, classification of searchable web databases.

2.1.1 Search Engines

Current day search engines [27, 59] (such as Yahoo! or Google) provide a simple interface to a database of Web pages. A user submits keywords of interest and gets back search results; web pages with links to documents containing the keywords. Note that search engines may search Internet sources other than Web pages: most commonly the archives of Usenet newsgroups. Many also have e-mail address directories and a directory of Internet resources arranged by topic. Some search engines allow users to search just the document title or the URL.

Unfortunately crawlers have limited capabilities for retrieving the information of interest and the significant irrelevance of search results. The following shortcomings of the existing search engines should be noted:

- **Lack of support of metadata queries:** The routine information about web document: author, date of last modification, length of text and subject matter (known as metadata) is very hard to be reliably extracted by current-day programs. As a result, little progress has been done by search engines in exploiting the metadata of web documents. For example, a web crawler might find not only the desired articles authored by Tim Berners-Lee but also find thousands of other articles in which this name is mentioned in the text or in a reference.
- **Lack of support of querying interlinked documents:** The hypertext nature of the Web is not so far exploited enough by search engines. Most web crawlers fail to support queries utilizing link information and to return such information as part of a query’s result.

Searches related to the hypertext structure of sites are believed to be very important but such searches are difficult to pose by the query mechanism offered by search engines.

- **Lack of support of structural queries:** Search engines do not exploit the internal structure of Web documents. One cannot formulate query based on hierarchical nature of the tag elements in a query. For example, if we are looking for documents that enlist the list of features of a notebook in a table in the web page, then a search engine may return thousands of documents containing wide variety of information related to notebooks most of which are not structured in the form of a table.
- **Lack of support of numeric comparisons:** The search is limited to string matching. Numeric comparisons cannot be done. Thus, for instance, the following query cannot be expressed: *Find hotels in Helsinki with rate per night less than \$80.*
- **Ignoring the geographical scope of web resources:** Current web crawlers have restricted capabilities in identifying the geographical scope of pages when computing query results. Therefore, query results include resources that are not geographically relevant to the user who issued the query. For instance, finding hotels, car rentals, and restaurants in or near specific region is not a simple task with current web search engines.
- **Limitation of querying dynamic Web pages:** Dynamic web pages are created based on a user's query by assembling information from one or more databases. These pages are not analyzed and indexed by web crawlers. Thus, search results ignore dynamic web documents. At first blush it may appear inessential but ignoring of dynamic pages by search engines means that the part of the Web as large as about 400 sizes of the static Web is disregarded. We describe the deep or hidden Web (consisting of dynamic web pages) in the next subsection.

2.1.2 Deep Web

Recent studies [25, 71, 72] have designated that a huge amount of content on the Web is dynamic. The classification of dynamic web content is given in [90]. Lawrence and Giles [71] estimated that about 80% of the content on the Web is dynamically generated. Technologies widely-used for dynamic content generation [1, 7, 60, 113] continue to increase the ratio between dynamic and static content in favor of the former.

The amount of dynamic web content is very considerable but only a little part of it is being crawled and indexed. Current day crawlers cover a portion of the Web called the *publicly indexable Web (PIW)* [71] but do

not crawl pages dynamically generated in response to queries submitted via the search HTML-forms, and pages that require authorization or prior registration. While this portion of the Web is not accessible through current search engines its dynamic content greatly outnumbers the PIW content in size and in relevancy of data contained. Most commonly, data in the non-indexed part of the Web is stored in large searchable databases and is accessible by issuing queries guided by web forms [114]. This large portion of the Web (called the hidden Web [46] or deep Web [25]) is “hidden” behind search forms in databases. The hidden Web is particularly important, as organizations with large amounts of information are gradually making their databases available online. It is often the case that a user can obtain the content of these databases only through dynamically generated pages, delivered in response to web queries.

It was estimated in [25] that:

- Public information on the deep Web is 400 to 550 times larger than the commonly defined Web.
- The deep Web contains 7,500 terabytes of information compared to nineteen terabytes of information in the surface Web (publicly indexable Web).
- The deep Web contains nearly 550 billions individual documents compared to the one billion of the PIW.
- The deep Web sites tend to return 10% more documents than the surface Web sites and nearly triple the number of quality documents.
- Sixty of the largest deep Web sites collectively contain about 750 terabytes of information (about forty times the size of the PIW).
- More than 200,000 deep web sites exists.

The deep Web is based on a generous amount of databases that can be accessed only through web search interfaces. Apparently, a web user needs to know which searchable databases are most likely to contain the relevant information that he/she is looking for. The next subsection describes the current state in web databases’ classification.

2.1.3 Classification of Web Databases

The accurate classification of searchable databases into topic hierarchies makes it easier for end-users to find the relevant information they are seeking on the Web. Yahoo!-like [10] directories available on many web portals may be used to find databases of interest hidden behind web forms. Also, several commercial web sites have recently started to manually classify searchable web databases into Yahoo!-like hierarchical classification schemes, so that web users can browse these categories to find the databases of interest. For

example, **InvisibleWeb** [97] contains a directory of over 10,000 databases, archives and search engines. The **InvisibleWeb** directory helps a web user to find locations of databases covering the user search subject. For instance, the user may search for any *used Ford Escort cars not older than 5 years*, the locations of web forms that allow to query databases containing information about used cars can be found by entering “*used cars*” in the search form of **InvisibleWeb**. Lu et al [75] described a method to cluster e-commerce search engines on the Web. Their approach utilizes the features available on the interface pages of such engines, including the label terms and value terms appearing in the search form, the number of images, normalized price terms as well as other terms. The experimental results based on more than 400 e-commerce search engines have demonstrated a good clustering accuracy of the approach.

The automation of the above search process is by using metasearchers [53]. A metasearcher receives a query from a user, selects the best databases to which to send the query, translates the query in a proper form for each search interface, and merges the results from the different sources. Works in [83, 115, 51] have been devoted to the interaction with searchable databases, mainly in the form of metasearchers. When the meta-searcher receives a user query, it consults its collected metadata and suggests to the user sources to try. This solution may not be as accurate as submitting the query to all sources, since the suggestions are only based on collection metadata. However, the query overhead is much less since queries are not executed everywhere. The problem of identifying document sources based on exported metadata is called the *text-source discovery problem*.

In [51], some solutions to the text-source discovery problem have been studied. The family of solutions was called GLOSS (*Glossary-of-Servers Server*). In particular GLOSS meta-searchers use statistical metadata, i.e., how many times each term occurs at each source. As it was shown, these “summaries” are small relative to the collection, and because they only contain statistics, are much easier to export by a source. Statistical summaries can be obtained mechanically, and hence are superior to manually produced summaries that are often out of date. Similarly, since they summarize the entire collection, they are better than summaries based on a single field (such as titles). GLOSS works best with a large collection of heterogeneous data sources. That is, the subject areas covered by the different data sources are very distinct from each other. In this case, the statistical summaries used by GLOSS strongly discriminate each source from the others. It should be noted that Gravano et al. do not compare the single and multiple engine scenarios. Firstly, in many cases one is not given a choice. For example, the documents may be owned by competing organizations that do not wish to export their full collections. On the Web, for instance, growing numbers of documents are only available through search interfaces, and hence un-

available to the crawlers that feed search engines. Secondly, if a choice is available, the factors to consider are very diverse: copyright issues regarding the indexing or warehousing of documents, the cost and scalability (storage, operations) of maintaining a single index, the frequency at which new documents are indexed, and the accuracy of the results obtained. Instead, the authors only consider a multiple-engine scenario, and study GLOSS solutions to the text-discovery problem. The “accuracy” of these solutions to what could be obtained by sending a query to all underlying search engines are compared.

The work in [115] evaluates the retrieval effectiveness of distributed information retrieval systems in realistic environments. The most important issue in searching a set of distributed collections is how to find the right collection to search for a query. The sheer number of collections in a realistic environment makes exhaustive processing of every collection infeasible. Furthermore, many collections are proprietary and may charge users for searching them. The only method for timely and economic retrieval is to constrain the scope of searching to those collections which are likely to contain relevant documents for a query. For this purpose, the proposed distributed information retrieval system adopted a widely used technique: it creates a *collection selection index*. This index consists of a set of *virtual documents*, each of which is a light-weight representation of a collection. Specifically, the virtual document for a collection is simply a complete list of words in that collection and their frequencies (numbers of containing documents). When a query is posted, the system first compares it with the virtual documents to decide which collections are most likely to contain relevant documents for the query. The document retrieval takes place only at such collections. According to the authors’ view, the virtual documents are a very concise representation, requiring less than 1.0% of space taken by the underlying document collections.

In [83], Meng et al. attacked the following problems, namely, the *database selection problem* and the *collection fusion problem*:

- Select databases that need to be searched and estimate the number of globally most similar documents in each database.
- Decide which documents from each selected database to retrieve.

In [83], the usefulness of a database to a query is defined to be the number of documents in the database that have potentials to be useful to the query, that is, the *similarities* between the query and the documents as measured by a certain global function are higher than a specified threshold. The authors have also studied the problem of guaranteeing all globally most similar documents from each local database be retrieved. The proposed solutions aim at minimizing the number of documents that are not globally

most similar to be retrieved while guaranteeing that all globally most similar documents are retrieved.

The research in [62] is devoted to automatically populating the given classification scheme with searchable databases where each database is assigned to the “best” category or categories in the scheme. This study builds on the preliminary work in [61] on database classification. Authors define a hierarchical classification scheme like the one used by *InvisibleWeb* as follows: *A hierarchical classification scheme is a rooted directed tree whose nodes correspond to (topic) categories and whose edges denote specialization. An edge from category v to another category v' indicates that v' is a subdirectory of v .*

The proposed algorithm does not retrieve or inspect any documents or pages from the database, but rather just exploits the number of matches that each query probe generates at the database in question. The *confusion matrix* [85] is built on the basis of query probing. Further, according to the document category distribution and the *Classify* algorithm each database is classified in a hierarchical classification scheme.

2.2 Querying Search Interfaces

Retrieving and querying web information have received considerable attention in the database research communities [46]. In this section, we review the research literature on querying and automatically filling out HTML forms.

2.2.1 W3QL and W3QS

W3QS (WWW Query System) [68, 69] is a project to develop a flexible, declarative, and SQL-like Web query language, W3QL. This language integrates both content and structure-based queries on web pages. It provides a novel mechanism to navigate forms in the queries. The underlying data model of W3QL includes nodes and links. Nodes refer to static web pages and those created dynamically by CGI scripts upon form submissions. Links represent the hyperlinks within web pages. It is noted that forms and their returned pages are connected by links in the data model. The example below illustrates the form query capability of W3QL:

Example 2.1: Suppose we want to use the form embedded in `http://lycospro.lycos.com` (see Figure 2.1) to find postscript files dealing with the subject of multimedia information kiosk. The corresponding W3QL query can be expressed as follows:

```
1  Select n3:
2  From n1, l1, (n2, l2), l3, n3
```

Figure 2.1: The Form Interface at <http://lycospro.lycos.com>.

```

3  where n1 in {http://lycospro.lycos.com};
4  Run learnform n1 cs76:0 if n1 unknown in Report;
5  Fill n1 as in Report with query =
    'multimedia information kiosk';
6  l1: PERLCOND 'l1.content = ~/^FORM/i';
7  n3: PERLCOND '(n3.format = ~/postscript/i)&&
8      (n3.content = ~/multimedia information kiosk/i)'
9  Using ISEARCHd

```

In this query, *n1*, *n2* and *n3* represent nodes, and *l1*, *l2* and *l3* represent links. The node *n3* represents the desired result. In line 2, the node *n1* in the From clause contains the form to be queried. The pair (*n2*, *l2*) represents an *unbounded length path* to reach the desired node *n3*, and the web page returned by the form is the first node of the path. The **Report** file in lines 4 and 5 is a *Database Of Forms* (DOF) supported by W3QS. **Learnform** in line 4 is an external program that deals with unknown forms encountered during a search. It takes two parameters: the node containing the form that needs to be learned, and the user's X-terminal name. In this case, the two parameters are *n1* and *cs76:0* respectively. Line 5 in the query specifies that the form is supplied with the input string "*multimedia information kiosk*". In W3QS, a form can also be completed based on past form-filling knowledge maintained by the DOF file. Line 6 constrains the link *l1* to be a form link instead of an ordinary hypertext link. Lines 7 and 8 specify the format and content condition that *n3* should satisfy respectively. The **ISEARCHd** stated in line 9 is a remote search program that implements different search algorithms for loading web pages from WWW and evaluating

their relevance.

W3QS offers mechanisms to learn, store, and query forms. However, it supports only two approaches to complete a form: either using past form-fill knowledge or some specific values. No other form-filling methods, such as filling out a form using multiple sets of input values or values obtained from queries to relational tables are provided. Furthermore, W3QS is not flexible enough to get a chain of multiple web pages returned by a form. For instance, if we would like to obtain a chain of first three result web pages generated by the Lycos server, we can construct a W3QL query as follows:

```
1  Select n2, n3, n4:
2  From n1, l1, n2, l2, n3, l3, n4
3  where n1 in {http://lycospro.lycos.com};
4  ...
```

We omit the other part of the query for simplicity. In the above query, `n2`, `n3`, and `n4` represent the chain of first three web pages returned by Lycos respectively. However, if Lycos generates only two web pages as the result of the query, W3QS will fail to obtain them since they do not satisfy the query. In this case, another query that returns a chain of one or two web pages has to be designed. These shortcomings have been addressed by our proposed web form query language.

2.2.2 ARANEUS

ARANEUS is a research project that introduced a set of tools and languages for managing and restructuring data coming from the World Wide Web [18, 81]. The data model of the project, called ARANEUS Data Model (ADM), models the internal structure of web pages as well as the structure of the web sites. Based on ADM, two languages, ULIXES and PENELOPE, are designed to support querying and restructuring process. ULIXES is used to build relational views over the web. The views can then be analyzed and integrated using standard database techniques. ULIXES queries are able to extract relational tuples from web pages that are defined with page schemes. PENELOPE on the other hand can be used to produce hypertext views from relational views.

In ADM, a set of homogeneous web pages can be modelled by a page scheme. A page scheme consists of one or more components corresponding to different parts of the web pages. HTML forms can be components in the page scheme. A form component is considered as a virtual list of tuples; each tuple has as many attributes as the fill-in fields of the form, plus a link to the resulting page. A web site can be seen as a collection of page schemes connected by links. Based on the page schemes, ULIXES can derive relational view over the web. We will illustrate queries on page schemes by

the UNIVERSITY of GREENWICH

.....internal
Staff Telephone Directory

Old Royal Naval College, Park Row, Greenwich, London SE10 9LS
020 8331 8000

< HOME PAGE < PREVIOUS PAGE

Useful Phone Books

PhoneBookUK
UK On-line Phone Directory

Yellow Pages
Yellow Pages On-line

Scoot
Business Directory Search

Staff Directory

You may search by surname or extension number.

Search for

Figure 2.2: Search interface on the Greenwich University’s website.

the following example:

Example 2.2: Figure 2.2 shows an HTML form for querying staff information at the University of Greenwich (http://www.gre.ac.uk/staff_directory). Figure 2.3 depicts the result web page returned by the form for staff with “Green” as their surname. Two page schemes, namely **Staff_SearchPage** and **Staff_ResultPage**, have been derived for the web pages in Figure 2.2 and Figure 2.3 respectively.

```

1.1 PAGE SCHEME Staff_SearchPage
1.2         Email:          TEXT;
1.3         NameForm:      Form    (SEARCH: TEXT;
1.4                                     Submit: LINK TO
                                       Staff_ResultPage;);
1.5         RelevantLinks:LIST OF(Label: TEXT;
1.6                                     RLinks: LINK TO
                                       RelevantPage;);

1.1 PAGE SCHEME Staff_ResultPage
1.2         StaffList:      LIST OF(Name: TEXT;
1.3                                     Department: TEXT;
1.4                                     ToPersonalPage: LINK TO
                                       Staff_Page;);

```

In the page scheme **Staff_SearchPage**, the form defined by **NameForm** consists of two attributes: a text field, and a link to the result web page defined by the page scheme **Staff_ResultPage**. The **Staff_SearchPage** also includes a list of hyperlinks leading to other web pages defined by the

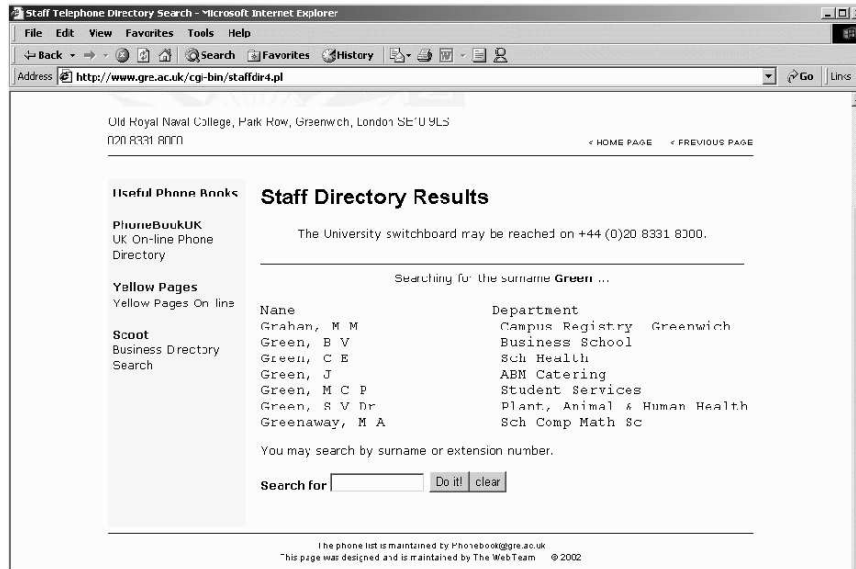


Figure 2.3: Search results for staff members with surname “Green”.

RelevantPage scheme. The page scheme, `Staff_ResultPage`, defines the returned staff information as a list of tuples consisting of names, departments and links to personal web pages. For simplicity, we have omitted description on the `RelevantPage` and `Staff_Page` page schemes.

Based on the above page schemes, a ULIXES query to find all the staff members with “Green” as surname and their departments can be formulated as follows:

```

1 DEFINE TABLE Staff_Dpt(Staff_Name, Dpt_Name)
2 AS Staff_SearchPage.NameForm.Submit →
   Staff_ResultPage.NameForm.Stafflist
3 IN GreenwichScheme
4 USING Staff_ResultPage.StaffList.Name,
5 Staff_ResultPage.StaffList.Department
6 WHERE Staff_SearchPage.NameForm.SEARCH = 'Green'
```

The dot operator (.) in this query denotes navigation within a page, and the link operator (→) denotes an inter-pages link. Line 1 defines the schema of the resultant relation. Line 2 describes the navigational expression to the web page returned by the form. `GreenwichScheme` in line 3 represents the set of page schemes for the web site of the University of Greenwich. Lines 4 and 5 define the mapping from the attributes of the page scheme `Staff_ResultPage` to relational attributes. Line 6 describes how the input value is given to the form.

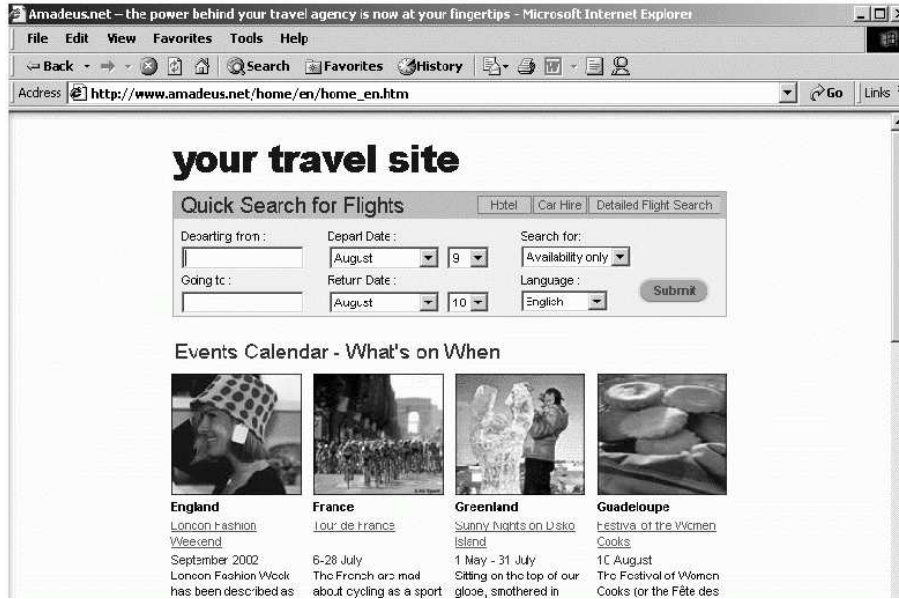


Figure 2.4: First form for finding flight information on the Amadeus web site.

It is noted that ADM's form representation is rather simple as it has omitted many important field types, such as checkbox, select, and others. As a result, the ADM's form type may not be able to represent most existing forms. Furthermore, without a database of forms, ADM queries have to involve web pages embedding the forms to be queried, e.g., `Staff_SearchPage.NameForm.SEARCH`. The overheads to download a web page and to extract its forms may slow down the query evaluation. Also, the query expressions may also look more complicated. In our proposed web form model, we allow forms to be pre-stored in a form database. A form can be queried directly without retrieving its HTML source.

2.2.3 Database System for Querying Forms

In [39], Davulcu et al. proposed a three-layered architecture for designing and implementing a database system for querying forms. Similar to the schema architecture of traditional relational databases, the proposed architecture consists of virtual physical layer, logical layer, and external schema layer. The lowest layer, virtual physical layer, aims to provide navigation independence, making the steps of retrieving data from raw web sources transparent to the users. Data collected from different web sites may have semantic or representational discrepancies. These differences are resolved at the logical layer that supports site independence. The external schema

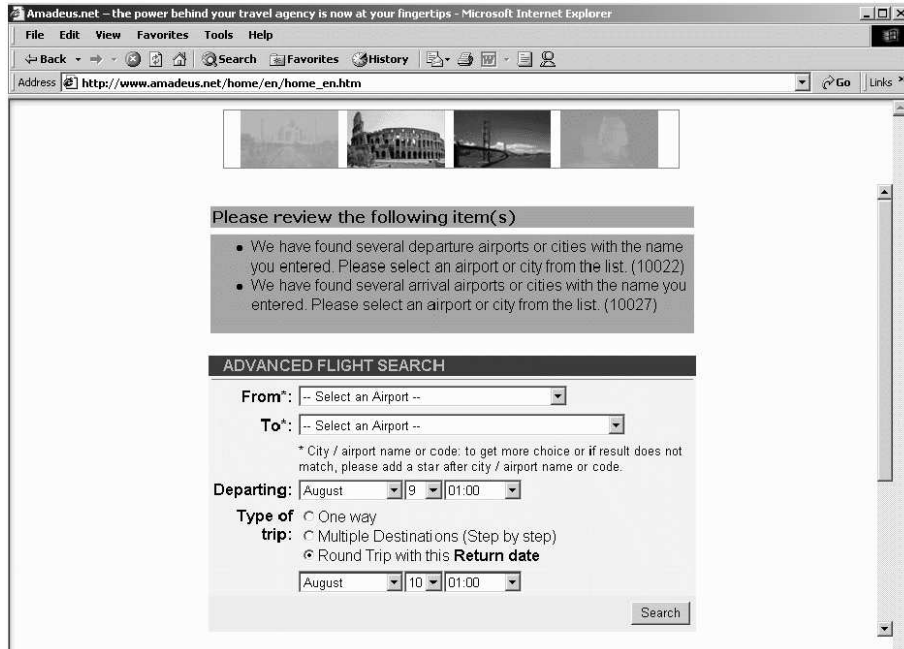


Figure 2.5: Second form for finding flight information on the Amadeus web site.

layer, extending the Universal Relation (UR) concept [77], provides a query interface that can be used by naive web users.

Unlike the other projects, the work by Davulcu et al. addresses the problem of navigating multiple consecutive forms. Consecutive forms refer to cases where the web page returned by a form request contains another form. To describe how forms can be navigated, a subset of serial-Horn Transaction F-logic [67] has been used to represent web pages, HTML forms, and other web objects. The same logical language is used to define *navigation expression* that describes how to access desired information residing in a site (or several sites). We illustrate it with the following example:

Example 2.3: Suppose we want to find flight information from the Amadeus¹ travel site. At this web site, flight information can be obtained in two steps. Firstly, we specify the source and destination cities, the departure and return dates in the form interface shown in Figure 2.4. Upon submitting the form request, a second form is returned as shown in Figure 2.5. We now specify the airports and the returned web page is shown in Figure 2.6. These two steps can be represented by a navigation expression as shown below:

¹http://www.amadeus.net/home/en/home_en.htm.

Flight	Departing	Arriving	Stops Aircraft	Duration	Seats available		
					1st	Biz	Eco
Singapore Airlines SQ 318	Changi (SIN), Singapore	09:00 Heathrow (LHR), London, United Kingdom	15:00 Non-stop /44	13h30mir	Yes	Yes	Yes
Virgin Atlantic VS 731C OP	Changi (SIN), Singapore	09:00 Heathrow (LHR), London, United Kingdom	15:30 Non-stop 744	13h30mir	N/A	N/A	Yes
Virgin Atlantic VS 732C OP	Changi (SIN), Singapore	12:35 Heathrow (LHR), London, United Kingdom	19:05 Non-stop 744	13h30mir	N/A	N/A	Yes
Singapore Airlines SQ 320	Changi (SIN), Singapore	12:35 Heathrow (LHR), London, United Kingdom	19:05 Non-stop 744	13h30mir	Yes	Yes	Yes
British Airways BA 733E OP	Changi (SIN), Singapore	22:45 Heathrow (LHR), London, United Kingdom	05:15 + 1 Non-stop day(s) 747	13h30mir	Yes	N/A	Yes
Qantas Airways LH 002	Changi (SIN), Singapore	22:45 Heathrow (LHR), London, United Kingdom	05:15 + 1 Non-stop day(s) /44	13h30mir	Yes	No	Yes

Figure 2.6: Search results for flight information on the Amadeus web site.

```

1 Flight(FlightDesc, Departing, Arriving, Duration) ←
2   AmadeusQuery1Pg.action:submit_form[object → form(f1);
3   doit@(DepartingFrom, GoingTo, DepartDate, ReturnDate) →
4     AmadeusQuery2Pg]
5   ⊗ AmadeusQuery2Pg.action:submit_form[object → form(f2);
6     doit@(From, To) → AmadeusResultPg]
7   ⊗ AmadeusResultPg.data_page[extract
8     tuple(FlightDesc, Departing, Arriving, Duration)]

```

In the above expression, the three web pages shown in Figures 2.4, 2.5 and 2.6 are represented by `AmadeusQuery1Pg`, `AmadeusQuery2Pg` and `AmadeusResultPg` respectively. Line 1 defines the relation schema to be extracted from the result page. Lines 2 and 3 describe how the first form in Figure 2.4 denoted by `f1` can be navigated to reach the second web page, `AmadeusQuery2Pg`. Lines 4 and 5 in turn describe how the form in `AmadeusQuery2Pg` can be navigated to obtain the web page `AmadeusResultPg`, where the flight instances reside. Line 6 and 7 specify the data extraction action to be performed on the page `AmadeusResultPg`. The authors mentioned that the complex navigation expression can be generated semi-automatically when a user navigates the web site using a web database system.

Our work differs from the work by Davilku et al. in two aspects. Firstly, we propose more advanced web form representation and user-friendly language for defining form queries. Secondly, we do not treat the form query

results as relations. For most web pages it is extremely difficult to transform the web page content into some relation table, so we limit our extraction process to extraction the useful data. We represent result pages (returned web pages that contain query results) as containers of result matches, each of which containing informative text strings and hyperlinks.

2.2.4 Crawling the Hidden Web

Another closely related work to querying HTML forms is the Hidden Web Exposer (HiWE) project at Stanford [90, 91]. Raghavan and Garcia-Molina propose a way to extend crawlers beyond the publicly indexable Web by giving them the capability to fill out Web forms automatically. Since developing a fully automatic process is quite challenging, HiWE assumes crawlers will be domain specific and human assisted. Starting with a user-provided description of the search task, HiWE learns from successfully extracted information and updates the task description database as it crawls. Besides, it also provides a label matching approach used to identify elements in a form, based on layout position, not proximity within the underlying HTML code. The authors also present the details of several ranking heuristics together with metrics and experimental results that help evaluate the quality of the proposed process.

Figure 2.7 illustrates the complete architecture of the HiWE crawler. It includes six basic functional modules and two internal crawler data structures. The basic crawler data structure is the *URL List*. It contains all the URLs that the crawler has discovered so far. When starting up the crawler, the *URL List* is initialized to a seed set of URLs. The *Crawl Manager* controls the entire crawling process. It decides which link to visit next, and makes the network connection to retrieve the page from the Web. The Crawl Manager passes the downloaded page over to the *Parser* module. In turn, the Parser extracts hypertext links from the page and adds them to the URL List structure. To process forms and extract hidden content, HiWE employs four additional modules and the *LVS Table*. The *LVS Manager* is responsible for managing additions and accesses to the LVS table. It provides an interface for various application-specific data sources to supply new entries to the table.

HiWE uses LITE (Layout-based Information Extraction) to extract information from both form and response pages. LITE is a new technique where in addition to the text the physical layout of a page is also used to aid in extraction. LITE is based on the observation that the physical layout of different elements of a Web page contains significant semantic information. For example, a piece of text that is physically adjacent to a table or a form widget (such as a text box) is very likely a description of the contents of that table or the purpose of that form widget.

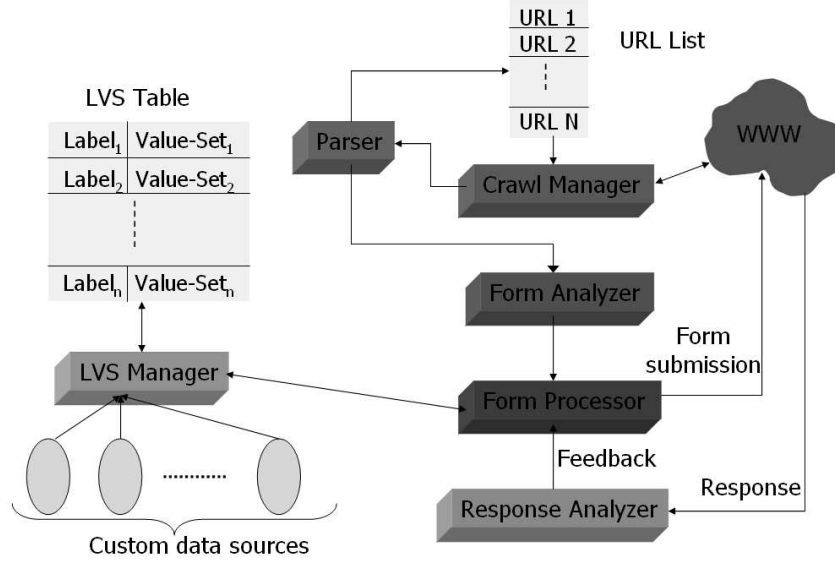


Figure 2.7: HiWE architecture.

The work in [65] has also presented a variety of text-based techniques for matching form elements to descriptive form text labels. The techniques in [65] are based on a detailed study of the most common ways in which forms are laid out on web pages.

There are some limitations of the HiWE design that if rectified, can significantly improve performance. The first limitation is inability to recognize and respond to simple dependencies between form elements (e.g., given two form elements corresponding to states and cities, the values assigned to the “city” element must be cities that are located in the state assigned to the “state” element). This problem is closely connected with the querying of consecutive forms. If the submission of one form generates another form then dependencies between the fields of these forms often exist. For instance, the select values in the “model” field of the second *AutoTrader* form shown in Figure 5.1(b) depends on the choice of “make” in the first *AutoTrader* form (see Figure 5.1(a)). Secondly, although the HiWE provides a very effective approach to crawl the deep Web, it does not extract data from the resulting pages.

The DeLa (Data Extraction and Label Assignment) system [109] sends queries via search forms, automatically extracts data objects from the resulting pages, fits the extracted data into a table and assigns labels to the attributes of data objects, i.e., columns of the table. The DeLa adopts the HiWE to detect the field labels of search interfaces and to fill out search interfaces. The data extraction and label assignment approaches are based

on two main observations. First, data objects embedded in the resulting pages share a common tag structure and they are listed continuously in the web pages. Thus, regular expression wrappers can be automatically generated to extract such data objects and fit them into a table. Second, a search interface, through which web users issue their queries, gives a sketch of (part of) the underlying database. Based on this, extracted field labels are matched to the columns of the table, thereby annotating the attributes of the extracted data.

The HiWE system (and hence the DeLa system) works with relatively simple search interfaces. Besides, it does not consider the navigational complexity of many deep web sites. Particularly, resulting pages may contain links to other web pages with relevant information and consequently it is necessary to navigate these links for evaluation of their relevancies. Additionally, obtaining pages with results sometimes requires two or even more successful form submissions (as a rule, the second form is returned to refine/modify search conditions provided in the previous form). We addressed these challenges and proposed a web form query language called DEQUEL that allows a user to assign more values at a time to the search interface's fields than it is possible when the interface is manually filled out. For instance, data from relational tables or data obtained from querying other web databases can be used as input values.

2.2.5 Modeling and Querying the World Wide Web

The work in [74] introduces a set of mechanisms for manipulating and querying web forms. This research mainly focuses on the query issues related to forms, the extraction of useful data from web pages returned by forms has been excluded from the research scope. The proposed solution requires web pages returned by forms to be stored in a web warehouse. It is considered as more appropriate and flexible to perform extraction on the stored web pages instead of incorporating extraction features into the proposed form query language. The study proposes:

- **Methodology for querying forms:** To derive useful information from forms, the methodology is proposed to describe the essential steps in querying forms.
- **Modeling and storage of form objects:** Before a form can be queried, it has to be appropriately modeled. A form object model represents the important form attributes that can be used in form queries. Based on the form object model, a form database can be implemented to store form objects for future queries.
- **Representation of dynamic web pages returned by forms:** To allow web pages returned by forms to be involved in form queries and

to be stored for future data extraction, some schema is defined to describe these web pages.

- **Design of a Form Query Language:** A form query language is proposed. The language supports flexible form queries. Special consideration has been given to form queries that involve multiple sets of input values. The form query language also supports navigation of web pages returned by forms. A form query processor is developed to support the proposed form query language.
- **Development of a Form Access and Storage System (FAST):** Based on the proposed form query methodology, A prototype system is developed to store form objects embedded in web pages and evaluate queries on them.

The design and implementation of [74] is based on the research project on web warehousing known as WHOWEDA [29, 86, 87]. The query schema of the form query language proposed in [74] is very similar to the web schema in the WHOWEDA's web model except additional assignments clauses used for specifying form input values or relations involved.

The system architecture of FAST is shown in Figure 2.8. The FAST consists of the following components: *Form Extraction UI*, *Form Query UI*, *Communication Manager*, *Web Page Loader*, *Form Retrieval Module*, *Form Extractor*, and *Form Query Processor*. The form extraction UI module allows users to specify URLs of the web pages each containing one or more forms. It calls the web page loader through communication manager to fetch a web page of some specified URL from the web. The forms embedded in the downloaded web page are formatted so that the user can select one or more forms to be stored in the form database for future queries. Each selected form is also assigned a unique name for identification purpose. The form query UI allows one to specify a form to be queried by its name. It calls the form retrieval module through communication manager to obtain the named form. If the form is not founded in the form database, an error message would be provided by the UI. Otherwise, the form object retrieved from the Form database is presented to the user. The user can then provide input values to the form, and submit the form query. The form query UI module analyzes the input values, constructs a form query file, and passes the query file to the form query processor through the communication manager. Once the form query processor completes the query evaluation, the list of hyperlinks to the original and locally cached result web pages will be displayed in the form query UI. The form query processor returns query results in the form of either a web table, or a web table and a relational table with mapping between their tuples. Once the form query processor completes a form query issued by the form query UI, the communication

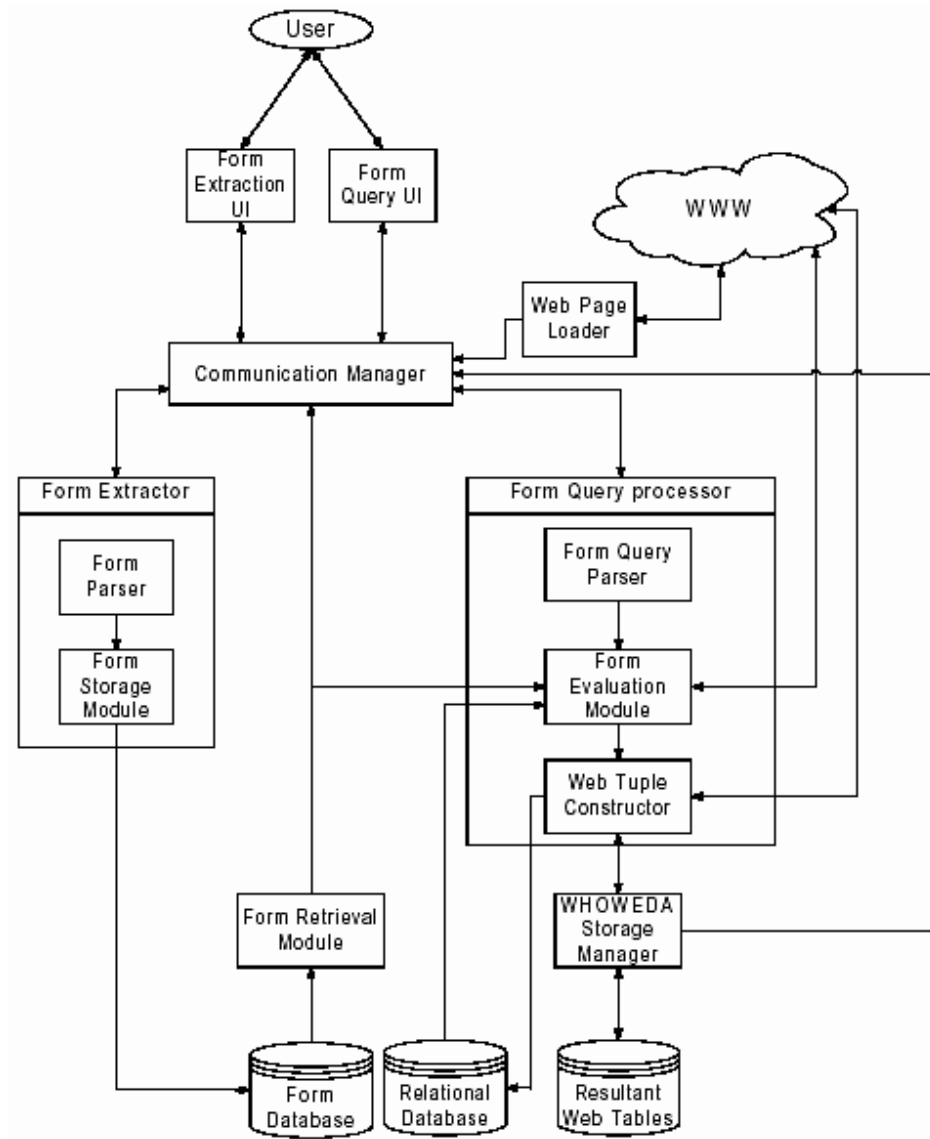


Figure 2.8: FAST architecture.

Table 2.1: Comparison of form query systems discussed in Section 2.2.

Form query system	Form model	Form storage	Query language & expressiveness	Multiple sets of input values	Consecutive forms	Navigation of result pages	Query results
W3QS	-	Yes	SQL-like	No	No	No	Web pages
ARANEUS	Simple	No	Complex	No	No	No	Relations
Transaction F-logic (section 2.2.3)	Simple	Yes	Complex	No	Yes	Yes	Relations
HiWE	Expressive	Yes	-	Yes	No	Yes	Web pages
FAST	Expressive	Yes	Complex	Yes	No	Yes	Web pages
DEQUE (Chapter 5)	Expressive	Yes	SQL-like	Yes	Yes	Yes	Result matches

manager will contact the WHOWEDA storage manager to obtain the list of hyperlinks to the original and locally cached result web pages and pass the hyperlinks to the form query UI.

In our study, we continue to develop the web form model. The concepts of form label, form domain, submission data set and super form described in Chapter 5 are added to improve the form representation. Our approach differs in the form query results’ extraction. While work [74] only stores all result pages, we introduce a result page representation and build a form query language such a way that its syntax allows one to extract useful data from returned web pages. Hence, the storage of results also differs from approach described in [74]. In addition, we simplify the form query language. Lastly, we address the problem of querying consecutive forms and study the dependency between forms.

2.2.6 Summary: Querying Search Interfaces

In Sections 2.2.1-2.2.5 we briefly described previous studies devoted to querying and automatically filling out web forms. Table 2.1 shows the differences between our query system for the deep Web (called DEQUE) described in Chapter 5 of this thesis and the systems reviewed in previous sections.

The comparison is based on the following system features: storage web forms in a database, the complexity of query language, assignment of multiple sets of input values, querying consecutive forms, result navigation, representation of query results, and support of form data modeling. Web forms are modeled by all systems except W3QS. An expressive form model allows a system to simulate most forms available on the Web while a simple model properly describes only primitive search interfaces. All systems except ARANEUS store forms in a database. The fourth column of Table 2.1 describes the complexity of the language used by a system. The “SQL-like” languages are considered to be more simple and usable ones. The fifth

column specifies the system capability in assigning multiple values to form controls. The FAST and our prototype system are able to fill out a form using values from results of queries to relational databases. The comparison table also shows if a query system is able to query consecutive forms and to retrieve all query results (i.e., navigate through multiple pages with results). The last column of Table 2.1 points out how query results are represented by each system.

To summarize, in this thesis we aim at presenting an advanced data model for representing search interfaces, proposing a user-friendly and expressive query language, and efficient extracting useful data from result pages.

2.3 Extraction from Data-intensive Web Sites

Not surprisingly, the explosive growth of the Web has made information search and extraction a harder problem than ever. Data extraction from HTML is usually performed by software modules called wrappers. Early approaches to wrapping Web sites were based on manual techniques [56, 93, 80]. We give a brief overview of these works in the next subsection.

2.3.1 Extracting Semi-structured Information from the Web

In [56], Hammer et al. have described a configurable extraction program for converting a set of hyperlinked HTML pages (either static or the results of queries) into database objects. Their program takes as input a specification that declaratively states where the data of interest is located on the HTML pages, and how the data should be “packaged” into objects. The descriptor is based on text patterns that identify the beginning and end of relevant data; it does not use “artificial intelligence” to understand the contents. This means that the proposed extractor is efficient and can be used to analyze large volumes of information. However, it also means that if a source changes the format of its exported HTML pages, the specification for the site must be updated. Since the specification is a simple text file, it can be modified directly using any editor. The authors described their approach to extracting semi-structured data from the Web using several examples. Specifically, they illustrated in detail how the extractor can be configured and how a TSIMMIS [36] wrapper is used to support queries against the extracted information.

In research [80], the authors addressed the manipulation of textual data, and proposed a language, called EDITOR, for searching and extracting regions in a document. EDITOR programs are based on two simple ideas, borrowed from text editors. In fact, editors provide a natural way of interacting with a document: when the user needs to restructure a docu-

ment, search primitives can be used to localize regions of interest inside text, and *cut&paste* operations to move regions around. Likewise, in EDITOR, “search” instructions are used to select regions in a document, and “cut & paste” to restructure them.

The work in [94] presents the World-Wide Web Wrapper Factory (W4F), a Java toolkit to generate wrappers for Web data sources. Some key features of W4F are an expressive language to extract information from HTML pages in a structured way, a mapping to export it as XML documents and some visual tools to assist the user during wrapper creation. Moreover, the entire description of wrappers is fully declarative. Sahuguet and Azavant [93] also demonstrated how to use W4F to create XML gateways, that serve transparently and on-the-fly HTML pages as XML documents with their DTDs.

The key problem with manually coded wrappers is that their creation and further maintenance are usually difficult and labor-intensive tasks.

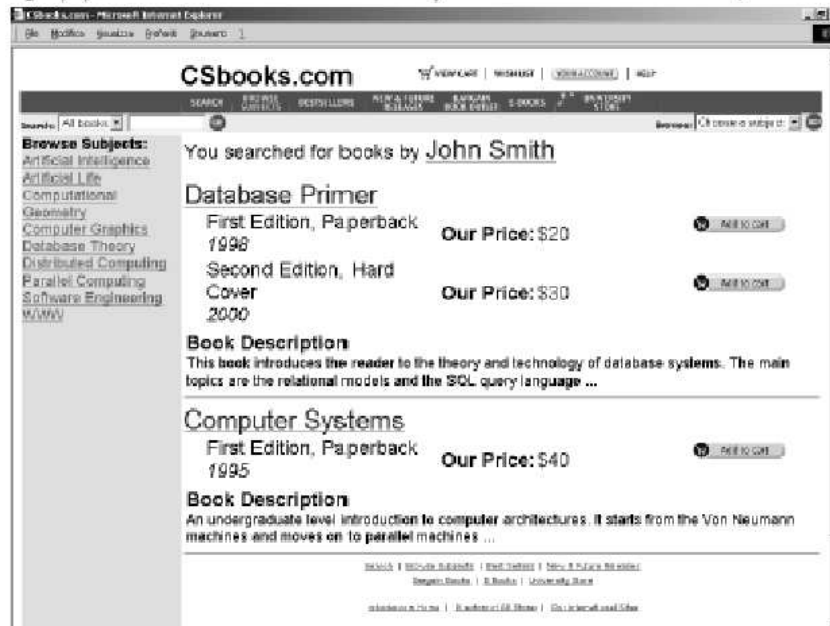
2.3.2 Automatic Data Extraction from Large Web Sites

The work in [38] describes a novel approach to the data extraction problem: the goal is that of fully automating the wrapper generation process, in such a way that it does not rely on any a priori knowledge about the target pages and their contents.

Pages in data-intensive sites are usually automatically generated: data are stored in a back-end DBMS, and HTML pages are produced using scripts (i.e., programs) from the content of the database. To give a simple but fairly faithful abstraction of the semantics of such scripts, the page-generation process is considered as the result of two separated activities: *(i)* first, the execution of a number of queries on the underlying database to generate a source dataset, i.e. a set of tuples of a possibly nested type that will be published in the site pages; *(ii)* second, the serialization of the source dataset into HTML code to produce the actual pages, possibly introducing URLs links, and other material like banners or images. A collection of pages that are generated by the same script called a *class of pages*.

Therefore, the problem studied in the work may be formulated as follows: *Given a set of sample HTML pages belonging to the same class, find the nested type of the source dataset and extract the source dataset from which the pages have been generated.* These ideas are clarified in Figures 2.9 and 2.10, which refer to a fictional bookstore site. In this example, pages listing all books by one author are generated by a script; the script first queries the database to produce a nested dataset in which each tuple contains data about one author, her/his list of books, and for each book the list of editions; then, the script serializes the resulting tuples into HTML pages (Figure 2.9). When run on these pages, the system (called **RoadRunner**) will compare the

http://www.csbooks.com/author?John+Smith



http://www.csbooks.com/author?Paul+Jones



Figure 2.9: Input HTML pages.

roadrunner output - microsoft internet explorer

file edit view history preferences document

Total number of SCHEMAs found: 1

Schema Number 1: A (B ((C) ? D E) + F) + Total Time: 0" 180 ms

sample1.html

A					
John Smith	B				F
	Database Primer	C	D	E	This book introduces the reader to the theory and technology...[TRUNCATED]
		First Edition, Paperback	1968	\$20	
		Second Edition, Hard Cover	2000	\$30	
	Computer Systems	First Edition, Paperback	1995	\$40	An undergraduate level introduction to computer...[TRUNCATED]

sample2.html

A					
Paul Jones	B				F
	XML at Work	C	D	E	A comprehensive description of XML and all related standards ...[TRUNCATED]
		First Edition, Paperback	1999	\$30	
	HTML and Scripts	null	1993	\$30	A useful HTML handbook, with a good tutorial on the use of sc...[TRUNCATED]
		Second Edition, Hard Cover	1999	\$45	
	JavaScripts	null	2000	\$50	A must in every Webmaster's bookshelf ...

Figure 2.10: Data extraction output.

HTML codes of the two pages, infer a common structure and a wrapper, and use that to extract the source dataset. Figure 2.10 shows the actual output of the system after it is run on the two HTML pages in the example. The dataset extracted is produced in HTML format. As an alternative, it could be stored in a database.

As it can be seen from the Figure, the system infers a nested schema from the pages. Since the original database field names are generally not encoded in the pages and this schema is based purely on the content of the HTML code, it has anonymous fields (labelled by A, B, C, D, etc.), which must be named manually after the dataset has been extracted.

We are interested in the RoadRunner project [38] because of two reasons. First of all, we propose the concept *super form* similar to RoadRunner's *class of pages*. According to our web form modeling described in Chapter 5, a super form is a form that combines forms generated by the same web script. The second reason is that we consider the RoadRunner's approach for extracting the content from a response page as very efficient and usable. However, we limit our extraction to defining HTML code corresponding to the resulting tuples on the web page.

Among other research efforts solely devoted to information extraction from resulting pages, Caverlee et al. [31, 30] presented the Thor framework

for sampling, locating, and partitioning the query-related content regions (called Query-Answer Pagelets or QA-Pagelets) from the deep Web. The Thor is designed as a suite of algorithms to support the entire scope of the deep Web information platform. The Thor data extraction and preparation subsystem supports a robust approach to the sampling, locating, and partitioning of the QA-Pagelets in four phases: 1) the first phase probes web databases to sample a set of resulting pages that covers all possible structurally diverse resulting pages, such as pages with multiple records, single record, and no record; 2) the second phase uses a page clustering algorithm to segment resulting pages according to their control-flow dependent similarities, aiming at separating pages that contain query matches from pages that contain no matches; 3) in the third phase, pages from top-ranked clusters are examined at a subtree level to locate and rank the query-related regions of the pages; and 4) the final phase uses a set of partitioning algorithms to separate and locate itemized objects in each QA-Pagelet.

2.4 Deep Web Characterization

To best of our knowledge, there are two works devoted solely to the characterization of the entire deep Web. The first one is a highly cited study [25] where *overlap analysis* approach was used. The technique called *random sampling of IP addresses* was used in the survey conducted in 2004 [34]. Below we discuss both these methods as well as key findings obtained in the surveys.

2.4.1 Overlap analysis

The characterization of deep Web was firstly performed in March 2000 by Michael Bergman. Several well-known estimates of the deep Web for March 2000 have been reported in his work [25]. Particularly, Bergman estimated that there were 43,000-96,000 deep web sites in the deep Web at that time (ultimate but disputable estimate in his study is even 200,000 deep sites for the year 2001) and that public information on the deep Web is 400 to 550 times larger than the commonly defined World Wide Web.

Estimates for the total number of deep web sites were obtained using the “overlap” analysis technique, which was originally applied to the characterization of the indexable Web [26, 71]. Overlap analysis involves pairwise comparisons of the number of listings individually within two sources, n_a and n_b , and overlap, n_0 , between them. Assuming random listings for both n_a and n_b , the total size of the population, N , can be estimated (see Figure 2.11). The estimate of the fraction of the total population covered by n_a is $\frac{n_0}{n_b}$; when applied to the total size of n_a an estimate for the total population size can be derived by dividing this fraction into the total size of

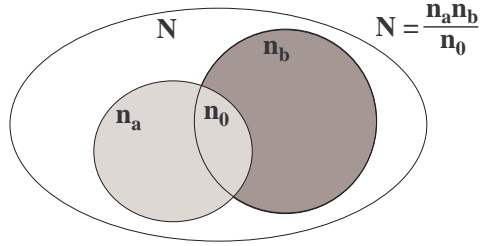


Figure 2.11: Total size estimation with overlap analysis.

n_a . These pairwise estimates are repeated for all of the individual sources used in the analysis. To illustrate, let us assume that, for example, the total population is 100. Then if two sources, A and B, each contain 50 items, we could expect on average that 25 of those items would be shared by the two sources and 25 items would not be listed by either. According to the formula above, the total population can be reconstructed as: $50/(25/50) = 100$.

Clearly, estimates obtained by overlap analysis are trustworthy only if certain conditions are met. First, it is important to have a relatively accurate estimate for total listing size for at least one of the two sources in the pairwise comparison. Second, both sources should obtain their listings randomly and independently from one another. However, as Bergman mentioned, the second condition was in fact violated since three listings of deep web sites (see [25] for their descriptions) used in the analysis were in no case random and independent.

2.4.2 Random Sampling of IP Addresses

The second survey [34] is based on the experiments performed in April 2004. In this work, the scale of the deep Web has been measured using the *random sampling of IP addresses* (*rsIP* for short) method, which was originally applied to the characterization of the entire Web [89]. The rsIP method estimates the size of the deep Web (specifically, number of deep web sites) by analyzing a sample of unique IP (Internet Protocol) addresses randomly generated from the entire space of valid IPs and extrapolating the findings to the Web at large. Since the entire IP space is of finite size and every web site is hosted on one or several web servers, each with an IP address², analyzing an IP sample of adequate size can provide reliable estimates for the characteristics of the Web in question. In [34], one million unique randomly-selected IP addresses were scanned for active web servers

²An IP address is not a unique identifier for a web server as a single server may use multiple IPs and, conversely, several servers can answer for the same IP.

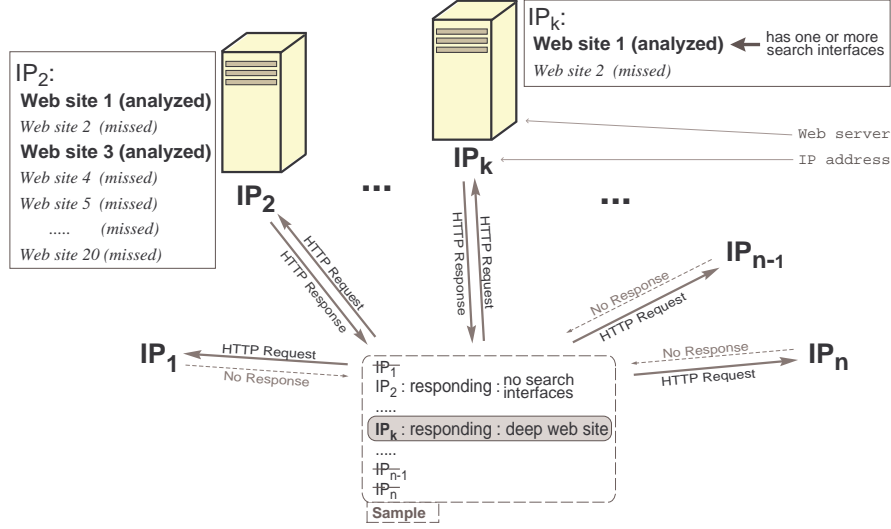


Figure 2.12: Random sampling of IP addresses method: sample of IPs (IP_1, \dots, IP_n) are tested for active web servers; detected web servers (with IP_2 and IP_k) are analyzed for the presence of interfaces to web databases; due to inability to find out all web sites hosted on a particular IP only three sites in total are analyzed while the rest is missed.

by making an HTTP connection to each IP. Detected web servers were exhaustively crawled and those hosting deep web sites (i.e., each of which has at least one search interface to a database) were identified. The technique is depicted in Figure 2.12, where one deep web site is found to be hosted on a web server with IP_k .

To reduce uncertainty when identifying deep web resources Chang et al. distinguished three related notions for accessing the deep Web - a *deep web site*, a *web database*, and a *web interface*. Among the findings obtained are the total number of deep web sites, web databases, and web interfaces, which were estimated as 307,000, 450,000 and 1,258,000 correspondingly.

Unfortunately the rsIP approach has several limitations. In next section we briefly discuss the two most important ones.

2.4.3 Virtual Hosting and DNS load balancing

The most serious disadvantage of the rsIP is ignoring *virtual hosting*, i.e., the fact that multiple web sites can share the same IP address. The problem here is that for a given IP address there is no reliable procedure to obtain a full list of web sites hosted on a web server with this IP. As a result, the rsIP method ignores a certain number of sites, some of which may apparently be deep web sites. To illustrate, Figure 2.12 shows that servers with IP_2 and

Table 2.2: Virtual hosting: the average number of hosts per IP address reported in several web surveys.

Short description of analyzed hostnames and reference	When conducted	Num.of hosts	Num.of IPs	Aver.num. of hosts per IP
<i>Russian Web</i> : all hosts with second-level domain names in .RU and .SU zones [107, 108]	03/2007	639,174	68,188	9.37
	03/2006	387,470	51,591	7.51
<i>Entire Web</i> : all hosts known to Netcraft [14]	04/2004	49,750,568	$\approx 4,400,000$	\approx 11.3
<i>Portuguese Web</i> : 85% of hosts in .PT domain, 12% in .COM, etc. [49]	04/2003	46,457	6,856	6.78

IP_k host twenty and two web sites correspondingly, but only three out of 22 web sites are actually crawled to discover interfaces to web databases . The numbers of analyzed and missed sites per IP in this example are not atypical: reverse IP procedure (described in Appendix C.2) usually returns from one to three web sites that are hosted on a particular IP address while hosting a lot of sites on the same IP is a common practice. Table 2.2 presents the average numbers of virtual hosts per IP address obtained in four web studies conducted in 2003-2007. The data clearly suggests that: (1) in average, one IP address is shared by 7-11 hosts; and (2) the number of hosts per IP increases over time [14, 49, 108]. Hence, the virtual hosting cannot be ignored in any IP-based sampling survey.

The next important factor overlooked by the rsIP method is *DNS load balancing*, i.e., the assignment of multiple IP addresses to a single web site [66]. For instance, Russian news site `newsru.com`, that is mapped to three IPs, is three times more likely to appear in a sample of random IPs than a site with one assigned IP. Since the DNS load balancing is the most beneficial for popular and highly trafficked web sites one can expect that the rsIP method bias caused by the load balancing is less than the bias due to the virtual hosting. Indeed, according to the SecuritySpace’s survey as of April 2004, only 4.7% of hosts had their names resolved to multiple IP addresses [15] while more than 90% of hosts shared the same IP with others (see the third row of Table 2.2). Nevertheless, the DNS load balancing is a substantial factor and needs to be taken into account.

2.4.4 National Web Domains

Several studies on the characterization of the indexable Web space of various national domains have been published (e.g., [20, 49, 95, 106]). The review work [19] surveys several reports on national Web domains, discusses a methodology to present these kinds of reports, and presents a side-by-side comparison of their results. At the same time, national domains of the deep Web have not been studied so far and, hence, their characteristics can be only hypothesized.

There are solid grounds for supposing that the estimates obtained in two aforementioned surveys of the entire deep Web [25, 34] are actually lower-bound because web databases in, at least, several national segments of the Web were for the most part ignored. For instance, the semi-automatic process of identifying a web interface to a web database in [34] consists of automatic extraction of web forms from all considered web pages, automatic removal of forms, which are definitely non-searchable, and finally manual inspection of the rest, potentially searchable forms, by human experts. Due to the limited number of experts (presumably just the authors of [34] were those experts) and, hence, the limited number of languages they were able to work with one can expect that a certain number of web interfaces in unknown (to experts) languages has not been taken into account. Notwithstanding that the approach used in [25] did not require multilingual skills from people involved in the study, we still argue that some number of non-English deep web sites has not been counted in this report as well. Indeed, the results produced by the overlap analysis technique depend significantly on “quality” of sources used in pairwise comparisons. The deep-Web directories considered by Bergman are mainly for English-speaking web users and, thus, omitted a number of national deep web sites. This makes the overlap analysis imperfect under the circumstances and suggests that the Bergman’s estimate for the total number of deep web sites is a lower bound. In this way, we consider our survey as an attempt to supplement and refine the results presented in [25, 34] by studying online databases in one particular national segment of the Web.

In this thesis, we adopted the rsIP method to our needs. Unlike [34] we noted several essential drawbacks (two of them were discussed in Section 2.4.3) of the rsIP method leading to underestimating of parameters of interest and suggested a way to correct the estimates produced by the rsIP. Additionally, we proposed two new techniques for the deep Web characterization, the *stratified random sampling of hosts* (called *srsh* further on) and *stratified cluster sampling of IP addresses* (*scsIP*) methods.

2.5 Finding deep web resources

Sections 2.2 and 2.3 discussed how to query web databases assuming that search interfaces to web databases of interest were already discovered. Surprisingly, finding of search interfaces to web databases is a challenging problem in itself. Indeed, since several hundred thousands of databases are available on the Web [34] one cannot be aware that he/she knows the most relevant databases even in a very specific domain. Realizing that, people have started to manually create web database collections such as the DBcat [41], a catalog of biological databases, or the Molecular Biology Database Collection [47]. However, manual approaches are not practical as there are hundreds of thousands databases. Besides, since new databases are constantly being added, the freshness of a manually maintained collection is highly compromised.

There are two classes of approaches to identify search interfaces to on-line databases: pre-query and post-query approaches. Pre-query approaches identify searchable forms on web sites by analyzing the features of web forms. Post-query approaches identify searchable forms by submitting the probing queries to the forms and analyzing the resulting pages.

Bergholz and Chidlovskii [24] gave an example of the post-query approach for the automated discovery of search interfaces. They implemented a domain-specific crawler that starts on indexable pages and detects forms relevant to a given domain. Next, the Query Prober submits some domain-specific phrases (called “positive” queries) and some nonsense words (“negative” queries) to detected forms and then assesses whether a form is searchable or not by comparing the resulting pages for the positive and negative queries.

Cope et al. [37] proposed a pre-query approach that uses automatically generated features to describe candidate forms and uses the decision tree learning algorithm to classify them based on the generated set of features. Barbosa and Freire [21, 23] proposed a new crawling strategy to automatically locate web databases. They built a form-focused crawler that uses three classifiers: page classifier (classifies pages as belonging to topics in taxonomy), link classifier (identifies links that are likely to lead to the pages with search interfaces in one or more steps), and form classifier. The form classifier is a domain-independent binary classifier that uses a decision tree to determine whether a web form is searchable or non-searchable (e.g., forms for login, registration, leaving comments, discussion group interfaces, mailing list subscriptions, purchase forms, etc.).

Chapter 3

Deep Web Characterization

This chapter surveys databases on one specific national segment of the Web. The survey is based on our experiments for the scale of the national deep Web conducted in summer 2005 and in September 2006.

To our knowledge, this survey is the first attempt to consider the specific national segment of the deep Web. The known deep Web characterization efforts (see Section 2.4) have predominantly concentrated on study of English deep web sites and, therefore, the estimates of the deep Web obtained in these works may be biased, especially owing to a steady increase in non-English web content. The national deep Web was studied on the example of the Russian segment of the Web (called *Runet* hereafter in this paper). There were several reasons to choose exactly the Russian part of the deep Web. Firstly, since Russian is written using the Cyrillic alphabet, which is non-Latin, one can expect that Runet is considerably more separated from the entire Web than, say, the German segment (where Latin alphabet is used). Secondly, we had an access to the data set provided by Yandex, a Russian web search engine (see Section 3.2 and Appendix C.3). And last but not least, having Russian as a mother tongue language was essential due to many web sites in Runet need to be manually inspected.

In June 2005 and in August 2005 and then in September 2006 we performed a series of experiments to estimate the number of deep web sites in Runet. We used three techniques: the adopted rsIP, the srsh and scsIP methods (see Section 2.4.4). The experiments themselves and the results for each method are described in the following sections.

3.1 Random Sampling of IP Addresses

We extracted all ranges of IP addresses used by Russian networks from the IP-Country database [6]. There were totally around 10.5 millions of IPs at the time of June 2005, $N = 10.5 \times 10^6$. Then, $n = 10.5 \times 10^4$ unique IP

addresses (1% of the total number) were randomly selected and scanned for active web servers (tools we used for that are mentioned in Appendix C.1). We detected 1,379 machines with web servers running on port 80. For each of these machines we resolved their corresponding hostnames: the first hostname was always an IP address itself in a string format; other hostnames were non-empty values returned by *gethostbyaddr* function. Next step was crawling each host to depth three¹. To not violate the sampling procedure we had to crawl only pages obtained from either the same IP or IPs that do not belong to Russian networks (Runet IPs, as having equal probability of selection in the sample, had to be ignored). The automatic analysis of retrieved pages performed by our script in Perl was started after that. All pages which do not contain web forms and pages which do contain forms, but those forms that are not interfaces to databases (i.e., forms for site search, navigation, login, registration, subscription, polling, posting, etc.) were excluded. In order to consider just unique search forms pages with duplicated forms were removed as well. Finally, we manually inspected the rest of pages and identified totally $x = 33$ deep web sites. It should be noted that unlike [34] we counted only the number of deep web sites. The number of web databases accessible via found deep web sites as well as the number of interfaces to each particular database were not counted since we did not have a consistent and reliable procedure to detect how many web databases are accessible via particular site. The typical case here (not faced in this sample though) is to define how many databases are accessed via a site with two searchable forms – one form for searching new cars while another for searching used ones. Both variants, namely two databases for used and new cars exist in this case or it is just one combined database, are admissible. Nevertheless, according to our non-formal database detection, 5 of 33 deep web sites found had interfaces to two databases, which gives us 38 web databases in the sample in total.

The estimate for the total number of deep web sites is:

$$\widehat{D_{rsIP}} = \frac{33 \times 10.5 \times 10^6}{10.5 \times 10^4} = 3300. \quad (3.1)$$

An approximate 95% confidence interval² for $\widehat{D_{rsIP}}$ is given by the following formula:

$$\widehat{D_{rsIP}} \pm 1.96 \sqrt{\frac{N(N-n)(1-p)p}{n-1}}, \quad (3.2)$$

where $p = \frac{x}{n}$ (see Chapter 5 in [105]). Thus, the total number of deep web sites estimated by the rsIP method is 3300 ± 1120 .

¹See [34] for discussion on crawling depth value. Figure 1.2 illustratively shows depth values for one of the depicted websites.

²This interval contains the true value of the estimated parameter with 95 percent certainty.

To our knowledge, there are four factors which were not taken into account in the rsIP experiment and, thus, we can expect that the obtained estimate $\widehat{D_{rsIP}}$ is biased from the true value. Among four sources of bias the most significant one is the virtual hosting. The analysis of all second-level domains in the .RU zone conducted in March 2006 [107] has shown that there are, in average, 7.5 web sites on one IP address (similar numbers were reported in other web surveys, see Table 2.2). Unfortunately, even with the help of advanced tools for reverse IP lookup (see Appendix C.2) there is not much guarantee that all hostnames related to a particular IP address would be resolved correctly. This means that during the experiment we certainly overlooked a number of sites, some of which are apparently deep web sites.

Next essential factor is *DNS load balancing*, i.e., the assignment of multiple IP addresses to a single web site [66]. A web site mapped to several IPs is more likely to be selected for the sample than a site with one IP address. Therefore, our rsIP estimate should be expected to be greater than the true value. Similar to the virtual hosting factor, there is no guarantee in detecting all web sites with multiple IPs. We checked all 33 identified deep web sites for multiple IPs by resolving their IP addresses to corresponding hostnames and then resolving those hostnames back to their corresponding IPs (the same technique as in [89]). Though no multiple IP addresses for any of these sites were detected by this procedure, we are not confident whether every deep web site in the sample is accessible only via one IP address. In any case, sites on multiple IPs are less common than sites sharing the same IP address and, hence, we believe that the virtual hosting's impact on the estimate should exceed one-site-on-multiple-IPs factor influence.

Third unconsidered factor is the exclusion of web servers running on ports other than 80 (default port for web servers). In our experiment we did not detect web servers that are not on port 80 and, obviously, missed a number of servers that may host deep web sites. However, the number of deep resources on non-default port numbers seems to be negligible since using non-default port numbers for web servers is not a widespread practice.

While previous three factors are about how well we are able to detect deep web sites in the sample the IP geodistribution factor concerns how well the whole IP pool covers the object of study, Runet in our case. Recall that our pool of IP addresses is all IPs assigned to the Russian Federation. Since web hosting is not restricted to geographical borders one can expect that a number of Runet web sites are hosted outside Russia. Analysis of all second-level domains in the .RU zone [107] revealed that, indeed, this is the case and approximately 10.5% of all studied domains are hosted on IPs outside the Russian Federation. Although only second-level RU-domains were investigated in [107] we suppose that the found distribution (89.5% of web servers are in Russia and the rest is outside) is applicable to all domains

related to Runet³. This allows us to make a correction to our rsIP estimate. Under the fact that our sample was selected from the population which covers just around 90% of Runet, we updated the estimate and, finally, got that there are approximately **3650±1250** (rounded to the nearest 50) **deep web sites in Runet**.

3.2 Stratified Random Sampling of Hosts

In this experiment we used the data set “Hostgraph” (its description is given in Appendix C.3) provided by Yandex, a Russian search engine. All hosts indexed by Yandex were extracted from the Hostgraph. Besides, “host citation index”, i.e. the number of incoming links for each host, was calculated. To improve method’s accuracy, the shortening procedure was applied to the list of extracted hosts:

- Web sites which are certainly not deep web sites were removed from the list. In particular, we removed all sites on free web hostings known to us.
- We grouped all host by their second and third-level domain names (for example, all hosts of the form **.something.ru* are in the same group). The largest groups of hosts were checked and groups with sites leading to the same web databases were removed. As an example, we eliminated all hosts of the form **.mp3gate.ru* (except the host *www.mp3gate.ru*) since the same web database is available via any of these hosts.

We stopped the procedure at the total list of $N = 299,241$ hosts. At the beginning, we decided to check our assumption that the proportion of deep web sites among highly cited sites is higher than among less cited sites. If so, applying stratified random sampling technique to our data would be more preferable than using simple random sampling.

To examine the assumption we divided the list of hosts into three strata according to the number of incoming links for each host. The first stratum contained the most cited $N_1 = 49,900$ hosts, less cited $N_2 = 52,100$ hosts were in the second stratum, and the rest, $N_3 = 197,240$ hosts, was assigned to the third stratum (the strata sizes are rounded to the nearest ten). Then, $n_1^* = 50$, $n_2^* = 50$, and $n_3^* = 100$ unique hosts were randomly selected from each stratum correspondingly. Similar to the rsIP method, each of the selected hosts were crawled to a depth three. While crawling we checked if

³The geodistribution of third and higher-level domains in the .RU zone should be almost the same. The distribution of second and higher-level domains in other than .RU zone (i.e., those ending with .com, .net, and so on) may differ but their fraction in all Runet domains is not so significant, around 25% according to Appendix C.3.

Table 3.1: Results of the preliminary stratified random sampling

Stratum(k)	N_k	n_k^*	d_k^*
1	49,900	50	7
2	52,100	50	2
3	197,240	100	1

Table 3.2: Results of the srsh experiment

Stratum(k)	N_k	n_k	d_k	\widehat{D}_k	$D_{k,cor}$	Duplication
1	49,900	294	35	5940	5600	10 of 35 resources have one or more duplicates: 4 duplicates in <i>stratum</i> ₁ , 1 in <i>stratum</i> ₂ , and 7 in <i>stratum</i> ₃
2	52,100	174	8	2395	2090	2 of 8 resources have one duplicate: 1 duplicate in <i>stratum</i> ₂ and 1 in <i>stratum</i> ₃
3	197,240	400	3	1480	100	0 of 3 resources have duplicates
Total	299,240	868	46	9815	7790	12 of 46 resources have duplicates

links point to pages located on the same host or on other hosts not mentioned in the total list of hosts. To meet the conditions of the sampling procedure all pages on hosts, which are in the list, were ignored. After that, the procedure becomes identical to the rsIP one (see Section 3.1). The results, number of deep web sites d_k^* detected in each stratum, are shown in Table 3.1.

It is easy to see from Table 3.1 that our assumption is correct and, indeed, the probability of having a web interface to a web database is higher for highly cited hosts than for less cited ones. Thus, for reliable estimation of the total number of deep web sites in Runet we decided to use the stratified random sampling approach with the same division into strata.

We selected $n_1 = 294$ (including n_1^* hosts already studied in the preliminary sampling), $n_2 = 174$ (including n_2^* hosts), and $n_3 = 400$ (including n_3^* hosts) unique hosts for each of the three strata correspondingly. The

process of analyzing totally 668⁴ sampled hosts and identifying deep web sites was the same as in the preliminary sampling of hosts. Our findings are summarized in Table 3.2, where d_k and $\widehat{D}_k = N_k \frac{d_k}{n_k}$ are the number of deep web sites in the sample from stratum k and the estimated total number of deep web sites in stratum k correspondingly.

The estimate for the total number of deep web sites is:

$$\widehat{D_{srsh}} = \sum_{k=1}^3 \widehat{D}_k = 9815. \quad (3.3)$$

An approximate 95% confidence interval for $\widehat{D_{srsh}}$ is given by the following formula:

$$\widehat{D_{srsh}} \pm 1.96 \sqrt{\sum_{k=1}^3 \frac{N_k(N_k - n_k)(1 - p_k)p_k}{n_k - 1}}, \quad (3.4)$$

where $p_k = \frac{d_k}{n_k}$ (see Chapter 11 in [105]). In this way, the total number of deep web sites estimated by the srsh method is 9815 ± 2970 . This estimate is not a final one, however. There are two factors which have to be considered in order to correct $\widehat{D_{srsh}}$.

The most significant source of bias in the srsh experiment is the host duplication problem. It is rather common that a web site can be accessed using more than one hostname. Naturally, several hostnames per one deep web site are typical as well. By using tools described in Appendix C.2 (resolving a hostname of interest to an IP address and then reverse resolving IP to a set of hostnames) and inspecting our list of hosts manually we were able to identify that 12 of 46 deep web sites are accessible via more than one hostname. The distribution of duplicates among different strata is given in Table 3.2 in the “*Duplication*” column. The correction to be done is pretty straightforward: the existence of one duplicate for a particular host means that this host is twice more likely to be in the sample than a host without any duplicate. Thus, the corrected estimate for the first stratum is $D_{1,cor} = \frac{N_1}{n_1} \times (31 + \frac{4}{2}) = 5600$. $D_{1,cor} \times \frac{1}{35}$ and $D_{1,cor} \times \frac{7}{35}$ deep resources should be excluded from the estimates for second and third stratum correspondingly since they were already counted in the estimate for the first stratum. Similarly, we obtained the corrected estimates for the second and third stratum: $D_{2,cor} = \frac{N_2}{n_2} \times (7 + \frac{1}{2}) - D_{1,cor} \times \frac{1}{35} = 2090$ and $D_{3,cor} = \widehat{D}_3 - D_{1,cor} \times \frac{7}{35} - D_{2,cor} \times \frac{1}{8} = 100$ deep web sites respectively.

The second factor, similar to the geodistribution factor in the rsrIP experiment, is how well the list of hosts we studied covers Runet. The list we

⁴200 of 868 hosts were already studied.

worked with contained all hosts indexed by Yandex, a Runet search engine, at the time of February 2005. Recent study [96] has shown that Yandex has one of the best coverage of Runet among the largest web crawlers indexing Russian part of the Web. In this way, one can expect that our list represented Runet with sufficient accuracy. More importantly, we believe that the only way for a web database content to be available for web users is having at least one web interface located on a page indexed by a search engine. Otherwise, not only data in this database is hidden but also its whole existence is completely unknown to anyone. Therefore, according to our point of view, the population of hosts used in the srsh experiment is essentially complete for purposes of detecting deep web sites. It should be noted however that since the “Hostgraph” data was created in February 2005 and our experiments were performed in June and August 2005 those deep web sites which appeared mainly in spring 2005 were not counted.

To sum up, due to the fact that at least one of four deep web sites is accessible via more than one hostname we corrected $\widehat{D_{srsh}}$ and obtained that the **total number of deep web sites in Runet** is around **7800±2350** (rounded to the nearest 50).

3.2.1 Subject Distribution of Web Databases

We manually categorized 79 deep web sites sampled in the rsIP and srsh experiments into ten subject categories: Libraries (*lib*), Online Stores (*shop*), Auto (*auto*), Business (*biz*), Address Search (*addr*), Law&Government (*law*), People Search (*pe*), Travel (*tra*), Health (*he*), and Science (*sci*). Note that more than one category may be assigned to a deep web site - for instance, nearly half of deep web sites assigned to the “Auto” category were also placed into “Online Stores” since these sites were auto parts&accessories online stores. Figure 3.1 shows the distribution of deep web databases over the categories. The particular observation we made is that almost 90% of deep resources in the category “Online Stores” (13 of 15 sites) have a navigational access to their data, i.e. these sites have not only one or several web search forms but also have a browse interface which allows a user to reach the necessary data from a web database via a series of links. In this way, such resources cannot be considered as entirely “hidden” from search engines since their content may be accessed by following hyperlinks only.

3.3 Discussion: results of the 2005 survey

The experiments provided us with two estimates for the total number of deep web sites in Runet: 3650±1250 as estimated by the rsIP method, and 7800±2350 as estimated by the srsh method.

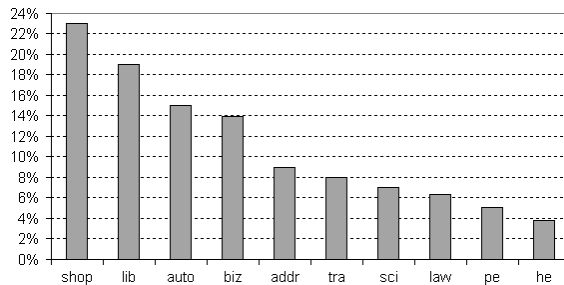


Figure 3.1: Distribution of deep web sites over subject category

In fact, there is no contradiction between the estimates since the rsIP method should give us a lower-bound estimate due to the virtual hosting factor (see Section 3.1) while the srsh method should result in an upper-bound estimate because of the host duplication problem (see Section 3.2). In any case, it is unquestionable that a 95% confidence interval for the total number of deep web sites in Runet is (2400,10150), that is, the **scale of the Russian deep Web is on the order of 10^3 resources**. We also believe that the estimate obtained by the srsh method is closer to the true number than the estimate by the rsIP method because the host duplication factor was at least partially addressed in the srsh experiment (duplicates were identified and then the corresponding correction was done) while the influence of virtual hosting on the rsIP estimate was just mentioned as very important but not measured quantitatively.

The quick and indirect attempt to correct the rsIP estimate is to reconstruct the process of detecting deep web sites for specifically designed list of IPs. In order to build such a list, we took 46 deep web sites detected by the srsh method and resolved their hostnames to the list of IP addresses. “Ideal” rsIP method should detect 46 deep web resources in this list, non-ideal rsIP method detects less due to shortcomings of IP-to-host resolving procedure. Our rsIP method was able to detect 25 (54%) deep resources, and the rest, 21 (46%) of deep web sites, were not detected⁵. Thus, we can expect that around 46% deep resources were missed in our rsIP experiment and, hence, the corrected rsIP estimate is 6800 ± 2300 . The intervals for the rsIP and srsh estimates, (4500,9100) and (5450,10150) correspondingly, are well overlapping, and their intersection, namely (5450,9100) or approximately (rounded to the nearest 100) **7300 ± 1800** , is our final estimate for the **total number of deep web sites in Runet in summer 2005**.

⁵In more than half cases no hostnames were resolved from an IP address and then the only URL to use `http://IP_address` returned just an error page or a web server default page.

It is interesting to compare the number of deep resources in Runet and in the entire Web. The number of deep web sites resources in the entire Web estimated by Chang et al. [34] is 307,000⁶ for April 2004 while our estimate for Runet obtained by the same method as in [34] is 3650 ± 1250 for summer 2005. The comparison suggests that in terms of the number of deep web sites the Russian deep Web is approximately the hundredth part of the entire deep Web. This roughly coincides with the portion of Russian web sites in the Web – survey [88] indicated that one percent of public sites in the Web in 1999 as well as in 2002 were in Russian.

3.4 Stratified cluster sampling of IP addresses

Real-world web sites may be hosted on several web servers, typically share their web servers with other sites, and are often accessible via multiple hostnames. All this makes an IP-based or hostname-based sampling approaches somewhat restricted and prone to produce biased estimates.

The process of solving domain aliasing problem in the srsh method offers a clue to a better sampling strategy. Hostname aliases for a given web site are frequently mapped to the same IP address. This is not really surprising since it is practical for a web administrator, who controls via which hostnames a particular web site should be accessible, to define hostname aliases by configuring only one web server (it means such hostnames are resolved to server’s IP). Thus, given a hostname resolved to some IP address, one can identify other hostnames pointing to the same web content by checking those hostnames that are also mapped to this IP. It is interesting to see here a strong resemblance to the virtual hosting problem when there is a need to find all hosts sharing a given IP address. At the same time, discovering hosts mapped to a particular IP in the srsh method is easier than the same task in the rsIP method since more information is available, namely, the overall list of hostnames can be utilized. Indeed, resolving all hostnames from the total list to their corresponding IP addresses provides us with the knowledge about “neighbors-by-IP” for any hostname to be sampled. Technically, by doing such massive resolving we cluster all known hosts into groups, each including hosts sharing the same IP address.

Once the overall hostname list is clustered by IPs we can apply the cluster sampling strategy, where an IP address is a primary sampling unit consisting of a cluster of secondary units, hostnames. The grouping hosts based on their IPs is, in fact, quite natural because this is exactly what

⁶No confidence intervals were mentioned in [34] but it is easy to calculate them from their data: particularly, a 95% confidence interval for the number of deep web sites is $307,000 \pm 54,000$. A 99% confidence intervals have also been specified in more recent work of the same authors [57].

happens on the Web, where every web server serves requests only to certain hosts (that are mapped to server’s IP).

To summarize, the suggested technique (called the *scsIP* method) for characterization of national deep web includes the following steps: (1) resolving a large list of hostnames relating to a studied national web domain to their IP addresses and grouping hosts based on their IPs; (2) generating a sample of IP addresses randomly selected from a list of all IPs resolved at step 1; (3) retrieving a set of hosts for each sampled IP and analyzing such hosts for the presence of search interfaces in a way similar to the *rsIP* and *srsh* methods. Among the advantages of the *scsIP* is taking into account all three sources of bias described in the previous section. There are not only pluses however: the most important limitations of the approach are high requirements to the original list of hostnames, which has to cover a studied segment of the Web with a good accuracy, and more complex dataset preparation. The latter, the preparation of the dataset used in our experiments, is described in the next section.

3.4.1 Dataset preparation and stratification

For our experiments conducted in September 2006 we extracted two lists of hostnames from datasets “Hostgraph” and “RU-hosts” (see Appendix C.4) and combined them into one list of unique hostnames. It is expected that the list well covers the Russian part of the Web. Each hostname in the list was resolved to a corresponding IP address (or, in case of DNS load balancing discussed above, to multiple IPs) using the *dig* [4] tool giving us a host-IP pair (or several such pairs). Depending on the presence of a “*www.*” prefix at the beginning each unresolved⁷ hostname was modified by either adding or removing the “*www.*” and resolved in this form again. For still unresolved hosts we repeated the resolve procedure in one week. Since some hosts may be resolved to invalid IPs we checked the validity of all found IP addresses. Technically, all IPs (e.g., 10.0.0.1 or 224.0.0.1) from unassigned IP address ranges (known as bogon lists [43]) are invalid and, hence, should be ignored. After removing all host-IP pairs with invalid IPs we resulted in 717,240 host-IP pairs formed by 672,058 unique hosts and 79,679 unique IP addresses.

The numbers specifically shows that DNS load balancing has a modest influence: most hosts (94.6%) are resolved to single IP address and only 5.4% (36,349) of hosts are mapped to two or more IPs. At the same time, our compiled dataset gives us yet another support for the magnitude of virtual hosting (see also Table 2.2): there are nine virtual hosts in average per one IP address⁸. The degree of IP address sharing is depicted in Figure 3.2.

⁷A host is unresolved if the *dig* does not return an IP address.

⁸A host resolved to two or more IPs is counted twice or more for each corresponding

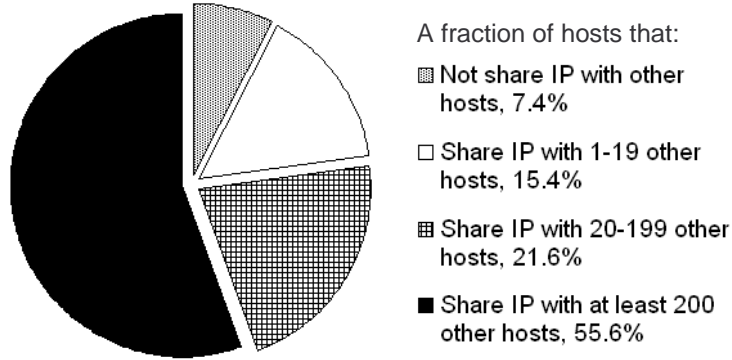


Figure 3.2: IP address sharing for our dataset.

Particularly, 55,6% (398,608) of all hosts in the dataset share their IPs with at least 200 other hosts.

Since the sampling procedure is simplified if each host is mapped to a single IP we restricted a host to have only one mapping to IP address and excluded all “redundant” host-IP pairs from the overall list. This left us with 672,058 hosts on 78,736 IP addresses.

We then clustered the overall list of hosts by their IPs and divided the list into three strata according to the number of hosts mapped to each IP. The idea behind such separation lies in the fact that IP addresses referred to a large number of hosts are good indicators of server hosting spam web sites [44] and, hence, deep web sites are unlikely to be found when analyzing such IPs. Another reason to stratify was to actually identify if deep web sites are tend to run on servers hosting only a few sites. Table 3.3 presents the specific criteria used for stratification as well as number of IP addresses and hosts in each stratum. For example, Stratum 1 includes such IP addresses, that are each associated with seven or less hostnames, while Stratum 3 combines IPs with no less than 41 hosts mapped to each.

3.4.2 Results of the 2006 survey

In our survey of Russian deep Web in September 2006 we used the dataset prepared as described above. For each stratum we randomly selected a set of IPs and analyzed hosts that are mapped to this IPs. While crawling we checked if links point to pages that belong to hosts on the same IP or hosts on IPs that are non-present in our dataset. To meet the conditions of the sampling procedure all pages on hosts with IPs that are present in the dataset and other than the analyzed one were ignored. Otherwise, the procedure of analyzing web content accessible via studied hosts is identical to

IP.

Table 3.3: Dataset stratification.

	Stratum 1	Stratum 2	Stratum 3	Total
Criteria (number of hosts per IP): x	$x \leq 7$	$7 < x \leq 40$	$x > 40$	
Number of IPs: $S_i, i = \{1, 2, 3\}$	71,486	5,390	1,860	78,736
Number of hosts: $H_i, i = \{1, 2, 3\}$	112,755	86,829	472,474	672,058

Table 3.4: Results of the scsIP experiment.

	Stratum 1	Stratum 2	Stratum 3	Total
Number of sampled IPs: s_i	964	100	11	1,075
Number of analyzed hosts: h_i	1,490	1,584	3,163	6,237
Number of deep web sites: d_i	62	38	46	146
Estimate for num.of deep web sites (rounded to the nearest hundred)	4,700	2,100	6,900	13,700
Number of databases: db_i	79	46	49	174
Estimate for num.of deep web sites (rounded to the nearest hundred)	6,000	2,500	7,300	15,800

the one we used in the rsIP and srsh methods. Table 3.4 shows the numbers of sampled IPs and analyzed hosts as well as the numbers of deep web sites and databases (they were counted unlike the rsIP and srsh methods) detected in each stratum.

The estimates for the total numbers of deep web sites and databases are: $D = \sum_{k=1}^3 \frac{d_k H_k}{h_k}$ and $DB = \sum_{k=1}^3 \frac{db_k H_k}{h_k}$ correspondingly (shown in Table 3.4). The formula for 95% confidence intervals for D and DB can be found in Chapter 12 of [105]. To summarize, the **total numbers of deep web sites and databases in Runet as of September 2006** estimated by the scsIP method are **13,700±3,900** and **15,800±4,200** correspondingly.

Though the scsIP method takes into account such significant sources of bias for the rsIP and srsh methods as virtual hosting and DNS load balancing the hostname duplication factor is still not fully considered. The problem is that some hostnames leading to the same web content may resolve to several IPs and, thus, such hosts and corresponding sites are more likely to be in a sample and be analyzed respectively. In this way, the obtained estimates for the the total number of deep web sites and databases are, in fact, upper-bound.

Chapter 4

Finding Deep Web Resources

The deep Web has been growing at a very fast pace. It is estimated that there are hundred thousands of deep web sites [34]. The deep web content residing in databases is available on demand, as web users issue their queries via search interfaces. Due to the huge volume of information in the deep Web, there has been a significant interest to approaches that allow users and computer applications to leverage this information. For example, in Chapter 5 we discuss how to query web databases and, particularly, we assume that search interfaces to web databases of interest are already discovered and known to our system. However, such assumption does not hold true mostly because of the large scale of the deep Web – indeed, for any given domain of interest there are too many web databases with relevant content. Thus, the ability to locate search interfaces to web databases becomes a key requirement for any application accessing the deep Web.

4.1 Introduction

Due to the dynamic nature of the Web, when new sources being added all the time and old sources modified or removed completely, it is important to automatically discover search interfaces that serve as the entry points to the deep Web. One of the challenges in such discovery is that search interfaces are very sparsely distributed over the Web, even within specific domains. For example, a topic-focused best-first crawler [33] was able to retrieve only 94 movie search forms while crawling 100,000 pages related to movies.

Finding of search interfaces to web databases is a challenging problem. Indeed, since several hundred thousands of databases are available on the Web [34] one cannot be aware that he/she knows most relevant databases even in a very specific domain. Even national (as it was shown in Chapter 3 of this thesis) or specific community-oriented (e.g., bioinformatics community) parts of the deep Web are too large to be fully discovered. Realizing the

need in better mechanisms for locating web databases, people have started to manually create collections of web databases such as the Molecular Biology Database Collection [47]. However, because of the scale of the deep Web, manual approaches are not practical. Besides, since new databases are constantly being added, the freshness of a manually maintained collection is highly compromised.

Surprisingly, existing directories of deep web resources (i.e., directories that classify web databases in some taxonomies) has extremely low coverage for online databases. In fact, completeplanet.com [3], the largest of such directories, with around 70,000 databases¹ covered only 15.6% of the total 450,000 web databases [34]. Clearly, currently existing lists of online databases do not correspond to the scale of the deep Web. Therefore, technique for automatic finding search interfaces is of great interest to the people involved in directories' building and maintaining.

To summarize, while relatively good approaches (not ideal though) for querying web databases have been recently proposed one cannot fully utilize them as most search interfaces are undiscovered. Thus, the ability to automatically locate search interfaces to web databases is crucial for any application accessing the deep Web.

4.2 Motivation and Challenges

The problem of automatic identifying search interfaces arose during our characterization studies (see Chapter 3). Basically, we analyzed all pages on a particular web site to detect pages containing search forms. For instance, as depicted in Figure 4.1, two forms on **Amazon** has to be identified as non-searchable (form for registration on the left) and searchable (form for advanced book search on the right). When analyzing a particular web page we (as well as Chang et al. in their characterization study [34]) used semi-automatic approach – we automatically filtered out all pages without web forms and pages with those forms that are not search interfaces to web databases (i.e., forms for site search, navigation, login, registration, subscription, purchasing, polling, posting, etc.). At the filtering stage, we used a set of heuristic rules² that distinguish searchable forms from non-searchable ones. For instance, forms with a password field or with a file upload field are non-searchable. The goal of the filtering performed automatically was to filter out as many non-searchable forms as possible and, more importantly, to not filter out any searchable interfaces since, otherwise, we could not consider our estimates for the total number of deep web sites as reliable.

¹According to [57], the coverage of completeplanet.com could be overestimated.

²Obtained by manual reviewing of around 120 searchable and non-searchable forms. Similar but more simplified heuristics was used in [70].

Non-searchable Interface

Registration

New to Amazon.com? Register Below.

My name is:

My e-mail address:

Type it again:

Birthday: Month Day (optional)

☐ Interested in a corporate account? ([Learn more](#))

Protect your information with a password
This will be your only Amazon.com password.

Enter a new password:

Type it again:

[Continue](#)

Searchable Interface

Author:

☐ Exact Name ☒ Last, First Name (or Initials) ☐ Start of Last Name

Title:

☒ Title Word(s) ☐ Start(s) of Title Word(s)

Subject:

☒ Subject Word(s) ☐ Start(s) of Subject Word(s)

Publisher:

ISBN:

Refine Your Search (optional):

Category:

Format:

Reader age:

Language:

Publication Date:

the year

2009

Sort Results by:

Bestselling

[Search now](#)

Figure 4.1: Non-searchable and searchable interfaces at *Amazon.com*.

In other words, we preferred to use a simple set of rules which filter out only part of non-searchable forms but none of searchable forms than more advanced heuristics that filter out most non-searchable forms together with a few searchable ones. After the filtering, we manually inspected the rest of pages and identified searchable interfaces. In average, only each sixth form were identified as searchable at this step. Manual inspection was, in fact, a bottleneck of our characterization studies: we were not able to enlarge the sample size and, hence, increase the accuracy of our estimates since we had a restriction defined by the number of pages to be inspected during a given amount of time. Therefore, the automatic approach to identify search interfaces can significantly improve the accuracy of the estimates obtained in deep Web characterization studies.

Constructing directories of deep web resources like the one described in [47] is another application, where there is a vital need in automatic identifying of search interfaces. Such directories then can be utilized by conventional search engines. Particularly, many transactional queries (i.e., find a site where further interaction will happen [28]) can be answered better if results of such queries contain links to pages with search interfaces via which a user can eventually find the necessary information. To make this a reality, a search engine should have a directory of web databases (more exactly, directory of links to pages with search interfaces to databases), where databases are classified into subject hierarchies, and some kind of mapping that associates user's query with most related web databases. For instance, a user who issued a car-related query might be suggested to extend his/her search using web forms for car search. In this way, directories of web databases may have an important role in improving support of transactional queries by search engines.

We can divide the problem of constructing directory of deep web resources into three parts:

1. Finding web pages with search interfaces. One of the problems here is that search interfaces are very sparsely distributed over the Web, even within specific domains. For example, a topic-focused best-first crawler [33] was able to retrieve only 94 movie search forms while crawling 100,000 pages related to movies. Therefore, it is useful to develop a strategy for visiting such web pages that are more likely than the average page to have a search interface.
2. Recognizing searchable forms in automatic way. The task can be formulated in the following way: for a given page with a form, identify automatically whether a form is searchable (search interface to web database) or non-searchable. It is a challenging task since there is a great variety in the structure and vocabulary of forms and even within a well-known domain (e.g., car classifieds) there is no common schema that accurately describes most search interfaces of this domain.
3. Classifying searchable forms into subject hierarchy. Given a search interface (identified at the previous step) to a web database and a hierarchical classification scheme (e.g., Yahoo!-like directory [10]), which is a set of categories, define categories related to a database. There are several issues that complicate multi-class classification of searchable forms. First, existing data sets of searchable interfaces to be used as training sets are not large enough for multi-classification tasks. Second, many search interfaces belong to more than one different domains (see example in the next section). Third, certain search interfaces (with a few visible fields) are hard to classify reliably as textual content of such interfaces provide us with little or no information related to the subject of underlying databases.

4.3 Our approach: Interface Crawler

Our main goal is to develop a system, which is able to detect efficiently and automatically whether a particular web form is searchable or non-searchable and to identify a main subject of a database accessible via a searchable form. The first step described in the previous section, building an effective crawler that trained to follow links that are likely to lead to pages with search interfaces, is just partially considered. One particular application where we want to use our system is the deep Web characterization studies and, thus, our main concern is how to avoid identifying searchable form as non-searchable rather than how to increase the ratio of pages with search

interfaces to pages visited by a crawler. Anyhow, we performed several experiments (see Section 4.5) on possible crawling strategies.

Since we want to detect all search interfaces located on a set of pages it is crucially important to recognize both HTML [114] and non-HTML search interfaces (such as interface implemented as Java applets [5] or in Flash [2]). Though nowadays most search interfaces are HTML forms more and more non-HTML forms appear on the Web. In addition, client-side scripts (mainly in JavaScript [45]) embedded within an HTML page technically complicate processing of a web page: in order to proceed to extracting structural and textual content of such interface an application has to not only retrieve a page but also process a page content by a layout engine³. Without rendering by a layout engine JavaScript-rich forms are typically not detected at all. It is worth to mention here that almost all known to us approaches to the deep Web do not consider client-side scripts and, hence, ignore forms with embedded client-side code.

After a form is detected on a web page, our task is to identify it as searchable (search interface to web database) or non-searchable (forms for site search, navigation, login, registration, subscription, purchasing, polling, posting, etc.). This can be done by building a binary (domain-independent) classifier. It was observed in several works (e.g., in [34]) that there are structural differences between searchable and non-searchable forms. For example, the average number of text fields in non-searchable forms is higher than those in searchable forms. Thus, we can use these differences for training a binary classifier. However, we may expect a problem with forms that have one or two visible fields as such forms give us very little information about their structure. For instance, forms for site search⁴ that typically have only one visible text field are non-searchable while the form with two (select and text) visible fields shown in Figure 1.6 is searchable. To overcome this issue, we decided to divide all forms into two groups: those with one or two visible fields of select or text types (we call them *u-forms* for short as searchable u-forms are often interfaces to **un**structured web databases), and those with more than two visible fields (called *s-forms* as searchable s-forms often lead to **str**uctured databases). Apparently, two binary classifiers for u- and s-forms should exist and be trained on slightly different sets of form features.

Next task is to find which subject categories cover web database content (that accessible via identified search interface) in the best way. Currently, we consider classification of web databases into ten domains: airfare, auto,

³Software that takes web content and displays the formatted content on the screen. A layout engine (e.g., Gecko [92]) is typically used for web browsers or other applications that require displaying of web contents.

⁴They are not counted as search interfaces since they allow to search through indexable site's web pages.

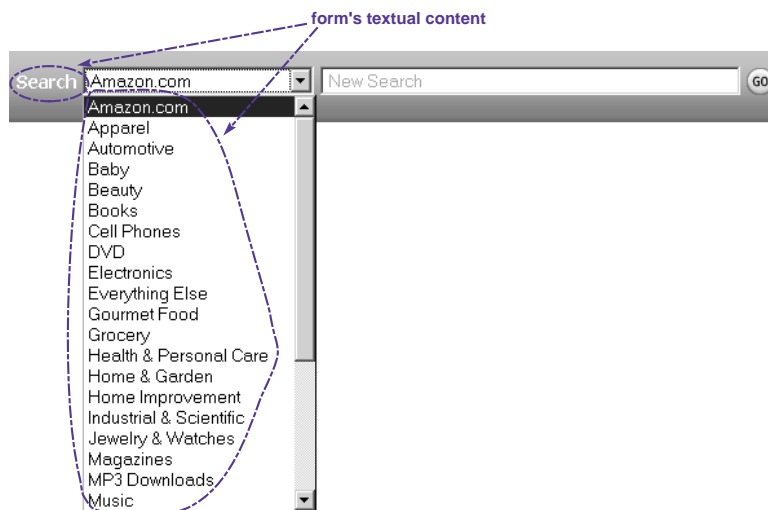


Figure 4.2: Search interface on the front page of *Amazon.com*.

book, travel, job, movie, music, real estate, rental, science. Classification of u- and s-forms into subject categories is performed separately. In both cases result of classifying a page containing a searchable form into subject hierarchies is considered to be only supplemental because a page in travel category can often contain an airfare search form. U-forms are sometimes quite problematic as they typically have little or no meaningful textual content⁵ and, moreover, often have to be classified into several domains. For example, consider search form on the front page of *Amazon.com* shown in Figure 4.2. Ideally, this form has one text and one select fields and has to be classified into three categories: book, movie, and music. However, all form’s textual content that can be extracted is pretty domain-independent: text string “Search” and option values of select field such as “Books”, “Apparel”, “Automotive”, etc. (see Figure 4.2). Our solution to this problem is to classify u-forms using post-query approach initially proposed by Gravano et al. [52]. The idea of post-query classification is issuing probing queries via searchable form and retrieving counts of matches which are returned for each probing query. If each probing query corresponds to some category than the number of returned matches points out the coverage of the database for this category (for example, if no matches are returned in the result of car-related query then database is not in auto domain). Unlike u-forms s-forms are easier to classify since, evidently, much more information about underlying database can be extracted from searchable s-form. In this way, we extract meaningful textual content of s-forms and pass it to the

⁵In case of HTML forms, textual content of the form is the text enclosed by the start and end `FORM` tags after the HTML markup is removed.

text classifier. Additionally, we can extract field labels and use them in the classification process as well. Particularly, He et al. [58] used field labels to cluster searchable forms. Though we do not use clustering approach in the current implementation of a system one can expect that clustering based on field labels might be a good supplement to the text classifier based on form's textual content.

Next section describes the architecture of the system called I-Crawler (**I**nterface **C**rawler) for automatic finding and classifying search interfaces.

4.4 Architecture of I-Crawler

Based on our approach described in the previous section we designed the architecture of the I-Crawler, which is shown in Figure 4.3. The I-Crawler consists of four main components: Site/Page Analyzer, Interface Identification, Interface Classification, and Form Database. The goal of the I-Crawler is to crawl suggested pages or web sites (as a typical web crawler does), extract all web forms from the visited pages, mark extracted forms as searchable or non-searchable, and store them in the Form Database. Additionally, searchable forms should be classified into the subject hierarchies.

The Site/Page Analyzer is responsible for form crawling efficiency. For a particular web page it analyzes a page's site, links located on a page, etc. and suggests which links should be processed first (for instance, those that are most likely to lead to search interfaces) and which should be ignored. Currently we do not pay much attention to the Site/Page Analyzer since we concentrate on finding as many search interfaces as possible and, thus, prefer to perform mostly unrestricted crawling of pages/sites of interest. However, if a task is to find search interfaces to web databases in one particular domain then the Site/Page Analyzer is crucially important.

The second component, the Interface Identification is the most important in the current implementation. It includes three parts: Interface Detector, Structure Extractor, and Binary Classifier. The Interface Detector is responsible for detecting a form within a web page. Since both HTML and non-HTML forms⁶ (such as forms implemented as Java applets or in Flash) has to further extracted from a web page the Interface Detector processes a web page like a web browser does. If no form is detected then nothing is passed to the underlying modules; otherwise the code describing the detected form on a web page is passed to the Structure Extractor and the code of web page with the form's code removed is passed to the Interface Classification component (namely, to the Page Classifier). The Structure Extractor extracts the structure of a form such as form's URL, total number of fields,

⁶In the current implementation, only Flash forms are handled. Technically, SWF-files are parsed using Perl library *Perl::Flash* [111].

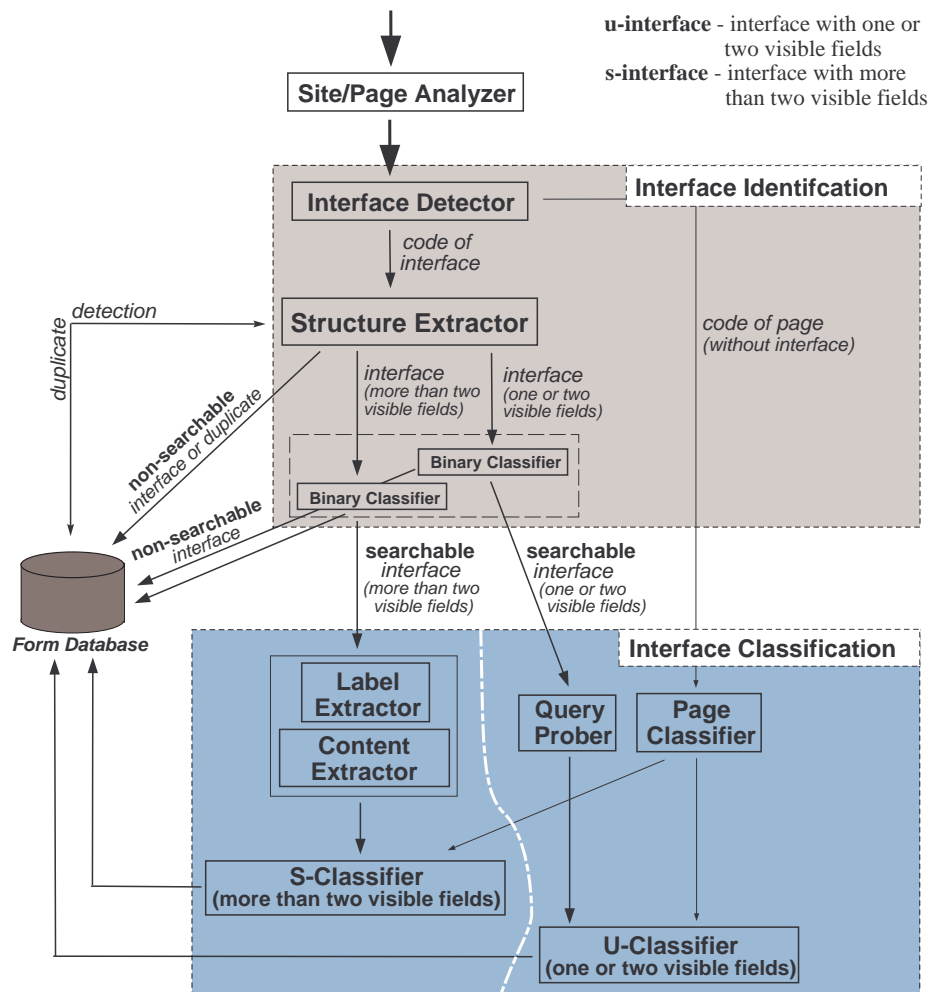


Figure 4.3: Architecture of I-Crawler.

number of text fields, etc. Information about form structure is explicitly defined in the the form code and, hence, the extraction of form structure process does not require any specific techniques. The Structure Extractor communicates with the Form Database to detect any duplicates that are already stored in the Form Database. It also has simple rules (e.g., a form with an upload file field is non-searchable) and stores non-searchable forms in the Form Database. If non-searchable or duplicate form is identified then the I-Crawler stops processing at this stage; otherwise, structural features of a form are passed to the Binary Classifier. In the current implementation, we have two binary classifiers – the first is for forms with one or two visible fields of select or text types (we call them *u-forms* for short) and the second is for the forms with more than two visible fields (*s-forms* for short). Both classifiers determines whether a form is searchable or non-searchable. If non-searchable form is identified then the I-Crawler stops processing at this stage; otherwise, searchable form is passed to the Interface Classification component for classification. Note that searchable forms of each type (u-forms and s-forms) are processed differently (see Figure 4.3).

The third component, Interface Classification is responsible for classification of searchable forms. U-forms are classified based on the results of form probing and results of page classification. S-forms are classified based on the extracted form content and field labels as well as results of page classification. The Page Classifier module analyzes the web page and assigns to it a score which defines the probability that a page belongs to a particular domain (in our experiments we built it using Rainbow [79]). The Query Prober module is an adopted version of the QProber system [52] that makes classification decisions by sending query probes through the u-form and analyzing number of matches reported for each query. The U-Classifier aggregates the classification information from the Page Classifier and the Query Prober and stores the u-form and its topic (or a set of topics) in the Form Database. Unlike u-forms s-forms are processed by the module which recognizes field labels of the s-form and extract the textual content of the form (e.g., predefined values of select fields). This information is then passed to the S-Classifier, which used it (together with supplementary data from the Page Classifier) to classify s-form into subject hierarchies. Finally, the s-form and its classification information is stored in the Form Database.

Last but not least, the Form Database component stores full information about all forms (searchable and non-searchable) encountered by the I-Crawler. This means that for each form all its features recognized and extracted during form identification and classification can be easily accessed at any time by any other component of the I-Crawler. Storing non-searchable forms is essential as allows to detect duplicate forms.

In the next section we describe our experiments and preliminary results obtained.

4.5 Experimental Results

We considered the following machine learning techniques: Support Vector Machine, Decision tree, and MultiLayer Perceptron. All classifiers were constructed with help of the WEKA package [112]. We used the following four datasets:

1. 216 searchable web forms from the UIUC repository [13] plus 90 searchable web forms⁷ collected by us and 300 non-searchable forms also collected by us.
2. Only searchable and non-searchable s-forms from the previous dataset.
3. 264 searchable forms from our collection of Russian search interfaces (described in Chapter 3) and 264 non-searchable forms in Russian collected by us.
4. 90 searchable u-forms collected by us and 120 non-searchable u-forms collected by us.

For each form in the datasets we retrieved the following 20 structural features: whether a form is HTML or non-HTML; presence of the string “search”, “find” or similar one within the `FORM` tags⁸; submission method; number of fields of each type (there are twelve types: text, select, etc.); number of items in selects; length of `action` attribute; sum of text sizes; presence of Javascript-related attributes; and number of tags within the `FORM` tags. The learning was performed using two thirds of each dataset and the testing using the remaining third. The error rates on test sets for each learning algorithms are shown in Table 4.1. We achieved slightly better results than the classifier described in [21] (last row of Table 4.1). It used 14 form features (that are mostly overlapping with the ones used by our learning algorithms) and its learning was performed using two thirds of dataset 1 (except they did not have additional 90 searchable forms, did not deal with JavaScript-rich and non-HTML forms and used distinct set of non-searchable forms) and the testing using the remaining third of dataset 1. More importantly, one can notice that classifier’s accuracy increases if dataset contains only s-forms and, thus, separation of forms into u- and s-forms is worth-while.

We selected decision tree algorithm (built on dataset 1) and applied it to finding search interfaces on real web sites. Three groups of web sites were studied: (1) 150 deep web sites randomly selected from our collection (see Chapter 3); (2) 150 sites randomly selected from “*Recreation*” category

⁷30 of which are JavaScript-rich and non-HTML forms.

⁸If it is an HTML form.

Table 4.1: Error rates for learning algorithms.

Learning Algorithm	Dataset			
	1	2	3	4
SVM	14.8%	12.9%	15.3%	18.6%
Decision tree	7.4%	6.2%	9.1%	15.7%
MultiLayer Perceptron	10.9%	10.7%	11.9%	14.3%
Decision tree in [21]	8.0%	-	-	-

Table 4.2: Number of web databases found.

Group of sites	Number of web dbs
(1)	39
(2)	31
(3)	3

of <http://www.dmoz.org>; and (3) 150 sites randomly selected based on IP addresses. Each site of each group was crawled by the I-Crawler to depth 5 (note that pages on sites of group (1) were ignored as we already know that there are search interfaces on these sites). All identified search interfaces were then checked and the number of found web databases were counted. The results, the number of found web databases for each group, is presented in Table 4.2.

The results clearly demonstrate that finding search interfaces (and eventually web databases) is more efficient if a crawler uses a certain strategy for visiting pages. Particularly, root pages of “already discovered” deep web sites are good start points for discovering new web databases.

4.6 Conclusion and Future Work

Due to the large scale of the deep Web the ability to automatically locate search interfaces to web databases becomes a key requirement for any application accessing the deep Web. In this chapter, we described the architecture of I-Crawler, a system for finding and classifying search interfaces. Specifically, the I-Crawler is intentionally designed to be used in deep Web characterization studies and for constructing directories of deep web resources. Unlike almost all other approaches existing so far, we recognize and analyze JavaScript-rich and non-HTML searchable forms. Though dealing with non-HTML forms is technically challenging it is an urgent issue as such forms are

going to reach a sizeable proportion of searchable forms on the Web. Our preliminary experiments showed that on similar datasets we were able to discover search interfaces more accurately than classifiers described in earlier works. Reliable classification of web databases into subject hierarchies will be the focus of our future work. One of the main challenges here is a lack of datasets that are large enough for multi-classification purposes.

Chapter 5

Querying the Deep Web

In this chapter, we present and discuss a query system for the deep Web called **DEQUE**¹ (Deep Web QUery SystEm) and address some of these above challenges. There are three steps to be performed for querying a web form within the framework of **DEQUE**. Firstly, a web form to be queried or several forms (if multiple form submission is required to obtain result pages) should be parsed and stored in a *form database* in accordance with the data model outlined in Sections 5.2 and 5.3. Storing is advisable as it quickens the query process. Nevertheless, non-stored forms can also be queried. In second step, a form query specified by the user is passed to **DEQUE** for validation and submission. A query is formulated in a web form query language called **DEQUEL** that allows the user to assign more values to form fields than it is possible when a form is manually filled out. Additionally, data from relational tables or data obtained from querying other web forms can be used as input values. The final step is retrieval of all result pages with query results and extraction of useful data from these pages according to the extraction conditions (if they are specified in a form query).

5.1 Introduction

The task of harvesting information from the deep Web can be roughly divided into three parts: (1) Formulate a query or search task description, (2) find sources that pertain to the task, and (3) for each potentially useful source, fill in the source's search form and extract and analyze the results. We will assume further that the task is formulated clearly. Step 2, source discovery, usually begins with a keyword search on one of the search engines or a query to one of the web directory services. The work in [33, 40, 78, 82] addresses the resource discovery problem and describes the design of topic-specific PIW crawlers. In our study, we assume that a potential source has

¹Pronounced as “deck”.

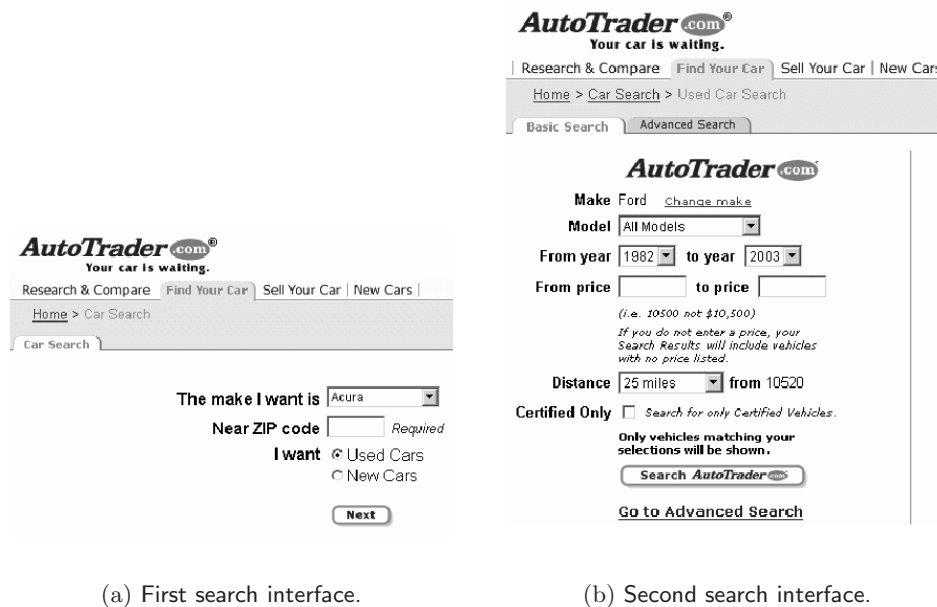


Figure 5.1: Search interfaces of *AutoTrader.com*.

already been discovered. So we limit our discussion to Step 3.

Retrieving and analyzing relevant information from the deep Web autonomously is a challenging problem. At present, the user is required to manually provide input values to web forms, and extract data from the returned web pages. The manual filling out forms is not feasible and cumbersome in cases of complex queries but these queries are essential for many web-based applications. We illustrate this with an example given below.

Example 5.1: The *AutoTrader* is the largest car Web site with over 1.5 million used vehicles listed for sale by private owners, dealers, and manufacturers. The search page² is shown in Figure 5.1(a). The web page contains a form for searching new or used cars. The form consists of a text box, a “Next” button, a “Make” selection menu and two radio boxes with options “Used Cars” and “New Cars”. The form returns a web page containing another form (called *child* form) shown in Figure 5.1(b) with fields for additional car search options. The submission of the form on the second page generates a web page containing the results of the query (see Figure 5.2). Suppose the user wishes to find information about “used” Japanese cars made in 1997 within US\$ 10,000 and available in “Chicago”. Especially, he/she is interested in “black” colored cars. To formulate such a query manually, the user has to fill up the form shown in Figure 5.1(a) by speci-

²Available at http://www.autotrader.com/findacar/index.jtmpl?ac_afflt=none.

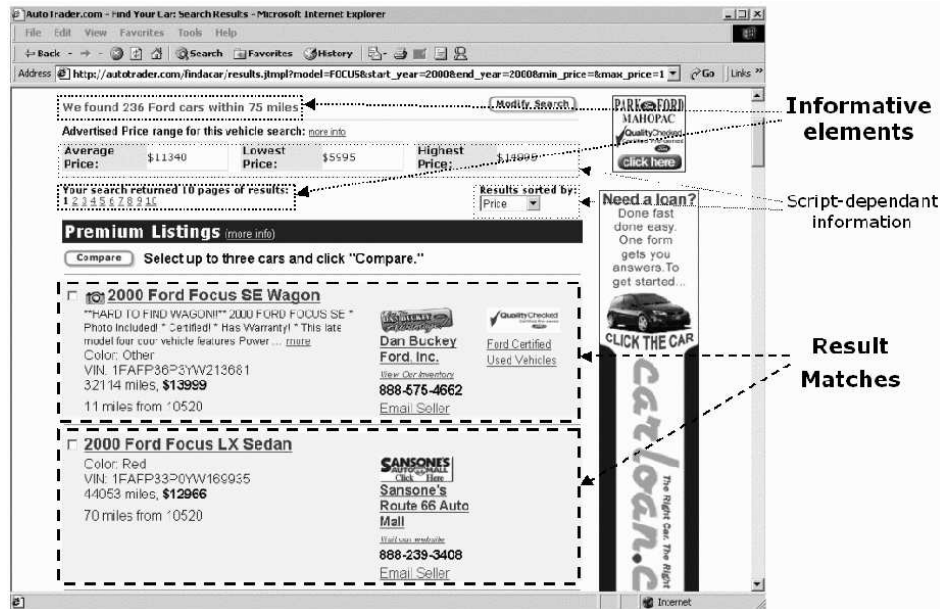


Figure 5.2: AutoTrader result page.

fying the make of the Japanese cars (eg. Honda, Toyota etc.) and the zip codes of Chicago in the fields labeled "The make I want" and "Near ZIP code" respectively. More importantly, the user has to formulate this query *repeatedly* for all different makes of Japanese car and at least one zip code of Chicago (that is, the user should fill in one of 88 zip codes corresponding to Chicago). To make matters worse, the user has to browse the results returned by each of this query to select all "black" colored Japanese cars. This is because the AutoTrader web site does not allow the user to specify the color of a car in the input forms (Figure 5.1). Indeed, even for one city and a list with a small number of car models, the process of filling out the AutoTrader forms, their submissions and looking through returned results is a very tedious and time-consuming affair.

The above task can be efficiently accomplished by using an automatic form querying system supported by a robust data extraction technique. However, there are several challenges in designing such an automated query mechanism as discussed below.

Automatic filling of forms: The task of automatic filling of forms is a challenging problem in the first place because of the variety of interfaces provided by web forms. Additionally, the user may not be aware of the values of all fields necessary to fill up the form. For example, the AutoTrader interface requires zip code as input. However, it is natural to assume that zip code(s) of a city is unknown to the user. Then, the query performing

the specified task should retrieve city zip codes from some zip database and substitute necessary input values for the corresponding form field.

Extraction of results: Another complex problem is to automatically extract query results from result pages since useful data is embedded into the HTML code. The search and the extraction of the required data from these pages are highly complicated tasks as each web form interface is designed for human consumption and, hence, has its own method of formatting and layout of elements on the page. For instance, Figure 5.2 depicts the original **AutoTrader** result page with formatting and non-informative elements (such as banners, advertisements, etc.). Accordingly, extraction tools must be able to filter out the relevant contents from the pages.

Navigational complexity: Dynamically generated web pages may contain links to other web pages containing relevant information and consequently it is necessary to navigate these links for evaluation of their relevances. Also, navigating such web sites requires repeated filling out of forms many of which themselves are dynamically generated by server-side programs as a result of previous user inputs. For example, the **AutoTrader** site produces a web page containing results after at least two successful form submissions (Figures 5.1(a) and 5.1(b)). These forms are collectively called *consecutive forms*.

Client-side programs: Lastly, client-side programs may interact with forms in arbitrary ways to modify and constrain form behavior. For instance, a text box control containing the total sales price in an order form might be automatically derived from the values in other text boxes by executing client-side script whenever the form is changed or submitted to the server. Programs written in JavaScript are often used to alter the behavior of forms. Unfortunately, it is computationally hard to automatically analyze and understand such arbitrary programs.

5.2 Modeling of A Single HTML Form

In this section, we discuss the data model for representing a single HTML form. Since for different HTML forms the nature and type of the layout markup are different, the web application should represent web forms in a uniform manner.

5.2.1 HTML Forms

An HTML form is a section of a document containing normal content, markup, special elements called controls (checkboxes, radio buttons, menus, etc.), and labels on those controls. Users generally “complete” a form by modifying its controls (entering text, selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a Web server, to a mail

server, etc.). Form controls are also known as *form fields*. An HTML form is embedded in its web page by a pair of “begin” and “end” `<FORM>` tags. Each HTML form contains a set of form fields and the URL of a server-side program (e.g., a CGI program) that processes the form fields’ input values and returns a set of *result pages*.

There are three essential attributes of the `FORM` element that specify how the values submitted with the form are processed: the value of the `action` attribute corresponds to the URL of a form processing agent (also, called server-side program), the HTTP method used to submit the form is defined by the `method` attribute, and the `enctype` attribute specifies the content type used for the form submission. For a web form F , we define submission information of a form F , $subinfo(F)$, as a list of three string elements specifying form action, HTTP method and content type, that is, $subinfo(F) = \{action, method, enctype\}$. For instance, the submission information of the form shown in Figure 5.1(a) is given by: $subinfo(FindCarForm) = \{http://autotrader.com/findcar/findcar_form2.jtmpl?ac_afflt=none, “get”, “application/x-www-form-urlencoded”\}$. Two forms F_1 and F_2 have the same submission information if: $subinfo(F_1) = subinfo(F_2)$.

5.2.2 Form Fields

The user fills out a form by associating a value or piece of text with each field of the form. A form field can be any one of the standard input objects: selection lists, text boxes, text areas, checkboxes, or radio buttons. Such tags as `BUTTON`, `INPUT`, `SELECT`, and `TEXTAREA` define form fields. The `name` attribute of the above tags defines the name of a form control (denoted by *fieldname*). In our study, we do not consider *file select*, *object* and *reset button* controls as they have no use in queries to most searchable databases deep behind web forms. The detailed description of each type is documented in HTML 4.01 specification [114]. A form field can be represented by the following attributes:

- **Field domain:** Field domain is a set of values (each value is a character string which can be associated with the corresponding form field. Some form fields have predefined domains, where the set of values are embedded in the web page with a form. Other fields have undefined domains (e.g., set of all text strings with specified length) from which their values can be chosen.
- **Field label:** Form fields are usually associated with some descriptive text³ to help the user understand the semantics of a field. Field label is

³Besides description of a form field such texts often indicate if a field is optional or required.

a string containing the descriptive information about the corresponding form field.

- **Initial field set:** Each field has initial value(s) (defined or undefined) which can be submitted with a form if the user does not modify a field through “completing” a form. Initial field set is a set of the field’s initial values. It is clear that for each form field an initial field set is a subset of the field domain.

Formally, given a field f of a web form F , $label(f_F)$ denotes a field label of a field f . Similarly, a field domain and an initial field set of a field f is denoted by $domain(f_F)$ and $iset(f_F)$ respectively. Note that $iset(f_F) \subset domain(f_F)$. We say that two form fields f and f' are the same, if: $formname(f) = formname(f')$, $type(f) = type(f')$, and $domain(f) = domain(f')$.

A field domain, field label and initial field set are defined for each field in a web form. For example, the field label of the “*address*” field is equal to “*Near ZIP code*”(see Figure 5.1(a)). Also, $domain(address) = \{ s, length(s) \leq 5 \}$, where s is a character string, and $iset(address) = \{ \emptyset \}$. The field label, domain and initial set of the “*make*” field in Figure 5.1(a) is given by the following: $label(make) = \{ \text{“The make I want is”} \}$, $domain(make) = \{ \text{“Acura”, “Alfa Romeo”, “AMC”, “Audi”, ...} \}$, and $iset(make) = \{ \text{“Acura”} \}$. Note that values of menu choices visible in a browser are not generally equal to values actually submitted with a form. Since the visible values are more informative, we consider a field domain as a set of visible menu choices. In the same manner, an initial field set is a set of pre-selected choices visible in a browser.

5.3 Modeling of Consecutive Forms

In the preceding section, we discussed how to model a single form. We now elaborate on the modeling of *consecutive forms*. Consider an HTML page containing one or more HTML forms. Every time we are interested in only one of these forms (called the *root form*). A response page is a page received in response to a form submission. In some cases, after submitting a *root form*, the returned page (or response page) contains another form (called the *child form*) which needs to be filled out. Similarly, after submitting a child form, the returned page may contain another child form to be filled out and so on. All the child forms are collectively called *descendant forms* for the given root form. The root and its descendant forms are collectively called *consecutive forms*. The submitted form is also the *parent form* for the following child form. Each descendant form is completely defined by its parent form, the values filled-in and the time of the parent form submission.

<p>The make I want is <input type="text" value="Acura"/></p> <p>Near ZIP code <input type="text" value=""/></p> <p>I want <input type="text" value="Ford"/></p>	<p>Make <input type="text" value="Ford"/> Change make</p> <p>Model <input type="text" value="All Models"/></p> <p><u>Certified Only</u> <input type="text" value="All Models"/></p> <p>From year <input type="text" value="2003"/></p> <p>From price <input type="text" value=""/></p> <p>Distance <input type="text" value="10520"/></p>
---	---

<p>The make I want is <input type="text" value="Acura"/></p> <p>Near ZIP code <input type="text" value=""/></p> <p>I want <input type="text" value="Toyota"/></p>	<p>Make <input type="text" value="Toyota"/> Change make</p> <p>Model <input type="text" value="All Models"/></p> <p><u>Certified Only</u> <input type="text" value="All Models"/></p> <p>From year <input type="text" value="2003"/></p> <p>From price <input type="text" value=""/> (i.e. 10500 not \$10,500)</p> <p>Distance <input type="text" value="10520"/></p> <p>Only <input type="text" value="MR2"/> selections will be shown.</p>
---	--

The "make" field of the root form

The "model" field of the descendant form

Figure 5.3: Form field dependency.

It also means that we always know the root form for each descendant form. A typical example of consecutive forms is the **AutoTrader** car search interface. After submission of the form (root form) shown in Figure 5.1(a), the returned page contains another form (child form) depicted in Figure 5.1(b). The form in Figure 5.1(a) is also the parent form for the form in Figure 5.1(b). Similarly, the latter is the descendant form for its parent form.

5.3.1 Form Type and Location

There are several differences between root and its descendant forms. One of them is the URL of a web page that contains a form. As a matter of fact, the main reason to have the URL of a page with a form is that an HTML code related to a form often specifies only the relative URL of a server-side program. Thereby, the page URL must be known to compose the absolute URL of a server-side program. The page URL of a child form can always be defined on the basis of the URL of a web page with a parent form, parent form's submission information, values submitted with a parent form and the submission time⁴. Thus, we do not consider the URL of a page with

⁴The URL of a response page equals to the URI of a server-side program if the POST HTTP method is used to send a submission data set to a program and to the URI of a server-side program with appropriately appended submitted values in case of the GET method using.

any descendant form as it is easily constructed using the URL of a page with a root form, and the processing information (submission information, submission data set, and submission time) of all forms beginning with a parent form and ending with a root form. In our study, we presume that the page URL of a root form is given by the user.

Formally, given a web form F , $formtype(F)$ specifies whether a form is root or descendant. $pageurl(F)$ is the absolute URL of a page that contains a form F , if F is a root form, and “null” otherwise. For example, the **AutoTrader** form shown in Figure 5.1(a) is the root form and $pageurl(\text{AutoTrader}) = \text{“http://autotrader.com/findacar/index.jhtml?ac_afflt=none”}$. Figure 5.1(b) depicts the **AutoTrader** child form for which: $formtype(\text{AutoTrader}_2) = \text{“descendant”}$ and $pageurl(\text{AutoTrader}_2) = \text{“null”}$.

5.3.2 Issues in Modeling of Consecutive Forms

There are some non-trivial issues concerning the modeling of consecutive forms. First, most consecutive forms have *dependencies* (called *form dependencies*) between root and descendant forms. For example, the **AutoTrader** search interface (see Figure 5.1) requires two consecutive forms to be fill-out, if the user searches for “Used Cars”, and three forms, if the “New Cars” option is chosen in the root form. Also, a *dependency* (called *form field dependency*) between form fields may exist. For instance, if the user selects “Toyota” make in the “make” menu, only “Toyota” models are available for the “model” select field of the **AutoTrader** child form shown in Figure 5.1(b). The dependency between “make” and “model” fields is shown in Figure 5.3. Choice of different car brands in the root form generates the child forms with different option values (corresponding to model names of the chosen car brand) for the “model” select field.

Second, each root form can generate a large number of different web pages containing the child forms. How these forms look like often depend not only on the values filled out in the root form but also on the submission time. In particular, it is irrational to store all possible child forms. For example, at least 48 different child forms⁵ are generated by a server-side program related to the **AutoTrader** root form (see Figure 5.1) only for searches for “Used Cars”. All these forms (one of such forms is depicted in Figure 5.1(b)) are highly similar to each other: the key difference is in the “model” select field. One can see that forms have different option values for the “model” field (Figure 5.4). Evidently, such similarity of descendant forms should be taken into consideration. We shall address this in Section 5.3.4 but first

⁵48 is the number of options in the “make” menu. In fact, the submission of data sets containing different values for the text “address” field does not return different child forms.

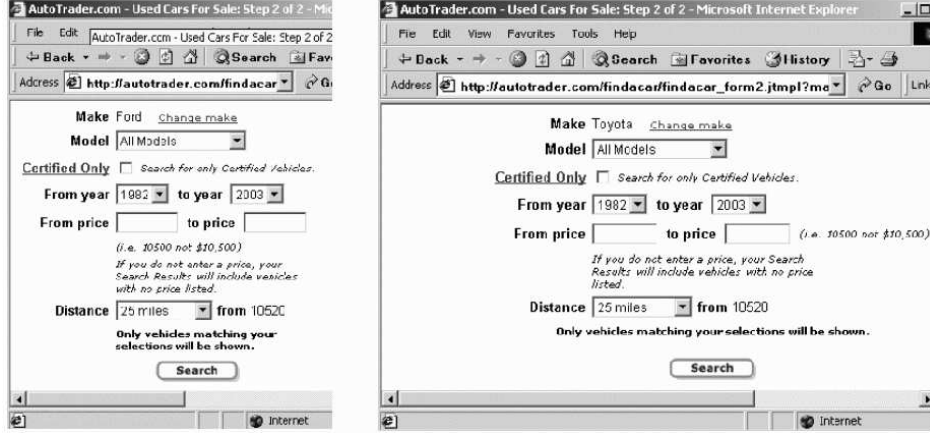


Figure 5.4: AutoTrader child forms.

introduce the notion of *submission data set*.

5.3.3 Submission Data Set of Consecutive Forms

When the user submits an HTML form, a user agent (web browser, etc.) processes it as follows: first, a form data set (a sequence of control name/value pairs) based on the values filled-in by the user is created. Then, the form data set is encoded to the content type specified by the `enctype` attribute of the `FORM` element. Finally, the encoded data is sent to a processing agent (server-side program) designated by the `action` attribute using the HTTP protocol (GET or POST) specified by the `method` attribute. The server-side program processes a form data set and returns a generated web page as the result of form submission. Formally, let F be a web form with form fields f_1, f_2, \dots, f_n . Let $\text{domain}(f_i)$ denotes the domain of the field f_i . Then, $S = \{s_1, s_2, \dots, s_n\}$ is the *submission data set* of F , if $\forall i = \overline{1, n}: s_i \in \text{domain}(f_i)$. We say that two submission data sets $S = \{s_1, \dots, s_n\}$ and $S' = \{s'_1, \dots, s'_n\}$ of a web form F are *equal* to each other, if $\forall i = \overline{1, n}: s_i = s'_i$. The following expression denotes that a web page with a form $F^{(S,t)}$ was returned by a server-side program as a result of submission of a form F_p at time t : $F_p(f_{p,1}, \dots, f_{p,n}) \xrightarrow{S,t} F_c^{(S,t)}(f_{c,1}^{(S,t)}, \dots, f_{c,k}^{(S,t)})$, where $F_p, F_c^{(S,t)}$ and S are a parent form with form fields $f_{p,1}, \dots, f_{p,n}$, a child form with fields $f_{c,1}^{(S,t)}, \dots, f_{c,k}^{(S,t)}$, and a submission data set of a form F_p respectively. We also define a set of child form's fields as follows: $\text{fieldset}(S, t) = \{\text{fieldname}(f_{c,1}^{(S,t)}), \dots, \text{fieldname}(f_{c,k}^{(S,t)})\}$. Similarly, we can describe several submissions of consecutive forms in the following way: $F_p \xrightarrow{S_1,t} F_{c_1} \xrightarrow{S_2,t} F_{c_2} \xrightarrow{S_3,t} \dots \xrightarrow{S_{m-1},t} F_{c_{m-1}} \xrightarrow{S_m,t} R$. This expression shows that

m forms: $F_p, F_{c_1}, \dots, F_{c_{m-1}}$ were submitted to obtain a result web page R where t is the submission time of the last child form submission (that is, $F_{c_{m-1}}$). We omit superscripts (S, t) as well as consecutive forms' fields to make the expression more compact.

Example 5.2: As an example, we describe the submission of the **AutoTrader** root form shown in Figure 5.1. We name this form *AutoTrader*. Suppose we search for “Used Cars”, then: $\text{AutoTrader}(\text{make}, \text{address}, \text{search_type}, \text{field_1}, \text{ac_afflt}, \text{borschtid}) \xrightarrow{S, t} \text{AutoTrader}_2$, where $S = \{\text{“Ford”}, \text{“10520”}, \text{“Used Cars”}, \text{“submit”}, \text{“none”}, \text{“21532053581465403997”}\}$ and $t = \text{“17 June 2002/6:34pm”}$. The result of submission is the page containing the **AutoTrader** child form (named *AutoTrader₂*) shown in Figure 5.1(b). This form contains eight visible and five hidden (invisible) form fields. The set of fields of the *AutoTrader₂* form is given by: $\text{fieldset}(S, t) = \{\text{model}, \text{certified}, \text{start_year}, \text{end_year}, \text{min_price}, \text{max_price}, \text{distance}, \text{field_1}, \text{advanced}^*, \text{advcd_on}^*, \text{make}^*, \text{address}^*, \text{search_type}^*\}$ ⁶. Note that hidden fields are often used to transmit the information about submitted values from one consecutive form to another. Thus, initial values of hidden fields *make*, *address*, *search_type* of the *AutoTrader₂* form are equal respectively to values assigned to fields *make*, *address*, *search_type* of the *AutoTrader* form (that is, “Ford”, “10520”, “Used Cars”).

5.3.4 Form Unions

We are now ready to discuss how to represent child forms. The idea of storing and querying consecutive forms is an attempt to combine forms. According to HTML specification, each form must have the **action** attribute in the **FORM** tag. This attribute specifies the URL of a server-side program to which the form contents will be submitted (if this attribute is absent, then the current document URL is used). Since web pages generated by the same server-side program are expected to be very similar, we will combine forms with the same submission information. A *form union* allows us to represent multiple forms sharing the same server-side program as one form. At the same time, each form included in a form union is easily accessible.

Definition 1 (Form Union) Consider a form F (root or descendant) and its two child forms: $F^{(S_1, t_1)}$ and $F^{(S_2, t_2)}$: $F \xrightarrow{S_1, t_1} F^{(S_1, t_1)}(f_1^{(S_1, t_1)}, \dots, f_k^{(S_1, t_1)})$ and $F \xrightarrow{S_2, t_2} F^{(S_2, t_2)}(f_1^{(S_2, t_2)}, \dots, f_m^{(S_2, t_2)})$. Suppose forms $F^{(S_1, t_1)}$ and $F^{(S_2, t_2)}$ have the same p ($p \leq \min(k, m)$) form fields, that is, fields of $F^{(S_1, t_1)}$: $f_1^{(S_1, t_1)}, \dots, f_p^{(S_1, t_1)}$ are the same as fields of $F^{(S_2, t_2)}$: $f_1^{(S_2, t_2)}, \dots, f_p^{(S_2, t_2)}$. Then, a union of forms $F^{(S_1, t_1)}$ and $F^{(S_2, t_2)}$ denoted by $W = F^{(S_1, t_1)} \cup F^{(S_2, t_2)}$ is defined, if and only if: $\text{subinfo}(F^{(S_1, t_1)})$

⁶Hidden fields are marked by asterisk.

$= \text{subinfo}(F^{(S_2, t_2)})$. Forms $F^{(S_1, t_1)}$ and $F^{(S_2, t_2)}$ are called component forms of a form union W . A form union W is considered as a child form of a form F and given by:

- Form fields of W are $f_1^{(S_1, t_1)}, \dots, f_p^{(S_1, t_1)}, f_{p+1}^{(S_1, t_1)}, \dots, f_k^{(S_1, t_1)}, f_{p+1}^{(S_2, t_2)}, \dots, f_m^{(S_2, t_2)}$;
- $\text{subinfo}(W) = \text{subinfo}(F^{(S_1, t_1)})$.

Note that $F^{(S, t)} = F^{(S, t)} \cup F^{(S, t)}$.

Example 5.3: Let us consider two child forms of the **AutoTrader** root form shown in Figure 5.4. They are results of submission of the root form depicted in Figure 5.1(a) with submissions data sets $S_1 = \{\text{"Ford"}, "10520", \text{"Used Cars"}, \text{"submit"}, \text{"none"}, "21532053581465403997"\}$, $t_1 = \text{"17 June 2002/6:34pm"}$ and $S_2 = \{\text{"Toyota"}, "10520", \text{"Used Cars"}, \text{"submit"}, \text{"none"}, "21532053581465403997"\}$, $t_2 = t_1$:

$\text{AutoTrader} \xrightarrow{S_1, t_1} \text{AutoTrader}_2^{(S_1, t_1)}$ and $\text{AutoTrader} \xrightarrow{S_2, t_2} \text{AutoTrader}_2^{(S_2, t_2)}$.

These forms (see Figure 5.4) are the same except one select field with name “model” and one hidden field with name “make” for which:

$\text{domain}(\text{model}^{(S_1, t_1)}) = \{\text{"All models"}, \text{"Aerostar"}, \dots, \text{"Escort"}, \dots\}$ (i.e., all “Ford” models), $\text{domain}(\text{make}^{(S_1, t_1)}) = \{\text{"Ford"}\}$ and $\text{domain}(\text{model}^{(S_2, t_2)}) = \{\text{"All models"}, \text{"4Runner"}, \dots, \text{"Corolla"}, \dots\}$ (“Toyota” models), $\text{domain}(\text{make}^{(S_2, t_2)}) = \{\text{"Toyota"}\}$.

The union form $\text{AutoTrader}_2^{\text{union}} = \text{AutoTrader}_2^{(S_1, t_1)} \cup \text{AutoTrader}_2^{(S_2, t_2)}$ is simply a copy of $\text{AutoTrader}_2^{(S_1, t_1)}$ (or $\text{AutoTrader}_2^{(S_2, t_2)}$) form plus two additional form fields: visible $\text{model}^{(S_2, t_2)}$ and invisible $\text{make}^{(S_2, t_2)}$ (or $\text{model}^{(S_1, t_1)}$ and $\text{make}^{(S_1, t_1)}$ respectively). Figure 5.5 depicts the representation of the $\text{AutoTrader}_2^{\text{union}}$ form given by our web form model.

The form union described in Example 5.3 contains nine visible and six invisible form fields. Note that each child form contains eight visible and five invisible fields. Thus, form unions allows us to store similar (sharing the same submission information) forms more effectively (fifteen fields instead twenty six in this example). On the other side, if necessary, we can easily obtain any component of a form union as we also store submission data set generating a child form and a set of child form’s fields.

5.3.5 Super Form

To query consecutive forms, our web form model must know the structure of all forms to be filled-out. As we noted above, most descendant web forms have very similar structure. This similarity motivates us to combine certain descendant forms into one form called super form. In the previous

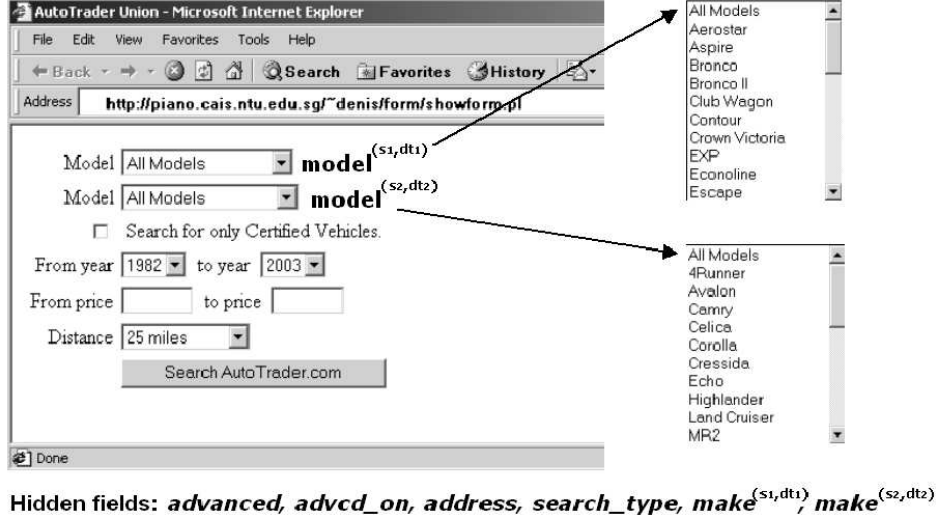


Figure 5.5: Union of two AutoTrader child forms.

section, we described how to construct a union of two forms. Note that a form union is also a form, and it shares the same submission information as its two component forms. Thus, lots of forms with the same submission information can be combined into one form. All child forms sharing the same **action** attribute (*component forms*) are combined into one form that contains all component forms' fields. At the same time, identical fields of child forms are presented in a super form only once. The formal treatment of a super form is given in the following way:

Definition 2 (Super Form) Consider a web form F with all possible submission data sets $S = \{ (S_i, t) \}$, where t is an arbitrary date/time. We suppose that a submission of the same sets but on the different time results in the same child form on the response page. Let a form $F^{(S_k, t)}$ be a child form of a form F with submission data set (S_k, t) , where $(S_k, t) \in S$. A super form F_{super} (or $F_{super}^{(S_k, t)}$) is given by:

$$F_{super} = F^{(S_k, t)} \cup \bigcup_{\forall S_j: (S_j, t) \in S: \text{subinfo}(F^{(S_j, t)}) = \text{subinfo}(F^{(S_k, t)})} F^{(S_j, t)}$$

The form $F^{(S_k, t)}$ is called a base form of the super form $F_{super}^{(S_k, t)}$. Note that $F_{super}^{(S_k, t)} = F_{super}^{(S_m, t)}$, if $\text{subinfo}(F^{(S_k, t)}) = \text{subinfo}(F^{(S_m, t)})$.

This definition introduces a new form called super form. We illustrate the notion of super form with an example below:

Example 5.4: Consider the **AutoTrader** root form shown in Figure 5.1(a). The submission of the **AutoTrader** form with the following submission data sets: $\{(S_i^{(used)}, t)\}$, where $S_i^{(used)} \in \{ \langle v_1, v_2, \text{"Used Cars"}, v_4, v_5, v_6 \rangle \mid v_1 \in \text{domain}(\text{make}), v_2 \in \text{domain}(\text{address}), v_4 \in \text{domain}(\text{field_1}), v_5 \in \text{domain}(\text{ac_affl}), v_6 \in \text{domain}(\text{borschtid}) \}$ returns web pages containing child forms (see forms in Figure 5.4) with identical submission information. The generated child forms differ from each other in the visible “*model*” field, domain values of which correspond to the chosen “*make*” in the root form, and invisible “*make*” field that simply contains the chosen “*make*”. The super form $F_{super}^{(used\ cars)}$ is equal to its base form (see Figure 5.4) plus 47 additional “*model*” select fields⁷ and 47 “*make*” hidden fields. The submission of the **AutoTrader** form with the following submission data sets: $\{(S_i^{(new)}, t)\}$, where $S_i^{(new)} \in \{ \langle v_1, v_2, \text{"New Cars"}, v_4, v_5, v_6 \rangle \}$ returns second series of child forms sharing the same submission information⁸. Thus, we can build the second super form $F_{super}^{(new\ cars)}$.

Among other things, dependencies between consecutive forms may be studied on the ground of a super form. Indeed, the analysis of submission data sets and generated child forms allows us to find existing dependencies. For instance, the constructed super form $F_{super}^{(used\ cars)}$ allows us to determine the dependence between the “*make*” field of the root form and the “*model*” field of any child form. This helps us to validate values in form queries. For instance, the query “*Find all Ford Corolla cars*” results in the web page with zero results as “*Corolla*” model doesn’t belong to the “*Ford*” makes. On the other side, the super form $F_{super}^{(used\ cars)}$ contains names of all possible used car models. Similarly, the super form $F_{super}^{(new\ cars)}$ may be used to find domain of all new car models. This domain does not necessarily identical to the domain of used car models as some car models are not manufactured in recent years (for example, “*Ford Bronco*” or “*Toyota Matrix*”).

A super form significantly simplifies the querying and storing consecutive forms. Indeed, to perform complex query (that requires much more than two submissions of a root form) on two consecutive forms we need to specify only two forms (root and super) instead of the number of combinations of a root and its child forms. For example, the simple search for “*Used Ford and Toyota cars within 10520 ZIP area*” requires four form submissions (two times of the root form in Figure 5.1(a) and one time each of child forms shown in Figure 5.4). Hence, three forms should be stored and then specified in the query. The super form based on any of these child forms reduces the number of forms to be specified to only two.

⁷47 is the number of options in the “*make*” menu of the **AutoTrader** root form exclusive of one option that corresponds to the base form.

⁸Note that the submission information of child forms generated by $\{(S_i^{(new)}, t)\}$ is different from the submission information of child forms generated by $\{(S_i^{(used)}, t)\}$.

Form Storing

Specify URL

Does this URL contain the root form? ☒ Yes ☐ No

Form ID, you should specify the parent form ID

Specify stored HTML file

Form reference name

Form description

Form has been extracted

ID / Reference name	15 : autoirador
URL	http://autotrader.com/findacar/index.jsp?ac_affli=none
FormType	root
Form fields	visible: make,address,search_type,field_1 invisible: borschid,ac_affli
Form description	AutoTrader.com - Used Cars For Sale: Step 1 of 2

Figure 5.6: Form Storing user interface.

Efficient storage is another advantage of our super form-based approach. The differences among descendant forms are often minor since the same server-side program generates them. A super form allows us to store only one base form and the differences between base form and all other forms with the same submission information. For instance, the form union presented in Example 5.3 requires storing only fifteen fields instead of twenty six. Thus, combining even two forms significantly reduce the number of fields stored.

5.3.6 Form Extraction

DEQUE presumes that all forms to be queried are stored in the form database according to the web form data model described earlier. This section describes the extraction of a web form from an HTML page.

An HTML page with a form to be extracted may be specified by the user through form storing GUI depicted in Figure 5.6. Since DEQUE allows to perform queries on consecutive forms, each response page must be examined for presence of web forms. Thus, for each web page (specified through GUI or retrieved from the Web) DEQUE constructs a logical tree representation of the structure of an HTML page, based on the Document Object Model (DOM) [12]. Next, the tree is forwarded to the following three extraction submodules of DEQUE that are responsible for the form extraction: *Form*

```

<td class=small valign="bottom" bgcolor="#FEF0DE">Departing from :</td>
1. <td valign=bottom bgcolor="#FEF0DE" class="small">Depart Date :</td>
2. <td valign=bottom bgcolor="#FEF0DE" class="small"> Search for: </td>
3. <td bgcolor="#FEF0DE" valign="bottom" colspan="-1">
4. <p align="left">&nbsp;</p></td></tr>
5. <tr bgcolor=#d2e1ff>
6. <td valign="top" class="small" bgcolor="#FEF0DE">&nbsp;</td>
7. <td valign="top" class="small" bgcolor="#FEF0DE">
<input maxlength=20 name=D_City size=12 style="width:110px">

```

Figure 5.7: Tag distance between field label and form field.

Element Extractor, *Label Extractor*, and *JS Function Extractor*.

Form Element Extractor: The form element extractor analyzes a tree to find the nodes corresponding to the FORM elements. If one or more elements (that is, HTML forms) exist, the pruned tree is constructed for each FORM element. For such tree construction we use only the subtree below the FORM element and the nodes on the path from the FORM to the root.

Further, the extractor retrieves data related to a form and its fields from the pruned tree in accordance with the web form model presented earlier. It should be noted that visible values as well as invisible values are retrieved. For example, consider the element `<option value="ALFA">Alfa Romeo</option>`, “Alfa Romeo” and “ALFA” are stored as visible and invisible option value respectively. The situation is worse with radio and checkbox fields. Visible values related to these types of form fields are not embedded in the INPUT elements that define such form fields. In most cases, we can find one of the visible values of a radio/checkbox field directly after the INPUT tag. For instance, “New Cars” as one of the visible values of the “search_type” field can be easily extracted from the following HTML code: `<input type="radio" name="search_type" value="new" onClick="changeList(1)">New Cars`. Unfortunately, there are web forms with more complex HTML markup in which the distance (in terms of the number of HTML tags) between a text element corresponding to a visible value and the INPUT element may be more than one tag.

Label Extractor: In DEQUE, the label extractor submodule is responsible for extraction of field labels. The label extractor (by default) begins with ignoring font sizes, typefaces and any styling information, so the corresponding pruned tree is simplified. We use a *visual adjacency* approach introduced in [90]. The key of this approach is that when the HTML code is rendered by the browser, the relationships between fields and their labels or fields and their visible values must be obvious to the user. In other words, irrespective

of how the page with the form is formatted, the phrase “*The make I want is*” (form label) or “*Used Cars*” (visual value) in Figure 5.1(a) must be visually adjacent to the select menu or one of the radio field choices respectively. Similarly, the text “*Near ZIP code*” must be visually adjacent to the corresponding textbox widget.

We use the following heuristic for identifying the label of a given form field (an analogous heuristic is used for domain values of radio/checkbox fields):

- Identify the pieces of text, if any, that are visually adjacent to the form field. We consider each piece of text as possible candidate to be a label of a form field, if a text piece contains less than eight words⁹, and the distance (in terms of HTML tags) between a text and a widget is less than eight. For example, consider Figure 5.7: the depicted piece of HTML code is a part of representation of the **Amadeus** form at http://www.amadeus.net/home/en/home_en.htm. As we can see the tag distance between the text “*Departing from :*” and related text box is seven tags.

For each candidate we compute actual pixel distances between the form widget and the candidate text pieces. For such computation, we use two JavaScript functions that returns the element’s real X and Y coordinates. On the base of the coordinates of a text element and a form field, the distance between them can be easily computed.

- If there are candidates to the left and/or above the form field, then we drop the candidates to the right and below. Note that visible values of radio/checkbox fields are usually to the right of the form field. Thus, we prefer the candidates to the right when extract domain values of the radio/checkboxes fields.
- If there are still two candidates remaining, the text piece rendered in bold or using a larger font size is chosen. Apparently, this step is omitted if the label extractor ignores styling information.
- If two candidates are not still resolved, then one of them is picked at random.

JS Function Extractor: Another extraction submodule of DEQUE is the JS¹⁰ function extractor. It is responsible for extracting JavaScript functions from the web pages. The main reason to extract such kind of functions is that values before submission to a server-side program would be checked in the same manner as in case of the manual form filling-out. Additionally,

⁹Most labels are either short words or short phrases.

¹⁰In our implementation we consider only JavaScript as a client-server script language.

nowadays web pages often contain scripts that define domain of values for some form field in dependence of the chosen value of another field.

In our work, we consider only extraction of client-side scripts. The content of the nodes related to the **SCRIPT** tags is considered for the function codes that are triggered if some form or field events occur. For instance, after extraction of the form event information by the form element extractor *formevent(AutoTrader)* = { “*onsubmit*”, “*return validateData();*” }, the JS function extractor searches the content of **SCRIPT** tags (or it searches the content of the file that may be linked to the web page as a container of client-side scripts for this page) for function named “*validateData*” and retrieves it.

5.4 Representation of Result Pages


In this section, we first discuss how the results returned by a deep Web query are represented in **DEQUE** and then describe an approach used to extract data from the result web pages.


5.4.1 Result Navigation


Perhaps the most common case is that a web server returns results a bit at a time, showing ten or twenty result matches per page. Usually there is a hyperlink or a button to get to the next page with results until the last page is reached. We treat all such pages as part of one single document by concatenating all result pages into one single page. Specifically, we will consider all the result web pages as one web page containing the search status string and N result matches, where, N may be specified in the web form query by one of the following special keywords: (1) **ALL** (default keyword) - all result matches from each page; (2) **FIRST(x)** - first x matches starting with the first result page; (3) **FIRSTP(y)** - all matches from first y result pages. The specified keywords should be specified in the extraction part of the **SELECT** operator that will be discussed later.

5.4.2 Result Matches

The next step to complete our result page representation is to examine result matches. An HTML code related to a match is often organized as tables using different web styles, fonts, colors, images and so on. As a matter of fact, only text elements and hypertext links of a result match are informative. Thereby, we ignore an HTML layout of a result match and focus on text strings and hyperlinks embedded in its HTML code. Thus, each result match is represented as a set of text strings and links. Links have their own internal structure similar to the structure of the HTML

☐  **2000 Ford Focus SE Wagon**
****HARD TO FIND WAGON!**** 2000 FORD FOCUS SE *
 Photo Included! * Certified! * Has Warranty! * This late
 model four door vehicle features Power ... [more](#)
 Color: Other
 VIN: 1FAFP36P3YW213681
 32114 miles, **\$13999**
 11 miles from 10520


Dan Buckey
Ford, Inc.
[View Our Inventory](#)
888-575-4662
 Email Seller


 Ford Certified
 Used Vehicles






link ₁	link ₂	text ₁	link ₃	text ₂	text ₃	text ₄	text ₅	link ₄	link ₅	link ₆	text ₆	link ₇	link ₈	link ₉
	2000 Ford Focus SE Wagon	**HARD TO FIND WAGON!** 2000 FORD FOCUS SE * Photo Included! * Certified! * Has Warranty! * This late model four door vehicle features Power ...	more	Color: Other	VIN: 1FAFP36P3YW213681	32114 miles. \$13999	11 miles from 10520		Dan Buckey Ford, Inc.	View Our Inventory	888-575-4662	Email Seller		Ford Certified Used Vehicles

Figure 5.8: Result match representation.

hyperlink, that is, the link label and the URL of the link. Note that if an HTML hyperlink label is an image, the corresponding link label is a text string defined by the `alt` attribute of the `IMAGE` tag or simply the word “Image”. Figure 5.8 shows the first result match from the `AutoTrader` result page (see Figure 5.2), and the text strings and links corresponding to this match (such strings and links may be stored in HTML or XML format¹¹). In this way, each result match is considered as a single row in a table with attributes of two types: *text* and *link*. Any value corresponding to the link type attribute consist of a hyperlink label and the URL of the hyperlink. The default attribute names are $text_i$, where i corresponds to the number of occurrence of the text element in the HTML code related to result match, and $link_j$, where j is the number of occurrence of the hyperlink in the code.

Since result matches even from the same result page may have different structure (in particular, different number of text strings or links), the representation of several matches in one table is ambiguous. For example, the second result match from the `AutoTrader` result page (see Figure 5.2) has five text elements and five hyperlinks, and hence, has ten attributes in the table representation. At the same time, the first match, which is shown in Figure 5.8, includes fifteen attributes (six text and nine link attributes). Actually, the finding of common attributes for all result matches extracted from result pages is a complicated problem. We give a brief description of our approach to this problem in Section 5.4.4.

¹¹Currently, we implement the *result database* capable to store an HTML code related to a result match. Thus, text strings and links can be easily extracted from the code.











link ₁	link ₂	text ₁	link ₃	text ₂	text ₃	text ₄	text ₅	link ₄	link ₅	link ₆	text ₆	link ₇	link ₈	link ₉
	2000 Ford Focus SE Wagon	**HARD TO FIND WAGON!!**2000 FORD FOCUS SE * Photo Included * Certified * Has Warranty! * This late model four door vehicle features Power ...		Color: Other	VIN 1FAFP36P3YW213681	32114 miles, \$13995	11 miles from 10520		Dan Buskey Ford, Inc.	View Our Inventory	888- 575- 4662	Email Seller		Ford Certified Used Vehicles
	2000 Ford Focus LX Sedan			Color: Red	VIN 1FAFP33P0YW169935	44053 miles, \$12966	70 miles from 10520		Saegone's Route 66 Auto Mall	Visit our website	888- 239- 3408	Email Seller		
	2000 Ford Focus SE Wagon				VIN 1FAFP36P5YW375828	32532 miles, \$13490	66 miles from 10520		Ray Price Motors			Email Seller		Mercury Certified Pre- Owned Vehicles

Figure 5.9: Result table for AutoTrader result pages.

For the time being, we assume that it is possible to build a *Result Table (RT)* for several result matches (or in other words, for result pages). Figure 5.9 depicts a partial view of the result table corresponding to the result matches from all AutoTrader result pages (the first result page is shown in Figure 5.2).

Initially, a result table is built with default attribute names: $text_i$ and $link_j$, where i and j are some indexes and $1 \leq i \leq k$, $1 \leq j \leq m$, where k and m are numbers of text and link attributes of a result table respectively. For the Autotrader example shown in Figure 5.9 – $k = 6$, $m = 9$ and the number of rows is 236. The operator DEFINE described in the next section allows the user to give result table's attributes more descriptive names.

5.4.3 DEFINE Operator

The operator DEFINE with keyword ATTRIBUTE is used to define more suitable attribute names for the result table. Two types of syntax are available. If the result table has already been created (with default attribute names), then the syntax (also see Appendix B.1.1) is as follows:

```
DEFINE ATTRIBUTE <default attrib. label> <new attrib. name>
                FOR <form label>
```

The statement defines a new attribute name for the specified default attribute name of the result table. Since the result table is created for the representation of the result pages generated by some server-side program (a processing agent of some web form) we specify the form name in the **FOR** clause. Thus, a result table corresponds to a form. Note that in case of query on consecutive forms a result table is defined for last submitted form.




link ₁	carpage	text ₁	link ₃	color	text ₂	text ₄	text ₅	link ₄	dealerpage	link ₆	text ₆	link ₇	link ₈	link ₉
	2000 Ford Focus SE Wagon	**HARD TO FIND WAGON!!** 2000 FORD FOCUS SE * Photo Included! * Certified! * Has Warranty! * This late model four door vehicle features Power ...	more	Color: Other	VIN: 1FAFP36P3YW213681	32114 miles, \$13999	11 miles from 10520		Dan Buccey Ford, Inc.	View Our Inventory	888-575-4662	Email Seller		Ford Certified Used Vehicles

Figure 5.10: Modified result table.

For example, the submission of the form depicted in Figure 5.1(b) produces the result pages (in particular, the page shown in Figure 5.2). If this form is denoted as “*AutoTrader2*” then column names in the result table can be defined using the **DEFINE** operator as follows:

```

DEFINE ATTRIBUTE link2 carpage FOR AutoTrader2;
DEFINE ATTRIBUTE link5 dealerpage FOR AutoTrader2;
DEFINE ATTRIBUTE text2 color FOR AutoTrader2.

```

These statements rename specified attributes of the result table for the **AutoTrader** result pages. The modified result table is shown in Figure 5.10.

The second type of syntax of the **DEFINE** operator is used to specify attribute name(s) if the result table was not generated before. In this form the **DEFINE** operator may be used to define extraction conditions for result pages generated by a particular server-side program. Recall that a web form specified in the **DEFINE** operator refers to a server-side program. The syntax (see Appendix B.1.2) of the operator is as follows:

```

DEFINE ATTRIBUTE <type> <set of attribute names>
    CONDITION <condition on text> | <condition on label>
    FOR <form label>

```

First, the operator specifies the type of attribute(s) (**TEXT** and **LINK** correspond to the text and link types respectively). Since more than one link or text may satisfy the extraction conditions, several attribute names may be specified. However, the operator requires at least one attribute name to be specified. Then, if necessary, the attribute names will be given by subindexing of the specified attribute name. The **CONDITION** clause specifies conditions on text strings or hyperlinks respectively of each result match. The satisfied text string or hyperlink will be presented in the result table as the value of the column specified by the attribute name.

The syntax assumes that each web form has its own set of the result table’s attributes. As mentioned earlier, the *result database* stores an HTML code of each result match. This allows us to define data for extraction from stored results anytime using of the **DEFINE** operator.

carpage
<u>2000 Ford Focus SE Wagon</u>
<u>2000 Ford Focus LX Sedan</u>
.....
.....
<u>2000 Ford Focus SE Wagon</u>
.....
.....

Figure 5.11: Result table based on extraction conditions.

The following example is also based on the results generated after submission of the **AutoTrader** form (see Figure 5.1(b)). Suppose that there were no any queries on the **AutoTrader** forms issued by the user. The user may assume what links and text elements may be on the result pages. In particular, it is clear that some links or text elements would contain text string “*Ford Focus*” if the user searches for “*Ford Focus*” cars using the **AutoTrader** web interface. Then, the **DEFINE** operator may be specified as follows:

```

DEFINE ATTRIBUTE LINK carpage
CONDITION (label contain “Ford Focus”, url contain http://autotrader.com)
FOR Autotrader2

```

If the result table is not created for the **AutoTrader2** form, the execution of this statement defines a table with one column called *carpage*. Then, the query (*find used “Ford Focus” cars made in 2000*) returns the results (the first result page is shown in Figure 5.2) that are represented as it is shown in Figure 5.11.

5.4.4 Result Extraction

At present, **DEQUE** extracts the pieces of HTML code that correspond to result matches. The main idea of such extraction is the regularity of HTML patterns related to result matches or, in other words, that a result page contains the number of HTML patterns possessing nearly identical structure. We can find them by searching an HTML tree for the number of sibling sub-trees. Figure 5.12 shows an example of the HTML tree. In this example, the HTML code contains twelve siblings sub-trees, tables (defined by **TABLE** tags) with the identical structure. These tables correspond to twelve result matches laid out on the result page. We additionally require that sub-trees to be extracted must contain several hyperlinks and text strings to distinguish them from subtrees related to different navigation menus available on the page.

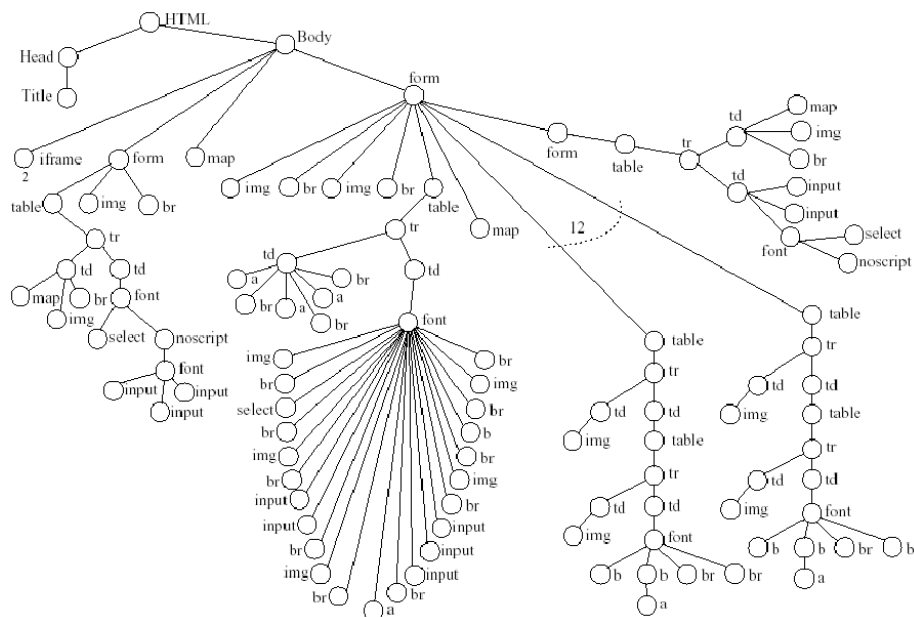


Figure 5.12: Example of HTML tree.

The result table for a set of result pages is built by DEQUE using the `DEFINE`-statements or conditions specified in a form query (see Section 5.5). In case no information on the result table is provided, a modified version of the approach in [38] is used. Pattern discovery in `RoadRunner` [38] is based on the study of similarities and dissimilarities between two HTML pages at a time; mismatches are used to identify relevant inner structures of result matches. In our approach, we compare first two result pages to determine a common inner structure of result matches and then extract common matches’ attributes from all result pages using the discovered pattern. However, this can be done effectively if there are at least two result pages both containing sufficient number (more than ten according to our experiments) of result matches. Otherwise, when only one result page is returned (or second result page contains a few result matches), DEQUE analyzes two HTML pages that are artificially constructed on the base of available result matches by separating them into two pages. The reader may refer to [38] for further details about identification of relevant inner structures within pages generated by web forms.

5.5 Deep Web Query Language (DEQUEL)

In this section, we discuss the base part of the DEQUEL devoted to the formulating form queries. The DEQUEL is aimed at providing such applications

as automated web agents searching for specific domain information, hidden Web crawlers, etc. with expressive query interface to data in deep Web. The proposed language, designed specifically to query web forms, is an expressive web query language that permits queries on topology (filling single or consecutive forms) and document structure (within result pages). Nevertheless, in the context of data extraction, the DEQUEL is less expressive than up-to-date wrapping languages [50].

Further, we describe the value assignment process, DEQUEL’s **SELECT** operator, and give examples of form queries formulated in the DEQUEL. The complete syntax of the DEQUEL is given in Appendix B.

5.5.1 Value Assignment

The result pages are generated by a server-side program on the basis of the values submitted via web forms. In the DEQUEL, a value or a set of values are assigned to a some form field in the following way: *form_name.field_name* = *value* or *form_name.field_name* = {*set_of_values*}. For instance, *AutoTrader.make* = {“Ford”, “Toyota”} assigns values “Ford” and “Toyota” to the “make” field of the **AutoTrader** interface. Corresponding initial field sets are assigned if values for some fields of a form are not specified in a query¹². As a rule, the value assignment for button and hidden type fields are not necessary. Thus, these types of fields are omitted in the **SELECT** statement.

The following type of the value assignment is especially useful for text type fields: *form_name.field_name* = {*LABEL, n*}, where *n* is a number. The keyword **LABEL** specifies that domain values of some field with similar descriptive information (label) will be assigned to the specified form field. *n* specifies how many domain values are assigned. For instance, if our form database stores some form with the field “zip” for which: *label(zip)* = “ZIP code” and *domain(zip)* = {“60601”, “60602”, ..., “60612”} then the assignment *AutoTrader.address* = {*LABEL, 3*} specifies that values “60601”, “60602”, and “60603” will be assigned to the “address” field. This is based on the assumption that the label of the “zip” field is semantically the closest one to the label of the “address” field which is: *label(address)* = “Near ZIP Code”.

The DEQUEL also allows us to assign values from relational tables and results of previous form queries (these results must be presented as result tables). The syntax is the following: *form_name.field_name* = {*relational_table_name.attribute_name, k*} or *form_name.field_name* = {*query_name.attribute_name, m*}, where *k* and *m* specify how many values will be assigned to a form field; *relational_table_name.attribute_name* defines the column of the specified relational table; and *query_name.attribute_name*

¹²Null string is assigned to an unspecified text field without predefined value.

specifies the column of the result table that stores results of specified form query. For example, suppose we need to send links to web pages containing information about used “*Ford Focus*” cars made in 2000 to some friends via SMS service. Suppose we use the text field named “*mes*” in the form called **SendMessage** for sending the SMS. Also, assume that our search using the **AutoTrader** web interface was stored in the result table (we call the table “*Focus2000*”) with one column called “carpage” (as it is shown in Figure 5.11). Then, we can assign two links from this stored result table to the “*mes*” field (that contains the text of SMS message) as follows: $SendMessage.mes = \{Focus2000.carpage, 2\}$.

Web forms impose some restrictions on values. Although the number of assigned values is not limited, only values pertaining to form field domains are processed. The rest of values are ignored. Thus, the assignment $AutoTrader.make = \{\text{“Ford”}, \text{“Microsoft”}, \text{“DELL”}\}$ will be transformed into $AutoTrader.make = \{\text{“Ford”}\}$ as “*Microsoft*” and “*DELL*” are not in $domain(make)$.

5.5.2 DEQUEL Syntax

The DEQUEL and especially its SQL-like retrieval operator, **SELECT**, are intended to provide more convenient and efficient way to fill out web forms. The syntax of the DEQUEL is as follows:

```

<query> ::= SELECT      [<number of results>]
                        <set of result table attributes>
                        [<set of assigned values>]
                        [AS <query label>]
                [ FROM   <source set>  ]
                [ WHERE  <assignment set> ]
                [ CONDITION <condition set> ]

```

The **SELECT**, retrieval operator of the DEQUEL, consists of four parts: *extraction*, *source specification*, *assignment*, and *condition* part. The first part of the **SELECT** statement concerns query results returned by the DEQUEL query processor. The number of results defines how many result matches should be extracted from the result pages for each submission data set defined in the assignment part. The set of the result table attributes can be specified if such table has been created before. However, the default attribute names $link_i$, $text_j$ may be used in case the result table has not been created. The **AS** clause specifies that the results of this query will be stored and defines the reference to these results.

Form(s) to be queried, relational table(s) used as a source of input data, the form URL(s) if form(s) is not pre-stored in the form database, and names of stored query results must be specified after the **FROM** clause in the “source set”.

href ₁	href ₂	href ₃	href ₄	href ₅	href ₆	href ₇	href ₈
1	60601	Chicago	IL	5,865	Covok	312	Central
2	60602	Chicago	IL	0	Covok	312	Central
3	60603	Chicago	IL	0	Covok	312	Central
4	60604	Chicago	IL	84	Covok	312	Central
5	60605	Chicago	IL	12,847	Covok	312	Central
6	60606	Chicago	IL	1,662	Covok	312	Central
7	60607	Chicago	IL	16,162	Covok	312	Central
8	60608	Chicago	IL	87,815	Covok	773	Central
9	60609	Chicago	IL	79,763	Covok	773	Central
10	60610	Chicago	IL	49,470	Covok	312	Central
11	60611	Chicago	IL	25,733	Covok	312	Central
12	60612	Chicago	IL	35,879	Covok	312	Central
...
87	60699	Chicago	IL	0	Covok	312	Central
88	60701	Chicago	IL	0	Covok	773	Central

(a) Chicagozips query results.

carpage	dealerpage	color
1997 Toyota Corolla DX	Toyota on Western	Color: Black
1997 Toyota Camry LE	Clean Cars	Color: Black
1997 Toyota Avalon	Car Outlet	Color: Black
1997 Nissan 200SX SE-R	Continental Nissan	Color: Black
1997 Nissan Maxima	Dodge City of Countryside	Color: Black
1997 Nissan PICKUP King Cab XE 4x4	Woodmar Auto Sales	Color: Black
...
1997 Honda Civic EX Coupe	Jacobs Twin Auto Plaza	Color: Black
1997 Honda Passport 4 Door 4x4	Car Mart USA	Color: Black

(b) Japancars97 query Results.

Figure 5.13: DEQUEL query results.

The next section of the **SELECT** statement, the **WHERE** clause defines values to be assigned to the form fields (the fields must pertain to forms specified in the **FROM** clause). Lastly, conditions on the data extracted from the result pages are specified in the **CONDITION** clause. In the current implementation, parentheses are not allowed and the priority of AND/OR in the condition set is based on the occurrence of operators from left to right.

We illustrate the syntax of the DEQUEL with some examples.

5.5.3 Examples of DEQUEL Queries

Example 5.5: Suppose that we wish to find the ZIP codes of Chicago, USA. The web form at <http://zipfind.net> (called ZIPFind) is used to find ZIP codes. Assume that this form has been stored in our form database. Then the query is formulated as follows:

```
SELECT ALL AS Chicagozips
FROM zipfind
WHERE zipfind.104 = "Chicago"
```

The result database stores the query results and creates the **Chicagozips** reference to them. Since we do not specify the result table attributes the default attribute names are used. The results of the query are shown in Figure 5.13(a).

Example 5.6: Suppose we wish to find “best”¹³ flights from Singapore to London on dates: October 28, 2002; November 14, 2002; and January 24, 2003 with available seats in business or economy classes. The Amadeus search interface is used for flight searching. To retrieve the relevant results we formulate the query as follows:

¹³The flight duration is minimal.

flight	depart	arrive	stops_aircraft	duration	business_seat	economy_seat	amadeus.D_month	amadeus.D_Day
Singapore Airlines SQ 318	09:00	:5:10	Non-stop 744	14h:10min	Yes	Yes	October	28
Singapore Airlines SQ 320	12:40	:8:50	Non-stop 744	14h:10min	Yes	Yes	October	28
British Airways BA 016	22:40	04:55 + 1 day(s)	Non-stop 744	14h:15min	Yes	Yes	October	28
British Airways EA 7371	08:05	:4:05	Non-stop 747	14h:00min	N/A	Yes	November	14
Qantas Airways Ltd QF 331	08:05	:4:05	Non-stop 744	14h:00min	Yes	Yes	November	14
Virgin Atlantic VS 7318	09:00	:5:10	Non-stop 744	14h:10min	N/A	Yes	November	14
British Airways EA 7371	05:00	:1:00	Non-stop 747	14h:00min	N/A	Yes	January 2003	24
Qantas Airways Ltd QF 331	05:00	:1:00	Non-stop 744	14h:00min	Yes	Yes	January 2003	24
Virgin Atlantic VS 7318	09:00	:5:10	Non-stop 744	14h:10min	N/A	Yes	January 2003	24

Figure 5.14: Seatsaval query results.

```

SELECT FIRST(3) flight, depart, arrive, stops_aircraft, duration, busi-
ness_seat, economy_seat, amadeus.D_Month, amadeus.D_Day
AS Seatsaval
FROM amadeus
WHERE amadeus.D_City = "Singapore" AND
      amadeus.A_City = "London" AND
      (amadeus.D_Month, amadeus.D_Day) = ("October", "28",
      "November", "14", "January 2003", "24")
CONDITION business_seat = (text contains "Yes") OR
      economy_seat = (text equal "Yes")

```

It should be noted that the assignment $(form.field_1, form.field_2) = (v_1, v_2, v_3, v_4)$ differs from the assignment $form.field_1 = \{v_1, v_3\}$ AND $form.field_2 = \{v_2, v_4\}$. The first means that a value v_1 is assigned to a field $field_1$ only if v_2 is assigned to $field_2$. While the last presumes that all possible combination of $\{v_1, v_3\}$ and $\{v_2, v_4\}$ may be assigned to the fields $field_1$ and $field_2$. Thus, in our example, the Amadeus form is not queried for such dates as *October 14, 2002* or *November 28, 2002* or *November 24, 2002*.

In the above query we specify using keyword FIRST(3) that first three result matches are extracted from the result page for each set submitted to the server-side program related to the Amadeus form. Further, we specify the result table's attribute names (suppose that the result table has been created and the table attributes have been given names such as *flight*, *depart* and so on) and assigned values. Note that the assigned values are not part of the result table. We specify them to make results clearer to understand (in this example, to distinguish flights on different dates). The **CONDITION** part of the query defines conditions on the values of the result table related to *business_seat* and *economy_seat* attributes. We require that the result match must have a value corresponding to its *economy_seat* attribute that equals "Yes" or a value for *business_seat* attribute that contains "Yes". The non-satisfied result matches are not presented in the query results. Figure 5.14 depicts the Seatsaval query results.

researcher:		japancars:	
id	name	Price	make
1	Coffey ET	10000	Toyota
2	Kulomaa MS
3	Lahti R	15000	Nissan
4	Lahtesmaa R
5	Kilpelainen I	12000	Mazda
	

Figure 5.15: Relational tables used in Examples 5.7 and 5.8: **researcher** and **japancars**.

Example 5.7: Given a list of researchers from some Graduate School related to natural sciences¹⁴, suppose that we wish to find all works published by these researchers in 2002. Assume that all researchers' names are stored in the relational table **researcher**(*id*,*name*) shown in Figure 5.15. The PubMed [8] form is used to search for published works. The query is formulated as follows:

```

SELECT authors, work, published, pubmed.TEXT
FROM pubmed, researcher
WHERE pubmed.db = "PubMed" AND
      pubmed.TEXT = {researcher.name,all}
CONDITION published = (text contains "2002")

```

According to the query, results are presented in the four-column table with attributes *authors*, *work*, *published*, *pubmed.TEXT*. Similar to the previous query the values assigned to the *pubmed.TEXT* field defines values of the fourth column. These values are taken from the specified relational table **researcher**. We can also specify how many values are used as input to the "*TEXT*" field of the PubMed interface. *all* defines that all corresponding values of the relational table are assigned to the "*TEXT*" field. In the case of the relational table above the specifying *pubmed.TEXT*=*{researcher.name,2}* is equivalent to the following assignment: *pubmed.TEXT*=*{ "Coffey ET", "Kulomaa MS" }*. Figure 5.16 shows the results of the query.

Example 5.8: Reconsider the query in Example 5.1. Let the forms in Figures 5.1(a) and 5.1(b) be named *Autotrader* and *Autotrader2* respectively. Assume that Japanese makes are stored in the relational table **japancars**(*price*,*make*) shown in Figure 5.15. Suppose that the following attributes

¹⁴We use data available at http://www.abo.fi/isb/research_groups.html.

authors	work	published	pubmed.TEXT
<u>Coffey ET, Smacene G, Hongisto V, Cao J, Brecht S, Herdegen T, Courtney MJ.</u>	c-Jun N-terminal protein kinase (JNK) 2/3 ...	J Neurosci. 2002 Jun 1;22(11):4335-45.	Coffey ET
<u>Karisola P, Alenius H, Mikkola J, Kalkkinen N, Helin J, Pentikainen OT, Repo S, Reunala T, Turjanmaa K, Johnson MS, Palosuo T, Kulomaa MS.</u>	The major conformational IgE-binding epitopes ...	J Biol Chem. 2002 Jun 21;277(25):22656-61.	Kulomaa MS
<u>Latinen OH, Hytonen VP, Ahlroth MK, Pentikainen OT, Gallagher C, Nordlund HR, Ovod V, Marttila AT, Porkka E, Heino S, Johnson MS, Airene KJ, Kulomaa MS.</u>	Chicken avidin-related proteins show altered biotin-binding and ...	Biochem J. 2002 May 1;363(Pt 3):609-17.	Kulomaa MS
<u>Tirola MA, Mannisto MK, Puhakka JA, Kulomaa MS.</u>	Isolation and characterization of <i>Novosphingobium</i> ...	Appl Environ Microbiol. 2002 Jan;68(1):173-80.	Kulomaa MS
...
...
<u>Karkonen A, Koutaniemi S, Mastonen M, Syrjanen K, Brunow G, Kilpelainen I, Teeri TH, Simola LK.</u>	Lignification related enzymes in <i>Picea abies</i> suspension cultures.	Physiol Plant. 2002 Mar;114(3):343-353.	Kilpelainen I

Figure 5.16: PubMed query results.

of the result table: *carpage*, *dealerpage*, *color* have been defined (see Section 5.4.3). Then the query is formulated as follows:

```

SELECT FIRSTP(2) carpage, dealerpage, color AS Japancars97
FROM autotrader, autotrader2, japancars, zipfind
WHERE zipfind.104={ "Chicago" } AND
      autotrader.make={japancars.make, all} AND
      autotrader.search_type="Used Cars" AND
      autotrader.zip={zipfind.rt.text2, 1} AND
      autotrader2.max_price="10000" AND
      autotrader2.from_year="1997" AND
      autotrader2.end_year="1997"
CONDITION color = (text contains "Black")

```

The FIRSTP(2) expression defines that result matches are extracted from the first two result pages. The zipfind form is used to find ZIP codes on the basis of city name. The submission of the zipfind returns 88 ZIP codes corresponding to "Chicago" (see Figure 5.13(a)). The expression {zipfind.rt.text₂, 1} defines that only one "text₂" value from the zipfind result table (named zipfind.rt) is used for providing ZIP code to the related AutoTrader form field. If the query presented in Example 5.5 was executed we can formulate this query by removing zipfind mentions in the third and fourth lines of the query and changing {zipfind.rt.text₂, 1} in the sixth line to {Chicagozips.text₂, 1}. The query results are shown in Figure 5.13(b).

Example 5.9: Suppose we wish to find only used Toyota black cars made in 1997 within US\$ 10000 available in Chicago. We can formulate a query similar to the previous but it is a good idea to reuse results of the query described in Example 5.8:

```

SELECT carpage, dealerpage
FROM Japancars97

```

WHERE *Japancars97.carpag*e = (label contains “*Toyota*”)

Here we used the *Japancars97* result table specified in Example 5.8. The results of the query are a two-column result table `result(carpag,dealer-pag)`.

5.5.4 DEQUEL Query Execution

This section describes the execution of the DEQUEL query formulated by the user through the form query UI. In the preceding sections, we introduced two operators: **DEFINE** and **SELECT**. The **DEFINE** operator specifies the extraction conditions on the data of the result pages. The query processing is defined exclusively by the **SELECT** operator.

Firstly, the *Query Processor* of DEQUE checks whether the same query was processed before. If yes, it returns the results of the processed query and indicates the date of processing. This definitely may lead to outdated results. For example, in case of querying a car classified database, some entries may describe cars which are not longer available. We did not deal with the problem of duplicate record detection [42] since, though clearly important, it is out of the scope of this thesis wherein we concentrate on querying and extraction aspects. Secondly, the specified DEQUEL query is parsed by the *Query Parser*. Currently, the query on two or more forms can be composed only if forms to be queried are consecutive (that is, these forms have the same root form). The **SELECT** statement envisages that a query may contain the URL of forms. Such forms are extracted before query evaluation.

All forms and specified relational tables (these tables must be also pre-stored in the relational database) are retrieved from the form and relational database by the *Storage/Retrieval Manager*. For each form field specified in a query, the *Query Evaluation Module* of DEQUE considers the form field domain, initial field set, label, and, at last, values indicated in the query. For instance, consider querying two consecutive forms: *autotrader(make, address, search-type, field_1, ac-afflt, borschtid)* (see Figure 5.1(a)) and *autotrader2(model, certified, start-year, end-year, min-price, max-price, distance, advanced, advcd-on, make, address, search-type)* (see Figure 5.1(b)). Suppose 24 values as relational input are assigned to the field *make*, five values are assigned to the field *max-price*, and initial field sets of all fields of both forms consist of one value. Then, 120 possible submission data sets will be validated. The potential submission data sets with the form field information such as field domains, labels, initial field sets for each form specified in the query are passed for the values’ validation.

For each form involving in the query, DEQUE considers the domain constraints corresponding to the form. Any potential submission data set must satisfy these domain constraints. Since we perform queries on consecutive

forms, two or more groups of the domain constraints may be considered. The steps given below validate the potential submission data set.

Consider a form F with fields f_1, \dots, f_k , its child form F_c with fields f_{k+1}, \dots, f_n and a potential submission data set $S = \{s_1, \dots, s_k, s_{k+1}, \dots, s_n\}$, where $\forall i = \overline{1, n}$: s_i is assigned to a field f_i . A data set S can be submitted if the following is true:

1. The data set $S' = \{s_1, \dots, s_k\}$ satisfies the domain constraints of a form F .
2. The data set $\{s_{k+1}, \dots, s_n\}$ satisfies the domain constraints of the form $F_c^{(S')}$ ($F_c^{(S')}$ is a result of submission of a form F with submission data set S').

The *HTTP Request Module* of DEQUE transforms the validated submission data sets into the HTTP GET or POST requests. If a query on consecutive forms is executed, only the last child form is submitted to its server-side program. Values assigned to the fields of all other consecutive forms are assigned to the hidden fields of the last child form. Indeed, the majority of descendant forms contains the hidden fields which initial values are equal to values assigned to the fields of the parent form. For example, consider the query on the **AutoTrader** consecutive forms: if the **AutoTrader** root form is submitted with values “Ford”, “10520” and “Used Cars” assigned to the “make”, “address” and “search_type” fields respectively then the child form will contain the following three hidden fields: “make”, “address” and “search_type”. The specified values of these fields are “FORD”¹⁵, “10520” and “used”.

5.6 Implementation

We have created a prototype system that allows the user to formulate DEQUEL-queries on the web forms that can be extracted from the Web, and extract and store useful data from the result pages. The implementation was conducted on a SUN workstation working under Solaris 2.7 operational system using Perl version 5.005_2 and employing MySQL (version 3.23.49) DBMS as the data storage. We also used ActiveState Perl 5.6.1 on a Pentium III workstation under Windows 2000 OS for our experiments related to HTML parsing. A web-based graphical user interface (GUI) was implemented using CGI programs written on Perl under control of the Apache Web Server (version 1.3.22). In this section, we present the architecture of the prototype system and summarize the significant results from our experiments.

¹⁵Note that invisible values are assigned to hidden fields.

5.6.1 System Architecture

As shown in Figure 5.17, our prototype system consists of the following components: *User Interface*, *Web Document Loader*, *HTML Parser*, *Query Processor*, *Extraction Module*, and *Storage/Retrieval Manager*.

The user interface shown in Figure 5.6 allows the user to specify URLs or local filenames of web pages containing one or more web forms (Form Storing UI). It calls the Web Document Loader to fetch a web page of some specified URL from the Web. The returned web page is parsed by the HTML Parser that constructs a logical tree representation of the downloaded web page, based on the Document Object Model (DOM) [12], and passes it to the Extraction Module. We use the Perl collection of modules *HTML-Parser* (version 3.26) and *HTML-Tree* (version 3.11) to parse HTML documents, to create HTML syntax trees and extract information from them.

The Extraction Module consists of five submodules: Form Element Extractor, Label Extractor, JS Function Extractor, *Response Navigator*, and *Result Match Extractor*. The first three submodules discussed in Section 5.3.6 are responsible for extracting HTML forms from a given web page. The form element extractor retrieves form data on the basis of the HTML syntax related to forms. The label extractor and JS (JavaScript) function extractor extract additional form data (form labels and related client-side scripts). The extracted data is stored in the Form Database. The last two submodules of DEQUE's Extraction Module: response navigator and result match extractor are responsible for result extraction. The response navigator retrieves all web pages linked to the returned page. If a result page contains an HTML form, the form is extracted by the form extraction submodules. According to the DEQUEL syntax, all forms to be submitted to obtain a result page should be specified in a query. Thus, we presume that each data set submitted by the HTTP request module¹⁶ generates a web page with results. However, web forms that may be contained in result pages are extracted by the extraction module and stored in the form database.

The response navigator analyzes an HTML tree for the following hyperlinks:

- A label of the hyperlink contains words such as “1”, “2”, “next” and so on.
- The URLs of these hyperlinks are very similar. For most cases, these URLs overlap the URL of the request.

The resultant URLs are given to the web document loader. The latter returns the result pages linked to the first returned page. Finally, all result

¹⁶Except when the request is constructed for the values' validation.

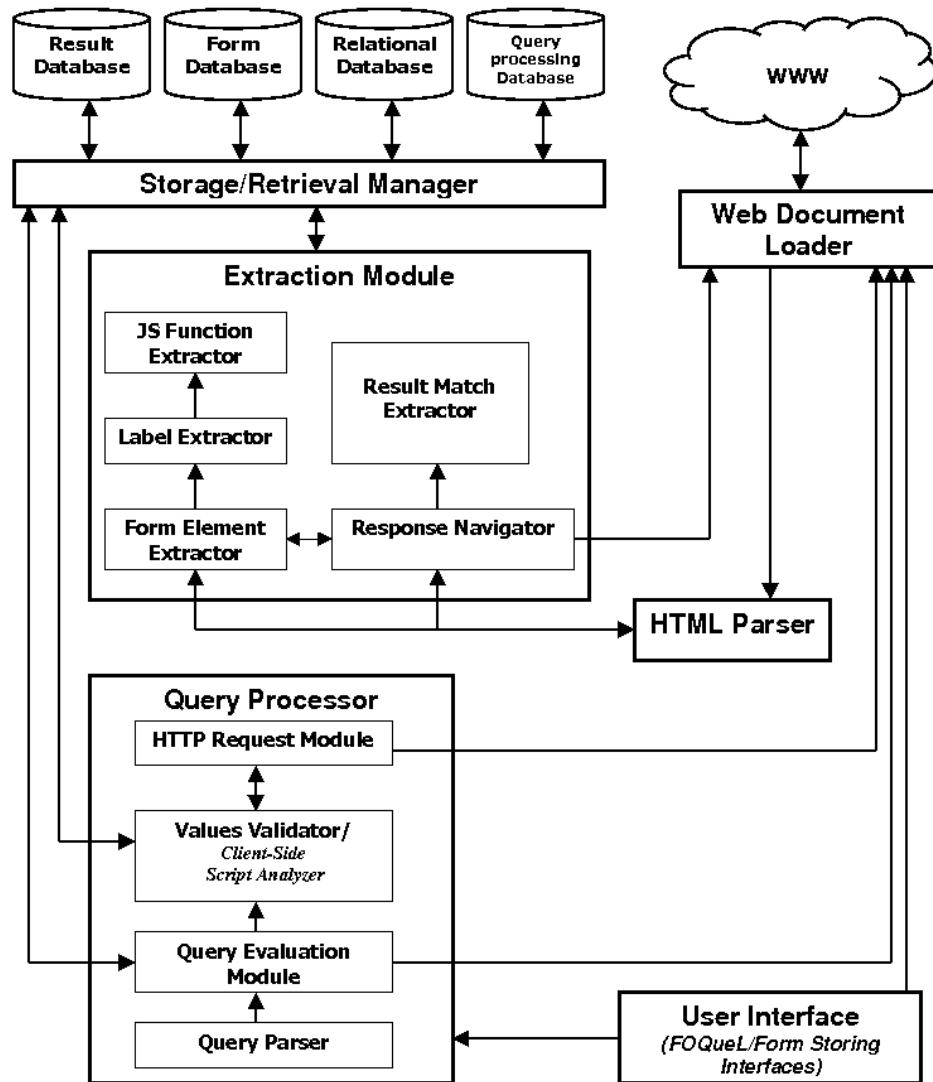


Figure 5.17: Architecture of DEQUE.

pages are passed to the result match extractor that extracts result matches from result pages and builds a result table using technique described in Section 5.4.4.

The DEQUEL user interface (see Figure 5.18) allows the user to specify queries on web forms using the DEQUEL. The formulated query is passed to the Query Processor (described in Section 5.5.4) consisting of the following components: Query Parser, Query Evaluation Module, *Values Validator*, and HTTP Request Module. The query parser parses the query and determines the steps of query execution to be given to the query evaluation module. The latter is responsible for deriving different sets of input values from the *Relational Database*, *Result Database*, and Form Database. The result database stores results of form queries. The query evaluation method also calls the web document loader if some form specified in the query is not pre-stored. The goal of the query evaluation module is to prepare submission data sets and pass them to the values validator for checking. The values validator checks the submission data set. The client-side script analyzer as a part of the validation component of the query processor allows the values validator to more accurately restrict the submission data set on the basis of the client-side scripts. The information about successful submission data set is stored in the *Query Processing Database* and given to the HTTP request module. This module simply submits the different form requests to the remote server-side program. Returned web pages are parsed by the HTML Parser and forwarded to the Extraction Module described above.

5.6.2 Form and Result Storage

The prototype form and result databases are implemented using the open source relational database management system MySQL. The form database consists of five tables: *form*, *field*, *value*, *label*, and *event* (shown in Table 5.1, where table's unique identifiers are in bold). The description of most attributes is given in Table 5.2. Each record of the *form* table describes a form. A super form (for instance, those presented in Section 5.4) is also described by one record in the *form* table. Each record of the *field* describes a form field pertaining to a some form. If a form field belongs to a descendant form, the information about the parent form submission (i.e., parent form identifier, submission data set, date and time of submission) is linked to the corresponding record. Each record of the *value* describes a value (visible and invisible) of a some form field. A record in the *label* table describes a label and domain of values related to a label. A record in the *event* stores a JavaScript function related to some form or form field event.

The Extraction Module stores query results. If a result table has been built and the result table attributes have been specified in a query, then the result of a form query is represented as the table. Otherwise, the query

Figure 5.18: DEQUEL user interface.

Table 5.1: Form Database Schema.

Table Name	Table Attributes
form	fid, faction, fenctype, fmethod, rfid, pfid, rfurl, fname, fevent_id, fdesc, fdt
field	ffid, form_id, ffname, fftype, session_id, ffevent_id, fflabel
value	field_id, visval, invisval, vselected, valtext, vlength
label	lid, lstr, lvalues
event	eid, estr, ecode

Table 5.2: Attributes of Form Database.

Attribute Name	Description
faction	URL of a server-side program handling a form
rfid, pfid	identifiers of a root and a parent form correspondingly
rfurl	URL of a page containing a form (<i>null</i> for descendant forms)
fevent_id	identifier of an event code for a form event
fdt	date and time of the form extraction
form_id	identifier of a form that contains this field
fftype	a type of a form field
session_id	identifier of a session (for form fields pertaining to descendant forms)
ffevent_id	identifier of an event code for a form field event
fflabel	a text string containing descriptive information about a field
field_id	identifier of a form field
visval, invisval	a visible and an invisible value correspondingly
lstr	a text string containing descriptive information (that is, label itself)
ecode	a code of a JavaScript function

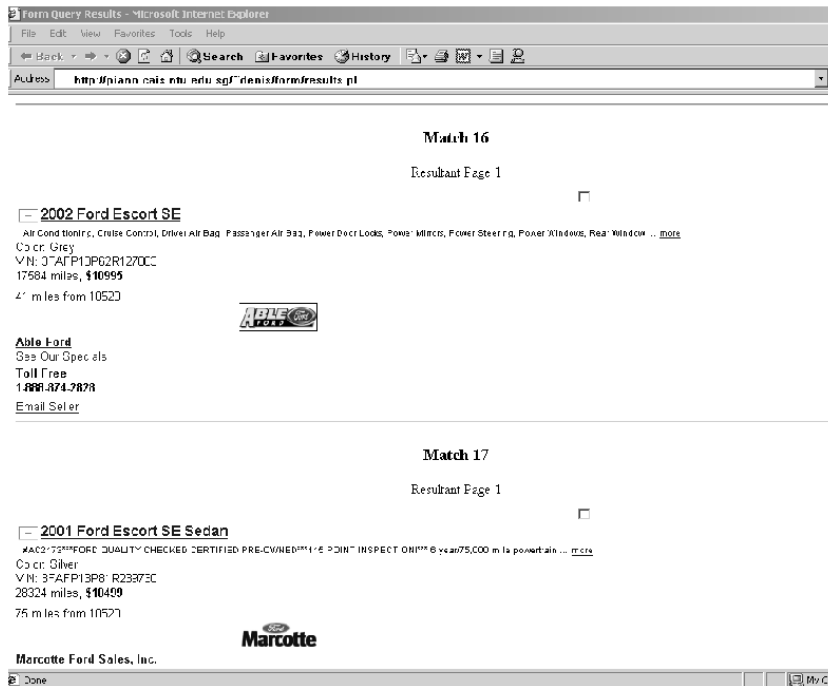


Figure 5.19: Example of query results.

result is a single page containing all result matches. Figure 5.19 shows an example of the results returned by the system after performing the following query on the AutoTrader forms: *Find used Ford Escort cars newer than 4 years within 10520*. Currently, extracted result matches are stored in the result database that consists of two tables: *resmatch* (describing a result match) and *rtname* (describing an attribute of some result table). Thus, more powerful extraction tool can easily analyze the results of form queries stored as pieces of HTML code (each related to one result match) to extract data of interest.

5.6.3 Experimental Results

We have performed queries on about 29 single and 8 consecutive forms. These forms are online interfaces to well-known large databases in different domains such as auto, shopping, science, web search, etc. We selected them for our experiments since the querying automation of such forms is highly desirable for a web user. Totally, the prototype stored 66 different forms (that is, the *form* table of the form database contains 66 records), including 36 root forms and 16 super forms. All forms having the same submission information, location and form type are represented by one “form” record

Table 5.3: Queried forms.

Form Name	URL	No of visible fields	No of invisible fields	Type	No of queries
AutoTrader	http://autotrader.com/findacar/index.jtmpl?ac_afflt=none	4	2	consecutive	240
ZIPFind	http://zipfind.net	3	-	single	40
Amadeus	http://www.amadeus.net/home/en/home_en.htm	9	15	consecutive	100
PubMed	http://www.ncbi.nlm.nih.gov/pubmed/	4	4	single	200
Google	http://google.com	3	3	single	170
ClassicCar	http://classiccar.carfrenzy.com/autos/index2.html	2	-	consecutive	110
Amazon	http://amazon.com	3	-	single	30
CiteSeer	http://citeseer.nj.nec.com/cs	3	1	single	60
Lycos Companies Online	http://business.lycos.com/companyresearch/crtop.asp	3	-	single	40
Powell's Books	http://www.powells.com/search/DTSearch/search.cgi	17	-	single	20
AA Flight Search	http://www.aa.com	13	20	consecutive	70
Phuket Hotel Guide	http://www.phuket-hotels.com/indexprices.htm	53	1	single	10
Froogle	http://froogle.google.com/	2	-	single	50
Yahoo!Autos	http://autos.yahoo.com/	3	5	consecutive	100
Carsearch.com	http://www.carsearch.com/	3	1	consecutive	100
Mobile.de	http://www.mobile.de/	40	1	single	110
Yahoo! RealEstate	http://realestate.yahoo.com/	7	7	single	80
SmartBargains	http://www.smartbargains.com/	3	2	single	100

in the form database. Several forms are not interesting as they only sort the results returned by the other forms. Table 5.3 shows some of the form URL, the form type, the number of visible and invisible fields, and the number of performed queries for several forms¹⁷ that were used to test our technique.

We have been successful with submission of 58 forms. That is, the system was able to process issued query in such way that the pages containing query results or pages containing child forms were generated. The automatically submission of the remaining forms mainly failed due to the number of built-in JavaScript functions or HTML frames.

The advantage of our query system depends on the type (single or consecutive) of a form to be queried. The system is faster than manual filling-out if at least two data sets are submitted with the consecutive forms and more than three data sets with a single form. This is based on the following evaluation. Our system provides the web-based interface (in other words, it is also a web form) to formulate form query. To submit one data set with a single form the user needs to interact with approximately x fields. The

¹⁷These forms are search interfaces to Web data in various domains.

Table 5.4: Label extraction results.

Form name	No of labels for text fields	No of extracted labels	No of labels for select fields	No of extracted labels	No of labels for radio and checkbox fields	No of extracted labels
AutoTrader	1	1	1	1	1	1
ZIPFind	1	1	-	-	-	-
Amadeus	2	2	4	4	-	-
PubMed	1	1	1	1	-	-
ClassicCar	-	-	1	1	-	-
Lycos Companies Online	1	0	-	-	2	0
Powell's Books	6	6	6	6	3	0
AA Flight Search	2	0	6	4	2	1
Phuket Hotel Guide	-	-	-	-	53	44
Mobile.de	2	2	17	11	20	15
Yahoo! RealEstate	1	1	4	4	1	1
Total	30	24	48	37	82	62

DEQUEL interface has the similar requirements. However, while submission of n data sets using a single form requires about $n * x$ interactions. With our system the user specifies n sets at about $x + n * 2$ interactions, where two is an empirical number that shows how many different values are in submitted data sets. Note that $x \geq 2$ as for any form the user must specify at least one value and press a button. After the comparison of the number of interactions we are able to obtain the above estimation for a single form. Similarly, we believe that submission of more than two data sets with consecutive forms (say, a parent and a child form) is faster using the DEQUEL interface.

Table 5.4 contains the results of the label extraction for different forms. In total, for specified forms, we observed that the implemented label extraction technique was able to achieve 80% and 77% accuracy in extracting labels for text and select fields correspondingly, and about 75% in the label extraction for radio and checkboxes fields.

The result pages connected by chain links were correctly retrieved with 89% accuracy. For the most part, chain links of the “unsuccessful” result pages are images ignored by the extraction module.

At the same time, the result extraction technique requires the further development. The main disadvantage is that each result page is analyzed apart from other connected pages. Thus, the system is sometimes unsuccessful in extracting results from pages containing a few result matches. For example, if query returns 53 results laid out on three pages then the extraction from the third page (containing three result matches) is sometimes unsuccessful. However, the extraction module successfully extracts all matches from all

Table 5.5: Result match extraction.

Form Name	No of queries	No of query results	No of extracted matches	Accuracy, %	Precision, %
AutoTrader	240	12340	15570	95	75
ZIPFind	40	1020	980	96	96
Amadeus	100	1200	1440	100	83
PubMed	200	1640	1390	97	82
Google	170	1700	2240	100	76
ClassicCar	110	790	610	76	98
Amazon	30	470	730	99	64
CiteSeer	60	870	710	80	98
Froogle	50	1000	1280	100	78
Yahoo!Autos	100	6920	6310	85	93
Carsearch.com	100	4350	5010	94	82
Mobile.de	110	4860	4030	79	95
Yahoo! RealEstate	80	960	880	80	87
SmartBargains	100	1540	1960	91	71

connected result pages that contain more than 10-20 result matches. Additionally, for many analyzed result pages the result match extractor returns more matches than really presented on the page as some irrelevant sub-trees are considered as result matches' sub-trees.

Table 5.5 shows the *accuracy* and *precision* of result matches' extraction for different web forms. The accuracy is defined as follows:

$$Accuracy = \frac{Number\ of\ relevant\ matches}{Number\ of\ query\ results} \times 100\%, \quad (5.1)$$

where the number of query results is the total number of actual results returned by the server-side program, and the number of relevant matches is the number of extracted matches corresponding to actual results. The precision is given by the following ratio:

$$Precision = \frac{Number\ of\ relevant\ matches}{Number\ of\ extracted\ matches} \times 100\%, \quad (5.2)$$

where the number of extracted matches is the total number of matches extracted by the extraction module from the returned pages. For example, the accuracy 80% means that system extracts only 80% of all matches that really presented on the returned pages. Additionally, the precision 70% shows that 30% of result matches extracted by the system do not contain any useful data. In the final analysis, our experiments show that automatic form querying is feasible, and that relatively few forms are queried incorrectly.

5.7 Conclusion

This chapter describes our approach to query the deep Web. In particular, we have proposed a data model for representing and storing HTML forms,

and a web form query language for retrieving data from the deep Web and storing them in the format convenient for additional processing. We presented a novel approach in modeling of consecutive forms and introduced the concept of the super form. The proposed web form query language DEQUEL is able to query forms (single and consecutive) with input values from relations as well as from result pages (results of querying web forms). Finally, we have implemented a prototype system based on the discussed methodologies.

In the process of DEQUE’s design and implementation we have addressed several challenges in building a query system for the deep Web. First, we introduced the concept of super form to simplify the process of querying and storing consecutive forms. Secondly, we introduced the DEQUEL to provide more convenient and efficient way to fill out web forms. Furthermore, we described our approach to extraction of query results from result web pages.

During our work on DEQUE, we noticed several interesting issues to be addressed in future work: (1) *Support of client-side scripts*: Currently, we only store form or field events and related functions’ codes in our form database. The next step is to perform queries on web forms considering the client-side scripts. Then, we can validate values to be submitted with a form using client-side functions. The support of built-in functions is also very desirable to identify the dependencies between form fields (in many current web forms, the chosen value of some field triggers the function that specifies domain values of another field). (2) *Dependencies between consecutive forms*: The modeling of dependencies between forms can significantly improve the performance of queries on consecutive forms as it helps to remove more irrelevant submission data sets before submitting them to a server-side program. We would like to investigate further on this issue. (3) *Understanding semantics of search interface*: “Understanding” query interfaces (i.e., extracting semantics of forms to be queried) can significantly improve automatic form filling. Particularly, DEQUE does not combine form-related HTML elements into groups according to their semantic role in a form. For instance, the field label “Author” in a form in Figure 1.3 should be associated with four fields (one text field and three radio buttons) while DEQUE assigns “Author” to the text field only. Zhang et al. [116] addressed the problem of extracting form semantics as well as proposed a promising technique to derive query conditions for a given search interface. (4) *Query translation*: There is a huge potential for DEQUE to develop a query translator as a system component. The goal of query translator is to translate a query already issued on some form to another form. For example, a specific query formulated in DEQUEL on Amazon book search form can be reformulated in a such way that it can be issued on the Barnes&Noble¹⁸ book search form. This is, in fact, a very

¹⁸<http://bn.com>

challenging task and largely unexplored by existing works. An interesting approach to this problem was presented in [117]. (5) *Junction of forms and results*: At present, DEQUE strictly separates forms and results. For example, there should be more than one query issued to collect data from a page also containing a child form and a page with final results generated after submission of a child form. The more flexible way of dealing with such cases should be provided in future. (6) *Data extraction*: We intend to investigate more robust algorithms to extract data from the result pages and storing them.

Chapter 6

Conclusions

During this thesis, we studied the deep Web, a large portion of the Web accessible via search interfaces, at three different levels. Our main objectives were to design and develop: an efficient system to retrieve data from the deep Web and an intelligent tool to automatically find and classify web databases. Additionally, our characterization efforts gave an insight into the structure of the deep Web.

The next section summarizes our contributions. Section 6.2 describes some guidelines for future work, and Sections 6.3 and 6.4 discusses future trends and concludes this thesis.

6.1 Summary of Our Contributions

In this thesis, we considered three classes of problems for the deep Web. This allowed us to better understand the topic and address its challenges.

We started by describing the deep Web in the context of availability information on the Web and summarized relevant literature on the topic. Literature review argued that problems of accessing web databases can be solved by combined efforts of database and information retrieval communities.

Characterization of the deep Web was the next phase of our efforts. We described the existing methodologies and clearly demonstrated their drawbacks, particularly ignoring the virtual hosting factor. We proposed two better methods and applied them to explore one national segment of the deep Web. Two consecutively conducted surveys allowed us to not only estimate the main parameters of national deep Web but also measure quantitatively the growth of deep Web.

We then focused on automatic finding of search interfaces to web databases. We described a technique for building an efficient search form identifier and proposed the architecture of I-Crawler, a system for finding and classi-

fying search interfaces. Specifically, the I-Crawler is intentionally designed to be used in deep Web characterization studies and for constructing directories of deep web resources. Unlike almost all other approaches existing so far, the I-Crawler is able to recognize and analyze JavaScript-rich and non-HTML searchable forms.

Lastly, we addressed the problem of querying web databases. We proposed a data model for representing and storing search interfaces, and a web form query language for retrieving data from deep web resources and storing them in the format convenient for additional processing. We presented a novel approach in modeling of consecutive forms and introduced the concept of the super form. The proposed web form query language DEQUEL is able to query forms (single and consecutive) with input values from relational tables as well as from the pages with results of querying other web forms. We described and implemented a prototype system based on the discussed methodologies.

6.2 Future Work

As most research works, this thesis opens more problem than it solves. We consider the following avenues as the most promising for the future work:

- **Support of client-side scripts:** Almost all works devoted to the deep Web have not been paying any attention to non-HTML code contained in many today's web pages. Such non-HTML code (mainly JavaScript) provides "client-side" dynamism of a page and is able to manipulate a page's content in the same way as "server-side" scripts on web servers do. The support of client-side scripts is urgent since most pages on the Web nowadays are extensively using client-side code.
- **Continuation of characterization efforts:** The methods described in Chapter 3 can be applied to the characterization of the entire deep Web. We still know little about the structure of the deep Web, particularly what specific set of features can be used to distinguish deep web sites from regular sites. Such knowledge is in high demand for many applications.
- **Building a relatively complete directory of deep web resources:** This can be considered as a final objective of our efforts in automatic finding and classifying search interfaces. We are going to use the approach described in Chapter 4 and build a form crawler that trawls the entire Web or some domain-specific segment of the Web, automatically detects search interfaces and classifies found interfaces into a subject hierarchy. Reliable classification of web databases into subject hierarchies will be the focus of our future work. One of the

main challenges here is a lack of datasets that are large enough for multi-classification purposes.

- **Construction and dissemination of datasets:** More datasets describing search interfaces to web databases should be available. The datasets simplify significantly the process of testing new methods beginning with constructing useful ontologies for querying web databases and ending with classification of search interfaces into multiple domains.

6.3 Future Trends

Typically, search interface and results of search are located on two different web pages. However, due to advances of web technologies, a web server may send back only results (in a specific format) rather than a whole generated page with results [48]. In this case, a page with results is generated on the client-side: specifically, a client-side script embedded within a page with search form receives the data from a web server and modifies the page content by inserting the received data. Ever-growing use of client-side scripts will pose a technical challenge [17] since such scripts can modify or constrain the behavior of a search interface and, moreover, can be responsible for generation of result pages. Another technical challenge is dealing with non-HTML search interfaces such as interfaces implemented as Java applets or in Flash [2]. In almost all approaches to the deep Web existing so far, non-HTML forms (as well as client-side scripts) are simply ignored. However, there is an urgent need in analyzing non-HTML forms since such forms are going to reach a sizeable proportion of searchable forms on the Web.

One of the general challenges of the deep Web is a simple fact that web search forms are designed for human beings or, in other words, they are user-friendly interfaces to web databases rather than program-friendly. In general, it is quite unrealistic to expect that a computer application overcomes a web user in accessing a web database through user-friendly interface. Fortunately, web databases have recently begin to provide APIs (i.e., program-friendly interfaces) to their content. However, one can anticipate merely a gradual migration towards access web databases via APIs and, thus, at least for recent years, many web databases will still be accessible only through search forms. Anyhow, the prevalence of APIs will presumably solve most of the problems related to the querying of web databases but, at the same time, will make the problem of integrating data from various deep web resources a very active area of research.

6.4 Conclusion

The existence and continued growth of the deep Web creates a major challenge for web search engines, which are attempting to make all content on the Web easily accessible by all users [76]. Though much research work has emerged in recent years on querying web databases, there is still a great deal of work to be done. The deep Web will require more effective access techniques since the traditional crawl-and-index techniques, which have been quite successful for unstructured web pages in the publicly indexable Web, may not be appropriate for mostly structured data in the deep Web. Thus, a new field of research combining methods and research efforts of data management and information retrieval communities may be created.

Bibliography

- [1] Adobe ColdFusion. <http://www.adobe.com/products/coldfusion>.
- [2] Adobe Flash. <http://www.adobe.com/products/flash>.
- [3] BrightPlanet's deep Web directory. <http://completeplanet.com>.
- [4] dig, a DNS query tool for Windows and replacement for nslookup. <http://members.shaw.ca/nicholas.fong/dig/>.
- [5] Java Applet. <http://java.sun.com/applets>.
- [6] MaxMind GeoIP Country[®] Database. <http://www.maxmind.com/app/country>.
- [7] PHP: Hypertext Preprocessor. <http://www.php.net>.
- [8] PubMed biomedical database. PubMed is a service of the U.S. National Library of Medicine that includes over 17 million citations from MEDLINE and other life science journals for biomedical articles back to the 1950s. Available at <http://www.ncbi.nlm.nih.gov/pubmed/>.
- [9] Robots exclusion protocol. <http://www.robotstxt.org/wc/exclusion.html>.
- [10] Yahoo! Directory. <http://dir.yahoo.com>.
- [11] WWW access to relational databases. <http://www.w3.org/History/1994/WWW/RDBGate/>, 1994.
- [12] Document Object Model (DOM) Level 2 HTML Specification Version 1.0 - W3C Candidate Recommendation, June 2002. <http://www.w3.org/TR/DOM-Level-2-HTML/>.
- [13] The UIUC web integration repository. <http://metaquerier.cs.uiuc.edu/repository>, 2003.
- [14] April 2004 Web Server Survey. http://news.netcraft.com/archives/2004/04/01/april_2004_web_server_survey.html, April 2004.

- [15] DNS load balancing report. http://www.securityspace.com/s_survey/data/man.200404/dnsmult.html, April 2004.
- [16] Serge Abiteboul. Querying semi-structured data. In *Proceedings of ICDT'97*, pages 1–18, 1997.
- [17] Manuel Alvarez, Alberto Pan, Juan Raposo, and Angel Vina. Client-side deep web data extraction. In *Proceedings of CEC-EAST'04*, pages 158–161, 2004.
- [18] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *Proceedings of VLDB'97*, 1997.
- [19] Ricardo Baeza-Yates, Carlos Castillo, and Efthimis N. Efthimiadis. Characterization of national Web domains. *ACM Trans. Internet Technol.*, 7(2), 2007.
- [20] Ricardo Baeza-Yates, Carlos Castillo, and Vicente López. Characteristics of the Web of Spain. *Cybermetrics*, 9(1), 2005.
- [21] Luciano Barbosa and Juliana Freire. Searching for hidden-web databases. In *Proceedings of WebDB'05*, pages 1–6, 2005.
- [22] Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden-web entry points. In *Proceedings of the 16th international conference on World Wide Web*, pages 441–450, 2007.
- [23] Luciano Barbosa and Juliana Freire. Combining classifiers to identify online databases. In *Proceedings of WWW'07*, pages 431–440, 2007.
- [24] A. Bergholz and B. Childlovskii. Crawling for domain-specific hidden web resources. *Proceedings of WISE'03*, pages 125–133, 2003.
- [25] Michael Bergman. The deep Web: surfacing hidden value. *Journal of Electronic Publishing*, 7(1), 2001.
- [26] Krishna Bharat and Andrei Broder. A technique for measuring the relative size and overlap of public web search engines. *Comput. Netw. ISDN Syst.*, 30(1-7):379–388, 1998.
- [27] Sergey Brin and Lawrence Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.
- [28] Andrei Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.

- [29] Y. Cao, E.-P. Lim, and W.-K. Ng. On Warehousing Historical Web Information. In *Proceedings of ER'2000*, 2000.
- [30] James Caverlee and Ling Liu. Qa-pagelet: Data preparation techniques for large-scale data analysis of the deep web. *IEEE Transactions on Knowledge and Data Engineering*, 17(9):1247–1262, 2005.
- [31] James Caverlee, Ling Liu, and David Buttler. Probe, cluster, and discover: Focused extraction of QA-Pagelets from the deep web. In *Proceedings of the 20th International Conference on Data Engineering*, pages 103–114, 2004.
- [32] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan-Kaufman, 2002.
- [33] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific Web resource discovery. *Computer Networks*, 31(11–16):1623–1640, 1999.
- [34] Kevin C.-C. Chang, Bin He, Chengkai Li, Mitesh Patel, and Zhen Zhang. Structured databases on the Web: observations and implications. *SIGMOD Rec.*, 33(3):61–70, 2004.
- [35] Kevin Chen-Chuan Chang, Bin He, Chengkai Li, and Zhen Zhang. Structured databases on the Web: Observations and implications. Technical Report UIUCDCS-R-2003-2321, University of Illinois at Urbana-Champaign, February 2003.
- [36] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, 1994.
- [37] Jared Cope, Nick Craswell, and David Hawking. Automated discovery of search interfaces on the web. In *Proceedings of the 14th Australasian database conference*, pages 181–189, 2003.
- [38] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of VLDB '01*, pages 109–118, 2001.
- [39] Hasan Davulcu, Juliana Freire, Michael Kifer, and I. V. Ramakrishnan. A layered architecture for querying dynamic web content. In *Proceedings of SIGMOD'99*, pages 491–502, 1999.

- [40] Michelangelo Diligenti, Frans Coetzee, Steve Lawrence, C. Lee Giles, and Marco Gori. Focused Crawling using Context Graphs. In *Proceedings of VLDB'00*, September 2000.
- [41] Claude Discala, Xavier Benigni, Emmanuel Barillot, and Guy Vaysseix. DBcat: a catalog of 500 biological databases. *Nucl. Acids Res.*, 28(1):8–9, 2000.
- [42] Ahmed Elmagarmid, Panagiotis Ipeirotis, and Vassilios Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.
- [43] Nick Feamster, Jaeyeon Jung, and Hari Balakrishnan. An empirical study of "bogon" route advertisements. *SIGCOMM Comput. Commun. Rev.*, 35(1):63–70, 2005.
- [44] Dennis Fetterly, Mark Manasse, and Marc Najork. Spam, damn spam, and statistics: using statistical analysis to locate spam web pages. In *Proceedings of WebDB'04*, pages 1–6, 2004.
- [45] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, 4th edition, 2001.
- [46] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998.
- [47] Michael Y. Galperin. The molecular biology database collection: 2007 update. *Nucleic Acids Research*, 35(Database-Issue):3–4, 2007.
- [48] Jesse Garrett. Ajax: a new approach to web applications. 2005. <http://www.adaptivepath.com/publications/essays/archives/000385.php>.
- [49] Daniel Gomes and Mário J. Silva. Characterizing a national community web. *ACM Trans. Internet Technol.*, 5(3):508–531, 2005.
- [50] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. In *Proceedings of PODS'02*, pages 17–28, 2002.
- [51] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. GLOSS: Text-Source Discovery over the Internet. *ACM Transactions on Database Systems*, 24(2):229–264, 1999.
- [52] Luis Gravano, Panagiotis Ipeirotis, and Mehran Sahami. Qprober: A system for automatic classification of hidden-web databases. *ACM Trans. Inf. Syst.*, 21(1):1–41, 2003.

- [53] Noah Green, Panagiotis Ipeirotis, and Luis Gravano. SDLIP + STARTS = SDARTS a protocol and toolkit for metasearching. In *Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries*, pages 207–214, 2001.
- [54] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O’Reilly, 1996.
- [55] Alon Halevy. Why your data won’t mix. *Queue*, 3(8):50–58, 2005.
- [56] Joachim Hammer, Hector Garcia-Molina, Junghoo Cho, Arturo Crespo, and Rohan Aranha. Extracting Semistructured Information from the Web. In *Proceedings of the Workshop on Management of Semistructured Data*, 1997.
- [57] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. Accessing the deep Web. *Commun. ACM*, 50(5):94–101, 2007.
- [58] Bin He, Tao Tao, and Kevin Chen-Chuan Chang. Organizing structured web sources by query schemas: a clustering approach. In *Proceedings of CIKM’04*, pages 22–31, 2004.
- [59] Allan Heydon and Marc Najork. Mercator: a scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.
- [60] Jason Hunter. *Java Servlet Programming*. O’Reilly Media, 2nd edition, 2001.
- [61] Panagiotis Ipeirotis, Luis Gravano, and Mehran Sahami. Automatic Classification of Text Databases through Query Probing. In *Proceedings of WebDB’2000*, 2000.
- [62] Panagiotis Ipeirotis, Luis Gravano, and Mehran Sahami. Probe, Count and Classify: Categorizing Hidden-Web Databases. *ACM SIGMOD*, 2001.
- [63] Bernard J. Jansen, Danielle L. Booth, and Amanda Spink. Determining the user intent of web search engine queries. In *Proceedings of WWW’07*, pages 1149–1150, 2007.
- [64] Bernard J. Jansen, Danielle L. Booth, and Amanda Spink. Determining the informational, navigational, and transactional intent of web queries. *Inf. Process. Manage.*, 44(3):1251–1266, 2008.
- [65] Oliver Kaljuvee, Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Efficient web form entry on PDAs. In *Proceedings of WWW’01*, pages 663–672, 2001.

- [66] Eric Dean Katz, Michelle Butler, and Robert McGrath. A scalable http server: the ncsa prototype. *Comput. Netw. ISDN Syst.*, 27(2):155–164, 1994.
- [67] Michael Kifer. Deductive and Object Data Languages: A Quest for Integration. In *Deductive and Object-Oriented Databases*, pages 187–212, 1995.
- [68] David Konopnicki and Oded Shmueli. W3QS: A Query System for the World Wide Web. In *Proceedings of VLDB'95*, pages 54–65, 1995.
- [69] David Konopnicki and Oded Shmueli. Information Gathering in the World-Wide Web: The W3QL Query Language and the W3QS System. *ACM Transactions on Database Systems*, 23(4):369–410, 1998.
- [70] Juliano Palmieri Lage, Altigran S. da Silva, Paulo B. Golgher, and Alberto H. F. Laender. Automatic generation of agents for collecting hidden web pages for data extraction. *Data Knowl. Eng.*, 49(2):177–196, 2004.
- [71] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, 1998.
- [72] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Intelligence*, 11(1):32–39, 2000.
- [73] Yunyao Li, Rajasekar Krishnamurthy, Shivakumar Vaithyanathan, and H. V. Jagadish. Getting work done on the Web: Supporting transactional queries. In *Proceedings of SIGIR '06*, pages 557–564, 2006.
- [74] Ee-Peng Lim, Chaohui Li, and Wee-Keong Ng. Querying Form Objects on the World Wide Web. Center for Advances Information Systems, School of Computer Engineering, NTU, Singapore, 2001.
- [75] Yiyao Lu, Hai He, Qian Peng, Weiyi Meng, and Clement Yu. Clustering e-commerce search engines based on their search interface pages using wise-cluster. *Data Knowl. Eng.*, 59(2):231–246, 2006.
- [76] Jayant Madhavan, Alon Y. Halevy, Shirley Cohen, Xin Luna Dong, Shawn R. Jeffery, David Ko, and Cong Yu. Structured data meets the web: A few observations. *IEEE Data Eng. Bull.*, 29(4):19–26, 2006.
- [77] D. Maier, J.D. Ullman, and M.Y. Vardi. On the Foundations of the Universal Relation Model. *ACM Transactions on Database Systems*, 9(2):283–308, 1984.

- [78] Andrew McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Building Domain-specific Search Engines with Machine Learning Techniques. In *Proc. AAAI-99 Spring Symposium on Intelligent Agents in Cyberspace*, 1999.
- [79] Andrew Kachites McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. <http://www.cs.cmu.edu/~mccallum/bow>, 1996.
- [80] Giansalvatore Mecca and Paolo Atzeni. Cut and Paste. *Journal of Computer and System Sciences*, 58(3):453–482, 1999.
- [81] Giansalvatore Mecca, Paolo Atzeni, Alessandro Masci, Paolo Meri-
aldo, and Giuseppe Sindoni. The ARANEUS Web-base Management
System. In *Proceedings of the International Conference on Manage-
ment of Data*, pages 544–546, 1998.
- [82] Filippo Menczer, Gautam Pant, and Padmini Srinivasan. Topical web
crawlers: Evaluating adaptive algorithms. *ACM Trans. Interet Tech-
nol.*, 4(4):378–419, 2004.
- [83] Weiyi Meng, King-Lup Liu, Clement T. Yu, Xiaodong Wang, Yuhsi
Chang, and Naphtali Rishe. Determining text databases to search in
the internet. In *Proceedings of VLDB’98*, pages 14–25, 1998.
- [84] Scott Mitchell. *Designing Active Server Pages*. O’Reilly Media, 2000.
- [85] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [86] W.-K. Ng, E.-P. Lim, S. S. Bhowmick, and S. K. Madria. Web Ware-
housing: Design and Issues. In *International Workshop on Data Ware-
housing and Data Mining (DWDM’98)*, November 1998.
- [87] W.-K. Ng, E.-P. Lim, C.-T. Huang, S. S. Bhowmick, and F.-Q. Qin.
Web Warehousing: An Algebra for Web Information. In *Proceedings
of the 3rd IEEE International, Conference on Advances in Digital Li-
braries*, 1998.
- [88] Edward T. O’Neill, Brian F. Lavoie, and Rick Bennett. Trends in the
evolution of the public Web. *D-Lib Magazine*, 9(4), April 2003.
- [89] Edward T. O’Neill, Patrick D. McClain, and Brian F. Lavoie. A
methodology for sampling the World Wide Web. *Annual Review of
OCLC Research 1997*, 1997.
- [90] Sriram Raghavan and Hector Garcia-Molina. Crawling the Hidden
Web. Technical Report 2000-36, Computer Science Dept., Stanford
University, December 2000.

- [91] Sriram Raghavan and Hector Garcia-Molina. Crawling the Hidden Web. In *Proceedings of VLDB'01*, 2001.
- [92] C.R. Reis and R.P. de Mattos Forte. An overview of the software engineering process and tools in the Mozilla project. In *The Open Source Software Development Workshop*, pages 155–175, 2002.
- [93] Arnaud Sahuguet and Fabien Azavant. Web Ecology: Recycling HTML Pages as XML Documents Using W4F. In *WebDB (Informal Proceedings)*, pages 31–36, 1999.
- [94] Arnaud Sahuguet and Fabien Azavant. WysiWyg Web Wrapper Factory (W4F). In *Proceedings of WWW'99*, 1999.
- [95] S. Sanguanpong, P. Piamsa-nga, S. Keretho, Y. Poovarawan, and S. Warangrit. Measuring and analysis of the Thai World Wide Web. In *Proceedings of the Asia Pacific Advance Network conference*, pages 225–330, 2000.
- [96] Ilya Segalovich, Yuri Zelenkov, and Denis Nagornov. Methods for comparative analysis of modern search systems and Runet size determination. In *Proceedings of RCDL'06*, 2006. In Russian.
- [97] Chris Sherman and Gary Price. *The Invisible Web: Uncovering Information Sources Search Engines Can't See*. Cyberage Books, 2001.
- [98] Denis Shestakov. A prototype system for retrieving dynamic content. In *Proceedings of Net.ObjectDays 2003*, 2003.
- [99] Denis Shestakov. Deep Web: databases on the Web. Entry in *Encyclopedia of Database Technologies and Applications*, 2nd edition, IGI Global, To appear.
- [100] Denis Shestakov, Sourav S. Bhowmick, and Ee-Peng Lim. DEQUE: querying the deep Web. *Data Knowl. Eng.*, 52(3):273–311, 2005.
- [101] Denis Shestakov and Tapio Salakoski. On estimating the scale of national deep Web. In *Proceedings of DEXA'07*, pages 780–789, 2007.
- [102] Denis Shestakov and Tapio Salakoski. Characterization of national deep Web. Technical Report 892, Turku Centre for Computer Science, May 2008.
- [103] Denis Shestakov and Natalia Vorontsova. Characterization of Russian deep Web. In *Proceedings of Yandex Research Contest 2005*, pages 320–341, 2005. In Russian.

- [104] Mike Thelwall. Extracting accurate and complete results from search engines: Case study Windows Live. *J. Am. Soc. Inf. Sci. Technol.*, 59(1):38–50, 2008.
- [105] Steven Thompson. *Sampling*. John Wiley&Sons, New York, 1992.
- [106] Gabriel Tolosa, Fernando Bordignon, Ricardo Baeza-Yates, and Carlos Castillo. Characterization of the Argentinian Web. *Cybermetrics*, 11(1), 2007.
- [107] A. Tutubalin, A. Gagin, and V. Lipka. Black quadrate: Runet in March 2006. Technical report, 2006. In Russian.
- [108] A. Tutubalin, A. Gagin, and V. Lipka. Runet in March 2007: domains, hosting, geographical location. Technical report, 2007. In Russian.
- [109] Jiyang Wang and Fred H. Lochovsky. Data extraction and label assignment for web databases. In *Proceedings of the 12th international conference on World Wide Web*, pages 187–196, 2003.
- [110] Walter Warnick. Problems of searching in web databases. *Science*, 316(5829):1284, June 2007.
- [111] Simon Wistow. Deconstructing flash: Investigations into the SWF file format. Technical report, Imperial College London, 2000. http://www.thegestalt.org/flash/stuff/Writeup/final_project.pdf.
- [112] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2nd edition, 2005.
- [113] Ron Wodaski. Active server pages technology feature overview, 1998. <http://msdn.microsoft.com/library/en-us/dnasp/html/aspfeat.asp>.
- [114] World Wide Web Consortium. HTML 4.01 Specification, December 1999. <http://www.w3c.org/TR/html401/>.
- [115] J. Xu and J. Callan. Effective Retrieval with Distributed Collections. In *Proceedings of SIGIR'98*, pages 112–120, 1998.
- [116] Zhen Zhang, Bin He, and Kevin Chen-Chuan Chang. Understanding Web query interfaces: best-effort parsing with hidden syntax. In *Proceedings of SIGMOD'04*, pages 107–118, 2004.
- [117] Zhen Zhang, Bin He, and Kevin Chen-Chuan Chang. Light-weight domain-based form assistant: querying web databases on the fly. In *Proceedings of VLDB'05*, pages 97–108, 2005.

Appendix A

Deep Web

A.1 User interaction with web database

A user interaction with a web database is depicted schematically in Figure A.1. A web user queries a database via a search interface located on a web page. First, search conditions are specified in a form and then submitted to a web server. Middleware (often called a server-side script) running on a web server processes a user's query by transforming it into a proper format and passing to an underlying database. Second, a server-side script generates a resulting web page by embedding results returned by a database into page template and, finally, a web server sends the generated page with results back to a user. Frequently, results do not fit on one page and, therefore, the page returned to a user contains some sort of navigation through the other pages with results. The resulting pages are often termed data-rich or data-intensive.

A.2 Key terms

Deep Web (also hidden Web): the part of the Web that consists of all web pages accessible via search interfaces.

Deep web site: a web site that provides an access via search interface to one or more back-end web databases.

Non-indexable Web (also invisible Web): the part of the Web that is not indexed by the general-purpose web search engines.

Publicly indexable Web: the part of the Web that is indexed by the major web search engines.

Search interface (also search form): a user-friendly interface located on a web site that allows a user to issue queries to a database.

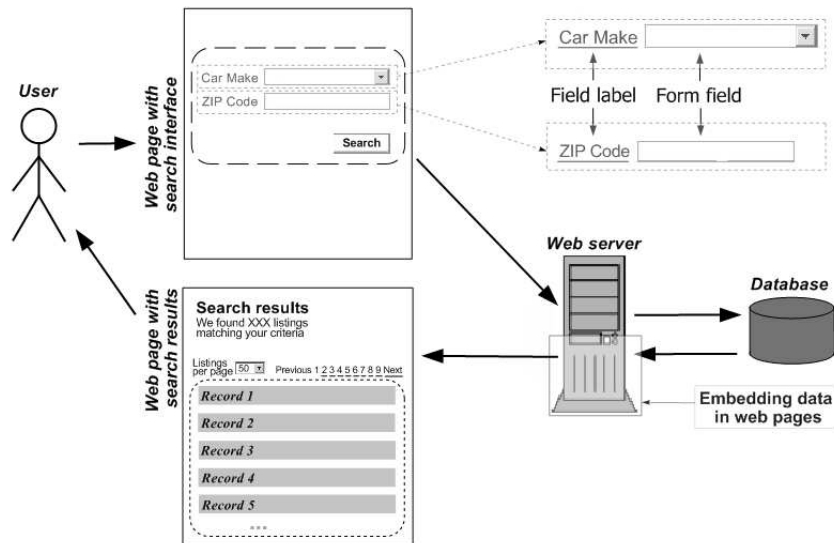


Figure A.1: User interaction with web database.

Web database: a database accessible online via at least one search interface.

Web crawler: an automated program used by search engines to collect web pages to be indexed.

Appendix B

DEQUEL Grammar

B.1 DEFINE Operator

B.1.1 First Type of Syntax

```
<define> ::= DEFINE ATTRIBUTE <default attribute label>  
           <new attribute name> FOR <form label>
```

```
<default attribute label> ::= text<number> | link<number>  
<new attribute name> ::= <value>
```

B.1.2 Second Type of Syntax

```
<define> ::= DEFINE ATTRIBUTE <type> <set of attribute names>  
           CONDITION <condition on text> | <condition on label>  
           FOR <form label>
```

```
<type> ::= TEXT | LINK  
<set of attribute names> ::= <new attribute name> |  
                           <set of attribute names>, <new attribute name>  
<condition on text> ::= ("text" <cond op> <value>)  
<condition on label> ::=  
    ("label" <cond op> <value>, "url" <cond op> <value>) |  
    ("label" <cond op> <value>) |  
    ("url" <cond op> <value>)  
<cond op> ::= "equal" | "contain"
```

B.2 SELECT Operator

```
<query> ::= SELECT [<number of results>] <set of RT attributes>  
              [<set of assigned values>] [AS <query label>]  
              [ FROM <source set> ]  
              [ WHERE <assignment set> ]  
              [ CONDITION <condition set> ]
```

```

<number of results> ::= ALL | FIRST(<number>) | FIRSTP(<number>)

<set of RT attributes> ::= "*" | <RT attribute> |
    <set of RT attributes>, <RT attribute>

<source set> ::= <source> | <source set>, <source>

<source> ::= <form source> | <alternate source>

<form source> ::= <form label> | <url> AS <new form name> |
    (<url>, <number>) AS <new form name>

<url> ::= <value>

<new form name> ::= <value>

<alternate source> ::= <relational table label> | <query result
label>

<RT attribute> ::= <attribute label> |
    <alternate source>.<attribute label>

<assignment set> ::= <assignment clause> |
    <assignment set> AND <assignment clause>

<assignment clause> ::=
    <form label>.<field label> <eq> <predicate> |
    (<set of fields>) <eq> <predicate>

<set of assigned values> ::= <assigned value> |
    <set of assigned values>, <assigned value>

<assigned value> ::= <form label>.<field label>

<set of fields> ::= <form label>.<field label> |
    <set of fields>, <form label>.<field label>

<predicate> ::= <value> | {<set of values>} |
    {<alternate source>.<attribute label>, <card number>} |
    {LABEL, <card number>}

<card number> ::= <number> | <all>

<condition set> ::= <condition on attribute> |
    <condition set> AND <condition on attribute> |
    <condition set> OR <condition on attribute>

```



```

<condition on attribute> ::=
    <RT attribute> <eq> <condition on text> |
    <RT attribute> <eq> <condition on label>

```

```

<all> ::= "all"

```

```

<eq> ::= "="

```

The intuitive meaning of the remaining nonterminals is the following:

<relational table label>	a name of relational table
<query result label>	a reference to stored query results
<attribute label>	an attribute name
<query label>	a new reference to results of this query
<form label>	a form name
<number>	a number
<value>	a text string

Appendix C

Deep Web Characterization

C.1 Detection of Active Web Servers

The task of scanning a list of IP addresses (or hosts) for active web servers can be accomplished in several ways. In this work, the following two free tools were effectually used and, thus, can be recommended by the author:

- WotWeb 1.08 (available at <http://keir.net/wotweb.html>) – a GUI tool specifically designed for finding active web servers. One of WotWeb’s nice features is a provided list of ports typically used by http-servers (besides default port 80 web servers can also be found on such ports as 8080, 8000, 81 and so on).
- Nmap 3.70 and higher (available at <http://insecure.org/nmap>) – a flexible and powerful network scanner with command-line interface. Web servers can be found by running nmap with the following options:
nmap -v -v -sT -P0 -p 80,OtherPorts -iL IPorHostList.file -oG Results.log

C.2 Reverse IP/Host Lookup tools

The following three tools were used for resolving IP addresses to hostnames and back:

- Reverse IP Lookup, Reverse NS Lookup, Whois Tools at <http://domainsdb.net>
- Reverse IP at <http://www.domaintools.com/reverse-ip>
- Advanced WHOIS Lookup! (beta) at <http://whois.webhosting.info>

Table C.1: Relative frequency of domains in the Hostgraph

Domains	*.ru	*.com	*.ua	*.net	others
Percentage(%)	75	7.2	6.8	4.1	6,9

C.3 Hostgraph fraction of zones

The “Hostgraph” is a graph of links among all hosts known to Yandex search engine at the time of February 2005. All outgoing links from different pages on one host to different pages on another host are considered as one link between these hosts.

C.4 RU-hosts

The “RU-hosts” is a list of all second-level domain names in the .RU zone registered at the time of April 26, 2006. It includes 510,655 domain names.

Turku Centre for Computer Science

TUCS Dissertations

72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory
86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web> Querying and Characterizing

TURKU CENTRE *for* COMPUTER SCIENCE

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Information Technologies



Turku School of Economics

- Institute of Information Systems Sciences

ISBN 978-952-12-2098-2

ISSN 1239-1883



Denis Shestakov

Denis Shestakov

Search Interfaces on the Web: Querying and Characterizing

Search Interfaces on the Web: Querying and Characterizing