Characterizing the deep Web

Master’s thesis
UNIVERSITY OF TURKU
Department of Information Technology
Computer Science
December 2011
Jyri Lehtonen
The deep Web is a section of the Web. It contains over 99% of the Web, yet the major search engines cannot access it efficiently. Defining the difficulties of automatically accessing the deep Web is the main research topic of this thesis. Two other thesis statements are also taken into account. First, we create an estimation of the current deep Web scale concluded from the previous scale-related research. Second, we address the problem of querying multiple deep Web resources “on the fly”.

The thesis is written in a pattern of first explaining what is currently considered as part of the deep Web content (i.e. dynamic pages, limited content and databases). This content is the part of the Web that the search engines do not index well, therefore users cannot query a general search engine index to find this data.

After the deep Web content has been defined, the thesis illustrates the data gathering techniques of the Web (i.e. screen scraping and APIs). These techniques are used when one creates a system that collects information from other servers (i.e. a Web crawler).

The final part of this thesis concentrates on how to query a deep Web resource, using theoretical and practical examples. The research is concluded with a project: the Scavenger Crawling System. The system was written for this thesis using the methods and techniques explained throughout this research. The project was a success: The system can query a deep web resource that is not predefined for it.

Keywords: Deep Web, Deep Web Navigation, Deep Web Analysis, Deep Web Crawling, Metasearch Engine, HTML Form Analysis, Web Crawler
ACKNOWLEDGEMENTS

I am grateful for the support of my thesis instructor Teuhola during this nine-month period that I have been working more or less to complete this research. Our meetings were always helpful and he was genuinely interested in my thesis from the beginning. I also wish to thank professor Nevalainen for his support during the initial discussions of the thesis theme and for the thesis seminar.

I would like to express my thanks to the IT-department of Turku, who gave me enough flexibility to attend my remaining examinations making it possible for me to graduate on time.

For the interesting conversations about the subject and general coolness, I would like to thank my childhood friends from Räntämäki for their support and insight to this thesis. Also, thank you Jenna for giving me assistance in the English language.

I would like to thank my parents also for their support through my whole studies.

Last but not least, I would like to note that I am indebted to my beautiful girlfriend for putting up with my six years of studying and working simultaneously. I know time has been scarce from time to time but my graduation will leave more time for our family for sure. We have been both waiting for this moment.

Turku 23.11.2011

Jyri Lehtonen
List of Figures

Table 1: Simple way of understanding surface Web scale ................................................................. 4
Table 2: Search engine results of the Boolean query OR ................................................................. 5
Figure 1: Matching of the top search engine results by Langreiter’s Graph .................................... 6
Figure 2: Results of the search engine overlapping comparison ...................................................... 8
Figure 3: The Classical Model of Information Retrieval from the Web ................................................ 9
Figure 4: High-level architecture of a Web crawler ........................................................................... 10
Figure 5: The size of the indexed World Wide Web ......................................................................... 13
Table 3: Surface Web vs. Deep Web (2000) .................................................................................... 16
Table 4: Surface Web vs. Deep Web (2005) .................................................................................... 16
Figure 6: Surface Web vs. Deep Web (scale 2000-2010) ............................................................... 17
Figure 7: AJAX Example: Select-Option ......................................................................................... 22
Figure 8: Typical login screen ............................................................................................................ 23
Table 5: Example of Robots Exclusion Standard ............................................................................ 23
Figure 9: Unlinked content .................................................................................................................. 25
Figure 10: Single Database Query Process ....................................................................................... 27
Figure 11: Network map with databases ......................................................................................... 29
Figure 12: An Overlap Analysis ......................................................................................................... 32
Figure 13: IP Sampling process ......................................................................................................... 34
Figure 14: The triangle problem in query processing ................................................................. 35
Figure 15: Visualization of input elements ......................................................................................... 36
Figure 16: Static tree of the input elements ....................................................................................... 37
Figure 17: Dynamic tree of the input elements .................................................................................. 37
Figure 18: Automating the Intermediate Page ............................................................................... 39
Table 6: KAPath reference ................................................................................................................ 41
Figure 19: Example of KAPath ........................................................................................................... 42
Figure 20: Source code example from Youtube ............................................................................. 43
Table 7: Screen scraping vs. APIs .................................................................................................... 45
Figure 21: A Web Mashup Application ............................................................................................. 48
Figure 22: Resource gathering techniques ...................................................................................... 50
Figure 23: Example of HTML Form ................................................................................................. 51
Table 8: FIND_FORM_FIELDS -algorithm ................................................................................ 53
Figure 24: The String Algorithm of form finding ........................................................................... 56
Table 9: GET_FORM_FIELD -algorithm ....................................................................................... 57
Table 10: cURL Reference ............................................................................................................... 58
Figure 25: Google HTML link results prefixed by an ascending number ........................................ 60
Figure 26: Translating queries between interfaces ......................................................................... 61
Figure 27: Semantic-then-Syntax framework ................................................................................. 63
Figure 28: Virtual Forms of QI2 ..................................................................................................... 65
Figure 29: Mechanics of Turbo10 metasearch engine ................................................................. 66
Figure 30: Adapter Manager of Turbo10 ....................................................................................... 67
Figure 31: Trawler Server and Web Browser in Turbo10 ............................................................... 68
Figure 32: DeepBot Architecture .................................................................................................... 71
Figure 33: Prototype Crawler Architecture ................................................................................... 72
Figure 34: UML of Class: The Crawling System ........................................................................... 73
Table 11: Crawling System: Attribute Descriptions ...................................................................... 74
Figure 35: UML of Class: The Crawling Application ................................................................... 77
Figure 36: Entity Relationship model of the Index ......................................................................... 81
Figure 37: Relational model of the Index ....................................................................................... 81
Figure 38: Scavenger Engine print flow ......................................................................................... 83
Figure 39: Scavenger statistics: crawling amounts ........................................................................ 84
Figure 40: Scavenger statistics: form field categorization ............................................................ 85
Figure A: The Scavenger Hardware Model. .................................................................................... 92
1 INTRODUCTION

Current search engines cannot index everything the Web has to offer. When we begin to characterize the problem, we are actually defining some of the technical complications of the current search engines. These complications divide the Web into two sections: the surface Web and the deep Web. The surface Web is the section that the search engines can index well while the deep Web is mostly denied from the search engine indexes.

Many questions arose during the deep Web characterization. First, the previous deep Web statistical research was conducted during 2000-2001 and 2004-2005. This leaves a gap of knowledge to the current statistical situation and understanding of the scale of the deep Web. Second, the references of this thesis often pointed out how difficult it is to collect information from the deep Web. This thesis will describe in detail the core problems of deep Web indexing. Last, the problem of collecting information from multiple deep Web sources is addressed and a technique is defined for it in theory and in practice. This thesis will address these research questions with emphasis on the second and third research question.

The characterization begins with giving sufficient background information about search engines and Web crawlers in general (Section 2). The most important subsection of Section 2 discusses the World Wide Web crawlers. In this subsection, the crawling strategies of a Web crawler will be defined and it will be shown that the deep Web is but one of the many possible strategies of a crawler. The scope of this thesis does not include other topics from the general search engine field than the basic introduction and the crawler definition. After these are explained, the main subject of this thesis can be fully explored.

During the deep Web introduction (Section 3), the previous statistical research is revealed. The introduction continues by defining what can be included as deep Web content and explaining the basics of the Web query interfaces. It is not in the scope of this thesis to create accurate estimates of the current statistics of the deep Web.
After the deep Web has been introduced, the problems of navigating it are explained (Section 4). The deep Web navigation can be defined as the techniques that allow Web developers to use Web data. These techniques are mandatory to gather any information from the Web, including the results of an automated deep Web query. First, the automated system must find the form field to query. Second, the found query interface will be used. Third, the data that the queried system returns must be collected using the various techniques described in Section 4.

After the data gathering techniques of the Web have been defined, the automated process of querying single or multiple Web databases is explained in detail (Section 5). It is out of the scope of this thesis to pursue the most efficient way to query multiple Web databases, further than explained in Section 5.3. It is a topic that can be continued in future research. To assemble the many definitions of this thesis to a working entirety, two deep Web search engines and their architectures will be presented at the end of Section 5.

The thesis concludes with a project to create a Web crawler (Section 6). The system is written from scratch for this thesis. After the system is created and the crawling computers are set up, the Web crawler is set to run for a 24-hour period to collect data. Several kinds of statistics are gained from the Web crawler’s run and will be displayed throughout Section 6. The purpose of the project is to create a system that is capable of finding an undefined deep Web resource. When the system met its time limit, another component of the Web crawler began to analyze the found data. Once the deep Web resource (Web database query interface) has been found, a query will be initiated through the interface and the results will be captured. The project was successful and the system queried a server that was not predefined for it.
2 SEARCH ENGINES

From a user perspective, a search engine is a query interface, which allows the user to locate his interests in the World Wide Web (WWW). The user can access the knowledge behind the query interface by using keywords. Most search engines search for Hypertext Markup Language (HTML) documents from the Internet and store them into an index database. Indexing can be defined as collecting data and storing it for effective retrieval (when a user requests it). Search engine indexing is mostly done by World Wide Web Crawlers.

The following sections are meant as a prerequisite to the deep Web. First, we will take a look of what is the area called surface Web (Section 2.1). Second, we will observe how much the search engine query results actually overlap each other (Section 2.2). Last, we will define the general architecture and an alternative architecture for the crawlers (Sections 2.3 and 2.4).

2.1 Defining the surface Web

From a search engine perspective, the Web can be divided into two parts. The surface Web is the smaller, the visible and the indexable part while the deep Web contains much more data, is mostly hidden from the search engines and is not trivial to index. The surface Web contains those Web documents that the search engines can gather for the public and display them as a query result. The technical details have changed during the last decade as some elements, which were listed as deep Web elements, have become surface Web elements. This phenomenon occurs when the search engines are developed further and the technical obstacle to index a deep Web element has been overcome.

The major search engines have been the same for a while now. Since the deep Web survey of Bin He (He, Patel, et al. 2005) during 2004-2005, the largest service providers to this day have been Google, Yahoo and Microsoft. It is through these service providers that one can begin to evaluate the scale of the surface Web. An
estimation of the scale can be achieved through the index size of the service providers using statistical methods. One must remember that there is some overlap in the search engine indexes and that none of the indexes cover the whole surface Web. The overlap and the relative size of the indexes must be estimated first to be able to get an estimate of the whole scale of the surface Web. The result is estimation because the scale and dynamic nature of the Web prevents one from calculating a precise result.

The average search engine user can create a scale related result by simply querying the search engines himself. The following experiment was performed on the 7th of October 2010. The experiment is divided into two parts. First, we query the search engines with a keyword. Second, we insert an overlapping keyword query. The purpose was to determine the differences that the three search engines have with the same keyword(s). Keywords that were chosen for the test are meant to be the most common words in the English language (with one exception). The test will produce the first estimate of the index size of the search engines. Results of the test are shown in Table 1.

The test will not determine much on its own. It is meant to show, in a simple way, the possible scale of the service providers’ indexes. Index size does not determine how accurate the results are or how much value the shown results give to the end-user. If it would be that simple to determine the most effective search engine, there would not be a need to research the subject any further. This kind of thought leads to search engine optimization and analysis, which are not part of this thesis.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Google</th>
<th>Microsoft</th>
<th>Yahoo</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>25,420,000,000</td>
<td>669,000,000</td>
<td>34,200,000,000</td>
</tr>
<tr>
<td>is</td>
<td>22,630,000,000</td>
<td>5,270,000,000</td>
<td>25,600,000,000</td>
</tr>
<tr>
<td>the</td>
<td>25,460,000,000</td>
<td>724,000,000</td>
<td>31,500,000,000</td>
</tr>
<tr>
<td>not</td>
<td>4,790,000,000</td>
<td>4,590,000,000</td>
<td>17,600,000,000</td>
</tr>
<tr>
<td>manufacturing</td>
<td>262,000,000</td>
<td>38,100,000</td>
<td>272,000,000</td>
</tr>
</tbody>
</table>

Table 1: Simple way of understanding surface Web scale
Yahoo had indexed documents of the chosen keywords more than Google or Microsoft had. Microsoft’s service Bing.com is far behind the other two competitors in the amount of shown result documents.

The keywords for the keyword overlap test were chosen to be the combination of the first phase keywords using the Boolean expression OR. The results are shown in Table 2. Rather than Yahoo having the most results, as they did in the first, Google has the most results in this test.

<table>
<thead>
<tr>
<th></th>
<th>Google</th>
<th>Microsoft</th>
<th>Yahoo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25,650,000,000</td>
<td>1,730,000,000</td>
<td>8,270,000,000</td>
</tr>
</tbody>
</table>

Further analyzing this keyword-overlapping test would be more suitable for a thesis related directly to known search engine functions. However, what seems to be out of place is the Boolean expression OR. The search engines allow users to input the expression between keywords. It should retrieve all the unique instances of one of the keywords or the other keywords or both or all of them. If any of the selected keywords are found in an index of a search engine, the result should be shown to the query creator. The more keywords are connected through OR expression the more results it should yield, which is actually seen when inserting the combination into Google.

We can assume that the syntax of the Boolean expressions may differ from one search engine to another or the “not” keyword is read as an expression itself. If we leave the negation out from the query, the result was still the same. Why the two other search engine results are as low as they are shown in the figure above is a noteworthy question but out of the scope of this thesis.
2.2 Search engine overlapping

In the previous section, we did a simple experiment to estimate the scale of the search engine indexes. The results that the simple keywords retrieved from the search engine indexes were all different among the search engines. The Web has quite a number of search engines available. The following question arises: Does a single search engine find all that a user needs or should users be querying multiple search engines or a metasearch engine (that has access to multiple search engines and draws using its own logic the best results from the search engines). It is easy for basic users to settle upon the thought that most of the search engines find just about the same objects and the selection of a search engine is not that relevant. This section will prove this kind of thought wrong.

A third party application was used to determine the answer for the question of multiple search engines. The first test is to calculate how many results from the top 100 rankings of the Google and Yahoo search engines are matching. Since the subject set is so small, we can only draw estimates. However, the top results of a search engine are the most relevant results the search engine has to offer about the keyword(s). Generally, users do not open the additional result pages the search engines offer. Users will instead observe the first result page and most importantly the first few of the shown results. The simple test is conducted by a Langreiter’s Graph. The keyword for the result-matching test was “Peikkoluola”.

![Matching of the top search engine results by Langreiter’s Graph](langreiter 2006)
The graph is actually a bipartite graph. We can say that a graph \( G = (V, E) \) is bipartite if its node set \( V \) can be partitioned into sets \( X (X_1, X_2, \ldots, X_n) \) and \( Y (Y_1, Y_2, \ldots, Y_n) \) in such a way that every edge in the graph has one end in \( X \) and the other end in \( Y \) (Kleinberg and Tardos 2006).

Figure 1 contains the results of the bipartite graph matching created by the third party application. Lines in the graph show a specific match and the nodes that the line connects show the position of the match in the search engine rankings. The results of Figure 1 show that only 17 of 100 documents match between the two selected search engines. After this simple test, let us next discuss the results of a more thorough analysis. The analysis is formally called overlap analysis and will be explained in more detail in Section 3.6 of this thesis.

The metasearch engine Dogpile’s development team has created an overlapping analysis research during 2007. (Dogpile 2007) In the research, nearly twenty thousand queries were ran on Google, Yahoo, Ask and Microsoft’s search engine. To be aligned with the rest of this thesis we will observe the results of Google, Yahoo and Microsoft. The queries returned 566,415 unduplicated results for these three search engines. Of these unduplicated results:

- 1% was shared by all three search engines (5,541)
- 6.2% were shared by two of the three search engines (35,191)
- 92.8% were unique to one of the three search engines (525,683)

Since the surface Web is enormous in size, the search engines cannot possibly crawl all the same pages, as they do not cover the surface Web fully. Every search engine has its own Web crawlers that are designed to fetch Internet objects through rules set for the software. The rules, starting addresses and the depth that a crawler will index documents from a found Web server are different from each other. Therefore, the Web crawlers are bound to take different routes as they travel through the Web for the search engines. Considering all these, it is no wonder that the overlap of the search engines is minimal even among the largest indexes. In Section 2.4, we will discuss Web crawlers
in more detail.

The set theory illustration of Figure 2 shows the relative overlapping of the three search engines discussed in this section. Let A be Google, B be Yahoo and C be Bing. As the search engines are taken into account on their own, A=147712, B=190475 and C=187496 unique hits were detected in the research made by Dogpile’s team. Let us next take a look at the intersections of the search engines from the same research:

\[ A \cap B = 11056, \quad A \cap C = 11447, \quad B \cap C = 12688. \]

Finally, the intersection of all the search engines as the metric for showing how many matches are found among these three search engines given the set of 566,415 unduplicated search results:

\[ A \cap B \cap C = 5541. \]

From these sources covered here, we can draw the estimate that search engine indexes do not overlap each other relevantly (even in the top Web site rankings that the engines generate).

Figure 2: Results of the search engine overlapping comparison
2.3 The classical model for information retrieval from the Web

While using the search engine services the user’s intent might not always be informational (the intent to acquire information). The user might be in need of a navigation assistance (I want to know the URL of the site) or it might be transactional (I want to know where I can shop). It is noted that informational searches constitute less than 50% of search engine searches (Broder 2001).

To understand the process how information is retrieved by search engines, one needs to understand the basics of information retrieval from the Web. The whole process begins with the user’s need. The interface that the user is using generates a query that is sent to the search engine system. The system selects the indexed documents with certain document matching rules designed for the system. After selection, the system shows the results to the user. Should there be a need to refine the query, this can be achieved after the initial results are generated. This information retrieval process is illustrated in Figure 3.

![Figure 3: The Classical Model of Information Retrieval from the Web](image-url)
2.4 World Wide Web crawlers

Search engines use a method called crawling to index the World Wide Web to a database. This means that a program is unleashed to run through the Web. Crawling through the surface Web is mostly made possible by moving from one Web anchor to another and storing that information for retrieval. Anchors are HTML objects that allow the user to move to another point of interest. For example, an anchor tag `<a href="http://www.peikkoluola.net">Peikkoluola</a>` would allow a user to enter the Internet site by clicking on the element. Therefore, the basic idea behind a crawler is a trivial one: Select an anchor from a set of candidates, download the associated Web pages, extract the anchors contained in the pages and add those anchors that have not been encountered before to the candidate set. Let us define a Web crawler as follows:

“A Web crawler is a program that given one or more seed URL’s, downloads the Web pages associated with these URLs, extracts any hyperlinks contained in them and recursively continues to download the Web pages identified by extracted hyperlinks”.

(Najork 2005)
A Web Crawler can be written in numerous programming languages available. The first question is to choose one of the languages - depending on the developers’ preferences. Famous Web crawler languages include C++, C#, Java, PHP and Python. The speed of the Web crawler is important, but since the most restrictive element in the process will be network latency, the choice of the programming language becomes less relevant. However, on more mandatory tasks (such as a business level crawler), a rule of thumb could be not to choose a scripting language for the project.

The high-level Web crawler architecture contains the **crawling system** and the **crawling application** components. The idea is that the crawling system interacts with the servers in the World Wide Web by downloading documents after the crawling application has decided what document to request next from the crawling system, given the previously crawled documents and the current state. Eventually documents get stored in the storage system that the Web crawler uses (generally a database).

*The Web crawling system* (as seen in Figure 4) includes robot exclusion, speed control and DNS (Domain Name System) resolution. First, the robot exclusion can be defined as the rules that the Web servers set to Web crawlers. These rules define how the client server wishes the Web crawler to index the site, for example, the client server files that are wished to be excluded from the indexing. Robot exclusion standard will be defined in more detail in Subsection 3.3.2. Second, a loosely defined crawler can end up harming a Web server. This is why there is a need to focus on the speed control of the system. The Web crawler must avoid pressuring a single server so that the server’s main services are hindered. Finally, the crawler system’s DNS resolution is responsible for initiating queries that lead to full translation of the domain name into an IP (Internet Protocol) address.

*The crawling application* is in charge of deciding what strategy to implement. There are many theories of crawling strategies available for implementation. Next, we will define some of them, which will later be used in the prototype crawler created for this thesis.
• **Breadth-first:** Crawlers start from a certain point (Web site) in the World Wide Web and begin systematically to process all the document’s anchors and continue further with every new anchor. The technique is FIFO queue, meaning that the first element in the queue will be the first taken out from it.

• **Recrawling:** Since the Web is dynamic and documents are constantly updated, the need to recrawl the pages to keep the index up-to-date is high. Recrawling can be done by simply initiating another breadth-first crawl or design a more complex set of rules that favors some sites more than others.

• **Focused:** The Web is full of dedicated search engines for a certain topic. A focused strategy means that the system tries to parse only the documents that are of interest to the search engine. Dedicated search engines generally have higher chance to have most up-to-date documents of its interest.

• **Deep Web:** Accessing the deep Web is one of the possible crawler strategies. The deep Web will be discussed in more detail later in this thesis.

There is, however, more than one architectural way to accomplish crawling. The first to be introduced is the traditional and widely used client-server architecture. Most search engines utilize the client-server architecture. There is a need to create a competitor for the dominating client-server architecture. Therefore, the more theoretical architecture of a distributed crawler will also be introduced.

### 2.4.1 Client-server architecture

In the traditional client-server architecture, the central server manages all the status information (Takahashi, et al. 2003). When all the data is stored to a central server, the amount of needed network resources grows as the size of the stored database and the amount of user activity increase. This leads directly to the fact that only a few companies in the world can manage the amount of information gathered by large-scale Web crawlers. Currently the stakeholders in the field are the three companies already introduced in this thesis: Google, Yahoo and Microsoft. There are quite a number of other search engines around but for some reason they do not match these three in popularity and efficiency. Finding further reasons for this is not in the line with the
subject of this thesis.

Most Web designers have probably noticed one downside of this architecture type. Crawling the Web and storing the indexed data can take a long while to complete because of the size of the surface Web. The size was estimated to be 11.5 billion documents in Gulli’s and Signorini’s research during 2005 (Gulli and Signorini 2005). Using a tool created by Maurice de Kunder, the size estimation of 2011 is at least 14 billion pages. The total estimation is created by estimating the indexes and the index overlap of the stakeholder search engines. Figure 5 shows the two total estimates, one starting with Yahoo (YGB) and one starting with Google (GYB). The difference in the lines of Figure 5 is because Google calculates every page as a unique web page, where Yahoo does not count all pages for the total amount. (Kunder 2011)

A single crawling can take up to a month or more to cover a significant portion of the Web (Takahashi, et al. 2003). Because of this, the service providers cannot give their customers fully up-to-date information through their search engines and even less if a Web page develops or transforms rapidly or the Web page is recently created.

![The size of the indexed World Wide Web](image)

**Figure 5: The size of the indexed World Wide Web**

(Kunder 2011)
2.4.2 Distributed Crawlers

Since the nature of the Web is distributed, a team from the University of Tokyo has pointed out an alternative idea to the traditional client-server architecture. They claim it is natural to crawl the distributed Web with ordinary PCs which themselves are already distributed all over the Web. The team claims it to be more economical if the PC users can provide the data they gather when browsing the Web for search engine indexes. The network traffic that the traditional architecture requires would be reduced significantly with a distributed alternative. (Takahashi, et al. 2003)

There are several technical difficulties that must be overcome to consider this type of architecture as a real alternative to the traditional one. This thesis introduces two of them. The first difficulty to be explained is the scalable management of status data. This means that simply distributing crawlers with the traditional client-server architecture would not improve overall performance at all. The reason for this is that the status data would still be on a central server. Every new resource (mainly every new URL) needs to be consulted with the central server. To overcome this, one must design an architecture that manages the URLs in a scalable manner. The second difficulty is the exploiting of dynamic and unreliable resources. There are resources that may not be available for usage. For example, they may be disconnected from the network (permanently or they can have power management issues). The architecture must tolerate this and continue to distribute work among participating resources in a scalable manner. (Takahashi, et al. 2003)
3 INSIGHT TO THE DEEP WEB

There are alternative names for the deep Web. Some, but not all, include the Hidden Web, the Invisible Web, the Deep Net and the Dark Web. The term “deep Web” was accepted through the writings of Michael K. Bergman in the white paper “Deep Web: Surfacing Hidden Value” (Bergman 2001). In this thesis from this point forward, the subset of the Web that the search engines do not index well or at all is called the deep Web, as an acknowledgement to the generalized terminology.

Through the previous section, it was made clear that the surface Web is a challenge to engineers. As if the surface Web would not be enough, we will introduce the statistics of the deep Web from several years ago (Section 3.1). This will give us an understanding of what we will be dealing with next. After the scale has been processed, the reasons why it is important to concentrate on the deep Web will be revealed (Section 3.2), and the actual content of the deep Web will be defined (Section 3.3). In the rest of the section, we will concentrate on the main content of the deep Web (i.e. the databases). How the database query interfaces are identified (Section 3.4) and what are the general methods of querying them (Section 3.5). The section ends by describing a few tools, which can be used to analyze the deep Web (Section 3.6).

3.1 Deep Web statistics

The scale of the surface Web in 2000 was measured to have approximately 2.5 billion documents at a growing rate of 7.5 million documents per day. The total size was approximated to be nineteen terabytes. The search engine that had indexed most of the surface Web was Google, which had crawled 1.35 billion documents. (Bergman 2001) That is just over half (54%) of the total size that the surface Web was estimated to be.

Let us compare the surface Web statistics to the statistics of the deep Web. The deep Web was measured to have approximately 550 billion documents that is 220 times more than the surface Web. The total amount of data was said to be 7 500 terabytes, which is nearly 400 times more than the surface Web. More than 200 000 deep Web sites were
estimated to exist. (Bergman 2001) In terabytes, the surface Web was only 0.002% of the Web and in document count, the surface Web was 0.004% compared to the size of the deep Web. The deep Web content quality was said to be 1000 – 2000 times better than that of the surface Web. The quality analysis was performed by BrightPlanet technology as part of Bergman’s research (Bergman 2001). This leads to a conclusion that ten years ago, when deep Web indexing was in its infancy, most of the Web was not reachable efficiently. Table 3 includes a summary of the statistics described in this paragraph.

Table 3: Surface Web vs. Deep Web (2000)

<table>
<thead>
<tr>
<th></th>
<th>Surface Web</th>
<th>Deep Web</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documents</td>
<td>2.5 billion</td>
<td>550 billion</td>
</tr>
<tr>
<td>Size</td>
<td>19TB</td>
<td>7500TB</td>
</tr>
</tbody>
</table>

Five years later another estimation of the deep Web scale was calculated. Crawling of the deep Web during 2005 resulted in increased results in deep Web content. A total of 307,000 deep Web sites were estimated to exist as well as 450,000 Web databases that were accessible by 1,258,000 query interfaces. The deep Web had grown 3 – 7 times larger during 2000-2005, as shown in Table 4. (He, Patel, et al. 2005)

Table 4: Surface Web vs. Deep Web (2005)

<table>
<thead>
<tr>
<th></th>
<th>Surface Web</th>
<th>Deep Web</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documents</td>
<td>11.5 billion</td>
<td>1650 – 3850 billion</td>
</tr>
<tr>
<td>Size</td>
<td>60TB – 100TB</td>
<td>22500 – 52500TB</td>
</tr>
</tbody>
</table>

When this thesis was written, the surface Web was estimated to be is at least 14 billion documents (Kunder 2011). With the average size of 25k / HTML document (The
Average Web Page 2008), the surface Web would amount to 465 TB. Should the deep Web still be 400 – 500 times larger than the surface Web, we would get a result of 8 000 – 10 000 billion documents taking up a total of 186 000 TB – 232 500 TB, as illustrated in Figure 6.

![Deep Web Scale](image)

Figure 6: Surface Web vs. Deep Web (scale 2000-2010)

3.2 Why is deep Web important?

There is a simple answer for this question: Because the deep Web is over 99% of the Web as pointed out in the statistics of Section 3.1.

The importance of the deep Web is obvious to businesses that require or trade information. Organizations that themselves have large amounts of high-quality information, are placing their content online: News broadcasts, government-related knowledge bases, trademarks and patents, (Garcia-Molina and Raghavan 2000) financial information, shopping catalogs, flight schedules and medical research are some examples of deep Web databases (Wright 2009). Some of the content is stored in the indexable surface Web. However, most of it remains hidden for the generic search engine because the data is stored deeper, requiring additional querying interfaces to be used to reach the information.
The deep Web contains much of the structured data that the whole Web has. (Madhavan, et al. 2008) Reaching the deep Web efficiently will do much more than just improve the quality of search results for users. It might reshape the way companies do business online (Wright 2009).

3.3 Content of the deep Web

To begin the characterization of the deep Web, one must be first introduced to the primary fields that together generate most of the deep Web content.

The classical view of the deep Web content can be divided into five fields. The first are the databases of the Web. Databases cover most of the deep Web. All the other fields are actually minor compared to the databases, which contain tables of data. These tables contain rows that are the working units of databases. Rows contain the information that is generally wanted to be retrieved in forms of custom queries. Rows themselves are divided into attributes. The attributes contain the values that are retrieved during queries. The only way to gather data from a database is to make a query into it. Web databases include, for example, Access, Oracle, Microsoft SQL Server, MySQL and PostgreSQL. How the databases are queried and what challenges do we face while automating the queries are explained in Section 5 and partly in Section 6.

The second content category is the dynamic Web pages. Dynamic Web pages are generated, for example, through accessing a database and showing the requested information or by the choices that the user does while exploring a Web site.

The third is the content that is not available for public. Such content includes password-protected sites, limited content sites or otherwise restricted sites that do not allow a Web crawler to enter.

The fourth is content that is not HTML. Such content includes images, videos and software that cannot be converted into HTML, since most of the generic Web crawlers process only HTML documents.
The last is the *unlinked content*. Since most of the Web crawlers rely on HTML anchors, content that is not linked from any document will not be found by these crawlers. In the following subsections, we will take a closer look at each of the minor deep Web fields.

3.3.1 Dynamic and scripted Web pages

Through the last decade, the Web has continued to change from static HTML documents into dynamic content. Dynamic content refreshes parts of itself without the need to get the whole document again from the server or generates the document “on the fly” without the need to have it predefined in the remote system. Static content is the easiest for a search engine to detect and index - there are no moving parts. Before we observe the impact that a dynamic document has considering the deep Web, let us define it first.

The dynamic Web page differs from the static document because the user experience might differ from one view to another. The site might give new feedback for the user on every visit to the site (time-related). Each or some of the users might experience the site in a different way (user-related). The users can interact with the site in a way that alters it (interaction-related) or they might not need to do anything for the site to change (script-related). The relations do not exclude each other, and one site can use one or all of them to create a dynamic experience.

Technologies that allow the creation of dynamic Web pages can be divided into two branches: Client-side programming and server-side programming. The first alters the site directly on the users Web browser, whereas the latter alters the site by communicating with the Web server to create new content. In client-side programming, we create interactions to respond immediately to the user’s mouse or keyboard inputs. Famous programming languages in this field include JavaScript, Flash-technologies and AJAX-technique. We must note here that AJAX (Asynchronous JavaScript and XML) is both server-side and client-side technology. In server-side programming, we can also create interactions to respond to the user’s input. For the server-side to comply, the
client must send a request to the server where the process will take place. The most used server-side programming languages include Java, PHP, Python and ASP.

The importance of dynamic Web pages is straightforward: We can offer services online that are more complex. The dynamic sites allow, for example, the creation of online stores that query the databases behind the service to publish the product lists depending on what the users have chosen as arguments for the query. The opposite action, in static HTML, would mean that every possible permutation of the product list is created as an HTML document. General maintenance alone would be troublesome to do on such a site.

Dynamic sites can be created “on the fly”, which means that there is no actual Web page on the server corresponding to the shown document. This is the major problem of indexing a dynamic Web site. Since the crawler is looking for documents, it cannot find a dynamic page unless it imitates the actions that cause the dynamic page to be created.

**Example 1: Dynamic URL**

A dynamic Web site can be detected usually quite easily because there are query strings attached to the URL of the document. Here is an example of a dynamic URL:

```
http://www.peikkoluola.net/lataamo/private.php?page=blog&id=82718211
```

What the dynamic URL actually does is to fetch specific data from a database and show it to the user by applying the formatting rules of the dynamic site. In the example case, the system is using a blog application and showing the blog entry at database identifier id=82718211.

System administrators can use a technique to help the indexing process of this kind of deep Web resource. This technique requires the administrators to configure their Web server to use markings that do not include a question mark. The reason for this is that the question mark was generally used as a “wall” for Web crawlers. From the Web crawler point of view, ignoring the question mark will lower the chance of getting lost
in the Web site parameters. This will generally save the crawler time to pursue other servers. The con of this technique is that a loosely defined crawler with limited speed control might overload the server by continuous queries. The technique is to remove query strings from the URL. Using the technique, we can transform our dynamic URL into a different format, as seen from the example link below.

http://www.peikkoluola.net/lataamo/private/php/blog/82718211

**Example 2: HTML document with dynamic content using AJAX and a database**

Content that is dynamically refreshed by the AJAX-technique is a challenge to index. In this example, we have a dropdown menu in which the user can choose different objects. Choosing any object from the dropdown menu will change the content of the Internet site immediately. The model-field is not static and cannot be indexed without actually using the dropdown menu to generate the content. The HTML outcome can look as the illustration in Figure 7.

```html
<h1>Choose your vehicle</h1>
<form action="#">
  <span>Year</span>
  <select id="modelYear" onchange="refreshModelList();">
    <option value="">Choose...</option>
    <option value="2006">2006</option>
    <option value="1995">1995</option>
  </select>
  <span>Car</span>
  <select id="make" onchange="refreshModelList();">
    <option value="">Choose...</option>
    <option value="Chevrolet">Chevrolet</option>
    <option value="Dodge">Dodge</option>
    <option value="Pontiac">Pontiac</option>
  </select>
  <span>Model</span><br />
  <select id="models" size="6" style="width:300px">
  </select>
</form>
```
When the attributes “year: 2006” and “car: Subaru” are chosen from the form field, the system generates a list of available models for the car. Clicking on the model will generate a car page (from the database) to show all the information of the vehicle. Unless the whole system is generated to rely on the tactic shown in our first example in this section, none of the car pages will be indexable by a general search engine making them instantly part of the deep Web. Techniques to overcome this kind of obstacle are explained in Sections 4 and 5.

3.3.2 Limited and private content

Limited content of the Web can be included in the list of challenges for the Web crawlers. Very similar to the concept of private content, limited content offers users, for example, a document for certain action. This action can include a transaction of currency, using personal details that have been purchased earlier or simply allowing the content to be visible for a limited period of time. The rest of the Web site is fully visible and accessible (therefore possible to index) but certain content in the HTML document might not be available for public by default. It is unlikely that a Web crawler can imitate complex site-specific actions required to gain access to the content.
Web sites that require authentication for users to access the content are also not indexable by a Web crawler. The Web crawler can index the login site but it cannot bypass the security. Figure 8 illustrates the common login screen found on various Internet sites that require authentication. Since the login feature requires human interaction (i.e. known username and password) and it is generally against the terms and conditions of the site, Web crawlers cannot enter them. Making the site or subsection of the site require authentication will prevent it from being indexed further.

![Lataamo Archives Login Screen](image)

Figure 8: Typical login screen

Some documents or folders are not wanted to be indexed and Web server administrators can define it themselves by using a common file that the Web crawlers are looking for on each server: robots.txt. This kind of proposition is called the Robots Exclusion Standard (The Web Robots Pages 2010). However, it does not force the Web crawlers to obey the wishes of the robots.txt file in any way, as it is rather a polite way of informing what is wished to be left out of the indexes. It has been noted that stakeholder Web crawlers (introduced in Section 2.4.1) are made to do so to be aligned with the standard. Table 5 describes an example of robots.txt-file.

<table>
<thead>
<tr>
<th>Table 5: Example of Robots Exclusion Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>**User-agent: ***</td>
</tr>
<tr>
<td>If the value is '*', the record describes the default access policy for any robot that has not matched any of the other records. It is not allowed to have multiple such records in the &quot;/robots.txt&quot; file.</td>
</tr>
</tbody>
</table>
3.3.3 Non-HTML content

During the last decade, non-HTML formats have been making progress to become indexable by the stakeholder search engines. Currently one can find files in PDF (Adobe), DOC (Microsoft Word), XLS (Microsoft Excel) and PPT (Microsoft PowerPoint) while searching the popular surface Web search engines (Invisible or Deep Web 2010). These file formats can be converted into HTML format thus making them part of the surface Web instead of the deep Web.

Progress has been made in indexing the multimedia of the Web. Several of the most popular search engines have included the option to search for images through their site. Some search engines offer audio and video searches. However, multimedia is generally found by searching for the text around the link. It would be technically complex to search through the media format’s meta-information or the text that is actually part of the picture or video itself. One could use such data to create more efficient multimedia indexing or multimedia retrieval.

One popular standard is the Exchangeable Image File format (EXIF). EXIF meta
information can be saved, for example, into JPEG or TIFF file formats and is popular among cameras (Exif Version 2.3 2010). The amount of metadata a simple picture has, viewed for instance from the Microsoft Windows 7 operating system, is astonishing. The categories range from description (title, subject, tags, and comments) to very technical details of the camera, which was used to take the picture. Integrating EXIF information into search engines could allow far more advanced search queries to be performed.

Officially, the popular standard for video metadata is considered to be the MPEG-7. However, because there are many different file formats for videos available, the standard is not as widely spread as in pictures. This does not mean that there is no metadata in videos. They have not been just as well standardized as in images, which is important for an automated system like the Web crawler.

However, as some of the documents have ascended into formats that can be indexed and thus become part of the surface Web, new types of documents appear that can be classified as a deep Web resource. Such new applications include the blogging sites of the Web (depending of the implementation: sites may require authentication, use databases and be generated dynamically) and social networking sites of the Web (usually require authentication and use generally heavy scripting or are based on RIA (Rich Internet Application)).

3.3.4 Unlinked content

As noted in the previous section, the Web crawler travels through links from one document to another. Should a document be without any other document linking to it, the Web crawler will not find the document as it has been “cut off” from the Web, as illustrated in Figure 9.

![Figure 9: Unlinked content](image-url)
Such content might be found using a different indexing system. It is a common procedure that Web administrators deny access to browse the Web server files. Should it be allowed, the crawler could browse the server for files and index them instead of following links from documents to another. Should the entire server be without anything linking to it, the crawler would not automatically find the server and thus the whole server would be unlinked content.

3.4 Basics of HTML Query interfaces

The deep Web databases are accessed through query interfaces. The HTML form fields contain the elements that are used to create the query. The HTML input elements include: Text fields, password fields, selection boxes (multiple choices from many possible choices), radio buttons (single choice from many possible choices), drop-down selection menus, buttons, upload fields for files and hidden values that are sent on submit.

The interfaces transfer the keywords into the Web server. From the keywords, a query is created. The server returns the results that the database has stored about the queried subject. Normally a user would write the form field elements and submit the form manually. However, to index the deep Web databases through a query interface, the search engines must automate this user driven process. The search engines must be able to read the query interfaces and submit the form themselves automatically in order to be able to index the results of the query. This is part of the automatic finding of search interfaces problem. This problem will be covered in more detail in Section 5.1 of this thesis.

Figure 10 is an example of a query interface to a deep Web site database and the process of the data retrieval that happens in the background while the user waits for the query results to appear on his desktop.
As the user enters his query to the query interface, the interface will transfer the query to the query validator. The query validation can be implemented in various ways depending on the need of the service provider. Some forms require specific characters to be used in a query or some form fields to be filled before the query can continue. The query validator can be divided between client-side and server-side programming. Should the system lack a query validator, it is a severe security issue for the database. The most basic operations of the query validator include parsing or preparing the query so that it cannot alter the database in a malicious way. One such way is called SQL Injection.

Unprotected query interfaces can lead up to a scenario where a user can change the intended query to something else. For instance, should a form field be used to login to a specific Web site, an unprotected form field can be tricked to display the database table names and a user can insert his own account into the database or empty the entire table or the database only by using his Web browser. The basics of SQL injection are that the user wants the system to show an error message containing important information about the system structure, for example the database table names. The simplest SQL Injection test is the quote mark (‘), which would raise an error in an unprotected system. The
basic rule against SQL Injection is that every element that a user can alter and submit
must be taken as a potential threat to the safety of the database and the whole Web site.
The input received from a form field must always be validated.

When the conditions of the query validator are met and finished, the query reaches the
actual database to search for information. After the specific rows and columns of the
database are selected with the SQL query, which is formed after the final phase of the
query validator, the retrieved data is ready to be assembled for the user. Depending on
the need of the service provider, the next phase can lead to an intermediate page or
directly to the result page. Through these pages, the user reaches the results of his query.
This process is walked through in Section 4.

3.5 Meta queries and automation

In order to control and collect relevant information through an easy source, one must
observe the structures above an individual search engine. One must form an efficient
system that does not simply access a predefined single deep Web resource but instead
can access multiple deep Web resources that were not predefined for the system.

So far, we have studied the process of a single query interface that accesses a single
database. In other words: A single Internet Web site and the database that the site uses.
We must move to a higher level to understand the overview of what we are dealing with
as we discuss about accessing the deep Web. Figure 11 represents a network of sites that
are linked together. As we connect to a new node in the graph, the node reveals new
paths for us to follow. However, there might not be a link back to the previous site,
thus the graph is directed. This directed graph expands on each discovered node. As
we expand it by crawling more and more Web sites, we will find similarly more and
more Web databases. Each new database deepens the Web and causes large-scale
integration challenges for a metasearch engine, as well as challenges of the dynamic
nature of the Web and its databases. (Chen-Chuan Chang, He and Zhang 2005). These
databases are the deep Web resources that we generally try to access when generating a
metasearch engine. Therefore, metasearch engines are query interfaces for a collection
The scale of the Web demands automation in efficient querying. Automation in this context means that one must automate the searching of required resources and after the resources are found, they must be queried for results. All this must be made in a time that a normal Web interface user can spare to wait for the query. As the user enters his desired keywords into the metasearch engine, it will perform both finding of sources and querying of sources, which means accessing deep Web resources that are relevant for the user. This means that the process must be done “on the fly”. The problem of querying multiple Web databases with different query interfaces is covered in Section 5.3.

### 3.5.1 Query Methods

Currently there are two identified query method categories available. The first category is the prior knowledge-based query method (PKB) and the second category is the non-prior knowledge-based method (nPKB).

In the prior knowledge-based query method, we need to construct the knowledge base beforehand. This knowledge base is used when the queries are generated. *Label value set* is an example of a PKB query method. In label value set, a predefined table is used for passing values to query forms. (Liu, et al. 2006) The disadvantages of PKB are...
twofold: The methods perform efficiently only when there are enough predefined data available to use and the interface of the query is simple (for example, a single text input element).

The non-prior knowledge-based methods generate the keywords that will be used in the queries. Generating occurs through analyzing the data records returned from the previous query. Hence, the process does not rely on predefined knowledge, which is a great advantage in the automation of the whole process. The first example of these methods is called most frequent keywords. The method relies on the idea that the next query will be generated using the most frequent keywords in the previous records.

Second, in greedy query selection, the candidate query keywords are generated from the obtained records. The keywords harvest rates are calculated to determine the keyword with the maximum expected harvest rate (HR). The keyword with the maximum expected HR will be selected for the next query. Given a target Web database DB, and a local database $DB_{local}$ containing the data records already crawled from DB, the harvest rate (HR) of query $q_i$ is defined as:

$$HR(q_i) = \frac{[num(q_i), DB) - num(q_i, DB_{local})]}{cost(q_i, DB)}$$

where $num(q_i), DB$ and $num(q_i, DB_{local})$ corresponds to the number data records matched by $q_i$ in DB and $DB_{local}$. The dividing $cost(q_i, DB)$ stands for the cost of obtaining all the result pages (Wu, et al. 2006).

The third nPKB method is a distinct attribute-value graph. From the graph we are searching for an optimal solution by transforming the problem into finding a weighted minimum dominating set which corresponds to the distinct attribute-value graph. An attribute-value graph for a Web database (AVG) $G=(V,E)$ is an undirected graph in which for each distinct attribute value $av_i \in AVG$ there exists an unique vertex $v_i \in V$. An undirected edge $(v_i, v_j) \in E$ iff $av_i$ and $av_j$ coexist in one relational instance. Each edge in AVG stands for a relational link between $av_i$ and $av_j$. The dominating set is a classical NP-complete problem in which graph $G=(V,E)$ is a subset $D$ of $V$. Every vertex that is not in $D$ is joined to at least one member of $D$ by an edge (Wu, et al. 2006).
The last example to be introduced is the *minimum executable pattern* (MEP) method. The query method extends query interface from a single textbox to a MEP set. The query is performed by choosing a MEP and MEP keyword vector. The next query is created with the maximum expected efficiency through the acquired information. (Liu, et al. 2006)

The nPKB improves the deep Web crawling capability over the PKB methods. However, these methods are not without drawbacks. The first drawback is that the queries are based on a single textbox and the candidate keywords are assumed to be suited to the textbox. Second, the query selection decision is based on the obtained records. Last, query selection lacks sufficient knowledge in the initial period. These three drawbacks might result in a phenomenon called Data Islands. (Liu, et al. 2006) In Data Islands, the total acquired records may constitute only a small fraction of the target deep Web database leaving a lot of data untouched.

### 3.6 Analyzing the deep Web

It has been pointed out earlier in this thesis that the deep Web is enormous in size compared to the surface Web. The difference between surface Web and deep Web has been defined and some challenges and techniques of crawling the deep Web have been explained. We return now to the question of the scale of the deep Web. How can we estimate the scale difference? What are the tools that allow us to calculate and derive estimates about the amounts of deep Web resources, their contents and create the statistics? This section will demonstrate two different tools that can be used to do these estimations.

#### 3.6.1 Overlap Analysis

To estimate the count and size of deep Web sites, the first of the techniques is called overlap analysis. From mathematics we can use Set Theory as a guideline for creating an overlap analysis of deep Web resources. The analysis uses pairwise comparisons of
sets. The pairwise estimates are repeated for all of the individual sources used in the analysis.

We have two sets, A and B, which overlap each other creating the intersection $A \cap B$. The total population of the set is $E$ (see Figure 12). Assuming random listings for both A and B, the total size of the population $E$ can be estimated. The estimate of the fraction of the total population covered by A is $\frac{A \cap B}{B}$ (Bergman 2001). When this is applied to the total size of A, an estimate for the total population size $E$ can be derived by dividing this fraction by the total size of A. This transforms our formula into: $\frac{|A|}{|A \cap B|}$ and furthermore from this we get $\frac{|AB|}{|A \cap B|}$.

**Example: Pairwise comparison**

Let us assume that we know our total population to be $E = 100$. Sets A and B contain both 50 items (A=50, B=50). We could predict that on average 25 of these items are shared by both A and B. The remaining 25 items would not be shared by either one. Using the formula defined above: $\frac{50}{25}$ resulting in the population of $E=100$.

![Figure 12: An Overlap Analysis](image-url)
The estimates gained from using an overlap analysis are only valid should a set of criteria be met. First, it is important to have a relatively accurate estimate of the total set size for at least one of the two sources (A or B) in the pairwise comparison. Second, both sources should obtain their listings randomly and independently of one another. (Bergman 2001) In order to analyze the total scale of the deep Web, an average site size in documents and storage use must be gathered and used as a multiplier applied to the entire population estimate. Obtaining the required information is not an easy task to accomplish, and it involves considerable time to evaluate each deep Web resource.

3.6.2 Random sampling of IP addresses

The random sampling technique estimates the count of the deep Web sites. The analysis is done by selecting a sample of IP addresses from the entire space of valid IP addresses and removing the reserved and unused IP ranges. Since the IP space is finite, that fact can be used to obtain estimates of the deep Web scale. (He, Patel, et al. 2005)

Random sampling is done by using an HTTP client to make an HTTP connection to the IP and download HTML pages. Web databases must be identified from these pages so that one can create the statistics of found deep Web resources.

Example: Random sampling

The sampling begins when $IP_1$ of the sample space $IP_s$ is sent an HTTP request. If the IP responds with an HTTP response, the system will try to download the HTML pages and find deep Web resources on it. If the current IP has a deep Web resource, the system stores the details of the analysis in a database for further reference. Should the IP address return no response, the system moves automatically forward to the next IP in the sample range ($IP_s = IP_1, IP_2, IP_3 ... IP_n$) until an IP sends an HTML response again (illustrated in Figure 13).
However, using random sampling ignores virtual servers completely, since there can be many servers behind an IP. A Survey done during 2010 (Web Server Survey 2010 2010), (Web Server Survey (By IP Address) 2010) shows that there were 4,013,620 IP addresses while there were 205,368,103 sites responding. Using these two surveys as a source, the hosts ratio over IP is as high as 51,17.
4 A DEEP WEB NAVIGATION PROCESS

This section concentrates on the process of reaching through the Web interface to the result page where the desired information lies. The process can be threefold starting from the interface page where the form fields are. The second step is the intermediate page between the interface page and the result page. This middle step is optional and not present in every interface by default. The third step is the result page where the desired information is presented in various formats.

To get the process started, the idea is twofold: First, the user has to identify and label the relevant form field input elements (Section 4.1). Second, one uses the user-provided input values at runtime to fill out the appropriate form fields and then check after submit if we have reached the result page. (Wang and Hornung 2008) In this thesis, one of the processes is defined by the *page-keyword-action paradigm*. The paradigm can be defined as a system that fills out forms with provided input parameters and submits the forms to the Web site (other intermediate page techniques are mentioned in Section 4.2). In other words, we continuously fill out and submit the form fields with legal value combinations to reach the result page. (Wang and Hornung 2008).

![Diagram](image)

**Figure 14:** The triangle problem in query processing

4.1 Form fields

The process begins with the identification of all the form fields that the Web site has. These will be stored in a database for further analysis. The analysis is responsible, for instance, for detecting if the input fields of the HTML forms are static or dynamic. This
can be done by observing the following phenomenon: After the first input element has been modified, are there any of the other input elements different from what they were before the first input element was modified. One can also observe the source code for calls to general dynamic functions and modify the system to react to them. Generally, dropdown menus are trivial to automate since there is no need to guess predicates. The system knows all the selectable options.

**Example: Identification of form fields**

To clarify this, we have an example (Figure 15), where the same site is separated in different views. In the first view, all the relevant form field input elements are listed by observing them from the source code. Second, the HTML version of the form field for the user is displayed. Third, a relation tree of the form field elements is created for both static and dynamic input elements (Figure 16 - 17).

![Figure 15: Visualization of input elements](image)
Web developer’s tool (designed for Mozilla Firefox) was used to retrieve the list in Figure 15. Web Developer 1.1.8 (author Chris Pederick) is a tool, which allows transforming the form field details into a table. The table separates the name and type of the input element as well as other specific details.

The graphical interface was loosely translated from the original Web site (www.vvo.fi), which does not support English interface. Nevertheless, the dynamic input elements in the graphical version are the two scrollable input element selections and the rest are static. The dynamic elements are Area: Town and Object: Type. Both Town and Type are generated automatically when the Area and Object input elements are chosen. The size-element and detail-element and all their sub-elements are static.

The relation tree of static input elements for the previously shown interface is shown in Figure 16.

The relation tree of dynamic input elements for the previously shown interface is shown in Figure 17.
4.2 Intermediate page

Depending on the implementation, an automated system must determine its current location after a query has been run successfully or unsuccessfully. This is most important in systems imitating user-like crawling such as the DeepBot (Section 5.4.2). The page where the query relocates the system might not be the result page, which we want to reach. It can be the intermediate page. Intermediate pages are not part of every deep Web searching interface, but common enough to be part of the navigation process. This intermediate page can vary greatly in technical details, design and accessibility. Varying usually occurs because each Web site is created by a single Web developer or a team. Unless they are using popular templates, the Web site design is unique.

First, intermediate pages can be divided into different types. The most noteworthy intermediate page is the error page. In the error page, the query has not been delivered successfully for one or more reasons. The queried system can inform the querier with a comment which states that the query was unsuccessful. The system can also display a detailed output on what the user needs to alter in his query. The reporting can be something in-between the previous two models or the system will inform nothing at all but does not perform the query, either. An automated system must be able to determine whether the query has been run or halted by a validation system. Should an error page appear, in a case where there are no results available to be shown by the run query, the system must recognize this and be able to distinguish an error page from a query with no results.

Second, the intermediate page can be a list of results related to the original keyword(s). This is a popular design in Web sites that focus on, for example, multimedia. Should the user insert a keyword to the query interface, the intermediate page could show all the related objects from which the user must select one in order to reach a result page. Suppose, for example, that a user is trying to search for the description of a movie called Alpha from a movie database. He enters the required keywords into the query interface. The intermediate page shows the user Alpha (1999), Alpha II (2002), Alpha III (2004) and The Cat and The Alpha (1967). Following these anchors would lead the user to the actual result page.
Third, the intermediate page can be a general version of the “list of results” mentioned in the previous paragraph. The user must issue a command in order to relocate into the result page of the query at hand.

This thesis demonstrates five different techniques to deal with the intermediate page. The first approaches the intermediate page problem with a page-keyword-paradigm. The system works through finding the correct path to get from the intermediate page. It identifies specific keywords in the HTML document and it determines through actions how the system should continue to reach the result page. When the result page has been found, its URL can be retrieved and processed in order to index the page and allow this deep Web resource to be stored and retrieved through a search engine query interface. Illustration of the process is in Figure 18. In the following subsection, we will observe with an example of how this path through the intermediate page could be found with the page-keyword-paradigm approach.

![Figure 18: Automating the Intermediate Page](image)

The second technique is called screen scraping. In screen scraping, the system tries to retrieve the data from the presentation layer of the queried system. Third, instead of using the presentation layer, we use directly the data layer through an Application Programming Interface (API). The fourth approach is a combination of the screen scraping and API. The last technique is not directly a tool to process an intermediate page but more a method, which combines all these techniques.
4.2.1 Path addressing language

Most deep Web intermediate pages and result pages are created dynamically. Although the pages are dynamic, they are created from a template that suits the rest of the Web site. In this template, one can characterize the similarities between the different dynamic pages, which are part of the template. The similarities in which we are interested here are the keywords. With keywords, we can determine where we ended up after submitting the form field. Keywords can include the original input element names or possible errors that occur in the submitting of the form field.

One technique to reach the result page from the intermediate page is by a path addressing language called Keyword-Action Path (KApath), which is a subset of XML Path Language (XPath). We will return to XPath in practice during Section 6.

These definitions of XPath are the most relevant to the subject at hand:

- XPath is a syntax for defining parts of an XML document
- XPath uses path expressions to navigate in XML documents

XPath is used to select specific nodes or whole node sets in an XML document. Hence, XPath uses a tree representation of an XML document for the navigation. The operators in XPath allow moving through the tree by selecting the specific nodes that are relevant for the current object. The original XPath 1.0 is widely used and appeared as a recommendation on 16th of November 1999. The follow up XPath 2.0 is less popular but contains a wider range of recommendations from 23rd of January 2007. (XPath Introduction 2010)

KApath was created in order to uniquely identify the appropriate HTML elements, which the stored actions should be executed on. In order to access the appropriate action element, the system finds the common ancestor of both the keyword element and the action attribute. The system descends downwards in the action attribute branch. Afterwards, the registered actions are executed for the found action attribute (Wang and Hornung 2008). KApath supports the path expressions that are illustrated in Table 6.
Example: KAPath expression

In the following example, we are observing an HTML table element. We need to create a navigation model for the intermediate page. The HTML element TBODY is a table body. It can be used to separate different sections. TBODY is the first parent node for both the keyword and the action attribute in this KAPath example (see Figure 19). The system automatically generates a KAPath expression, which allows optional elements between the keyword and the first parent node. These optional HTML elements in this example are TR, TD, and H2.

An absolute path is a path that points directly with the full path from the beginning element to the last wanted one. The absolute path starting from the root of the tree to the action attribute is as follows, which equals to the action attribute tree structure:


The tree structure for the keyword and the action attribute is visualized in Figure 19. The tree structure for the action attribute of the example is as follows:

- TBODY (TR, TR) / TR (TD, TD) / TD (INPUT)

(Wang and Hornung 2008)
Finally, the KApath expression created from the list of KApath expressions:

- \( /P::P/TBODY[@a1=v1][@a2=v2]/Child::Child/INPUT[@a3=v3] \).

The KApath expression is twofold: Parent nodes and child nodes. The first part of the expression means that the parent node in the DOM tree will be selected, if it has an attribute \( a1 \) with value \( v1 \) (and attribute \( a2 \) with value \( v2 \)). The second part of the expression means that when the parent node is found, the child node in the DOM tree will be selected if it has the input attribute \( a3 \) with value \( v3 \).

Based on the user’s browsing behavior, we can create a system to draw the complete navigation model. With the action log, the system can determine paths and tree structures for each action. (Wang and Hornung 2008)

- Identify the intermediate page keyword by clicking on the text in the Web page.
- Determine the closest HTML element and store the relevant information.
- Monitor user behavior and store actions that lead to the result page.

(Wang and Hornung 2008)

Figure 19: Example of KAPath
4.2.2 Screen scraping

Another technique to navigate through an intermediate page is screen scraping. We can call screen scraping synonymously as automated parsing of the source code that is used to generate Web pages (Alba, Bhagwan and Grandison 2008).

Source codes are, however, generated by different tools or methods. Some Web pages are generated by automated generators while others have dynamic content in them in forms of AJAX or client side scripting. All of these have direct impact on the readability of the source code since they make the HTML markings more complex. Figure 20 is a fine example of complex HTML.

To be able to use screen scraping and get the important data out from the source code the system must be able to parse out the parts that are not needed or be able to collect specific parts of the source code. Screen scraping can provide imprecise or erroneous data when the source code of the site changes (Alba, Bhagwan and Grandison 2008). Small changes in the target site might not affect the outcome. Should the entire layout of the site change, at least the screen scraping parser should be recoded. This means that screen scraping is taking place on the presentation layer of the product. This is the layer, which is visible for users. Screen scraping will be inspected in Sections 5.1 and 6 in more detail.

Figure 20: Source code example from Youtube
4.2.3 Application Programming Interface (API)

Web Services (WS) technology, along with Web 2.0 tools, have encouraged Internet-based products to provide Application Programming Interfaces (API) for the development community. APIs are suitable, for example, for creating automated queries. (Alba, Bhagwan and Grandison 2008) Today, there are increasingly large numbers of APIs available for developers: Google API, Amazon API, Facebook API, YouTube API, and so on. These interfaces provide an alternative technique to screen scraping mentioned in the previous section.

The main difference is that the API structure tries to separate the data layer (databases and structures) from the presentation layer (GUI and the user experience). API gives developers a way to ignore the presentation layer and access the data layer directly. If the product is structured well, the ever-changing presentation layer will not stop the developers from using the more stable API as a base of their operations. The developers can rely on the fact that their own programs will not need to be altered on every major presentation layer update. By developing a resourceful API, the developers can also be given access to information that is not directly available from the presentation layer, where the screen scraping takes place. However, even APIs do not always provide a reliable consistent behavior. The API is released by the product developers, and it might suffer from poor documentation or quirky functionality implementations, which reduce the efficiency of the interface. (Alba, Bhagwan and Grandison 2008)

Currently screen scraping works better than APIs because of three reasons. First, the screen scraping is more popular than APIs because screen scraping is more in use since only a few Web site offers an API. Second, Web sites are motivated to ensure that their presentation layer is flawless. This includes high structurality, consistency and accuracy in the layer shown to the end user. This suits well for screen scraping as it relies on such features. Third, administrators are concentrating on the presentation layer, since that is what the user will be interacting with. (Alba, Bhagwan and Grandison 2008) This leaves much less time to be spared on the data layer and the development community. Comparison between screen scraping and APIs is shown in Table 7.
In the rest of this section, we will be observing what kind of functionality and limitations two products, with a functioning API, provide. The products selected for this observation are Facebook and YouTube.

### Example 1: Facebook

The Facebook API is divided into five core concepts. First, there are the social plugins. These plugins allow developers to add features to their own Web sites that allow the direct manipulation of Facebook social activities. The plugins may, for example, allow users to like or recommend the developer’s site on Facebook and comment the Facebook page directly from the developer’s site without finding it first on Facebook. Second, the Graph API is the core of the Facebook platform. This API allows representing objects (for instance people, photos, events or pages) and the connections between them. Third, the social channels allow developers to integrate their Web pages with news feeds and requests. Fourth, the Facebook authentication enables developer’s applications to interact with the Graph API on behalf of Facebook users. It also provides a single-sign-on mechanism. Last, the Open Graph protocol allows developers to integrate their pages into the social graph. (Facebook Core Concepts 2011)

Since Facebook relies on user experience, the API must be limited enough to protect the product users from allowing malicious applications run rampant on their site. On the other hand, the API must provide enough functionality to allow the development

### Table 7: Screen scraping vs. APIs

<table>
<thead>
<tr>
<th>Feature</th>
<th>Screen Scraper</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accurate</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>System</td>
<td>Complex</td>
<td>Simple</td>
</tr>
<tr>
<td>Semantically coherent</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>System functionality updates</td>
<td>Is affected</td>
<td>Less affected</td>
</tr>
<tr>
<td>Presentation layer updates</td>
<td>Is affected</td>
<td>Not affected</td>
</tr>
</tbody>
</table>

(Alba, Bhagwan and Grandison 2008)
community to create useful applications on it. Facebook relies on authentication to view much of its content (since users can protect themselves by disallowing others to see their information). Therefore, the API products can only see what the user can see. To allow the application to gather information from the user’s account, users must first approve the application. Most of the functionality is still read-only, even if the application is approved by the user. Allowing write functions would be dangerous for security, but there are some available. The APIs can publish to the user’s feed with a time limit to avoid creating spam. The API can send notifications or emails to the user’s friends with a sending limit to avoid spam. The API can upload photos to the user’s photo bank or create a status text with additional permission by the user.

**Example 2: YouTube**

YouTube’s commitment to the developer community includes APIs and tools. To stay aligned with the section theme, we will only cover the APIs here. First, the Data API allows developers to integrate YouTube functionality to their own applications or Web sites. The developers can create applications that, for instance, use the YouTube searching, upload videos to the site or create playlists on YouTube. Second, the Player APIs allow the developers to gain control over the YouTube video playback on their own site. Basic settings can be configured and customized controls can be created. (YouTube APIs and Tools 2011)

The API is not without limitations. Uploading with the API is limited to 15-minute long videos that cannot be larger than 64GB and a developer account cannot have more than 2000 videos uploaded. There is also a limitation against excessive bandwidth usage, which is returned by an error message containing `too_many_recent_calls`. When using the YouTube search functionality through the API, the search results are limited to a total of 1000 results.
4.2.4 Screen Scraping API

When a popular Internet product has not developed or released their own API for developers to use, it might give space for “rogue APIs” to form. These kinds of APIs are not released through the official product development team but rather by the development community creating screen scrapers for the products. These APIs do not get access to the data layer of the product in the same way as the official API would get. However, those who would use the third party API would get the benefits of using an API in the first place. Thus, the more accurate definition would be Screen Scraping Application Programming Interface (ScrAPI) suggested by Paul Bausch in 2002. (Alba, Bhagwan and Grandison 2008)

The development community has produced an increasing amount of scripts dedicated to scraping the popular Internet products available today. These scripts can be used legally if it is accepted in the product terms and conditions. This must be taken into account on every product since their terms of use differ from one another. Usually products that want to reduce the screen scraping of their sites tend to publish an API for the developers to use. The API is created in a way that reduces the payload of the screen scraping which reflects as decreased system load.

4.2.5 Web Mashups

Web Mashups are not exactly a technique or theory how to get past the intermediate page of the triangle problem (Figure 14). Web Mashups are a technique to combine one or more deep Web source results into a different entity (What is a Mashup 2010). The system collects information from intermediate pages and result pages (screen scraping) or from the data layer of the product (APIs). The technique can be applied after the required information has been obtained from the selected products. How the system is built is entirely up to the development team. It can use screen scraping, APIs, both of them or it can use a data storage created and maintained by the development team themselves for gathering data for the Mashup.
Example: A Web Mashup Application

In Figure 21, we have an example of a Web Mashup application. Web products in this example are shops that offer their customers online catalogs for browsing. The fourth product is interested in delivering a service that offers more information about a specific catalog item. The three other shops (Web product 1, 2 and 3) create an arrangement to allow the fourth product to legally operate. The fourth product will then use screen scraping to browse the contents of the two first Web catalogs. This is done because the shops offer nothing for the Web development community that could be used to aid the fourth product in its task. However, the third shop has created a useful API for their catalog system, and the fourth product can use this API to query the catalog.

After the system has been created, the fourth product will be able to show their customers the collection of the catalogs of the three shops. As the customers of the fourth product query the system, they will be able to see the closest location where a single item can be purchased. This is made possible by integrating Google Maps API for each catalog and then showing the shop location on a map.
5 SEARCHING THE DEEP WEB

Many questions arose throughout the previous sections and the current section will concentrate on giving a more practical definition of what the challenges of automating the queries from the deep Web are.

The section begins with the task of finding a search interface among the source code of a Web site (Section 5.1). After the form field has been found, it will be queried (Section 5.2). However, a single database query can be created by simply observing how the system operates. To move a step further, a technique to query multiple databases is described (Section 5.3). Finally, to summarize most of the topics discussed in this thesis, two metasearch engine architectures are explained (Section 5.4).

5.1 Automatic finding of search interfaces

To be able to query multiple Web databases (covered in Section 5.3) we must first find an answer to (or to define) the interface of the process. Since Web databases are queried through the HTML form fields located inside an HTML document, we must find them in order to perform actions upon them. There are two ways the forms fields can be located throughout the Web: One is the manual insertion technique, the other is the automatic finding technique. Though the techniques differ from one another, their implementations have similar aspects.

In the manual insertion technique, the metasearch engine allows its customers or users to point out databases for it. Those who are interested in helping out or have other interests can insert a database to the metasearch engine collection (which might or might not get parsed). This technique means that the collection of databases, which the metasearch engine queries, is sustained and updated by its potential users or groups of interest. In some cases, this technique might suit the needs of the metasearch engine. However, the drawback is that the collection is sustained by human effort.
To avoid the drawbacks of the manual insertion technique, let us introduce the *automated finding technique*. In this technique, the system is given the responsibility to search for relevant databases instead of human insertion. The technique overcomes mainly the need of human interaction in the indexing process. However, the system will require a lot of effort put into it to automate the process.

The system works as a specialized Web crawler. The targets of crawling are not the traditional HTML documents, but rather the specific form elements inside the HTML documents. How the actual query might be done to a form field, is covered in Section 5.2. Figure 22 illustrates the two potential techniques that the metasearch engine can use to gather its index of Web databases. As seen in the figure, the techniques do not exclude each other. There is also some automation in the manual insertion technique, when the system searches for the valid database entry.

Both of the techniques must parse through the DOM (Document Object Model) elements in the source code of the Web site to locate a form field. After a form field has been located, there may be a series of qualification rules to determine if the found form field is a valid database query interface based on predefined or intelligent system rules.

Figure 22: Resource gathering techniques
Should the element qualify as a valid interface, the Web site is then added to the index of the metasearch engine as a site to be queried in further metasearch queries. The main difference between these two techniques is that the automatic finding technique uses a Web crawler to find its sources while the manual insertion technique uses human input to find its sources.

**Example: Query a single HTML form field element**

The traditional form field contains some basic elements as noted previously in Sections 3.4 and 4.1. The following HTML code will introduce us closer to those elements in a practical example.

```html
<form name="QueryInterface" method="post" action="Query.php">
  <input type="text" size="25" name="SearchString" />
  <input type="radio" name="RentSale" value="Rental" />
  <input type="radio" name="RentSale" value="Sale" />
  <input type="Submit" name="SubmitSearch" value="Search!" />
</form>
```

With some CSS (Cascading Style Sheets) marking, we can transform the previous HTML form field into a simple query interface for a Web site. The example is shown in Figure 23. Since the names of the query interface are straightforward, we have little trouble to pick up this interface with a crawler that searches for databases containing apartments. However, since the element names are written by a Web developer, they might be whatever he would have wanted them to be. In such a case, we would not have found anything concerning apartments in the element names. With the previous scenario, we have already found our first exception to take into account for the crawler system: What to do when the form field element names are not relevant.

![Figure 23: Example of HTML Form](image)
Form fields have four distinctive attributes in them (not counting Javascript). These are action, method, name and enctype. The “**action attribute**” points to the URL where the script is that is called to process the form. It can point to an external file (<form action="submit.php">) or it can point back to the same file where the form is (<form action="<?php $_SERVER['PHP_SELF'] ?>").

The “**method attribute**” is twofold. It can be either POST or GET. When using POST, the user will not see the data as it is passed forward. POST is generally used while transferring large information sheets. GET, on the other hand, will be shown to the user in the URL. Should the user submit the form in Figure 23 with GET, the URL would be shown as search.php?SearchString=Turku&RentSale=Sale. By default, POST is more secure in transferring information such as username and password since they are not shown.

The “**name attribute**” is the identifier, which is used when scripts are called to process actions with this form. The identifier is treated as a String. For example, if a JavaScript function would be issued to alter some element inside the form, it would first have to call the form with its name.

Should one need to alter the way the data is encoded, the last of the four attributes is needed. By default, the “**enctype attribute**” is not needed to be mentioned and it uses the "application/x-www-form-urlencoded" value. However, there are some special cases when the encoding must be changed, in other words, the enctype attribute must be mentioned in the form. One of these cases is when a file is being uploaded through the form field. The value to use would be: "multipart/form-data". (PHP Manual 2011)

As the Web crawler enters the site, which contains the form field of Figure 23, it could store the site address in a database and continue its journey through the links that the Web site has. The rest is left for the crawler system. Either the source code is just stored in the crawler database or it is parsed onsite. Either way, the crawler system initiates a parser to walk through the code. The parser will be responsible for the HTML exploration. The parser will find the form field and trigger another process to validate
the contents of the found form field either by simply validating HTML attribute values or by using the form field to generate query results. Section 6 will cover in more detail the technical basics of writing a Web crawler.

To be able to find a form field in the first place, we must be able to read the source code of a Web site using a technique from Section 4 of this thesis. This can be achieved in different ways depending on which programming language we have chosen. We will first introduce an algorithm for this and then an implementation with PHP.

FIND_FORM_FIELDS Algorithm gains the HTML document as a String parameter D. If D is empty or null, the algorithm will return null. If it is not such, it has data in it. In such an event, D is walked through to detect the beginning of a form field. When found, it will call upon another algorithm to parse the contents of the form field. If the String D does not contain the beginning of a form, the algorithm will return null.

Table 8: FIND_FORM_FIELDS -algorithm

```
FIND_FORM_FIELDS(D)
IF D is NULL
    THROW exception: NULL
ELSE
    WHILE D contains data
        PARSE_FORM_FIELD(D)
    ENDWHILE
RETURN form-field array
```

We begin the construction of the PHP implementation systematically. First, we will insert our specified Web site URL as a variable. Second, we will use a built-in function of PHP to gather the data of an HTML document and proceed according to the algorithm.
$internet_site = "http://www.peikkoluola.net";
$site_contents = file_get_contents($internet_site);

if ($site_contents === false) {
    /* An exception: no source code is stored */
}
else {
    /*Success: Source code is stored in the variable */
}

With the file_get_contents function, we can get the source code of a Web site (notice that the function has been deprecated in favor of the technique used in Section 5.2). The function will return a Boolean value false if the function cannot retrieve the Web site source code. (PHP Manual 2011) Therefore, we will be using condition (IF – ELSE) to determine whether the function has retrieved anything from the selected Web site. Using the function, we can move on to parse the source code to determine a simple way of detecting form fields.

The “stristr function” returns all of the remaining data from a String starting from the first occurrence of the selected argument (PHP Manual 2011). In this scenario, we will receive the remaining of the source code starting from the first occurrence of the String: “<form”. Should the function find this String from the source code, it will continue to another IF-ELSE condition according to the example below:

    /*Success: Source code is stored in the variable */
    else {
        if(stristr($site_contents, '<form') == true) {
            /* Success: Form field has been found from source code*/
        }
        else {
            /* Failure: No form fields detected in the source code*/
        }
    }

Now we are able to create a function that receives a Web site as a parameter. The advantages include the fact that we can now receive a list of Web sites we want to iterate through. This is in line with finding search interfaces automatically.
function find_form_fields ($internet_site) {
    $site_contents = file_get_contents($internet_site);
    if ($site_contents === false) {
        /* An exception: no source code is stored */
    }

    /*Success: Source code is stored in the variable */
    else {
        if (stristr($site_contents, '<form') == true) {
            /* Success: Form field has been found from source code*/
        } else {
            /* Failure: No form fields detected.*/
        }
    }
}

After finding a form field from a Web site, we need to get that form field for closer inspection. The following example will focus on the case when the source code of a Web site is retrievable. Let us first separate the form field from the rest of the HTML and store it in its own variable. The implementation in PHP is straightforward:

/*Success: Source code is stored in the variable */
else {
    /* Success: Form field has been found from source code*/
    if (stristr($site_contents, '<form') == true) {
        $form_field = stristr($site_contents, '<form');
        $form_field = stristr($form_field, '</form>', TRUE)
        return $form_field; }
    else { /* Failure: No form fields detected.*/
    }
}

If the source code contains the string “<form”, the function will store the source code to a variable. The variable $form_field will contain the source code starting from the first occurrence of “<form”. Next, the function will search the stored partial source code for the ending tag of the form field. The function “stristr” will return the beginning of the String before the found parameter, if it is used with the Boolean value “true”. When the
closing tag is found, the function retrieves the string between the tags.

The algorithm can be defined as a string problem. We have a string and we are only interested in a part of it. It is rarely at the beginning or at the end so we must be able to deal with it in between. Figure 24 illustrates this problem and a simple solution to it. The solution is not efficient since the source code could be huge in size and we are interested in only a small part of it.

The PARSE_FORM_FIELD(D) algorithm is called from the FIND_FORM_FIELDS(D) algorithm. The parameter D is the string of the HTML document containing a form field detected by the first algorithm. First, we need to divide the string into parts for further managing. We will first cut out all the excess information before the form field. Next, the algorithm expects a closure tag. It will read the HTML until a closure tag is detected. The data between the tags is the form field and it will be stored in an array. The algorithm will continue to inspect for any additional form fields in the source code and issue the PARSE_FORM_FIELD algorithm for any remaining form fields as well. This continues as long as there is data left in the string to analyze. In the end FIND_FORM_FIELDS(D) will return an array containing all the found form fields in a HTML document for further analysis.

With this method, the HTML document form fields have been separated. They can now be easily inspected. The following section will explain how to query a stored form field.
5.2 Querying a single Web database

Finding a form field from the source code varies in complexity depending on the chosen programming language. The example in the previous subsection was quite straightforward. Finding the form field however is only the beginning of the task. After the site has been scanned and a field detected, we must issue rules what to do with it.

Should the need be to query only one Web database, we could learn by hand how the system works and initiate an automatic query through that knowledge. To automatically query the Google search interface, we could use the example mentioned in the previous section. However, there is yet another way in PHP to achieve the outcome. We can use cURL library addition in PHP to manage our query. cURL can be activated by configuring the php.ini-file inside the PHP installation directory. cURL is the currently preferred tool to use in this kind of task.

cURL has a set of operations, which can be used by default. The definitions of the operations used here are the following:

<table>
<thead>
<tr>
<th>GET_FORM_FIELD(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Find the form elements using XPath</td>
</tr>
<tr>
<td>2) Get the whole element, including the beginning and the closure HTML form tag.</td>
</tr>
<tr>
<td>3) Return a form field element in a format that can be stored in the array of the FIND_FORM_FIELDS-algorithm.</td>
</tr>
</tbody>
</table>

Table 9: GET_FORM_FIELD-algorithm
Example: Query Google Search Interface
First, we need to create a few queries by hand to observe which parameters Google uses to generate a result. The outcome of the observation is that the most important parameter is “search?q=”, which contains the search keyword. The rest of the parameters are about formatting the shown results. We will create a function that receives the search keyword as its own parameter and use cURL as the method of data collecting.

Notice: This kind of practice is against the terms of Google and is meant only to be a theoretical example of automatically querying a general search engine. To query Google automatically, the company has insisted that developers use Google’s Query API interface. (Google Query API 2011) API interfaces are explained in more detail in Section 4.2.3.

Table 10: cURL Reference
(PHP Manual 2011)

<table>
<thead>
<tr>
<th>CURLOPT_URL</th>
<th>The URL to fetch. This can also be set when initializing a session with curl_init().</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURLOPT_USERAGENT</td>
<td>The contents of the &quot;User-Agent: &quot; header to be used in a HTTP request. Can be changed to mimic different browsers.</td>
</tr>
<tr>
<td>CURLOPT_CONNECTTIMEOUT</td>
<td>The number of seconds to wait while trying to connect. Use 0 to wait indefinitely.</td>
</tr>
<tr>
<td>CURLOPT_RETURNTRANSFER</td>
<td>TRUE to return the transfer as a string of the return value of curl_exec() instead of outputting it directly.</td>
</tr>
</tbody>
</table>
function query_google($search_string) {
    $query = "http://www.google.com/search?q=" . $search_string . "&num=100&start=1";
    $curl_handle=curl_init();
    curl_setopt($curl_handle,CURLOPT_URL,$query);
    curl_setopt($curl_handle,CURLOPT_USERAGENT,"Mozilla/4.0 (compatible;)");
    curl_setopt($curl_handle,CURLOPT_CONNECTTIMEOUT,2);
    curl_setopt($curl_handle,CURLOPT_RETURNTRANSFER,1);
    $data = curl_exec($curl_handle);
    curl_close($curl_handle);

    if (empty($data)) { /* There is nothing in the query */ }
    else { /*What do we want to do with the results of the query? */ }
}

The results obtained from the example are called an intermediate page (explained more in Section 4.2). In this example, we are only interested in the returned links, so the intermediate page is sufficient for the task. Should we query a different, more specific database (for example movie descriptions) and would like to store those results in a database of our own, we would need to maneuver through the intermediate page to get to the result pages.

To get the query results into usable format, we would still need to create a parser to format the links in a way that suits our needs. The following code will work as an example. It generates an array element from every HTML paragraph element. As a result, all of the Google GUI will be in the first array element. We can then remove the first array element and print all the query HTML link results prefixed by an ascending number as shown in Figure 25. The chosen keyword was “Peikkoluola”.

In PHP, “explode” function divides a String into an array and “array shift” function drops the first element of an array. (PHP Manual 2011) We will divide the gained source code using separator “<p>” and drop the first element. The remaining of the “query_parser” function generates the ascending prefixes.
function query_parser($source_code) {
    $separated = explode("<p>", $source_code);
    array_shift($separated);

    $nro = 1;
    foreach($separated as $query_hit) {
        print "<b>" . $nro . "</b> 
        print $query_hit;
        $nro++;
    }
}

In most of the cases, we need to collect information from various sources. Naturally, the above method will not work well since we need to develop a system for each single query interface manually. The workload will increase linearly with the interface count. In the worst scenarios, we need to develop both the querying system and the parsing system by hand to suit the requirements of the deep Web resource.
5.3 Querying multiple Web databases

In the previous subsections, we have found automatically a form field in a Web site. This form field has been validated to contain the information we need by a simple algorithm. In this section, we will introduce the theory of efficient “on the fly” querying of multiple Web databases. We could use the same method as mentioned in the previous section to continue our crawling, but should the crawler system be put into more pressure, algorithms that are more efficient must be developed.

A large-scale process, such as searching for valid deep Web resources “on the fly”, faces several semantic challenges. For the system to be efficient, it must be able to retrieve “on the fly” valid query interface conditions and attributes and use them to generate queries automatically. How can the system match the found query interfaces? Each query interface must be assumed different. How does a key attribute on one interface match in another? For example, if we are searching for books, how does the key attribute author vary between the interfaces? The system must be able to understand key attribute synonym correspondences. The last crucial question is how to translate the queries between the deep Web resources? The system must rewrite the query so that the target resource can use it. (Chen-Chuan Chang, He and Zhang 2005)

![Figure 26: Translating queries between interfaces](Chen-Chuan Chang, He and Zhang 2005)
We will use book searching as our example in this paragraph and discuss in detail this “on the fly” query translation problem. Figure 26 illustrates the following definition:

“Given a specific source query $Q_s$, which is instantiated from source interface $Q_1$ and a target interface $Q_2$, query translation maps among semantically related predicates (as given by a schema matcher), and outputs a target query $Q_t$ which is close to $Q_s$ and expressible against the target interface.” (Chen-Chuan Chang, He and Zhang 2005)

In Figure 26, the translations are formed by Boolean operators, which connect the different queries as different filled forms, which are supported by the target interface $Q_2$. After the translation has been generated, the translator can execute the query through the interface and apply Boolean operators to complete the translation. The challenge in this process is that there is no predefined knowledge of the sources. As the system reaches an interface, it must process it ”on the fly”.

The theory behind the idea is that one wants to search for individual predicates. In particular, instead of searching translation space of the entire query $Q_s$, one hopes to focus on each queried predicate individually. For example, we can search for the closest translation for only the predicate “price” in Figure 26. One expects to get its correct translation by combining the closest translations for all the predicates. The theory is presented as semantic-then-syntax framework illustrated in Figure 27.

The system is divided into two parts: Semantic Mapping and Syntax Construction. The role of Semantic Mapping is to create the closest predicate translation $Q'$. Syntax Construction takes care of the expressibility of querying $Q'$ on the target interface. Since most of the deep Web resource interfaces differ from one another and the process is meant to be done “on the fly”, there is no translation knowledge customized for one specific deep Web resource. The query method used is nPKB. This is the challenge that the semantic-then-syntax framework tries to approach.

First, let us define and examine the illustration’s Semantic Mapping part: Given a source query $Q_s = c_1 \land ... \land c_l$ where $c_1 ... c_l$ are predicates, semantic mapping
generates the closest translation $Q' = c_1^* \land \ldots \land c_l^*$ by mapping each predicate $c_i$ to $c_i^*$ separately. (He, Zhang and Chen-Chuan Chang, MetaQuerier: Querying Structured Web Sources on-the-fly 2005)

Example: Expressing two Query Interfaces

Given a source query $Q_s = [\text{author: } \text{contains } \text{Tom Clancy}] \land [\text{title: } \text{contains } \text{Red Storm}] \land [\text{price: } \leq 35]$, we are given a closeness metric $C$ as minimal subsuming translation. The Semantic Mapping component will map each of the three predicates into corresponding predicates in the target interface which satisfies our closeness metric $C$. The closest translation to the target interface would be the following: $Q' = [\text{author: } \text{contains } \text{Tom Clancy}] \land [\text{title: } \text{contains } \text{Red Storm}] \land [\text{price: } \text{between } 0, 25] \land [\text{price: } \text{between } 25, 45]$. 

After the closest translation has been generated, the system continues to the Syntax Construction component of the framework. This component makes sure that the query $Q'$, generated in the Semantic Mapping component, is possible to run at the target interface $QI_2$. At the current situation, it is not possible since $QI_2$ only allows for one keyword: Author or title (see Figure 26). Currently the query $Q'$ includes both author and title keywords. The query must be transformed into Boolean expression subqueries, which satisfy the expressibility of the target interface $QI_2$. 

Figure 27: Semantic-then-Syntax framework

(Chen-Chuan Chang, He and Zhang 2005)
Transforming the query into Boolean expressions is the key to successful translation of the original query into the target query interface. The translated query $Q'$ is expressed in Boolean as $C_1 \lor C_2 \lor \ldots \lor C_m$ where each $C_i$ consists of a conjunction of predicates over the same set of attributes. In the example, we would define it as $C_1 \lor C_2$ where:

- $C_1 = [\text{author: contain Tom Clancy}] \land [\text{title: contain Red Storm}] \land [\text{price: between 0 , 25}]
- $C_2 = [\text{author: contain Tom Clancy}] \land [\text{title: contain Red Storm}] \land [\text{price: between 25 , 45}].$

Both of them have the same set of attributes (author, title, and price). Next, we must come up with how to express each $C_i$ to be usable in the target interface. The problem can be defined as follows:

“Given a query $C_i$ as a conjunctive query, how to rewrite it as a Boolean expression $D_1 \land D_2 \land \ldots \land D_r$ where each $D_j$ can be expressed using one of the virtual forms of the target interface. (He, Zhang and Chen-Chuan Chang 2005)

As the remaining implementation, one can choose from many algorithmic alternatives. The one chosen here is a greedy approach, which iteratively chooses a virtual form that maximally covers the predicates until all predicates are covered. As we transform $C_1$ we choose $F_1$ which covers the author and price and leaves title uncovered. Next, we choose $F_2$ which covers the title and price and leaves author uncovered.

- $D_1 = [\text{author: contain Tom Clancy}] \land [\text{price: between 0 , 25}]
- D_2 = [\text{title: contain Red Storm}] \land [\text{price: between 25 , 45}]

At this moment, all of our predicates are covered and we have generated translation $D_1 \land D_2$. Using the same strategy, we transform $C_2$ into translation $D_3 \land D_4$, which is:

- $D_3 = [\text{author: contain Tom Clancy}] \land [\text{price: between 25 , 45}]
- D_4 = [\text{title: contain Red Storm}] \land [\text{price: between 25 , 45}]

As a conclusion, the source query $Q_s$ is transformed into $(D_1 \land D_2) \lor (D_3 \land D_4)$. (He, Zhang and Chen-Chuan Chang 2005)

The reasons for choosing this greedy method are two-fold. First, by choosing the form with the most predicates (for example $D_2$), the generated query is the most restrictive.
The most restrictive queries result in minimal costs of running the query. An opposite example would be using $D_5$, which could only include the title attribute and its value. Second, some of the attributes in query interfaces cannot be queried by themselves. These attributes require another attribute to be sent with them into the database. The virtual forms, which would be expressible to the target interface QI2, are illustrated in Figure 28.

![Virtual Forms of QI2](image)

(Chen-Chuan Chang, He and Zhang 2005)

5.4 Mechanics of a deep Web search Engine

The deep Web search engines differ from the traditional search engines. The challenges that the deep Web search engines face, include establishing a connection to the deep Web resources, process and select the data that is the most relevant to the user and do this as fast as possible since the common user usually requires fast response to his queries.

Two different metasearch engine architectures will be defined in the subsequent sections. The first is a commercial system that does not use automated crawling in order to gather more resources. It concentrates on reaching multiple deep Web resources during a query in order to show the results. The second is a prototype system, which uses crawling to gather its sites and allows users to create queries to the index that the system creates.
5.4.1 Turbo10

Turbo10 is a commercial search engine designed to access deep Web resources in order to produce search results (http://www.turbo10.com/). The idea is that there are enormous amounts of topic-specific querying interfaces in the Web. These interfaces each produce their own search results when queried. A metasearch engine is the one to query the query interfaces. Turbo10 is such a metasearch engine (later referred to as “metasearch engine”). Metasearch engine performs the relevance ranking, topic clustering and result merging in the client Web browser, not in the server. Since there can be complications, when some of the slower deep Web resources might not respond in the required time, the metasearch engine gathers results asynchronously to the browser, after the first deep Web resource has responded with results (Hamilton 2002). This method cuts out the possibility that the user would be waiting one of the many deep Web resources to respond when all the others have already delivered results.

The mechanics of this metasearch engine can be divided into three categories: Adapter Manager, Trawler Server and Web Browser. The Adapter Manager (illustrated in Figure 30) is responsible for storing the deep Web resources. In Turbo10, the developers need to manually insert, and keep an option for others to insert, the URL of the resource that contains the query interface to their database. (Hamilton 2002) Therefore, the system uses the manual insertion technique to find the query interfaces. Hence, this metasearch engine does not try to crawl more resources on its own, but relies on that the resources are supplied to it manually. The Adapter Manager has a Form Finder –component, which will process the manually inserted URL for query interfaces (as explained in Section 5.1). After the query interfaces have been identified, the Adapter Manager
proceeds to identify the parameters required to use the search engine behind the query interface (as explained in Section 5.2). This leads to a system that identifies the differences between the query interface, intermediate page and result page. The navigation process through the intermediate page was discussed in Section 4.

![Diagram of Adapter Manager of Turbo10](http://www.dmoz.org)

Figure 30: Adapter Manager of Turbo10 (Hamilton 2002)

The Back Link Finder component retrieves the top 50 Web pages that point to a target engine's search form. The context of each back link is extracted using structural cues found in the text of each back link page. All the contexts from these retrieved links are then passed to the Centroid Finder unit. The Centroid Finder contains the terms that define the search engine. Distinctiveness is measured by using the precomputed Inverse Document Frequency (IDF) values from an inverted index (an index data structure for content, the purpose of which is to allow fast full text searches) of the Open Directory Project database. (Hamilton 2002) Open Directory Project (http://www.dmoz.org) is the largest, most comprehensive human-edit directory of the Web. The goal of the project is to provide organized frames for the Web as its scale continues to grow.

The metasearch engine will proceed to do a test query into the deep Web resource. Should the test succeed the information is passed to Result Page Finder to validate that the resource returned a valid result page. The Extractor Finder contains the operations that would allow the result page links to be parsed successfully. The Extractor locates
structural information in the result page that matches the search result list and selects the candidate links. The Extractor defines a pattern from the candidate links that allows matching of different result pages from the same engine. (Hamilton 2002)

The metasearch engine Trawler Server has a collection of Adapters that are handled by the Adapter Manager. Successfully tested adapters are moved into the Trawler Server adapter pool. When the Web browser inputs a search query, it is passed to the Trawler Server Browser Manager. The Browser Manager communicates with the Trawler, which gets the usable adapters from the Adapter Pool. The Browser Manager afterwards returns the results to the Web browser. Trawler Server and Web browser manager are illustrated in Figure 31.

The work that the metasearch engine does, when a user queries it, is not this complex. The whole system is not used in that action. The system is dividable into two parts: The gathering of resources and the delivering of resources. The gathering used the manual insertion technique and the Adapter Manager system (which is not needed when a user creates a query). In the delivering, we are using a predefined adapter pool in the Trawler server. The query is passed to the server and the server returns the correct adapters. The rest of the process is asynchronously created on the browser.

Figure 31: Trawler Server and Web Browser in Turbo10
(Hamilton 2002)
Example: Querying Turbo10 metasearch engine

The current Turbo10 metasearch engine allows searching for products from different service providers or search for Internet sites by using keywords. In this example, we observe the product search. Using the technique mentioned in Section 5.2, we can create an application, which would use Turbo10.

Since we are dealing with one query interface, we can learn how it works by conducting a search through it. Turbo10 product search has gathered results from many service providers from the United Kingdom and several international marketplaces. The search engine uses the GET method of form submittal. Users can see the parameters they used to make the query from the URL of the current search. When creating a query, the URL results as: http://turbo10.com/shop?q=example, meaning that we can alter the shop?q=value parameter to create a screen scraper of our own. In Turbo10, it would be quite easy for an automated system to detect the relevant input field from the source code as well. The input field containing the search has relevant attribute names: <input type="text" name="q" value="" class="search_field">. There is no public information about a Turbo10 API. To use Turbo10 results, for instance in a Web Mashup, one would need to use screen scraping to gather the data.

5.4.2 DeepBot

In the previous subsection, we had a system that put a lot of effort in parsing and validating the inserted deep Web resources. The DeepBot is a prototype system, the architecture of which was published in August 2006. Unlike Turbo10, the DeepBot searches for the resources itself. The main features of the system are the automated customized browsers and form detection.

The architecture of the system contains the following crawler components: A configuration manager, download manager and content manager. Route manager, a database for crawled documents, an indexer, lightweight crawler browsers and an interface for users to search from the system are each elements of the architecture but not part of the actual crawler components. The architecture is illustrated in Figure 32.
The first of the crawler components is the *configuration manager*. This component is the beginning of the system. It holds the initial routes where the crawling should begin, how deep on each site the crawling will be done, the rules of how to handle different types of documents that will occur on the path and the exclusion list of URLs. The second component is the *download manager*. This component is responsible for choosing the correct document handler for the associated document the crawler is trying to download. The last component is the *content manager*, which defines the parsers for the system. These parsers will determine if the current document is valid to be stored for indexing and they create post-storing processes that parse information from the document. (Alvarez, et al. 2006)

The systems outside the crawler engine are supporting the primary engine. The route manager holds the master list of routes that the crawlers can access. Each one of the crawlers will pick a route from the master list after the crawling process has started. The lightweight automated customized browsers have been built by using standard browser APIs. By implementing them, one can better deal with, for instance, client-side scripting, session mechanisms and redirection managing. (Alvarez, et al. 2006)

The Traditional way of creating a Web crawler would include that the links are actually stored from the HTML anchor elements. From the element, the anchor attribute is collected since it contains the URL of the link. However, DeepBot uses actual browsers to do the crawling. This has the advantage that the session identifiers can be stored and complex source code scripting can be avoided because there is an actual “click” on the link instead of getting the information out of the anchor attribute (Alvarez, et al. 2006). There is as well no need to take redirecting into account, since the lightweight browsers get redirected just as the normal user browsers would do. The navigation sequence in the lightweight browsers is implemented by the NSEQL programming language, which was created to be used with the browsers.

The content manager component of the crawler engine holds a Form Analyzer. This subcomponent processes form fields found on Web sites. The action is threefold. First,
for every domain the system tries to match its attributes with the fields of the form by using visual distance and text similarity heuristics not further introduced in this thesis. Second, from the output of the first action, the relevance of the form field is calculated in respect of the domain. Finally, if the form is found relevant, the crawler will use the form for queries. (Alvarez, et al. 2006)
6 CRAWLING THE DEEP WEB

The previous sections of this thesis contained mostly theoretical tools and techniques to crawl the deep Web. In this section, I will design and implement a Web crawler and issue a deep Web query to the created system. The tasks of the project are threefold. The primary task is to query a deep Web database without creating site-specific code or using a designated API for the target system. The crawler system is not meant to work on every possible search interface, hence it will be released as a prototype and as a work in progress. The primary task is achieved when at least one deep Web resource replies to the queries. Creating a system that achieves the primary task will give us insight to the deep Web in practice. The secondary task is to create a functional Web crawler that searches for indexable HTML documents. At this point, there will be no ranking system or a query interface for users to search from the index. The secondary task will cover the surface Web procedures in practice. The tertiary task is to gather statistics with the system. Several of the system functions are only created to gather statistics from the Web crawler’s actions.

The system is twofold: we need a crawling system to gather Web sites and we need a crawling application to handle the querying of deep Web resources (architecture introduced in Section 2.4). The other mandatory system components are the crawler configuration and the index database. This prototype architecture is illustrated in Figure 33 and will be referred as the Scavenger from this point forward.

![Figure 33: Prototype Crawler Architecture](image-url)
The Scavenger *crawling system* class (Section 6.1) contains the functions to connect and download the HTML document, find all the anchors in the HTML document and gather the metadata of the HTML document. The *crawling application* class (Section 6.2) analyzes the HTML documents for form fields, sets a categorization on every found form field, finds out if the found form field uses POST or GET method on submit and initiates a query to an indexed form field. The *crawler configuration* (Section 6.4) is responsible for running the Scavenger system. It also manages the crawling strategies (previously defined in Section 2.4). The index database is modeled in Section 6.3. After the Scavenger system has been defined, the results of the system launch will be listed in Section 6.5.

### 6.1 Scavenger Class: The Crawling System

The project begins with designing a functional Web crawler. We have a need of finding a search interface so that we can query it (introduced in Section 5.1 and Section 5.2). There were two ways to achieve this: Allow human insertion of search interface Web sites to the index, or crawl the surface Web to find the search interfaces. In this project, the crawling of surface Web was chosen as the method of finding the search interfaces.

<table>
<thead>
<tr>
<th>Crawling System</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ sourceCode : String</td>
</tr>
<tr>
<td>+ anchors : Array()</td>
</tr>
<tr>
<td>+ currentUrl : String</td>
</tr>
<tr>
<td>+ connectionTime : Float</td>
</tr>
<tr>
<td>+ htmlSize : Integer</td>
</tr>
<tr>
<td>+ htmlTitle : String</td>
</tr>
<tr>
<td>+ htmlDescription : String</td>
</tr>
<tr>
<td>+ htmlKeywords : String</td>
</tr>
<tr>
<td>+ currentDate : Date()</td>
</tr>
<tr>
<td>+ Document_downloader(URL in String) : URL as String</td>
</tr>
<tr>
<td>+ Anchor_finder(Source code in String) : Array of Strings</td>
</tr>
<tr>
<td>+ Metadata_gatherer(Source code in String)</td>
</tr>
</tbody>
</table>

Figure 34: UML of Class: The Crawling System
Figure 34 illustrates the UML (Unified Modeling Language) description of the Crawling System class. There are nine attributes and four functions in the class. The description of each attribute is listed in Table 11. The rest of this subsection will define all the crawling system class functions.

| + sourceCode: | Contains the complete source code of an HTML document as a simple package of data. The source code is stored as a String. |
| + anchors: | Is a String array containing all the HTML anchors and their attributes and values that the current HTML document has. |
| + currentUrl: | A String that contains the current URL that the crawling system is processing. |
| + connectionTime: | The time that was taken while the system connected to the site. |
| + htmlSize | The total size of the HTML source code in bytes shown as Integer. |
| + htmlTitle | The String representation of the HTML Title element. |
| + htmlDescription | The HTML meta element description stored as a String. |
| + htmlKeywords | The HTML meta element keywords stored as a String. |
| + currentDate | The date when the crawler indexed the document. |

**The constructor of the Crawling System class**

The constructor is given the URL of the site to be crawled and the crawling strategy as parameters. First, the constructor initiates the class variables. The initiation is done by first collecting the source code of the HTML document. This is done by calling the *document downloader* function to process the URL. After the HTML document has been stored in the class variable, the *anchor finder* function will use XPath to gather all the anchors from the HTML document. The *meta gatherer* function will extract the meta elements from the HTML document also using XPath. The constructor will also create a timestamp for the database.
After the functions have completed processing the HTML source code, the construct will clean the String class variables. The cleaning means replacing all the invalid characters that might cause problems when storing the String into the database (defined in Section 3.4).

Should there be HTML source code returned by the document downloader function, the construct will determine the next action depending on the crawling strategy parameter. In the end, the construct will store the document and all the class variables into the index database if the crawling strategy allows that.

**Function: Document Downloader**

This function is responsible for establishing the connection to a remote server. The connection is created by using cURL (introduced in Section 5.2). The function receives the site to connect as a parameter. The cURL options used in this function are the following: Connect to the parameter site, timeout if no connection is established in five seconds, return the connected document as a String and create its own headers. If the connection is successful and the function has stored the source code of the connected site, three other cURL operations are run and stored in the crawling system class variables: The current URL, the size of the HTML document in bytes and the time that was used to connect to the site. If the system did not manage to download the source code, the system will not proceed any further with the URL and continues according to the crawling strategy.

**Obey the Robots Exclusion Standard**

To be aligned with the Robots Exclusion Standard (RES) (introduced in Subsection 3.3.2), the Scavenger has integrated RES into its systems. Integration has been implemented by creating an array of the excluded links and the array will deny indexing of the current anchor if it has been set to be denied. However, it will not check if one server has linked to a denied section of another server. It will only respect the RES from the Web server’s local exclusion list of the currently pending HTML document anchors.
**Function: Anchor Finder**

When the anchor finder function is called, there is definitely some source code available for the function to use. The source code is given as a parameter to the function. The anchors are found by using the DOM structure and XPath (previously in Section 4.2.2).

```php
$DOM = new DOMDocument();
@$DOM->loadHTML($htmlDocument);

$xpath = new DOMXPath($DOM);
$html_links = $xpath->evaluate("/html/body//a");
```

The XPath query finds all the links that are in the HTML document. Should there be any elements in the `html_links` array variable, the function will proceed to iterate the array and extract all the `href`-attribute values that the anchor elements have. These attributes contain the URLs that the Scavenger wants to collect and crawl later. Any duplicate links are deleted from the array at this point and the function returns the unique listing of the HTML anchors.

```php
if(count($html_links) != 0) {
    $array_of_links = array();
    foreach ($html_links as $link) {
        $href = $link->getAttribute('href');
        array_push($array_of_links, $href);
    }
    $array_of_links_no_duplicates = array_unique($array_of_links);
    return $array_of_links_no_duplicates;
}
```

**Metadata Gatherer**

The function works very similarly compared to the *anchor finder* function. Only the XPath queries are different to collect the correct HTML components. The function begins with loading the source code and creating an XPath object. The XPath queries to collect the requested elements are the following:

- **Title-element:** `xpath->query("//title")`
- Meta-element (keywords): $xpath->query("//meta[@name='keywords']");
- Meta-element (description): $xpath->query("//meta[@name='description']");

There are different XPath functions in PHP to collect the values of these queries. To collect the value of an HTML element, we need to call the `nodeValue` function ($title->nodeValue;). To save the attribute value of an HTML element, we need a different function, `getAttribute` ($key->getAttribute('content');) (PHP Manual 2011)

6.2 Scavenger Class: The Crawling Application

The crawler application is responsible for characterizing the deep Web resources, which the crawler system detects and indexes. At this point, the crawler application class does not have any attributes since the class is used differently from the crawler system. There are four functions in this class. UML illustration of the class is in Figure 35.

<table>
<thead>
<tr>
<th>Crawling Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Find_form_fields()</td>
</tr>
<tr>
<td>+ Form_field_categorization()</td>
</tr>
<tr>
<td>+ Form_field_method()</td>
</tr>
<tr>
<td>+ Query_form_field()</td>
</tr>
</tbody>
</table>

Figure 35: UML of Class: The Crawling Application

**Function: Find Form Fields**

The function observes all form fields from the indexed HTML source code and indexes them separately. The function calls attributes from the database, including the already indexed source code. Once the database attributes have been made into function variables, XPath will be used to evaluate the DOM structure. The XPath `evaluate` and `query` functions can be used to retrieve the HTML elements. The difference is that the `evaluate` function returns a count of the found items (PHP Manual 2011). The XPath query to search for form fields is the following:

- $xpath->evaluate("/html/body//form")
Should the current HTML source code include any form fields, the whole document will be inserted into an array using line break as a separator for each array element. For every array element, the function will search for the beginning tag of the form field element. If the beginning tag is found, the function will search the end tag of the form field. This will be done, as many times as there are elements in the array. The complete form field will be stored in a database upon detection and narrowing. This algorithm was first defined in Section 5.1.

**Function: Form Field Categorization**

Since the aim of the Crawler Application is to search for form fields that we can query for results, we are only interested in part of the form fields. Form fields can be categorized into several different types. We are only interested in the type that allows actual keyword searching. Hence, we will create a categorization system. The system will prevent the querier from inserting queries to a wrong category of form fields that do not return any relevant information.

The categorization begins by listing all the current form fields that have been indexed, but have not yet been categorized. Each of the form fields will be inserted in an array. The array elements will be the lines of the form field since the element separator is the line break. The function will go through each line of HTML and search for specific predefined keywords, therefore the function uses a PKB-method of predicate discovery (defined in Section 3.5.1). Once a specific keyword has been identified, the system categorizes the form field in the database.

**Function: Form Field Method**

This simple function will collect the information whether the form fields are using GET or POST method on submit. Similar to the other functions, the process begins by listing the form fields in the database. The listing is then run through an iterator, which searches for the method-attribute of the form field using XPath. Once found, it will extract the value of the attribute and increase an integer value of the corresponding method variable.
Function: Query Form Field

This function is the most important in the system considering the topic of this thesis. We wanted a system that could extract information from a deep Web resource. This function is still very simple. It will not take into account the diversity that the Web offers. Since we are using a far more trivial system than the one discussed in Section 5.3, I am content with one result from a deep Web resource to be able to conclude the project.

The form fields have been indexed by the find form fields function. They have been categorized depending on their content by the form field categorizer function. We can now list the form fields, which contain searching predicates as a list of possible search interfaces. Once listed, the action attribute of the form field is the next target. This attribute contains the URL of the script the form field will call once it has been submitted. The function extracts the action attribute by using the familiar XPath technique:

- Collect form fields: $xpath->query("//form")
- Collect action attribute: $action=$field->getAttribute("action").

After the action attribute has been extracted, the function proceeds to list all the input elements in the form field. These elements contain different type attributes from which the text type is the most common in keyword search interfaces. The function uses XPath to collect the names and values of the input elements to form an automated query to submit.

The data to be posted is as follows: $post_data[$input_name] = "$input_value". For every element in the $post_data array, the function will construct a suitable query. The query is constructed by imitating the submitting of the form field in the URL when the form would be submitted by a GET method by linking the listed input elements with an “&” and pointing their values as “=” creating an automated query of: Input1=value1&input2=value2& ... &input(n)=value(n).

The system uses XPath to gather all the input element names and values. The different
input element types will be automatically included in the construction of the query.

Once the function has constructed the query, it needs to be directed to a search interface. The function initializes cURL for the task. The options that the cURL uses in this function include:

- CURLOPT_TIMEOUT, to disconnect after several seconds if there is no reply.
- CURLOPT_USERAGENT, to announce that it is a bot instead of a human.
- CURLOPT_RETURNTRANSFER, to get the possible query results as a String.
- CURLOPT_POSTFIELDS, to post the constructed query into the interface.

The function displays query results, which are successful, to the Web crawler engine interface since the aim of the project is completed should the system get a result.

6.3 MySQL Database: The Index

The database model for the index is twofold: An ER-model (Entity Relationship), which is transformed into a relational model. The index is not a complicated database structure in this project. The main function of the database is simply to store the crawler results in the obvious way and allow the crawler application to categorize and analyze the found HTML form fields.

The ER-model is a popular high-level conceptual data model. The model is used for the conceptual design of database applications. The model describes data as entities, relationships and attributes. (Elmasri and Navathe 1994) The index database has three entities: Pages, Forms and Links. Two of the three entities are related to each other, in other words, Pages to Forms and Pages to Links. All entities have attributes. These attributes will contain the data that the class functions extract from the HTML source code or directly from the cURL connection. The primary attributes are underlined. The ER model of the index database is illustrated in Figure 36.
The relational model represents the database as a collection of relations. The primary key is underlined and foreign key dependencies are shown as an arrow. When a relation is thought of as a table of values, each row in the table represents a collection of related data values. (Elmasri and Navathe 1994) The relational model obtained from the ER model is illustrated in Figure 37. The database is constructed using these models.

### 6.4 The Web Crawler System

The main class of the Scavenger system is the Web Crawler. It is possible to configure the crawler to use all or a subset of the system components by changing Boolean variables. The configuration supports changing the starting location of the crawling as well as the crawling strategy. These must be changed from the Web crawler source code.
directly, since at this point, there is no administrator GUI (Graphical User Interface) to allow the changes to be made with a Web browser.

The rest of the system is divided into three categories: Starting the crawling system, initiating the crawling application and using the crawling strategy. The crawling system is called with a twofold scenario: If the database is currently empty, the class will begin the crawling at the predefined starting location. Should there be already rows in the database, the class will check what crawling strategy has been pointed out to carry on and use that strategy to continue crawling.

The Crawling Application is more straightforward than the Crawling System. Once the Crawling System has shut down, it is the Crawling Applications turn to initialize. The form field functions are called one by one and the results are shown in the engine print flow.

The strategies are actually related mostly to SQL queries. The crawling can be changed by retrieving the database list in a different way. In Breadth-first crawling, the system collects all the links from all the pages and continues to collect all the links from all the pages hence creating a list. The Depth-first crawling requires some additional coding in the Scavenger. The aim of the depth-first is to index the current page and the links of the page. Once this is completed, the depth-first crawling finds a link that is not pointing at the same domain as the current Web site. The crawling will continue to that site, postponing the crawling of all the other links. The Recrawling strategy is a simple SQL command that retrieves the list of URLs, which have been indexed, and point these out for the crawling system to index again. The deep Web strategy in this project means that the system successfully queries a search interface that it has indexed using the crawler system.

The Crawling engine contains debug and status information. Before every new major function call, the system will print out the current timestamp and the current action it is carrying out. Once the action is completed, the system will print out another timestamp and a notice of completion. The engine print flow is illustrated in Figure 38.
Results of the Scavenger Crawl

The crawling system was run on the 14th of June 2010 at 10:00 and lasted for 24 hours. The crawling system consisted of four crawling computers and one crawling server. The server had an index database for each of the crawlers. The system model and hardware can be seen in Appendix 1. The network connection was tested before the crawl. The actual download rate of the network connection was 45Mb/s and the actual upload speed was 24Mb/s. The total bandwidth usage during the 24 hour period for the crawlers and the server combined were 8 Gb for downloading and 6.5 Gb for uploading. Each of the crawlers was given a different starting location. All the starting locations were Finnish servers.

During the 24 hour crawl, a total of 1 634 484 Internet links were found. The amount of downloaded Internet pages was 100 416. The crawl showed that from this sampling, the average amount of HTML anchors on each site is 16. The hourly download rate was approximately 75 700 Internet links and 3 700 Internet pages. The amount of SQL queries to the crawler server was 7 150 312. The statistics were gathered from each of the separate databases designed for the crawlers. The amount of page and link crawling for each crawler is illustrated in Figure 39.
The crawlers had the same hardware and the same bandwidth available for them. However, they still yielded very different results. Crawlers #2 and #3 functioned well and travelled the Web in a similar way. The last two crawlers were the curiosity of the project. Crawler #1 indexed far less links than #4, however it still had indexed more Web pages. The paths that the crawlers are given determine mainly the results that they will receive. What lies on the path however can only be taken into account by programming functionality and different behavior for such events. The fourth crawler malfunctioned to an unknown error and ended its indexing during midnight. The crawler also failed to reset the crawler machine automatically to continue crawling. A failsafe system is mandatory for crawling machines to function properly. The first
crawler had no difficulties during its path and its bandwidth usage was similar to the others. Still it succeeded crawling the least amount of links even when the fourth crawler malfunctioned halfway.

The total size of the index is 4.5Gb. The average HTML page size can be calculated from the stored HTML sizes in the database. The average HTML document size in this sampling was 28.98 kilobytes.

The previous statistics were generated between the crawling system and the index database. After the 24 hours had passed, the crawling system was shut down to stop generating additional data. The crawler application was run to analyze the indexed contents. The application detected from the loaded Web pages a total of 29 819 form fields. Based on this sampling, an Internet site has a 27 % chance to contain a form field. The form field categorizer looked for the form methods and revealed that 20 576 was using the POST method and 8 820 the GET method meaning that 69 % of the detected form fields were using POST. The categories of the form fields were threefold: Login-based form fields, search-based form fields and all the other types of form fields. The form field categorization is illustrated in Figure 40.

![Form-field Categorization](image)

**Figure 40: Scavenger statistics: form field categorization**
At this point, the Scavenger system had indexed a small part of the surface Web and found some entrances to the possible deep Web resources. The crawler application continued from this point forward to work with the found deep Web entrances. The total amount of form fields categorized to search is 9,632 as seen from the Figure 40. This is the amount of objects, with no predefined interface logic, that the crawler application will try to query. At the current situation, the system has no usable knowledge of what kind of search interfaces the system has actually indexed. The final phase of the Scavenger system runs a tool reporting, how many of the possible deep Web resources actually sent back query results to our system. If even one deep Web resource gave actual query results, the project is concluded.

The reason for concluding the project after one result is simple. We did not want to cause heavy load on servers with our system, since the system is proven to be working if a single result is gained.

Since there were many search interfaces found and indexed, the crawler application was limited to try to query the interfaces in hope of succeeding to receive a response and query results. The results were parsed to contain only the kind of anchors that had the same keyword as in the actual query. This eliminates the possibility that the result page has site-specific headers or footers that would otherwise be included in the query results.

The querying of the deep Web resources gained its first result at http://www.ask.com and the system ended the iteration. The achieved result will not undergo any parsing system since the Scavenger project was restricted. The result is the whole source code that the successful query produces. The result that the system produced can be seen in Appendix 2. A deep Web resource answered to the Scavenger’s call and therefore the project is successful. The various parts of the Scavenger system source code can be found in Appendixes 3 - 5.
6.6 Project conclusion

Creating an efficient search engine is a long-haul project for the surface Web alone. The amount of hardware and bandwidth requirements for large-scale use is a problem for a fledgling search engine. The crawling itself would not require considerable amounts of bandwidth if the amount of non-distributed crawling machines were below twenty. The main bandwidth usage would be due to the service that the users would query to get the results from the indexed database.

From the software point of view, the crawler system must be able to deal with many obstacles, since it is set to run through the unknown. Two of the Scavenger project crawlers did not function properly. The reasons for the malfunctions were a server time out and a crawler trap. The crawlers must be created so that they would automatically resume their purpose, should the system return an error that would terminate the crawling process. This would have allowed the other of the malfunctioned crawlers to resume its path. To avoid crawler traps, we would need to set stricter limits how to conduct a crawl in the current domain. Since the Scavenger system had no real limitation concerning a domain, the other malfunctioned crawler was stuck in one domain for the total running time of the project. It had started to index every single blog post and blog update that it could. The server logs of the crawlers also point out that one of the properly functioned crawlers was about to do the same since it were caught during the last two hours of the crawl to do the same. To avoid trapping one would need to generate a depth system to deny the crawler to pursue a site too deep or other logic to resume crawling on a different domain unless the whole site is actually wanted to be indexed.
7 CONCLUSION

It has been noted before that the basic architecture of a Web crawler is quite simple. To answer the question why the architecture becomes complicated and requires a lot of research is a simple one as well. It is the scale of the surface Web. To define a crawler, all its engineering challenges are related to the scale. The crawler’s complexity becomes visible in a high-performance situation like offering a huge database to thousands of users while achieving satisfying speed.

In order to keep pages up-to-date in an example of a ten billion Web page scale, the crawler is required to download over 4000 pages in a second. This brings forth the requirements of hardware. The leading search engine companies have many clusters of computers designed for the crawling and storing of data. The crawler must be distributed over multiple computers that are called crawling machines. These machines must be able to pursue multiple downloads in parallel. (Najork 2005) In addition to this, the machines must not overload a Web server they are crawling with requests that would ultimately deny the service that the single Web server is trying to provide. The denial-of-service attack (DoS) is generated in a similar way. A malfunctioning crawler might initiate DoS attack on a server should it, for example, get stuck in a loop on a target server and does not have a reliable exception handling.

The technical limitations of automatically accessing Web databases are the main reasons why it is difficult to access the deep Web. The scale is the problem here as well. The first task is to find a deep Web query interface. Should the need be in only several known deep Web resources, the system can be built to rely on either the server’s presentation layer (screen scraping) or data layer directly (API) in which case the first task can be ignored. To complete the first task one needs a Web crawler to index the surface Web. After a part of the surface Web has been indexed, one needs to parse the indexed data to find form fields from the HTML documents. Since not all of the form fields are doorways to deep Web resources, the system should discard, with automated logic, the irrelevant form fields from further processing. The logic can take advantage of techniques like the path addressing language (KAPath, XPath) and screen scraping.
After the previous subtasks are completed, we have a system that contains deep Web resource query interfaces in its index.

The second task is to query the found deep Web resources with a query that could return a result that the system is interested to index. This can be achieved by using programming language specific function libraries (cURL) or programming language specific features (built in methods). Should the need be to query multiple search interfaces on-the-fly we need to create a system that can translate a query for all the interfaces in the context. This can be achieved by a query translator using for example the semantic-then-syntax framework (Chen-Chuan Chang, He and Zhang 2005).

After the query has been submitted, the next step leads either to an intermediate page or directly to the result page. The outcome is dependent on the type of the deep Web resource. The interfaces can be divided into two groups. First, the query interfaces (for example Google) offer a possibility for their users to search with keywords. This ultimately leads to an intermediate page that contains a list of possible results for the user. The intermediate page is always a list of results that must be followed further to reach the result page. Second, the interfaces that offer less dynamic possibilities for a search will result directly to a result page. These kinds of interfaces rarely allow keyword searches. Instead, they work through highly hierarchical selection menus that are generally easy to automate. Once all the selections have been made, the result will be shown.

The statistical gap from the previous research of 2001 and 2005 was given a nonscientific estimation in Figure 6. This was enough to answer the possible current deep Web scale in the scope of this thesis. The sections answered the research questions set for this thesis and left open a few paths for future work, including the efficient querying of multiple Web databases, the scientific scale analysis of the current deep Web and the possibility to continue the Scavenger system project.
8 REFERENCES


Chen-Chuan Chang, Kevin, Bin He, and Zhen Zhang. "MetaQuerier over the Deep Web: Shallow Integration across holistic sources." Computer Science Department, University of Illinois, 2005.

Dogpile. "Different Engines, Different Results: Web searchers not always finding what they're looking for online." A research study by Dogpile.com in collaboration with researches from Queensland University of Technology and the Pennsylvania State University, 2007.


Wu, Ping, Ji-Rong Wen, Huan Liu, and Wei-Ying Ma. "Query Selection Techniques for Efficient Crawling of Structured Web Sources." 2006.


## APPENDIX 1: SCAVENGER HARDWARE AND MODEL

### Crawlers Hardware (Identical)

<table>
<thead>
<tr>
<th>Model</th>
<th>HP Compaq DC 7800p Small Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel® Core™ 2 Duo, E6550 @2,33GHz (x86)</td>
</tr>
<tr>
<td>RAM</td>
<td>2 000 MB</td>
</tr>
<tr>
<td>Hard disk</td>
<td>120 Gb</td>
</tr>
</tbody>
</table>

### Crawler Server

<table>
<thead>
<tr>
<th>Model</th>
<th>HP Compaq DC 7900 Small Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel® Core™ 2 Duo, E8400 @3,00GHz (x64)</td>
</tr>
<tr>
<td>RAM</td>
<td>4 000 MB</td>
</tr>
<tr>
<td>Hard disk</td>
<td>1 TB</td>
</tr>
<tr>
<td>WAMP</td>
<td>Apache 2.2 / MySQL Server 5.5 / PHP 5.3.6</td>
</tr>
</tbody>
</table>

![Figure A: The Scavenger Hardware Model](image)

---

92
APPENDIX 2: THE DEEP WEB RESULT

[11:29:45]**** Crawling Application - System Initiated ****
[11:29:45]"Crawling Application - Querying Form-fields..." ****

http://www.ask.com

POST DATA: Array ( [q] => TEST )
POST DATA: Array ( [q] => TEST [search] => TEST )

POSTSTRING=q=TEST&search=TEST&qsc=TEST&o=TEST&l=TEST

http://www.ask.com/web


Web Images News Videos More

Ask

TEST

Test.com Web Based Testing and Certification Software v2.0
Easily Author and Administer your own Training Content, Tests, and Certification Programs Online.
Test.com is Web Based Software.
test.com/

Exam
Discshop.fi, istä löydät elokuva ruoan & edullisesti. Osta heti!
www.discshop.fi/DVD

Speakeasy - Speed Test
Test your Internet Connection with Speakeasy's reliable and accurate broadband speed test. What's your speed?
www.speakeasy.net/speedtest/

Related Searches
Fun Tests
Love Tests
Personality Test
Personality and IQ Testing
Personality Quiz
Career Test
Stupid Test
Insanity Test

...
APPENDIX 3: CRAWLING CONFIGURATION

<?php

/* configure */
include("CrawlingSystem.php");
include("CrawlingApplication.php");
$startingLocation="http://www.peikkoluola.net";
$CrawlingSystem=true;
$CrawlingApplication=true;
$CrawlingDeepWeb=false;

/*strategies are:
* 1) Breadth-first
* 2) Depth-first
* 3) Recrawl
* 4) Deep-web
*/
$crawlingStrategy="Breadth-first";
/************************************************/

if($CrawlingSystem) {
    $systemStartTime=date("H:i:s");
    print("<b style='color:red'>[" . date("H:i:s") . "]**** Crawling System - System Initiated - Strategy: $crawlingStrategy ****</b><br ">");
    $crawling_strategy=crawling_strategy($crawlingStrategy);
    $result=mysql_query($crawling_strategy) or die(mysql_error());
    $result_count=mysql_num_rows($result);
    print("<b style='color:red'>[" . date("H:i:s") . "]**Crawling System : Query for links Completed (Links: $result_count) ****</b><br ">");

    /*********************************************/

    /*********************************************/
}
if(mysql_num_rows($result) == 0) {
    print("<b style='color:red'>[" . date("H:i:s") . "]**Crawling System : Index is empty, begin at predefined location ****</b><br />
    $crawl = new CrawlingSystem($startingLocation, $crawlingStrategy);
    //var_dump(get_object_vars($crawl));
}
else {
    print("<b style='color:red'>[" . date("H:i:s") . "]**Crawling System : Index is NOT empty, begin at indexed links ****</b><br />
    while($row=mysql_fetch_array($result)) {
        print("<b style='color:red'>[" . date("H:i:s") . "]**Crawling System : Iterating through indexed links... ****</b><br />
        $crawl_this_site=$row[0];
        $crawl = new CrawlingSystem($crawl_this_site, $crawlingStrategy);
        //var_dump(get_object_vars($crawl));
    }
    print("<b style='color:red'>[" . date("H:i:s") . "]**Crawling System : Iteration completed! ****</b><br />
}
print("<b style='color:red'>[" . date("H:i:s") . "]**** Crawling System - System Shutdown ****</b><br />
$systemShutdownTime=date("H:i:s");
$crawlingTime=strtotime($systemShutdownTime) - strtotime($systemStartTime);
print("<b style='color:red'>-- Crawling took seconds: [$crawlingTime]</b><br /><br /><br />
if($CrawlingApplication) {
    $system = new CrawlingApplication();
** Crawling Application - System Initiated ****</b><br />

if(!$CrawlingDeepWeb) {
    print("<b style='color:green'>[" . date("H:i:s") . "]** Crawling Application - Finding unique form-fields in index... ****</b><br />
    $system->find_form_fields();
    $sql="SELECT id FROM forms";
    $result=mysql_query($sql) or die(mysql_error());
    $result_count=mysql_num_rows($result);
    print("<b style='color:green'>[" . date("H:i:s") . "]** Crawling Application - Finding unique form-fields in index Completed! (Forms: $result_count ) ****</b><br />
}

if(!$CrawlingDeepWeb) {
    print("<b style='color:green'>[" . date("H:i:s") . "]** Crawling Application - Categorizing Form-fields... ****</b><br />
    $system->form_field_categorization();
    print("<b style='color:green'>[" . date("H:i:s") . "]** Crawling Application - Categorizing Form-fields Completed! ****</b><br />
}

if(!$CrawlingDeepWeb) {
    print("<b style='color:green'>[" . date("H:i:s") . "]** Crawling Application - Querying Form-fields methods... ****</b><br />
    $system->form_field_method();
    print("<b style='color:green'>[" . date("H:i:s") . "]** Crawling Application - Querying Form-fields methods Completed! ****</b><br />
}

if($CrawlingDeepWeb) {
    print("<b style='color:green'>[" . date("H:i:s") . "]** Crawling Application - Querying Form-fields... ****</b><br />
    $system->query_form_field();
    print("<b style='color:green'>[" . date("H:i:s") . "]** Crawling Application - Querying Form-fields...
function crawling_strategy($strategy) {
    switch($strategy) {
    case "Breadth-first":
        $new_strategy="SELECT link FROM links";
        break;
    case "Depth-first":
        $result=mysql_query("SELECT id FROM links LIMIT 1") or die(mysql_error());
        $result_count=mysql_num_rows($result);
        if($result_count == 0) {
            $new_strategy="SELECT link FROM links";
        } else {
            
        
        }
    case "Recrawl":
        
    
}
$new_strategy="SELECT url FROM pages";
break;

case "Deep-web";
   // käy läpi form fieldejä sisältävät sivustot... ja... jotain..
   $new_strategy="SELECT url FROM forms";
   break;

default:
   $new_strategy="Breadth-first";
}

return $new_strategy;
?>
APPENDIX 4: CRAWLING SYSTEM

<?php

include("connectDB.php");

/**
 ** Crawling System
 ** Component of a Prototype Web Crawler
 ** 28.5.2011 Jyri Lehtonen
 **
 ** This component is tasked to retrieve all HTML links (anchors)
 ** from a designated web page. Should a Depth be other than 1,
 ** it will continue through the found links to collect the
 ** links that are on those documents also.
 **
 */

Class CrawlingSystem {

    public $sourceCode;
    public $anchors;
    public $currentUrl;
    public $connectionTime;
    public $htmlSize;
    public $htmlTitle;
    public $htmlDescription;
    public $htmlKeywords;
    public $currentDate;

    public function __construct($beginningSite, $crawlingStrategy) {
        $this->sourceCode = $this->document_downloader($beginningSite);

        if(!empty($this->sourceCode)) {
            $this->anchors = $this->anchor_finder($this->sourceCode);
            $this->metadata_gatherer($this->sourceCode);
            $this->currentDate = date("Y-m-d");
        }
    }
}
if($crawlingStrategy == "Depth-first") {
    preg_match('/^(?:http://)?([^/]+)@i', $this->currentUrl, $matches);
    $this->currentUrl = "http://" . $matches[1];
}

if($crawlingStrategy != "Recrawl") {
    $sql_page='INSERT IGNORE INTO pages (url, title, keywords, description, visited, size, time, source)
VALUES("" . $this->currentUrl . ",", "," . $this->htmlTitle . ",", "," . $this->htmlKeywords . ",", "," . $this->htmlDescription . ",", "," . $this->currentDate . ",", "," . $this->htmlSize . ",", "," . $this->connectionTime . ",", "," . $this->sourceCode . ");
}

if($crawlingStrategy == "Recrawl") {
    $sql_page='INSERT IGNORE INTO pages (url, title, keywords, description, visited, size, time, source)
VALUES("" . $this->currentUrl . ",", "," . $this->htmlTitle . ",", "," . $this->htmlKeywords . ",", "," . $this->htmlDescription . ",", "," . $this->currentDate . ",", "," . $this->htmlSize . ",", "," . $this->connectionTime . ",", "," . $this->sourceCode . " ON DUPLICATE KEY UPDATE title="" . $this->htmlTitle . "", keywords="" . $this->htmlKeywords . ", description="" . $this->htmlDescription . "", visited="" . $this->currentDate . "");
}
size="" . $this->htmlSize . "", time="" . $this->connectionTime . ", source="" . $this->sourceCode . "";

mysql_query($sql_page) or die(mysql_error());

$sql = "SELECT max(id) FROM pages";
$result=mysql_query($sql) or die(mysql_error());
while($row=mysql_fetch_array($result)) { $currentPageID=$row[0]; }

/* Robots exclusion standard */
preg_match('^(?:http://)?([^/]+)@i', $this->currentUrl, $rootUrl);
$rootUrl = "http://" . $rootUrl[1];
'excluded_links' = $this->obey_robots_exclusion($rootUrl);

foreach ($this->anchors as $alink) {
    if(!in_array($alink, $excluded_links)) { //exclusion std
        //print "The current link: $alink - if denied should not be seen<br>";
        if($alink == "/" | $alink == ".") { $alink = $this->currentUrl; }
        if(substr($alink, 0, 1) == "?") { $alink = $this->currentUrl . $alink; }
        if(substr($alink, 0, 1) == "/") { $alink = $this->currentUrl . $alink; }

        if(substr($alink, 0, 5) == "http:" ) {
            $sql_links='INSERT IGNORE INTO links (page, link) VALUES("' . $currentPageID . ", " . $alink . ")';
            mysql_query($sql_links) or die(mysql_error());
        }
    }
}

/*
 * Parameter: The site from which the crawling will begin, how many links to follow?
 * Return: the source code of the current site being crawled
 */
* Uses cURL to gather data from internet sites (Features):
* - Will drop connection, if no respond in 5 seconds.
* - Mimics as if the viewer of the site is using Mozilla Firefox.
* /
(public function document_downloader($beginningSite) {
  $curlHandle=curl_init();
  curl_setopt($curlHandle, CURLOPT_URL, $beginningSite);
  curl_setopt($curlHandle, CURLOPT_CONNECTTIMEOUT, 0);      //The number of seconds to wait while trying to connect. Use 0 to wait indefinitely.
  curl_setopt($curlHandle, CURLOPT_TIMEOUT, 5);      //The maximum number of seconds to allow cURL functions to execute.
  curl_setopt($curlHandle, CURLOPT_RETURNTRANSFER, true);
  curl_setopt($curlHandle, CURLOPT_USERAGENT, "Mozilla/4.0 (compatible;)");
  $htmlDocument = curl_exec($curlHandle);
  if ($htmlDocument) {
    $this->currentUrl = curl_getinfo($curlHandle, CURLINFO_EFFECTIVE_URL);
    $this->htmlSize = curl_getinfo($curlHandle, CURLINFO_SIZE_DOWNLOAD);
    $this->connectionTime = curl_getinfo($curlHandle, CURLINFO_CONNECT_TIME);
    return $htmlDocument;
  } else { /*what to do when site does not exists */
    curl_close($curlHandle);
  }
}

// returns an array of excluded links from this site.
public function obey_robots_exclusion($root_url) {
  $root_robots=$root_url . "/robots.txt";
//print $root_robots;

$curlHandle=curl_init();
curl_setopt($curlHandle,CURLOPT_URL,$root_robots);  
//The number of seconds to wait while trying to connect. Use 0 to wait indefinitely.
curl_setopt($curlHandle,CURLOPT_CONNECTTIMEOUT,0);
curl_setopt($curlHandle,CURLOPT_TIMEOUT, 5);  //The maximum number of seconds to allow cURL functions to execute.
curl_setopt($curlHandle,CURLOPT_RETURNTRANSFER,true);
curl_setopt($curlHandle,CURLOPT_USERAGENT, "ScavengerBot/1.0 Temporary Scraping");
$robotsExclusion = curl_exec($curlHandle);
$returnCode = curl_getinfo($curlHandle, CURLINFO_HTTP_CODE);

if($returnCode != 404) {
    $robotsExclusion=explode("Disallow:", $robotsExclusion);
    array_shift($robotsExclusion);  //get rid of possible text
    foreach ($robotsExclusion as & $disallowed) {
        $disallowed = trim($disallowed);  //get rid of whitespace
        $disallowed = $root_url . $disallowed;
    }
    return $robotsExclusion;
} else { /* The robots.txt does not exist */
    $emptyArray=array();
    return $emptyArray;
}
curl_close($curlHandle);
public function anchor_finder($htmlDocument) {
    $DOM = new DOMDocument();
    @$DOM->loadHTML($htmlDocument);
    $xpath = new DOMXPath($DOM);
    $html_links = $xpath->evaluate("/html/body//a");
    if(count($html_links) != 0) {
        $array_of_links = array();
        foreach ($html_links as $link) {
            $href = $link->getAttribute('href');
            array_push($array_of_links, $href);
        }
        $array_of_links_no_duplicates = array_unique($array_of_links);
        return $array_of_links_no_duplicates;
    }
}

/* Parameter: The HTML document from which we want the metadata.
 * Return: null [inserts into objects variables directly]
 * Uses PHP's DOM functions to parse the HTML document:
 * - Uses XPath to gather HTML tags from the document. */
public function metadata_gatherer($htmlDocument) { //
$DOM = new DOMDocument();
@$DOM->loadHTML($htmlDocument);
$xpath = new DOMXPath($DOM);

$html_title = $xpath->query("/\text\title");
foreach ($html_title as $title) {
    $this->htmlTitle = $title->nodeValue;
}

$html_keywords = $xpath->query("/\meta[@name='keywords']");
foreach ($html_keywords as $key) {
    $this->htmlKeywords = $key->getAttribute('content');
}

$html_description = $xpath->query("/\meta[@name='description']");
foreach ($html_description as $desc) {
    $this->htmlDescription = $desc->getAttribute('content');
}
APPENDIX 5: CRAWLING APPLICATION

```php
<?php

/**
 ** Crawling Application
 ** Component of a Prototype Web Crawler
 ** 21.6.2011 Jyri Lehtonen
 **
 **
 */

class CrawlingApplication {

    public function __construct() {

    }

    /*
     ***
     ** Observer all form-fields in HTML source code and index them
     **  TODO: But do it so, that there is no need to index those already indexed!
     */
    public function find_form_fields() {

        $sql="SELECT source, title, id, url FROM pages";
        $result=mysql_query($sql) or die(mysql_error());
        while($row=mysql_fetch_array($result)) {
            $html_document=$row[0];
            $html_title=$row[1];
            $html_id=$row[2];
            $html_url=$row[3];

            if ($html_document) {
                $DOM = new DOMDocument();

```
@$DOM->loadHTML($html_document);  

 xpath = new DOMXPath($DOM);  
 $html_forms = $xpath->evaluate("/html/body//form");  

 if(count($html_forms) != 0) {  
 $html_document_array = explode("\n", $html_document);  
 foreach($html_document_array as $line){  
  if(stristr($line, '<form') == true) {  
   $form_field_total = stristr($html_document, $line);  
   $form_field_total = stristr($form_field_total, '</form>', TRUE);  
   $sql="INSERT IGNORE INTO forms (page, url, content) VALUES("$html_id",
 "$html_url","$form_field_total");  
   mysql_query($sql) or die(mysql_error());  
  }  
  }  
 }  
 else { //There is no form field  
 }  
 else {  
 //html document is null, but this should not get this far anyway in this case  
 }  

}  

}  

/*  
*** Observe the indexed form-fields and categorize them into types  
** but do not categorize a form-field that has been already categorized.  
*/  
public function form_field_categorization() {  

```php
$sql="SELECT id, content FROM forms ORDER BY id AND type=null"
$result=mysql_query($sql) or die(mysql_error());

$html_document_array = array();
while($row=mysql_fetch_array($result)) {
    $form_id=$row[0];
    $html_document_array = explode("\n", $row[1]);
    foreach($html_document_array as $line) {
        //print "<br>searching for search strings... $form_id<br>
        if(stristr($line, "login") == true || stristr($line, "register") == true) {
            $form_type="Login";
            $sql_query="UPDATE forms SET type="$form_type" WHERE id=$form_id";
            $result_query=mysql_query($sql_query) or die(mysql_error());
        }
        if(stristr($line, "search") == true) {
            $form_type="Search";
            $sql_query="UPDATE forms SET type="$form_type" WHERE id=$form_id";
            $result_query=mysql_query($sql_query) or die(mysql_error());
        }
    }
}
}

public function form_field_method() {
    (int)$post=0;
    (int)$get=0;

    //get the search form fields
    $sql='SELECT content FROM forms';
    $result=mysql_query($sql) or die(mysql_error());
    while($row=mysql_fetch_array($result)) {
        $form_field=$row[0];
    }
```

//get the action attribute from the form field
$DOM = new DOMDocument();
@$DOM -> loadHTML($form_field);
$xpath = new DOMXPath($DOM);
$form_field = $xpath->query("//form");

foreach ($form_field as $field) {
  $action=$field->getAttribute("method");
  $action=strtolower($action);
  if($action == "post") { $post=$post + 1; }
  if($action == "get")  { $get=$get +1; }
}

print "Forms had = POST ($post) and GET ($get) <br>";

}

public function query_form_field(){
  //get the search form fields
  $sql_query='SELECT content, url FROM forms WHERE type="Search" LIMIT 1';
  $result_form=mysql_query($sql_query) or die(mysql_error());
  print mysql_num_rows($result_form);
  while($formrow=mysql_fetch_array($result_form)) {
    $form_field=$formrow[0];
    $url=$formrow[1];
    print $url . "<br>";
    //get the action attribute from the form field
    $DOM = new DOMDocument();
@$DOM -> loadHTML($form_field);
$xpath = new DOMXPath($DOM);
$form_field = $xpath->query("//form");

foreach ($form_field as $field) {
    $action=$field->getAttribute("action");
    $post_data=Array();
    $inputs = $xpath->query("//input");
    foreach ($inputs as $input) {
        $input_name=$input->getAttribute("name");
        $input_value=$input->getAttribute("value");
        // $post_data["$input_name"] = "$input_value";
        $post_data["$input_name"] = "TEST";
        print("POST DATA:");
        print_r($post_data);
        print("<br>");
    }
    foreach ($post_data as $key => $value) {
        $post_items[] = $key . "=" . $value;
        $post_string = implode("&", $post_items);
    }
    $post_string=$post_string;
    $action=$url . "/".$action;
    $curlHandle=curl_init($action);
    curl_setopt($curlHandle, CURLOPT_TIMEOUT, 5);
    curl_setopt($curlHandle, CURLOPT_USERAGENT, "Mozilla/4.0 (compatible;)");
    curl_setopt($curlHandle, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($curlHandle, CURLOPT_POSTFIELDS, $post_string);
    $result = curl_exec($curlHandle);
    print_r(curl_getinfo($curlHandle));
    curl_close($curlHandle);
    print $result;