Toward Learning Based Web Query Processing

Yanlei Diao Hongjun Lu Songting Chen Zengping Tian

Department of Computer Science
Hong Kong University of Science & Technology
Hong Kong, China
diaoyl@cs.ust.hk luhj@cs.ust.hk

Department of Computer Science
Fudan University
Shanghai, China
stchen@fudan.edu.cn zptain@fudan.edu.cn

Abstract

In this paper, we describe a novel Web query processing approach with learning capabilities. Under this approach, user queries are in the form of keywords and search engines are employed to find URLs of Web sites that might contain the required information. The first few URLs are presented to the user for browsing. Meanwhile, the query processor learns both the information required by the user and the way that the user navigates through hyperlinks to locate such information. With the learned knowledge, it processes the rest URLs and produces precise query results in the form of segments of Web pages without user involvement. The preliminary experimental results indicate that the approach can process a range of Web queries with satisfactory performance. The architecture of such a query processor, techniques of modeling HTML pages, and knowledge for query processing are discussed. Experiments on the effectiveness of the approach, the required knowledge, and the training strategies are presented.

1. Introduction

The Internet and the Web have changed everything. It is estimated that the publicly indexable Web now contains about 600 million pages, encompassing approximately 6 terabytes of text data [15]. The Web has become everyone's information source. Each day, a huge number of people search the Web for information of interest, such as news, prices of goods, research papers, etc. With the excitement on electronic commerce growing, the Internet will also become a common platform for conducting business. The usage of the Web therefore will increase more dramatically.

Search engines [4] are widely used to locate information across the Web. Unfortunately for users who are used to retrieving information from database systems, searching from the Web is sometimes frustrating. For example, if they would like to find the lowest price for a certain part in a database, a simple SQL statement does the job. However, it may cost hours to search for the lowest price from the Web, if they have the stamina to find it. One problem of search on the Web is that search engines return very large hit lists with low precision. Users have to sift relevant documents from irrelevant ones by manually fetching and browsing pages. Another discouraging aspect is that URLs or whole pages are returned as search results. It is very likely that the answer to a user query is only part of the page (like one field in a relation). Retrieving the whole page actually leaves the task of search inside a page to Web users. With these two aspects remaining unchanged, Web users will not be freed from the heavy burden of browsing pages and locating required information, and information obtained from one search will be inherently limited.

While the dissimilarity between querying the Web and querying a database is caused by the fundamental differences between the Web and a database system, which will most likely remain, researchers from different disciplines have been trying to improve the situation.

A wide range of research work has been reported in IR to improve the easiness and effectiveness of querying the Web, including developing better classification mechanisms, building more effective indices, using better searching strategies and ranking functions, etc.

Using intelligent agent to help users is originated from the AI community. In the context of querying the Web,

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

such agents can learn user profiles or models from user search behaviors, and then employ the learned knowledge to predicate URLs that may have interesting information, thus providing suggestions to users. Some assistant agents, such as *Syskill & Webert* [20] and *WebWatcher* [13], help users in an interactive mode. Some other assistant agents, *Fab* [3] and *InfoSpider* [17], use heuristics and work autonomously to find interesting pages.

Researchers from the database community take another approach. They view the Web as a large distributed database system and apply database technologies to Web queries. The related efforts include Web query language design and wrapper generation. The Web query languages are classified into two generations [10]. The first generation, including *W3QL* [14] and *WebSQL* [18], aimed to unify content based queries and link structure based queries. The second generation of Web query language, such as *WebOQL* [2] and *StruQL* [9], has the ability to access the structure of the Web objects and to create new complex structures from the query results. Research work on wrapper generation tackles the fundamental difficulty in querying the Web, i.e., Web pages are not well structured and there is no schema that describes the contents of Web pages. It exploits the formatting information on Web pages to hypothesis the underlying structure of a page. With this structure a wrapper that facilitates queries on the page is generated [1, 5, 8, 12, 16, 21].

While there are various issues in Web query processing and different approaches to tackling these issues, we describe in this paper our efforts to build an on-line query processing system that enables users to query the Web with ease and obtain the results in a database-like fashion. By our proposed approach, a user first issues a key word query (probably not precise). It is passed to a general search engine such as *Yahoo!*. The search engine returns URLs of Web pages that might contain requested information. At the beginning, these Web pages are retrieved and presented to the user for browsing. During the browsing, the system records down the segment of a Web page that contains the query result and the sequence of hyperlinks through which the user navigates to find it. Query results and user actions are analyzed. After the user browses a few pages, the system knows what the user exactly wants and becomes capable of scanning Web pages and following links, if necessary, to locate the query results. Finally the system presents to the user the *segments* of Web pages instead of the original Web pages. A segment can be a paragraph in text, a table or a list.

To our knowledge, it is the first query processing system that processes ad hoc queries on HTML pages and automatically extracts segments of pages as query results. Despite the superficial similarity with a large body of related work, this system is unique at the following aspects:

- Unlike information retrieval systems or intelligent agents that return URLs or Web pages as query results, our system returns segments of documents as the query answer (correspondingly in relational databases if a field is the answer, the field but not the whole table is returned).
- The system does not require a prior knowledge about users such as user profiles. Moreover, it does not require preprocessing of Web pages such as generating wrappers either. As a consequence, the system is well suited to processing ad hoc queries that can hardly be handled by static hypertext analysis [5, 6], agents or system using wrappers.
- The system exploits the page formatting and the linkage information simultaneously to automate query processing. Recently there has been a surge of research work either on hyperlinks to help Web search [4, 11, 17] or on internal structure of HTML pages for wrapper generation and *Information Extraction* [7, 8]. However, combining these two in the context of a query processing system is new and poses great challenges.

We call our approach a *learning-based* approach because the system learns about the exact query requirements and the efficient way to locate the information during a query process. As a result of the learning, the system is able to deliver to users the results that better match users' needs in a more concise form. The learning approach brings the following advantages to our query processing system.

- Users can still express their queries in keywords, which is the easiest way for casual Web users. If a user is not familiar with the vocabulary of information suppliers, the query specification may even be vague. To bridge the gap between the issued keywords and the real user requirements, the system learns the precise requirements from users.
- The system has the capability of navigating in the neighborhood of the page where the specified keywords occur. Often the result is not in the page that contains the keywords but is one link or two away. This is especially true when the specified keywords are not very precise. Learning to navigate enables the system to find results that keyword search fails to find.
- Although the learned knowledge is useful to one query, it helps to make 100% use of the hit list returned by a search engine. Users are relieved from browsing dozens or hundreds of Web pages in order to obtain all the required information.
- As a background process of a browser, learning is nearly imperceptible to users and only minimal effort such as clicking and marking in the first few sites is required from them.

A prototype system has been implemented using the approach. The preliminary results are encouraging. User query requirements and navigation heuristics can be

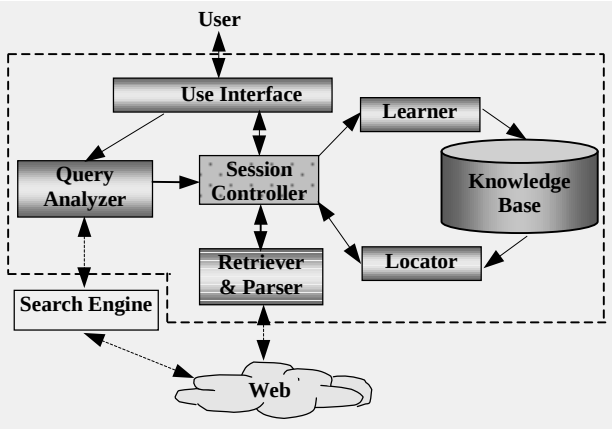

Figure 1: System Architecture

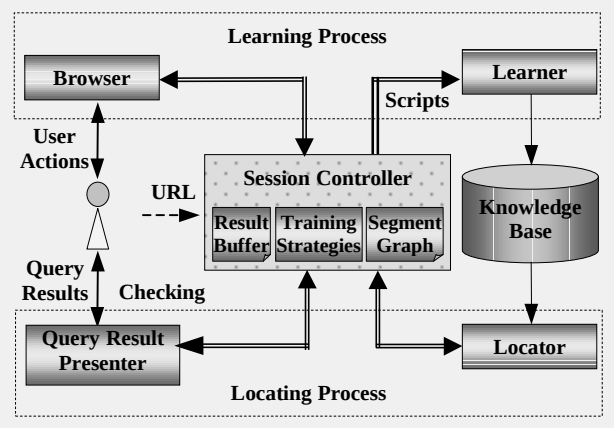

Figure 2: A Query Session

reasonably well captured and stored in a rather simple form. Given a set of about 100 URLs, users need to browse no more than 10 of them to make the system capable of locating the queried segments or denying the Web sites with the correctness rate higher than 80%.

The remainder of the paper is organized as follows. Section 2 describes the learning based Web query processing approach in detail. The knowledge to be learned, its representation and the acquisition process are described in Section 3. Section 4 describes how a user query is processed using the learned knowledge. Experiments conducted to evaluate the approach are presented in Section 5. Section 6 concludes the paper with some discussions on future work.

2. Learning-Based Web Query Processing

In this section, we describe the architecture of a learning-based Web query processing system and explain how a user query is processed.

2.1 A Learning-Based Query Processing System

Figure 1 depicts the reference architecture of a learning-based Web query processing system. It consists of seven major components: *User Interface*, *Session Controller*, *Query Analyzer*, *Learner*, *Locator*, *Retriever & Parser*, and *Knowledge Base*. The User Interface provides users with a friendly environment to work with the system. It accepts user queries and presents results to them. A browser with extended capability to capture user actions is also an important component of it. When a user is browsing Web pages, it records three types of user actions:

- following a hyperlink to browse another page;
- marking a segment that contains the required information; or
- rejecting a site that does not contain the required information.

Query Analyzer analyzes a user query and converts it into a search condition according to the requirement of the search engine that is employed to return URLs from the Web. The set of URLs is passed to Session Controller as the input of the other components in the system.

As a learning-based system, the system can work in two different modes, the *learning mode* and the *processing mode*. When the system works in the learning mode, Learner is activated by Session Controller to generate knowledge from those captured actions and located query results. The learned knowledge is stored in Knowledge Base. When the system works in the processing mode, Locator is activated to apply the knowledge and locate the segment that contains the required information. The two working modes are switched back and forth based on training strategies. The main task of Session Controller is to coordinate the interaction among various components of the system.

Since the result from the search engine is a set of URLs, a Retriever is integrated into the system to retrieve Web pages. The Parser parses each retrieved page and generates an internal data structure that is used later for presentation, learning and query processing.

2.2 A Query Session

To have better understanding of how a query is processed, we describe a query session in detail. As described above, after Session Controller receives the URLs, the system works in either the *learning mode* or the *processing mode*. Corresponding to these two modes, there are two types of processes: the *browsing process* and the *locating process*. Figure 2 depicts the details of the processes and associated data flow (Query Analyzer, and Retriever & Parser are omitted in this figure).

Session Controller activates the processes according to a training strategy stored in its *Training Strategies Module*. A training strategy defines when the learning process should be invoked and how the learning mode and processing mode are interleaved. Three strategies supported by the system are:

Sequential training. It partitions the URLs returned from the search engine into two sets in the original order.

For the first set, the system works in the learning mode. After training, the system processes the second set in the processing mode until the query session is completed.

Random *training.* Similar to sequential training, the system first works in the learning mode and then turns to the processing mode. But it randomly picks a number of sites from the returned URLs for training. The rationale is that the randomly picked sites may be more representative than those on the top of the returned list.

Interleaved training. When interleaved training is used, the system switches back and forth between the learning mode and the processing mode before a stopping criterion is met. It works as follows. At the beginning of a query session, the system is in the learning mode. After a few sites are browsed, the system tries to locate results in the processing mode. As long as the user confirms the results are correct, it remains in the processing mode. When the user finds an incorrect result, the system switches to the learning mode and learns from the incorrectly processed site. This process goes until a stopping criterion for interleaved training (a certain number of browsing processes or an accuracy threshold) is met. After that, the system remains in the processing mode until all sites are processed.

During a browsing process, given a URL, Session Controller first asks Retriever & Parser to retrieve the page and transform it to a segment tree. It adds the tree to the segment graph, an internal data structure maintained by the *Segment Graph Module* (segment tree and segment graph are defined in the next section). Then the controller sends the tree to Browser where the tree is presented for browsing. If the user chooses a link, the system goes to process a new page. The process is repeated until the user marks a query segment or rejects the site. User behaviors, either choosing a link or marking a segment, are recorded on the segment graph. For a successfully located site, the controller generates intermediate files, *knowledge scripts* from the segment graph. The scripts are finally sent to Learner for knowledge generation.

In a locating process, given a URL, Session Controller receives a segment tree from Retriever & Parser and adds the tree to the segment graph. Then it sends the tree to Locator. Locator returns a decision of choosing a link, finding a segment or rejecting the site. If a link is chosen, the system goes to process the new page and asks Locator to make another decision. The process ends when Locator finds a query segment or rejects the site. The located segment is sent to the *Result Buffer Module* that communicates with *Query Result Presenter* in the interface to present the result. When Interleaved training is used and the stopping criteria for training is not met, Query Result Presenter asks the user to check results. If the system returned a wrong result, a browsing process is activated for the current site. The only difference from a normal browsing process is that some pages can be fetched directly from the segment graph.

A query session terminates when all the URLs returned from the search engine are either browsed or processed. If there are too many URLs returned, heuristics can be used to terminate a session.

3. Learning from Users for Query Processing

To facilitate Web query processing, the data on the Web should be properly modeled. Moreover, the system must have the knowledge about how to navigate through the Web to locate information in a page. The most efficient way to obtain such knowledge is to learn from users. This section presents our approaches to these two major issues.

3.1 Modeling A Web Site

The Web consists of a number of Web pages connected by hyperlinks. In order to obtain queried information from a large number of Web pages, both the internal structure of a Web page and the linkage between Web pages need to be captured. We begin with the modeling of a single Web page.

Usually a Web page is an HTML document that contains a sequence of tag delimited data elements. As an atomic element, one that does not contain other elements in it, may not contain enough information to meet the query requirements, *segment* that is a group of elements is used as the unit in our model. In other words, we partition documents into segments each of which serves as a candidate of the answer to a query. Four major segment types are *paragraph*, *table*, *list* and *heading*. Segments can be nested, that is, a segment can include a number of sub-segments. An HTML document is the largest segment. Each segment has one attribute, *content*, which consists of all textual data in the scope delimited by the start tag and the end tag of the segment, thus including the content of its sub-segments. Content is used to check if a segment meets the query requirements. To facilitate navigation, another attribute, *description*, is designed for each segment (more discussion in § 3.2). It is summarized from the content using certain heuristics. For example, the description of a table segment can be the table caption or the title row. *Hyperlinks*, a special type of elements in HTML pages, can be represented in the same way as segments. The anchor is its description and the URL is its content. To take advantage of the hierarchical structure, each hyperlink is associated with its parent segment, i.e. the smallest segment that contains it[1].

With the notation of segments, a Web page can be modeled as a *segment tree*: The root is the Web page itself; the internal nodes are segments that contain sub-segments; the leaves are atomic segments, the minimal units in this model. Each node has two attributes and is

[1] Whether anchor should be included in the parent segment is a technical issue. Currently we include it in if the parent segment contains other text.

```
<html><head><title> … Hotel </title></head>
<body><p>1999 Room Rates</p>
<table><tr><td><ul>
<li><a href="ac01a.html">Guest Room</a></li>
<li><a href="ac02a.html">Executive Suite</a></li>
</ul></td>
<td> Special Promotion <br><table>
<tr><td>Room Type</td>
<td>Single/Double (HK$)</td></tr>
<tr><td>Standard</td><td>1000</td></tr>
<tr><td>Excutive Suite</td><td>2750</td></tr>
</table></td></tr></table>
</body></html>
```

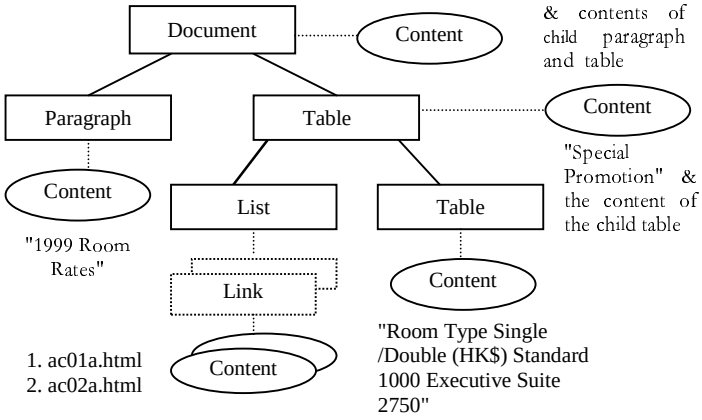

Figure 3: An HTML Document and the Corresponding Segment Tree

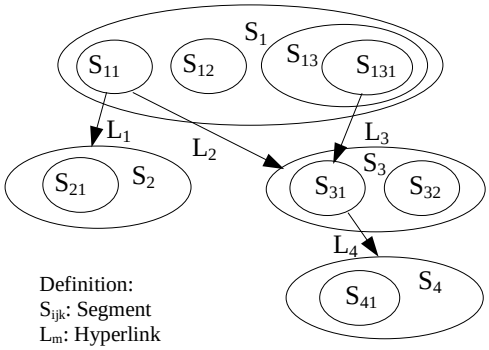

Figure 4: A Segment Graph

associated with hyperlinks it contains. An example of HTML page and the corresponding segment tree is shown in Figure 3 (the attribute, description, is omitted).

Externally, Web pages are connected through hyperlinks. If we view a Web page as a node in a graph and a hyperlink as a directed edge from the page containing it to the pointed page, then Web pages in a site can be represented by a directed graph. If we further ignore backward links, links pointing to one part of the same page, and links pointing to pages outside the current site, a Web site can be modeled as hyperlink-connected segment trees, called *Segment Graph*. The entering page, one pointed by a URL returned form the search engine, is the *root* of the graph. With such a model, *site*, which will be used very often later in this paper, refers to the collection of Web pages that are reachable from the root and of the same base URL as the root. We define the *depth* of a segment is the number of hyperlinks followed to reach the Web page that contains the segment. Note that segments on the same Web page may have different depths along different paths. Segments on the root page have depth 1. The *Level* of a segment is the minimal depth among all hyperlink paths in the segment graph.

We would like to emphasize that it is not our intention to provide a complete and sound model for Web pages and the Web. The sole objective of the above model is to facilitate the retrieval of meaningful query results in the form of segment that is small in size but carries sufficient semantics. The segment graph that combines the intra-document structure with the inter-document linkage can well serve the purpose. An example of such a graph is presented in Figure 4.

3.2 Knowledge for Locating Queried Segments

Let us consider the task of Web query processing. If our system could exhaustively search the segment graph and choose the most relevant segment from all in the graph, the problem would be simplified as hypertext classification. Unfortunately it is not feasible for an on line query processor because a segment graph can be very large. To restrict the search scope, navigation from the root should be terminated if the system finds a segment on a page that meets query requirements well enough or concludes the page is not relevant and will not lead to a relevant document. In other words, on each page, a decision of choosing a link, finding a segment, or giving up this page should be made. Though hyperlinks and internal page structure have been extensively studied by others, they made a decision either among all links or among all segments. A decision made between links and segments is something new and requires these two types of data structures are comparable.

One observation concerns the conventions of composing hypertext documents. Hyperlinks usually convey descriptive information of the pointed documents while segments that meet the query requirements contain both the descriptive information and the query result. For example, we would like to retrieve admission requirements of graduate applicants. The anchor "Admissions" only tells the link points to a page related to admission requirements. The queried segment contains both the descriptive information of admissions and the concrete requirements such as GPA or test scores. Links and segments, two structures presenting different information are hardly comparable by one mechanism.

To make the query system workable, two types of knowledge are designed. One is *Navigation Knowledge* that only concerns descriptive information and helps find a path from a given URL to the queried segment on a Web page. The other type of knowledge examines whether a segment meets the query requirements on both the descriptive information and the result. It is referred to as *Classification Knowledge* because it is in fact used to classify a segment into one of the two classes, containing or not containing the query result.

Attribute content and attribute description are designed for navigation knowledge and classification knowledge, respectively. Note that lengths of links and segments may differ remarkably. We assume the description of a segment can summarize the semantics of the segment and use it in navigation to avoid bias that element lengths bring. Another rationale of using description is that in self-describing languages like *Extensible Markup Language*, element names serve naturally as element descriptions so that our model carries over directly to them.

3.2.1 Navigation Knowledge

Navigation knowledge is generated from user actions of following hyperlinks to locate query results. A path that starts from the entering page of a site and ends at the queried segment is called a navigational path, represented as (*link*→)* *segment*, where * means any number of occurrences. For example, if segment S_{41} in Figure 4 is a queried segment, one possible navigational path is $L_2 \rightarrow L_4 \rightarrow S_{41}$. A hyperlink usually occurs in some segments in a document. Information of those segments also helps determine whether a link should be followed. To capture such information, we extend the navigational path with all segments that contain the links on the path, which is called *extended navigational path*. In our example, the extended navigational path to locate S_{41} is $(S_1 \rightarrow S_{11} \rightarrow L_2) \rightarrow (S_3 \rightarrow S_{31} \rightarrow L_4) \rightarrow (S_4 \rightarrow S_{41})$. A segment or a link appearing on the extended navigational path is called a *component* of it, e.g. S_{11}, L_4, S_{41}, etc. Extended navigational paths can be easily obtained from segment graphs in browsing processes.

To generate navigation knowledge from an extended navigational path, the first step is to assign a weight, denoted as *W*(*component*), to each component on the path. This weight tells how closely a component is related to the query result. One intuition is that the closer to the queried segment, the higher weight the component gets. Then the issue is at what rate the weight of a component decays along the extended navigational path. Instead of hypothesizing the rate, we assume the queried segment is the most closely related to itself (its weight is 1) and let the path length determine the rate. Suppose *D* to be the depth of a queried segment. On the i^{th} page along the path, N_i is the number of components appearing on the path. The weight of j^{th} component ($j <= N_i$) on the i^{th} page is given by:

$$W(component_{ij}) = (i-1)/D + 1/D * j/N_i . \qquad (1)$$

The weighting scheme guarantees $(i\text{-}1)/D < i/D$, i.e. the weight of a component on the $i\text{-}1^{th}$ page is less than that on the i^{th} page. The second term of the formula, $1/D * j/N_i$, ensures with the same depth the more specific information a segment conveys, the more weight it gets. In other words, a child segment gets more weight than its parent. A link gets more weight than all segments containing it. Continue with the example in Figure 4. Some components on the path are assigned weights as follows:

Depth:	Depth 1	Depth 2	Depth 3
Path:	$S_1 \rightarrow S_{11} \rightarrow L_2$	$S_3 \rightarrow S_{31} \rightarrow L_4$	$S_4 \rightarrow S_{41}$

$W(S_{11}) = 0/3 + 1/3 * 2/3 = 2/9$ the 2nd component at depth 1
$W(L_4) = 1/3 + 1/3 * 3/3 = 2/3$ the 3rd component at depth 2
$W(S_{41}) = 2/3 + 1/3 * 2/2 = 1$ the 2nd component at depth 3

The next step is to assign weights to terms that describe a component on the path. Since the attribute, *description*, provides such descriptive information, we choose it to represent a component. Then for each component, only words that are in the description and consist of alphabetic letters are selected. A stop list and stemming are further applied to them. The derived words are called terms. In our algorithm, each term in the description of a component is assigned a weight, *w*(*term*), which is equal to the weight of the component divided by the number of terms in the description.

The weight of a term tells the term's importance in leading to the queried segment. By our weighting scheme, it is determined by the position of the component that contains the term as well as the number of terms in the component's description. Term weight is accumulated through all browsing processes. The navigation knowledge, represented as a set of (*term*, *weight*) pairs, is stored in the navigation knowledge base.

3.2.2 Classification Knowledge

The task of examining whether a segment meets query requirements is cast in the Bayesian learning framework because it has provided good performance in text and hypertext applications [5, 6]. Two different models, the *multi-variate Bernoulli* model and the *multinomial* model in this framework are reported in [19]. By their report, the multinomial model usually outperforms the multi-variate Bernoulli model. Therefore we adopt the multinomial model. The *classification knowledge* is the knowledge that will be used by the Bayesian classifier.

Classification knowledge takes the form of a set of triplets, ($feature_i$, N_{i1}, N_{i2}), where N_{i1} is the number of occurrences of $feature_i$ in the content of queried segments, and N_{i2} is the number of occurrences of $feature_i$ in the content of segments that do not meet query requirements.

```
1   Algorithm LocatingProcess ( URL: the URL of a page, QueryResult:
    returned value of the query )
2   begin
3       Stack SegmentStack, LinkStack;
4       Float SegmentMax, LinkMax;
5       Tree SegmentTree;

6       QueryResult := NIL;
7       SegmentTree := Retriever&Parser(URL);
8       Separate(SegmentTree, SegmentStack, LinkStack);
9       if ( StopNavigation( ) == TRUE )
10          PopAll(LinkStack);
11      ApplyNavigation(SegmentStack, LinkStack);
12      while ( ( QueryResult == NIL ) AND (SegmentStack != NIL
                OR LinkStack != NIL) ) do
13          SegmentMax := GetMaxScore(segmentStack);
14          LinkMax := GetMaxScore(linkStack);
15          if ( SegmentMax >= LinkMax )
16              ApplyClassification(SegmentStack, QueryResult);
17              PopAll(SegmentStack);
18          else
19              URL := Pop(LinkStack);
20              if ( Unvisited(URL) == TRUE AND
                    StopNavigatoin( )==FALSE )
21                  LocatingProcess(URL, QueryResult);
22      end while
23  end.
```

Figure 5: The Locating Algorithm

Knowledge generation involves two issues, feature generation and selection of training samples.

Features are extracted from the content of a segment. Unlike most IR systems that only consider English words as features, we also consider values and complex data types. We define five basic feature types, *float*, *integer*, *English word* (consisting of alphabetic letters), *special word* (consisting of alphanumeric letters) and *special character*, and four complex feature types, *date*, *time*, *email address*, and *telephone number*. A lexical program using regular expressions extracts all these features.

To train the classifier, the user-marked segments are treated as positive samples. As for negative samples, only those segments on the same page as the marked segments are selected. Those pages without marked segments are discarded to avoid excessive negative samples. Then the generation of classification knowledge is straightforward. During the browsing process, when the user marks a queried segment, the system collects $feature_i$ in its content together with the number of occurrences N_{i1}, and $feature_j$ in other segments on the same page together with the number of occurrences N_{j2}. For each feature, numbers of occurrences in both classes are accumulative through all browsing processes.

Since both types of knowledge involve terms, they are organized by *Tries* for efficient access.

4. Query Processing Using Learned Knowledge

In this section, we describe how the learned knowledge is used to locate queried segments in the locating processes.

4.1 Algorithm for Locating Queried Segments

A locating process takes one URL returned from the search engine as the entrance to a site. Then it traverses the segment graph built on the fly. As an online Web query processor, the system will perform badly if general graph searching approaches like breadth-first or depth-first search are used. Considering hyperlinks and segments on a page simultaneously further complicates the search process. The learned knowledge helps the system locate query results efficiently and effectively.

By our approach, the choice between hyperlinks and segments on each page determines the navigation in a site. If a hyperlink is chosen, the locating process goes to the pointed page. If it fails to find a queried segment by following the link, it makes another choice between unvisited hyperlinks and unprocessed segments. If a segment is chosen, classification knowledge is applied to check if it meets the query requirements. If it does, it is returned as the query result and navigation terminates. Otherwise another choice is made between unvisited hyperlinks and not processed segments. If no result is found after all links and segments are processed, the locating process backtracks. The process is running in a recursive fashion.

The key issue is how to make a choice between hyperlinks and segments on a Web page. Navigation knowledge is used. It analyzes the descriptive information of links and segments on the same Web page, and assigns a weight to each of them. This weight uniformly tells how closely one element, either a link or a segment, is related to the query result. Then links and segments, the two different types of data, are sorted by the assigned weights. The element with the highest weight will be chosen for further processing.

Figure 5 presents the locating algorithm. The URL is first passed to *Retriever&Parser* that retrieves the Web page and parses it into a segment tree (line 7). Function *Separate* stores links and segments in *LinkStack* and *SegmentStack*, respectively (line 8). It pushes both atomic segments and segments that contain other segments into the stack. The classifier decides which segment answers the query best. The *ApplyNavigation* function assigns weights to segments and links using the navigation knowledge and sorts them in a decreasing order of the weights in *SegmentStack* and *LinkStack*, respectively (line 11). Each pass inside the *while* loop makes a choice between the link with the highest weight and the segment with the highest weight (line 13-15) as described above. The only change here is that if the weight of the segment is higher, function *ApplyClassification* applies

classification knowledge to **all** segments on this page and determines if one of them meets the query requirements (line 16). It is because retrieving a new page on the Internet takes far more time than processing a number of segments on the local machine. Besides, it is reasonable to assume if one segment provides best information about the query result among all links and segments on a page, it is more likely to find the result on this page instead of following a hyperlink. Function *StopNavigation* terminates the navigation if the locating process has already visited a certain number of pages and is still trying to visit more.

4.2 Application of Navigation Knowledge

Function *ApplyNavigation* first assigns weights to segments and links using the navigation knowledge learned during training, and then sorts them by the weights in a descending order.

To assign a weight, *W*, to a segment or a link, terms are extracted from its description in the way as described in § 3.2.1. Let the terms be t_1, t_2 … t_n, and their weights be w_1, w_2 … w_n. *W* is computed as:

$$W = max\ (topJ\ (w_1, w_2, \ldots, w_n)) . \quad (2)$$

Function *topJ* in the equation returns a set of term weights that are top *J%* highest in the navigation knowledge base. By considering top *J*% of terms only, those segments and links that are remotely related to the query result will be filtered out. When *J* is reasonably low, *topJ* returns an empty set for segments and links consisting of only terms with low weights. With an empty weight set, *W* is equal to 0. Those segments will be removed from the stacks later in the sorting process.

Function *max*, is used to keep the most descriptive information in segments and links. Rather than using an average or other function of the weights returned by *topJ*, we choose the max function because the relevance of a segment or a link is often conveyed by very few words. The convention of HTML pages is using short informative sentence fragments [7], which is especially true in lists, headings, hyperlinks, etc. As a result of this function, each segment or link is assigned a weight that is expected to best represent its relevance.

4.3 Application of Classification Knowledge

Function *ApplyClassification* calls a naïve Bayesian classifier to apply classification knowledge to one or multiple segments. The classifier used is adopted from [19] with some modification.

We begin with classification of one segment. For each segment, features are extracted from its content as described in § 3.2.2. The class label *C* (queried segment, or not in this application) of the segment *D'*, is given by:

$$C = \arg\max_k P(C_k \mid D') = \arg\max_k P(D' \mid C_k)P(C_k)$$
$$= \arg\max_k P(F_1 \mid C_k)...P(F_n \mid C_k)P(C_k) , \quad (3)$$

where C_k is a class label (k = 2 in this application) and F_i is a feature in the segment.

The estimation of the probability of feature F_i on condition of class k and each class prior are computed using the classification knowledge as follows:

$$P(F_i \mid C_k) = \frac{1 + N_{ik}}{|V| + \sum_{t=1}^{|V|} N_{tk}} , \quad (4)$$

$$P(C_k) = \frac{\sum_{t=1}^{|V|} N_{tk}}{\sum_{k=1}^{K}\sum_{t=1}^{|V|} N_{tk}} , \quad (5)$$

where N_{ik} comes from the triple (F_i , N_{i1} , N_{i2}) with k=1,2 and $\sum_{t=1}^{|V|} P(F_t \mid C_k) = 1$. To handle the probability of features that do not occur in training samples, smoothing of add-by-one is used. |V| is the vocabulary size of the classification knowledge.

To find the queried segment from all segments on a page, the function estimates the confidence of a segment being classified to certain class k, denoted by α_k, where $\sum_k \alpha_k = 1$. For a given segment, α_k (k=1,2,…,K) is calculated from the following equations:

$$\frac{E_1}{\alpha_1} = \frac{E_2}{\alpha_2} = \cdots = \frac{E_K}{\alpha_K} \text{ and } \sum_{k=1}^{K} \alpha_k = 1, \quad (6)$$

where $E_k = P(F_1|C_k)P(F_2|C_k)\ldots P(F_n|C_k)\ P(C_k)$. Let C_1 denotes the class of queried segments. Given a set of segments D_1, D_2, …, D_m , function *ApplyClassification* filters segments whose classification confidence α_1 is lower than a threshold and chooses a segment *D* with the largest α_1 from all kept segments:

$$D = \arg\max_j \{\alpha_{j1} \mid \alpha_{j1} > Threshold, j = 1, \ldots m\}, \quad (7)$$

where α_{j1} is the classification confidence α_1 with which class label C_1 is assigned to segment D_j . If all segments have α_1 lower than the threshold, *ApplyClassification* returns no result and the locating process goes on.

5. Performance Evaluation

A prototype of the system has been implemented based on the proposed approach. The system is implemented in Visual C++. *Yahoo!* is used as the external engine. The URLs of Web page matches are used in processing. A series of experiments were conducted to evaluate the proposed approach and study the related issues. In this section, we describe these tests and discuss the results.

5.1 Evaluation Metrics

For a given query and a URL returned from the search engine, the system either returns a segment or a conclusion that no result is found from the related Web site. We label the first case as *Found* and the latter one *Not Found*. For both cases, we use *Right* or *Wrong* to indicate whether the system makes a correct decision. Using the four terms, a query result belongs to one of the

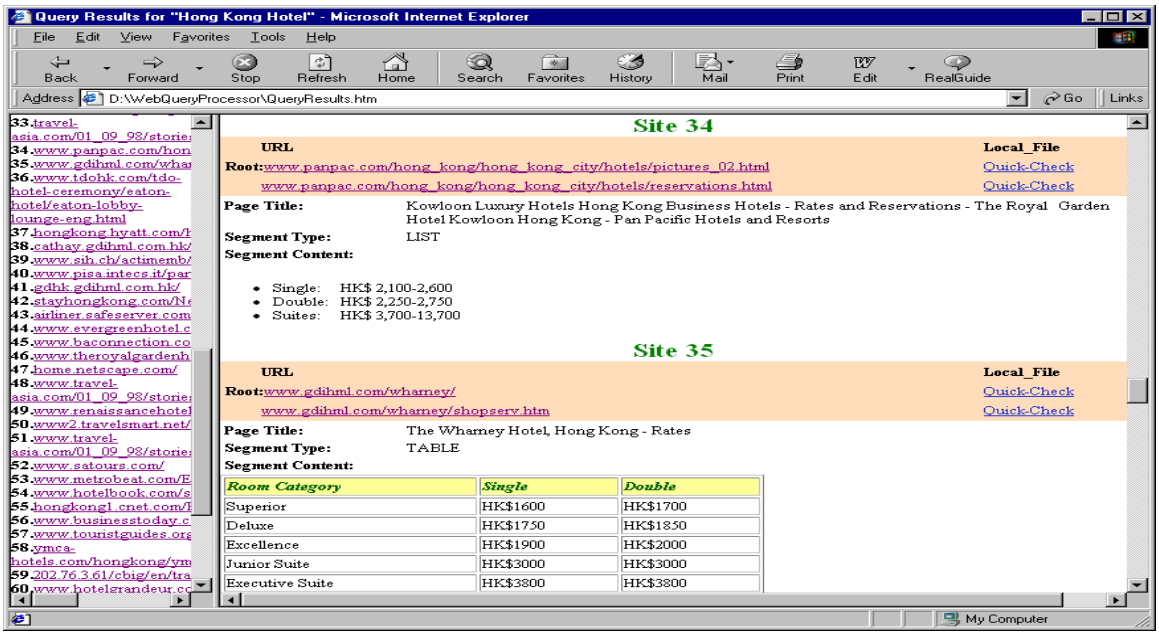

Figure 6: A Sample Output

four categories. A queried segment is a segment that satisfies user query requirements.

Right Found:	The queried segment is found.
Wrong Found:	A segment other than the queried segment or from an irrelevant site is returned.
Right Not Found:	No segment is returned from an irrelevant site.
Wrong Not Found:	The system fails to locate the queried segment that the site contains.

To evaluate *effectiveness*, the following metrics are defined:

Precision = *# Right Found / # Found* ,
Recall = *# Right Found / # Sites Containing Queried Segments* ,
Correctness = *# Right Sites / # Sites Processed* .

In the definition of *correctness*, *# right sites* is the number of the sites for which the system makes correct decisions. That is, it either locates a queried segment, or indicates correctly that a site does not contain a queried segment.

To evaluate efficiency, we take visited pages as the measure because the time of processing a page is insignificant compared to retrieving the page through the Internet. Two measurements are defined. The *absolute path length* is the number of visited pages to locate a queried segment or to conclude that no queried segment can be found for the site. The *relative path length* to locate a queried segment is the ratio between the absolute path length and the level of the queried segment (i.e. the length of the shortest path to locate this segment). The two metrics are presented as:

Absolute Path length = *# Visited pages* ,
Relative Path Length = *# Visited pages / Level of the Queried Segment*.

5.2 The System Capability

Before quantitative analysis of the system performance, we first present sample query results that indicate the capability of the system. A user posted a query consisting of 3 words, *Hong Kong hotel*, with the intention of finding hotel room rates in Hong Kong. The query was passed to *Yahoo!* and a set of URLs was returned, which is shown in the left frame of Figure 6, a snapshot of the system output. The right frame shows the query results after seven successful browsing processes during which the system learned the knowledge about the query. Currently the right frame presents the results located for Site 34 and 35. From the results we can see some of the novel features of the system.

- In addition to URLs and page titles that ordinary search engines can return, our system returns segments of the Web pages that contain queried information. The result of site 34 is a list and of site 35 is a table.
- The query results contain exactly the information that the user is looking for. Note that the segments from site 34 and 35 do not contain any input keyword but meet the requirement of room rates that are not specified in the keyword query. It indicates that the system learned the query requirement from the user.
- Both segments are from pages whose URLs are not directly returned by *Yahoo!*. It indicates that the system learned how to follow hyperlinks to the page that contains a queried segment.

Query	URL Selected	URL Used	Train-ing	Test-ing	Irrele-vant	Rele-vant
Q1	100	69	9	60	31	29
Q2	100	71	9	62	24	38

Table 1: URLs Used for Query Processing

Query	Found		Not Found		Correct-ness	Precision	Recall
	R	W	R	W			
Q1	23	7	26	4	81.7%	76.7%	79.3%
Q2	28	4	21	9	79.0%	87.5%	73.7%

Table 2: Basic Performance of the System

5.3 The Effectiveness of the System

In this subsection, we present the results of two queries to illustrate the effectiveness of the system. Two queries used are:

Q1: Hong Kong hotel *room rate,*

Q2: Hong Kong hotel.

The intention of the user was to locate the room rates of hotels in Hong Kong. Processing and usage of the URLs returned from *Yahoo!* is summarized in Table 1. Only 100 top-ranked URLs were chosen for processing. Among them, some URLs were removed. Examples of removed URLs include non-accessible ones, duplicates, URLs pointing to non-HTML documents[2], URLs pointing to non-English documents, etc. The sequential training strategy was used and the number of URLs used for training is shown in column *Training*. The relevancy of a site was determined by examining page contents manually.

The results of the experiments are summarized in Table 2. It can be observed that, with sequential training, the correctness for both queries reaches about 80%. Considering the big discrepancy between the keywords expressed in a user query and the exact requirement, the results are really encouraging. It justifies the basic approach described in the previous sections.

To understand why the system failed to locate some queried segments, we examined the cases where the system chose segments that did not meet query requirements. The major reason is that those returned segments contain much noise, namely the words that are very close to those used to locate queried segments. One returned segment is as follows:

"*QUEEN ELIZABETH 2. Standby fares for the six day transatlantic crossings on this famous ship are available on Nov. 21 (New York to* Southampton*) for $1,199 per person, double occupancy and on the Dec. 14 sailing (Southampton to New York) for $1,099. The single supplement is $350.*"

This segment contains words "*single*" and "*double*", symbol $ and numbers that are important clues to locate segments for room rates of hotels.

[2] Processing dynamic Web pages is still under development.

Accuracy	Query 1			Query 2		
	C	P	R	C	P	R
Both Types of Knowledge	81.7%	76.7%	79.3%	79.0%	87.5%	73.7%
Bayesian Only	58.3%	51.2%	69.0%	38.7%	34.0%	42.1%
Navigation Only	36.7%	28.8%	55.6%	29.0%	26.8%	39.5%

* C for correctness, P for precision and R for recall

Table 3: Effects of Using Different Types of Knowledge

Correctness	Query 1: Level=			Query 2: Level=		
	I	1	2	I	1	2
	31/60	27/60	2/60	24/62	12/62	26/62
Both Types of Knowledge	0.838	0.815	0.5	0.875	0.917	0.654
Bayesian Only	0.484	0.704	0.5	0.333	0.75	0.269
Navigation Only	0.212	0.6	0	0.125	0.583	0.308

* I means irrelevant

Table 4: Analysis of Different Types of Knowledge

There are a number of reasons that the system failed to locate a queried segment. Among four such cases of Q1, three of them were caused by the low confidence of being classified as queried segments, which indicates there is deficiency in the generated classification knowledge. Another such case resulted from deprecated use of HTML tags in the original document: A hyperlink uses an image element as the content but leaves the attribute ALT of tag IMG blank. As a result, the system could not find any descriptive information about the link and thus ignored the path that leads to the queried segment.

5.4 Effectiveness of the Knowledge

To verify the effectiveness of using the two types of knowledge, navigation and classification, we modified the system so that it can be configured to apply only one type of knowledge in the process of learning and locating.

Two queries and the testing environment are the same as Table 1. The results are summarized in Table 3. The results of using both types of knowledge are also included for easy comparison.

It can be clearly seen that, the system employing both types of knowledge performs much better than those that employ only one type of knowledge. To analyze the reason for the poor performance of using only one type of knowledge, we classified sites by the level of the queried segment. In Table 4, the title cells specifies the level distribution and the fraction stands for among all test sites, how many of them belong to the level. Usually the coarser the query, the more sites belong to the level above one. The correctness of each group is reported in table cells.

One finding is that the system with one type of knowledge works reasonably only when the queried segment occurs on the first page. Its ability to filter irrelevant sites and to navigate through hyperlinks is very limited. In contrast, employing two types of knowledge manifests good performance of irrelevance filtering and navigation. Even if the queried segment is on the first

page, using one type of knowledge is still less accurate then using two types of knowledge. The coordination of navigation knowledge and classification knowledge provides a good way to process Web queries.

5.5 Effects of Training Strategies

Our system supports three training strategies, *sequential training*, *random training* and *interleaved training*. This group of experiments was designed to find out which training strategy provides best performance in terms of both accuracy and efficiency. Besides the training strategies, the training size was another focus of this part. In an online query system, asking users to browse many sites is impractical. In our tests, we varied the training size from 3 to 10 for each training method. For interleaved training, the stopping criterion of training was a pre-defined number of browsed sites, i.e. the training size. The two queries of hotel room rates were used again. Observations are reported as follows. Figures are omitted in the interest of space.

- Random training performs badly in terms of effectiveness and efficiency. Assumption that randomly picked sites are more representative than those on the top of the returned list is not true.
- As the training size increases, the difference between sequential training and interleaved training is enlarged. With 10 training sites, interleaved training beats sequential training by 10% in all effectiveness metrics. The better performance of interleaved training comes from its way of updating knowledge -- updating it when the system makes a mistake.
- In terms of efficiency, interleaved training strategy is also the best. Relative path length for a right "Found" segment is in the range of 1.0 to 1.2. That means the system almost always navigates through the shorted path to locate a result. The absolute path length for a right "Not Found" is from 1 to 1.5 pages.

A final note is that, when interleaved training strategy was used with 10 training sites, the number of wrong decisions of Q1 was reduced from 11 to 4 (refer to Table 2). Obviously the knowledge learned by using interleaved training improved greatly.

5.6 Experiments on a Range of Queries

We made some observations of issues related to implementing a Web query processor from two queries concerning room rates of hotels. To better evaluate the effectiveness and efficiency of such a processor, we tested a range of queries on it. Three query requirements with distinct characteristics were selected.

QR1: *room rates of Hong Kong hotels* (included for comparison). It targets at prices, which is well defined and easy to identify by a user during the browsing process. The system is expected to extract the price information during the locating process.

Query Requirement (QR)	**Keyword Query (KQ)**
QR1: room rates of Hong Kong hotels	KQ11: "Hong Kong hotel room rate"
	KQ12: "Hong Kong hotel"
QR2: admission requirements on graduate applicants	KQ21: "requirements graduate applicant"
	KQ22: "graduate applicant"
QR3: data mining researcher	KQ3: "data mining researcher"

Table 5: Query Requirements and Keyword Queries

Performance	**QR1**		**QR2**		**QR3**
	KQ11	**KQ12**	**KQ21**	**KQ22**	**KQ3**
Correctness	0.93	0.9	0.84	0.91	0.83
Precision	0.92	0.92	0.85	0.88	0.64
Recall	0.88	0.94	0.94	0.91	0.67
Relative Path Length (Found)	1	1.21	1.08	1.1	1
Absolute Path Length (Not Found)	1.3	1.57	2.5	1.76	1.67

Table 6: Results of A Range of Queries

QR2: *admission requirements*. A user may not know the exact query requirements when she/he issues the keyword query. During the browsing process, she/he makes it clear that the requirements concern concrete items such as degree, GPA, GRE, TOEFL, etc. The system is expected to extract these items as query results. Compared to QR 1, the query results have larger variance because they may contain different sets of items as the need is.

QR3: *data mining researcher*. The query target is in fact a concept. The user wants to extract segments of pages as evidence that a person is a data mining researcher. It is even hard for human readers to tell what these segments should contain and such decision is very subjective. During the browsing process, the user gets to know a data mining researcher can be reflected by research interests, research projects, professional activity, etc. The system is expected to recognize the pieces of the evidence and return the correct segments.

For the first two query requirements, we issued two keyword queries with different precision. All five keyword queries are listed in Table 5. For each keyword query, from the URLs returned by *Yahoo!*, we collected the top 100 and cleaned them for later tests[3]. The interleaved training strategy was used with training size of 10.

The quantitative results are presented in Table 6. Our system works well for the first 4 queries. Accuracy is above 80% and in some queries it reaches 90%. For the last query, precision and recall are not as high as those are in other queries. Fortunately the system is still capable of filtering out irrelevant sites, thus to make the correctness reasonably good. The relative path length to locate a queried segment is close to 1. The absolute path length in an irrelevant site is no more than 2.5 pages.

[3] Manual check of each site in the test prevented us from enlarging the URL list. We assume the system will process other URLs with the same performance as it does with the first 100.

Another observation is that the performance of our system is not affected much by how precise the keyword query is. Thus users do not need to worry about the exact words when they issue the queries. In the browsing process they can tell the requirements by their actions. The system will learn them and use them in the locating process.

6. Conclusion

In this paper, we proposed a novel approach for processing queries on the Web. Taking such approach, a user can issue queries in free text sentences and get the results in the form of segments containing the required information. Minimum involvement is required from the user to train the system. To process a query, a general-purpose search engine is employed to get the initial relevant URLs, which are taken as the input of a series of browsing and locating processes. During the browsing processes, the system learns user requirements and the way he navigates through the hyperlinks to locate the segments that meet query requirements. During the locating processes, such learned knowledge is applied to locate the queried segments from a large number of Web pages without interaction of the user. Our preliminary experiments produced encouraging results, which shows that the proposed approach is able to tackle the difficult problem of queries on the Web.

A prototype system has been developed based on the proposed approach. More comprehensive experiments are being conducted. Our future work includes better knowledge representation and more sophisticated algorithms for learning and applying knowledge. To process a wide range of Web queries, HTML page segmentation is another issue that deserves further study.

Acknowledgement

This work is partially supported by a grant from the National 973 project of China (No. G1998030414) and a grant from the Research Grant Council of the Hong Kong Special Administrative Region, China (AOE98/99.EG01)

References

[1] N. Ashish, C. Knoblock. Wrapper Generation for Semi-structured Internet Sources. *SIGMOD Record*, 26(4), pp. 8-15, Dec. 1997.

[2] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Proc. of ICDE 98*, pp. 24-33, Feb. 1998.

[3] M. Balabanovic. An Adaptive Web Page Recommendation Service. In *Proc. of 1st International Conference on Autonomous Agents*, pp. 378-385, 1997.

[4] S. Brin, L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proc. of the 7th WWW Conference*, pp. 107-117, 1998.

[5] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigam and S. Slattery. Learning to Extract Symbolic Knowledge from the World Wide Web. In *Proc. of AAAI-98*, 1998.

[6] S. Chakrabarti, B. Dom and P. Indyk. Enhanced Hypertext Categorization using Hyperlinks. In *Proc. of ACM-SIGMOD 98*, pp. 307-318, Jun. 1998.

[7] D. DiPasquo. Using HTML Formatting to Aid in Natural Language Processing on the World Wide Web. Senior Honors Thesis, School of Computer Science, CMU, May 1998.

[8] D. Embley, Y. Jiang, Y. Ng. Record-Boundary Discovery in Web Documents. In *Proc. of ACM-SIGMOD* 99, pp. 467-478, May 1999.

[9] M. Fernandez, D. Florescu, Dan Suciu. A Query Language for a Web-Site Management System. *SIGMOD Record*, 26(3), pp. 4-11, 1997.

[10] D. Florescu, A. Levy and A. Mendelzon. Database Techniques for the World Wide Web: A Survey. *SIGMOD Record*, 27(3), pp. 59-74, Sep. 1998.

[11] D. Gibson, J. Kleinberg and P. Raghavan. Inferring Web Communities from Link Topologies. In *Proc. of ACM Hypertext 98*, pp. 225-234, Jun. 1998.

[12] J. Hammer, M. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, R. Yerneni. Template-Based Wrappers in the TSIMMIS System. In *Proc. of ACM-SIGMOD*, pp. 532-535, May 1997.

[13] T. Joachims, D. Freitag, T. Mitchell. WebWatcher: A Tour Guide for the World Wide Web. In *Proc. of the* 1997 *International Joint Conference on Artificial Intelligence*, pp. 770-775, Aug. 1997.

[14] D. Konopnicki, O. Shmueli. W3QS: A Query System for the World Wide Web. In *Proc. of VLDB 95*, pp. 54-65,1995.

[15] S. Lawrence and C.L. Giles. Accessibility of information on the Web. *Nature*, 400(8), pp. 107-109, Jul. 1999.

[16] L. Liu, W. Han, D. Buttler, C. Pu, W. Tang. An XML-Based Wrapper Generator for Web Information Extraction. In *Proc.of ACM-SIGMOD 99*, pp. 540-543, May 1999.

[17] F. Menczer, R. Belew. Adaptive Retrieval Agents: Internalizing Local Context and Scaling up to the Web. *Technical Report* CS98-579, University of California, San Diego, 1998. Available: http://www.cse.ucsd.edu/~rik/*papers/arachnid/arachnd-mlj.ps.*

[18] A. Mendelzon, G. Mihaila, T. Milo. Querying the World Wide Web. *International. Journal on Digital Libraries*, 1(1), pp. 54-67, 1997.

[19] A. McCallum, K. Nigam. A Comparison of Event Models for Naïve Bayes Text Classification. *Working Notes of AAAI/ICML-98 Workshop on Learning for Text Categorization*, pp. 41-48, 1998.

[20] M. Pazzani, J. Muramatsu, D. Billsus. Syskill & Webert: Identifying Interesting Web Sites. In *Proc. of AAAI-96*, pp. 54-61, 1996.

[21] A. Sahuguet, F. Azavant. Building light-weight wrappers for legacy Web data-sources using W4F. In *Proc. of VLDB 99*, pp. 738-741, Sep. 1999.