Applications of a Web Query Language

Gustavo O. Arocena
University of Toronto
gus@db.toronto.edu

Alberto O. Mendelzon
University of Toronto
mendel@db.toronto.edu

George A. Mihaila
University of Toronto
georgem@db.toronto.edu

August 12, 1997

Abstract

In this paper we report on our experience using WebSQL, a high level declarative query language for extracting information from the Web. WebSQL takes advantage of multiple index servers without requiring users to know about them, and integrates full-text with topology-based queries.

The WebSQL query engine is a library of Java classes, and WebSQL queries can be embedded into Java programs much in the same way as SQL queries are embedded in C programs. This allows us to access the Web from Java at a much higher level of abstraction than bare HTTP requests.

We illustrate the use of WebSQL for application development by describing two applications we are experimenting with: Web site maintenance and specialized index construction. We also sketch several other possible applications.

Using the library, we have also implemented a client-server architecture that allows us to perform interactive intelligent searches on the Web from an applet running on a browser.

1 Introduction

Developing database applications has been made dramatically easier, faster, and more reliable by the use of declarative data languages such as SQL. We conjecture a similar payoff will come from using declarative high-level languages to access information on the World Wide Web. We have designed and built a prototype of such a language, called WebSQL [Mih96, MMM96], and in this paper we illustrate the use of WebSQL for application development by describing in some detail two applications, Web site maintenance and specialized index construction, and sketching several others.

What sort of applications do we have in mind? Consider for example the *WebGlimpse* system developed at the University of Arizona [MSG97]. WebGlimpse combines searching

and browsing by allowing a user to request at any time a full-text search, not on the whole Web through some large index server, but on the *neighborhood* of a given page that has been reached by browsing. For example, if I am looking for Canadian price lists for IBM Thinkpads, I can navigate to the IBM Canada home page by going directly to `www.ibm.com` and following a couple of links, and then ask for all pages that contain keywords such as "price" and are reachable from this home page by a short path, say no more than three links in length. [1] Using WebSQL would facilitate building an application like WebGlimpse because a highly flexible specification of what is a "neighborhood" can be encapsulated in a declarative WebSQL query, and the details of index access for full-text search hidden under the covers.

To recapitulate, we see a Web query language not primarily as an end-user tool (although we have built interfaces that support interactive WebSQL queries), but as an aid in application development, to be used in conjunction with a programming language, just as SQL is used to build database applications by embedding it in a programming language.

The structure of the rest of the paper is as follows: in Section 2 we present an intuitive introduction to WebSQL and the data model behind it (for a complete description of the language, including a formal specification of its semantics, refer to [Mih96]), in Section 3 we give examples of applications using embedded WebSQL, in Section 4 we sketch the current implementation of the query engine and of the several available user interfaces, in Section 5 we explain how to use the WebSQL library for application development and in Section 6 we present our conclusions.

2 A Relational View of the WWW

WebSQL proposes to model the web as a relational database composed of two (virtual) relations: Document and Anchor (see Figure 1). The Document relation has one tuple for each document in the web and the Anchor relation has one tuple for each anchor in each document in the web. This relational abstraction of the web allows us to use a query language similar to SQL to pose the queries.

If Document and Anchor were actual relations, we could use SQL to write queries. For example, the next SQL query would give us all the pairs of URLs of documents with the same title:

Query 1:

```
SELECT d1.url, d2.url
FROM Document d1, Document d2
WHERE d1.title = d2.title
      AND NOT (d1.url = d2.url)
```

But we cannot effectively compute queries like the previous one. Since the Document and Anchor relations are completely virtual and there is no way to enumerate all the documents

[1] Unfortunately, there does not seem to be any such price list in the vicinity of the IBM Canada home page.

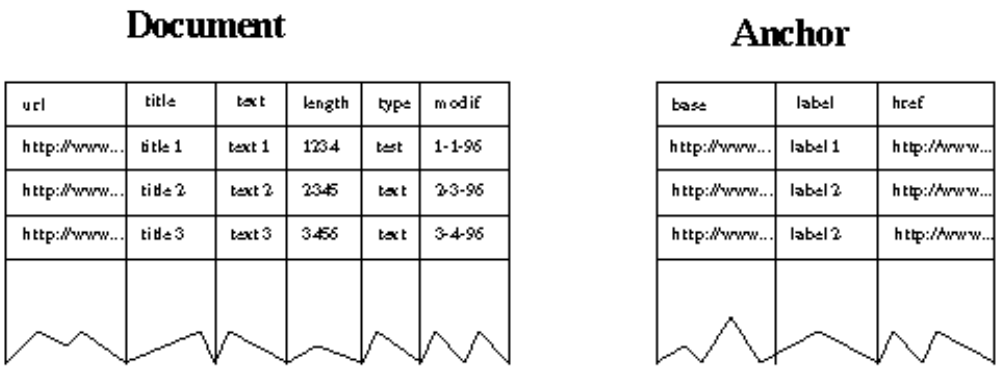

Figure 1: The Web According to WebSQL

in the web, we cannot operate on them as we would do with relations in a conventional relational database. For this reason, instead of operating directly on the virtual relations, WebSQL operates on materialized portions of them. A portion of the Document relation is materializable if we can somehow enumerate the URLs of all the documents it contains, and a portion of the Anchor relation is materializable if we can enumerate the URLs of the documents that contain the anchors in that portion. Once the materialized portions have been defined, we can manipulate them as traditional relations. For example, the next WebSQL query, which is a slight variant of the previous one, can be effectively computed:

Query 2:

```
SELECT d1.url, d2.url
FROM Document d1 SUCH THAT d1 MENTIONS "something interesting",
     Document d2 SUCH THAT d2 MENTIONS "something interesting"
WHERE d.title = d2.title AND NOT (d1.url = d2.url)
```

The only difference between the above two queries is the addition of the "SUCH THAT ... MENTIONS ..." construct to the definition of each variable. This construct allows us to define a materialized portion of the web that contains all the documents that mention some keyword in their text. The implementation of MENTIONS consists of a query to one of the global index servers (AltaVista, Excite, HotBot, etc.).

3 Intelligent search

As we explained in the introduction, queries to index servers are one of the two ways in which people search for information in the web. The other is manual navigation. WebSQL also helps automate the navigation process. For example, the query:

Regexp	Meaning
–>–>=>	Path of length three composed of two local followed by one global link
(–>|=>)	Path of length one, either local or global
–>*	Local paths of any length
=> –> *	One global link followed by any number of local links
= |#>|–>	Local path of length zero or one

Table 1: Examples of Path Regular Expression

Query 3:

```
SELECT d.url, d.title
FROM Document d SUCH THAT "http://www.somewhere.com" -> d
```

retrieves the title and the URL of all documents that are pointed to from the document whose URL is "http://www.somewhere.com" and that reside in the same server as this one. The arrow between the URL and the variable specifies the path that must be followed in order to find the desired documents (in this case, a path of length one within the same server). Had we used '=>' instead of '–>', we would have specified a path of length one but where the destination document is in a different server than the document of origin. We call '–>' and '=>' local and global link types, respectively.

Another example of path specification is '–>=>', which specifies paths of length two, composed of a local link followed by a global link. In general, link types can be combined in full regular expressions, yielding what we call Path Regular Expressions. The alphabet for these regular expressions is { –>, =>, #>, = }. We have not yet mentioned the symbols '#>' and '=', which represent an internal link within the same document and a path of length zero, respectively. Table 1 gives more examples.

Using path regular expressions we can specify interesting search patterns. For example, the next query searches for pages related to databases in the web site of the Department of Computer Science of the University of Toronto:

Query 4:

```
SELECT d.url
FROM Document d SUCH THAT "http://www.cs.toronto.edu" ->* d,
WHERE d.text CONTAINS "database" OR d.title CONTAINS "database"
```

We can also combine queries to index servers with navigational queries. In the following query, we search for job opportunities for software engineers.

Query 5:

```
SELECT d1.url, d1.title, d2.url. d2.title
FROM Document d1 SUCH THAT d1 MENTIONS "employment job opportunities",
```

```
     Document d2 SUCH THAT d1 =|->|->-> d2
WHERE d2.text CONTAINS "software engineer"
```

We first query an index server to find pages that mention the keywords "employment job opportunities" and then, from each of these pages, we follow the local paths of length zero, one or two to find pages that contain the keywords "software engineer." In the previous queries we only used the Document relation. The information in anchors can also be useful for specifying searches. For example, suppose we want to find the pages describing the publications of some research group. If we know the URL of the home page of this group and we assume that the anchors pointing to the pages describing the publications contain the word "papers" in their label, we could write the query:

Query 6:

```
SELECT a1.href, d2.title
FROM Document d1 SUCH THAT "http://www.university.edu/~group" ->* d1,
     Anchor a1 SUCH THAT base = d1,
     Document d2 SUCH THAT a1.href -> d2,
WHERE a1.label CONTAINS "papers"
```

With this query we explore the pages in the web site in search for pages containing anchors with the word "papers" in their label. Once we find them, we add a tuple to the answer containing the URL and the title of the document pointed to from this anchor. The next query illustrates a different search strategy for the same problem: instead of looking at the text in the anchors, we look at the URL they refer to and add to the answer the URL and the title of those documents that contain anchors pointing to compressed Postscript files.

Query 7:

```
SELECT d1.url, d1.title
FROM Document d1 SUCH THAT "http://www.university.edu/~group" ->* d1,
     Anchor a1 SUCH THAT base = d1
WHERE filename(a1.href) CONTAINS "ps.gz" OR filename(a1.href) CONTAINS "ps.Z"
```

Observe that the path regular expression '–> *' in Query 6 matches paths whose length is one unit shorter than those matched by the same expression in Query 7.

3.1 User-defined link types

The alphabet of path regular expressions can be extended in order to introduce predicates that test for properties of links other than locality to a server or to a document. This allows us to express richer conditions on paths. We refer to this facility as *user-defined link types*. Suppose we want to know which documents of a set of documents mention the word "Canada" in their titles. If these documents are arranged in a list such that the label of the link from a document to the next is the string "Next," we can define a link type as follows:

```
DEFINE LINK [next]
AS label CONTAINS "Next";
```

A link will match the link type [next] only if its label contains the word "Next." Once we have defined the link type, we can use it in a path regular expression as we did with the predefined link types. The query that looks for the information we want is:

Query 8:

```
SELECT d.url
FROM Document d SUCH THAT "http://x.y.z/TheStartingPage.html" [next]* d,
WHERE d.title CONTAINS "Canada"
```

3.2 Using the context of anchors

In many cases, the text surrounding an anchor can be a valuable source of information for deducing the meaning of the relation represented by the link. In order to be able to refer to this text, WebSQL provides the *context* attribute, whose meaning is specified by the user. For example, if the document pointed to from "http://x.y.z/songs.html" contains anchors pointing to audio files in *wav* format, and the text surrounding each of these anchors gives a description of the content of the corresponding audio file, then the following query searches for songs interpreted by The Beatles:

Query 9:

```
DEFINE CONTEXT BEGIN = <HR>, END = <P>;

SELECT a.href
FROM Anchor a SUCH THAT base = "http://x.y.z/songs.html"
WHERE file(a.href) CONTAINS ".wav" AND a.context CONTAINS "beatles"
```

The DEFINE CONTEXT declaration specifies that the context associated to each anchor is the piece of text between the first ⟨HR⟩ tag that precedes it and the first ⟨P⟩ tag that follows it. This piece of text can be referred to in the query using the *context* attribute for variables ranging over the *Anchor* relation.

The general rule defining the meaning of the *context* attribute is that, for each anchor, the document source is scanned backward (resp. forward) until the target specified in BEGIN (resp. END) is found. The value of *context* is the text (excluding markup) between the two points thus defined. In addition to HTML tags, relative offsets (in number of tags) can be used to define the endpoints.

4 Other applications

As explained in the Introduction, WebSQL was not only meant for being used interactively but also, and mainly, for developing applications and tools that could benefit from accessing

the Web at a high level of abstraction. Both the interactive and the embedded uses of the language involve searches, but of different kinds. In interactive searches, as we saw above, we are likely to exploit the language capabilities to query index servers, test conditions on different parts of the documents, and to use path regular expressions to explore neighborhoods of certain documents. On the other hand, the searches we are likely to do when using embedded queries exploit the three elementary tasks that can be automated by using WebSQL: a) given a URL, retrieve the corresponding document from the web; b) iterate over all the anchors in a document that has been retrieved and c) specify a collection of documents by means of a path regular expression.

Below we show examples of the kind of things that can be done by combining these three basic operations. All these examples have to do with tasks that are likely to be routinely done when maintaining a web site.

4.1 Finding Broken Links

The following query finds the "broken links" in a page:

Query 10:

```
SELECT a.href
FROM Anchor a SUCH THAT base = "http://x.y.z/ThePageToTest.html"
WHERE protocol(a.href) = "http" AND doc(a.href) = null
```

This query scans all the http anchors in a page and returns the value of the HREF field for those that point to unexistent documents (when a URL is valid, the function *doc* returns the tuple describing the document; otherwise it returns null). With a slightly different query, we can find all the broken links in pages that can be reached by recursively following links from the starting URL:

Query 11:

```
SELECT d.url, a.href
FROM Document d SUCH THAT "http://x.y.z/TheStartingPage.html" ->* d,
     Anchor a SUCH THAT base = d
WHERE protocol(a.href) = "http" AND doc(a.href) = null
```

This query gives us a list of pairs of URLs such that the second one is a broken link in the page pointed to by the first one. Observe that we have only added to the previous query a component to the FROM clause for iterating over all the local documents that can be reached from the starting one, and we have added to the result a field that identifies the document where the broken link occurs.

4.2 Defining the content of a full-text index

Many researchers or research groups publish web pages that give access to their on-line publications. These pages typically contain, for each available publication, a piece of text

describing it and the URL of the corresponding Postscript file. Thus, we can regard each source of on-line publications as a set of pairs *(url, text)*, and we can build a full-text index based on the descriptive text. Naturally, we want to use a WebSQL query to obtain the set of pairs for each source. We show how to do this below.

One of the authors (Mendelzon) describes his on-line publications in a web page with entries of the form:

```
<A HREF="ftp://ftp.db.toronto.edu/pub/papers/cascon95-web-hyplus.ps.Z">
 Masum Hasan, Dimitra Vista, Alberto Mendelzon,
 <B>Visual Web Surfing with Hy+,</B>
 in
 <EM>Proc. CASCON'95, Toronto, November 1995. </EM>
</A>
```

As we can see, all the descriptive data for a paper is in the label of the hyperlink that points to the corresponding Postscript file. This organization allows us to use the following simple query to retrieve the pairs *(url, text)* describing Mendelzon's publications:

Query 12:

```
SELECT a.href, a.label
FROM Anchor a SUCH THAT base = "http://www.cs.utoronto.ca/~mendel/papers.html"
```

On the other hand, researchers from the University of Pennsylvania working on database programming languages use a fairly different style of description. An index page provides, for each paper, its title and a hyperlink labeled "See here for the abstract", and each abstract page provides the title, authors, publication and abstract of the paper, along with a hyperlink labeled "See here for the paper", which points to the corresponding Postscript file. Thus, we can use Query 13 to build the set of pairs *(url, text)* for this source.

Query 13:

```
DEFINE LINK [here] AS label CONTAINS "here";
SELECT e.url, d.text
FROM Document d SUCH THAT "www.cis.upenn.edu/~db/langs/allpapers.html" [here] d,
     Document e SUCH THAT d [here] e
```

When the descriptive text for the papers of a source is not completely contained in hyperlink labels or in separate documents, but is a piece of text that surrounds each hyperlink, we can use the *context* attribute to extract it. For example, the Computer Science Department of Tel-Aviv University provides a web page describing the available on-line publications which is organized as an HTML list with entries of the form:

```
<LI> S. Abiteboul, S. Cluet, T. Milo,
<A HREF="http://www.math.tau.ac.il/~milo/sigmod95.ps">
  A Database Interface for Files Update
```

```
</A>.
<i> Proc. ACM SIGMOD Int. Conf. on Management of Data 1995 </i>
San Jose, May 1995.
```

The label of the hyperlink contains only the title of the document, and the other details about the paper surround the hyperlink. Since each of these entries is an element in a list, we can characterize the context of each hyperlink as the piece of text between two consecutive LI tags, and we can retrieve the set of pairs *(url, text)* for the Tel-Aviv source with the query:

Query 14:

```
DEFINE CONTEXT BEGIN = <LI>, END = <LI>;

SELECT e.href, e.context
FROM Anchor e
SUCH THAT base = "http://www.math.tau.ac.il/~milo/dept/papers.html"
WHERE e.url CONTAINS ".ps.Z"
```

4.3 Finding References from Documents in Other Servers

Suppose that we have a page with some links to pages in other sites and we want to know if our site is referenced from those pages or from the pages referenced by them. The following query would do it:

Query 15:

```
SELECT d.url
FROM Document d SUCH THAT "http://x.y.z/TheStartingPage.html" => d,
     Document d1 SUCH THAT d =>|->=> d1,
     Anchor a SUCH THAT base = d1
WHERE server(a.href) = "our.server"
```

4.4 Mining Links

Most tools for searching and indexing the Web only exploit the text in a page, ignoring another rich source of information: the links between pages. Ellen Spertus argues [Spe97] that the information in links can be exploited in several useful ways.

For instance, using just syntactic information, one can classify hypertext links within a site as *upward* in the file hierarchy, *downward* or *crosswise* and links to other sites as *outward*. One can then use this classification in the definition of several heuristics for inferring the topic of a page based on the topic of a starting page and the link between the starting page and this page. For example, Spertus formulates the following heuristic:

Heuristic 1 (Yahoo Closure). Starting at a Yahoo page, following downward or crosswise links leads to another Yahoo page whose topic is a specialization of the original page's topic.

This heuristic can be expressed in WebSQL in a straightforward manner, by defining appropriate link types:

Query 16:

```
DEFINE LINK [downward] AS
server(href) = server(base) AND path(href) CONTAINS path(base);

DEFINE LINK [crosswise] AS server(href) = server(base)
AND NOT (path(href) CONTAINS path(base) OR path(base) CONTAINS path(href));

SELECT x.url
FROM Document x
SUCH THAT "http://www.yahoo.com" ([downward] | [crosswise])* x
```

It is worth mentioning that, although a page might be reachable through several paths, the query engine avoids revisiting it. Also, the graph traversal algorithm guarantees simple paths, thus avoiding cycles.

Among the applications envisioned by Spertus to demonstrate the value of topic inference we can mention: finding moved pages, finding similar pages, avoiding irrelevant pages, finding pages with a given topic or type, finding persons and finding social chains. For each of these applications, the author proposes a search heuristic that makes use of link information. We believe that the development of these tools can be substantially simplified by using WebSQL as the underlying Web search mechanism.

5 Implementation

This section presents the current implementation of the WebSQL compiler, query engine, and user interfaces.

Both the WebSQL compiler and query engine are implemented as a set of Java [SM] classes, which form the *WebSQL class library*. The library can be used from any Java program.

The WebSQL system architecture is depicted in Figure 2.

The Compiler and Virtual Machine. The WebSQL compiler parses the query and translates it into a nested loop program in a custom-designed object language. The object program is executed by an interpreter that implements a stack machine. Its stack is heterogeneous, that is, it is able to store any type of object, from integers and strings to whole vectors of Document and Anchor tuples. The evaluation of the range specified in the FROM clause is done via specially designed operation codes whose results are vectors of Document or Anchor tuples.

The Query Engine. Whenever the interpreter encounters an operation code corresponding to a range specifying condition, the query engine is invoked to perform the actual evaluation. Depending on the type of condition, this involves either sending a request to index servers or a depth-first traversal of a sub-part of the document network.

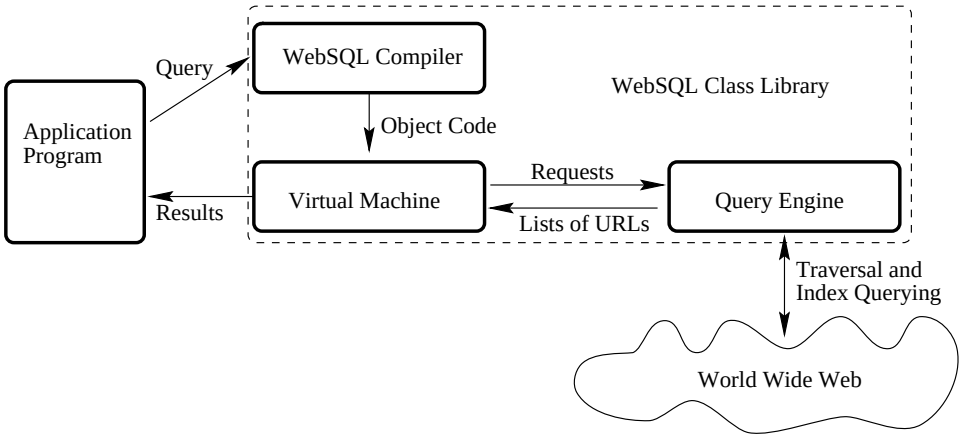

Figure 2: The Architecture of the WebSQL System

We have developed three different interfaces for interactive use. The simplest interface is an HTML form connected to a CGI script. The user can either fill in the form to assemble a query or type a complete WebSQL query directly. When the Submit button is pressed, the query is sent to the CGI script that invokes a stand-alone Java application running on our server. This application parses the query, and if no errors are found, hands it over to the query execution engine, which produces the result as a list of tuples that gets formatted into an HTML table and is shipped back to the user. This interface, although slow and with limited user interaction, has the advantage that it can be used from any browser.

For Java-aware browsers (such as Netscape 3.0), we have developed a much more user-controllable front-end in the form of a Java applet. Figure 3 shows a snapshot of this applet running in a Netscape browser. Among the features of this applet we can mention:

- several query templates for typical tasks are provided; a complete query is obtained by filling in parameters such as a URL or a keyword;
- the result tuples are computed (and showed in a separate frame) incrementally, either one by one or in groups of five;
- the execution can be interrupted at any time;
- if the query result contains URLs, one can click on them and the corresponding document is shown in a separate browser window;

Due to security restrictions in accessing remote servers, an applet cannot compute the results of a query directly. Therefore, we designed this applet using the *client-server* paradigm; we have split the application into two separate processes: the applet itself, responsible for the user interaction, and a server process (running on our local Web server) that does the actual processing. The applet communicates with the server through a custom designed protocol that gives the user total control over the query execution.

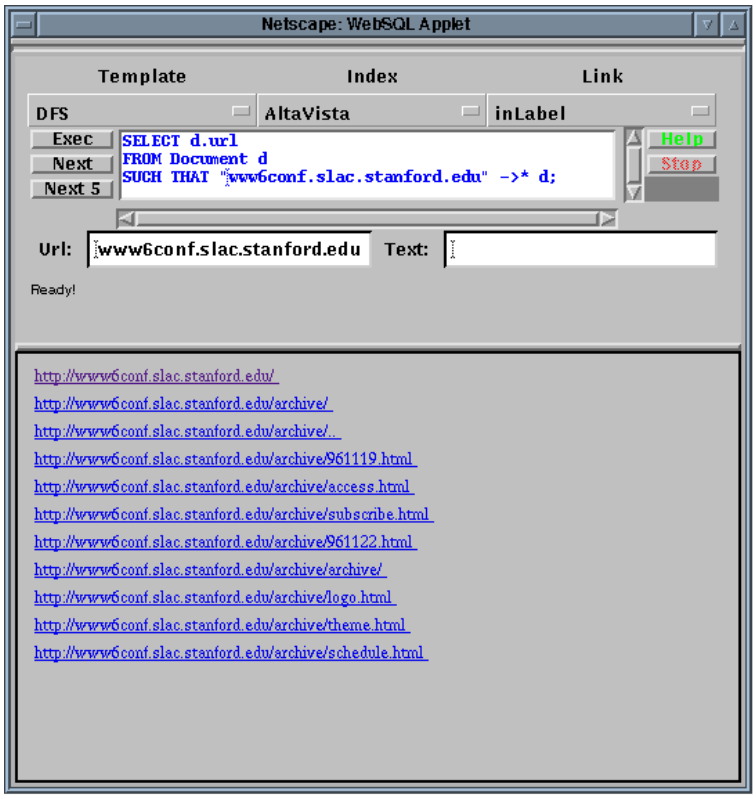

Figure 3: The WebSQL Applet

Finally, we have yet another interface, this time a stand-alone Java application that, once installed on the user's machine, runs locally and provides similar functionality as the applet.

Also, we are making available (in compiled form) the entire library of Java classes needed to develop applications on top of our system, together with Java source files for some sample applications using WebSQL. This library can be thought of as an application programming interface (API) to the WebSQL engine. Our declared intention is to provide a tool for programmatic access to the Web which can be used in the same way as embedded SQL is used for querying conventional databases.

All these interfaces, together with the complete syntax specification of the language and a comprehensive collection of examples are publicly available from the WebSQL home page [AM].

6 Using the WebSQL Class Library

The WebSQL class library has been designed in such a way as to allow programmers to easily embed WebSQL queries in their applications. In this section we will explain, using an example, how to execute a query from within a Java program and how to retrieve the

answer tuples.

The code in Figure 4 defines a Java class called Example which executes a hard-coded WebSQL query and then prints the answer tuples.

In order to execute a WebSQL query, an instance of the WebSQLServer class is created with the statement:

```
eng = new WebSQLServer(query, mon, 0);
```

where `query` is a string variable containing the actual query, `mon` is an instance of `DefStatusMonitor` (used for tracing the query execution) and the third parameter is the maximum number of HTTP requests that can be made during the execution of the query (zero means unlimited number).

The `elements()` method of class `WebSQLServer` returns an instance of the familiar interface `Enumeration`, which provides access to the tuples of the answer through an iteration mechanism similar to the cursors used to access tuples from relational tables. Each invocation of `nextElement()` triggers the computation of a new tuple, which is represented as a `Vector` object. Documents are retrieved from the Web on demand, when they are needed to compute the value of the next tuple.

7 Conclusions

We have overviewed the main features of WebSQL, a declarative language for querying the World Wide Web introduced in [Mih96] and [MMM96] and we have illustrated some practical applications of the language. Thus, we have shown how WebSQL can be used to automate routine Web site maintenance procedures, such as finding broken links in a collection of documents, finding references in documents in other servers to local pages, and defining the scope of a full-text index.

We described our prototype Java-based implementation of a WebSQL query execution engine together with several user interfaces and the embedded WebSQL application programming environment. Although fully accessible from the Web, our complex user interfaces are not meant to replace simple keyword-based search engines. Just as SQL is by and large not used by end users, but by programmers who build applications, we see WebSQL as a tool for helping build Web-based applications more quickly and reliably.

In our data model the text of an HTML document is a monolithic object, and therefore its analysis and interpretation is limited to simple text matching techniques. This is due to the fact that HTML is designed for presentation, not for describing semantic content, which makes information extraction from arbitrary documents difficult. We are currently working on extensions of the language that will make use of the document structure when it is known.

There is also a great deal of scope for query optimization. We do not currently attempt to be selective in the index servers that are used for each query, or to propagate conditions from the WHERE to the FROM clause to avoid fetching irrelevant documents. It would also be interesting to investigate a distributed architecture in which subqueries are sent to remote servers to be executed there, avoiding unnecessary data movement.

```
import java.util.*;
import WebSQL.*;

public class Example{
  public static void main(String args[]) {
    WebSQLServer eng = null;
    DefStatusMonitor mon = new DefStatusMonitor();

    String query = "select x.url, x.title "
                 +"from document x such that x mentions \"Java\";";

    try{
      System.err.println("Initializing WebSQL server ...");
      eng = new WebSQLServer(query, mon, 0);
    }catch(Exception e){
      System.err.println("Couldn't create server.");
    }

    // Print the tuples in the result
    for (Enumeration e = eng.elements(); e.hasMoreElements(); ) {
      // Retrieve the next tuple
      Vector tuple = (Vector) e.nextElement();

      //Dump it to standard output
      for (int i = 0; i < eng.tupleSize; i++)
        // Print the i-th component of the tuple
        System.out.print(tuple.elementAt(i) + " ");
      System.out.println();
    }
  }
}
```

Figure 4: A simple embedded WebSQL Java application

References

[AM] G.O. Arocena and G.A. Mihaila. WebSQL Home Page. http://www.cs.toronto.edu/~websql.

[Mih96] G. A. Mihaila. WebSQL - an SQL-like Query Language for the World Wide Web. Master's thesis, University of Toronto, 1996.

[MMM96] A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. In *Proc. of PDIS'96*, 1996.

[MSG97] Udi Manber, Mike Smith, and Burra Gopal. WebGlimpse - Combining browsing and searching. In *Proceedings of the 1997 Usenix Technical Conference*, 1997. ftp://ftp.cs.arizona.edu/people/udi/webglimpse.ps.Z.

[SM] Sun Microsystems. Java (tm): Programming for the Internet. http://java.sun.com.

[Spe97] Ellen Spertus. ParaSite: Mining Structural Information on the Web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 201–211, Santa Clara, CA, April 1997.