WebOQL: Restructuring Documents, Databases and Webs

Gustavo O. Arocena, Alberto O. Mendelzon
Department of Computer Science, University of Toronto
{gus, mendel}@db.toronto.edu

Abstract

The widespread use of the Web has originated several new data management problems, such as extracting data from Web pages and making databases accessible from Web browsers, and has renewed the interest in problems that had appeared before in other contexts, such as querying graphs, semistructured data and structured documents. Several systems and languages have been proposed for solving each of these Web-data management problems, but none of these systems addresses all the problems from a unified perspective. Many of these problems essentially amount to data restructuring: we have information represented according to certain structure and we want to construct another representation of (part of) it using a different structure. We present the WebOQL system, which supports a general class of data restructuring operations in the context of the Web. WebOQL synthesizes ideas from query languages for the Web, for semistructured data and for website restructuring.

1 Introduction

The widespread use of the Web has originated many new data management problems and has renewed the interest in problems that had been addressed before in other contexts. Among the new problems we can mention: Web querying [16, 17, 18] (i.e., declaratively expressing how to navigate one or more portions of the Web to find documents with certain features), Web-data warehousing [15] (i.e., extracting data from Web pages to populate a database, possibly for integrating the data with data from other sources) and website restructuring [7, 13] (i.e., exploiting the knowledge about the organization of highly structured websites for defining alternative views over their content).

Problems that have been revisited due to the popularity of the Web include: querying structured documents [1, 12, 14], querying semistructured data [3, 8] and querying graphs [19].

Many systems and languages have been proposed for solving each of these Web-data management problems, but none of these systems provides a framework for approaching the problems from a unified perspective. Moreover, none of these systems provides a combination of architecture, data model and query language that makes possible to effectively extract information from on-line structured documents without building custom-tailored programs.

In this paper we present the WebOQL system, whose goal is to provide such a framework. The WebOQL data model supports the necessary abstractions for easily modeling record-based data, structured documents and hypertexts. The query language allows us to restructure an instance of any of these three types of objects into an instance of any other one.

WebOQL synthesizes ideas from query languages for the Web, for semistructured data and for website restructuring and makes several contributions, most notably, the idea of querying documents by manipulating their abstract syntax trees and the support of the concept of *web* as a data type. The usual approach to querying structured documents is to use custom-tailored wrapper programs that map documents to instances of some data model [1, 7, 12, 14, 15]; the main disadvantage of this approach is that a wrapper program must be built for each new document (or set of documents with similar structure), usually using either a parser generator or a Perl-like filtering language. In WebOQL, an abstract syntax tree for every document of the same family (e.g. HTML) is built by the same wrapper, whatever the structure of the document might be; the query language is powerful enough to query or restructure these trees in a variety of ways.

In WebOQL webs are an abstraction supported by the data model; a web can be used to model a small set of related pages (for example, a manual), a larger set (for example, all the pages in a corporate intranet) or even the whole WWW. Having webs as "first-class citizens" is the key for expressing many restructuring operations.

The features mentioned above enable the development of many useful applications such as: querying small databases represented as documents (catalogs, price listings, touristic guides, etc.), restructuring single pages (for example, converting a large page into a set of smaller hyperlinked pages), restructuring sets of pages (for example, given a set of pages, creating an index page containing a hyperlink to each of them, and adding to each of the original pages a hyperlink pointing to the index page) and integrating information extracted from heterogeneous Web sources (for example, extracting headlines from several on-line news sources).

WebOQL's architecture is based on the common "middleware" approach to data integration used in several other projects [3, 13], that is, the use of a flexible common data model and wrappers that map data represented in terms of the sources' models to the common model. This facilitates the integration of information from other sources, like databases and file systems.

1.1 Related Work

As mentioned above, WebOQL synthesizes ideas from diverse research areas. Below is an overview of similarities and differences with several systems.

Web Queries. With Web query languages, such as WebSQL [6, 18], W3QS [16] and WebLog [17], we share the idea of viewing the Web as a database that can be queried using a declarative language. But these languages suffer from a common limitation: lack of support for exploiting document structure. An early attempt of exploiting document structure is present in WebLog, but it is applicable only to documents with a simple, flat structure. WebOQL's navigation patterns are a generalization of WebSQL's path regular expressions. Like in W3QS, in WebOQL it is possible to traverse trees and graphs using either depth-first or breadth-first search.

Semistructured Data. The main obstacles to exploiting the internal structure of Web documents are the lack of a schema or type and the potential irregularities that can appear for that reason. The problem of querying data whose structure is unknown or irregular has been addressed, although not in the context of the Web, by the so-called query languages for semi-structured data Lorel [3] and UnQL [8]. These systems use a very low-level representation of data, based on graphs. UnQL's data model was influential in our design. A problem with semistructured data models so far is that they provide very few modeling abstractions (essentially, only labeled graphs). Notably, they do not support ordered collections. We believe that the necessary flexibility required for modeling loosely structured information should not imply the lack of support for basic abstractions such as records, nesting, references and ordering. A schema-free data model that reflects this belief is, in fact, one of the contributions of our work. The explicit support of order is a key element for modeling structured documents; references allow us to model hyperlinks between documents; using records we can easily represent relational tables without needing to devise ad-hoc encodings to simulate them.

Website Restructuring. On the other hand, in order be able to express the kind of restructurings we mentioned above, the query language must be capable not only of manipulating the structure of documents, but also of providing a mechanism for generating arbitrarily linked sets of documents. Such facility is present in website restructuring systems like Araneus [7] and Strudel [13]. These systems exploit the knowledge of a website's structure for defining alternative views over its content. Araneus' approach is highly typed: pages in the website must be classified and formally described before they can be manipulated; in WebOQL we favor a more dynamic approach, in which the structure of pages is captured in the queries themselves; furthermore, WebOQL is capable of querying pages with irregular structure and pages whose structure is not fully known. Strudel uses a graph-based data model, where nodes represent documents, i.e., it does not model the internal structure of documents. An interesting result is that Strudel's query language exactly captures all queries expressible in first-order logic extended with transitive closure. WebOQL can compute transitive closure, but the characterization of its expressive power is not fully precise yet. Both Araneus and Strudel handle URLs similarly to *oid*s in OODBMSs: they provide facilities for creating URLs using "Skolem functions" [2], and for assigning URLs to documents. In WebOQL, URLs are just strings. As we will see, this approach is very flexible and simpler. Queries can generate URLs just by catenating other strings.

Structured Documents. The idea of querying structured documents has been previously investigated in [14], in the context of office information systems, and in [1], in the context of the integration of SGML with databases. Although largely different from one another, both approaches are strongly typed. In [1], documents are mapped to an instance of an object oriented database by means of semantic actions attached to a grammar. Then the database representation can be queried using the query language of the database. A novel aspect of this approach is the possibility of querying the structure by means of path variables. In [14], documents are modeled using nested ordered relations. This model is similar to WebOQL's, except that it is strongly typed. The query language is a generalization of nested relational algebra.

1.2 The Rest of the Paper

In Section 2 we introduce WebOQL's data model and most aspects of the query language by means of a comprehensive list of examples. Although we have defined a formal semantics for the model [4], space limitations prevent us from presenting it in this paper. Rather, we will try to convey the intuition behind the model, and thus we will focus on the pragmatic aspects. In Section 3 we introduce webs and show how they can be used. In Section 4 we introduce features for manipulating documents and semistructured data; we also give an example of "Document Patterns", a formalism close in spirit to the concept of "Query by Example". In Section 5 we present the results of our preliminary work in characterizing WebOQL's expressive power. Finally, in Section 6 we present our conclusions, status of the current implementation and possible directions of future work.

2 WebOQL

WebOQL's data model is based on ordered trees; we can think of a web as a graph of trees. The goal of the query language is, in general, to be able to navigate, query and restructure graphs of trees.

2.1 A Tree-based Data Model

The level of abstraction in WebOQL's data model is not as light-weight as OEM [20] or similar models and not as heavy-weight as the more traditional schema-based models. Using an analogy from the compiler field, we can assimilate WebOQL's data model to an intermediate language used for optimizations: it is not as low level as machine language but, at the same time, not as high level as the source language. The main data structure provided by WebOQL is the *hypertree*, which we introduce below.

Hypertrees. Hypertrees are ordered arc-labeled trees with two types of arcs, internal and external. Internal arcs are used to represent structured objects and external arcs are used to represent references (typically hyperlinks) among objects. Arcs are labeled with records. The only atomic data type is the string. Figure 1 shows a hypertree containing descriptions of publications from several research groups.

In diagrams, we use full lines for internal arcs, and dotted lines for external arcs. External arcs cannot have descendants, and the record that labels them must have a field named *Url*. URLs are just strings (with no additional semantics). Interpretation of URLs is left to wrappers that connect WebOQL to the external world.

Hypertrees are a very useful data structure because they subsume the three abstractions we want to support: collections, nesting and ordering. Moreover, with the distinction between internal and external arcs, the notion of reference is also captured by our trees, and the fact that labels are records allows us to easily represent the ubiquitous collections of records. However, since there is no type associated to a node, the records in the outgoing arcs can be heterogeneous. Note, for example, that there is no *Publication* field for the paper "Cobol in AI" in Figure 1, whereas such field is present for the paper "Assembly for the masses."

When modeling information residing in the Web, a hypertree is likely to correspond to a document. But a hypertree can also represent a relational table, a Bibtex file, a directory hierarchy, etc. In the rest of the paper, we will often say *tree* instead of hypertree.

Webs. Although hypertrees are the key abstraction in WebOQL's world view, WebOQL supports a higher level abstraction that enables us to model sets of related hypertrees: the web. A *web* is a pair (t, F) consisting of a hypertree t and a function F that maps URLs to hypertrees. We refer to these two components as the *schema* and the *browsing function* of the web, respectively.

We say that the pair composed of a URL u and the hypertree $F(u)$ is a *page* in that web, and we say that $F(u)$ is the *content* of the page. The browsing function implicitly defines a graph, where the nodes are pages and there is an arc between node a and node b if the content of the page at

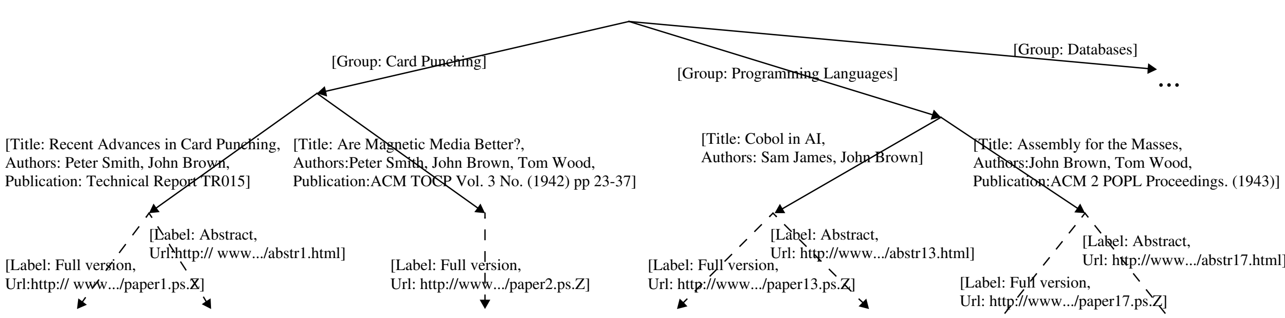


FIGURE 1. A Hypertree Containing a Publications Database

node *a* contains an external arc whose *Url* attribute is the URL of the page at node *b*. The schema of a web is likely to provide “entry points” to the graph. If the schema is null, then we must know one or more URLs to be able to enter the graph. A web can be used to model a small set of related pages (for example, a manual), a larger set (for example, all the pages in a corporate intranet) or even the whole WWW.

Both hypertrees and webs can be manipulated using WebOQL. In the next subsections we introduce the main features of the language by example. See [4] for a formal presentation of the data model and the query language, and see [5] for an on-line demo with live examples.

Simple Trees, Subtrees and Tails. Let us now define some terms we will use quite frequently in the sequel. Given a tree *t*, we say that the *tails of t* are the trees obtained by chopping off prefixes of *t*, the *simple trees of t* are the trees composed of one arc followed by a (possibly null) tree that stems from *t*’s root, and the *subtrees of t* are the trees at the end of the arcs that stem from *t*’s root. Figure 2 illustrates the ideas graphically.

2.2 The Query Language

As with the data model, the goals that guided the design of the query language were largely pragmatic. The overall goal of WebOQL is to perform complex restructuring operations. This implies the ability to build both deeply nested structures and arbitrarily linked hypertexts. However, WebOQL can only express feasible queries, i.e., queries of polynomial complexity. Regarding expressive power, WebOQL can simulate all operations in nested relational algebra and can compute transitive closure on an arbitrary binary relation.

First Example. The main construct provided by WebOQL is the familiar **select-from-where** (or, more briefly, **sfw**). Let us see an example of its use. Suppose that the name *csPapers* denotes the papers database in Figure 1, and that we want to extract from it the title and URL of the full version of papers authored by “Smith”. Query 1 shows how to do it. The result is displayed to the right of the query.

In Query 1, *x* iterates over the simple trees of *csPapers* (i.e., over the research groups) and, given a value for *x*, *y* iterates over the simple trees of the only subtree of *x* (i.e.,

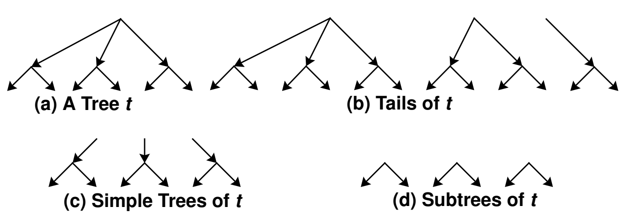
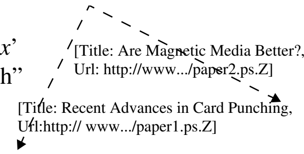


FIGURE 2. Tails, Simple Trees and Subtrees

***Q1*:** **select** [*y.Title*, *y*’*.Url*]
from *x* **in** *csPapers*, *y* **in** *x*’
where *y.Authors* ~ “Smith”



over the papers of the research group represented by *x*). The quote is the symbol for the *Prime* operator, which returns the first subtree of its argument. The dot is the symbol for the *Peek* operator, which extracts a field from the record that labels the first outgoing arc of its argument. The square brackets denote the *Hang* operator, which builds an arc labeled with a record formed with the arguments (in this example, the field names are inferred, but they can be explicitly indicated, as we will see in other examples). Finally, the tilde represents the string pattern matching predicate: its left argument is a string and its right argument is a *grep* string pattern.

The answer to a **sfw** query is obtained as follows: for each instantiation of the variables in the **from** clause (in the order induced by the trees from which variables take their values), check the condition in the **where** clause; if it is true, evaluate the query in the **select** clause and append its result to the answer.

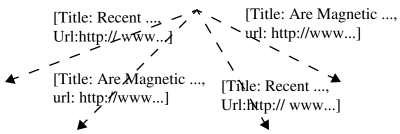
The **sfw** construct can be seen as a generalization of the *map* second order function found in functional programming languages.

Manipulating Trees. Queries need not involve the **sfw** construct. Like OQL [11], WebOQL is a purely functional language. In addition to the *Prime*, *Peek* and *Hang* operators introduced above, WebOQL provides three more tree operators. We introduce them in the next examples.

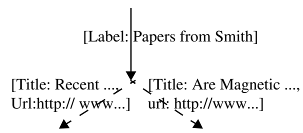
Concatenate, illustrated in Query 2, allows us to juxtapose two trees (*q1* denotes the result of Query 1). Query 3 illustrates the general form of the *Hang* operator, which takes a record and a tree as arguments, and “hangs” the tree from a new arc labeled with the record; when the tree argument is **null** (this constant denotes the null tree), we can elide it, along with the slash; thus, we can simply write ‘[*Tag*:“LI”]’ instead of ‘[*Tag*:“LI” / **null**]’; also, when the string value for a field is obtained from a *peek* operation, it is not necessary to explicitly give it a name, unless we want to rename it; for instance, we can write ‘[*x.Tag* / **null**]’, or simply ‘[*x.Tag*]’, instead of ‘[*Tag*:*x.Tag* / **null**]’. We can combine *Hang* and *Concatenate* operations to create trees purely from constants, as shown in Query 4. Note that this tree represents a fragment of HTML code composed by a list followed by an anchor.

Queries 6 and 7 illustrate the *Head* and *Tail* operators, which give us the first simple tree of a tree and all but the first simple tree of a tree, respectively. *Head* (resp. *Tail*) has an extended version, which allows us to get (resp. discard) the first *n* simple trees of a tree, for a nonnegative

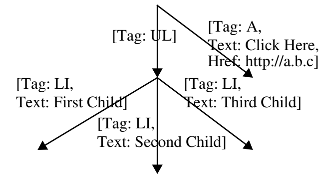
Q2*: *q1* + *q1



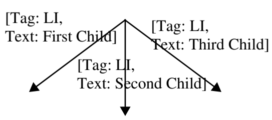
***Q3*: [*Label*:"Papers from Smith" / *q1*]**



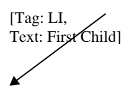
***Q4*: [*Tag*:"UL" /**
[*Tag*:"LI", *Text*:"First Child"] +
[*Tag*:"LI", *Text*:"Second Child"] +
[*Tag*:"LI", *Text*:"Third Child"]
] + [*Tag*:"A", *Href*:"http://a.b.c",
***Text*:"Click Here"]**



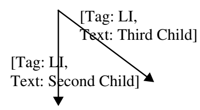
***Q5*: *q4* '**



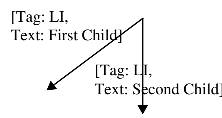
***Q6*: *q5* &**



***Q7*: *q5* !**



***Q8*: *q5* & 2**



integer *n*. Query 8 illustrates how to get the first two simple trees of a tree.

As we explained above, *Peek* allows us to extract a field from an arc's label. For example, '*q1.Title*' is the string "Recent Advances in Card Punching". If the cited field does not exist, *Peek* returns **nil**, which represents the value "undefined". For example, '*q1.Tag*' evaluates to **nil**. Any comparison against **nil** evaluates to false, even '**nil = nil**'. Related to the **nil** constant is the *isField* operator (denoted by the question mark), that tests for the presence of a field in an arc's label; for instance, '*q1*?*Text*' evaluates to *true*, whereas '*q1*?*Tag*' evaluates to *false*. This relaxed typing is useful when dealing with semistructured data.

3 Wrappers, URL Dereferencing and Webs

An important issue we have not yet addressed is: what is the input to a WebOQL query? The WebOQL approach to this issue is simple and flexible: URL dereferencing.

Dereferencing a URL means replacing it with the result of applying the browsing function of the current web to it (see Subsection 2.1). Every query is executed in the context of a web, which we refer to as the "current web". If not otherwise indicated, the current web is assumed to be the WWW plus the other data sources accessible via wrappers. But we can write queries that create new webs, and we can use them as the default web for the execution of further queries. We will see how in the next subsection.

If *u* is a URL, the result of the query '**browse**(*u*)' is the content of the page identified by *u*, according to the current web. For instance, '**browse**("http://www.w3c.org").*Tag*' returns the name of the tag associated to the first subtree of the W3C home page (see Subsection 4.1 to get an idea of what an HTML documents looks like in WebOQL). A URL *u* is considered *defined* in a web if **browse**(*u*) is nonnull in that web.

Unlike other proposals, where URLs are generally handled similarly to *oid*s in an object database, WebOQL URLs are simply strings. The interpretation of a URL is up to the wrappers connected to the system. In the current implementation, we use the convention that, if a URL to be dereferenced contains a colon, the prefix before the colon identifies a wrapper, and the suffix is the actual request to be sent to that wrapper. We have wrappers that map HTML documents, the file system hierarchy and relational tables to hypertrees. The mapping from an object to a hypertree can be done in one step or on demand; for instance, a relational table is mapped on demand, as its tuples are required for instantiating a variable during query execution.

Restructuring Webs. The previous examples illustrated how to perform tree restructuring. In the general case, a WebOQL query can not only restructure trees within a given web, but also restructure webs. A web restructuring query is a function that maps a web into another; the schema of the new web may be an arbitrary hypertree and the browsing function of the new web is obtained by redefining the value returned by the browsing function of the old web for a number of URLs (pages whose URL is not targeted by the query are left unchanged). As a particular case, the browsing function of the new web can just 'extend' that of the old web by associating nonnull hypertrees to URLs that were previously undefined.

The primary mechanism for creating webs is the **as** clause in the **sfw** construct. When we explained the semantics of **sfw**, we did not mention the fact that **sfw** creates a web, not just a tree. For instance, Query 1 is in reality shorthand for:

***Q9*: this | select** [*y.Title*, *y*'.*Url*] **as schema**
from *x* **in** *csPapers*, *y* **in** *x*'
where *y.Authors* ~ "Smith"

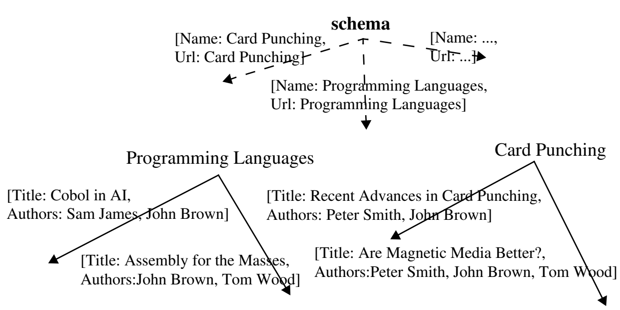
The **this** keyword denotes the current web and the vertical bar is the syntax for composing web queries (we informally refer to it as the *Pipe* operator, although it is not a real operator). **as schema** indicates that the result of the query will form the schema of a new web. In this case, the new web differs from the current web only in the schema. The **as** clause also allows us to define a new browsing function. We do this by specifying a URL instead of the keyword **schema**. For example, Query 10 creates a new page for each research group (using the group name as URL). Each page contains the publications of the corresponding group.

***Q10*: this | select** *x*' **as** *x.Group*
from *x* **in** *csPapers*

In general, the **select** clause has the form '**select** q_1 **as** s_1, q_2 **as** s_2, ... , q_m **as** s_m', where the q_i's are queries and each of the s_i's is either a string query or the keyword **schema**. The **as** clauses are evaluated from left to right; the ones containing the **schema** keyword specify how to create the schema of the new web, whereas the ones containing strings (which are interpreted as URLs) specify how to create the pages in which the old and the new webs differ.

The next example clarifies the idea. Suppose that we want to generate, from the *csPapers* tree, a web containing one page for each research group, consisting of the title and author of all its publications, and an index page, that lists all the groups and provides links to their pages. This is what Query 11 does. In the diagram representing the result, we put the URL of each page just on top of its content, and we omitted all pages whose content did not changed (which could amount to the whole WWW).

***Q11*:** *newWeb* ← **select unique**
[*Name*:*x.Group*,*Url*:*x.Group*] **as schema**,
[*y.Title*, *y.Authors*] **as** *x.Group*
from *x* **in** *csPapers, y* **in** *x*'



When the **select** keyword is followed by the **unique** keyword, then none of the trees built by **sfw** will contain two outgoing arcs with the same label. Only the first occurrence of an arc with a given label is kept in the answer; the duplicates, along with the trees that hang from them are eliminated (in our example, **unique** guarantees that one arc per group is added to the index page, instead of one per each paper). In Query 11, we used an arrow to assign a symbolic name to the newly created web. This naming facility is not part of the query language; it is analogous to a macro definition.

Composing Web Restructurings. A natural question at this point may be: once we compute a new web, what can we do with it?. There are two primary uses for a web: querying it (i.e., performing further restructurings) or returning it to the host application (for example, for the application to make the web's pages visible to a browser). Suppose we want to make the pages resulting from Query 11 visible to a browser. Since these pages do not specify the formatting details for presenting their content in HTML, there must exist either an application program that translates all the pages to HTML using a fixed formatting style (for example, HTML tables) or an application program tailored to format the output of this particular query. But instead of returning the web resulting from Query 11, we can create a new web where the pages created by Query 11 are restructured to contain HTML formatting tags. This is what Query 12 does. Two of the resulting HTML pages are displayed in Figure 3.

***Q12*:** *newerWeb* ← *newWeb*
| **select** [*Tag*: "H3", *Text*: *y.Title*] +
[*Text*: *y.Authors*] + [*Tag*: "HR"]
as *x.Url*
from *x* **in schema,** y **in browse**(*x.Url*)
| **select** [*Tag*: "H2", *Text*:"Publications of the " * *x.Name* * " Group"]
+ browse(*x.Url*) +
[*Tag*: "A", *Text*: "To Index", *Href*: "Index of Projects.html"]
as *x.Url* * ".html"
from *x* **in schema**
| **select** [*Tag*: "H2", *Text*: "Index of Projects"] +
[*Tag*: "UL" /
select [*Tag*: "LI" /
[*Tag*:"A", *Text*:*x.Name*, *Href*:*x.Url* * ".html"]
]
from *x* **in schema**
] **as** "Index of Projects.html"

Let us analyze how Query 12 works. *newWeb* is piped into the first **sfw** query (i.e., it is used as the current web during the evaluation of that query), which restructures each of the project pages by adding HTML formatting to the different fields (see Figure 3b); note that **browse**(*x.Url*) is a use of a page with URL *x.Url*, whereas *x.Url* appearing after **as** is a definition of a new page with this URL. The second **sfw** query simply adds a heading and a link pointing

```
<H2> Index of Projects </H2>
<UL>
<LI>
<A HREF ="Card Punching.html">
   Card Punching
</A>
</LI>
<LI>
<A HREF ="Programming Languages.html">
   Programming Languages
</A>
</LI>
<LI> ...
</UL>
```

(a) "Index of Projects.html"

```
<H2>Publications of the
Card Punching Group </H2>
<H3> Recent Advances in Card Punching </H3>
      Peter Smith, John Brown
<HR>
<H3> Are Magnetic Media Better ?</H3>
      Peter Smith, John Brown, Tom Wood
<HR>
<A HREF ="Index of Projects.html">
      To Index
</A>
```

(b) "Card Punching.html"

FIGURE 3. Result of Query 12 in HTML

to the index page to each of the group pages; the star symbol denotes the string concatenation operation. Finally, the last query creates an HTML page for the index by converting the schema to an HTML unordered list preceded by a heading.

It is worth mentioning some details before continuing: when a **sfw** query is used in a context where a tree is expected, the schema of the resulting web is taken as the value of the query. Conversely, when a tree query is used in a context where a web is expected, the result of the query is interpreted as a redefinition of the schema of the current web. **void** denotes the empty web, which is composed of a null hypertree and a browsing function that evaluates to **null** for any argument. **void** allows us to create "closed" webs, which have no access to external data.

4 Documents and Semistructured Data

Web documents are often cited as examples of semistructured data, since their structure is not constrained by a schema and may present irregularities. In this section we show how we can model and manipulate documents in WebOQL.

4.1 Modeling Structured Documents

The novel aspect of the modeling technique we present is that, as opposed to other proposals, we do not rely on custom-tailored external programs for mapping each document to an instance of the data model. One of the wrappers in the current implementation of WebOQL generates annotated abstract syntax trees (*AST*s) from arbitrary HTML documents. We can then effectively manipulate documents (or set of hyperlinked documents) since, in most documents, the physical structure implied by markup reflects the logical relationships between information items.

Figure 4 presents three views of an HTML document containing descriptions of publications (given that the whole tree does not fit in the page, we have omitted several portions and used ellipsis instead).

The rules for generating the *AST*s are mostly self-evident: each arc corresponds either to a subdocument enclosed in an occurrence of a paired tag (for example, the root arc of the tree in Figure 4 corresponds to the subdocument enclosed between <HTML> and </HTML>), to a nonpaired tag (like
), or to a piece of untagged text. A dummy tag named *NOTAG* is used in the latter case; this makes it possible to refer to untagged portions of text in queries (for example, to the titles of papers). Arcs corresponding to the *A* tag are external; all other arcs are internal. Internal arcs have three attributes: *Source*, *Text* and *Tag*, corresponding to the piece of HTML code, the text excluding markup and the tag of the subdocument, respectively. External arcs have one more attribute (*Url*) which corresponds to the destination of the anchor.

4.2 Restructuring Documents

The semistructured nature of documents makes it difficult to manipulate their components. Two features of WebOQL are particularly useful for addressing this problem: navigation patterns and tail variables.

Navigation Patterns. In the previous examples, variables have ranged over the simple trees of a tree. This is not the only possibility; in fact, it is the simplest one. In general, variables can range over subtrees located at any depth, and

```
<HTML>
<H1>Publications of Research Groups at CS Department</H1>
<H2> Card Punching </H2>
<UL>
<LI>
    <CITE> Recent Advances in Card Punching <BR>
    <B> Peter Smith, John Brown </B> <BR>
    Technical Report TR015 </CITE> <BR>
    <A HREF="http://.../paper1.ps.Z"> Full version </A>
    <A HREF="http://.../abtstr1.html"> Abstract </A> <BR>
</LI>
<LI>
    <CITE> Are Magnetic Media Better? <BR>
    <B> Peter Smith, John Brown, Tom Wood </B> <BR>
    ACM TOCP Vol. 3 No.1 (1942) pp 23-37</CITE> <BR>
    <A HREF="http://.../paper2.ps.Z"> Full version </A>
</LI>
</UL>

<H2> Programming Languages </H2>
<UL>
<LI>
    <CITE> Cobol in  AI <BR>
    <B> Sam James, John Brown </B> <CITE> <BR>
    <A HREF="http://.../paper13.ps.Z"> Full version </A>
    <A HREF="http://.../abstr13.html"> Abstract </A> <BR>
</LI>
...
<H2> Databases </H2>
...
</HTML>
```

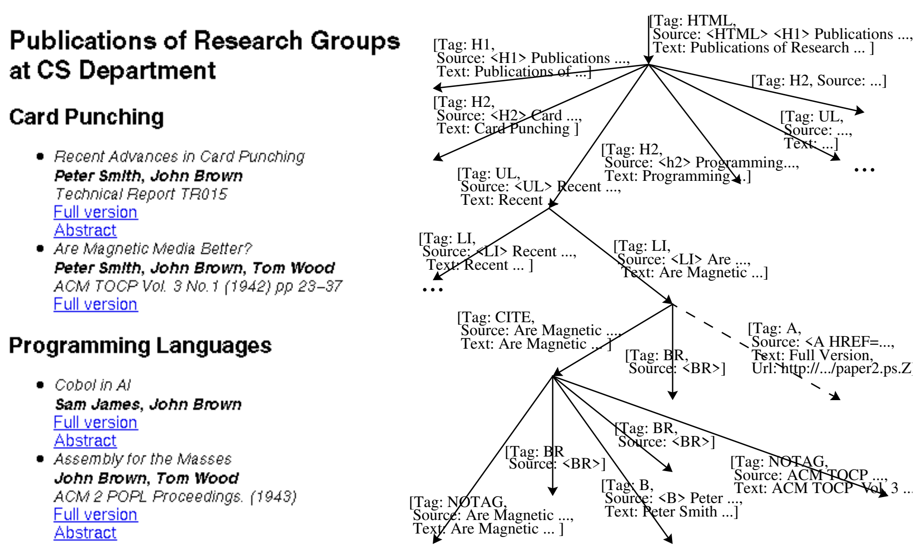


FIGURE 4. Three Views of an HTML Document

even over subtrees of several (linked) hypertrees. The range of variables can be specified using *navigation patterns* (*NP*s), which are regular expressions over an alphabet of record predicates; they allow us to specify the structure of the paths that must be followed in order to find the instances for variables.

*NP*s are mainly useful for two purposes. First, for extracting subtrees from trees whose structure we do not know in detail or whose structure presents irregularities. For example, we need not know the structure of the document in Figure 4 in detail to extract the names of all research groups; all we need to know is that these names are tagged with *H2*, as illustrated in Query 13.

***Q13*:** **select** [*x*.*Text*]
from *x* **in** "papers.html" **via** ^*[*Tag* = "H2"]

In the *NP* '^*[*Tag* = "H2"]', '^' and '[*Tag* = "H2"]' are record predicates: the first one is true of an arc if the arc is internal, and the second one is true if the arc has a *Tag* attribute with value "H2". Thus, this *NP* matches paths composed of any number of internal arcs (star, as usual, means Kleene closure) followed by an arc corresponding to a piece of text tagged with *H2*.

The opposite to ^ is >, which is true of an arc if the arc is external. Thus, for example, '[**not**(*Tag* = "TABLE")]*>' specifies all paths in a tree that lead from the root to an anchor not enclosed in a table.

*NP*s match paths starting at the root of the source tree. For each matching path *p*, the associated variable is instantiated to the simple tree (see Figure 2) starting at *p*'s last arc. When the *NP* is omitted (as we have done in earlier examples), [**true**] is assumed by default; thus, '*x* **in** *csPapers*' is shorthand for '*x* **in** *csPapers* **via** [**true**]'. Variables are instantiated following the order in which paths are matched during a left to right depth-first or breadth-first search (the default is breadth-first; to use depth-first, we write **viadfs** instead of **via**).

The second important use for *NP*s is for iterating over trees connected by external arcs. In fact, the distinction between internal and external arcs in hypertrees becomes really useful when we use navigation patterns that traverse external arcs. Suppose that we have a software product whose documentation is provided in HTML format and we want to build a full-text index for it. These documents form a complex hypertext, but it is possible to browse them sequentially by following links having the string "Next" as label. For building the full-text index we need to feed the indexer with the text and the URL of each document. We can obtain this information using Query 14:

***Q14*:** **select** [*x*.*Url*, *x*.*Text*]
from *x* **in browse**("root.html") **via** (^*[*Text* ~ "Next"]>)*

If an external arc is matched in the middle of a path, the *Url* attribute of this arc is dereferenced, and the navigation continues through the tree thus obtained. We can view this process as an on-demand materialization of the graph induced by the browsing function. Note that starred *NP*s can potentially traverse a large fraction of the WWW.

Tail Variables. The trees generated by Query 12 for each research group have a flat physical structure. However, its logical structure is that of a heading followed by a list of components, each one representing a paper (see Figure 3b). Suppose we want to restructure the list of papers for a group into an HTML ordered list. The language features we have seen so far do not enable us to express such a query. This problem (and others) can be solved in WebOQL by using *tail* variables: when we use a variable name beginning in uppercase, the variable iterates not over simple trees, but over tails (see Figure 2), i.e., instead of keeping just the first simple tree at the end of a matching path, we keep this simple tree and all the simple trees to its right. Using tail variables, we can express our query in this way:

***Q15*:** [*Tag*: "OL" **/**
select [*Tag*: "LI" **/** *X*&3]
from *X* **in browse**("Card Punching.html")!
where *X*.*Tag* = "H3"
]

Using tail variables we can also easily express queries such as "extract all the tables that are preceded by a heading containing the word *service*", or "build a list with all the subdocuments enclosed between two consecutive *HR*s". We present two more examples below.

Suppose we want to collect publications metadata available from documents like the one in Figure 4 to warehouse them in a local relational table with schema (*title, authors, publication, ps-url, abstract-url*). Assuming "http://a.b.c/papers.html" is the URL of the document in Figure 4, Query 16 restructures this metadata source into a set of records with the required schema:

***Q16*:** **select** [*title*: *y*''.*Text*,
authors: *y*''!!.*Text*,
publication: *y*''!!!!.*Text* ,
ps-url: *y*'!!.*Url*,
abstract-url: *y*'!!!!.*Url*] **as** "pubsDb: insert"
from *X* **in browse**("http://a.b.c/papers.html")', *y* **in** *X*!'
where *X*.*Tag* = "H2"

Variable *X* is successively instantiated to each tail whose first descendant is a group name and whose second descendant represents the list of papers for the group; *y* is then instantiated to each paper. Note that there is no abstract for the paper "Are Magnetic Media Better"; WebOQL handles irregularities like this one smoothly: instead of raising run-time errors, all invalid tree operations return **null**. Also note that we use the URL "pubsDb: insert" as the target for the result. As far as WebOQL semantics concerns, this string has no special semantics. However, the implementation can recognize the "pubsDb:" prefix and actually perform insertion operations into the database as the query is being executed.

Query 16 gives a feeling for how we can use WebOQL to integrate information extracted directly from HTML documents and use it to populate a local database. It is easy to imagine an example that works in the opposite direction, i. e., one that generates one or more HTML pages from the result of a query to a relational table.

A variation of query 16 restructures our HTML document into the *csPapers* tree we have used in the examples of Sections 2 and 3:

***Q17*:** *csPapers* ←
    select [*Group: X.Text* /
        select [*Title*:*y*'' .*Text*, *Authors*:*y*'' !!.*Text*,
                Publication:*y*'' !!!!.*Text* /
            [*Label:* "Full Version", *y*'!!.*Url*] +
            [*Label:* "Abstract", *y*'!!!!.*Url*]
          ]
        from *y* **in** *X*!'
      ]
    from *X* **in browse**("http://a.b.c/papers.html")'
    where *X.Tag* = "H2"

Note that we assign the name *csPapers* to the result; in the queries presented in Sections 2 and 3, we used the *csPapers* name as denoting a hypertree, thus implicitly referring to the schema of this web.

Document Patterns. After using WebOQL for extracting information from several on-line sources, we made two observations: first, for some documents, the queries may be fairly complex and difficult to read; second, subqueries with a common structure ("idioms") appeared rather frequently. Thus, we developed a pattern language that can be thought of as an incarnation of the concept of "Query by Example" applied to documents. A document pattern is composed of HTML tags, string patterns, variables and a few other syntactic devices. The pattern in Figure 5 restructures the document in Figure 4, eliminating the classification into groups and making the title the label of an anchor that points to the full version of the paper.

A document pattern specifies a mapping between two webs. The construct between the *USING* and *GIVING* keywords is the input pattern, and the construct between the *GIVING* and *END* keywords is the output pattern. Intuitively, the ellipsis mean "search through the document structure", the curly brackets mean "repeat the application of this pattern" and ANY is a "wildcard" that matches any simple tree. Patterns are automatically translated to WebOQL queries.

```
SCAN
      "http://a.b.c/papers.html"
USING
      . . . <LI>
            <CITE> Title ANY <BR>
            <B> Authors </B> <BR>
            Publication ANY
            </CITE> <BR>
            <A HREF=FullUrl> ANY </A>
         </LI>
GIVING
      <H2> "Publications of all Groups" </H2>
      {
         <A HREF=FullUrl> Title </A> <BR>
         <I> Title </I> <BR>
         Authors <HR>
      }
END
```

FIGURE 5. A Document Pattern

5 Complexity and Expressive Power

The complexity of any WebOQL query is polynomial in the size of the input. This is easy to see for all operations (and compositions thereof) except **sfw** operations containing *NP*s and/or several **as** clauses. Finding all nodes reachable through paths that match a *NP* (starting from a given tree) has polynomial cost [19], and a query can create a number of documents which is polynomial in the size of the input. Thus the composition of queries that contain *NP*s and/or several **as** clauses is also polynomial.

WebOQL can simulate all nested relational algebra operators. For projection, selection, union, and cartesian product, the simulation is trivial. Difference can be simulated like in SQL, by nesting in the **where** clause. Queries 16 and 17 suggest how to simulate the *unnest* and *nest* operators of nested relational algebra, respectively. Transitive closure on an arbitrary binary relation can be simulated by first generating a web that represents the graph of the relation explicitly (that is, a page for each value and an external arc between two pages if the pair of corresponding values is in the relation) and then traversing this web using the *NP* '>*'.

6 Conclusions and Further Work

We have presented the WebOQL system, which is based on a language that supports a general class of data restructuring operations. WebOQL provides a framework for approaching many Web-data management tasks from a unified perspective. The data model supports abstractions, such as records, ordered trees, hyperlinks and webs, that allow us to easily model Web data, and the query language provides powerful primitives for tree and web restructuring and hypertext navigation. Both the data model and the query language are flexible enough for accommodating lack of knowledge of the structure of the data to be queried and potential irregularities, or even lack of explicit structure in this data, which are common issues in the context of the Web. See [5] for an on-line demo containing live examples ranging from document restructuring to integration of information extracted from several on-line news sources.

We have implemented WebOQL and the document pattern translator in Java. WebOQL queries can be embedded in Java programs, and new wrappers can be dynamically added to the system. The WebOQL parser generates an internal algebraic representation for the queries. In particular, the **sfw** construct is translated to simpler operations of more algebraic nature. We then directly interpret the algebraic representation without performing optimizations. In fact, query optimization and techniques for efficient execution are the most likely sources of future work.

On the theoretical side, we are working on the formal semantics of document patterns and in a more precise characterization of WebOQL's expressive power. The presence of order, web creation and regular expressions makes this problem particularly challenging. The most appropriate formalism for analyzing WebOQL's expressive power seems to be structural recursion [9, 10]. Structural recursion forms are recursive definitions of systematic traversals of structured objects. Different forms of structural recursion yield query languages with different expressive power.

If we ignore web creation and tail variables, the expressive power of WebOQL lies between the *EXT* and *VEXT* forms of structural recursion proposed in [10]. For instance, the query "extract all anchors from the tree corresponding to an HTML document" cannot be expressed in *EXT*, whereas it can be expressed in WebOQL with *NP*s. On the other hand, *VEXT* allows us to simulate *NP*s and, more interestingly, allows to express queries like "change all the *H3* headings to *H2* headings in the tree corresponding to an HTML document"; this query cannot be expressed in WebOQL, basically because WebOQL cannot, in general, preserve the structure of the input in the result.

Tail variables are not captured by any of the structural recursion forms presented in [10], but a new form can be easily defined that captures them. Finally, the possibility of defining webs adds a new dimension to expressive power. For instance, it allows us to compute transitive closure on an arbitrary binary relation, something that, according to [8], seems not to be expressible by means of structural recursion.

Acknowledgement: this project was supported by the Information Technology Research Centre of Ontario and the Natural Sciences and Engineering Research Council of Canada.

References

[1] S. Abiteboul, S. Cluet, V . Christophides, T. Milo, G. Moerkotte, J. Simeon, *Querying Documents in Object Databases*, in Journal of Digital Libraries 1(1)5-19, 1997.

[2] S. Abiteboul, P. Kanellakis, *Object identity as a query language primitive,* in Proc. of ACM SIGMOD Int. Conf. on Management of Data, pp. 159-173, 1989.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J.L. Wiener, *The Lorel Query Language for Semistructured Data*, in Journal of Digital Libraries 1(1)68-88, 1997.

[4] G. Arocena, *WebOQL: Exploiting Document Structure in Web Queries*, Master's Thesis, University of Toronto, 1997.

[5] G. Arocena, *The WebOQL Home Page,* http://www.db.toronto.edu/~weboql/.

[6] G. Arocena, A. Mendelzon, G. Mihaila, *Applications of a Web Query Language*, in Proc. of 6th. Int. WWW Conference, Santa Clara, California, April 1997.

[7] P. Atzeni, G. Mecca, P. Merialdo, *Semistructured and Structured Data in the Web: Going back and Forth*, in Proc. of the Workshop on Semi-structured Data, Tucson, Arizona, May 1997.

[8] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, *A query language and optimization techniques for unstructured data,* in Proc. of ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada, pp. 505-516, 1996.

[9] P. Buneman, S. Davidson, D. Suciu, *Programming Constructs for Unstructured Data*, in Proc. of 5th Int. Workshop on DBPL, Gubbio, Sept. 1995.

[10] P. Buneman, S. Naqvi, V. Tannen and L. Wong, *Principles of Programming with Complex Objects and Collection Types,* in Theoretical Computer Science 149, pp. 3-48, 1995.

[11] R. Cattell (Ed.), *The Object database standard, ODMG-93*, Morgan Kaufmann Publishers, San Francisco, Calif., 1996.

[12] V . Christophides, S. Abiteboul, S. Cluet and M. Scholl, *From structured documents to novel query facilities,* in Proc. of ACM SIGMOD Int. Conf. on Management of Data, pp. 313-324, 1994.

[13] M. Fernandez, D. Florescu, A. Levy, D. Suciu, *A Query Language and Processor for a Web-Site Management System*, in Proc. of the Workshop on Semi-structured Data, Tucson, Arizona, May 1997.

[14] R. Güting, R. Zicari, D. Choy, *An algebra for structured office documents,* in ACM TOIS 7(2), pp. 123-157, 1989.

[15] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, A. Crespo, *Extracting semistructured information from the Web*, in Proceedings of the Workshop on Semi-structured Data, Tucson, Arizona, May 1997.

[16] D. Konopnicki, O. Shmueli, W3QS: *A query system for the World Wide Web*, in Proceedings of the 21th Int. Conf. on Very Large Databases, Zurich, pp. 54-65, 1996.

[17] L. Lakshmanan, F. Sadri, I. Subramanian, *A declarative language for querying and restructuring the Web*, in Proceedings of the 6th Int. Workshop on Research Issues in Data Engineering, New Orleans, 1996.

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

[19] A. Mendelzon, P. Wood, *Finding regular simple paths in graph databases*, SIAM J. Comp. 24:6, pp 1235-1258, 1995.

[20] Y. Papakonstantinou, H. Garcia Molina, J. Widom, *Object exchange across heterogeneous information sources*, in Proceedings of the 11th Int. Conf. on Data Engineering, Taipei, pp. 251-260, 1995.