Querying the World Wide Web

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

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

Tova Milo
Tel Aviv University
milo@math.tau.ac.il

Abstract

The World Wide Web is a large, heterogeneous, distributed collection of documents connected by hypertext links. The most common technology currently used for searching the Web depends on sending information retrieval requests to "index servers" that index as many documents as they can find by navigating the network. One problem with this is that users must be aware of the various index servers (over a dozen of them are currently deployed on the Web), of their strengths and weaknesses, and of the peculiarities of their query interfaces. A more serious problem is that these queries cannot exploit the structure and topology of the document network.

In this paper we propose a query language, WebSQL, that takes advantage of multiple index servers without requiring users to know about them, and that integrates textual retrieval with structure and topology-based queries. We give a formal semantics for WebSQL using a calculus based on a novel "virtual graph" model of a document network. We propose a new theory of query cost based on the idea of "query locality," that is, how much of the network must be visited to answer a particular query. We give an algorithm for characterizing WebSQL queries in terms of query locality. Finally, we describe a prototype implementation of WebSQL written in Java.

1 Introduction

The World Wide Web[BLCL+94] is a large, heterogeneous, distributed collection of documents connected by hypertext links. Current practice for finding documents of interest depends on *browsing* the network by following links and *searching* by sending information retrieval requests to "index servers" that index as many documents as they can find by navigating the network. The limitations of browsing as a search technique are well-known, as well as the disorientation resulting in the infamous "lost-in-hyperspace" syndrome. As far as keyword-based searching, one problem with it is that users must be aware of the various index servers (over a dozen of them are currently deployed on the Web), of their strengths and weaknesses, and of the peculiarities of their query interfaces. To some degree this can be remedied by front-ends that provide a uniform interface to multiple search engines, such as Multisurf[HGN+95], Savvysearch [Dre96], and Metacrawler [SE95].

A more serious problem is that these queries cannot exploit the structure and topology of the document network. In the hypertext literature, the need for such queries is well documented. For example, in an often-cited paper on the limitations of hypertext systems, Halasz says: [Hal88]

> Content search ignores the structure of a hypermedia network. In contrast, structure search specifically examines the hypermedia structure for subnetworks that match a given pattern.

and goes on to give examples where such queries are useful.

In the Web context, suppose for example that we are looking for an IBM catalog with prices for personal computers. A keyword search for the terms "IBM," "personal computer," and "price," using MetaCrawler, returns 92 references, including such things as an advertisement for an "I Bought Mac" T-shirt and the VLDB '96 home page. If we know the web address (called URL, or "uniform resource locator") of the IBM home page is `www.ibm.com`, we would like to be able to restrict the search to only sites located on the same server as the IBM home page and directly or indirectly reachable from this page. With currently available tools, this is not possible because navigation and query are distinct phases: navigation is used to construct the indexes, and query is used to search them once constructed. We propose instead a tool that can combine query with navigation. The emphasis must be however on *controlled* navigation. There are currently tens of millions of documents on the Web, and growing; and network bandwidth is still

a very limited resource. It becomes important therefore to be able to distinguish in queries between documents that are stored on the local server, whose access is relatively cheap, and those that are stored on remote serves, whose access is more expensive. It is also important to be able to analyze a query to determine its cost in terms of how many remote document accesses will be needed to answer it.

There are other reasons why queries that combine content and structure are important. We see a query language not primarily as an end-user tool, but as an aid in application development, just as SQL is used in combination with a programming language for building database applications. Many applications on the Web require accessing the structure defined by the links. For example, typical tasks in web site management include checking for dangling links, finding frequently used paths, finding unreachable documents, etc. In Section 6 we list other applications that would be facilitated by our language.

In this paper we propose a query language, *WebSQL*, that takes advantage of multiple index servers without requiring users to know about them, and that integrates textual retrieval with structure and topology-based queries. After introducing the language in Section 2, in Section 3 we give a formal semantics for WebSQL using a calculus based on a novel "virtual graph" model of a document network. In Section 4, we propose a new theory of query cost based on the idea of "query locality," that is, how much of the network must be visited to answer a particular query. We give an algorithm for characterizing WebSQL queries with respect to query locality. Finally, in Section 5 we describe a prototype implementation of WebSQL written in Java. We conclude in Section 6.

Related Work

There has been work in query languages for hypertext documents[BK90, CM89, MW93] as well as query languages for structured or semi-structured documents[GZC89, ACM93, CACS94, QRS+95]. Our work differs significantly from both these streams. None of these papers make a distinction between documents stored locally or remotely, or make an attempt to capitalize on existing index servers. As far as document structure, we only support the minimal attributes common to most HTML documents (URL, title, type, length, modification date). We do not assume the internal document structure is known or partially known, as does the work on structured and semi-structured documents. As a consequence, our language does not exploit internal document structure when it is known; but we are planning to build this on top of the current framework.

Closer to our approach are the W3QL work by Shmueli and Konopnicki[KS95] and the WebLog language of Lakshmanan *et al.* [LSS96]. Neither one provides a clean semantics or methods for query analysis, which we consider two of the main contributions in our work.

Regarding the language design, our motivation is very similar to W3QL, but the approach is substantially different. They emphasize extensibility and interfacing to external user-written programs and Unix utilities. While extensibility is a highly desirable goal when the tool runs in a known environment, we aim for a tool that can be downloaded to an arbitrary client and run with minimal interaction with the local environment. For this reason, our query engine prototype is implemented in the Java programming language [SM]. and (part of it) can be downloaded as an applet by a Java-aware browser. Another difference is that we provide formal query semantics, and emphasize the distinction between local and remote documents. This makes a theory and analysis of query locality possible. On the other hand, they support filling out forms encountered during navigation, and discuss a view facility based on W3QL, while we do not currently support either. In this regard, it is not our intention to provide a fully functional tool, but a clean and minimal design with well-defined semantics that can be extended with bells and whistles later.

Another recent effort in this direction is the WebLog language of Lakshmanan *et al.* [LSS96]. Unlike WebSQL, WebLog emphasizes manipulating the internal structure of Web documents. Instead of regular expressions for specifying paths, they rely on Datalog-like recursive rules. Their paper does not describe an implementation or formal semantics.

Finally, Abiteboul and Vianu [AV97] propose formal models of general web queries and discuss various notions of computability.

2 The WebSQL Language

In this section we introduce our SQL-like language for the World Wide Web. We begin with a simple relational model of the WWW. Then we give a few examples of queries and present the syntax of the language. The formal semantics of queries is defined in the following section.

One of the difficulties in building an SQL-like query language for the Web is the absence of a database schema. Instead of trying to model document structure with some kind of object-oriented schema, as in [CACS94, QRS+95], we take a minimalist relational approach. At the highest level of abstraction, ev-

ery Web object is identified by its Uniform Resource Locator (URL) and has a binary content whose interpretation depends on its type (HTML, Postscript, image, audio, etc.). Also, Web servers provide some additional information such as the type, length, and the last modification date of an object. Moreover, an HTML document has a title and a text. So, for query purposes, we can associate a Web object with a tuple in a virtual relation:

$$Document[url, title, text, type, length, modif]$$

where all the attributes are character strings. The *url* is the key, and all other attributes can be null.

Once we define this virtual relation, we can express any *content query*, that is, a query that refers only to the content of documents, using an SQL-like notation.

Example 2.1 *Find all HTML documents about "hypertext".*

SELECT $d.url, d.title, d.length, d.modif$
FROM Document d
SUCH THAT d MENTIONS "hypertext"
WHERE $d.type$ = "text/html";

Since we are not interested just in document content, but also in the hypertext structure of the Web, we make hypertext links first-class citizens in our model. In particular, we concentrate on HTML documents and on hypertext links originating from them. A *hypertext link* is specified inside an HTML document by a sequence, known as an *anchor*, of the form `<A HREF=`*href*`>`*label*`</A>` where *href* (standing for *hypertext reference*) is the URL of the referenced document, and *label* is a textual description of the link. Therefore, we can capture all the information present in a link into a tuple:

$$Anchor[base, href, label]$$

where *base* is the URL of the HTML document containing the link, *href* is the referred document and *label* is the link description [1]. All these attributes are character strings.

Now we can pose queries that refer to the links present in documents.

Example 2.2 *Find all links to applets from documents about "Java".*

SELECT $y.label, y.href$
FROM Document x
SUCH THAT x MENTIONS "Java",
Anchor y SUCH THAT $base = x$
WHERE $y.label$ CONTAINS "applet";

In order to study the topology of the Web we will want sometimes to make a distinction between links that point within the same document where they appear, to another document stored at the same site, or to a document on a remote server.

Definition 2.1 *A hypertext link is said to be:*

- interior *if the destination is within the source document;* [2]
- local *if the destination and source documents are different but located on the same server;*
- global *if the destination and the source documents are located on different servers.*

This distinction is important both from an expressive power point of view and from the point of view of the query cost analysis and the locality theory presented in Section 4.

We assign an arrow-like symbol to each of the three link types: let $\mapsto$ denote an *interior* link, $\rightarrow$ a *local* link and $\Rightarrow$ a *global* link. Also, let $=$ denote the empty path. Path regular expressions are built from these symbols using concatenation, alternation ($|$) and repetition ($*$). For example, $= | \Rightarrow \ . \ \rightarrow^*$ is a regular expression that represents the set of paths containing the zero length path and all paths that start with a global link and continue with zero or more local links.

Now we can express queries referring explicitly to the hypertext structure of the Web.

Example 2.3 *Starting from the Department of Computer Science home page, find all documents that are linked through paths of length two or less containing only local links. Keep only the documents containing the string 'database' in their title.*

SELECT $d.url, d.title$
FROM Document d SUCH THAT
"http://www.cs.toronto.edu" $= | \rightarrow | \rightarrow \rightarrow d$
WHERE $d.title$ CONTAINS "database";

[1] Note that *Anchor* is not, strictly speaking, a relation, but a multiset of tuples – a document may contain several links to the same destination, all having the same label.

[2] In HTML documents, links can point to specific named fragments within the destination document; the fragment name is incorporated into the URL. For example, `http://www.royalbank.com/fund.html#DP`; refers to the fragment named `DP` within the document with URL `http://www.royalbank.com/fund.html`. We will ignore this detail in the rest of the paper.

Of course, we can combine content and structure specifications in a query.

Example 2.4 *Find all documents mentioning 'Computer Science' and all documents that are linked to them through paths of length two or less containing only local links.*

SELECT $x.url, x.title, y.url, y.title$
FROM Document x SUCH THAT
x MENTIONS "Computer Science",
Document y
SUCH THAT $x = |\rightarrow|\rightarrow . \rightarrow\ y$;

```
Query := SELECT AttrList FROM DomainSpec
            [ WHERE Condition ] ;
AttrList := Attribute {, Attribute}
Attribute := Field | TableVar.Field
Field := Id
TableVar := Id
DomainSpec := DomainTerm {, DomainTerm}
DomainTerm := Table TableVar
                  SUCH THAT DomainCond
DomainCond := Node PathRegExp TableVar
          | TableVar MENTIONS StringConstant
          | Attribute = Node
Node := StringConstant
          | TableVar
Condition := BoolTerm {OR BoolTerm}
BoolTerm := BoolTerm {AND BoolTerm}
BoolTerm := Attribute = Attribute
          | Attribute = StringConstant
          | Attribute CONTAINS StringRegExp
          | (Condition)
PathRegExp := Link
          | PathRegExp *
          | PathRegExp PathRegExp
          | PathRegExp "|" PathRegExp
          | ( PathRegExp )
Link := = | #> | => | ->
```

Figure 1: A BNF Specification of WebSQL

Note we are using two different keywords, MENTIONS and CONTAINS, to do string matching in the FROM and WHERE clauses respectively. The reason is that they are used differently. MENTIONS is normally used with the first argument free and the second bound; that is, given a string, it returns the set of all documents that contain that string. CONTAINS is normally used with both arguments bound; it checks whether a given document contains a given string. This distinction is reflected both in the formal semantics of Section 3 and in the implementation. In fact, the implementation evaluates conditions in the FROM clause by sending them to index servers. The result of the FROM clause, obtained by navigation and index server query, is a set of candidate URL's, which are then further restricted by locally evaluating the conditions in the WHERE clause. This introduces opportunities for an optimization that moves local CONTAINS conditions to remotely evaluated MENTIONS conditions, avoiding unnecessary data movement across the network, but we do not discuss query optimization in this paper.

The BNF specification of the language syntax is given in Figure 1. The syntax follows the standard SQL `SELECT` statement. All queries are refer to the WWW database schema introduced above. That is, *Table* can only be `Document` or `Anchor` and *Field* can only be a valid attribute of the table it applies to.

3 Formal Semantics

In this section we introduce a formal foundation for WebSQL. Starting from the inherent graph structure of the WWW, we define the notion of *virtual graph* and construct a calculus-based query language in this abstract setting. Then we define the semantics of WebSQL queries in terms of this calculus.

3.1 Data Model

We assume an infinite set D of *data values*, and a finite set T of *simple Types* whose domains are subsets of D. Tuple types $[a1 : t1, ..., a_n : t_n]$ with attributes a_i of simple type t_i, $i = 1 \ldots n$ are defined in the standard way. The domain of a type t is denoted by $dom(t)$. For a tuple x, we denote by $x.a_i$ the value v_i associated with the attribute a_i.

We distinguish a simple type $Oid \in T$ of *object identifiers*, and two tuple types $Node$ and $Link$ with the following structure:

$$Node = [id : Oid, ..., a_i : t_i, ...]$$

$$Link = [from : Oid, to : Oid, ..., b_j : t_j, ...]$$

The attribute names in the two definitions are all distinct. We shall refer to tuples of the first type as *Node objects* and to tuples of the second type as $Link$

objects. In our model of the World Wide Web, documents will be mapped to $Node$ objects and the hypertext links between them to $Link$ objects. In this context, the object identifiers (Oid) will be the URL's.

Virtual Graphs

The set of all the documents in the Web, although finite, is undetermined: no one can produce a complete list of all the documents available at a certain moment. There are only two ways one can find documents in the Web: navigation starting from known documents and querying of index servers.

Given any URL, an agent can either fetch the associated document or give an error message if the document does not exist. This behavior can be modeled by a *computable* partial function mapping Oid's to $Node$ objects. Once a document is fetched, one can determine a finite set of outgoing hypertext links from that document. This can also be modeled by a computable partial function mapping Oid's to sets of $Link$ objects. In practice, navigation is done selectively, by following only certain links, based on their properties. In order to capture this, we introduce a finite set of unary *link predicates* $\mathcal{P}_{Link} = \{\alpha, \beta, \gamma, ...\}$.

The second way to discover documents is by querying index servers. To model the lists of URL's returned by index servers we introduce a (possibly infinite) set of unary *node predicates* $\mathcal{P}_{Node} = \{P, Q, R, ...\}$ where for each predicate P we are interested in the set $\{x | x \in dom(Oid),\ P(x) = true\}$. For example, a particular node predicate may be associated with a keyword, and it will be true of all documents that contain that keyword in their text.

Definition 3.1 *A* virtual graph *is a 4-tuple* $\Gamma = (\rho_{Node}, \rho_{Link}, \mathcal{P}_{Node}, \mathcal{P}_{Link})$ *where* $\rho_{Node} : dom(Oid) \to dom(Node)$ *and* $\rho_{Link} : dom(Oid) \to 2^{dom(Link)}$ *are computable partial functions,* $\mathcal{P}_{Node}$ *is a set of unary predicates on* $dom(Oid)$*, and* $\mathcal{P}_{Link}$ *is a finite set of unary predicates on* $dom(Link)$*.*

- *The set* $\{\rho_{Node}(oid) | oid \in dom(Oid)$ *and* ρ_{Node} *is defined on* $oid\}$ *is finite;*
- *if* $\rho_{Node}(oid) = v$ *then* $v.id = oid$*;*
- *for all* $oid \in dom(Oid)$*:* $\rho_{Node}(oid)$ *is defined* $\Leftrightarrow$ $\rho_{Link}(oid)$ *is defined;*
- *if* $\rho_{Link}(oid) = E$ *then* E *is finite and for all* $e \in E, e.from = oid$ *and* $\rho(e.to)$ *is defined (we say that* e *is an edge from* $v_1 = \rho_{Node}(e.from)$ *to* $v_2 = \rho_{Node}(e.to)$*);*
- *every predicate* $\alpha \in \mathcal{P}_{Link}$ *is a partial computable Boolean function on the set* $dom(Link)$*, and* α *is defined on all the links in* $\rho_{Link}(oid)$ *whenever* $\rho_{Link}(oid)$ *is defined.*
- *the function* $val : \mathcal{P}_{Node} \to 2^{dom(Oid)}$*, defined by* $val(P) = \{x | x \in dom(Oid),\ P(x) = true\}$ *is computable;*

Note that a virtual graph $\Gamma = (\rho_{Node}, \rho_{Link}, \mathcal{P}_{Node}, \mathcal{P}_{Link})$ induces an underlying directed graph $G(\Gamma) = (V, E)$ where $V = \rho_{Node}(dom(Oid))$ and $E = \bigcup_{oid \in dom(Oid)} \rho_{Link}(oid)$. However, a calculus cannot manipulate this graph directly because of the computability issues presented above.

3.2 The Calculus

Now we proceed to define our calculus for querying virtual graphs. We introduce path regular expressions to specify connectivity-based queries. We then present the notions of *range expressions* and *ground variables* to restrict queries so that their evaluation does not require enumerating every node of the virtual graph, and finally we define calculus queries.

Path regular expressions

Consider a virtual graph Γ and denote its underlying graph $G(\Gamma) = (V, E)$. A *path* in Γ is defined in the same way as in a directed graph: if $e_1, e_2, ..., e_k \in E$, we call $p = (e_1, e_2, ..., e_k)$ a path if and only if for every index $1 \leq i < k$, $e_{i+1}.from = e_i.to$. A path p is called *simple* if there are no different edges $e_i \neq e_j$ in p with the same starting or ending points.

In order to express queries based on connectivity, we need a way to define graph patterns. Recall $\mathcal{P}_{Link}$ is the set of link properties in a virtual graph. Let $\alpha \in \mathcal{P}_{Link}$ be some link property, and let e be a link object. If $\alpha(e) = true$ then we say that e has the property α. We define the *set of properties* of a link e by $\Lambda(e) = \{\alpha \in \mathcal{P}_{Link} | \alpha(e) = true\}$. We sometimes choose to view $\Lambda(e)$ as a formal language on the alphabet $\mathcal{P}_{Link}$. For each property α that is true of e, $\Lambda(e)$ contains the single-character string α. To study the properties of a path, we extend the definition of the set of properties of a link to paths as follows: if $p = (e_1, ..., e_k)$ is a path then we define

$$\Lambda(p) = \Lambda(e_1)...\Lambda(e_k)$$

where $LL' = \{xx' | x \in L, x' \in L'\}$ is the concatenation of the languages L and L'.

Example 3.1 *Suppose we want to require that a property* α *hold on all the links of a path* $p = (e_1, ..., e_k)$*. This can be expressed easily in terms of* Λ *by requiring that* $\alpha^k \in \Lambda(p)$*.*

In order to specify constraints like in the example above we introduce *path regular expressions*, which are nothing more than regular expressions over the alphabet $\mathcal{P}_{Link}$. With each regular expression R over the alphabet $\mathcal{P}_{Link}$ we associate a language $L(R) \subseteq \mathcal{P}^*_{Link}$ in the usual way.

Definition 3.2 *We say that the path p matches the path regular expression R if and only if:*

$$\Lambda(p) \cap L(R) \neq \emptyset$$

In other words, the path p matches the path regular expression R if and only if there is a word w in the set of properties $\Lambda(p)$ that matches the regular expression R.

Range expressions

The algebra for a traditional relational database is based on operators like select (σ), project (π) and Cartesian product ($\times$). Because all the contents of the database is assumed to be available to the query engine, all these operations can be executed, in the worst case by enumerating all the tuples. In the case of the World Wide Web, the result of a select operation cannot be computed in this way, simply because one cannot enumerate all the documents. Instead, navigation and querying of index servers must be used. We want our calculus to express only queries that can be evaluated without having to enumerate the whole Web. To enforce this restriction, we introduce *range conditions*, that will serve as restrictions for variables in the queries.

Definition 3.3 *Let $\Gamma = (\rho_{Node}, \rho_{Link}, \mathcal{P}_{Node}, \mathcal{P}_{Link})$ be a virtual graph. Let $G(\Gamma) = (V, E)$ be its underlying graph. A* range atom *is an expression of one of the following forms:*

- *$Path(u, R, x)$ where u, x are Oids or variable names, and R is a path regular expression;*
- *$P(x)$ where $P \in \mathcal{P}_{Node}$ and x is an Oid or a variable name;*
- *$From(u, x)$ where u, x are Oids or variable names;*

A range expression *is an expression of the form: $\mathcal{E} = \{x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n | A_1, A_2, \ldots, A_m\}$ where $A_1, A_2, ..., A_m$ are range atoms, $x_1, x_2, ..., x_n$ are all the variables occurring in them and $T_i \in \{Node, Link\}$ specifies the type of the variable x_i, for $1 \leq i \leq n$.*

Consider a valuation $\nu : \{x, y, z, \ldots\} \rightarrow V \cup E$ that maps each variable into a node or an edge of the underlying graph. We extend ν to $dom(Oid)$ by $\nu(oid) = \rho_{Node}(oid)$, that is, ν maps each Oid appearing in an atom to the corresponding node. The following definition assigns semantics to range atoms.

Definition 3.4 *Let $\Gamma = (\rho_{Node}, \rho_{Link}, \mathcal{P}_{Node}, \mathcal{P}_{Link})$ be a virtual graph. Let A be a range atom. We say that A is validated by the valuation ν if:*

- *for $A = Path(u, R, x)$, there exists a simple path from $\nu(u)$ to $\nu(x)$ matching the path regular expression R;*
- *for $A = P(x)$, $P(\nu(x).id) = true$.*
- *for $A = From(u, x)$, $\nu(x).from = \nu(u).id$*

Now we can give semantics to range expressions.

Definition 3.5 *Consider a range expression $\mathcal{E} = \{x_1, ..., x_n | A_1, ..., A_m\}$. Then the set of tuples*

$$\Psi(\mathcal{E}) = \{[\nu(x_1), ..., \nu(x_n)] | \nu \text{ is a valuation s.t. } A_1, ..., A_m \text{ are all validated by } \nu\}$$

is called the range of $\mathcal{E}$.

Example 3.2 *The set of all nodes satisfying a certain node predicate P together with all their outgoing links may be specified by the following range expression: $\{x : Node, y : Link | P(x), From(x, y)\}$*

Although Definition 3.5 gives a well-defined semantics for all range expressions, problems may arise when examining the evaluation of $\Psi(\mathcal{E})$ for certain expressions $\mathcal{E}$. For example, expressions like $\{x : Node, y : Node | Path(x, \alpha, y)\}$ (find all pairs of nodes connected by a link of type α) or $\{x : Node, y : Link | From(x, y)\}$ (find all nodes and all links outgoing from them) cannot be algorithmically evaluated on an arbitrary virtual graph, since their evaluation would involve the enumeration of all nodes. We impose syntactic restrictions to disallow such range expressions, in a manner similar to the definition of safe expressions in Datalog. We omit the details for lack of space and assume from now on that all range expressions are restricted in this way.

Queries

After restricting the domain from a large, non-computable set of nodes and links to a computable set, we may use the traditional relational selection and projection to impose further conditions on the result set of a query. This allows us to introduce the general

format of queries in our calculus. We assume a given set $\mathcal{P}_s$ of binary predicates over simple types. Examples of predicates include equality (for any type), various inequalities (for numeric types), and substring containment (for alphanumeric types).

Definition 3.6 *A* virtual graph query *is an expression of the form:* $\pi_L\ \sigma_\phi\ \mathcal{E}$ *where:*

- $\mathcal{E} = \{x_1 : T_1, ..., x_n : T_n | A_1, ..., A_m\}$ *is a range expression;*
- L *is a comma separated list of expressions of the form* $x_i.a_j$ *where* a_j *is some attribute of the type* T_i*;*
- ϕ *is a Boolean expression constructed from binary predicates from* $\mathcal{P}_s$ *applied to expressions* $x_i.a_j$ *and constants using the standard operators* $\wedge$, $\vee$, *and* $\neg$*;*

The semantics of the select (σ) and project (π) operators is the standard one.

3.3 WebSQL Semantics

We are now ready to define the semantics of our WebSQL language in terms of the formal calculus introduced above. To do this, we need to model the Web as a virtual graph $W = (\rho_{Node}, \rho_{Link}, \mathcal{P}_{Node}, \mathcal{P}_{Link})$: $dom(Oid)$ is the infinite set of all syntactically correct URL's, and for every element $url \in dom(Oid)$, $\rho_{Node}(url)$ is either the document referred to by url, or is undefined, if the URL does not refer to an existing document. Note that $\rho_{Node}(url)$ is computable (its value can be computed by sending a request to the Web server specified in the URL). Moreover, $\rho_{Link}(url)$ is the set of all anchors in the document referred to by url, or is undefined, if the URL does not refer to an existing document. One can extract all the links appearing in an HTML document by scanning the contents in search of `<A>` and `</A>` tags. This means that the partial function $\rho_{Link}(url)$ is computable. In order to model content queries we consider the following set of $Node$ predicates: $\mathcal{P}_{Node} = \{C_w | w \in \Sigma^*\}$ where, for each $w \in \Sigma^*$, $C_w(n) = true$ if the document n contains the string w. Finally, we consider the following set of $Link$ predicates: $\mathcal{P}_{Link} = \{\rightarrow, \Rightarrow, \mapsto\}$ in accordance with the definition of path regular expressions in WebSQL.

The semantics of a WebSQL query is defined as usual in terms of selections and projections. Thus a query of the form:

SELECT L
FROM $\mathcal{E}$
WHERE ϕ

translates to the following calculus query: $\pi_L \sigma_\phi \mathcal{E}'$ where $\mathcal{E}' = \{x_1, ..., x_n | A_1, ..., A_n\}$ is obtained from $\mathcal{E} = C_1, ..., C_n$ by using the following transformation rules:

- if C_i = Document x SUCH THAT $u\ R\ x$ then $A_i = Path(u, R, x)$
- if C_i = Document x SUCH THAT x MENTIONS w then $A_i = C_w(x)$
- if C_i = Anchor x SUCH THAT x.base = u.url then $A_i = From(u, x)$

We only allow as legal WebSQL queries those that translate into calculus queries that satisfy the syntactic restrictions of Section 3.2.

4 Query Locality

Cost is an important aspect of query evaluation. The conventional approach in database theory is to estimate query evaluation time as a function of the size of the database. In the web context, it is not realistic to try to evaluate queries whose complexity would be considered feasible in the usual theory, such as polynomial or even linear time.

For a query to be practical, it should not attempt to access too much of the network. Query analysis thus involves, in this context, two tasks: first, estimate what part of the network may be accessed by the query, and then the cost of the query can be analyzed in traditional ways as a function of the size of this subnetwork. In this section, we concentrate on the first task. Note that this is analogous, in a conventional database context, to analyzing queries at the physical level to estimate the number of disk blocks that they may need to access.

For this first task, we need some way to measure the "locality" of a query, that is, how far from the originating site do we have to search in order to answer it. Having a bound on the size of the sub-network needed to evaluate a query means that the rest of the network can be ignored. In fact, a query that is sensitive only to a bounded sub-network should give the same result if evaluated in one network or in a different network containing this sub-network. This motivates our formal definition of query locality.

An important issue is the cost of accessing such a sub-network. In the current web architecture, access

to remote documents is often done by fetching each document and analyzing it locally. The cost of an access is thus affected by document properties (e.g. size) and the by the cost of communication between the site where the query is being evaluated and the site where the document is stored. Recall that we model the web as a virtual graph. To model access costs, we extend the definition of virtual graphs, adding a function $\rho_c : dom(Oid) \times dom(Oid) \rightarrow \mathcal{N}$, where $\rho_c(i,j)$ is the the cost of accessing node j from node i.

4.1 Locality Cost

We now define the formal notion of locality. For that we first explain what it means for two networks to contain the same sub-network. We assume below that all the virtual graphs being discussed have the same sets of node and link predicate names.

Definition 4.1
Let $\Gamma = (\rho_{Node}, \rho_{Link}, \rho_c, \mathcal{P}_{Node}, \mathcal{P}_{Link})$, $\Gamma' = (\rho'_{Node}, \rho'_{Link}, \rho'_c, \mathcal{P}'_{Node}, \mathcal{P}'_{Link})$ *be two (extended) virtual graphs. Let* $W \subseteq dom(Oid)$. *We say that* Γ **agrees** *with* Γ' *about* W *if for all* $w, w' \in W$,

1. $\rho_{Node}(w) = \rho'_{Node}(w)$, $\rho_{Link}(w) = \rho'_{Link}(w)$, $\rho_c(w,w') = \rho'_c(w,w')$,
2. *for all the node predicates* P_{Node}, *if* $n = \rho_{Node}(w)$ *is defined, then* $P_{Node}(n)$ *holds in* Γ *iff it holds in* Γ',
3. *for all the link predicates* P_{Link} *and for all links* $l \in \rho_{Link}(w)$, $P_{Link}(l)$ *holds in* Γ *iff it holds in* Γ'.

Informally this means that the two graphs contain the sub-network induced by W, the nodes of W have the same properties in both graphs, and in both graphs this sub-network is linked to the rest of the world in the same way.

Definition 4.2 *Let* Q *be a query, let* $\mathcal{G}$ *be a class of virtual graphs, let* $\Gamma \in \mathcal{G}$ *be a graph, and let* $W \subseteq dom(Oid)$. *We say that query* Q **depends** *on* W, *(for* Γ *and* $\mathcal{G}$*), if for every graph* $\Gamma' \in \mathcal{G}$ *that agrees with* Γ *about* W, $Q(\Gamma') = Q(\Gamma)$, *and there is no subset of* W *satisfying this.*

W is a minimal set of documents needed for computing Q. Note that W may not be unique. This is reasonable since the same information may be stored in several places on the network.

Definition 4.3 *For a virtual graph* Γ, *a set of oid's* W *for nodes in* Γ, *and some oid* i *for a node in* Γ, *the* **cost of** W **from** i *is the sum of* $\rho_c(i,w)$ *over all documents* w *in* W *such that* $\rho_{Node}(w)$ *is defined.*

Definition 4.4 *The* **locality cost** *of a query* Q, *when evaluated at node* i *of virtual graph* Γ, *is the maximum, over all sets* W *on which* Q *depends in* Γ, *of the cost of* W *from* i.

We are interested in bounding this cost with some function of the cost of accessing all the nodes of the network, that is, the sum of $\rho_c(i,j)$ over all j such that $\rho_{Node}(j)$ is defined. If this total cost is n, note that the locality cost of a query is at most linear in n.

Obviously, queries with $O(n)$ locality are impractical - the whole network needs to be accessed in order to answer them. We will be interested in constant bounds, where the constants may depend on network parameters such as number of documents in a site, maximal number of URL's in a single document, certain communication costs, etc.

In general, access to documents on the local server is considered cheap, while documents in remote servers need to be fetched and are thus relatively expensive. To simplify the discussion and highlight the points of interest we assume below a rather simple cost function. We assume that local accesses are free, while the access cost to remote documents is bounded by some given constant. (Similar results can be obtained for a more complex cost function). In the next few examples, suppose the queries are evaluated at the node "http://www.cs.toronto.edu". Consider first the query

Q_0: SELECT x
FROM Document x SUCH THAT
"http://www.cs.toronto.edu" $\rightarrow$ x

where "http://www.cs.toronto.edu" is at the local server. The query accesses local documents pointed to by the home page of the Toronto CS department, and no remote ones. Thus the locality cost is $O(1)$. On the other hand, the query

Q_1: SELECT x
FROM Document x SUCH THAT
"http://www.cs.toronto.edu" $(\rightarrow \mid \Rightarrow)$ x

accesses both local and remote documents. The number of remote documents being accessed depends on the number of anchors in the home page that contain remote URL's. In the worst case, all the URL's in the page are remote. If k is a bound on the number of URL's in a single document, then the locality cost of this query is $O(k)$.

As another example, consider the queries

Q_2: SELECT x
FROM Document x SUCH THAT
"http://www.cs.toronto.edu" $\rightarrow^*$ x

Q_3: SELECT x
FROM Document x SUCH THAT
"http://www.cs.toronto.edu" $\Rightarrow . \rightarrow^*$ x

Q_4: SELECT x
FROM Document x SUCH THAT
"http://www.cs.toronto.edu" $(\rightarrow \mid \Rightarrow)^*$ x

Query Q_2 accesses local documents reachable from the CS department home page, and is thus of locality $O(1)$. Query Q_3 accesses all documents reachable by one global link followed by an unbounded number of local links. If k is a bound on the number of URL's in a single document, and s is a bound on the number of documents in a single server, then the locality cost of the query is $O(ks)$. This is because in the worst case all the URL's in the CS department home page reference documents in distinct servers, and all the documents on those servers are reachable from the referenced documents. The last query accesses all reachable documents. In the worst case it may attempt to access the whole network, thus its cost is $O(n)$.

The locality analysis of various features of a query language can identify potentially expensive components of a query. The user can then be advised to rephrase those specific parts, or to give some cost bounds for them in terms of time, number of sites visited, CPU cycles consumed, etc., or, if enough information is available, dollars. The query evaluation would monitor resource usage and interrupt the query when the bound is reached.

If the query language is too complex, locality analysis may be very complex or even impossible. Nevertheless, in many cases it can be done effectively and efficiently. This in particular is the case for the WebSQL query language. In the next subsection, we show that locality of WebSQL queries can be determined in time polynomial in the size of the query.

4.2 Locality of WebSQL queries

We start by considering simple queries where the FROM clause consists of a single path atom starting from the local server, as in the examples above. We then analyze general queries.

Analyzing single path expressions

The analysis is based on examining the types of links (internal, local, or global) that can be traversed by paths described by the path expression. Particular attention is paid to "starred" sub-expressions since they can describe paths of arbitrary length. Assume that the query is evaluated at some node i, and let n denote the cost of accessing the whole network graph from i. Let k be some bound on the number of URL's appearing in a single document, [3] and s some bound on the number of documents in a single server.

1. Expressions with no global links can access only local documents and thus have locality O(1).

2. Expressions containing global links that appear in "starred" sub-expression, can potentially access all the documents in the network. Thus the locality is $O(n)$;

3. Expressions with global links, but where none of the "starred" sub-expressions contain a global link symbol, can access remote documents, but the number of those is limited. All the paths defined by such expressions are of the form $\rightarrow^{l_1} . \Rightarrow . \rightarrow^{l_2} . \Rightarrow \ldots \rightarrow^{l_m}$. Let $l = 1 + max(l_1, \ldots, l_m)$. The number of documents accessed by such path is bounded by

$$\min(n, (m(l+1)(k\ min(s, k^l))^m))$$

where k and s are the bounds above. This is because the number of different documents reachable by a path $\rightarrow^{l_i}$ is at most $\min(s, k^{l_i})$, and in the worst case all the $k \min(s, k^{l_i})$ URL's in those documents point to files on distinct servers. The number of documents reached at the end of the path is thus at most $(k \min(s, k^l))^m$. In order to reach those documents, all files along the path need to be fetched. To get a bound on this, we multiply the number by the length of the path. Bounds for m and l can be computed in polynomial time by analyzing the path expression. (Details omitted). [4]

The number m is bounded by the number of global links appearing in the given path expression. l can be computed by analyzing the regular expression. (details omitted). All this can be done in time polynomial in the size of the expression. [5]

Observe that if any of the starred sub-expressions contains a local link, then $l = s$. This is because,

[3] If no such bound exists, every path expression containing a global link is of locality cost $O(n)$ (because in the worst case a single document may point to all the nodes in the network).

[4] The expression above is a simple upper bound on the locality cost. A tight bound can also be computed in polynomial time. The exact expression is complex and does not add much insight to the analysis, so we omit it.

[5] The above expression is an upper bound on the locality cost. The exact cost can also be computed in polynomial time. We chose to present the above bound since the exact expression is very complex and does not add much insight to the analysis.

in the worst case, such sub-expressions will attempt to access all the documents in the server. In this case the bound becomes $\min(n, (m(s+1)(k\ s)^m))$. Since servers may contain many documents, the locality cost may be very high. This indicates that such queries are potentially expensive and that the user should be advised to provide the query evaluator certain bounds on resources.

Analyzing queries

To analyze a WebSQL query it is not sufficient to look at individual path atoms. The whole FROM clause needs to be analyzed. For example, consider a query

Q_5: SELECT z, w
FROM Document x
SUCH THAT x MENTIONS "PDIS96"
Anchor y SUCH THAT base $=$ x
Document z SUCH THAT y $\rightarrow^*$ z
Document w SUCH THAT x $\rightarrow$ w

The path expressions in the query involve only local links. But since the links returned by the index server may point to remote documents, the paths traversed by the sub-condition

FROM Document x
SUCH THAT x MENTIONS "PDIS96"
Document w SUCH THAT x $\rightarrow$ w

are actually of the form $(\Rightarrow\ .\ \rightarrow)$. Similarly, since the links traversed in "Anchor y SUCH THAT base $= x$" can be global, the paths traversed by the sub-condition

FROM Document x
SUCH THAT x MENTIONS "PDIS96"
Anchor y SUCH THAT base $=$ x
Document z SUCH THAT y $\rightarrow^*$ z

are of the form $(\Rightarrow\ .\ \Rightarrow\ .\ \rightarrow^*)$. Thus the regular expression describing the path accessed by the query is $((\Rightarrow\ .\ \rightarrow)\ \mid\ (\Rightarrow\ .\ \Rightarrow\ .\ \rightarrow^*))$. Assume that querying an index server can be done with locality c, and that the number of URL's returned by an index server on a single query is bounded by some number m. The locality cost of evaluating this query is therefore bounded by c plus m times the locality bound of $(\Rightarrow\ .\ \rightarrow)\ \mid\ (\Rightarrow\ .\ \Rightarrow\ .\ \rightarrow^*)$.

Interestingly, every FROM clause of a WebSQL query can be transformed into a regular expression describing the paths accessed in its evaluation. This, together with the locality cost of querying the index severs used in the query, and the bounds on the size of their answers, lets us determine the locality of queries in time polynomial on the size of the query. This, together with the locality cost of querying the index severs used in the query, and the bounds on the size of their answers, lets us determine the locality of the FROM clause in time polynomial on the size of the query. Bounding the locality of the FROM clause provides an upper bound on the locality of the whole query; a slightly better bound can be obtained by analyzing the SELECT and WHERE clauses.

We omit the details for lack of space.

5 Implementation

This section presents our prototype implementation of a WebSQL compiler, query engine, and user interface.

Both the WebSQL compiler and query engine are implemented entirely in Java [SM], the language introduced by Sun Microsystems with the specific purpose of adding executable content to Web documents.

A prototype user interface for the WebSQL system is accessible from the WebSQL home page (`http://www.cs.toronto.edu/~georgem/WebSQL.html`) as a CGI script.

The WebSQL system architecture is depicted in Figure 2.

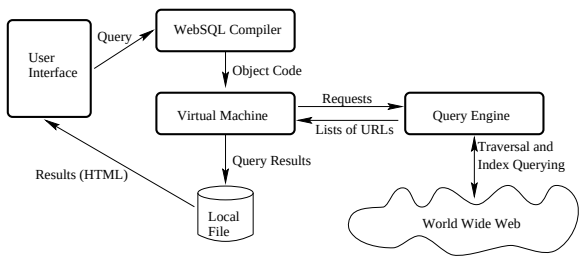

Figure 2: The Architecture of the WebSQL System

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 Node and Link objects. The evaluation of range atoms is done via specially designed operation codes whose results are vectors of Node or Link objects.

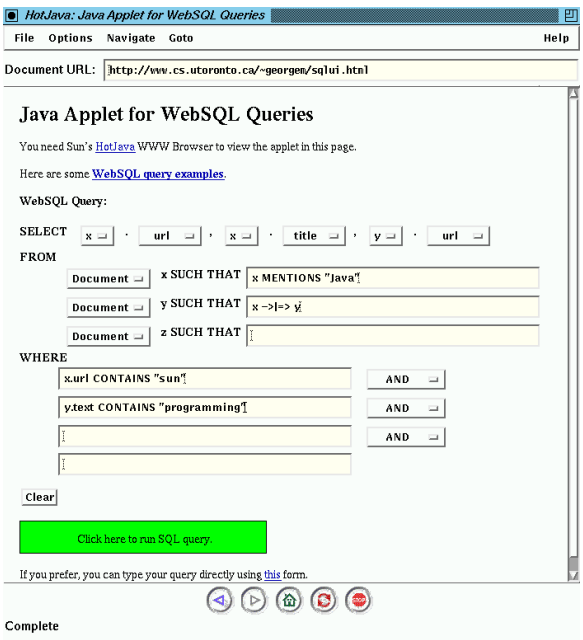

Figure 3: The WebSQL User Interface

The Query Engine

Whenever the interpreter encounters an operation code corresponding to a range atom, the query engine is invoked to perform the actual evaluation. There are three types of atoms, according to Definition 3.3. Let us examine each of them in sequence:

- if $A = Path(u, R, x)$ the engine generates all simple paths starting at u that match R, thus determining the list of all qualifying values of x. Mendelzon and Wood give in [MW95] an algorithm for finding all the simple paths matching a regular expression R in a labeled graph. We adapted this algorithm for the virtual graph context. (For full details see [Mih96]).

- if $A = C_w$ the engine queries a customizable set of known index servers (currently Yahoo and Lycos) with the string w and builds a sorted list of URL's by merging the individual answer sets;

- if $A = From(u, x)$ the engine determines first if u is an HTML document and if it is, it parses it and builds a list of $Link$ objects out of the set of all the anchor tags; if u is not an HTML document, the engine returns the empty list.

The User Interface

The appearance of the user interface is shown in Figure 3, a screen-shot of the HotJava browser.

Figure 4: WebSQL Query Results

The input form can be used as a template for the most common WebSQL queries making it easier for the user to submit a query. If the query is more complicated it can always be typed into an alternative text field. After the query is entered it may be submitted by pressing the appropriate button. At that point, the interface collects all the data from the input fields and assembles the WebSQL query. Then the query is sent to the Parser, which checks the syntax and produces the object code. The object code is then executed by the Interpreter and finally a query result set is computed. This set is formatted as an HTML document and displayed by the browser. All URL fields that appear in the result are formatted as anchors so that the user may jump easily to the associated documents. Figure 4 contains a screen shot of a typical result document.

Performance

The execution time of a WebSQL query is influenced by various factors related to the network accesses performed in the process of building the result set. Among these factors we can mention the number and size of the transferred documents, the available network bandwidth, and the performance and load of

the accessed Web servers. Because our query processing system does not maintain any persistent local information between queries it has to access the Web for every new query. Therefore, care must be taken when formulating queries by estimating the number of documents that have to be retrieved. We executed a number of queries by running the Java implementation described in the previous section from within an instance of the HotJava browser running under Solaris 2.3 on a SUN Sparcserver 20/612 with 2 CPUs and 256 Mbytes of RAM.

The execution times for the queries we tested vary between under ten seconds, for simple content queries, to several minutes for structural queries involving the exploration of Web subgraphs with about 500 nodes.

6 Discussion

We have presented the WebSQL language for querying the World Wide Web, given its formal semantics in terms of a new virtual graph model, proposed a new notion of query cost appropriate for Web queries, and applied it to the analysis of WebSQL queries. Finally, we described the current prototype implementation of the language.

Looking at Figure 3, one is skeptical that this complex interface will replace simple keyword-based search engines. However, this is not its purpose. 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. Some examples:

Selective indexing: As the Web grows larger, we will often want to build indexes on a selected portion of the network. WebSQL can be used to specify this portion declaratively.

View definition: This is a generalization of the previous point, as an index is a special kind of view. Views and virtual documents are likely to be an important facility, as discussed by Konopnicki and Shmueli [KS95], and a declarative language is needed to specify them.

Link maintenance: Keeping links current and checking whether documents that they point to have changed is a common task that can be automated with the help of a declarative query language.

Several directions for extending this work present themselves. First, instead of being limited to a fixed repertoire of link types (internal, local, and global), we would like to extend the language with the possibility of defining arbitrary link types in terms of their properties, and use the new types in regular expressions. For example, we might be interested in links pointing to nodes in Canada such that their labels do not contain the strings "Back" or "Home."

Second, we would like to make use of internal document structure when it is known, along the lines of [CACS94] and [QRS+95].

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.

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario.

References

[ACM93] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In *Proceedings of the 19th VLDB Conference*, 1993.

[AV97] S. Abiteboul and V. Vianu. Queries and computation on the Web. In *Proc. ICDT '97*, 1997.

[BK90] C. Beeri and Y. Kornatzky. A logical query language for hypertext systems. In *Proc. of the European Conference on Hypertext*, pages 67–80. Cambridge University Press, 1990.

[BLCL+94] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The World-Wide Web. *Comm. of the ACM*, 37(8):76–82, 1994.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. ACM SIGMOD'94*, pages 313–324, 1994.

[CM89] M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in Graphlog. In *Hypertext'89*, pages 269–292, 1989.

[Dre96] Daniel Dreilinger. Savvysearch home page. 1996. `http://guaraldi.cs.colostate.edu:2000/`.

[GZC89] Ralf Hartmut Güting, Roberto Zicari, and David M. Choy. An algebra for structured office documents. *ACM TOIS*, 7(2):123–157, 1989.

[Hal88] Frank G. Halasz. Reflections on Notecards: Seven issues for the next generation of hypermedia systems. *Comm. of the ACM*, 31(7):836–852, 1988.

[HGN+95] Masum Z. Hasan, Gene Golovchinsky, Emanuel G. Noik, Nipon Charoenkitkarn, Mark Chignell, Alberto O. Mendelzon, and David Modjeska. Visual Web surfing with Hy+. In *Proceedings CASCON '95*, Toronto, November 1995. IBM Canada. `ftp://db.toronto.edu-/pub/papers/cascon95-multisurf.ps.Z`.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the World Wide Web. In *Proc. of VLDB'95*, pages 54–65, 1995.

[LSS96] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. A declarative language for querying and restructuring the Web. In *Proc. of 6th. International Workshop on Research Issues in Data Engineering, RIDE '96*, New Orleans, February 1996. In press.

[Mih96] G. A. Mihaila. WebSQL - an SQL-like query language for the world wide web. Master's thesis, University of Toronto, 1996.

[MW93] T. Minohara and R. Watanabe. Queries on structure in hypertext. In *Foundations of Data Organization and Algorithms, FODO '93*, pages 394–411. Springer, 1993.

[MW95] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comp.*, 24(6), 1995.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases, Proceedings of the DOOD '95 Conference*, pages 319–344, Singapore, December 1995. Springer.

[SE95] Erik Selberg and Oren Etzioni. Multi-service search and comparison using the MetaCrawler. In *Proceedings of the Fourth Int'l WWW Conference*, Boston, December 1995. `http://www.w3.org/pub/-Conferences/WWW4/Papers/169`.

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