JODL Manuscript-Nr.
(will be inserted by hand later)

Querying the World Wide Web

Alberto O. Mendelzon[1], George A. Mihaila[1], and Tova Milo[2]

[1] Department of Computer Science and CSRI, University of Toronto
[2] Computer Science Department, Tel Aviv University

December 9, 1996

Abstract. [1] 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 with respect to 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. For example, suppose 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 pages directly or indirectly reachable from this page, and stored on the same server. 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.

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-

Correspondence to: Alberto O. Mendelzon
[1] A preliminary version of this paper was presented at the 1996 Symposium on Parallel and Distributed Information Systems

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. Query cost analysis is a prerequisite for query optimization; we give an algorithm for characterizing WebSQL queries with respect to query locality that we believe will be useful in the query optimization process, although we do not develop query optimization techniques in this paper. 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, NBY95]. 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 is the W3QL work by Shmueli and Konopnicki[KS95]. Our motivation is very similar to theirs, 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 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. The paper does not describe an implementation or formal semantics.

The work we describe here is not specifically addressed to digital libraries. However, a declarative language that integrates web navigation with content-based search has obvious applications to digital library construction and maintenance: for example, it can be used for building and maintaining specialized indices and online bibliographies, for defining virtual documents stored in a digital library, and for building applications that integrate access to specific digital libraries with access to related web pages.

2 The WebSQL Language

In this section we introduce our SQL-like language for the World Wide Web. We begin by proposing a relational model of the WWW. Then, we give a few examples for 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, every 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 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.[2] All these attributes are character strings.

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

Example 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 1. *A hypertext link is said to be:*

- interior *if the destination is within the source document;*[3]
- 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 Sect. 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 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";

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

Example 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;

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 mean different things. Conditions in the FROM clause will be evaluated 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 evaluating the conditions in the WHERE clause. This distinction is reflected both in the formal semantics and in the implementation. We could remove the distinction and let a query optimizer decide which conditions are evaluated by index servers and which are tested locally. However, we prefer to keep the explicit distinction because index servers are not perfect or complete, and the programmer may want to control which conditions are evaluated by them and which are not.

The BNF specification of the language syntax is given in the Fig. 1. The syntax follows the standard SQL `SELECT` statement. All queries 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

[2] 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.

[3] In HTML, 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.

```
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 := PathTerm {| PathTerm}
PathTerm := PathFactor {PathFactor}
PathFactor := PathPrimary[*]
PathPrimary := Link
Link := = | ↦ | → | ⇒
```

Fig. 1. WebSQL Syntax

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 and the Node and Link tuples will model the WebSQL Document and Anchor virtual tables introduced informally in Section 2.

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 2. *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;*

Example 5. A virtual graph:
$\Gamma = (\rho_{Node}, \rho_{Link}, \mathcal{P}_{Node}, \mathcal{P}_{Link})$
where:
$Node = [url : Oid, title : string, text : string]$,
$Link = [from : Oid, to : Oid, label : string]$,
$dom(Oid) = \{url_1, url_2, url_3\}$,

url	$\rho_{Node}(url)$
url_1	$[url_1, \text{“}Toronto\text{''}, \text{“}Visiting\ Toronto?...\text{''}]$
url_2	$[url_2, \text{“}Ottawa\text{''}, \text{“}The\ national\ capital...\text{''}]$
url_3	$[url_3, \text{“}New\ York\text{''}, \text{“}Tourist\ information...\text{''}]$

url	$\rho_{Link}(url)$
url_1	$\{[url_1, url_2, \text{“}Go\ to\ Ottawa\text{''}],$ $[url_1, url_3, \text{“}Go\ to\ New\ York\text{''}]\}$
url_2	$\{[url_2, url_1, \text{“}Back\ to\ Toronto\text{''}]\}$
url_3	$\{[url_3, url_1, \text{“}Back\ to\ Toronto\text{''}]\}$

and $\mathcal{P}_{Node} = \{Contains_{tourist}, Contains_{capital}\}$, $\mathcal{P}_{Link} = \{Contains_{go}, Contains_{back}\}$.

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. This captures our intuition about the World Wide Web hypertext graph, whose nodes and links can only be discovered through navigation.

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 6. 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. *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 4. *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 $\nu : \{x, y, z, \ldots\} \to V \cup E$ 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 5. *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 6. *Consider a range expression $\mathcal{E} = \{x_1, ..., x_n | A_1, ..., A_m\}$. Then the set of tuples $\Psi(\mathcal{E}) = \{[\nu(x_1), ..., \nu(x_n)] | \nu$ is an valuation s.t. $A_1, ..., A_m$ are all validated by $\nu\}$ is called the range of $\mathcal{E}$.*

Example 7. 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: $\{x : Node, y : Link | P(x), From(x, y)\}$

Ground Variables and Ground Expressions

Although Definition 6 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.

Consider a virtual graph $\Gamma = (\rho_{Node}, \rho_{Link}, \mathcal{P}_{Node}, \mathcal{P}_{Link})$. Let us examine the evaluation of atoms for all the three cases in Definition 4:

- if $A = Path(u, R, x)$, determining all pairs of nodes (u, x) separated by simple paths matching the path regular expression R is possible only if u is a constant. Indeed, if u is known, we can traverse the graph starting from u to generate all simple paths matching R, thus determining the values of x. If u is a variable and x is a constant, since Web links can only be traversed in one direction, there is no way to determine the values of u for arbitrary R without enumerating all the nodes in the graph. All the more so when both u and x are variables. A similar argument shows that u must be a constant in $A = From(u, x)$.
- the atoms of the form $P(x)$ where $P \in \mathcal{P}_{Node}$ pose no problem since the set $\{x \in V | P(x) = true\}$ is computable (by Definition 2).

The above considerations lead to the following definition.

Definition 7. *A variable x occurring in a range atom A is said to be* independent *in A if it is the only variable in the atom. If two variables u and x appear in an atom A in this order (i.e. $A = Path(u, R, x)$ or $A = From(u, x)$), then we say that x* depends *on u in A.*

The idea is that independent variables can be determined directly, whereas dependent variables can be determined only after the variables they depend upon have been assigned values. The following definition gives a syntactic restriction over range expressions that will ensure computability.

Definition 8. *Let $\mathcal{E} = \{x_1 : T_1, ..., x_n : T_n | A_1, ..., A_m\}$ be a range expression. A variable $x \in \{x_1, ..., x_n\}$ is said to be* ground *in $\mathcal{E}$ if there exists an atom A_i such that either x is independent in A_i, or x depends in A_i on a variable u that is ground in $\mathcal{E}$. The expression $\mathcal{E}$ is said to be* ground *if all the variables in $\mathcal{E}$ are ground.*

Theorem 1. *Consider a virtual graph $\Gamma = (\rho_{Node}, \rho_{Link}, \mathcal{P}_{Node}, \mathcal{P}_{Link})$ and a range expression $\mathcal{E} = \{x_1 : T_1, ..., x_n : T_n | A_1, ..., A_m\}$. If $\mathcal{E}$ is ground then $\Psi(\mathcal{E})$ is computable.*

Proof. Consider the following *dependency graph* $G_D = (V_D, E_D)$ where $V_D = \{x_1, ..., x_n\}$ and $E_D \subseteq V_D^2$ being the dependency relation (i.e. there is an edge from x_i to x_j if and only if x_j depends on x_i in some atom A_k). We distinguish two cases depending on the presence of cycles in G_D. We will consider first the acyclic case and then we will reduce the cyclic case to the acyclic one by transforming the expression E into an equivalent one.

Case I (G_D acyclic): By doing a topological sort we can construct a total order among the variables which is compatible with the dependency relation, that is, a permutation $\sigma \in S(n)$ s.t. $(x_{\sigma(i)}, x_{\sigma(j)}) \in E$ implies $i < j$.

To simplify the notation, we can rename the variables according to the permutation σ so that each variable x_i depends only on variables preceding it in the list $x_1, ..., x_n$.

For each variable x_i we define the set $I_{x_i} \subseteq \{1, ..., m\}$ as the set of the indices of the atoms where x_i occurs either as an independent or dependent variable. Since every variable is ground, all sets I_{x_i} are non-empty.

Since x_1 does not depend on any other variable and is ground, all its occurrences in atoms are independent occurrences. This means that we can compute the set of values of x_1 in $\Psi(\mathcal{E})$:

$$\pi_{x_1}\Psi(\mathcal{E}) = \bigcap_{i \in I_{x_1}} \{x | A_i(x) = true\}$$

Furthermore, let us consider one element c_1 in this set (if this set is empty, then $\Psi(\mathcal{E})$ is also empty and its computation is complete). We replace all occurrences of x_1 in atoms with the constant c_1 (denote the transformed atoms by $A_i[x_1/c_1], 1 \leq i \leq m$). The occurrences of x_2 in the atoms A_i with $i \in I_{x_2}$ are either independent or dependent on x_1. Therefore, after the substitution of x_1 by c_1 all the occurrences of x_2 in $A_i[x_1/c_1]$ with $i \in I_{x_2}$ became independent. This means that we can compute the set of all values of x_2 in the tuples where x_1 is c_1:

$$\pi_{x_2}\sigma_{x_1=c_1}\Psi(\mathcal{E}) = \bigcap_{i \in I_{x_2}} \{x | A_i[x_1/c_1](x) = true\}$$

Now we consider an arbitrary element c_2 of the above set (if it is empty, then go back and choose another value for x_1). We replace all occurrences of x_2 in atoms with the constant c_2 and iterate the process, sequentially for all the other variables. That is, we compute the sets:

$$\pi_{x_k}\sigma_{x_1=c_1 \wedge ... \wedge x_{k-1}=c_{k-1}}\Psi(\mathcal{E}) =$$

$$= \bigcap_{i \in I_{x_k}} \{x | A_i[x_1/c_1, ..., x_{k-1}/c_{k-1}](x) = true\}$$

sequentially, for $2 \leq k \leq n$. Once we have computed the last set, for every element c_n in that set we add the tuple $[c_1, c_2, ..., c_n]$ to $\Psi(\mathcal{E})$. Then, recursively, we take another value for c_{n-1} in its set and recompute the set for x_n, and so on, until we compute all the tuples. This recursive procedure is described in Fig. 2.

Procedure GENERATE $(\mathcal{E}, k)$
  $M = \bigcap_{i \in I_{x_k}} \{x | A_i(x) = true\}$
  for all $c \in M$
    $X(k) = c$
    if $(k < n)$
      GENERATE($\mathcal{E}[x_k/c], k+1$)
    else
      $\Psi(\mathcal{E}) = \Psi(\mathcal{E}) \cup \{[X(1), ..., X(n)]\}$
    end if
  end for
end GENERATE

/* Main Program */
INPUT: $\mathcal{E} = \{x_1 : T_1, ..., x_n : T_n | A_1, ..., A_m\}$
OUTPUT: $\Psi(\mathcal{E})$
METHOD:
  var X : array 1..n of Object
  $\Psi(\mathcal{E}) = \emptyset$
  GENERATE($\mathcal{E}, 1$)

Fig. 2. Algorithm computing $\Psi(\mathcal{E})$

Case II (G_D cyclic): Consider a cycle in G_D: $x_{i_1} \rightarrow x_{i_2} \rightarrow ... \rightarrow x_{i_k} \rightarrow x_{i_1}$. From the definition of atoms we infer that all the variables in the cycle are of the type $Node$ (a $Link$ variable always has outdegree zero in the dependency graph). From the fact that all variables are ground we infer that at least one of the vertices in the cycle has an incoming edge from outside the cycle or has an independent occurrence in some atom. Without loss of generality we can consider that x_{i_1} has this property.

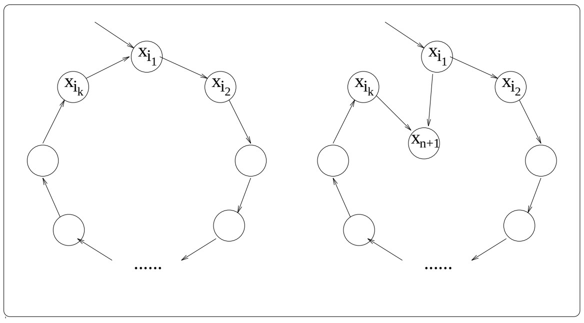

Fig. 3. Breaking a Dependency Cycle

We introduce a new variable x_{n+1} and replace all occurrences of x_{i_1} in the atoms where it depends on x_{i_k} by x_{n+1}. In this way, the edge $x_{i_k} \rightarrow x_{i_1}$ is replaced by an edge $x_{i_k} \rightarrow x_{n+1}$ thus breaking the cycle (see Fig. 3). Also, we add a new atom $A_{m+1} = Path(x_1, \epsilon, x_{n+1})$ to $\mathcal{E}$. Please note that all variables are still ground in the modified expression. We denote the new expression $\mathcal{E}$'. The atom A_{m+1} ensures that in all tuples in $\Psi(\mathcal{E}')$, $x_{n+1} = x_2$ (two different nodes cannot be separated by an empty path). This means that $\Psi(\mathcal{E}) = \pi_{x_1,...,x_n}\Psi(\mathcal{E}')$.

The new dependency graph G'_D has at least one cycle less than G_D. By iterating this procedure until there are no cycles left we obtain an expression $\mathcal{F}$ that can be evaluated using the method from Case I. Then, as the last step, we compute $\Psi(\mathcal{E}) = \pi_{x_1,...,x_n}\Psi(\mathcal{F})$. This concludes the theorem's proof.

Remark 1. The converse of the above theorem is not true, since there are computable range expressions which are not ground. For example, consider $\mathcal{E} = \{x : T, y : T, z : T | P(x), Path(y, False, z)\}$, where $False \in \mathcal{P}_{Link}$ is an unsatisfiable link predicate. Here y and z are not ground but $\Psi(\mathcal{E}) = \emptyset$, and therefore trivially computable. However, one can prove[4] that every computable range expression is equivalent to a ground range expression.

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 9. *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

[4] This comes as a consequence of a more general theorem in [MM96]

<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: $\mathcal{P}_{Link} = \{\mapsto, \rightarrow, \Rightarrow\}$ 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 Sect. 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 sub-network. 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 10. *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.

We next consider locality of queries. In our context, a query is a mapping from the domain of virtual graphs, to the domain of sets of tuples over simple types. As in the standard definition of queries, one can further require the mapping to be *generic* and *computable*. Since this is irrelevant to the following discussion, we ignore this issue here. Formal definitions of genericity and computability in the context of Web queries can be found in [AV97, MM96].

Definition 11. *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 when evaluated at node i* **depends** *on W, (for Γ and $\mathcal{G}$), if $i \in W$ and 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. If Q is evaluated at some node i, then the cost of accessing all the documents in such W is the sum of $(\rho_c(i, w))$ over all documents w in W such that $\rho_{Node}(w)$ is defined. We are interested in bounding 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.

Definition 12. *The* **locality cost** *of a query Q, when evaluated at node i, is the maximum, over all virtual graphs $\Gamma \in \mathcal{G}$, and over all sets W on which Q depends in Γ, of the cost of accessing every document in W from node i.*

We are interested in bounding the locality cost of a query with some function of the cost of accessing the whole network. 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). For a few examples, consider 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.

A query Q can be computed in two phases. First, the documents W on which Q depends on are fetched, and then the query is evaluated locally. Of course, for this method to be effective, computing which documents need to be fetched should not be more complex than computing Q itself. The following result shows that computing W is not harder, at least in terms of how much of the network needs to be scanned, than computing Q.

Proposition 1. *For every class of graphs $\mathcal{G}$ and every query Q, the query Q' that given a graph $\Gamma \in \mathcal{G}$ returns a W s.t. Q depends on W, also depends on W.*

Proof. We use the following auxiliary definition.

Definition 13. *Let Q be a query, let $\mathcal{G}$ be a class of graphs, let $G \in \mathcal{G}$ be a graph, and let W be a set of nodes in G. We say that Q is* **W-local** *for G and $\mathcal{G}$, if for every graph $G' \in \mathcal{G}$ that agrees with G about W, $Q(G') = Q(G)$.*

Note that a query Q **depends** on W for G and $\mathcal{G}$, if Q is W-local and there is no $W' \subset W$ s.t. Q is W'-local for G. We shall call such W a window of Q in G. (Not that there may be many windows for Q in G.)

The proof is based on the following claim:

Claim. For every two graphs G_1, G_2, every set of nodes W belonging to both graphs, and every query Q, the following hold:
(i) If G_1 agrees with G_2 about W and Q is W-local for G_1, then Q is also W-local for G_2.
(ii) If G_1 agrees with G_2 about W and Q depends on W for G_1, then Q also depends on W for G_2.

Proof. (Sketch) We first prove claim (i). If G_1 is W-local, then for every graph $G' \in \mathcal{G}$ that agrees with G_1 about W, $Q(G') = Q(G_1)$. This in particular holds for G_2. Also, since G_2 agrees with G_1 about W, then the set of graphs that agree with G_1 about W is exactly the set containing all the graphs agreeing with G_2 about W.

Thus for every graph $G' \in \mathcal{G}$ that agrees with G_2 about W, $Q(G') = Q(G_2)$, i.e. Q is W-local for G_2 . Claim (ii) follows immediately from claim (i).

We are now ready to prove the proposition. The proof works by contradiction. Clearly the window W' on which Q' depends must contain W (since W is part of the answer of Q'). Assume that $W \subset W'$. We shall show that Q' is W-local, a contradiction to the minimality of W'.

Assume Q' is not W-local. Then there must be some graph G' that agrees with G about W but where Q' has a different answer. i.e. Q does not depend on W for G'. But claim (ii) above says that if G agrees with G' about W and Q depends on W for G,then Q also depends on W for G'. A contradiction.

Although encouraging, the above result is in general not of practical use. It says that computing W is not harder in terms of the data required, but it does not say how to compute W. In fact it turns out that if the query language is computationally too powerful, the problem of computing W can be undecidable. For example, if your query language is relational calculus augmented with Web-SQL features, the W of a query $\{x \mid \phi \wedge Q_4\}$ is the whole network if ϕ is satisfiable, and is empty otherwise. (ϕ here is a simple relational calculus formula and has no path expressions).

If the query language is too complex, locality analysis may be very complex or even impossible. Nevertheless, there are many cases where locality cost can be analyzed effectively and efficiently. This in particular is the case for the WebSQL query language. The fact that the language makes the usage of links and links traversal explicit facilitates the analysis task. 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,[5] and s some bound on the number of documents in a single server.

[5] 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).

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 $\min(s, k^{l_i})$.Each of these documents may contain k global links, and in the worst case all the $k \min(s, k^{l_i})$ links 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, in the current Web architecture, 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.
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.[6] All this can be done in time polynomial in the size of the expression. Thus the whole bound can be effectively computed.
The above expression is a simple upper bound on the locality cost. A tight bound 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, so we omit it.
Observe that if any of the starred sub-expressions contains a local link, then $l = s$. This is because, 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

[6] It suffices to build a NFA for the expression and compute the lengths of the maximal path between two successive global links, not counting epsilon moves and internal links. l_i is infinite if the path between two successive global links contain a cycle with at least one local link, in which case $min(s, k^{l_i}) = s$.

Q_5: SELECT z, w
FROM Document x
SUCH THAT x MENTIONS "VLDB96"
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 "VLDB96"
Document w SUCH THAT $x \rightarrow w$

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

FROM Document x
SUCH THAT x MENTIONS "VLDB96"
Anchor y SUCH THAT base = x
Document z SUCH THAT $y \rightarrow^* z$

are of the form $((\rightarrow|\Rightarrow).(\mapsto|\rightarrow|\Rightarrow).\rightarrow^*)$. Thus the regular expression describing the path accessed by the query is $(((\rightarrow|\Rightarrow).\rightarrow) \mid ((\rightarrow|\Rightarrow).(\mapsto|\rightarrow|\Rightarrow).\rightarrow^*))$. 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 $(((\rightarrow|\Rightarrow).\rightarrow) \mid ((\rightarrow|\Rightarrow).(\mapsto|\rightarrow|\Rightarrow).\rightarrow^*))$.

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 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 first sketch below an algorithm for building a regular expression corresponding to the paths traversed by the FROM clause. Next we explain how the SELECT and WHERE clauses can be used in the locality analysis.

To build the regular expression we use the auxiliary notion of *determination.*

Definition 14. *Recall that every domain term A_i in the FROM clause corresponds to an atom A'_i in the calculus. We say that A_i* determines *a variable x, if x is independent in A'_i or it depends on another variable of A'_i. For two domain terms A_i, A_j, we say that A_i determines A_j if A_i determines any of the variables in A_j.*

For simplicity, we assume below that every variable in the FROM clause is determined be a single domain term. Observe that this is not a serious restriction because every query can be transformed to an equivalent query that satisfies the restriction: whenever a variable depends on several terms, the variables in all these terms but one can be renamed, and equality conditions equating the new variables and the old one can be added to the WHERE clause.

Given this assumption, the determination relationship between domain terms can be described by a forest. This forest is then used to derive the regular expression. The forest is built is follows: The nodes are the atoms in the FROM clause, and the edges describe the dependency relationship between variables.

Note that the roots in such a forest are either index terms (that is, MENTIONS terms), or anchor/path terms with a constant URL as the starting point. The non-root nodes are anchor or path terms. For example, the forest of query Q_5 above contains a single tree of the form

FROM Document x SUCH THAT x MENTIONS "VLDB96"
. | |
Document w SUCH THAT $x \rightarrow w$ Anchor y SUCH THAT base= x
. |
. Document z SUCH THAT $y(\rightarrow)^*$ z

The next step is to replace each term node in the forest by a node containing a corresponding path expression: Index terms are replaced by $\rightarrow \mid \Rightarrow$. Non-root anchor terms are also replaced by $\mapsto \mid \rightarrow \mid \Rightarrow$. Root ones are replaced by the same expression, if the constant URL appearing in them is at the local server, and by $\Rightarrow .(\mapsto \mid \rightarrow \mid \Rightarrow)$ otherwise. Non-root path terms are replaced by the regular expression appearing in the atom. Root ones with a URL at the local server are also replaced by this regular expression, and otherwise by $\Rightarrow$ concatenated to the expression. So for example, the above tree becomes

$\rightarrow \mid \Rightarrow$.
/ \
/ \
$\rightarrow$ $\mapsto \mid \rightarrow \mid \Rightarrow$
|
|
$\rightarrow^*$

Finally, we take the obtained forest and build from it a regular expression. This is done by starting at the leaves and going up the forest, concatenating the regular expression of each node to the union of the expressions built for the children, (and finally taking the union of all the roots of the forest). The expression thus obtained for the above tree is $(\rightarrow|\Rightarrow).(\rightarrow \mid ((\mapsto|\rightarrow|\Rightarrow).\rightarrow^*))$.[7]

The locality of the FROM clause is a bound on the locality of the query. A slightly better bound can be obtained by analyzing the SELECT and WHERE clauses. For the SELECT clause, if the documents corresponding to variables on which no other variable depend are not used, (i.e. only data from their URL is retrieved), it means that the last link of the path is not traversed.

[7] Which is equivalent to the expression $(((\rightarrow|\Rightarrow).\rightarrow) \mid ((\rightarrow|\Rightarrow).(\mapsto|\rightarrow|\Rightarrow).\rightarrow^*))$. we obtained in the intuitive discussion above.

This can be easily incorporated into the locality computation of the regular expression. For the WHERE clause, if the condition there is unsatisfiable, the locality can be immediately reduced to $O(1)$.

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. Java applications incorporated in HTML documents, called *applets*, reside on a Web server but are transferred on demand to the client's site and are interpreted by the client.

A prototype user interface for the WebSQL system is accessible from the WebSQL home page (`http://www.cs.toronto.edu/~georgem/WebSQL.html`) through a CGI script. Also, we have developed a standalone, GUI-based, Java application that supports interactive evaluation of queries.

The WebSQL system architecture is depicted in Fig. 4.

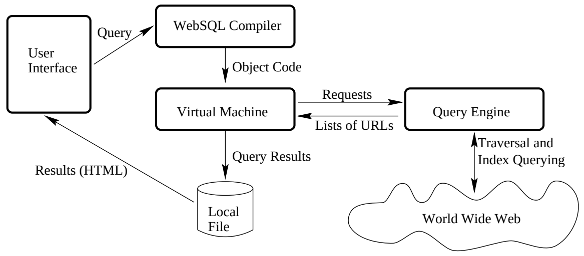

Fig. 4. The Architecture of the WebSQL System

The Compiler and Virtual Machine

Starting from the BNF specification of WebSQL we built a recursive descent compiler that, while checking for syntactic correctness, recognizes the constructs in a query and stores all the relevant information in internal structures.

After this parsing stage is complete, the compiler generates a set of nested loops that will evaluate the range atoms in the FROM clause. Consider the following query template:

$$
\begin{array}{ll}
\text{SELECT} & Fields(x_1, ..., x_n) \\
\text{FROM} & Obj\ x_1 \text{ SUCH THAT } A_1 \\
 & Obj\ x_2 \text{ SUCH THAT } A_2 \\
 & ... \\
 & Obj\ x_n \text{ SUCH THAT } A_n \\
\text{WHERE} & Condition(x_1, ..., x_n)
\end{array}
$$

The WebSQL compiler translates the above query to a program in a custom-designed object language implementing the pseudo-code algorithm depicted in Fig. 5.

```
Compute D_1 = {x_1 | A_1 = true}
for all x_1 ∈ D_1
   Compute D_2 = {x_2 | A_2 = true}
   for all x_2 ∈ D_2
      ...
         Compute D_n = {x_n | A_n = true}
         for all x_n ∈ D_n
            if Condition(x_1, x_2, x_3, ..., x_n) = true
               Write (Fields(x_1, x_2, x_3, ..., x_n))
            end if
         end for
      ...
   end for
end for
```

Fig. 5. The Nested Loops Generated by the Compiler

Note that this nested loops algorithm is equivalent to the recursive algorithm used in the proof of Theorem 1 (for this restricted form of domain specifying expressions).

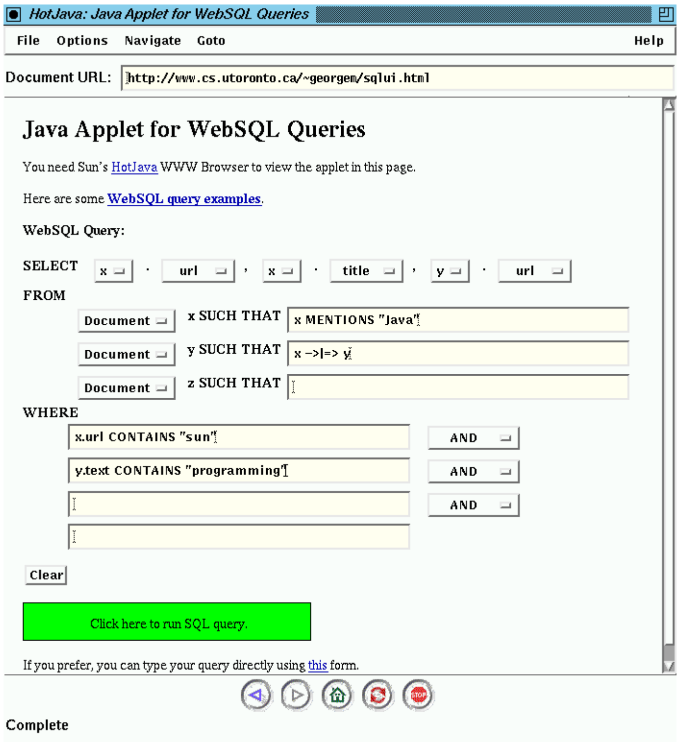

Fig. 6. The WebSQL User Interface

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.

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 4. 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

In order to make WebSQL available to all WWW users, we have designed a CGI C program invoked from a HTML form. The appearance of the HTML form is shown in Fig. 6, a screen shot of the Hotjava browser.

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 Java applet 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. Fig. 7 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

Fig. 7. WebSQL Query Results

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 applet 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 Fig. 6, 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 Information Technology Research Centre of Ontario, the Natural Sciences and Engineering Research Council of Canada, and the Israel Academy of Sciences and Humanities.

BLCL+94

[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.

[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.

[MM96] A.O. Mendelzon and T. Milo. Formal models of web queries. 1996. sumbitted for publication.

[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.

[NBY95] Gonzalo Navarro and Ricardo Baeza-Yates. A model to query documents by contents and structure. In *Proc. ACM SIGIR '95*, pages 93–101, 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`.

This article was processed by the author using the LaTeX style file *cljour2* from Springer-Verlag.