Just-In-Time Databases and the World-Wide Web

Ellen Spertus
Mills College
5000 MacArthur Blvd.
Oakland, CA 94613 USA
+1 510-430-2011
spertus@mills.edu

Lynn Andrea Stein
MIT AI Lab
545 Technology Square
Cambridge, MA 02139 USA
+1 617-253-2663
las@ai.mit.edu

1. ABSTRACT

It is not always possible or practical to store very large data sets in a relational database. For example, in a rapidly changing large-scale environment with distributed control, such as the World-Wide Web, a strict relational approach is not feasible. Nevertheless, it is desirable for a user to be able to make SQL queries on the entire data set, because SQL is well known, supported, and understood. We introduce "*just-in-time databases*", which allow the user to query the entire data set as though it were in a relational database. The underlying engine brings data into the relational database only when it is required by user queries. We describe how "just-in-time databases" are implemented, using the World-Wide Web as an example.

1.1 Keywords

Semi-structured information, Web, relational databases, SQL, federated databases, Internet

2. INTRODUCTION

2.1 Motivation

With the rise of federated databases comes the need to query very large, heterogeneous, distributed sets of read-only documents. Relational databases provide a powerful abstraction for manipulating data. The division of data into relations helps users comprehend the data and allows them to specify queries in Structured Query Language (SQL). This need to structure information is greatest when a data set is very large; however, traditional database management systems require that the entire data set be stored in a relational database, which is not always possible.

Although not usually classified as such, the World-Wide Web can be seen as a federated database. Specifically, it is a very large distributed collection of documents, composed of a number of different types of semi-structured information. Because of the Web's size, dispersion, and flux, it would be impossible to store the entire Web in a relational database.

We introduce "just-in-time databases", which provide the user with the illusion that an entire massive data set is in a local relational database. In reality, data is retrieved or computed only as needed. This allows the benefits of the relational database model to be obtained even when it is impractical or impossible to actually store all the data in a relational database. We provide algorithms for implementing just-in-time databases, using the Web as an example.

2.2 Overview

A just-in-time (JIT) database specification consists of:

- a relational database schema, which is first used to define tables in an underlying locally-stored traditional relational database management system (RDBMS)
- a set of functions for instantiating complete table rows, given values for a subset of columns

The illusion that the entire data set is in a local relational database is effected by the JIT interpreter. When the user issues a query on a JIT database, the JIT interpreter does the following:

1. Determines what data must be filled into the local database in order to answer the query, computing and inserting the data if it is not already present.
2. Passes the original query to the underlying RDBMS, returning the answer to the user.

In effect, the relational database serves as a cache on the data set, containing only the data needed to answer queries that have been made.

The next section contains definitions useful for describing the behavior of the system more precisely, followed by the algorithms used to determine what information must be inserted into the actual relational database. We then

illustrate our approach by describing an implemented JIT database for the Web. We conclude with a discussion of the significance of this work and how our Web database compares to other ways of extracting data from the Web.

3. JUST-IN-TIME DATABASES

3.1 Defining columns

A table in a relational database consists of a set of columns. We call a column *c* a *defining column* if an executable function exists that maps any value of *c* to all sets of legal values for the other columns. This function may not refer to the table itself (e.g., do "select" queries).

For example, if a table SQUARES contains two columns, *base* and *square*, representing an integer and its square, respectively, both columns would be defining columns. (*squares* and other tables used in this section are shown in Figure 2. Throughout this paper, table names appear in SMALL CAPS and column names in **bold face**.) If the value of **base** is known, the legal value for **square** can be easily computed. If the value of **square** is known, the legal associated values for **base** can be computed. If **square** is zero, there will be precisely one legal value for **base:** zero. If **square** is the square of any other integer, there will be two legal values for **base**, the positive and negative square roots. If **square** is not the square of an integer, there will be no legal values for *base*. In any case, knowing the value of either column determines the legal values for the other column.

On the other hand, a table ELECTION (Figure 2) whose two columns **voter** or **ballot** would have no defining columns if it described a secret ballot. Knowing the identity of a voter or the contents of a ballot would not allow determination of the other column value.

3.2 Fetch

We define an imperative operator *fetch* that, when given the values of one or more defining columns, augments a table by adding all legal relations having the given values. The syntax of the basic form of *fetch* is:

fetch tableName(*col0=value0, … colN = valueN*)

where *tableName* is the name of a table, *col0* through *colN* are defining columns, and *value0* through *valueN* are, respectively, legal values. For example, consider the following instance:

fetch SQUARES(*base*=4)

which would add the relation {base=4, square=16} to table SQUARES, unless it was already present. It is permissible that, as a side effect, additional relations may be added to SQUARES or other tables.

The full syntax for the *fetch* statement is shown in Figure 2. An optional "from" clause can be used to specify a set of values for a defining column. For example, if table PRIMES (Figure 1) has a column **number** containing the prime numbers less than 1000, the following statement would insert into SQUARES the primes between ten and fifty and their squares (unless they were already in the table):

fetch SQUARES(**base**=p.**number**)
from PRIMES p
where (p.**number** > 10 and p.**number** <50)

Within the *fetch* statement, a defined column can be assigned a single value, a conjunction, or a disjunction. For example, the following table would augment SQUARES with the squares of the numbers from one to five:

fetch SQUARES(**base** = 1 | 2 | 3 | 4 | 5)

The following statement would add no entries to SQUARES:

fetch SQUARES(**base** = 1 & 2)

since no value for **square** is associated with **base** values of both one and two. A conjunction that could augment *Squares* is:

fetch SQUARES(**base** = -2 & 2)

since there is a value for **square** (4) that is associated with both **base** values.

SQUARES

base	square
0	0
-1	1
1	1
-2	4

ELECTION

voter	ballot
Alice	yes
Bob	no
Carol	yes
David	yes

PRIMES

prime
2
3
5
7

Figure 2: Tables SQUARES, ELECTION, *and* PRIMES, *used as examples*

```
fetchStatement   =   "fetch" <tableName> "(" fetchList ")"
                     ( "from " <tableList>
                       ("where " logicExpression)?
                       ("group by " columnList)?
                       ("having " logicExpression)? )?

fetchList        =   namedArgument (", " namedArgument)*

namedArgument    =   <columnName> "=" fetchExpression

fetchExpression  =   expression "|" expression
                 |   expression "&" expression
                 |   expression
```

Figure 2: The grammar for fetch *statements. Nonterminals tableList, logicExpression, columnList, orderList, and expression are defined elsewhere [15] and have the same meanings as the corresponding SQL nonterminals.*

3.3 Virtual and Physical Tables

We use the term *virtual table* to describe a table whose contents are computed only as needed. Every virtual table must have at least one defining column. The term *physical table* describes ordinary SQL tables, whose contents are created with the "insert" instruction.

3.4 $Select_p$ and $Select_v$

The user need not be aware whether or not information is present in the relational database underlying the virtual tables, nor even if a table is virtual or physical. The JIT interpreter uses the internal $select_p$ statement, which performs the corresponding *select* query on the underlying *physical* database. In contrast, the $select_v$ statement is used to query the *virtual* database. Each $select_v$ statement is translated into zero or more *fetch* statements, followed by a $select_p$ statement. For example, consider the following statement:

$select_v$ s.**square**
from SQUARES s, PRIMES p
where s.**base** = p.**number**
and p.**number** < 100

It is converted into the following sequence:

fetch SQUARES(**base**=p.**number**)
from PRIMES p
where p.**number** < 100

$select_p$ s.**square**
from SQUARES s, PRIMES p
where s.**base** = p.**number**
and p.**number** < 100

The $select_p$ statements are interpreted by the underlying RDBMS.

3.5 Algorithms

At the core of the JIT interpreter is a set of algorithms for converting $select_v$ statements into *fetch* and $select_p$ statements. The algorithms are described in full elsewhere [15]. The process involves finding an ordering $\{t_1,\ldots t_n\}$ for referenced tables such that for every table t_i, either:

1. t_i is a physical table, or
2. a defining column of t_i is bound to a simple data object (such as an integer) or to a column of a table t_j, where $i > j$.

This allows a sequence of *fetch* statements to be generated to populate the local database so the query can be answered. Conjunctions and disjunctions are treated specially to provide some query optimization.

4. A Just-In-Time Database for the Web

This section describes a just-in-time database representing the Web. First, we describe the schema; then, we provide an example of its use.

4.1 Background

The Web can be viewed as a graph distributed over a network. All of its non-leaf nodes are written in *hypertext markup language (HTML)*, which includes tags and attributes to specify intra-document structure, such as a page's logical structure or how portions of it should be visually rendered. (Leaf nodes may be in HTML or other formats, such as Postscript.) HTML can also express inter-document structure through labeled *hyperlinks* relating one page to another. Each page is addressed through one or more *uniform resource locators (URLs)*, which are themselves structured. This abundance of structure has been underutilized because of the absence of a unifying interface to the different types of structural information. just-in-time Databases address this problem by providing a relational database interface to the Web, allowing the user to query the Web in SQL, taking full advantage of its many types of structural information.

4.2 Schema

Relations are divided into four categories: (1) basic relations, (2) tag and attribute relations, (3) relations built on tags, and (4) relations for second-hand information.

4.2.1 Basic Relations

In order to provide useful information about the Web, it is necessary that strings, URLs, and Web pages can be represented. The VALSTRING relation, shown in Figure 3 is used to associate an integer, **value_id**, with a string, **textvalue**; the integer is used in other tables to refer to the string. The column **textvalue** is a defining column. If a new **textvalue** is given, a new line is added to the table, with a new integer and the given **textvalue**.

The following relations all depend on VALSTRING:

1. URLS, which describes the set of URLs representing the same page.
2. PAGE, which represents information corresponding to a single page.
3. PARSE, which represents the components of a URL.

The URLS relation, also shown in Figure 3, is used to map one or more **value_id**s representing URL strings to a unique **url_id**. If there were a one-to-one correspondence between URL strings and **url_id**s, it would not be needed, but multiple URL strings can reference the same page. For example, both "www.ai.mit.edu" and "www.ai.mit.edu/index.html" reference the same document, so they are assigned the same **url_id**, with different variant numbers. Strictly speaking, there is no way to tell if two URLs reference the same file or merely two copies of a file. We use the heuristic that two URLs reference the same page and should have the same **url_id** if they retrieve identical contents and the URLs have the same host machine. Because the information known about URLs can change, it

VALSTRING

colname	*type*
value_id	int
textvalue*	text

URLS

colname	*type*
url_id	int
value_id*	int
variant	int

PAGE

colname	*type*
url_id*	int
contents*	text
bytes	int
md5	char(32)
when	datetime

Figure 3: The VALSTRING, URLS, and PAGE relations. Asterisks indicate defining columns.

PARSE schema

colname	*coltype*
url_value_id*	int
component	char(5)
value	char(255)
depth	int

PARSE relation

url_value_id	**component**	**value**	**depth**
950	"host"	"www.ai.mit.edu"	
950	"port"	80	
950	"path"	"index.html"	1
950	"path"	"people"	2
950	"ref"	"t"	

*Figure 4: The parse relation and an example of its contents for the **url_id** corresponding to "www.ai.mit.edu/people/index.html#t"*

is possible for the **url_id** associated with a string to change, in which case all **url_id**s in system and user tables are changed correspondingly. The **value_id** column is a defining column. If a new **value_id** is given, a new line is created with a unique **url_id** and a **variant** value of zero.

The PAGE relation (Figure 3) contains information obtained the last time a Web page was retrieved, including the text of a page, its size, a time stamp, and the results of a Message Digest 5 (MD5) hash function [11] on the page's text, providing a quick way to tell if two pages contain the same text. The **valid** field encodes the results of the last attempted retrieval of the page: "y" if it was successfully retrieved or "n" if it could not be retrieved. The columns **url_id** and **contents** are defining columns. If a **url_id** is given, the system can fetch the associated page from the Web and fill in the other fields. If a wildcard expression is given for **contents**, such as "%VLDB%", the interpreter can ask a search service, such as AltaVista (www.alta-vista.digital.com), to return all pages (that it knows of) containing the substring. These are then added to the table.

The PARSE relation (Figure 4) encodes a parsed version of a URL string. This is useful for determining if two URLs are on the same host machine or if one is above the other in the directory hierarchy. The component field contains "host", "port", "path", or "ref", and the value field shows the associated value. For the "path" component, the depth within the file hierarchy is also indicated, starting at the file name. Note that one **url_id** can have multiple parses, if multiple URLs correspond to that page. The following annotated transcript demonstrates the use of these basic relations.

*What **value_id** corresponds to the string "www.ai.mit.edu/index.html"?*
select **value_id** from VALSTRING where **textvalue**="www.ai.mit.edu/index.html"

value_id
950

*What **url_id** corresponds to the URL represented by the string "www.ai.mit.edu/index.html"?*
select **url_id** from URLS where **value_id** = 950

url_id
817

*Show all the known **url_id**s for this page.*
select * from URLS where **url_id** = 817

url_id	**value_id**	**variant**
817	950	1
817	1140	2

*What strings correspond to these **url_id**s?*
select v.**textvalue** from VALSTRING v, URLS u
where u.**url_id** = 817 and v.**value_id** = u.**value_id**

textvalue
"www.ai.mit.edu"
"www.ai.mit.edu/index.html"

*What is the host component of the URL corresponding to **value_id** 950?*
select **value** from PARSE where **url_value_i**d = 950 and component = "host"

value
"www.ai.mit.edu"

TAG

colname	*type*
url_id*	int
tag_id	int
name	char(15)
startOffset	int
endOffset	int

ATT

colname	*type*
tag_id*	int
name	char(15)
value_id	int

Figure 5: The TAG and ATT (attribute) relations

4.2.2 Tag and Attribute Relations

Hypertext consists of text annotated with tags and attributes. Hence, it is necessary that our system can represent tag and attribute information, which it does through the TAG and ATT relations, shown in Figure 5.

The TAG relation contains one line for each tag or set of paired tags encountered. (Single tags include "<hr>" (horizontal rule); paired tags include "<title>" and "</title>".) Information stored includes the **url_id**, tag name, the start and end character offsets of the tag within the document, and a unique **tag_id**, which is used to reference attributes. The **url_id** column is a defining column. Given a **url_id**, the system can retrieve a page and parse it, filling in the TAG table (and the ATT table as a side effect).

The ATT relation contains one line for each attribute name and value in a document. The name is a short string and the value an index into VALSTRING. We consider the anchor text enclosed by a pair of tags to be the value of the attribute "anchor". The following transcript illustrates the use of the TAG and ATT relations:

What tags occur within the first 100 characters on "www.mit.edu"?
select t.**name**, t.**tag_id** from TAG t, URLS u, VALSTRING v where v.**textvalue** = "www.mit.edu" and u.**value_id** = v.**value_id** and t.**url_id** = u.**url_id** and **startOffset** < 100

name	tag_id
!DOCTYPE	1080
HTML	1081
HEAD	1082
TITLE	1083

What are the attributes of the "TITLE" tag?
select * from ATT where **tag_id** = 1083

tag_id	name	value_id
1083	anchor	1105

What is the text associated with the "TITLE" tag?
select **textvalue** from VALSTRING where **value_id** = 1105

textvalue
"SIPB WWW Server Home Page"

4.2.3 Relations Built on Tags

Information about the structure of headers and lists on a page and hyperlinks across pages can be inferred from the information in the TAG and ATT tables, but for greater convenience we provide the user with specialized tables HEADER, LIST, and LINK, shown in Figures 6 and 7.

Through the HEADER table, we keep track of the levels of headings at each point of change. HTML supports six levels of headings (H1–H6). The **struct** column consists of an array of six bytes, each of which corresponds to one level of heading. At the beginning of a page, all six bytes are initialized to zero. Whenever a <H*n*> tag is encountered, byte (n-1) is incremented and all bytes with higher offsets are clear. The **ord** element is an ordinal number associated with each change of header structure. The defining column is **url_id**, since, given a **url_id**, a page can be fetched and parsed. As a side effect of computing the **header** relation, the **tag** and **att** relations are also computed.

The LIST relation keeps track of the list structure of a document. There are four types of lists supported by HTML: ordered lists, unordered lists, directories, and menus. These are not differentiated among in the LIST relation (but are in the TAG relation). Lists can be arbitrarily nested in HTML; the LIST relation keeps track of up to six levels of nesting of lists. As with HEADER, the only defining column is **url_id**. The details [15] are similar to, but not identical to, those for the HEADER relationship.

The LINK relation contains one line for each hyperlink encountered, storing the **source_url_id**, anchor text **value_id**, and destination **url_id**. For example, there is a hyperlink from the MIT AI Lab home page with anchor text "LCS" to the MIT Laboratory for Computer Science. The corresponding **url_id**s for the source ("www.ai.mit.edu") and destination ("www.lcs.mit.edu") URLs and the **value_id** for the anchor text ("LCS") appear as one line in the relation, as do the LIST and HEADER structures at which the hyperlink occurs. The **source_url_id** column is a

HEADER

colname	*type*
url_id*	int
struct	binary(6)
ord	int
offset	int

LIST

colname	*type*
url_id	int
struct	binary(6)
ord	int
offset	int

Figure 6: The HEADER and LIST relations

LINK

colname	*type*
source_url_id*	int
anchor_value_id*	int
dest_url_id*	int
hstruct	binary(6)
lstruct	binary(6)

Figure 7: The LINK relation

defining column because, given a URL, the system can retrieve the corresponding page, parse it, and determine the other fields. Less obviously, **anchor_value_id** and **dest_url_id** are also defining columns. This is done by having our system access AltaVista, which allows queries for pages that contain a given string as either anchor text or as the target of a hyperlink, although it does not (and could not) guarantee that it will return complete set. The system then verifies that the hyperlink still exists before adding it to the LINK relation.

4.2.4 Relations for second-hand information

In addition to direct information about the text and hyperlinks of Web pages, indirect information can be obtained from Web tools. It is important however to distinguish between verified and second-hand information, hence the RCONTAINS table, for reported contents of pages, and the RLINK table, for reported links. These are illustrated in Figure 8.

The RCONTAINS relation indicates the claim by the search engine shown in the **helper** column that a certain page contains a certain piece of text. The value of the **num** column (by default 10) determines the maximum number of such references added.

The RLINK relation contains search engine claims about hyperlinks among pages. Because of the functionality provided by the public search engines, the **source_url_id** field will always have a non-null value, but exactly one of **anchor** and **dest_url_id** will be null. In other words, a line can indicate the reported source and destination of a hyperlink, or it can indicate the reported source and anchor text of a hyperlink. The missing information, assuming the reported link actually exists, can be extracted from the LINK relation. As with RCONTAINS, the **helper** and **num** fields are used to specify the search engine and maximum number of lines added.

4.2.5 Bookkeeping

For each table, the system adds a column, **compute_id,** to keep track of what *fetch* statement caused each line to be inserted and when the *fetch* was performed [15]. This provides a hook for the user to ask questions about how fresh data is and how the Web has changed.

RCONTAINS

colname	*coltype*
url_id	int
value*	char(255)
helper	char(16)
num	int

RLINK

colname	*coltype*
source_url_id	int
anchor*	char(255)
dest_url_id*	int
helper	char(16)
num	char(16)

Figure 8: The RCONTAINS and RLINK relations

4.3 Application: A Recommender System

One useful class of information retrieval applications is recommender systems [33], where a program recommends new Web pages (or some other resource) judged likely to be of interest to a user, based on the user's initial set of liked pages *P*. A standard technique for recommender systems is extracting keywords that appear on the initial pages and returning pages that contain these keywords. Note that this technique is based purely on the text of a page, independent of any inter- or intra-document structure.

Another technique for making recommendations is collaborative filtering [13], where pages are recommended that were liked by other people who liked *P*. This is based on the assumption that items thought valuable/similar by one user are likely to by another user. As collaborative filtering is currently practiced, users explicitly rate pages to indicate their recommendations. We can think of the act of creating hyperlinks to a page as being an implicit recommendation. In other words, if a person links to pages *Q* and *R*, we can guess that people who like *Q* may like *R*, especially if the links to *Q* and *R* appear near each other on the referencing page (such as within the same list). This makes use of intra-document structural information. One can use the following algorithm to find pages similar to *P1* and *P2*:

1. Generate a list of pages *Parent* that reference *P1* and *P2*.
2. Generate a list of pages *Result* that are pointed to by the pages in *Parent* (i.e., are siblings of *P1* and *P2*).
3. List the pages in *Results* most frequently referenced by elements of *Parent*.

Figure 9 shows a sample run of this algorithm. Figure 10 shows the implementation.

Some improvements to the application that have been implemented and can be expressed easily [15] are:

1. Only return target pages that include a keyword specified by the user.
2. Return the names of hosts frequently containing referenced pages.
3. Only return target pages that point to one or both of *P1* and *P2* .
4. Only follow links that appear in the same list and under the same header as the links to *P1* and *P2* .

Preliminary evaluation [15] suggests that the last optimization yields results superior in some ways to the best text-based Web recommender system. Note that these heuristics simultaneously take advantage of *inter-document*, *intra-document*, and *intra-URL* structure.

Elsewhere, we discuss a home page finder, a moved page finder, and a technique for finding pages on given topics [14][15].

v.vcvalue	score
www.nasa.gov	13
www.nsf.gov	12
www.fcc.gov	5
www.nih.gov	5
daac.gsfc.nasa.gov	5
www.whitehouse.gov	4
www.cdc.gov	4
www.doc.gov	4
www.doe.gov	4
www.ed.gov	4

*Figure 9: First ten results of running SimPagesBasic (**Figure 10**) with the **url_id**s corresponding to "www.nsf.gov" (National Science Foundation) and "www.nasa.gov" (National Aeronautics and Space Administration).*

```
SimPagesBasic(page1id, page2id, threshold)

    // Create temporary data structures
    create table PARENT(url_id int)
    create table RESULTS(url_id int, score int)

    // Insert into parent the pages that reportedly link
    // to both pages that we care about
    insert into PARENT (url_id)
    select distinct r1.source_url_id
    from RLINK r1, RLINK r2
    where r1.source_url_id = r2.source_url_id
    and r1.dest_url_id = page1id
    and r2.dest_url_id = page2id

    // Store the pages pointed to by the parent pages,
    // along with a count of the number of links
    insert into RESULTS (url_id, score)
    select l.dest_url_id, COUNT(*)
    from LINK l, PARENT p
    where l.source_url_id = p.url_id
    group by l.dest_url_id

    // Show the URLs of pages most often pointed to
    // and the number of links to them
    select v.textvalue, count(*)
    from LINK l, PARENT p, VALSTRING v, URLS u
    where l.source_url_id=p.url_id
    and l.dest_url_id = u.url_id
    and u.value_id=v.value_id
    group by v.textvalue
    having count(*) ≥ threshold
    order by count(*) desc
```

Figure 10: Code for SimPagesBasic

5. Discussion

Just-in-time databases allow the user read access to a distributed semi-structured data set as though it were in a single relational database. One application of this work is a JIT database for the World-Wide Web, whose abundant structure has been underutilized because of the difficulty in cleanly accessing it. Enabling the use of SQL in querying the World-Wide Web provides a basis for powerful Web applications. We compare JIT databases to other database-oriented systems for accessing semi-structured information on the Web.

5.1 Related Work

An extractor developed within the TSIMMIS project uses user-specified wrappers to convert web pages into database objects, which can then be queried [3]. Specifically, hypertext pages are treated as text, from which site-specific information (such as a table of weather information) is extracted in the form of a database object. The Information Manifold provides a uniform query interface to structured information sources, such as databases, on the Web [5]. Both of these systems differ from our system, where each page is converted into a set of database relations according to the same schema.

This work is influenced by WebSQL, a language that allows queries about hyperlink paths among Web pages, with limited access to the text and internal structure of pages and URLs [8][7]. In the default configuration, hyperlinks are divided into three categories, internal links (within a page), local links (within a site), and global links. It is also possible to define new link types based on anchor text; for example, links with anchor text "next". All of these facilities can be implemented in our system, although WebSQL's syntax is more concise. While it is possible to access a region of a document based on text delimiters in WebSQL, one cannot do so on the basis of structure. Some queries we can express but not expressible in WebSQL are:

1. How many lists appear on a page?
2. What is the second item of each list?
3. Do any headings on a page consist of the same text as the title?

W3QL is another language for accessing the web as a database, treating web pages as the fundamental units [4]. Information one can obtain about web pages includes:

1. The hyperlink structure connecting web pages
2. The title, contents, and links on a page
3. Whether they are indices ("forms") and how to access them

For example, it is possible to request that a specific value be entered into a form and to follow all links that are returned, giving the user the titles of the pages. It is not possible for the user to specify forms in our system (or in

WebSQL), access to a few search engines being hardcoded. Access to the internal structure of a page is more restricted than with our system. In W3QL, one cannot specify all hyperlinks originating within a list, for example.

An additional way in which our SQL base makes our system differ from all of the other systems is in providing a data model guaranteeing that data is saved from one query to the next and (consequently) containing information about the time at which data was retrieved or interpreted. Because the data is written to a SQL database, it can be accessed by other applications. Another way our system is unique is in providing equal access to all tags and attributes, unlike WebSQL and W3QL, which can only refer to certain attributes of links and provide no access to attributes of other tags.

5.2 Future Work

The current system is a prototype. The JIT interpreter would be much more efficient and robust if it were integrated with a SQL database server. Currently, no guarantees are made as to atomicity, consistency, isolation, and durability. Compilation of virtual queries would be more efficient than the current interpretation, as would be exposing virtual queries to optimization. While we described here a JIT database backed up by a distributed semi-structured database (e.g., the Web), domains that are computed dynamically instead of retrieved are also possible.

6. ACKNOWLEDGMENTS

We are grateful to Oren Etzioni, Keith Golden, Ken Haase, Tom Knight, Pattie Maes, and Alberto Mendelzon for valuable discussions that improved this research. Ellen Spertus received funding from the Intel Foundation. Lynn Andrea Stein received funding through National Science Foundation Young Investigator Award Grant No. IRI-9357761 and through the Office of Naval Research through the Science Scholars program at the Mary Ingraham Bunting Institute of Radcliffe College. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of sponsoring agencies.

7. REFERENCES

[1] Gustavo O. Arocena, Alberto O. Mendelzon, and George A. Mihaila. Applications of a Web query language. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Cruz, CA, April 1997.

[2] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A Query Language and Optimization Techniques for Unstructured Data in *Proceedings of the 1996 ACM SIGMOD Iinternational Conference on Management of Data*.

[3] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the Web. In *Proceedings of the Workshop on Management of Semistructured Data,* Tucson, Arizona, May 1997.

[4] David Knopnicki and Oded Shmueli. WWW Information Gathering: The W3QL query language and the W3QS system. *ACM Transactions on Database Systems*, September 1998.

[5] Alon Y. Levy, Anand Rajaraman , Joann J. Ordille , Query answering algorithms for information agents. *Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, 1996.

[6] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54-66, September 1997.

[7] Alberto Mendelzon, George Mihaila, and Tova Milo. Querying the world wide web. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, Miami, FL, 1996.

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

[9] Dave Raggett. Html 3.2 reference specification. World-Wide Web Consortium technical report, January 1997.

[10] Paul Resnick and Hal R. Varian. Recommender systems (introduction to special section). *Communications of the ACM*, 40(3):56–58, March 1997.

[11] Ronald Rivest. The MD5 message-digest algorithm. Network Working Group Request for Comments: 1321, April 1992.

[12] Jacques Savoy. Citation schemes in hypertext information retrieval. In Maristella Agosti and Alan F. Smeaton, editors, *Information Retrieval and Hypertex*t, pages 99–120. Kluwer Academic Press, 1996.

[13] Upendra Shardanand and Pattie Maes. Social information filtering: Algorithms for automating "word of mouth". In *Computer-Human Interaction (CHI)*, 1995.

[14] Ellen Spertus. ParaSite: Mining structural information on the web. *In The Sixth International World Wide Web Conference*, April 1997.

[15] Ellen Spertus. ParaSite: Mining the Structural Information on the World-Wide Web. PhD Thesis, Department of EECS, MIT, Cambridge, MA, February 1998.