DISTRIBUTED QUERY PROCESSING ON THE WEB

Nalin Gupta Jayant R. Haritsa Maya Ramanath

Technical Report
TR-1999-01

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

Distributed Query Processing on the Web

Nalin Gupta
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore 560012, India

Jayant Haritsa*
Supercomputer Education and Research Centre
and Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore 560012, India

Maya Ramanath
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, India

Abstract

The problem of how to model and express queries on Web resources has been extensively studied. Designing processing engines for these queries has received comparatively little attention, however, and the few efforts in this area have assumed a centralized processing architecture wherein data is shipped from remote sites to the query site. We present here the design and implementation of **WEBDIS**, a distributed query processing engine that adopts a query-shipping model where queries are forwarded from one site to another. The WEBDIS design addresses several interesting issues that arise in the distributed context including determining query completion, supporting query termination, preventing multiple recomputations of the same query at a site, etc. The WEBDIS system has been implemented in Java and is currently operational.

*Contact Author : haritsa@dsl.serc.iisc.ernet.in

1 Introduction

In the initial stages, the interaction between Web technology and database technology was limited to where the Web was used primarily as a *communication medium* for transporting queries and data between networked users and relational database servers. The main challenge here was the design of interfaces that would facilitate embedding of SQL queries and their results into HTML and several such interfaces were developed – for example, the WWW Connection interface for IBM's DB2 system [15].

Later on it was realized that the Web could *itself* be viewed as an enormous (and potentially the largest in the world) database of information. Porting classical database technology onto the Web is rendered difficult, however, due to the heterogeneous, dynamic, hyper-linked and largely unstructured format of the Web and its contents. Further, the absence of a controlling entity equivalent to a database administrator makes it impossible to regulate the growth of the Web.

To address the above problems, a variety of data models and query languages for the Web have been proposed over the last few years. These include the SQL-like W3QL with interfaces to Unix tools [11], the declarative logic-based WebLog [12], the hyperlink-pattern-based WebSQL [14], as well as the OQL-based Lorel [3] and the graph-based UnQL [5] for semi-structured databases. However, the development of suitable *query processing strategies* for these models has received comparatively little attention. In particular, most of the previous efforts assume a "data shipping" mode wherein data is downloaded from remote sites to the query-server, queries are processed locally against these documents, and then further data is downloaded from the network based on these results.

A data shipping approach suffers from several disadvantages, similar to those already observed for traditional distributed database systems, including the transfer of large amounts of unnecessary data resulting in network congestion and poor bandwidth utilization, the client-site becoming a processing bottleneck, and extended user response times due to sequential processing.

Several web applications are more naturally processed in a distributed manner. For example, search engines [1] which index a substantial part of the web have to import millions of documents from various web-sites. Instead of downloading such a huge number of documents, it would be easier if the processing of documents took place at the web-sites themselves and only the results were sent back. As another example, applications which build "site maps" [2] for a particular domain of web-servers would require all hyperlinks from those web-sites to be extracted. Instead of downloading all documents from these web-servers and extracting links locally, it would reduce network traffic if processing was done at the web-servers themselves and only the list of links sent back. If not completely, atleast if part of the processing was done by suitable specification of structural predicates, it would considerably reduce the network congestion and speed up the map generation.

In our research, we have considered a "query shipping" approach wherein queries emanating from the user-site are *forwarded* from one site to another on the Web, the query is processed at each recipient site, and the associated results are returned to the user. That is, the queries are processed in a *truly distributed* fashion.

1.1 Information Model

Our goal is to efficiently support queries which may have predicates on both *structure* and *content* [12] – for example, a query like "Find all the Postscript documents that are two hyperlinks away from the ICDE home page".[1] We assume that all the sites involved in the distributed processing of these queries are hosting query processors that are capable of receiving and locally processing the query. However, beyond this basic requirement, the sites are not required to know anything about the (Web) structure or content of any other site. Further, the query can be submitted at any of the sites participating in this distributed processing effort.

At first sight, the above requirement of having query processors at all Web sites may appear unrealistic to fulfill – however, such distributed facilities are already becoming prevalent with the rapid spread of mobile agent technology. Similar architectures have also been successfully implemented in the Condor distributed job execution facility [13] (now productized by IBM and called LoadLeveler). Further, even if some sites were to refuse to participate in this effort, we can always revert back to the traditional centralized approach for the queries related to these sites.

In order to restrict the search space to a feasible level, the user has to first specify an initial set of *StartNodes*, which indicate the documents in the Web from which the search for results should begin. The set of StartNodes are obtained from either the user's domain knowledge or from existing search-indices (this process can be automated and made invisible to the user). Apart from the StartNodes, the user also has to specify the path to indicate how the query should traverse the Web and what processing should be carried out at each node along the path. Each web-server which receives a query, processes the query and determines the set of *NextNodes* to which the query should be forwarded to. The process goes on until the query has been completely answered.

1.2 Design Challenges

Within the framework of the above information model, a variety of interesting conceptual and practical issues arise in implementing a distributed processing approach to Web queries:

Query Language and Rewriting: Web queries can be broadly classified into two basic types [11]: (a) *content queries*, where the predicates are on the contents of Web resources, and (b) *structural* queries, where the predicates are on the hypertext organization. A structural query that is forwarded from one site to another may have to be *rewritten* to reflect the path already traversed. Therefore, a language to elegantly model Web queries and to facilitate their rewriting is required.

Query Completion: Since the query migrates from site to site without explicit user intervention, it is not easy to know when a query has *fully* completed its execution. Therefore, a special mechanism for detecting query completion has to be implemented.

Query Termination: If a user decides to cancel an ongoing query, this message has to be sent to all the sites that are currently processing the query. Again, as for query completion, determining this set requires special mechanisms to be put into place. Further, due to the inherent asynchrony involved in distributed processing, it is possible that a site to which a termination message has been sent may have already forwarded the query to other sites. In this situation, either it or the original query-server have to now send a fresh set of

[1] Note that such queries cannot be handled by current search-engines.

termination messages to these sites – this process can cascade to an arbitrary extent, similar to the problem of "anti-messages" chasing "event messages" in distributed optimistic simulation [7]. Therefore, a bounded termination mechanism has to be incorporated.

Query Recomputation: Since the Web has an uncontrolled hyper-linked structure, a site may receive the same user query *multiple times* due to the existence of a variety of paths from the query-server to this site. Therefore, a scheme to recognize duplicate arrivals and thereby prevent unnecessary recomputations is required. Note that this is especially important not only to save on local processing but also because of the *cascading effect* such recomputations could have due to query forwarding.

Returning Results: After the local results of a user query are computed at a site, how should they be returned to the user? One option would be for the results to trace back the path used by the query in reaching this site. An alternative solution would be for the processing site to directly send it to the query-server but this would require additional connection-related metadata to be sent along with the query to the query-server sites.

In summary, for all of the above reasons, developing a viable distributed Web query processing engine involves significant design complexity. We report here on the design and implementation of a Java-based prototype system, called **WEBDIS**, that attempts to efficiently address these challenges. WEBDIS was built as part of the DIASPORA project, a querying system for the WWW [16].

Apart from answering ad-hoc user queries, the WEBDIS system can also be used in a variety of Web-related applications. For example, it can be used to process queries whose purpose is to "gather" similar information from several different sites. By a suitable choice of predicates, queries can be restricted to either physical or logical hypertext organizations. Similarly, WEBDIS can be used for maintenance activities such as detecting the presence of "floating links" (links pointing to non-existent documents), a commonly encountered problem in web-site administration.

1.3 Organization

The remainder of this paper is organized as follows: The architectural design, the performance optimizations and the implementation details of the WEBDIS system are described in Sections 2, 3 and 4, respectively. A sample query execution on WEBDIS is presented in Section 5. Related work on the integration of Web and database technology is reviewed in Section 6. Finally, in Section 7, we present the conclusions of our study and identify future research avenues.

2 The WEBDIS System Architecture

In this section, we describe the architectural design of the WEBDIS system in detail. Due to space limitations, we only highlight the main features here. In our design, apart from incorporating several new features to serve the special needs of the distributed environment, we have attempted to utilize as far as possible the data modeling and query language concepts already presented in the "centralized" literature, with a view to facilitating the migration path from centralized to distributed processing.

For ease of exposition, we first define the following terms, some of which were introduced in [14]:

Node: A node is a web-resource (e.g. HTML documents, XML documents, multimedia files) residing at a web-site.

Link: A link is the hyperlink from one node to another. It can be of three types: *interior*, *local* or *global* depending on whether the destination of the hyperlink is within the web-resource, on the same server, or on a different server, respectively. For completeness, a *null* link category which refers to the web-resource itself is also included. In the sequel, we will denote the interior, local, global and null links using the symbols I, L, G and N, respectively.

Web-Query: This is an alternating sequence of node-queries and PREs, both of which are defined below.

Node-Query: A query to be evaluated locally on a given node.

Path Regular Expression (PRE) : Traversal paths on the Web are defined using Path Regular Expressions, which are built from the above link symbols using the *concatenation* ($\cdot$), *alternation* ($|$) and *repetition* (*) operators. For example, $\mathrm{N} \mid \mathrm{G} \cdot (\mathrm{L}\ ^*4)$ is a PRE that represents the set of paths containing the zero length path and all the paths that start with a global link and continue with zero or more local links upto a maximum of four.

User-site: The web-server at which the user submits the query.

StartNodes: The set of nodes at which the query processing commences.

Query-server: A Web-server which participates in the distributed processing of the web-query.

2.1 Web Model

Our model of the Web is very similar to that proposed in [14]. In particular, we model the Web as a *directed graph*, wherein each vertex is a Node and each edge is a Link (as per the above definitions). Further, traversal paths on the Web are defined using PREs.

2.2 Node Model

For ease of exposition, we assume in this paper that all web-resources are expressed in the form of HTML documents [6]. To model these documents, we employ the *relational document model* suggested in [14]. In particular, we model each document as tuple entries in the **DOCUMENT**(url, $title$, $text$, $length$), **ANCHOR**($label$, $base$, $href$, $ltype$) and **RELINFON**($delimiter$, url, $text$, $length$) "virtual" relations.

The DOCUMENT relation has a single entry per document and captures its identifying attributes and textual content. The ANCHOR relation contains one entry for each hyperlink contained in the document and includes information about the source ($base$) and destination ($href$) URLs as well as the link type ($ltype$) and the hypertext ($label$). While these two relations were proposed in [14], we have augmented this model with the RELINFON relation. The concept of a *rel-infon* was proposed in [12] to identify a group of homogeneous or related information in a HTML document and is formed by using the tag information in the HTML document. For example, text within a <**B**> ... </**B**> segment could be regarded as a rel-infon. The tag to be used is specified by the $delimiter$ attribute. Incorporating the rel-infon construct provides flexibility in querying information that is localized within regions of a document.

2.3 Query Specification

As mentioned earlier, a web-query is a sequence of node-queries connected by PREs. Given a starting point for the web-query's execution, a node or a set of nodes is reached by traversing the first link in the first PRE. On reaching this set of nodes, if the remaining PRE contains a null link, then the first node query is evaluated. Then the next link in the PRE is traversed to reach the next set of nodes. This process is repeated until there are no more node-queries to be processed.

Formally, a web-query can be stated as follows :

$$\mathcal{Q} = \mathsf{S}\ p_1\ \ q_1\ \ p_2\ \ q_2 \cdots p_n\ \ q_n$$

where S is the set of StartNodes from where the query begins its execution, q_k is the k^{th} node-query, and p_k is the PRE to be satisfied in traversing from the node which evaluates q_{k-1} to the node which evaluates q_k.

In WEBDIS, users specify web-queries through an SQL-like language called **DISQL**, which is internally translated into the above formalism. Each DISQL query is composed of a sequence of sub-queries (each subquery essentially maps to a $p_i q_i$ combination) – the *from* clause in each sub-query operates on the set of associated virtual relations (Document, Anchor and Relinfon), and also specifies the PRE p_i. An optional *where* clause is provided to specify conditions to be satisfied by the result tuples. The required attributes (specified by the user in his query) are then extracted from the tuples and sent to the user.

At the user interface level DISQL provides a single *select* clause that includes all the attributes requested by the user from among the entire set of virtual relations accessed in the web-query. Subsequently, however, the select-clause is split such that each node-query only refers to attributes of the virtual relations that are locally created for the node. That is, we assume that each node-query can be completely processed locally and inter-site communication or synchronization is not required.[2]

We now introduce the DISQL language using the following example queries (the complete grammar is given in [8]):

Example Query 1: Extract all the global links in the HTML documents on the Database Systems Lab web-server starting from the lab's homepage.

```
select  a.base, a.href
from    document d such that "http://dsl.serc.iisc.ernet.in" L* d
        anchor a
where   a.ltype = "G"
```

The query follows all local links starting from `http://dsl.serc.iisc.ernet.in`. It returns URL and the hyperlinks of each document which satisfy the condition `a.ltype = "G"`.

Example Query 2: Starting from the Computer Science and Automation (CSA) department homepage (http://csa.iisc.ernet.in) find the laboratories page and from the homepage of each lab, find the name of the convener of that lab.

While the previous query had only one node-query, this query has two node-queries. Further, in formulating the query the user additionally supplies his domain knowledge, which is as follows:

[2] We are currently working on augmenting the WEBDIS system to also handle node-queries that refer to multiple documents.

- the laboratories page of the CSA department has the substring ‘lab’ in its title and it is one local link away from the homepage of the department.
- each of the lab homepages are one global link away from the laboratories page of the CSA department and the name of the convener of each lab can be found within one local link from the lab’s homepage.
- the name of the convener is usually succeeded by a horizontal line.

For the above characterization, the associated DISQL query is written as follows:

```
select  d0.url, d1.url, r.text
from    document d0 such that "http://csa.iisc.ernet.in" L d0,
where   d0.title contains "lab"
        document d1 such that d0 G·(L*1) d1,
        relinfon r such that r.delimiter = "hr",
where   (r.text contains "convener")
```

There are two node-queries associated with the above web-query. The first node-query merely finds the document `d0` which is the ‘laboratories’ page of the CSA web-site and returns its URL. The next node-query is evaluated by document `d1` which is at a distance of one global link followed by a maximum of one local link from document `d0`. This node-query finds the name of the convener of the lab and returns the URL of the HTML document and the name back to the user.

The query is equivalent to the following :

$$\mathcal{Q} = \text{http://csa.iisc.ernet.in} \;\; \text{L} \;\; q_1 \;\; \text{G}\cdot(\text{L}^*1) \;\; q_2$$

where q_1 is

```
select  d0.url
from    document d0,
where   d0.title contains "lab"
```

and q_2 is

```
select  d1.url, r.text
from    document d1,
        relinfon r such that r.delimiter = "hr",
where   (r.text contains "convener")
```

2.4 Query Processing

We assume that a query processor capable of handling the DISQL queries discussed above is executing as a daemon process at each web-site .

The query is first sent to the query-servers corresponding to the *StartNodes* specified by the user in his DISQL query. Each of these query-servers completes its local processing of the query and sends back the generated results, if any, to the user-site. Further, based on the structural patterns encoded as PREs in the query, it may *forward* the rest

of the query to another set of query-servers. This set of query-servers is determined from the hyperlinks contained in the locally processed documents. These query-servers also perform similar query processing operations and the process continues until all the paths that match with the PRE have been fully explored and there are no more node-queries remaining.

To process a node-query, a query-server dynamically creates a temporary in-memory database of the virtual relations associated with the document. We assume that it is feasible to construct the database in memory since individual Web documents are typically small in size. After processing the node-query, the query-server immediately purges the database.[3]

The above scheme of distributed processing results in a simple query processor.

2.5 Query Forwarding

A node along a path visited by a web-query plays one of the following roles :

1. **ServerRouter node** :

   Here, a node-query of the web-query is locally evaluated against the contents of the node. If the node-query is successful (an answer is found), the results pertaining to this node-query are returned to the user. Further, if any more node-queries are left in the web-query then the node forwards this remaining part of the web-query to a subset of the links emanating from the node – the subset is determined by the current PRE to be satisfied. For example, the subset could be "all global links". If the node-query is unsuccessful, however, no further action is taken.

2. **PureRouter node** :

   Here there is no node-query to be evaluated. Instead, the query is simply forwarded on a subset of the links emanating from the node – the subset is determined by the PRE specification, similar to that for ServerRouters described above.

The exact steps are given below:

- if the node is a ServerRouter node, then

  1. evaluate the node-query
  2. if the current node contains no answer, stop processing, else continue
  3. create a duplicate query, called a *clone*, from the currently received web-query for each of the NextNodes as determined by the PRE at the current node
  4. for each NextNode,
     - modify the PRE information carried by the clone to reflect the traversal of the query to the NextNode
     - include the URL of the NextNode in the clone

[3] Of course, if the site expects that a node will receive several queries, it can choose to retain the associated database so that the construction cost does not have to be paid repeatedly.

– dispatch the clone to the host site of the NextNode[4]

- if the node is a PureRouter node then process from step 3 above

It should be noted that at each node along a path, the web-query gets cloned to reach the next node. This leads to multiple clones of the original query, perhaps in different states of completion, moving along various possible paths specified by the PRE.

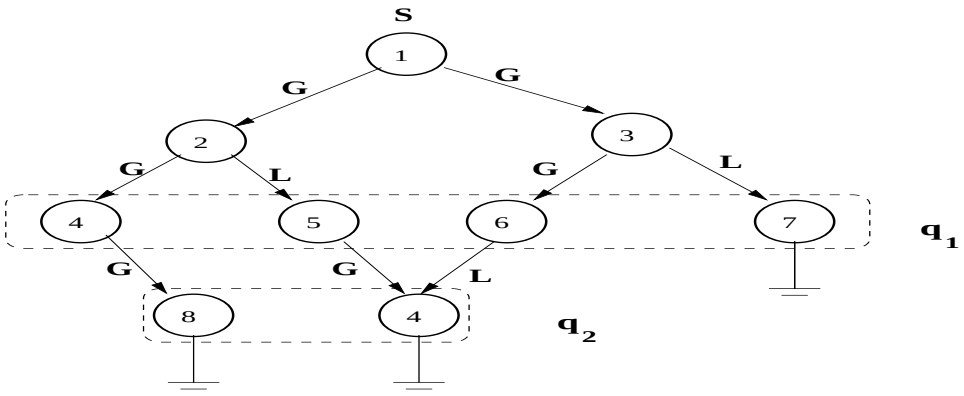

Figure 1: Web Traversal Path

To illustrate the above concepts, consider the web-query given by the following specification:

$$Q = S\ G \cdot (G \mid L)\ q_1\ (G \mid L)\ q_2$$

For this query, let $\{1,2,3,4,5,6,7,8\}$ be the nodes visited by the query as shown in the web traversal diagram in Figure 1. Here, nodes $\{1,2,3\}$ are PureRouters whereas $\{4,5,6,7,8\}$ act as ServerRouters. Note that Node 4 acts *twice* as a ServerRouter – once for processing q_1 and once for processing q_2 except that after answering q_2 no further forwarding takes place. Also, note that Node 7 becomes a "dead-end" since it fails to satisfy query q_1 and therefore does not forward the web-query any further.

2.6 Returning Results of Node-queries

As mentioned in the Introduction, one option for returning the local results of a query to the user would be to trace back the same path that the query took in reaching this site. However, this option has serious drawbacks: First, the entire history of nodes visited along the path will have to be maintained – that is, we cannot forget the past. This may prove expensive in terms of both storage and bandwidth utilization. Second, if the path is long, the results may take considerable time to reach the user. Finally, it requires query-servers to not only forward queries but also to forward results (in the reverse direction), thereby increasing the load on these servers.

In light of the above, we have instead chosen to set up a scheme wherein the results are *directly returned* from the query-server to the user-site. This is achieved by the user-site opening a *listening communication socket* to receive results – the associated port number is sent along with the web-query. When a query-server wishes to communicate results, it utilizes the IP address of the user-site and the port number which came along with the web-query to directly

[4]Note that a node-query needs to be explicitly "forwarded" only if the next node to be considered resides on a different site.

transmit the results to the user. Note that this scheme does not suffer from any of the drawbacks mentioned for the previous option.

2.7 Recognizing Query Completion

Since, as described above, clones of the web-query migrate from site to site without explicit user intervention, it is not easy to know when a web-query has *fully* completed its execution and all its results have been received – that is, how do we know for sure whether or not there still remain some clones that are active in the network. Note that solutions such as "timeouts" are difficult to implement in a coherent manner given the considerable heterogeneity in network and site characteristics. They are also unattractive in that a user may have to always wait until the timeout to be sure that the query has finished although it may have actually completed much earlier.

To address the above problem, we have incorporated in WEBDIS a special mechanism called the CHT (Current Hosts Table) protocol, described in the next subsection.

2.7.1 The CHT Protocol

To describe the CHT mechanism, we first need to define the processing state of a query. For our purposes, the state of a query Q_{clone}, denoted by $State(Q_{clone})$, is completely captured by the following:

num_q : The remaining *number* of node-queries yet to be processed by this clone before completion. Note that only the number is required, not the details of the queries.

$rem(p_i)$: The remaining part of the current PRE to be traversed before the next node-query can be evaluated.

So, for example, ($State(Q_{clone})$ = (2, G · L)), denotes that there are two more node-queries yet to be processed and that the traversal path to the next node-query is a global link followed by a local link.

For each user query submitted at a user-site, the local WEBDIS client process maintains a table called the *Current Hosts Table* (CHT). This table keeps track of all the nodes currently hosting clones of the user query. The attributes of the table are: (1) The URL of the node, and (2) The state $State(Q_{clone})$ of the query at the node. As described earlier in this section, after a clone arrives at a query-server and is processed, the local WEBDIS server determines the set of nodes to which the new set of clones should be forwarded. Before forwarding the query to these nodes, the query-server sends the information in the form of a list of rows to be added to the CHT at the user-site, along with the query identifier. It also adds its own URL and the state of the clone it received on top of the list. When the user-site receives this list, it marks the entry in its CHT corresponding to the top-most entry in the list as deleted. When all the entries in the CHT have been marked deleted, it can be concluded that the web-query has been completely processed.

Note that, at the query-server, only after the list is successfully sent is the clone forwarded to the next set of nodes. The reason we process in this particular order is to ensure that the CHT at the user-site will *always have complete knowledge* about the nodes at which the query is supposed to be currently executing and will therefore always be able to detect query completion. If the opposite order had been used, it is possible that the query may have been forwarded but the CHT not updated due to a transient communication failure between the current site and the user query site. This could lead to the possibility of the user-site wrongly determining that a query has completed when in fact it is still operational in the Web.

```
send_query (WebQuery Q) {
1.      nodes = get the URLs of the StartNodes from Q
2.      pre = get the first PRE to be satisfied
3.      numNQ = no. of node-queries in Q
4.      for each element n in nodes {
5.          enter (n, (pre, numNQ)) into CHTable
6.          dispatch Q to site(node)
7.      }
}
receive_results (Results result, List CHT_node) {
8.      repeat {
9.          store result
10.         mark the entry corresponding to CHT_node.topmostEntry in the CHTable as deleted
11.         merge the remaining entries in CHT_node into the CHTable
12.     } until (all entries in CHTable are marked deleted)
13.     process results for display
}
```

Figure 2: **The User-Site Algorithm**

2.8 Query Termination

If a user decides to cancel an ongoing query, this message has to be communicated to all the sites that are currently processing the query. One option would be for the user-site to actively send termination messages to all the sites associated with the nodes listed in the Current Host Table. An alternative would be to purge the query locally at the user-site and to close the listening socket associated with the query – subsequently, when any of the sites involved in the processing of this query attempt to contact the site to return the local results, the connection will fail – this is the indication to the query-server to locally terminate the web-query. Note that since we insist that the CHT related information should first be sent to the user-site before forwarding the query to other sites, we do not run into the problem mentioned in the Introduction of termination messages having to chase query messages along the Web.

The algorithm executed at the user-site is given in Figure 2. It shows two functions send_query (), which initially sends the web-query $\mathcal{Q}$ to the StartNodes, and receive_results () which processes the results and the CHT received from each query-server. The algorithm executed by the query-servers is given in Figures 3 and 4. The first function, process_query () receives the clone of the modified web-query from another query-server and, for each destination node of that clone, calls the process () function. The process () function carries out the actual evaluation of the query (if required) and returns (a) the results (if any), (b) the next set of clones to be forwarded, and (c) the CHT to be returned to the user-site.

```
process_query (WebQuery Q_clone) {
1.      nodes = get the URLs of the destination nodes from Q_clone
2.      pre = get the current PRE to be satisfied
3.      qid = query id of Q_clone
4.      if (pre contains null link)      // evaluate the query if null link is in pre
5.          nodeQuery = get the next node-query to be processed
6.      else
7.          nodeQuery = null
8.      for each element n in nodes {
9.          (clones, results, CHT) = process (n, nodeQuery, pre, Q_clone)
10.         if ((clones, results, CHT) are all null)
11.             continue     // n is a dead-end, no more processing needs to be done
12.         resultSet += (n, results)
13.         CHTset += (n, CHT)     // the complete CHT for all nodes which are sent back to client
        // contains all the clones for all the nodes in the form (site, Q'_clone)
14.         cloneSet += clones
15.     }
16.     forwardSet = all elements (site, Q'_clone) extracted from cloneSet
17.     dispatch (results, CHTset) back to client
18.     if (dispatch of results is successful)
19.         for each element f in forwardSet
20.             dispatch f.Q'_clone to f.site
}
```

Figure 3: **The Query-Server Algorithm - 1**

3 Performance Optimizations

In this section, we discuss some of the optimizations that we have included in the WEBDIS system to minimize the computation and communication overheads involved in supporting the distributed query processing framework.

3.1 Eliminating Query Recomputations

Due to the highly interconnected hypertext structure of the web, in the WEBDIS system, clones of a web-query may visit a node *multiple* times following different paths. In this situation, there are two possible cases:

1. The clone reaches the node in a *different state* of computation as compared to previous clones
2. The clone reaches the node in effectively the *same state* of computation as a previous clone

The above two possibilities are illustrated in Figure 5, which is based on the following web-query: S G · (G \| L) q_1 (G \| L) q_2. Here we see that Node 4 is visited five times a, b, c, d, e and that the state of computation in *c*, *d* and *e* are the same.

```
process (URL node, NodeQuery q, PRE p, WebQuery Q_clone) {
1.      if (q is not null) {
2.          results = evaluateQuery (q)      // node is a ServerRouter node
3.          if (results is null)
4.              return (null, null, null)      // node is a dead-end and no further processing takes place
5.      else      // node is PureRouter node
6.          results = null
7.      }
8.      construct set traverse = set of links to be followed from node as indicated by p
9.      construct the anchor table for node
10.     construct list CHT = null
11.     construct hashtable clones = null            //used to construct min. no. of clones
12.     for each link l in traverse {
13.         forward = select anchorTable.href such that anchorTable.ltype = l.ltype
14.         if (forward is null) then return (results, null, null) // node is a dead-end
15.         modifiedPRE = modify p to reflect traversal of next link of type l.ltype
16.         for each element f in forward {
        // check if a clone corresponding to site of node f has already been constructed
17.             Q'_clone = clones.get(site(f))
18.             if (Q'_clone is null) {       // if not, make a new clone
19.                 Q'_clone = clone (Q_clone)
20.                 set the current PRE in Q'_clone to modifiedPRE
21.                 add f to Q'.dest   // Q'.dest contains the URLs of the destination nodes
22.                 clones.put(site(f), Q'_clone)
23.             else      // otherwise, add f to the destination list of the clone
24.                 add f to Q'_clone.dest
25.                 clones.put(site(f), Q'_clone)
26.             }
27.             CHT +=f
28.         }
29.     }
30.     CHT += node      // node is inserted as the topmost element in CHT
31.     return (clones, results, CHT)
}
```

Figure 4: **The Query-Server Algorithm - 2**

While evaluating the node-query q_1 is mandatory in b, it is obviously a waste with respect to evaluating node-query q_2 in *c, d and e*. Note that if we do not detect the duplicate cases and blindly compute all queries that are received, not only is it a waste locally but subsequently the same sequence of steps followed by a previous clone will

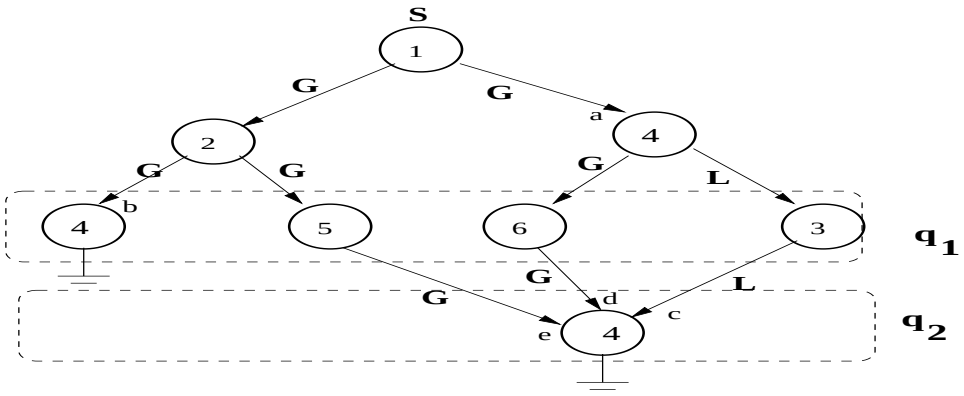

Figure 5: Multiple visits to a Node

take place – in effect, we will have a "mirror" clone chasing a previously processed clone over the Web. This will also have repercussions at the user site since the same set of results will be received multiple times and these will have to be filtered. In short, permitting duplicate query processing can have serious computation and communication performance implications.

From the above discussion, it is clear that each query server should be able to evaluate the current state of a clone and also store this information locally in order to permit future comparisons. Our solution to this issue is described in the remainder of this subsection.

3.1.1 Node-Query Log Table

The steps taken by the query server to process the query now involves the following :

1. At each site, we maintain a *log table* which contains the following information with regard to node-queries that have previously visited the site:

   URL_{Node} : The URL of the node on which the query is processed

   $Query_{ID}$: The global identifier of the query

   $State_{Query}$: The state of the query ($num\_q, rem(p_i)$)

   We denote a log-entry as [$URL_{Node}, Q_{ID}, State(Q_{clone})$].

2. When a new query arrives for a node, check whether an "equivalent" entry exists in the log table. We explain equivalent entries below.

3. If an equivalent entry exists for that node, purge the query for that node, otherwise, make a new entry corresponding to this query in the log table, rewrite the query, if required, (described below) and continue processing the query.

Equivalent Entries in the log table

Whenever a new clone arrives at the site, a new log-table record is created for it and it is checked whether an "equivalent" entry is already present in the log table. Obviously, one kind of equivalence is when the new record is completely identical to an existing entry, that is, they exactly match on all the three fields described above – in this case, the incoming query is dropped.

There are more subtle equivalences, however, that arise when all the fields are the same including num_q except that there are differences in the $rem(p_i)$ value. This is shown in the following example: Consider the case where $rem(p_i)$ of the log entry is $\text{L*2} \cdot \text{G}$ and that of the new entry is $\text{L*1} \cdot \text{G}$ – in this case the $\text{L*2} \cdot \text{G}$ is effectively a "superset" of $\text{L*1} \cdot \text{G}$ in that it will have already covered all paths that $\text{L*1} \cdot \text{G}$ would have taken. Therefore the new entry should not be considered. A more complex case is if the $rem(p_i)$ of the new entry has $\text{L*4} \cdot \text{G}$ – here some of the paths have already been considered earlier (those corresponding to $\text{L*2} \cdot \text{G}$) but not all – for example, the path $\text{L} \cdot \text{L} \cdot \text{L} \cdot \text{G}$ would not have been processed (assuming it exists). So, here we have a case of the new entry being a "superset" of the existing log entry and we have to ensure that *only the difference is processed*. For this, the query would have to be *rewritten*.

In general, let $(\text{A*m}) \cdot \text{B}\ q_k \cdots$ be the query received for node N by a query-server. Let there be a log entry $[URL_N, Q_{id}, (t, (\text{A*n} \cdot \text{B}))]$, where Q_{id} is the ID of the currently received query and t is num_q of $State(Q_{clone})$.

We now take the following steps based on the values of m and n :

m $\leq$ n : Ignore the new entry and don't process the query further, i.e. purge the query for N since it can't visit a node which has not been visited previously.

m $>$ n : Do the following :

1. Replace the existing log entry with the new entry; $[URL_N, Q_{id}, (t, (\text{A}*\text{m}) \cdot \text{B})]$.
2. Rewrite the query replacing $\text{A*m} \cdot \text{B}$ with $\text{A} \cdot \text{A*(m-1)} \cdot \text{B}$ and then undertake the normal query processing.

Step 2 above implies that, we are effectively forcing node N to be a PureRouter node. Because, otherwise, if B included the null link, we would have already evaluated the query at the current node. It is easy to see that the query will be rewritten at the first n nodes it subsequently encounters. Therefore, it may appear that a more efficient solution would have been to rewrite the query only once as $\text{A}^{n+1} \cdot \text{A}*(\text{m} - \text{n} - 1) \cdot \text{B}$ where A^i denotes A concatenated with itself i times. This would indeed be correct in ensuring that this clone subsequently only chooses paths that have not already been taken – the problem, however, is that comparing and updating the log table entries at the downstream nodes becomes ambiguous. For example, it would not be possible to distinguish between a "real" PRE that has $\text{L} \cdot \text{L}$ and a rewritten version of a PRE that originally had L*2 . Therefore, to avoid this problem, we have instead adopted the query-multiple-rewrite approach in the WEBDIS system.

If no equivalence of the above forms can be established with any existing log entry, a new entry is entered into the log table and the query is subsequently processed in the normal manner.

To ensure that the log table does not take undue space, the old entries in the table are periodically purged. The periodicity of the purging is a configuration parameter that should be set based on the disk storage available and the

processing duration of typical web-queries. Note that even if the purging time is incorrectly set too low resulting in duplicate Web queries being recomputed, it only affects the performance of the system but not the correctness of the results returned to the user.

Another point to note is that with the incorporation of the Node-query Log Table, a minor modification has to be made to the CHT protocol discussed in the previous section: A new entry that is equivalent to a previous entry should not be entered into the CHT since this new entry represents a duplicate query that will be detected and dropped at the target node.

3.2 Reduction of Network Traffic

We now move on to discussing other optimizations included in the WEBDIS system to minimize the network traffic:

1. No web resource is ever downloaded to perform a query operation over it since all processing is done at the node's home site itself. This is in marked contrast to the centralized approaches taken in search engines and in many of the previously proposed Web querying systems, including [14, 12, 11].

2. As mentioned in Section 2, the results obtained at each site are directly transferred to the user-site and do not have to retrace the path taken by the query in reaching this site. This is achieved by sending the IP-Address and listening port-number of the user-site as part of the web-query.

3. The node-query results and the newly generated set of (NextNode, QueryState) information to be added to the CHT at the user site are *shipped together*. Further, if a query is received for multiple nodes at the same web-site, all the associated results and NextNodes are batched together and sent to the user-site.

4. When forwarding a query, if the same query applies to multiple documents that are all physically located at a common remote site, the query is sent *only once* along with a list of all the nodes on which it should be processed.

5. Query termination is implemented passively, as described in Section 2.8, therefore not requiring additional termination messages from the query site to the sites currently hosting the query clones.

4 Implementation Details

In the previous sections, we discussed the architectural design of the WEBDIS system, and the associated optimizations. We follow up in this section with a discussion of the implementation details.

The WEBDIS system has been completely implemented in Java using JDK1.1.6 and JFC1.1 [18]. The PRE and DISQL parsers were generated using the *JavaCC* [9] parser generator. The serialization capabilities of Java were exploited to facilitate the forwarding of queries from site to site.

The four basic components of the WEBDIS system are the Web-Query Object, the User-Interface, the Client Process and the Server Process. In the remainder of this section, we describe these components.

4.1 Web-Query Object

The Web-Query Object is composed of two components: A QueryID and a sequence of node-queries. The QueryID component of a query carries the following information : (a) **UserID**: login-name of the user on his machine (b) **IP-Address**: internet address of the user's machine (c) **Port Number**: port number at which results are received (d) **Query Number**: locally unique number given to the query submitted by the user
The above information is used for sending back results directly to the user, for making log entries at the query-servers, and for collecting all the results of a web-query in a single file.

4.2 User Interface

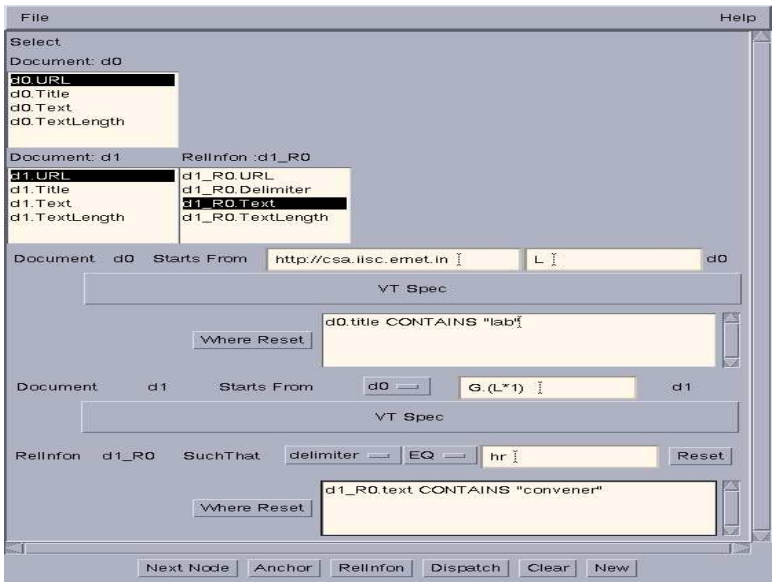

Figure 6: Sample Web Query

The WEBDIS system provides a simple and dynamic graphical user interface, which hides most of the syntactic details required to specify the DISQL query. Most of its features are automatic, including generating the variable names for table declarations, the select list, the previous anchor tables list, etc. The user only needs to fill in the conditions he wants the various node-queries to satisfy (leaving a field blank implies no conditions apply). A sample picture of the GUI is shown in Figure 6, corresponding to the second example query discussed in Section 2.

4.3 Client Process

After the user submits the query using the above interface, a series of operations are undertaken by the WEBDIS client process at the user-site.

1. The *DISQL Parser* receives the DISQL query from the User-Interface and converts it into an equivalent set of node-queries.

2. The *Result Collector* opens a listening socket at a currently unused port number and waits for results to arrive on this port. The port number is also included in the web-query object.

3. The *Query Dispatcher* receives the web-query object from the parser along with the set of StartNodes. It makes a socket connection with each of the StartNodes and dispatches the query objects on these connections.

4.4 Server Process

The WEBDIS system's Query Server runs as a daemon process at all the web sites. The server is composed of the following components:

1. The *Query Receiver* runs as a thread of the query server and listens on a *common pre-specified port number* at all sites.

2. The *Query Processor* is a thread of the query server and sequentially processes the queue of pending web-queries. The algorithm is given in Figures 3 and 4.

3. The *Query Dispatcher* is used by the query processor to forward each web-query to its associated set of NextNodes.

4. The *Result Dispatcher* sends back the results of a query to the user site using the network location information that is encapsulated in the QueryID component of the web-query object.

5. The *Database Constructor*, which is part of the query processor, builds the virtual relation database in memory for each node-query. A single pass is made over the associated document and during this process the tuples belonging to the various tables are formed one by one and then inserted into the appropriate table.

5 Sample Query Execution

As mentioned earlier, the WEBDIS system is fully operational. In this section, we demonstrate the working of the system through an actual execution of the second example query discussed in Section 2 on the Web network in our campus.

The user first enters the query using the GUI interface previously shown in Figure 6. The query then traverses through a variety of nodes and this traversal path is shown in Figure 7 (to ensure readability, only a part of the traversal path is shown). The state of the query as it traverses from node to node is also shown in this figure.

The query initially starts from the homepage of the CSA department with two node-queries and the first path expression consisting of only one local link corresponding to the first node-query. The query then follows all local links from the CSA homepage, but finds only one page which satisfies the condition in the first node-query (the substring "lab" in the title of the document). The query's state is now changed to reflect the fact that it now has one node-query to process with the PRE $G \cdot L^*1$, i.e., following a global link and then a maximum of one local link. Some of the nodes return answers and forward the query, while others may simply forward the query. Some nodes turn out to be dead-ends since they either fail the node-query restrictions or do not have any links satisfying the current PRE specification. Finally, when there are no more node-queries left, query forwarding is stopped.

The final results of the above query are shown in Figure 8.

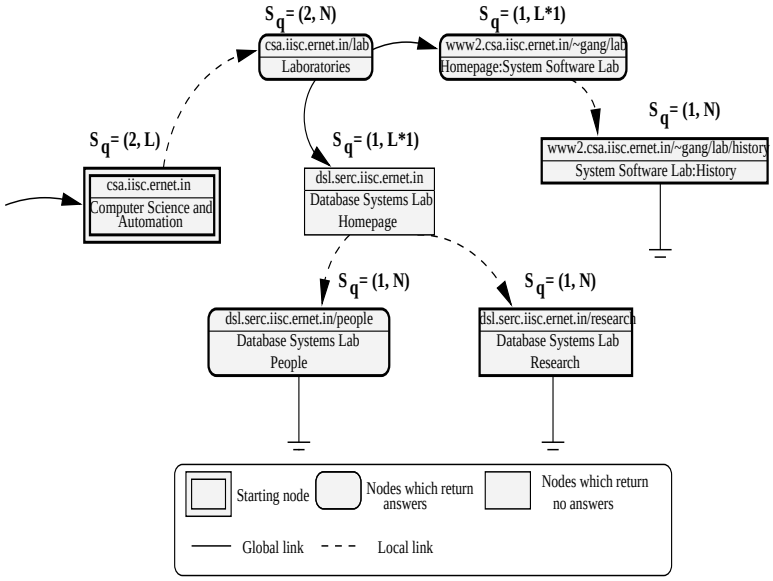

Figure 7: Traversal of the Query

6 Related Work

As mentioned in the Introduction, most of the earlier work on integrating Web and database technology has focussed on the data model and query specification aspects and has assumed a centralized query processing architecture. With regard to distributed query processing specifically, which is the focus of this paper, we discuss some related work in this section.

An algorithm for distributed processing of query paths using asynchronous message passing is presented in [4]. The query path is successively shortened as and when a part of it is satisfied by a site. The remaining part of the query is sent to successive sites. This approach appears similar to ours, but their termination detection is quite different: A message is sent along with the query from the user site to the StartNode and when the StartNode acknowledges this message, it indicates the termination of the query. The StartNode acknowledges the message only if all the nodes to which it had forwarded the query have acknowledged the message. This propagates to further nodes until a node is reached which answers the query. This is in contrast to our approach where the termination detection is the responsibility of the user site only.

A different type of distributed processing algorithm is outlined in [17]. Here, the basic idea is to send the query to the various sites, where they are processed and the results are sent back to the client for recombination. This paper also describes query decomposition for distributed querying. There are some important differences in this approach compared to ours. First, all the sites involved in processing the query are assumed to be completely known apriori whereas we do not make that assumption. Second, in their algorithm, at each site all the "input nodes" (that is, all the local documents which are hyper-referenced from documents residing at other sites) need to be known, whereas no such knowledge is required in our approach.

Both the above papers focus primarily on the theoretical aspects and do not describe any implementation mechanisms for their models.

Recently, a distributed query processing algorithm where agents called "navigators" are dispatched to various

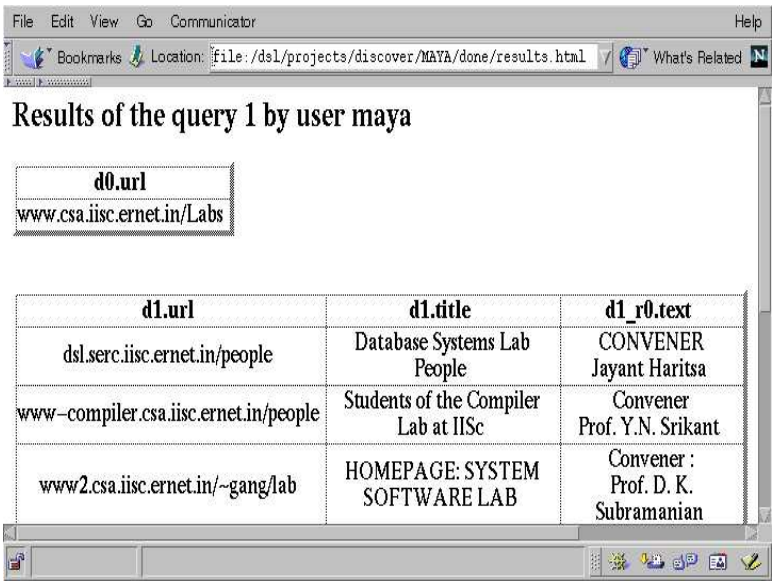

Figure 8: Results of the Query

web-sites to find "qualified paths" for a given PRE was described in [10]. In this algorithm, an automaton is first constructed for a given PRE. This automaton is then broken down into sub-automatons each of which may be dispatched to different sites to determine if nodes which satisfy the PRE of the sub-automatons exist. The main difference between this approach and ours is that the navigators are *co-ordinated centrally* by the client, whereas, no such co-ordination is required in our approach.

7 Conclusions

In this paper, we have presented the design and the prototype implementation of the WEBDIS system, intended for processing queries in a distributed manner over the Web. Apart from incorporating several new features to serve the special needs of the distributed environment, the WEBDIS design attempts to utilize and integrate as far as possible the data modeling and query language concepts already presented in the centralized literature, with a view to facilitating the migration path from centralized to distributed processing. In particular, WEBDIS builds on the Web and document models proposed in [12] and [14].

Users interact with the WEBDIS system through DISQL, an SQL-like declarative query language. Both content and structural queries are supported in the language. User queries are decomposed into an equivalent set of node-queries which are forwarded from site to site using a socket communication platform. Query completion is detected through the Current Hosts Table at the user site while duplicate queries are recognized at the server sites using the Node-query Log Table. Further, a variety of optimizations to reduce computation and communication overheads have been incorporated in the design. Overall, WEBDIS's distributed processing scheme results in a simple query processor unlike the complex stack machine of [14] and the need to coordinate parallel processes in [11].

The WEBDIS system is currently operational and has been implemented in the Java platform-independent language. We expect that the system will be of use in a variety of Web-related applications, including development of search-engine indices and link maintenance, apart from answering ad-hoc user queries.

7.1 Future Work

We are working on extensions to our basic distributed query processing scheme to incorporate web-servers which do not participate in the WEBDIS system. As mentioned in the Introduction, queries related to documents on such web-servers can be handled in the traditional manner by retrieving all documents from the remote site and then applying the query predicates locally at the user-site. Therefore, we can expect a gradual migration path for WEBDIS from a largely centralized to a fully distributed system as more and more sites begin to host query servers.

The DISQL query language is being improved to handle multi-document node-queries and more complex structural predicates. We are also working on supporting approximate queries and graceful recovery from node failures. Further, we are exploring ways in which existing search-indices can be used to augment the user's domain knowledge.

References

[1] "A Webmaster's Guide to Search Engines", http://searchenginewatch.com/webmasters/index.html

[2] "An Atlas of Cyberspaces", http://www.cybergeography.org/atlas/atlas.html

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. Weiner, "The Lorel query language for Semistructured Data", *Journal of Digital Libraries, 1(1)*, 1997.

[4] S. Abiteboul and V. Vianu, "Regular Path Queries with Constraints", *Proc. of 16th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, May 1997.

[5] P. Buneman, S. Davidson, G. Hillebrand and D. Suciu, "A Query Language and Optimization Techniques for Unstructured Data", *Proc. of ACM SIGMOD Conf. on Management of Data*, June 1996.

[6] T. Berners-Lee and D. Connolly, "Hypertext Markup Language - 2.0", *RFC1860*, November 1995.

[7] R. Fujimoto, "Parallel Discrete-Event Simulation", *Comm. of the ACM, 33(10)*, October 1990.

[8] N. Gupta, "DISCOVER the Web-Database", ME Thesis, Dept of Computer Science and Automation, Indian Institute of Science, July 1997.

[9] Java Compiler Compiler, (JavaCC) Version 0.6(Beta), http://www.suntest.com/JavaCCBeta.

[10] K. Katoh, A. Morishima, and H. Kitagawa "Navigator-based Query Processing in the World Wide Web Wrapper", *Proc. of the 5th Intl. Conf. of Foundations of Data Organization, Nov. 1998*

[11] D. Konopnicki and O. Shmueli, "W3QS: A Query System for the World-Wide-Web", *Proc. of 21st VLDB Conference*, September 1995.

[12] L. Lakshmanan, F. Sadri and I. Subramanian, "A Declarative Language for Querying and Restructuring the Web", *Proc. of 6th. Intl. Workshop on Research Issues in Data Engineering*, February 1996.

[13] M. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations", *Proc. of the 8th Intl. Conf. of Distributed Computing Systems*, June, 1988.

[14] A. Mendelzon, G. Mihaila and T. Milo, "Querying the World Wide Web", *Journal of Digital Libraries, 1(1)*, 1997.

[15] T. Nguyen and V. Srinivasan, "Accessing Relational Databases from the World Wide Web", *Proc. of ACM SIGMOD Conf. on Management of Data*, June 1996.

[16] M. Ramanath. DIASPORA: A Fully Distributed Web-Query Processing System. Master's Thesis, Indian Institute of Science, http://dsl.serc.iisc.ernet.in/thesis/maya.ps.gz.

[17] D. Suciu, "Distributed Query Evaluation on Semistructured Data", http:// www.research.att.com/~suciu/strudel/external/files/_F662777668.ps, 1997.

[18] Sun Microsystems. The Java Language : Programming for the Internet, http://java.sun.com.