Querying Semantically Tagged Documents on the Web*

Ziv Bar-Yossef† Yaron Kanza‡ Yakov Kogan‡ Werner Nutt§
Yehoshua Sagiv‡

Abstract

We have developed and implemented QUEST—a system for QUErying Semantically Tagged documents on the World-Wide Web. The advent of new markup languages like XML makes it likely that documents on the Web are not simple HTML documents, but contain in addition objects that represent the semantic structure of the document. When such documents are available, more powerful methods to access and query information on the Web will become possible.

We describe the system QUEST, which is designed to query documents in the markup language OHTML and to generate such documents as query results. OHTML combines HTML and objects of a semistructured data model.

The new features of QUEST are: (1) it can be used to query a combination of hypertext and object model; (2) the results of queries are OHTML pages and thus of the same type as the data being queried; (3) QUEST implements a new approach for querying semistructured data that allows answers to be incomplete.

Our experience in querying semantic documents on the Web can be useful for the design and implementation of query languages for XML. This paper provides an overview of the QUEST system and of the experience gained from its design and implementation.

1 Introduction

The enormous growth in the usage of the World-Wide Web as a way to access information, suggests that the Web will soon become a more database-like information source than it is today. The main obstacle for the evolution of the Web into one giant database is that most of the information accessible through the Web is stored as HTML files and that HTML lacks semantics. The lack of semantics makes it difficult to distinguish between different pieces of information in these pages. Currently, researchers are using information retrieval and machine learning techniques to reconstruct the semantic structures implicit in Web pages. However, these approaches tend to be inaccurate and insufficient for extracting all the information available.

In the QUEST project we have used OHTML [KMSS98], which enriches HTML with semantic tags that define an object structure. The semantic tags are hidden as HTML comments and hence, their existence is transparent to HTML browsers. We started our

*This research was supported by Grant 8528-95-1 of the Israeli Ministry of Science.

†Department of Electrical Engineering and Computer Science, U.C. Berkeley, Berkeley CA 94720.
‡The Hebrew University, Jerusalem 91904, Israel.
§German Research Center for Artificial Intelligence (DFKI GmbH), 66123 Saarbrücken, Germany.

project before the advent of XML. The techniques we developed for OHTML, however, are also applicable to HTML with "semantic islands" formulated in XML.

The data on the Web are irregular and do not conform to a rigid schema. They evolve rapidly and their structure is constantly changing. For these reasons we regard the information available from semantically-tagged pages on the Web as a *semistructured* database (see [Abi97, Bun97]). As a concrete data model we decided for the "object exchange model" OEM that was proposed for semistructured data in the *Tsimmis* project at Stanford [PGMW95].

This paper describes QUEST, a system for querying semantically tagged documents on the Web. QUEST treats a set of OHTML pages as a semistructured database. The usage of semantic tags allows one to pose more precise queries than is possible over untagged HTML pages. We consider tagged pieces of information in Web pages as atomic values. QUEST queries refer to both, the structure of objects and their atomic values.

The novel aspects of QUEST include: (1) a graphical query language; (2) the possibility to specify queries that retrieve incomplete information, thus taking into account the incompleteness of semistructured sources; (3) answers of queries that are an extension of the initial database.

QUEST provides a graphical query language for querying the database. We believe that in this way complex queries can be specified more accurately than by a textual query languages such as Lorel [AQM+96].

An important feature of QUEST is its ability to deal with the incompleteness of information in semistructured sources by admitting incomplete answers. Technically, this is achieved by retrieving assignments to the query variables that do not satisfy all the constraints in a query, but that satisfy them "maximally" (cf. [KNS99]).

QUEST creates as the result of a query a set of new semantically tagged Web pages that extend the current database. That enables one to post a query to the result of a previous query and therefore to compose queries. This is a common feature in relational databases but not of query languages for the Web.

QUEST uses a main memory approach when querying the Web. The Web is used as the database and QUEST does not use an extra database other than the Web. This approach is becoming more and more popular, since it avoids duplicating data and creating redundancy by storing data in more that one place.

During the last few years many languages have been proposed for querying the World-Wide Web motivated by the growing use of the Web as a heterogeneous information source. Languages such as W3QL [KS97, KS95], WebLog [LSS96], WebSQL [MM97, MMM97], WebOQL [AM98] and others have been developed. Many early languages for querying the Web were based on navigational search and queries were executed by a search engine for finding HTML pages on the Web. As a further development languages where designed for Web site management, e.g., Strudel [FFK+98] and Araneus [AMM97, MAM+98]. Araneus and Strudel supply tools to create new Web pages. Other query languages have been developed for querying semistructured data that do not necessarily reside on the Web, e.g., Lorel [AQM+96, AQM+97, MAG+97, QWG+96] and UnQL [BDHS96].

Today, with XML [Con98] becoming a standard, querying semantically tagged pages becomes a research issue. Languages such as XML-QL [DFF+98] and XQL of Microsoft have been proposed recently. These languages though, have not yet been implemented and we believe that our experience can be useful for the design and implementation of languages for XML.

QUEST was developed and implemented at the Hebrew University of Jerusalem. It is a client-server system where the client is a Java applet. Due to security considerations it is not yet available on the Web.

2 QUEST and How It Is Used

In this section we give an overview of QUEST from the user's perspective. We show how a user can formulate queries and view their results. In later sections, we will describe QUEST's query language and its components in more detail.

To illustrate the usage of our system, we rely on a running example based on the Web site of the Israeli economic magazine GLOBES [GLO], which holds information about the Israeli economy. We concentrate on the part of the site that deals with stock portfolios recommended by financial analysts. A portfolio comprises a group of companies recommended for investment. There is a general style in the design of the HTML documents which describe portfolios. However, each one of the portfolios has its particular schema. The schemas differ in the attribute names and in the hierarchies of the objects they contain. For example, some portfolios are flat lists of companies preceded by a short introduction while other portfolios group companies by sectors, e.g. "Electronics," "Chemical," etc. The portfolio pages are a good practical example for semistructured data. They contain incomplete data without a strict schema and they contain concrete information, such as prices, dates, etc., along with descriptions, images, links, etc. It seems natural that one would like to ask queries against these pages.

For our experiments, we copied pages of the GLOBES site to our computer, and tagged the pages according to the OHTML data model. In some cases, pages could be tagged by wrappers written in Perl, in other cases, we added tags manually. On this set of pages, we defined queries and evaluated them in QUEST.

2.1 Overview of the Querying Process

QUEST is a system for querying semantically tagged documents on the World-Wide Web (WWW). By semantically tagged documents we mean documents in which a "role" is explicitly given for some or all parts, and the roles specifies the meaning of these parts. For example, the homepage of a person may contain different numbers. By adding semantic tags one can make clear that one number is the phone number at the persons office, and another one is the ZIP code in the address.

One of the purposes of the recently proposed markup language XML is to express the semantics of document parts through structured labels. We started our project before the advent of XML and devised our own language OHTML [KMSS98], in which a layer of semantic objects can be added to HTML documents through tags. We give a more extensive description of OHTML and its usage in Section 3.

We consider a set of OHTML documents as a *database.* A database has two aspects. The first is the visualization of the HTML part of the documents as shown by a browser. We call this the *visual view.* The second aspect is the graph structure of the set of objects contained in the database. We call this the *semantic view.* Figure 1 shows how QUEST provides simultaneously the two views of the GLOBES database. The two aspects are connected in that a user can click on a node in the graph of the semantic view and, as a

result, the portion of the HTML document that is the visual view of the selected object is shown by the browser.

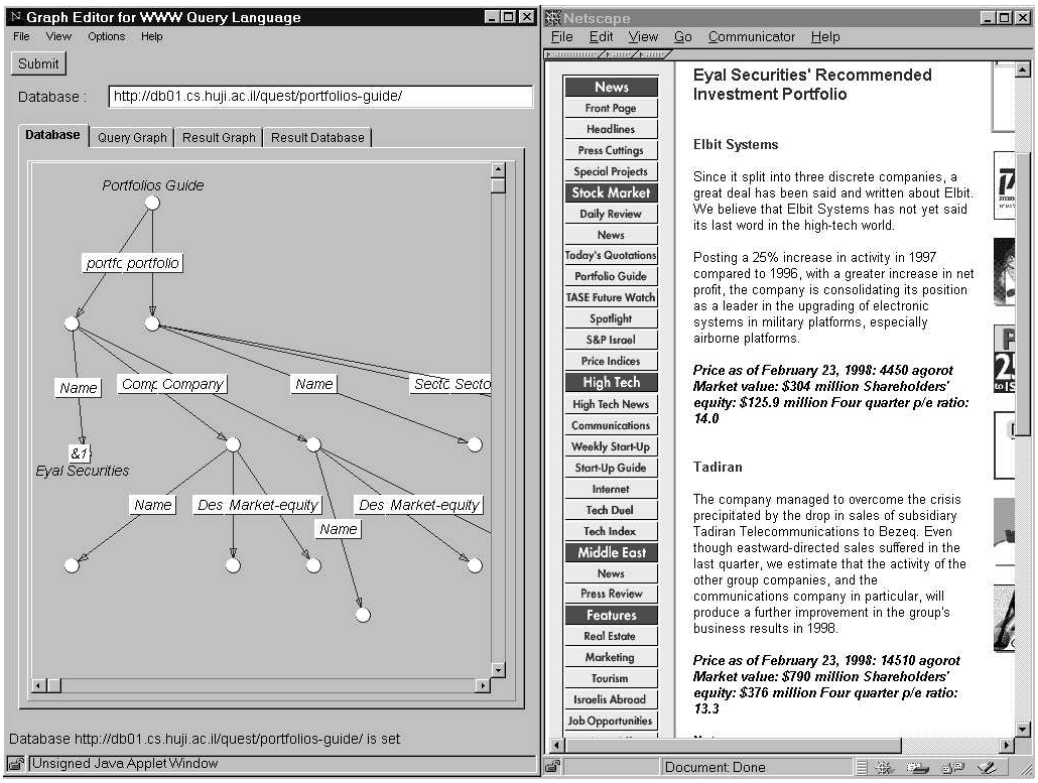

Figure 1: The two aspects of a database

The fact that the underlying structure of a database is a graph influences the way queries in QUEST are formulated and results are displayed. Typically, when a user wants to pose a query, she first explores the graph of the database. It is important to know the graph structure in order to design a meaningful query. A QUEST query essentially consists of two graphs, the *query graph* and the *result graph*, where the query graph determines how the object graph of a database is explored and how data are retrieved, and the result graph describes the object structure produced by the query. Both, the query and the result graph are drawn using a graphical user interface. In Section 4 we discuss the components of a query and how queries are evaluated over a database. In Section 5 we cover the usage of a query graph and the creation of query results when submitting a query.

QUEST is a client-server system. The query is created by the client. and when submitted it is transfered to the server for evaluation over the database. After evaluating a query, the server creates the query result, which is an extension of the given database. That implies that the result is a set of OHTML documents. The location of the result in the database is then sent to the client and that part of the extended database is displayed. Like the display of a database, the display of the result contains both the semantic view and the visual view.

As a query system that facilitates the querying process we just described, QUEST is a combination of tools that allow a user to browse a database, to construct a new or edit an existing query, to evaluate a query over a given database, and to construct the result.

2.2 The Components of a Query

There are three main entities involved in querying, the given database, the query, and the result database, which is an extension of the given database.

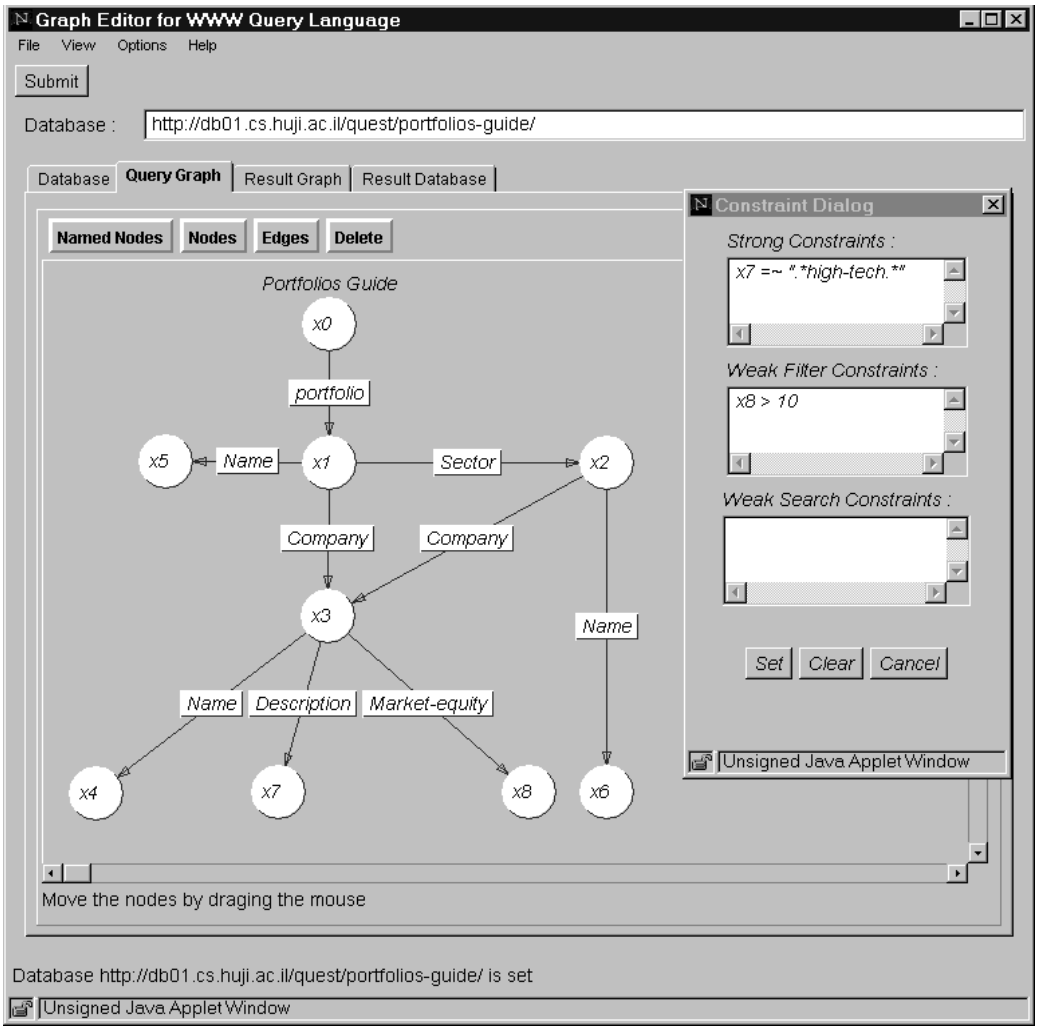

Figure 2: The query graph

A query is evaluated in three phases. The first phase is the *search phase* in which information is extracted from the database. The second phase is the *filtering phase* in which the extracted information is subjected to additional constraints. The third phase is the *construction phase* in which an extension of the existing database is constructed from the extracted information. A particular feature of QUEST is that it allows one to query

incomplete information. The distinction between searching and filtering is necessary in order to retrieve answers that may be incomplete but contain maximal information (see Section 4).

A query consists of two main parts. The first part defines the information to be extracted from the database and the constraints for filtering that information. These parts play similar roles as the FROM and the WHERE clauses in an SQL query. The second part defines how to create the result from the information that was found in the first phase. That role is similar to the SELECT clause in SQL.

The main parts are further divided into the following components:

The Query Graph is a graph that is matched against the database graph during the search phase. Figure 2 shows a the graph of a query that searches the Portfolios Guide database for high-tech companies whose market per equity value is greater than 10.

The Search Constraints are a set of constraints that further specify how to match the query against a database.

The Filter Constraints are applied during the filtering phase and filter the information that was found during the search phase.

The Result Graph is a graph that defines the graph structure of the result database. It defines which new database objects are to be created as the result of a query, and how to connect these objects to each other and to existing database objects. The result graph defines the semantic view of the result database. An example of a result graph is given in Figure 3.

The Templates are textual entities that specify how to construct the HTML part of the result database and how to combine the OEM structure of the result with HTML segments to create OHTML pages. The templates define the visual view of the result database. In the graphical user interface, one can decorate every node in the result graph with a *node template,* as shown in Figure 3. Alternatively, one can create template files (see e.g. Figure 7) that also contain a description of the result graph.

The nodes in a query graph are variables. A matching of a query graph to the database is, therefore, an assignment of database objects to the query variables. When evaluating a query over a database, we compute in the search phase *matchings,* which are assignments that match the query graph. The matchings are then filtered to yield *solutions.* From the solutions a result database is created, which is called the *query answer.*

The result database, similarly to the original database, is a set of OHTML documents. The system displays the graph of the result database as an answer to the query. The query graph, the constraints, the result graph, and the templates in Figure 2 and 3 together form a query. The result of posing this query to the Portfolios Guide database is shown in Figure 8.

3 Data Model

QUEST is a system for querying hypertext documents that have an additional object structure. Since we expect to find in the future such documents on the Web, we cannot

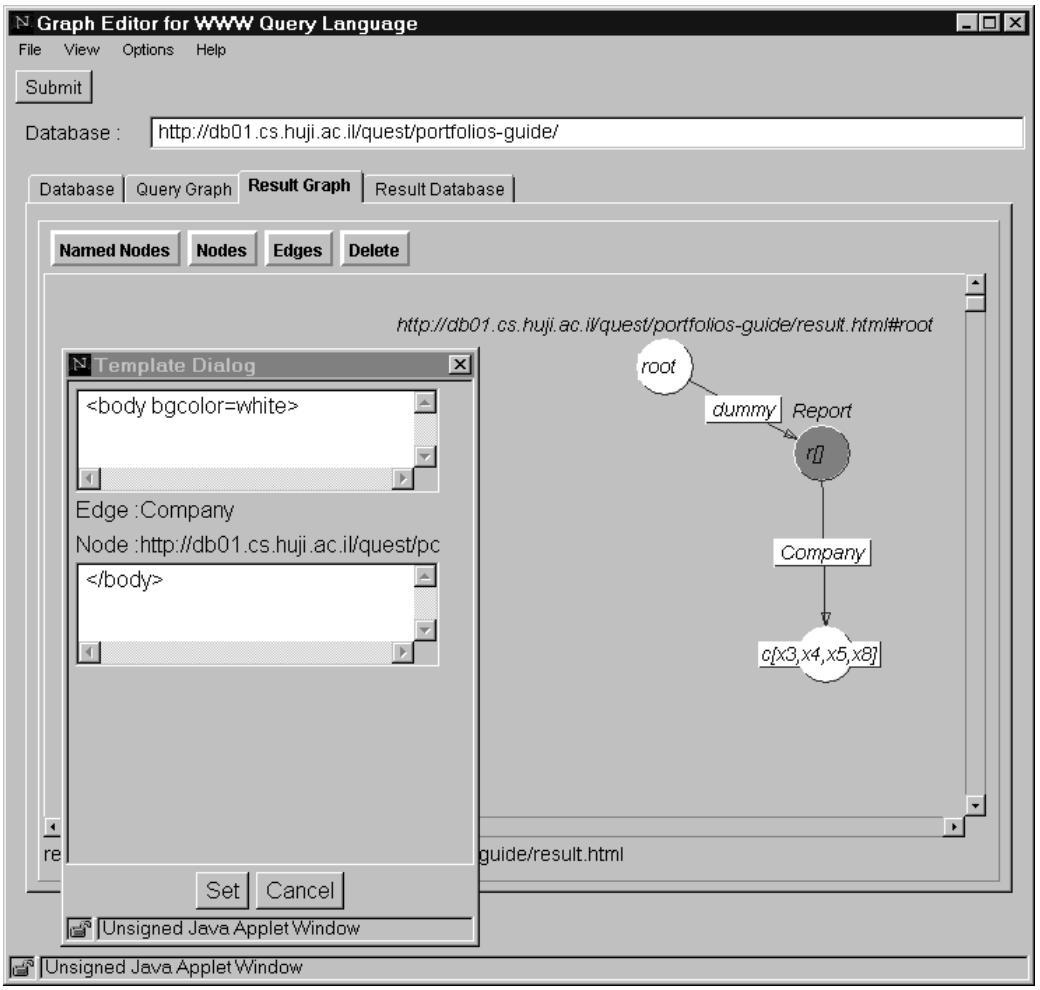

Figure 3: The result graph

assume that the objects comply with a fixed schema like in a classical database. Such structures that show some regularity, but do not follow a strict schema like a standard database are captured by semistructured data models [Abi97].

As the data model for the semantic layer in our system, we chose the semistructured *Object Exchange Model* (OEM) of [AQM+96, AQM+97]. Similarly as in other semistructured models, an instance of OEM is a labeled directed graph. The nodes of the graph are *objects* and each one of them has a unique *object identifier* (oid). Some of the objects have *names* and are called *named objects*. The named objects are the entry points to the database and the labels serve as aliases for them. We require that every object in the database is reachable from some named object through a path in the database graph. An object that is not reachable cannot be accessed and is therefore ignored. *Atomic objects* are objects that have no outgoing edges. They contain a value of an atomic type, such as *integer*, *real*, *string*, *gif*, *html*, *audio*, *java*, etc. Objects that have outgoing edges are

complex objects. Differently from the definition of OEM [AQM+96, AQM+97], we assume that each atomic object has a unique type. If no type is explicitly defined for an atomic object, we assume by default that its type is *html*.

Figure 4 shows as an example a database graph from the Portfolio Scenario. A set of objects in the database that refers to the same concept is considered as a *class*. In the Portfolios Guide database, for example, we have the class of companies that consists of the objects &6, &11 and &15. That lack of a rigid schema is evidenced by a number of irregularities: (1) There are different paths to objects of the same class. For example, Portfolio-Company and Portfolio-Sector-Company are different paths to a company object in the database. (2) When regarding the outgoing edges of an object as its attributes, different objects of the same class may have different attributes. For example the object &11 has no Description attribute and the object &15 has no Market-equity attribute.

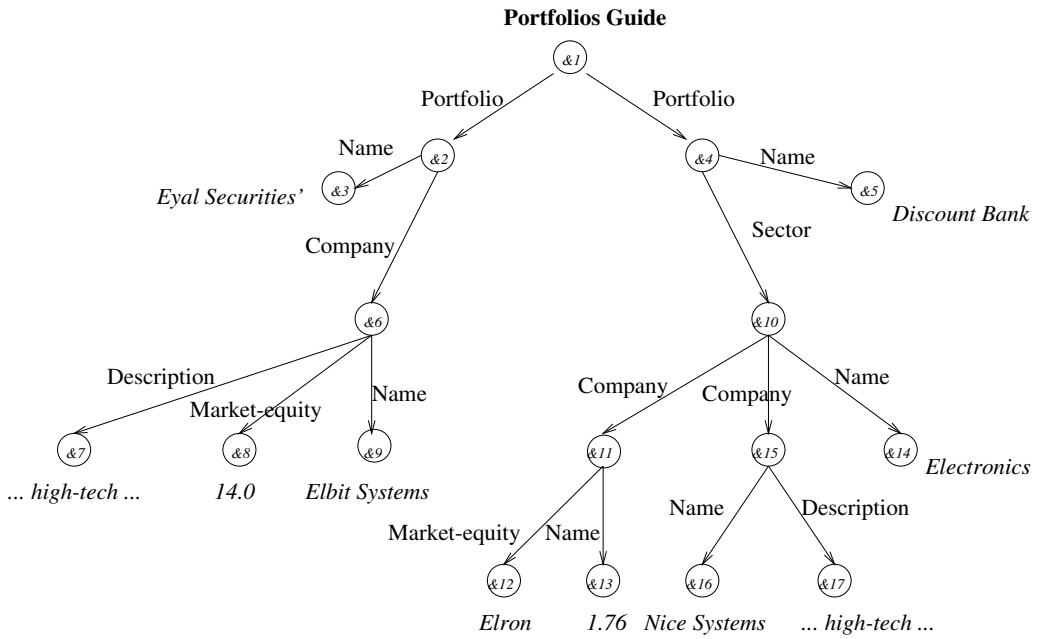

Figure 4: The Portfolios Guide as an OEM database

Since for us databases and queries are graphs with a similar structure, we introduce a common abstraction, which we call skeletons. A *skeleton* is a directed graph with a partial function ν that assigns names to some of the nodes in the graph, such that distinct nodes have distinct names, and such that each node is reachable from a named node. A *database* is a skeleton with two functions, one that maps edges to labels, and one that maps atomic nodes to values.

Using the skeleton as the common abstraction of the basic components leads to a high degree of uniformity of both the conceptual level and the implementation level, and hence is suitable for object-oriented design and programming. In the implementation of QUEST this is reflected by a class hierarchy where databases, query graphs and result graphs are refinements of the class "skeleton."

At present, the information accessible through the Web comes in the form of HTML pages. Ideally we would like to see the WWW as a storage of semistructured data. The main difficulty in querying HTML files is their lack of semantics. Today, the new XML standard adresses this problem, in that it allows for instance to insert "semantic islands" of structured and labeled information into HTML. When we started to develop the QUEST system, XML was not available yet. So, we used OHTML [KMSS98] to add semantics to HTML pages.

OHTML is an extension of HTML that superimposes an OEM object structure on top of HTML pages, by adding to the pages tags that are hidden inside HTML comments. Thus, one can tag an HTML document without affecting the display of the document by a browser. The tags are used to define objects and references among those objects. Objects in OHTML can be spread over more than one HTML page. Thus, a set of OHTML document contains a textual representation of an OEM structure.

Figure 5 shows OHTML code that defines an object Portfolios Guide with two portfolio subobjects, like in the GLOBES database whose graph is shown in Figure 4. The subobjects are in the scope of the tags of the first object.

In order to interpret OHTML documents as OEM graphs, we add object identifiers to the declared objects, in a way that enables us to reference objects and avoids the duplication of identifiers. This is done by associating objects with identifiers that are a combination of a uniform resource locator (URL) and an anchor address. This establishes a fixed connection between the OEM graph of a database and the OHTML documents representing the same database. The usage of URLs also provides entry points to the database, since browsers are capable of reaching a Web page through its URL.

In OHTML one can also define labels as object references. A label indicates an edge from the closest object, in whose scope the label is found, to the next subobject. For example, in our example there is an edge labeled portfolio from the object with oid &1 to the two subobjects with oid's eyal.html#&0 and discount.html#&0. These labels represent the labels attached to the edges in the OEM graph. Finally, OHTML also allows one to define atomic nodes and to associate a type to each value, such as *integer*, *gif*, *java*, etc. The combination of HTML and OEM tags in OHTML documents allows for the two parallel views of a document, the semantic view, through the OEM graph of the database, and the visual view, through the HTML display of the document.

While the QUEST system operates on databases consisting of a set of OHTML documents, the underlying OEM graph is the structure that is most heavily exploited for querying.

4 QUEST Queries and Their Evaluation

In Section 2.2 we introduced the components of a query. We now show how query graph, search constraints, and filter constraints are used to compute the solution of a query over a given database.

A *query graph* is a skeleton whose nodes and edges are variables. Each edge of the query graph is labeled by an *edge label,* which can be a simple string or a regular expression over strings. Figure 2 shows an example of query graph.

```
<HTML>
  <TITLE>
    Index of Portfolios Guide
  </TITLE>

<BODY>
  <!-- (LABEL)Porfolios Guide(/LABEL) -->
  <!-- (OBJ id=&1 type=set name="Portfolios Guide") -->

  <H3>Portfolios Guide:</H3>

  <UL>
    <LI><A HREF="eyal.html">
      <!-- (LABEL) -->portfolio<!-- (/LABEL) --></A>
      <!-- (OBJREF)eyal.html#&0(/OBJREF) -->
    <LI><A HREF="discount.html">
      <!-- (LABEL) -->portfolio<!-- (/LABEL) --></A>
      <!-- (OBJREF)discount.html#&0(/OBJREF) -->
  </UL>
  <!-- (/OBJ) -->

<HR>

  <CENTER>
    This page is a simplified version of the OHTML page
    with portfolio suggestions.
  </CENTER>
</BODY>
</HTML>
```

Figure 5: An OHTML page with tags

4.1 Matchings

The query graph is matched against the database graph, that is node variables in the query graph are mapped to node objects in the database, and edge variables are mapped to edges or paths in the database. We distinguish between *total matchings,* i.e., mappings in which all query variables are bound to a database node or a path, and *partial matchings* in which some variables remain unbound. If a variable is not bound, we alternatively assume that it is mapped to the symbol $\perp$, called *null*.

Semistructured data occur in situations where no rigid schema is imposed on data, and data may be incomplete. This is typically the case on the Web. Thus, allowing only total matchings for a query is too restricted, since this would assume that information appears in certain concrete patterns and is complete. For this reason, QUEST has been designed in such a way that it can handle incomplete information and return incomplete answers.

In order to give an intuitively clear definition of matchings between query graphs and database graphs, we first define total matchings that are similar to the results of queries in other query languages and do not allow for incomplete solutions. Then we define partial

matchings that do allow for incomplete solutions.

The definition of matchings is based on viewing edges in a query as constraints. The labels of edges in the query are constraints in that only certain edges or, more generally, paths match a label, and in addition, the topology of the matching portion of the database must be the same as the topology of the query. More precisely, suppose e is an edge variable in the query graph that links the node variable x to the node variable y and that is labeled with the string or the regular expression l. Moreover, suppose that μ is an *assignment,* i.e., a mapping that maps node variables to database objects and edge variables to paths in the database graph. Let $\pi = \mu(e)$ be the path assigned to e. Then we say that μ *satisfies* the edge constraint of e if

1. π is a path in the database from $\mu(x)$ to $\mu(y)$, i.e., the source of e is mapped to the source of π, and the target of e is mapped to the target of π;

2. the sequence of labels of the edges on the path π satisfies the regular expression l.

In case the edge e is only labeled with the string l, the last condition means that $\mu(e)$ consists of a single edge, and this database edge is labeled with the same string l.

We also view the names of the named nodes in the query graph as constraints. We say that the *name constraint* n is satisfied if the query node with the name n is mapped to a database object with the same name.

We now give a definition of total matchings.

Definition 4.1 (Total Matchings) *A total matching is an assignment of objects and edges of the database to the variables of the query such that the following holds:*

1. *each name constraint is satisfied;*

2. *each edge constraint is satisfied.*

In other words, a total matching requires all variables in a query to be bound and all constraints to be satisfied.

In *partial assignments,* node and edge variables may remain unbound. In other words, a partial assignment maps variables to database entities or to $\perp$. Thus, the requirement that all edge constraints and name constraints be satisfied has to be relaxed. We say that a partial assignment is *defined* for a node (edge) variable if it maps the node (edge) to a non-null object (database edge). Total matchings are defined for all the nodes and the variables of a query graph. A partial assignment of a query graph to a database is defined for some of the query nodes and some of the query edges, and hence, defines a subgraph of the query graph that is constructed from the nodes and edges in the query for which the assignment is defined.

We define partial matchings as follows.

Definition 4.2 (Partial Matchings) *A partial matching is a partial assignment with the following properties:*

1. *for each named node in the query, for which the assignment is defined, the name constraint is satisfied;*

2. *for each query edge, for which the assignment is defined, the edge constraint is satisfied;*

3. *the subgraph which is defined by the assignment is a skeleton.*

Condition 3 in the above definition requires that every node and every edge to which a non-null value is assigned be reachable from one of the named nodes in the graph defined by it.

Consider the query graph in Figure 2. The following table shows the maximal partial matchings produced by the query. For easier comprehension we have replaced the oid's of atomic objects by their values.

No.	x0	x1	x2	x3	x4	x5	x6	x7	x8
1	&1	&2	$\perp$	&6	Elbit Systems	Eyal Security	$\perp$	"..high-tech.."	14.0
2	&1	&4	&10	&11	Elron	Discount Bank	Electronics	"..high-tech.."	1.76
3	&1	&4	&10	&15	Nice Systems	Discount Bank	Electronics	$\perp$	$\perp$

Table 1: Maximal matchings

The partial matchings in Table 1 are maximal since none of the null values in each matching can be replaced by a database object in a way that will satisfy the edge constraint in the query graph.

4.2 Constraints

In addition to the constraints implicit in the query graph, explicit constraints can be specified. Some of them, e.g. constraints on edge labels expressed by regular expressions, further direct the search for matchings. They are to be filled into the slot of weak *search constraints*. The other constraints are used for *filtering* the matchings delivered by the search.

Constraints are Boolean combinations of operators that are applied to expressions. Atomic expressions are either constants (numbers or strings) or variables occurring in the query graph.

A variable in a constraint may represent two alternative values: either the value of atomic node in the database to which it is bound or the object identifier of the node. In our data model, every atomic node has a type. When a variable stands for an identifier, we assume that its type is *string*. When it stands for a value, it refers to the type of that value according to the type of the assigned atomic database node. Operators in a constraint require operands of a certain type and produce a result of a certain type. If, for example, an operator requires numeric arguments, its operands may be numeric constants, variables whose type is numeric, or expressions whose result type is numeric. Operators that require string operands have two versions: one that assumes that variable operands stand for values and one that assumes that variable operands stand for identifiers. The constraint operators of QUEST are given in Figure 6. A *simple constraint* is a constraint of the form $a_1 \theta a_2$, where a_1 and a_2 are atomic expressions and θ is one of the operators in Figure 6.

Operator Group	*Operators*	*Operands Type*	*Result Type*	*Variable operands stand for*
Boolean operators	`&&,||,!`	Boolean	Boolean	—
Arithmetic operators	`+,-,*,/,%`	Numeric	Numeric	—
Relational operators	`<,<=,>,>=`	Numeric	Boolean	—
Equality operators	`==,!=`	Anything	Boolean	Node value
Identifier equality operators	`=o=,!o=`	String	Boolean	Node identifier
Regular expression operators	`=~`	String	Boolean	Node value
Identifier regular exp. operators	`=o~`	String	Boolean	Node identifier

Figure 6: QUEST constraints operators

To take into account partial matchings, we define two ways to evaluate constraints with respect to an assignment, *strong evaluation* and *weak evaluation.* The point is that we still want to evaluate constraints if the assignment is undefined for some query variables.

Consider a simple constraint $a_1 \theta a_2$ and a partial assignment μ for the variables in the query. For the sake of simplicity, we adopt the convention that μ is defined for all constants and maps a constant to itself. Some of the comparison operators given in Figure 6, expect atomic values as arguments. If the the variable in the argument position is bound to a complex database object, the constraint is no satisfied. In the other cases, the constraint can be evaluated in the following two ways:

Strong Evaluation: $a_1 \theta a_2$ is satisfied if μ is defined for both a_1 and a_2 and the values to which a_1 and to a_2 are bound by μ satisfy θ, but the constraint is not satisfied if $\perp$ is assigned to one of the arguments;

Weak Evaluation: $a_1 \theta a_2$ is satisfied if one of the following is true

1. μ is not defined for a_1; or
2. μ is not defined for a_2; or
3. μ satisfies $a_1 \theta a_2$ under strong evaluation.

If a constraint is satisfied under strong (weak) evaluation, we say that it is *strongly satisfied* (*weakly satisfied*). Satisfaction of Boolean combinations of simple constraints can be defined in the obvious way. Each constraint in a query is entered either as a weak or a strong constraint. Weak constraint have to be weakly satisfied and strong constraints have to be strongly satisfied.

4.3 Search Constraints and Maximal Matchings

During the search phase of the query evaluation, QUEST uses search constraints in addition to the edge constraints. We extend the definition of a matching by requiring not only that it satisfies the edge constraints of the query graph, but also that it satisfies the (weak) search constraints. Although in QUEST it is allowed to enter arbitrary constraints as search constraints, it appears to be reasonable to only use constraints on edge labels as search constraints.

In the search phase the information required for the next phases of the query is collected. By the definition of matchings as partial assignments satisfying weak constraints,

it is possible that a matching satisfies constraints by not binding the variables involved in it. This is unwanted if it is possible to find bindings for some variables such that the search constraints are still satisfied. For this reason, we are not interested in arbitrary matchings, but in maximal matchings, i.e., matchings that cannot be extended over the given database without violation some constraint. Intuitively, maximal matchings contain maximal information, which is the best we can expect when information in the database may be incomplete.

Formally, if μ and μ' are two matchings, we say that μ *subsumes* μ' if for every variable x for which μ' is defined, μ maps x to the same database object as μ', and similarly for the edges. In other words, it means that μ is the same as μ' except that it may be defined for some entities in the query-graph for which μ' is not defined. We say that a matching is *maximal* if it is not subsumed by any other matching. Given a query and a database, the result of the search phase, when evaluating the query over a database, is the set of maximal matchings. Thus, the search constraints, namely, the name constraints, the edge constraints, and the weak search constraints, attract assignments as large as possible. This generalizes the notion of full disjunction of [RU96, GL94].

4.4 Filter Constraints

During the second phase of the query evaluation the maximal matchings from the search phase are filtered. There are two sets of filter constraints, namely strong constraints and weak filter constraints, which are entered through the query editor in the user interface. These constraints are used to filter the maximal matchings by removing the matchings that do not satisfy the constraints. The filtered maximal-matchings, are called *solutions*. The solutions are maximal matchings that strongly satisfy the strong constraints and weakly satisfy the weak filter constraints.

The distinction between strong constraints and weak filter constraints is due to the presence of partial assignments. The basic distinction between them is that strong constraints are only satisfied if variables are bound, and thus require certain information to be present. Weak constraints do not require information to be present: if information is available in a matching, and it does not comply with the constraint, then the matching is dismissed; however, if the information is not available, then we give the matching the benefit of the doubt and keep it. For instance, if we pose a query where we ask for companies that have a market value of at least 500 Mio Dollar, then we will not receive in the result companies for which it is known that their market value is below this figure, but we may receive companies for which the market value is unknown.

For strong constraints it does not make a semantic difference whether they are applied during the search or the filtering phase. Thus, it is even advisable to use them as early as possible during the search in order to prune dead ends of the search space.

For weak constraints, however, the difference is important. Search constraints do not dismiss assignments. If a search constraint is added, it introduces new variables and new possibilities to satisfy it. Intuitively, search constraints are like baits that lure assignments as large as possible. Search constraints are *monotonic*, since the addition of search constraints may change the set of solutions, but does not reduce its size.

Filter constraints, by their very nature however, eliminate assignments. In fact, one of the reasons why we want maximal matchings as results of the search phase is to give

as much power as possible to the weak filter constraints. Weak filter constraints are *not monotonic* in the sense that adding a filter constraint can reduce the size of the solutions set.

Weak constraints have a different effect as search constraints than as filter constraints. This is not the case with strong constraints. The reason for this is that for strong constraints to be satisfied the variables in the constraints must be defined by the satisfying assignment. Strong constraints can filter out a matching and cannot change the matching to assign $\perp$ to one of the query variables. For this reason, strong constraints are all considered filter constraints in QUEST.

The set of solutions, as the result of the filtering phase, is used for the creation of the result. We discuss this topic in the next section.

5 Constructing Results

The result of a query in QUEST is a set of OHTML pages that extend the database over which the query is posed. Since the search and filtering phase produce sets of partial matchings, we need a mechanism to define and create OHTML pages out of partial assignments. When creating OHTML pages, we must take into account the two aspects of OHTML, the semantic view and the visual view. Thus, the answer returned by a query must include an OEM graph, which is embedded in HTML pages by means of OHTML tags.

In principle there are two ways to combine the two aspects, depending on which aspect we give priority. The first approach is to produce an OEM graph and and decorate its nodes and edges with HTML. The second is to create HTML pages, into which objects and edges of the OEM structure are implanted. Both approaches lead to OHTML pages. In QUEST, there are mechanisms realizing each of the two approaches. We will discuss only the first one, which gives priority to the semantic structure when constructing the answer.

We are using two formalisms in the creation of the result. The first is a *result graph,* which determines the OEM structure of the answer. The second is a set of OHTML templates that are used to decorate the OEM structure with HTML tags.

5.1 Result Graph

When creating an OEM structure out of the solutions to the query, two main tasks have to be fulfilled. The first is to create new objects, and the second is to create links among these new objects and links from the new objects to other database objects.

These two functions are accomplished with a *result graph,* which is a skeleton with labeled edges and nodes that are flat terms. A *flat term* is either a variable or a term of the form $f[x_1, \ldots, x_n]$, where f is a Skolem function and $x_1, \ldots, x_n$ are node variables occurring in the query graph. The idea is that new objects are generated by applying Skolem functions to existing objects.

The solution of a query are (partial) assignments of database objects to node variables of the query. Suppose $f[x_1, \ldots, x_n]$ a flat term in the result graph. If μ is a solution, then $[\mu(x_1), \ldots, \mu(x_n)]$ is a tuple of database objects with oid's, say, $o_1, \ldots, o_n$. For each such tuple arising from a solution, we create a new object with oid $f[o_1, \ldots, o_n]$.

There are different ways to handle tuples with nulls. One is, not to create new objects from tuples that include null values. We have found this approach too restrictive. Instead,

we treat each null value as a unique database object. In this way, we take into account that partial solutions may not bind all variables, and yet are able to utilize the partial information to create new objects.

Summarizing, new objects are generated in three steps.

1. For each solution, create tuples of objects according to the tuples of variables occurring in the flat terms of the result graph;

2. Apply the Skolem function of each term to the oid's of the objects in the tuples built from the solutions;

3. Remove duplicates, that is, remove oid's that are created by applying the same Skolem-function symbol to the same tuple of database objects and null values.

Because of Step 3, two different solutions and a term may create either two different result objects or only one result object. Only a single object is generated when the same Skolem-function is used and the same entities are assigned to the variables of the term.

Once the result objects are generated, links are introduced between them according to the linking in the result graph. Let t_1 and t_2 be two terms in the result graph that are connected by an edge, labeled with l, from t_1 to t_2. Then for every solution μ we proceed as follows. Let o_1 and o_2 be the two new objects that result from applying the Skolem functions in the terms t_1 and t_2 to the objects assigned by σ to the variables of the terms. Then o_1 and o_2 are connected in the result database by an edge from o_1 to o_2 labeled with l.

A special case of a flat term is a variable. In this case, no new objects are created, since no Skolem function is applied to oid's of previous objects. Such terms define connections of result objects to objects of the database over which the query is evaluated. We only process bindings of atomic database nodes to such terms because otherwise existing objects in the database would be modified as a result of the query.

QUEST requires that the result graphs are acyclic. In addition, the list of variables in each node must include all the variables of its parent. This requirement is due to the following reason.

Assume, for example, that $f[x]$ is a parent term and $g[y]$ is a child term. Let $\mu_1 = \{x/o_1, y/o_e\}$ and $\mu_2 = \{x/o_2, y/o_e\}$ be two solutions. Then $f[o_1]$, $f[o_2]$ and $g[o_e]$ are the newly generated objects. When OHTML pages are created that contain these objects, each one of the objects $f[o_1]$ and $f[o_2]$ must encapsulate in its OHTML representation the representation of the object $g[o_e]$. Duplicating an object is not a good solution to the problem of more than one object encapsulating another object, since this would actually be the same as creating several objects. Using references cannot solve this problem either, since when one object encapsulates another object, it means that the HTML text inside the inner object must be used in the display of the encapsulating object. References of OHTML and hyper-links of HTML cannot help, because HTML does not allow documents that are distributed over more than one file.

Another requirement on result graphs is that the variables that are encapsulated by a node (a term) must be included in that term. The variables that a node encapsulates are those variables that appear in that node as atomic values or as part of an atomic value. Such a variable can appear directly in a node if the node is an atomic node or indirectly if the node encapsulates another node that encapsulates the variable. If a term in the

result graph violates one of these two requirements on is argument variables, then QUEST automatically adds the missing variables.

Since only one object can exist in the uppermost level of each OHTML page, a dummy root object is produced for each OHTML page in the result database. This root object encapsulates the objects in the uppermost level of the page and the HTML text that appears before and after that object. The root object is an instantiation of a result graph term whose Skolem-function symbol is *root*. The *root* term defines the partition of the result database into pages, where each new object created by applying the Skolem function *root* is a new page.

5.2 OHTML Templates

QUEST uses OHTML templates to create the HTML that embeds the OEM structure of the result. In the query interface, one can add to a node a preceding and a succeeding HTML text. Actually, the text is HTML with references to leaf node variables of the query. The text segments, after instantiating the variables with the atomic values to which they have been bound, surround the result objects created from this node.

For each node in the result graph the system produces a *node template* from the node term and the surrounding HTML text. The node template defines the visual display of the objects that are created from that node. Each node template consists of parameterized HTML text and OHTML tags that define the objects to be created from the template, as well as the edges to encapsulated objects and to objects being referenced. Hence, each node template is essentially an OHTML document with variables from the query. The node templates are combined into a *query template*, according to the result graph.

When new OHTML pages are generated from a query template, the template is evaluated over the solutions and each variable in the template is bound to database objects and/or to $\perp$. If the database objects are atomic then the variable is replaced by the values of the objects. If the database objects are complex then a reference to the objects is created.

Figure 7 shows a query template. In the template there are two Skolem functions, `report` and `company`. The variables `x4`, `x5`, `x8` are embedded into the text.

6 System Architecture

We now describe the implementation of the QUEST system. We developed all components of the system from scratch rather than to use an existing system for managing semistructured data as a backbone. The reason is that until now such systems exist only as research prototypes. Moreover, there was no system that supports querying semantically tagged hypertexts or executes queries that produce incomplete answers. The components of QUEST are a parser of OHTML documents, a query processor, a server, a client, a graphical user interface, wrappers, and an OHTML editor. In the following, we explain one by one the functionality and the implementation of the components.

Basic Data Structures. The implementation of QUEST relies on a hierarchy of Java classes. This hierarchy follows the definitions of the data-model and the elements of the query-language given in the previous sections. There are several *basic classes* like *Skeleton*, *DataBase*, *QueryGraph*, *ResultGraph*, *Template*, *Constraint*, and *Assignment* that set up

```
<HTML>

<!-- (LABEL)Report(/LABEL) -->
<!-- (OBJ id=report[] type=set name="Report") -->
    <BODY bgcolor=white>
    <!-- (LABEL)Company(/LABEL) -->
    <!-- (OBJ id=company[x3] type=set) -->
        <HR>
        <P>
        <H3><CENTER>$x4$</CENTER></H3>
        <P>
        Recommended by: <I>$x5$</I>
        <P>
        Market per equity: $x8$
    <!-- (/OBJ) -->

    </BODY>
<!-- (/OBJ) -->

</HTML>
```

Figure 7: A textual representation of a template

the programming framework. These classes are the core of the QUEST system. In order to facilitate the data exchange between different modules of the system and over the network, instances of these classes can be serialized. That means, instances can easily be converted into character strings and reconstructed from such a string representation.

Parser. For our application scenario, we assume that databases consist of OHTML documents that reside on the Web. Therefore, they have to be parsed before they can be further processed. In the current implementation, OHTML documents are stored on disk. They are loaded into corresponding data structures in main memory by the parser.

Also, query graphs and result graphs are serialized when transferred between different modules of the system (either in a single program or over the network), and have to be parsed in order to be reconstructed. The performance of the parser is thus a crucial issue.

Query Processor. The query processor receives as an input a database, a query graph, a result graph and constraints. The output of the query processor is a result database. A result database is "published" by the query processor as a collection of OHTML documents.

The query processor evaluates queries in three steps:

1. computing maximal matchings and solutions;
2. constructing the result database;
3. generating OHTML documents corresponding from the templates.

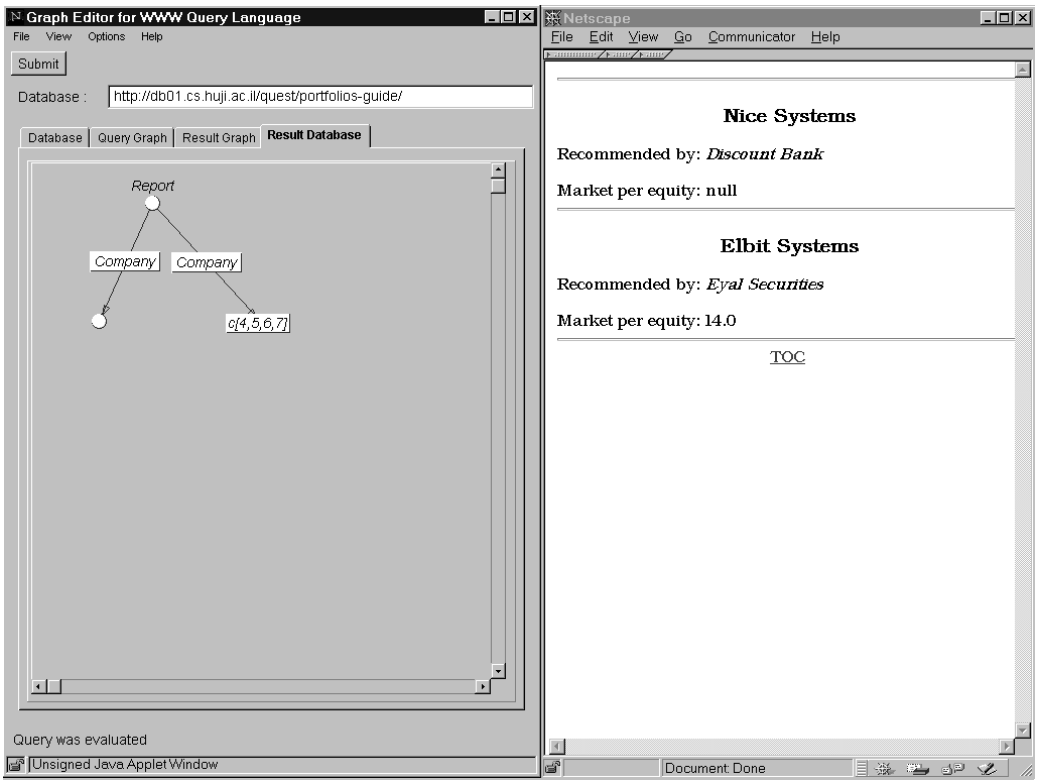

Figure 8: The result database produced when evaluating our example query (query graph in Figure 2 and result graph in Figure 3) over the database in Figure 1

If the only constraints passed to the query processor are weak search constraints, then the evaluation takes only time that is polynomial in the size of the input and output. Evaluating a query that involves strong constraints and/or weak filtering constraints has NP-complete combined complexity, but the data complexity is still PTIME. (The foundations of query processing in QUEST are described in [KNS99].)

Client and Server. The main thread of the QUEST server is to wait for client connections. When a client requests a connection, a new thread is spawned on the server and the connection is established. Then, to execute a query, the client sends to the server the query components: the URL of the database, the query graph, the result graph, and the constraints.

The server loads the database into the main memory and then runs the query processor to evaluate the query. The evaluation of the query leads to the construction of a result database. Then the server either sends the result to the client as is, or publishes it on the secondary storage device as a collection of OHTML documents, and only the URL of the published database is sent to the client.

The client is a Java application that communicates with the server according to a querying protocol. The protocol permits the client to do the following:

- send a database, a query graph, a result graph, or constraints in their serialized form or as URLs;
- request to execute the query;
- request and receive the result database in its serialized form or as a URL.

Figure 9 shows the flow of data structures during query evaluation.

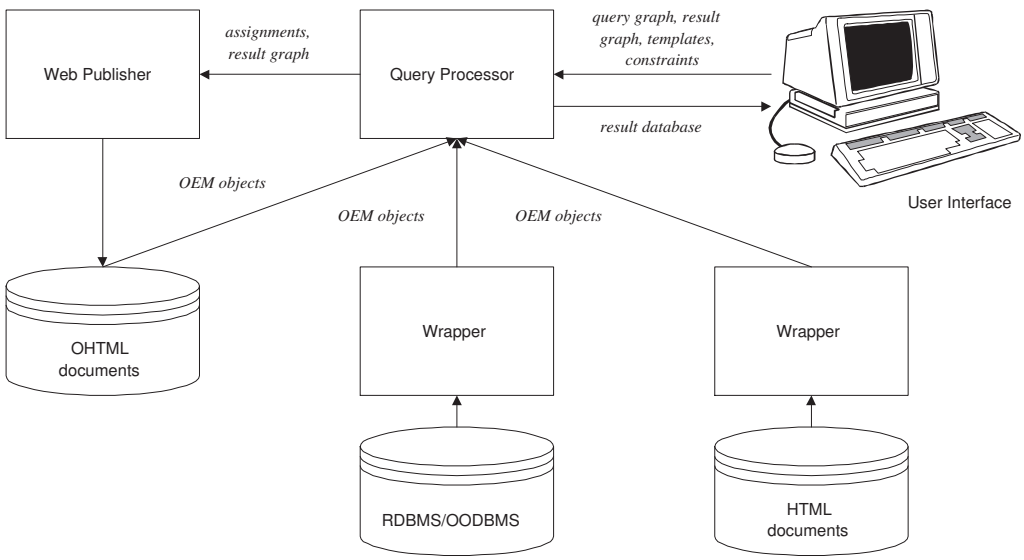

Figure 9: The data flow of query processing.

The server and client implementation rely on the same framework of Java classes. Thus, in principle, a query might be evaluated either in by client or by the server. In our implementation, a client is an applet running in a Web browser. We do not expect the user's workstation to have enough computing power and main memory for query processing. Moreover, an ordinary Java applet is restricted in its ability to write on local storage when it is running in a Web browser.

This architecture brings about a number of advantages. The user need not install a client application on her workstation because the application is a Java applet. The client utilizes the computing power and the disk space of the server for processing its queries. In particular, the client can utilize the server's cache. This speeds up the answering of queries. A drawback of processing queries by the server is that shipping result databases to the client may be costly.

Graphical User Interface. The client, which is a Java applet, provides a user with a GUI for creating new queries, editing queries, and for viewing a database. When it is loaded into the Web browser, the applet creates a frame outside of the browser's window. The frame has four panels: one to view databases, one for editing query graphs, another one for editing result graphs and templates, and finally one for browsing the result database.

7 Conclusions

We have developed the QUEST system and experimented with it: we have downloaded pages from several Web sites, among them the GLOBES site [GLO], annotated them with OHTML tags, and queried them. To conclude, we summarize the experience we have gained from this work and suggestions for further research.

Partial answers. QUEST retrieves partial answers in cases where information is incomplete. We found cases in the GLOBES scenario where providing partial answers is a useful feature. Consider for example the query: retrieve all the companies for which the market value per equity is more then 10 and the word "high-tech" appears in their description. Let the first condition be a weak filtering constraint and let the second condition be a strong constraint.

In the other words, if a company object has an outgoing edge labeled with "Market equity" we will check that the corresponding value is greater then 10; if a company object does not have such an edge we still would like to have the company in the answer. The reason is that we want to get all the answers that satisfy the conditions we provide. Moreover, we want to receive those answers that may satisfy these conditions but where the database does not have enough information to realize it. In this case, a user could check such answers himself and make a proper decision.

The problem we encountered regarding partial answers arose when constructing OHTML documents. Suppose we defined a template for an answer. A template has variables that should be substituted with values of objects coming from the answer. However, if the variables are unbound, should a null value be represented in the answer, or should those parts of the answer that contain null be dismissed? It seems that for a satisfying solution it should be possible to specify in the templates which possibility to choose.

Graphical Query Language. We found that while a graphical query language clarifies complex queries, it is less convenient for simple queries. It is much easier for a user to write a path expression rather than draw a query graph that corresponds to such path expression. On the other hand, there are cases when a query involves several path expressions, and the same variable may appear in more than one path expression. In these cases a graphical query language is preferable.

Simple User Interface. In QUEST a user sees the semantic view of a database in the form of a graph. This representation is helpful for a simple database that describes, for instance, a single home page. Complex databases, however, may have a vast number of objects and edges. A graph representation of such databases becomes a tight net of overlapping nodes and edges if we want to show all the connections, and graph browser that allows to focus on certain parts of the database while creating condensed abstractions of others may be helpful.

Our main effort now is to alter the system to support XML. We believe that the principles used in QUEST are so general and important that they can be carried over to a query language for XML documents.

References

[Abi97] S. Abiteboul. Querying semi-structured data. In Ph. Kolaitis F.N. Afrati, editor, *International Conference on Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18, Delphi (Greece), January 1997. Springer-Verlag.

[AM98] G.O. Arocena and A.O. Mendelzon. WebOQL: Restructuring documents, databases, and webs. In *Proc. 14th International Conference on Data Engineering*, pages 24–33, Orlando (Florida, USA), February 1998. IEEE Computer Society.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To weave the web. In *Proc. 23nd International Conference on Very Large Data Bases*, pages 206–215, Athens (Greece), August 1997. Morgan Kaufmann Publishers.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. Technical report, The Stanford University Database Group, 1996.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AV97] S. Abiteboul and V. Vianu. Queries and computation on the web. In Ph. Kolaitis F.N. Afrati, editor, *International Conference on Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 122–133, Delphi (Greece), January 1997. Springer-Verlag.

[BDHS96] P. Buneman, S.B. Davidson, G.G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal (Canada), June 1996.

[Bun97] P. Buneman. Semistructured data. In *Proc. 16th Symposium on Principles of Database Systems*, pages 117–121, Tucson (Arizona, USA), May 1997. ACM Press.

[Con98] World Wide Web Consortium. Extensible markup language (XML) 1.0. http://www.w3.org/TR/REC-xml, 1998.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Applications of XML-QL, a query language for XML. http://www.w3.org/TR/NOTE-xml-ql, 1998.

[FFK+98] M.F. Fernandez, D. Florescu, J. Kang, A.Y. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pages 414–425, Seattle (Washington, USA), June 1998. ACM Press.

[FPS97] M.F. Fernandez, L. Popa, and D. Suciu. A structure-based approach to querying semi-structured data. In *6th International Workshop on Database Programming Languages*, Estes Park (Colorado, USA), August 1997. Springer-Verlag.

[GL94] C.A. Galindo-Legaria. Outerjoins as disjunctions. In *Proc. 1994 ACM SIGMOD International Conference on Management of Data*, pages 348–358, Minneapolis (Minnesota, USA), May 1994. ACM Press.

[GLO] GLOBES. http://www.globes.co.il.

[KMSS98] Y. Kogan, D. Michaeli, Y. Sagiv, and O. Shmueli. Utilizing the multiple facets of WWW contents. *Data and Knowledge Engineering*, 28(3):255–275, 1998.

[KNS99] Y. Kanza, W. Nutt, and Y. Sagiv. Incomplete answers for queries over semistructured data. In *Proc. of the Workshop On Query Processing for Semistructured Data and Non-standard Data Formats*, Jerusalem (Israel), January 1999.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the world-wide web. In *Proc. 21st International Conference on Very Large Data Bases*, pages 54–65. Morgan Kaufmann Publishers, August 1995.

[KS97] D. Konopnicki and O. Shmueli. W3QS—A system for WWW querying. In *Proc. 13th International Conference on Data Engineering*, page 586, Binghamton (United Kingdom), April 1997. IEEE Computer Society.

[LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. A declarative language for querying and restructuring the web. In *Proc. 6th International Workshop on Research Issues on Data Engineering - Interoperability of Nontraditional Database Systems*, pages 12–21, New Orleans (Louisiana, USA), February 1996. IEEE Computer Society.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 3(26):54–66, 1997.

[MAM+98] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. The Araneus web-base management system. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pages 544–546, Seattle (Washington, USA), June 1998. ACM Press.

[MM97] A.O. Mendelzon and T. Milo. Formal models of web queries. In *Proc. 16th Symposium on Principles of Database Systems*, pages 134–143, Tucson (Arizona, USA), May 1997. ACM Press.

[MMM97] A.O. Mendelzon, G.A. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In P.S.Yu and A.L.P. Chen, editors, *Proc. 11th International Conference on Data Engineering*, pages 251–260, Taipei, March 1995. IEEE Computer Society.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In Tok Wang Ling, Alberto O. Mendelzon, and Laurent Vieille, editors, *Proc. 4th International Conference on Deductive and Object-Oriented Databases*, volume 1013 of *Lecture Notes in Computer Science*, pages 319–344, Singapore, December 1995. Springer-Verlag.

[QWG+96] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J.D. Ullman, and J.L. Wiener. Lore: A lightweight object repository for semistructured data. In *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, page 549, Montreal (Canada), June 1996.

[RU96] A. Rajaraman and J.D. Ullman. Integrating information by outerjoins and full disjunctions. In *Proc. 15th Symposium on Principles of Database Systems*, pages 238–248, Montreal (Canada), June 1996. ACM Press.