Querying Semantically Tagged Documents on the World-Wide Web*

Ziv Bar-Yossef[1], Yaron Kanza[2], Yakov Kogan[2], Werner Nutt[3], and Yehoshua Sagiv[2]

[1] Department of Electrical Engineering and Computer Science,
U.C. Berkeley, Berkeley CA 94720
zivi@cs.berkeley.edu
[2] Dept. of Computer Science
Hebrew University, Jerusalem, Israel
{yarok, yakov, sagiv}@cs.huji.ac.il
[3] German Research Center for Artificial Intelligence
(DFKI GmbH), 66123 Saarbrücken, Germany
nutt@dfki.de

Abstract. QUEST is a system for Querying Semantically Tagged documents on the World-Wide Web. The advent of new markup languages, such as XML, facilitates authoring of Web documents that contain not just HTML tags for instructing a browser how to view a document, but also contain objects that represent the semantic structure of the document. When such documents become widely available, more powerful methods to access and query information on the Web will be possible. The QUEST system was designed and implemented for querying and manipulating documents written in the markup language OHTML. OHTML combines HTML and objects of the OEM data model. QUEST has several new features. First, QUEST can be used to query a combination of hypertext and object structures. Second, The results of queries are OHTML pages and thus of the same type as the data being queried. Third, QUEST implements a new approach for querying semistructured data that produces meaningful answers even when the input data is incomplete, i.e., when some variables of the query cannot be bound to database values. Finally, the experience of developing and using QUEST for 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 its components.

1 Introduction

The enormous growth in the usage of the World-Wide Web as an information source suggests that the Web will evolve into a platform with more database tools. One major obstacle in the evolution of the Web into one giant database

* This research was supported by Grants 8528-1-95 and 9481-1-98 of the Israeli Ministry of Science.

is the lack of semantics in HTML pages, which makes it difficult to distinguish between different pieces of information in Web pages. To overcome this problem, we have used OHTML [KMSS98], which enriches HTML with semantic tags that define an object structure. In OHTML, the semantic tags are hidden as HTML comments and, hence, their existence is transparent to HTML browsers. The object structure imposed by OHTML on the data of the Web is in the style of the *object exchange model* (OEM) that was proposed for *semistructured data* (see [Abi97,Bun97,PGMW95]). The development of OHTML started before the advent of XML. However, we believe that the techniques developed in QUEST for OHTML are also applicable to documents formulated in XML.

This paper describes QUEST, a system for QUErying Semantically Tagged documents on the Web. The QUEST system was developed and implemented at the Hebrew University of Jerusalem. 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 both to the structure of objects and to 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; and (3) answers of queries may become extensions of the initial database.

Motivated by the growing use of the World-Wide Web as a heterogeneous information source, many systems for querying the Web were developed. Among them W3QL [KS97,KS95], WebLog [LSS96], WebSQL [MM97,MMM97], and WebOQL [AM98]. As a further development, systems were designed for Web site management, e.g., Strudel [FFK+98] and Araneus [AMM97,MAM+98]. In comparison to these systems, QUEST has the advantages of (1) using semantic tags for more accurate querying, and (2) dealing robustly with incomplete information. The second novelty is also an advantage when comparing QUEST to query languages, such as Lorel [AQM+97,MAG+97,QWG+96] and UnQL [BDHS96], that were developed for querying semistructured data in general, and not just in the context of the Web.

Today, as XML [Con98] is becoming a standard, querying semantically tagged documents is an important research issue. Languages such as XML-QL [DFF+98] and XQL of Microsoft have been proposed recently. These languages are at an early stage of implementation, and we believe that our experience can be useful for the design and implementation of query languages for XML.

2 Data Model

QUEST is a system for querying hypertext documents that also embed some object structures. Due to the diversity of the Web, we cannot expect that documents with object structures will conform to a fixed schema, as in a classical database. Object structures that show some regularity, but do not follow a strict explicit schema, are captured by semistructured data models [Abi97].

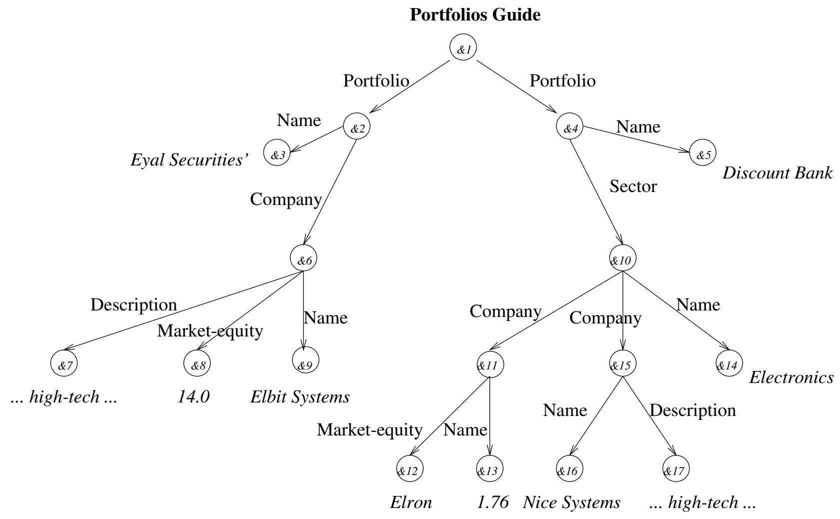

Fig. 1. A Portfolios Guide as an OEM database

We chose the *Object Exchange Model* (OEM) of [PGMW95] as the data model for the semantic layer in our system. OEM is a semistructured data model that represents databases as labeled directed graphs. Each node of a graph is an *object* with a unique *object identifier* (oid). Some objects have *names* and are called *named objects*. The named objects are the entry points to the database, and the names serve as aliases to those objects. Each object in the database must be 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. An *atomic object* is an object that has no outgoing edges. It contains a value of an atomic type, such as *integer*, *real*, *string*, *gif*, *html*, *audio*, *java*, etc. Objects that have outgoing edges are *complex objects.* Figure 1 shows, as an example, a database graph of a portfolio guide.

Since databases and queries are graphs of a similar structure, we introduce a common abstraction, called 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 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 skeletons as a common abstraction of the basic components provides a high degree of uniformity, at both the conceptual level and the implementation level. In the implementation of QUEST, this is reflected in the class hierarchy, where databases, query graphs and result graphs are all extensions of the "skeleton" class.

One purpose of the recently proposed markup language XML is to express the semantics of certain parts of a document by means of markup tags. We

```
<HTML>
    <TITLE> Index of Portfolios Guide </TITLE>
    <BODY>
    <!-- (LABEL)Portfolios 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>
```

Fig. 2. An OHTML page with tags

started our project before the advent of XML and created the tagging language OHTML [KMSS98]. OHTML is an extension of HTML that superimposes an OEM object structure on top of an HTML page, by adding semantic 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. Thus, a set of OHTML document contains a textual representation of an OEM database.

Figure 2 shows OHTML code that defines a Portfolios Guide object with two portfolio subobjects, similarly to the Portfolios Guide database of Figure 1. Note that the tags of a subobject are nested inside the tags of the parent object.

In order to interpret OHTML documents as OEM graphs, we add object identifiers (oid's) to the objects defined by OHTML tags. Thus, objects can be referenced, and each object has a unique oid. The oid of an object is a combination of a uniform resource locator (URL) and the offset of the object from the beginning of the page. URLs also provide entry points to the database, since browsers are capable of reaching a Web page through its URL.

In OHTML one can also use references to object id's. In Figure 2, for example, there are references to two subobjects having the oid's eyal.html#&0 and discount.html#&0. These two subobjects are children of the object having the oid &1 and are connected to their parent via edges labeled with portfolio. Since the two subobjects are not located physically immediately after the labeled edges leading into them, oid references are used. Finally, OHTML also allows one to declare the type of each atomic node, e.g., *integer*, *gif*, *java*, etc.

3 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 consists of a group of companies recommended for investment. There is a general style in the design of the HTML documents that describe portfolios. However, each one of the portfolios has its own 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, tagged these pages with OHTML tags, and queried them in QUEST.

3.1 Overview of the Querying Process

We consider a set of OHTML documents as a *database*. A database has two aspects. The *visual view* is the visualization of the HTML part of the documents as shown by a browser. The *semantic view* is the second aspect, and it is the graph structure of the set of objects contained in the database. Figure 3 shows how QUEST provides simultaneously the two views of the GLOBES database. The existence of two parallel views for each OHTML document is due to the combination of HTML tags and OEM structures in the documents.

The display of the database graph in the visual view familiarize the user with the structure of the database and thus, gives her the ability to design meaningful queries. A QUEST query essentially consists of two graphs, the *query graph* and the *result graph*. The query graph determines how the object graph of a database is explored and how data are retrieved. 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 query graphs and their evaluation over a database. In Section 5 we cover the usage of a result graph for the creation of result pages when submitting a query.

QUEST is a client-server system. The query is created in the client part, and when submitted, it is transferred 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 as a URL and that part of the extended database is displayed. The display of the result

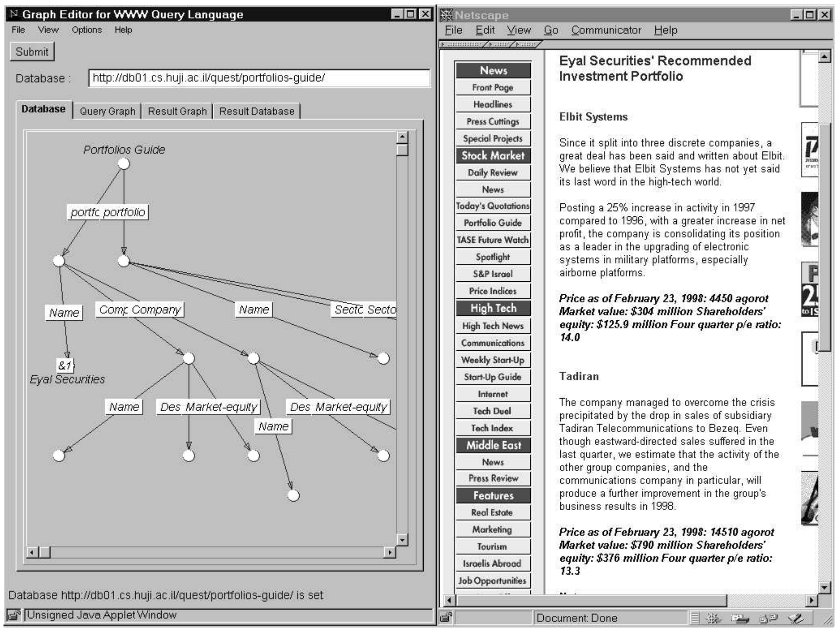

Fig. 3. The two aspects of a database—semantic (left) and visual (right)

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.

A query in QUEST is evaluated in three phases. The first phase is the *search phase* in which information is extracted from the database. In this phase the query graph is matched to the database graph in search for similarity of patterns. We thus call *matchings* to the result of the search phase. The second phase is the *filtering phase* in which the extracted information is subjected to additional constraints. We call *solutions* to the matchings that remain after the filtering phase. The third phase is the *construction phase* in which an extension of the existing database is constructed from the extracted information. The result of the construction phase is called the *query answer*.

QUEST allows incomplete answers when querying incomplete information. Yet, we require an answer to have a maximal information content. The distinction between searching and filtering is necessary in order to apply some constraints only to matchings that contain maximal information (see Section 4).

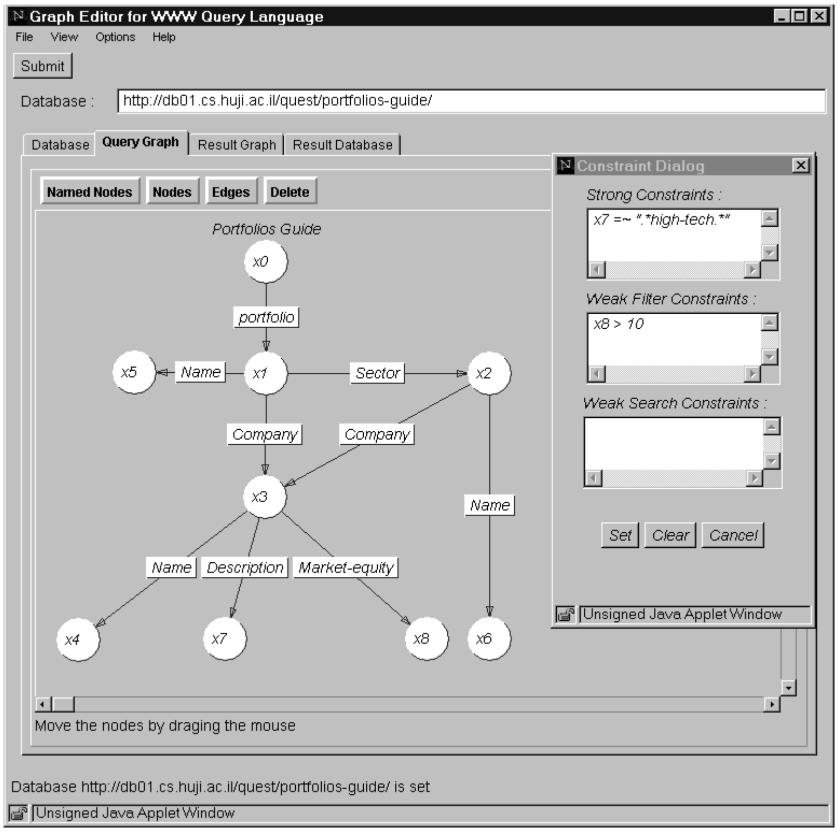

Fig. 4. The query graph

3.2 The Components of a Query

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. This part plays similar roles as the FROM and the WHERE clauses in an SQL query. The evaluation of this part is the search phase and the filtering phase. 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 and is used for the construction phase.

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 4 shows 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.

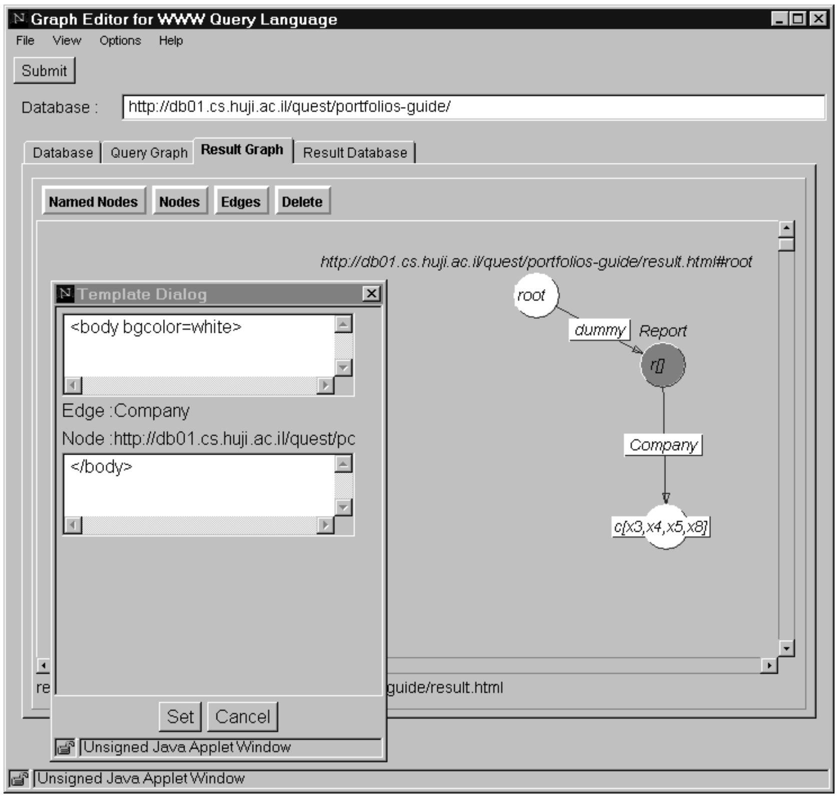

Fig. 5. The result graph and a node template

The Filter Constraints are applied during the filtering phase and filter the information that was found in 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 5.

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 5. Alternatively, one can create template files (see e.g. Figure 7) that also contain a description of the result graph.

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 4 and 5 together form a query. The result of posing this query to the Portfolios Guide database is shown in Figure 6.

4 QUEST Queries and Their Evaluation

In Section 3.2, we introduced the components of a query. We now show how a query graph with search constraints and filter constraints is evaluated.

A *query graph* is a skeleton, where each node and each edge is associated with a distinct variable. Each edge also has an *edge label,* which is a simple string or a regular expression over strings. Figure 4 shows an example of a query graph. Note that variables associated with edges are not shown. We sometimes say "node variable X" when we mean "the node associated with variable x."

4.1 Matchings

The query graph is matched against the database graph by mapping each node variables of the query graph to a node object on the database, and mapping each edge variable to an edge or a path in the database. We distinguish between *total matchings,* i.e., mappings in which all query variables are bound to database nodes or paths, and *partial matchings* in which some variables remain unbound (and are assumed to be mapped to the symbol $\perp$, called *null*).

Due to the semistructured nature of the Web, data in the Web do not conform to a rigid schema, and thus the data may be incomplete. Allowing only total matchings for a query is too restrictive, since this would assume that information appears in certain concrete patterns and is complete. For this reason, QUEST has been designed so that it can handle incomplete information and return incomplete answers.

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

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 edge labels on the path π satisfies the regular expression l.

If the label l of the edge e is just a string, the last condition means that $\mu(e)$ consists of a single database edge that 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.

Usually, query languages consider only total matchings, i.e., partial information is ignored when answering a query. Total matchings are defined as follows.

Definition 1 (Total Matchings). *A total matching is an assignment of objects and edges of the database to the variables of the query, such that each name constraint is satisfied and each edge constraint is satisfied.*

That is, 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, i.e., variables are mapped either to database entities or to $\perp$. Thus, the requirement that all edge constraints and name constraints be satisfied has to be relaxed. Essentially, we require that name and edge constraints be satisfied only when the corresponding variables are assigned non-null values; moreover, we also require that the portion of the query graph that is assigned non-nulls will be a skeleton.

Formally, 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). Partial matchings are defined as follows.

Definition 2 (Partial Matchings). *A partial assignment μ is a partial matching if it has the following properties.*

1. *if μ is defined for a named node of the query, then the name constraint is satisfied;*
2. *if μ is defined for an edge of the query, then the edge constraint is satisfied;*
3. *the edges and nodes for which μ is defined form a skeleton.*

Condition 3 means that if x is either a node or an edge that is assigned a non-null value, then there is a path from a named node to x, such that μ assigns non-null values to all nodes and edges on that path.

4.2 Constraints

In addition to the constraints implicit in the query graph, explicit constraints can be specified. Explicit constraints are either *search constraints* or *filtering constraints.* Furthermore, in the presence of nulls, constraints can be satisfied either *weakly* or *strongly.* We first define weak and strong constraints.

Constraints are expressions combined of Boolean operators and *atomic expressions.* Atomic expressions are either constants or variables that occur in the query graph. A variable in a constraint can be bound either to a value of an atomic database node or to an object identifier of a node. Thus, we have two sets of comparison operator: $C_v = \{<, \leq, >, \geq, ==, !=, = \ \}$ for comparing values and $C_i = \{$=o=, !o=$\}$ for comparing the identities of database objects. 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 a comparison operator from C_v or C_i.

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. The comparison operators in C_v expect atomic values as arguments. If the variable in the argument position is bound to a complex database object, the constraint is not satisfied. In the other cases, the constraint can be evaluated in the following two ways:

1. *Strong Evaluation*: the constraint $a_1\theta a_2$ is satisfied if μ is defined for both a_1 and a_2 and the values to which a_1 and a_2 are bound by μ satisfy θ;
2. *Weak Evaluation*: the constraint $a_1\theta a_2$ is satisfied if one of the following is true: (1) the assignment μ is not defined for a_1 or μ is not defined for a_2; (2) the assignment μ satisfies $a_1\theta a_2$ under strong evaluation.

If $\perp$ is assigned to some variable of a simple constraint, then the constraint is never satisfied under strong evaluation and is always satisfied under weak 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 as a strong constraint.

4.3 Search Constraints and Maximal Matchings

For each explicit constraint, the user specifies whether it is weak or strong and, furthermore, whether it is to be used in the search phase or in the filtering phase.

During the search phase of the query evaluation, QUEST constructs matchings for the variables of the query graph. These matchings must satisfy Definition 2 and, furthermore, each explicit search constraint must be satisfied either weakly or strongly, as specified by the user.

The partial matchings constructed during the search phase may exhibit some redundancies, since a partial matching μ may yield another partial matching μ' by making μ defined for fewer variables. Formally, we say that μ *subsumes* μ' if for every variable x for which μ' is defined, $\mu(x) = \mu'(x)$. In other words, μ is the same as μ', except that μ 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. To avoid redundancies as well as unnecessary computations, the search phase should only construct maximal matchings. Note that maximal matchings cannot be extended over the given database without violating some constraint. Intuitively, maximal matchings contain maximal information, which is the best we can expect when information in the database may be incomplete. Maximal matchings can be viewed as a generalization of the notion of full disjunction [RU96,GL94].

Consider the query graph in Figure 4. Table 1 shows the maximal partial matchings produced by the query, when evaluated over the database depicted in

Figure 1. For easier comprehension, we have replaced the oid's of atomic objects by their values. We only show the assignments to the node variables, since (in this example) these assignments uniquely determine the assignments to the edges.

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 of the query in Figure 4

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.

4.4 Filter Constraints

During the second phase of the query evaluation, the maximal matchings from the search phase are filtered. Filter constraints are either strong constraints or weak constraints, as specified by the user. The maximal matchings that satisfy all the filter constraints are called *solutions*.

There is a need for both strong constraints and weak constraints due to the presence of partial information. The basic difference between the two is that strong constraints are only satisfied if variables are bound, and thus certain information is required to be present in order to satisfy strong constraints. Weak constraints do not require information to be present. If information is available in a given matching and that information violates the constraint, then the matching is dismissed; however, if the information is not available, then the matching gets the benefit of the doubt and is retained. For example, if a query asks for companies that have a market value of at least 500 million dollar, then we will not receive in the result companies for which it is known that their market value is below that figure, but we may receive companies for which the market value is unknown.

For strong constraints, it does not matter whether they are applied during the search phase or during the filtering phase. The reason for that is that a strong constraint is satisfied only if all the variables in the constraint are assigned non-null values. Consequently, it is advisable to apply strong constraints as early as possible during the search phase in order to prune the search space.

For weak constraints the situation is different, since a weak constraint may be satisfied by changing the assignments of some variables to nulls. Therefore, it makes a difference whether a weak constraint is applied during the search phase or during the filtering phase. Note that if a new weak constraint is added to a query as a search constraint, then it may change the result by forcing some null assignments to variables that previously were assigned non-nulls. However,

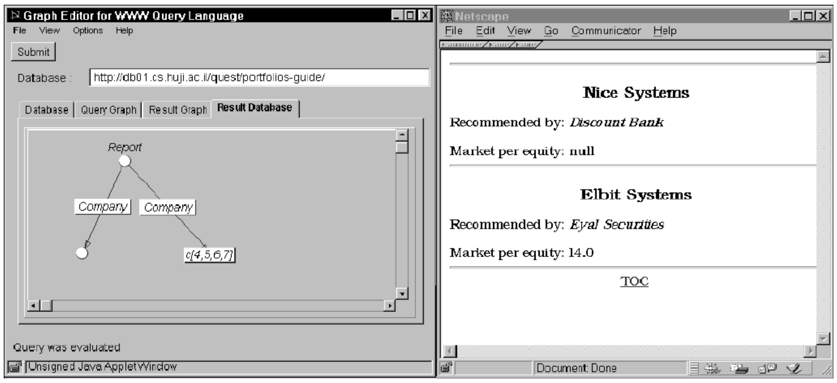

Fig. 6. The result database produced when evaluating our example query (the query graph of Figure 4 and the result graph of Figure 5) over the database of Figure 3

the new weak search constraint will not decrease the number of solutions to the query. If the same weak constraint is used in the filtering phase, then it may decrease the number of solutions to the query. Intuitively, weak search constraints have the effect of luring maximal matchings to be as large as possible. Once the maximal matchings are produced, the filter constraints are used to eliminate some of those matchings. In fact, one reason for having maximal matchings as the result of the search phase is in order to give as much elimination power as possible to the weak filter constraints.

Note that the edge constraint and the name constraint (Definition 2) are a form of weak search constraint.

The set of solutions, which is obtained as the outcome 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 phases produce sets of partial matchings, we need a mechanism to convert those partial matchings into OHTML pages. When creating OHTML pages, we must take into account the two aspects of OHTML, namely, the semantic view and the visual view. Thus, the answer returned by a query must include an OEM graph, and that graph must be embedded in HTML pages by means of OHTML tags.

In principle, there are two ways to combine the two aspects, depending on which aspect is given priority. The first approach is to produce an OEM graph and decorate its nodes and edges with HTML. The second is to create HTML pages, and embed in those pages objects and edges of the OEM structure. 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 use two formalisms in the creation of the result. The first is a *result graph* that 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 edges among these new objects and edges from the new objects to other database objects.

These two functions are accomplished by means of a result graph. A *result graph* is essentially a skeleton with edges that are labeled with *edge labels* and nodes that are labeled with 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 solutions of a query are (partial) assignments of database objects to node variables of the query. Suppose that $f[x_1, \ldots, x_n]$ is 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 solution μ, we create a new object with the oid $f[o_1, \ldots, o_n]$. Note that if two solutions μ_1 and and μ_2 are equal on all the x_i, then only one object is created for them.

There are different ways to handle tuples with nulls. One approach is to create new objects only when all the variables of a term are bound to non-null values. However, this approach is too restrictive. Instead, we treat each null value as a unique database object. In this way, we take into account partial solutions that may not bind all variables, and we utilize this partial information in order to create new objects.

Summarizing, new objects are generated as follows. First, in each solution μ, replace every null with a new unique non-null value. Secondly, for each solution μ and each flat term $f[x_1, \ldots, x_n]$, create a new object having the oid $f[\mu(x_1), \ldots, \mu(x_n)]$ (duplicates are removed).

Once the result objects are generated, edges are introduced between them according to the edges in the result graph. Suppose that there is an edge, labeled with l, from node n_1 of the result graph to node n_2. If there is a solution μ, such that μ generates objects o_1 and o_2 from the flat terms of n_1 and n_2, respectively, then we create an edge, labeled with l, from o_1 to o_2.

Suppose that n is a leaf node (i.e., a node without any outgoing edges) of the result graph, and let t be the flat term of n. If o is an object created from a solution μ and the flat term t, then o is an atomic object. Since atomic objects have values, each leaf node of the result graph has an associated string s. The string can include variables of atomic nodes of the query graph. Such variables should be enclosed by the $ sign, i.e., x. Note that the string may be just a variable. The variables in the string are instantiated according to the solution

μ, and the instantiated string becomes the value of the atomic object generated from μ and the leaf node. Since an atomic object can have just a single value, each variable appearing in the string of a leaf node of the result graph must also appear in the flat term of that leaf node. This requirement guarantees that an atomic object is created for each distinct value that is produced by applying solutions to the string of the leaf node.

A special case of a flat term is a variable, e.g., x. In this case, no new objects are created, since no Skolem function is applied to existing oid's. Therefore, when a variable is used as a term, it actually defines connections between result objects and objects of the database over which the query is evaluated. Since we want to avoid situations in which the result graph implies that a new outgoing edge has to be added to an existing object, we allow a variable as a term only in leaf nodes of the result graph. This requirement guarantees that new edges are added only between two new objects, or between a new object and an object that already exists in the database.

QUEST requires result graphs to be acyclic. In addition, the list of variables in the term of 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 they contain these objects, and each one of the objects $f[o_1]$ and $f[o_2]$ must encapsulate in its OHTML representation the representation of the object $g[o_e]$. Thus, we need to have the ability to break an HTML page into pieces stored in more than one physical location. This resemble the usage of parameter entities in XML, but is not an HTML feature.

QUEST automatically adds missing variables to flat terms, when those variables are needed according to the requirements specified in this section.

Since only one object can exist in the uppermost level of each OHTML page, a dummy root object is created for each OHTML page produced in the result. Such a root object encapsulates the objects in the uppermost level of the page and the HTML text that appears before and after those objects. In order to create root objects, the root of the result graph is required to have a flat term that has the Skolem function symbol *root* and some variables. Each instantiation of this flat term by some solution μ will create a new root object that will reside in a new OHTML page. Thus, the *root* term defines the partition of the result database into pages.

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 given node a preceding and a succeeding HTML text. Actually, the text is HTML with references to variables of atomic nodes of the query. The variables in the text segments are instantiated to the atomic values to which they have been bound, and the instantiated text segments surround the result objects that are created from the given node.

```
<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>
```

Fig. 7. A textual representation of a template

In the actual implementation, the system produces, for each node in the result graph, a *node template* that consists of the term of the given node 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 leading to immediate subobjects and to other objects that are referenced by the given object. Hence, each node template is essentially an OHTML document with variables from the query. The node templates are combined into a *query template*, which may consist of one ore more OHTML pages (with variables).

New OHTML pages are generated from a query template by evaluating the template over the solutions to the query. For each solution, each variable of the template is bound either to a complex database object (more precisely to an oid reference), to the value of an atomic object, or to a null.

Figure 7 shows a query template, where `report` and `company` are two Skolem functions, and `x4`, `x5` and `x8` are variables embedded in the text.

6 Conclusions

We have designed and implemented QUEST—a graphical query language for semantically tagged pages on the Web. QUEST was implemented in Java and thus has the benefits of Java applications, such as system independence, object oriented design, etc. QUEST has a client-server architecture, where the client is a Java applet. It uses a main memory approach when querying the Web.

The most novel feature of QUEST is its ability to query incomplete information and return incomplete maximal answers as a result. We believe that the ability of QUEST to query incomplete information is of great importance, due to the semistructured nature of the Web. We also believe that the mechanism of finding maximal matchings is natural for querying partial information in general, and not just in QUEST. For more details on the foundation of query processing in QUEST see [KNS99].

QUEST provides a graphical query language. We believe that the graphical interface facilitates easy and succinct formulation of complex queries that may involve a number of path expressions and constraints. Similar queries are not expressed as easily in textual query languages, such as Lorel [AQM+97]. Moreover, the graphical interface also facilitates construction of new Web pages that have both HTML tags and object structures. Hence, the principles of this graphical interface may also apply to querying XML documents and generating new pages for the result by means of style sheets.

Currently, our main effort is to alter the system to support XML. We believe that the principles used in QUEST are sufficiently general and important to be carried over to query languages for XML documents.

References

[Abi97] S. Abiteboul. Querying semi-structured data. In *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+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.

[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.

[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. Queries with incomplete answers over semistructured data. In *"Proc. 18th Symposium on Principles of Database Systems"*, "Philadelphia (Pennsylvania, USA)", may 1999. ACM Press.

[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 *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.