A Graph-oriented Query Language for Semi-Structured Data: Theoretical and Practical Analysis

Jimmy Elkada

A Major Report
in The Department of Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

ABSTRACT

A Graph-oriented Query Language for
Semi-Structured Data: Theoretical and Practical Analysis

Jimmy Elkada

This study examines the theoretical foundations and the practical aspects of the graph-oriented query language for the semi-structured data (SSD) proposed in [KNS99]. SSD is a data model that is designed for heterogeneous data sources. It allows for information integration and information sharing over the Internet. Several query languages for the SSD model have been proposed but none has been standardized yet. This paper analyzes the *graph-oriented* query language proposal, and suggests ways in which it can be further improved to fit the SSD model.

Acknowledgements:

First and foremost, I would like to thank my supervisor, Professor Goste Grahne, for taking me on as a student about two years ago, even though he knew little about me. He has been a role model to me in his dedication and professionalism. My choice of career has been greatly influenced by Mr Grahne and I hope that I can live up to his high standards.

Table of Contents

Part 2: Software Document

List of Figures

List of Tables

Part 1: Paper Summary and Analysis

1 Introduction

Semi-structured data (SSD) has recently emerged as an important topic of study in database for a variety of reasons. First, there are heterogeneous data sources such as the Web, which we would like to treat as databases and cannot be constrained by a schema. Second, due to information sharing in both the commercial and the academic fields, it is desirable to have an extremely flexible format for data exchange between disparate data sources. Third, it may be helpful to view structured data as semi-structured for the purposes of web browsing and web information sharing.

There are several query languages that have been proposed to deal with the SSD model, such as LOREL and XML, most of which resemble one another and follow the SQL format that was developed for the relational model. A different kind of query language for SSD is now emerging, one in which the query has a graph format. This graph-oriented query language seems to be more natural to a SSD model, which is also represented as a graph. The database graph is queried according to a query graph template (with graph variables) by finding maximal possible matchings and then filtering them according to the user's request.

In [KNS99], *Queries with Incomplete Answers over Semistructured Data*, the authors suggest such a graph-oriented query language. This query language allows for partial answers to a query, in that it does not require that all graph variables be matched to the database graph. Incomplete answer should be an important feature of any SSD query language since such a language deals with non-structured heterogeneous data, which does not abide by a strict and complete schema. The above paper also discusses other aspects unique to a graph-oriented query language such as containment mapping (homomorphism) which lays the ground for query optimization.

The main purpose of Part 1 of the document is to discuss SSD query languages in general and the suggested graph-oriented query language in particular. It will summarize, analyze and then criticize the above research paper ([KNS99]). The summary explains the key ideas in the paper in a simple, intuitive manner.

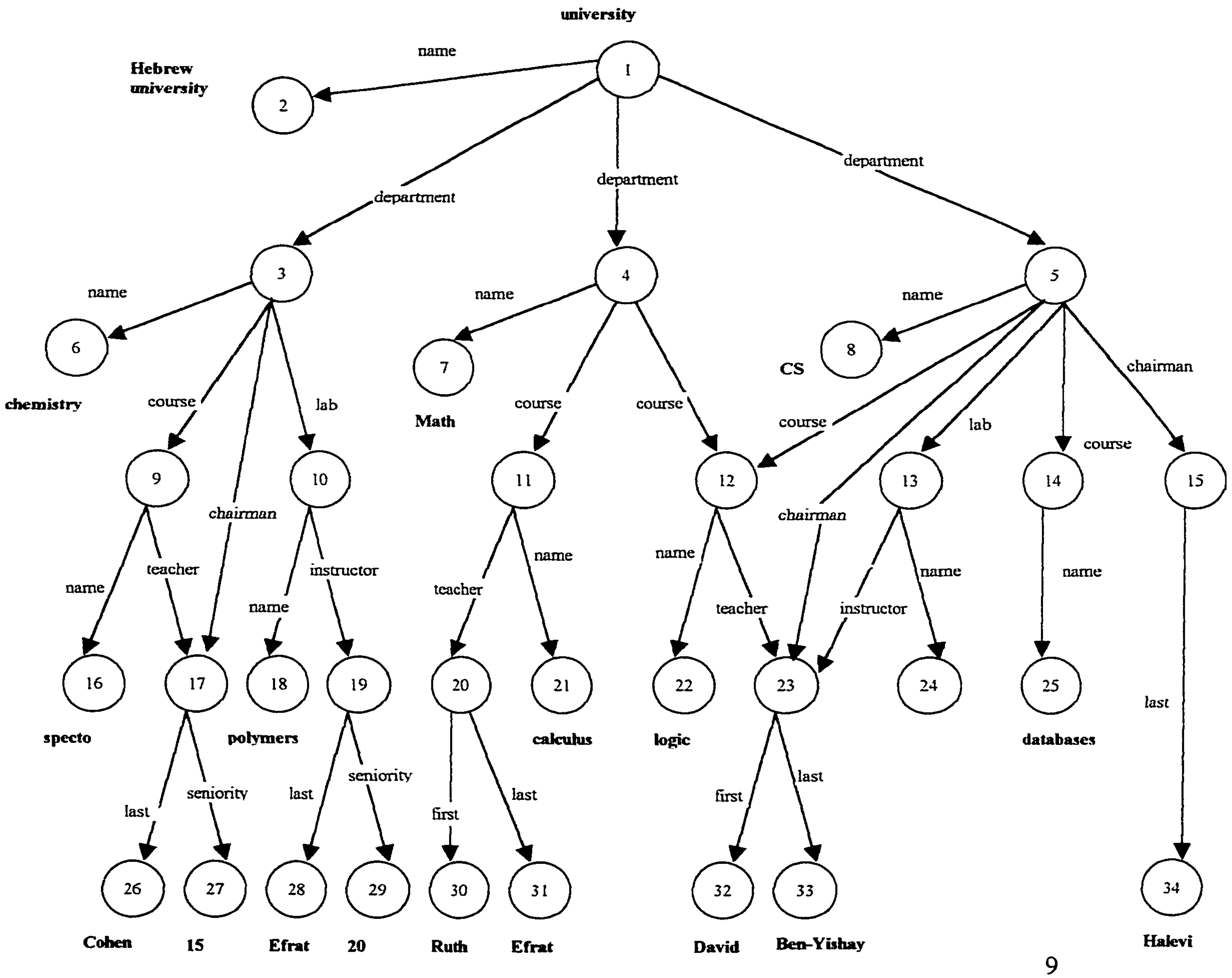
Part 2 of this document describes the implementation of the graph-oriented language proposed in Part 1. This part is structured according to software engineering principles on which the program was developed, and thus, includes requirement specification, software design, implementation and testing. The program is available for use on the web at: www.cs.concordia.ca/~grad/elkada/project
This document is available online at: www.cs.concordia.ca/elkada/project/docs/report.doc

2 Query Languages and Semi-structured Data

2.1 SSD Overview

Semi-structured data (SSD) is often referred to as a "schemaless" data model, in which there is no separate description of the type or the structure of the data. The data consist of label-value pairs such that the label is the "attribute" of the data. The main advantages of using such a data model, where data and metadata are mixed, are data heterogeneity and data integration, which are basic features of the data used over the Internet. The label-value pairs of data can be graphically represented as nodes in a graph with labeled edges that connect the data nodes.

Figure 1: The Hebrew University database (example of a SSD)



2.2 SSD Query Languages

The development of query language (QL) for semi-structured data is very much in its infancy and currently there is *no* standard SSD query language. There are several proposed query languages for SSD such as LOREL, UnQL and XML-QL, discussed shortly. These are similar in terms of syntax and semantics, but each one contributes different aspects to the research of SSD query language. They all use syntax similar to SQL (select-from-where) but their semantics are slightly different.

One of the main features of SSD QL is the ability to reach to an arbitrary depth in the data graph. To do this a SSD query language needs to exploit, in some form, the notion of path expression. David Maier [Mai86] laid out the features common to any query language for SSD. These include the following:

- Regular path expressions – the SSD model is represented as a graph whose nodes are the data to be queried. Therefore, a SSD QL should be able to fetch information stored at any level of the database graph. Regular expressions with elaborated wildcard features are major component of any QL that deals with a database graph representation.
- Expressive power – another important QL feature is the ability to support all kinds SQL or RA operations. Since most databases today are based on the relational model, and these databases can be viewed as a SSD graph, the success of a SSD QL would depend on its ability to perform common SQL operations, most notably, the join operation.
- Precise semantics –when the meaning of each QL operation is well defined, query transformation and optimization can take place. A QL should enable its user to express the same query in different ways. Such query equivalence is the key to an efficient DBMS that perform intelligent optimizations.
- Compositionality – in relational database the output of a query is also a database from which another query can be composed. It would be desirable that the same idea be applied to a SSD QL. When the output of SSD query can used as input to another, more elaborated QL operations can be defined.
- Structure conscious – a SSD QL should be able to infer and exploit the database schema. As the name suggests, SSD is partly structured and not entirely non-structured. Since a well-defined schema does not exist in SSD, a QL should take into account the general structure of the data and be able to use this information to optimize its search mechanisms.
- Program manipulation – any SSD QL should have a simple and clear but verbose core language. A QL that supports many operations allows for better query optimization.

The above features can be used, among other database principles, to evaluate the contribution of the proposed graph-oriented query language to this recent topic of research. However, in order to investigate, analyze and compare the various SSD query languages a brief review of the current trend is needed. The following sections describe common SSD query languages. This description is neither formal nor complete. The syntax and the semantics are demonstrated using examples that are based on the Hebrew University database graph given previously in figure 1 (for more refer to [Abi97]).

2.2.1 LOREL

LOREL (Lightweight Object REpository Language) is a query language in LORE, a system for managing SSD developed at Stanford, which was derived by adapting OQL (Object Query Language) to querying SSD. LOREL is based on the syntax of OQL and makes substantial use of regular path expressions that allow reaching an arbitrary depth in the data graph. The following is an example of a query in LOREL that asks for all the teachers' last name in the CS department of the Hebrew University.

```
select  result: X.last
from    university._*.teacher X
where   university.department.name = "CS"
```

2.2.2 XML-QL

XML-QL (Extensible Markup Language – Query Language) is a query language for the World Wide Web, which combines XML/HTML syntax with SSD QL techniques. It uses variables to which data is bound ($N in the example below) and an output template to construct the result, which is also in XML. Like LOREL, XML-QL also uses path expressions to match a sequence of edges. It uses where-construct syntax, which is similar to the select-from-where syntax of SQL and OQL. The *construct* clause corresponds to *select* whereas the *where* clause combines the *from- where* parts of the query. Also, it deals with optional elements through nested queries. The following is a simple example of a query in XML-QL that asks for all teachers' family names in the CS department.

```
where  <*.teacher> <family_name> $N </> </>,
       <university>
              <department> <name> $D </> </>
       </>, $D = "CS"
       in "www.somewhere/db.xml"
construct <result>
              <family_name> $N </>
          </>
```

2.2.3 GQL

In the graph-oriented query language (GQL) proposed in [KNS99] the query takes on a graph format, in which the pattern used in the query is more "visual". It is claimed that a graph-oriented query model is more natural and intuitive for graph-oriented database model. In this model the query is a graph whose nodes are variables that need to be bound (matched) by database values, according to some given constraints (filter constraints).

Thus, queries are evaluated in two phases: (1) match phase and (2) filter phase. In phase 1, the database is searched for the pattern/template specified in the graph query and the result is a list of possible answers that are matched from the database. These are called *maximal matching* since they represent the maximal possible information that can be obtained from the given query. In phase 2, these maximal matchings are filtered in order to output only these matches that are needed according to the filtered constraints given by the user. Section 4, Query Evaluation, discusses these two phases in more detail and provides some algorithms and examples.

3 Graph–oriented Query Model

This section describes the syntax and the semantics of the graph-oriented QL proposed in [KNS99] and provides some examples that demonstrate the mechanisms of the GQL. The following notations and definitions are simple and intuitive and their purpose is to provide some theoretical background and terminology, so that the GQL model can be further explored in and analyzed in the coming sections.

3.1 Syntax

As mentioned earlier in the introduction, the focus of this paper is to summarize and analyze the graph model, therefore a detailed definition and a precise notation with comprehensive theorem proofs are not needed here. However, some notation is needed for the algorithms and theorems described in later sections.

A label directed graph (LDG) over a set of nodes N is denoted as $G = (N, \cdot^G)$ where $\cdot^G$ associates with each label $l \in L$ a binary relation $l^G \subseteq N \times N$ between the nodes. For example, in the database graph given in section 2.1 $\cdot^G$ associates with the label "department" the binary relation $\{<1,3> <1,4> <1,5>\}$. Also, the binary relation l^G can be viewed as a function $l^G : N \rightarrow 2^N$. In the above example $1 \rightarrow \{3, 4, 5\}$, also denoted as $3 \in l^G(1)$, $4 \in l^G(1)$, $5 \in l^G(1)$.
Thus, the notation for an **ldg** is $G = (N, \cdot^G)$.

A **rooted ldg** G is denoted as a triple $G = (N, r_G, \cdot^G)$ where $r_G \in N$ is the root node. A **database** D is denoted as the 4-tuple $D = (O, r_D, \cdot^D, \alpha)$ where α is a function that maps each terminal node to an atom (for example, in the university database α maps the terminal node 26 to the atom "Cohen"). Here O is the set nodes in the database graph, which are actually a set of object id.

A **query** Q is the triple Q = (G, F, x) where G is a query graph whose nodes are variables, F is a set of filter constraints, discussed later (see section 3.2.2), and x is a tuple of variables occurring in N (a sub set of the nodes in the query graph), which is the output of the query (in figure 1 example, these are x_1 and x2).
The **query graph** $G = (N, r_G, \cdot^G)$ can be viewed as a set of constraints over N denoted as Cons(G). Therefore, an edge constraint ulv is in Cons(G) if there is a label l from u to v. (i.e. $u \in l^G(v)$). The constraints in Cons(G) are called *search constraints* and are used to search for maximal matching (in the example on the following page (u, lab, w) is an edge constraint in the set Cons(G))

An **assignment** μ is a mapping/binding of the query variables to the database objects. A variable v is bound if $\mu(v) \neq \perp$ (null). An assignment μ satisfies an edge constraint ulv if $\mu(u) \in l^G(\mu(v))$, i.e. the database graph contains the pair ($\mu(u)$, $\mu(v)$). In the example of figure 1, there is an assignment μ that satisfies the edge constraint (y, first, x_1) because there exists a database edge (20, first, 30) such that $30 \in l^G(20)$.

3.2 Semantics

There are four types of query semantics that can be applied to the database graph: STRONG matching, WEAK matching, AND-matching and OR-matching. These semantics express different levels of strictness in which variables are bound to database objects allowing for a more flexibility in incorporating incomplete matching to the query variables. The actual algorithms for implementing them are given later. The following is description and explanation of these semantics using the query example in figure 2.

3.2.1 Search Constraints

STRONG matching
An assignment μ is a STRONG matching if μ satisfies *every edge constraint* in the query graph. Therefore, it requires a total assignment. For the query in figure 1, there would be no matchings under this semantics since there is no corresponding y node in the database graph, such that all of its children x_1, x_2, x_3 exist. STRONG semantics can be used in the filter phase to assure that node variables exist in the solution set. Since SSD is inherently incomplete this semantics is not usually used in the search phase.

WEAK matching
An assignment μ is a WEAK matching if μ satisfies *every edge constraint* in the query graph *whenever it is defined.* Therefore, it requires only partial assignment. The strict requirement of STRONG semantics, that an assignment μ must satisfy *all* constraints, is loosened to an assignment that satisfies all constraint only if they exist in the database. WEAK matching requires that if μ is defined for u and v then μ has to satisfy the constraint ulv. This semantics is explained later in the example of figure 2.

AND-matching
An assignment μ is an AND-matching if μ satisfies *all incoming constraints* of a node y whenever $\mu(y) \neq \perp$. Therefore, in the maximal matching set, database nodes which have more than one incoming edges are included only if all of these incoming edges are matched. In the figure 1 example, the query node y is matched only to node 23 from the database. Node 23 is the only y node that actually has 2 incoming edges in the database. Under AND-semantics, the query in figure 1 asks for people who are both course teachers and lab instructors.

OR-matching
An assignment μ is an OR-matching if μ satisfies *some incoming constraints* of a node y whenever $\mu(y) \neq \perp$. Therefore, in the maximal matching set, database nodes which have more than one incoming edges are included if at least one of these incoming edges are matched. In the figure 1 example, y is never assigned a null value because at least one of its incoming edges exists. Under OR-semantics, this query asks for people that are either course teachers or lab instructors.

The sets of STRONG, WEAK, AND and OR matchings denoted as $\mathrm{Mat}_{\text{-}s}(Q)$, $\mathrm{Mat}_{\text{-}w}(Q)$, $\mathrm{Mat}_{\text{-}\cap}(Q)$, $\mathrm{Mat}_{\text{-}\cup}(Q)$ respectively, have an interesting property:
(this property does not apply to maximal matchings)

$$\mathrm{Mat}_{\text{-}s}(Q) \subseteq \mathrm{Mat}_{\text{-}\cap}(Q) \subseteq \mathrm{Mat}_{\text{-}w}(Q) \subseteq \mathrm{Mat}_{\text{-}\cup}(Q)$$

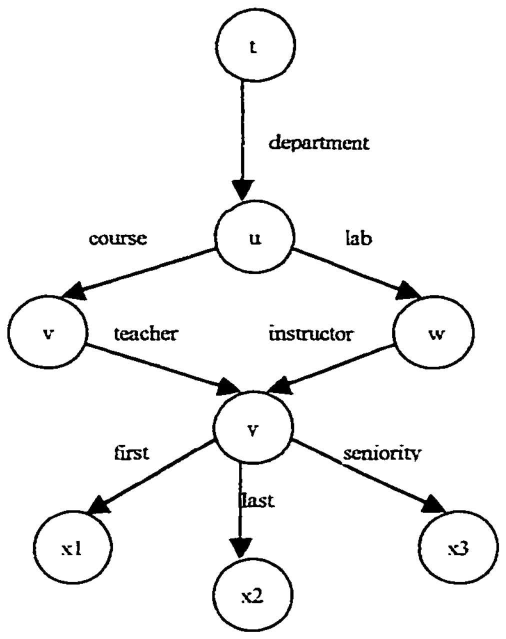


Figure 2: Query graph asking for course teachers and lab instructors.

	t	u	v	w	y	X1	X2	X3
AND (1)	1	3	9	10	⊥	⊥	⊥	⊥
(2)	1	4	11	⊥	⊥	⊥	⊥	⊥
(3)	1	4	12	⊥	⊥	⊥	⊥	⊥
(4)	1	5	12	13	23	32	33	⊥
(5)	1	5	14	13	⊥	⊥	⊥	⊥
OR (1)	1	3	9	10	17	⊥	26	27
(2)	1	3	9	10	19	⊥	28	29
(3)	1	4	11	⊥	20	30	31	⊥
(4)	1	4	12	⊥	23	32	33	⊥
(5)	1	5	12	13	23	32	33	⊥
(6)	1	5	14	13	23	32	33	⊥

Table 1: Maximal matchings for the graph query in figure 2 (search phase).

3.2.2 Filter Constraints

Filter Constraint Types

As mentioned in section 3.1, a query Q is the triple Q = (G, F, x) where G is a query graph, F is a set of filter constraints and x is the set of the output variables (subset of N – the set of all graph variables). Filter constraints F reduce the set of maximal matching to a set of solutions, and output variables reduce the set of solutions to a set of answers. There are three types of filter constraints that can appear in F:

- *Atomic Constraint* – constraints of the form $u =_v v$ or $u \neq_v v$ where u is a query node and v is a database. For example, x_1 = "David".
- *Object Comparison* – constraints of the form $u_1 =_o u_2$ or $u_1 \neq_o u_2$ where u_1 and u_2 are objects in the database. For example, $x_1 = x_2$ requires that the person's first name be equal to the last name.
- *Existence Constraint* – denoted as !v requires that a variable v be bound. For example, $!x_1$ in figure 1 requires that the first name be in the solution.

Filter Constraint Satisfaction

- *Strong Satisfaction* - a filter constraint is strongly satisfied, denoted as $\mu \models_s C$, if the variable in this constraint is defined/bound and the equality/inequality is true. For example, in table 1 with the atomic filter constraint x_1 = "David" under AND semantics with *strong* satisfaction, no matching will be included in the solution set (x1 is either not defined or not equal to "David").
- *Weak Satisfaction* - a filter constraint is weakly satisfied, denoted as $\mu \models_w C$, if the variable in this constraint is undefined/non-bound (but if the variable is defined, then the equality/inequality must be true for the constraint to be weakly satisfied). For example, in table 1 with the atomic filter constraint x_3 = "20" under AND semantics with weak satisfaction matching (4) will be included in the solution set since this variable is undefined.
- Existence Constraint - an assignment μ *strongly/weakly* satisfies an existence constraint !v if $\mu(v) \neq \perp$, i.e. if the variable v actually exists in the database.

As mentioned earlier, the solution set (obtained from the filter phase) is eventually reduced to the answer set according to the x set in the query Q = (G, F, x). x is the set of output variables. For example, if $x = \{x_1, x_3\}$ for the query in figure 1 under OR semantics and weak satisfaction of the constraint x_3 = "20" the answer set would be: $\{ <\perp, 29>, <30,\perp>, <32, \perp> \}$

3.3 Example

As mentioned earlier, the query evaluation process is performed in two steps. In the first step, the search phase, we retrieve as many matchings that are as full as possible. In the second step, the filter phase, we filter out the unneeded matchings according to the filter constraints and then obtain the output variables as answers. The following graph-oriented query in figure 2 asks for the last name of a person that is both a course teacher and department chairman in the university. The query implies AND semantics and the output variable v_5 (last name). Lets assume that there is also an existence constraint v_3! which means that the person node v_3 must exist in the filter phase.

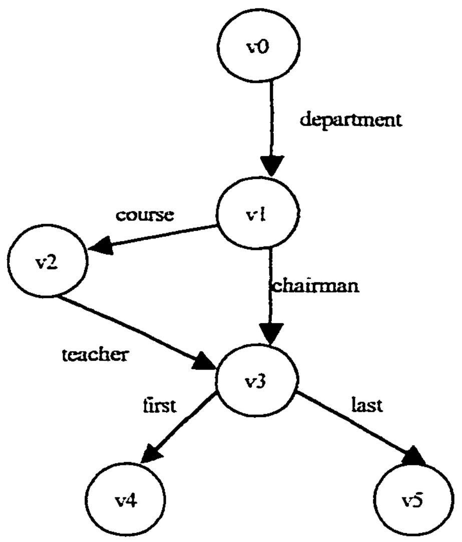


Figure 3: the query asks for the first and last names of people who are both course teachers and department chairmen in the Hebrew university.

In the search phase the maximal matching set under AND-semantics would be:

	V0	V1	V2	V3	V4	V5
(1)	1	3	9	17	⊥	26
(2)	1	4	11	⊥	⊥	⊥
(3)	1	4	12	⊥	⊥	⊥
(4)	1	5	12	23	32	33
(5)	1	5	14	⊥	⊥	⊥

Table 2: Maximal matchings for the query in figure 3 under AND semantics

In the filter phase, we have to examine the filter constraints. We have an existence constraint !v_3 which means that person node must exist in the solution set. Thus, only matchings (1) and (4) from table 2 will be included in the solutions as shown in table 3: (the other matchings are not included since node v_3 is assigned null):

	V0	V1	V2	V3	V4	V5
(1)	1	3	9	17	$\perp$	26
(2)	1	5	12	23	32	33

Table 3: Solution set for the query in figure 3 under AND semantics

Finally, the answers are obtained from the solution by keeping only the output variables, which is v_5 for the given query.

	V5 (last name)
(1)	26 (Cohen)
(2)	33 (Ben-Yishay)

Table 4: Answer set for the query in figure 3 under AND semantics

In the above example we would have obtained the same results under WEAK semantics since WEAK semantics requires that if nodes v_1 and v_3 are bound then there must exist a "chairman" edge. If node v_3 had 3 incoming edges (teacher, instructor and chairman) then under WEAK semantics the query would ask for all people who are chairman and either a teacher or an instructor.

4 Query Evaluation Process

The query evaluation process includes two phases: (1) search phase, in which the maximal matching set is found, and (2) filter phase, in which the solution set is constructed (by filtering the maximal matching set). These two phases are detailed next, in sections 4.1 and 4.2. Note that each phase contains several mechanisms in order to produce the matching/solution set. These are the techniques that provide the proposed GQL with its particular feature of obtaining incomplete/partial answers to SSD queries. As mentioned in the Introduction, one of the objective of the GQL in [KNS99] is create a QL flexible enough to deal with non structured database graph.

4.1 Search Phase

The following sections describe the algorithms that are used in the search phase in order to compute the set of maximal matchings for tree and acyclic graphs under the different semantics. The proofs of the algorithms for cyclic graphs are NP complete are not included here. The first case, which is the simplest, is tree query graph. The second case, acyclic query graph, is an extension/generalization of the tree graph algorithm.

The algorithms use the notion of topological order $V_0 < V_1 < \ldots < V_k$ which is the sorting or ordering of the graph nodes such that $V_i < V_j$ if there is an edge from V_i to V_j.

4.1.1 Tree graph

When the query graph looks like a tree, then the query result is the same for WEAK, AND and OR semantics since only one incoming edge is allowed. The following algorithm, EvalTreeQuery, describes how to find the set of maximal matching for a query that is a tree graph.

EvalTreeQuery

```
Let $V_0 < V_1 < \ldots < V_k$ be a topological order
$S = \{V_0/r_d\}$
for i = 1 to k do
        for each $\mu \in S$ do
            if $(V_{i-1}, l, Vi)$
                then $S = S \cup \{\mu \oplus [Vi/o] \mid o \in l^G(Vi)\}$
                else $S = S \cup \{\mu \oplus [Vi/\perp]\}$;
        return S;
```

$\mu \oplus [Vi/o]$ means: bind Vi to the database object o, in addition to the variables already bound in μ.

4.1.2 Acyclic graph (dag)

In a dag query graph there may be more than one incoming edge for a node and hence, unlike tree graphs, the query result depends on the semantics used. The idea in EvalDagQuery algorithms is to extend the previous EvalTreeQuery such that nodes which have more than one incoming edges need to be assigned/matched according to the semantics used in order to be included in the maximal matching set.

The following describes the rest of the algorithms that can be used in the search phase. Since these are extensions of EvalTreeQuery, only the main ideas of the other algorithms are provided here (see [KNS99] p.13-21 for details).

EvalDagQuery for AND/OR semantics

We can use the same algorithm for AND/OR semantics, with parameter $\sigma \in \{\cup,\cap\}$, indicating which semantics AND/OR is used. Like EvalTreeQuery, in EvalDagQuery we start by establishing a linear ordering of the query nodes $V_0 < V_1 < \ldots < V_k$. For every query node Vi, we extend the set of maximal matchings with assignments to database object that correspond to that node. The difference is that in EvalDagQuery, depending on the given semantics $\sigma \in \{\cup,\cap\}$, a node Vi will be included in the result, if it there is an assignment which satisfies some or all of Vi incoming nodes.

Thus, EvalDagQuery introduces a new notation, Ext-σ(Vi) where $\sigma \in \{\cup,\cap\}$, to check the incoming nodes of Vi according to σ, the given semantics. In the case of AND semantics, Ext-σ(Vi) = $\varnothing$ if not all incoming edges are satisfied, and in the case of OR semantics, Ext-σ(Vi) = $\varnothing$ if none of the incoming edges is satisfied.

EvalDagQuery for WEAK semantics

As mentioned earlier, WEAK semantics require that if μ is defined for u and v then μ has to satisfy the constraint ulv. Thus, in EvalWeakDagQuery, when a query node Vi is processed, all the incoming edges Ei need to be considered. Whenever a query edge Ei appears in the database graph, the source node V', in V'EiVi, needs to be bound in the assignment μ_{i-1}. If V' is defined and bound, then Vi/o is added to μ_i. Otherwise all the paths leading to Vi need to be reset to $\perp$ since it does not pass via V' which is not bound.

4.2 Filter Phase

As mentioned earlier, there are three types of filter constraints: atomic, object comparison and existence. The filter phase algorithm is given only for existence constraint in tree query graph. For all other cases, proofs are given to show that the problems complexity function is NP complete which suggests that they cannot be solved in a polynomial time.

An existence constraint !V in a graph query requires that a query node V be bound to a non-null value in the solution. Clearly, the path to V should also be non-null. Therefore, all paths to existence constraints should be fully matched to objects from the database. These existence paths of the query tree compose the *strongly evaluated subtree* of the query since they need to be evaluated under STRONG-semantics.

The suggested algorithm, EvalStrongTree computes the set of strong matchings for the strongly evaluated subtree of the query, in two phases: (1) create binary relation for each edge in the strongly evaluated query, such that each binary relation contains all database edges that satisfy the query edge and (2) compute the join of all the relations created using the algorithm for computing an acyclic join by a semi-join program.

5 Query Containment

The previous section explained the mechanisms behind one of the main features of the proposed GQL, the ability to obtain incomplete answers for a schema-less database. Another important contribution of the proposed GQL is in the area of query optimization. Query optimization is applied in DBMS in order to enhance performance and thus, render the QL more practical.

5.1 Optimization Notions

One of the important features of any query language, as discussed in section 2.2, is precise semantics that allows query optimization. This section summarizes some of the optimization notions for the proposed GQL such as containment, equivalence and mapping, and then uses these concepts to discuss the containment proofs, under the different semantics.

Let Q1 and Q2 be two graph queries.
The following notions can be defined for GQL:
Containment: $Q1 \subseteq_{\sigma} Q2$ if $Ans\text{-}_{\sigma}(Q1) \subseteq Ans\text{-}_{\sigma}(Q2)$ where $\sigma = \{S, W, \cup, \cap\}$
Therefore, Q1 is contained in Q2 under σ semantics, if for every $a1 \in Ans\text{-}_{\sigma}(Q1)$ there is an $a2 \in Ans\text{-}_{\sigma}(Q2)$ such that $a1 \subseteq a2$.
Equivalence: $Q1 \equiv Q2$ iff $Q1 \subseteq_{\sigma} Q2$ and $Q2 \subseteq_{\sigma} Q1$ where $\sigma = \{S, W, \cup, \cap\}$
Clearly, 2 queries are equivalent if and only if they subsume each other with respect to σ.
Mapping: mapping φ from Q1 to Q2 is homomorphism if:
(1) φ maps root to root (i.e. $\varphi(r_1)=r_2$) and
(2) φ maps output variables to output variables (i.e. $\varphi(X_{i1})=X_{i2}$, $X_i \in X$) and
(3) φ maps edge constrains to edge constraints (i.e. $\varphi(u)l\varphi(v) = ulv$)

The proofs of the homomorphism theorem for graph queries under AND-semantics and OR-semantics are given in [KNS99], so the following sections only state the theorem and then informally discuss and summarize the proofs, in a point-form easy-to-understand manner (homomorphism means that query *mapping* implies *containment* and vice-versa).
The abbreviation C.M used below stands for Containment Mapping

5.2 Containment under AND-semantics

Theorem: (containment under AND-semantics)
Let Q1 and Q2 be two graph queries.
$Q1 \subseteq Q2$ iff there is a C.M from Q2 to Q1 under AND-semantics.

Proof: the proof is very similar to the *classical* proof for conjunctive queries as in [Ull89], which shows:

(1) if $(Q2)\varphi=Q1$ then $Q1\subseteq Q2$ and

(2) if $Q1\subseteq Q2$ then $(Q2)\varphi=Q1$.

Part (1) is proven by composing the set of maximal solutions $\mu_2=\mu_1°\theta$, and showing that Q2 produces the same answer as Q1. Part (2) is proven by evaluating both Q1 and Q2 over G1 (the query graph of Q1) which produces answers a1 and a2, then showing that there must be C.M. from a2 to a1.

5.3 Containment under OR-semantics

The main idea for checking if one query is contained in another, under OR-semantics, is to decompose each query to many tree queries and then compare them. Each sub tree of the query graph is called a *spanning tree* and the set of all spanning trees of a query is called the *tree expansion* of the query. The tree expansion of a query Q is defined as:

$\Gamma_Q=\{(T,x) \mid T \text{ is a spanning tree of } Q\}$

In order to prove the theorem of containment under OR-semantics, the following corollary and proposition are defined and proven:

Corollary: (containment among *tree* queries)
This corollary is similar to the theorem from the previous section except that it applies only to tree queries, instead of general queries, but for all types of semantics.
Let Q1 and Q2 be two tree queries.
$Q1 \subseteq Q2$ iff there is a C.M from Q2 to Q1 under WEAK, AND and OR-semantics.
Proof: we know already that the corollary hold for AND-semantics because tree query is sub case of general query. Since for tree queries we get the same query answer under all semantics, the corollary holds under WEAK, AND and OR-semantics.

Proposition: Let Q = (G, x) be a query, and Γ_Q be the tree expansion of Q.
Then we have $Ans_{\cup}(Q) = \cup\ Ans_{\cup}(Qi)\ \forall Qi\in\Gamma_Q$.
Proof: Intuitively the theorem means that the union of all the answers of the spanning query trees is the same as the answer of the general query. Let the answer to the general query Q=(G, x) be $a=\mu(x)$. Let the spanning tree queries be $Q_1=(G,x_1)$, ... , $Q_n=(G,x_n)$ and let the answers to these queries be $a_1=\mu_1(x_1)$, ..., $a_n=\mu_n(x_n)$.
Since $x = x_1 \cup \ldots \cup x_n$ we can conclude that $a = a_1 \cup \ldots \cup a_n$.

Theorem: (containment under OR-semantics)
Let Q1 and Q2 be two graph queries.
$Q1 \subseteq Q2$ iff there is a C.M from Q2 to Q1 under OR-semantics.

Proof: very similar to the *classical* one for conjunctive queries as in [Ull89], which shows that both (1) if $(Q2)\varphi=Q1$ then$Q1\subseteq Q2$ and (2) if $Q1\subseteq Q2$ then $(Q2)\varphi=Q1$ hold.
Part (1) is proven by decomposing Q1 and Q2 to Γ_{Q1} and Γ_{Q2}.
Using the proposition:
$\text{Ans-}\smile(Q1) = \cup\ \text{Ans-}\smile(Qi)\ \forall Qi\in\Gamma_{Q1}$ and $\text{Ans-}\smile(Q2) = \cup\ \text{Ans-}\smile(Qi)\ \forall Qi\in\Gamma_{Q2}$.
Since there is a c.m. from Q2 to Q1, then $\Gamma_{Q1} \subseteq \Gamma_{Q2}$, which means that
$(\cup\text{Ans-}\smile(Qi)\ \forall Qi\in\Gamma_{Q1}) \subseteq (\cup\text{Ans-}\smile(Qi)\ \forall Qi\in\Gamma_{Q2})$.
Thus, $\text{Ans-}\smile(Q1) \subseteq \text{Ans-}\smile(Q2)$ and $Q1 \subseteq Q2$.
Part (2) is proven by decomposing Q1 and Q2 to Γ_{Q1} and Γ_{Q2}.
Using the proposition:
$\text{Ans-}\smile(Q1) = \cup\ \text{Ans-}\smile(Qi)\ \forall Qi\in\Gamma_{Q1}$ and $\text{Ans-}\smile(Q2) = \cup\ \text{Ans-}\smile(Qi)\ \forall Qi\in\Gamma_{Q2}$.
If $Q1\subseteq Q2$ then $\text{Ans-}\smile(Q1) \subseteq \text{Ans-}\smile(Q2)$ and also
$(\cup\ \text{Ans-}\smile(Qi)\ \forall Qi\in\Gamma_{Q1}) \subseteq (\cup\ \text{Ans-}\smile(Qi)\ \forall Qi\in\Gamma_{Q2})$, and using the corollary above: for each $Qi\in\Gamma_{Q1}$ and for each $Qi\in\Gamma_{Q2}$, $(Qi2)\varphi=Qi1$, so $(Q2)\varphi=Q1$.

6 Criticism and Future Research

This section describes the strengths and weaknesses of the GQL proposed in [KNS99], and discusses which features of the model should be further studied and/or improved. The following criticism is based on examining the basic features of any query language for SSD, comparing the GQL with other current query languages for SSD and verifying that the objectives stated in the Introduction and in the Abstract were actually met. In addition, some suggestions will be given in order to improve the proposed GQL.

One of the main advantages of having a graph-oriented query language is its support for incomplete answers. Partial answers are achieved through the use of weak search and filter constraints and the various semantics that allow different levels of strictness in the way that database elements are bound to the graph variables. As mentioned in section 1, Introduction, any query language for SSD should take into account partial answers to queries since SSD is heterogeneous, it has no fixed schema and many of its elements may be optional.

To illustrate how the proposed GQL supports better incomplete answers compared to XML-QL for example, consider the following query. Assume for simplicity that there is a books database, in SSD format, with a <BOOK> elements that can have two sub elements <TITLE> and <PRICE>. Assume also that we want all book titles and, where available, book prices too. The following is a straightforward solution in XML-QL:

```
where  <BOOK>
         <TITLE> $T </TITLE>
         <PRICE> $P </PRICE>
         </BOOK> in "www.ex-books.xml",
construct <RESULT>
               <BOOKTITLE> $T </BOOKTITLE>
               <BOOKPRICE> $P </BOOKPRICE>
         </RESULT>
```

Such a query in XML-QL will result only complete answers since the pattern insists that <PRICE> be present. Thus, books without a price are not reported. XML-QL uses nested queries format to deal with optional elements, and the correct solution would be:

```
where  <BOOK> $B  </BOOK> in "www.ex-books.xml",
         <TITLE>  $T </TITLE> in $B
construct <RESULT>
               <BOOKTITLE> $T </BOOKTITLE>
               where <PRICE> $P </PRICE> in $B
               construct <BOOKPRICE> $P </BOOKPRICE>
           </RESULT>
```

Obviously, the way XML-QL deals with incomplete answers, through nested queries, is quite cumbersome even for this simple query. It is difficult to read and maintain such a format and it seems its too complicated for the given simple query.

The graph-oriented query in figure 3 is much more clear, simple and intuitive. Since it is a tree query it gives the same answer under the different semantics.
GQL uses algorithms that assume that in most cases answer are incomplete because data is heterogeneous.

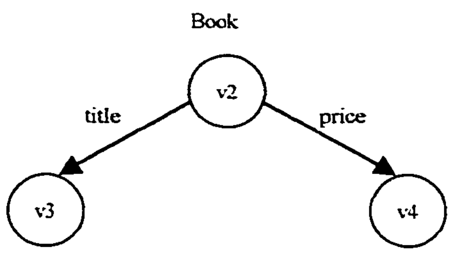


Figure 4: a query that asks for books' titles and prices

Another advantage related to the proposed GQL, which simplifies the query format, understanding and manipulation, is the case of union such as the query in figure 2 under OR-semantics. In the current SSD query languages it is impossible to write a union operation in one simple query. In fact, in the XML-QL proposal in [DFFLS98] and [ABS99] this is not possible at all. In a graph-oriented QL union is achieved easily using the OR and WEAK semantics.

An additional strength of the proposed GQL is query optimization. As mentioned in section 2.2, one of the important features of a query language for SSD is a precise and simple semantics that enables query optimization. Unlike other current QLs for SSD, the suggested GQL sets the foundation for query optimization by defining and proving the homomorphism theorem for graph-oriented queries under the different semantics (see section 5). Optimization notions are difficult to define in languages like XML-QL since they use syntax similar to SQL, which is designed relational database not SSD. In this respect, GQL is more appropriate for a graph-oriented database in the same way that SQL is appropriate for a relational-oriented database.

The main advantage of the graph-oriented model could be also viewed as a weakness. The proposed GQL although more natural and intuitive for graph databases, stands in contradiction to the principle idea in databases of three levels of abstraction.
Actually, this constitutes a problem for all SSD query languages, which do not distinguish between logical and physical levels of abstraction. However, this distinction is further blurred with a GQL since both the query language and the database rely on graph mechanisms for storing the data and exploring it. The difficulty in designing a modular three level architecture model for SSD emanates from the fact that the middle level (logical/conceptual level) requires the existence of a schema that allows for query manipulation regardless of the actual data.

Another point, related to the three-level architecture in database systems, is the need for a mathematical foundation for query manipulation and transformation. One of the reasons that the relational model is so popular is that it achieves this kind of abstraction. Query transformation and optimization is based on relational algebra and set theory. Therefore, SQL has a strong theoretical foundation, taken from mathematics, which gives it its expressive power as a query language. On the other hand, GQL is not based on such theoretical foundations (maybe graph theory from mathematics) that abstract it from the database.

Section 2.2, SSD query languages, discusses the features that any SSD QL should have.
The main feature missing in the GQL proposal is regular path expressions. GQL, in its current format, doesn't allow reaching an arbitrary depth in the database graph. In LOREL, for example, we can write in the "from" clause the expression "university._*.name", and in the "where" clause the expression name="Cohen", in order to find a path of any depth that starts with "university" as the root and ends with the name "Cohen". This is not possible in the proposed GQL because queries have fixed depth. Regular path expressions must be integrated into any GQL so that it can be used for SSD. One way this can be achieved is discussed shortly.

Structure consciousness is another feature of a QL for SSD, mentioned in section 2.2. It refers to the ability of the QL to infer and to exploit the database schema. The proposed GQL does not discuss the reality that in many cases the SSD has a general or a basic structure that is distinguished from the optional elements. SSD, as the name suggests, is partly structured and not completely non-structured and any SSD QL should address this issue. XML-QL, for example, provides a simple and useful structure conscious mechanism through the optional use of DTD.

An additional feature mentioned in section 2.2 is expressive power, the ability of the QL to support of all kinds of SQL or RA operations. Since most databases today are relational, a SSD QL would have to be compatible with relational database systems. This implies that when the input to the GQL processor is relational database (and not SSD), the correct result will be produced. Clearly, this is not the case in the proposed GQL since its algorithms search graphs and not tables. This, however, can be solved by converting the tables into graphs. But even then the GQL processor cannot perform all SQL equivalent operation, most notably the join operation.

Other structure conscious QLs, such as LOREL, do support join operations. The LOREL query on the next page computes the natural join of r1 and r2 (the common attribute is b) and projects on attributes a and c. The relations are represented by the following SSD instance:

```
{ r1: { row: {a:1, b:2}, row: {a:1, b:3} },
  r2: { row: {b:2, c:4}, row: {b:2, c:3} } }

% Query q-join
select  a:A, c:C
from    r1.row X,
        r2.row Y,
        X.a A, X.b B, Y.b B', Y.c C
where   B = B'
```

The above join operation is not possible in GQL since it has no reference to metadata. Future research is required to suggest new ways in which two SSD graphs can be joined. The main difficulty stems from the lack of schema on which join operations are based.

Unlike the proposed GQL, LOREL allows for both label and path variables. In a GQL, node variables are allowed, but not edge variables. One way to incorporate regular expressions into a GQL would be to allow edge variables to exist. These could be combined with wildcards in order to reach an arbitrary depth in the database graph. Figure 4 shows such an example in which the depth of the graph-oriented query is not fixed. The star denotes one or more occurrences of an edge that leads to a person node p that has a first and a last name. Clearly, the algorithms would have to be modified accordingly.

Finally, although query containment is discussed in [KNS99], which lays the foundation for GQL optimization, the paper does not discuss graph operations by which query manipulation and transformation can take place. As mentioned earlier, such graph operations would require a sound theoretical background.

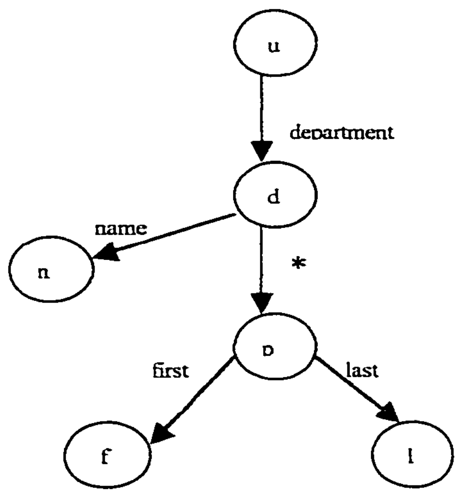


Figure 5: a query asking for all staff members at any depth.

Part 2: Software Document

Part 2 documents the software development process and includes the main stages that are employed in most software system developments according to well-known software engineering principles [Som97]. It starts with section 7, Software Requirement Specifications, which describes, in high-level terms, the services that the system provides and the constraints under which it operates. Section 8, Software Design, specifies various system components and their relations, using different models of different level of abstraction. Section 9, Verification and Validation, describes the software testing process and ensures that it meets the user requirements and the specifications laid out in section 7. Finally, section 10 concludes the GQL proposal from a more practical standpoint, and revisits the theoretical results in Part 1, using the empirical results from part 2.

7 Software Requirement Specifications

7.1 General Description

7.1.1 Purpose

The main purpose of the software product is to implement the graph-oriented query language proposed in [KNS99], *Queries with Incomplete Answers over Semistructured Data.* The program will enable users to query the university database described earlier in section 2, using a graph query format. Users can select the graph nodes to be matched to the database graph and enter some filter constraints in order to get a more specific query result. The program is publicly available on the World Wide Web at www.cs.concordia.ca/elkada/project).

7.1.2 Context

The software product is independent of other related resources. It operates in the context of the World Wide Web. The database that users can query will be provided on a separate web page (www.cs.concordia.ca/elkada/project/db.html). User parameters are sent to the server where query processing takes place and the query results are sent back to the user in a table form, with possibly empty cells, to emphasize the query's language support for incomplete answers.

7.1.3 Users

The program would be mostly used for academic purposes. People interested in SSD and their query languages would be able to experiment using the GQL tool and analyze its contribution to this domain of study. Thus, most users are computer literate and have both domain-specific and system knowledge.

7.2 Glossary

Semi- structured data (SSD) – "schemaless" data model, in which there is no separate global description of the structure or the type of the data. The data consist of label-value pairs such that the label is the attribute/description of the value. It is a heterogeneous model in which the same data content is represented in various structures (as graphs).

Graph-oriented query language – a query language for SSD in which the query is represented by a graph, very much like the database itself (see section 3.1)

Query graph – graph in which the nodes are variables that represent data values to be matched to the database graph. The edges in the query graph, which connect the nodes, are also to be matched to the database graph (see section 3.1).

Query Semantics – the level of strictness in which a query graph can be matched to the database graph. These are described in part 1 section 3.2

Matching – correspondence between query nodes/edges and database nodes/edges. The matching result depend on the semantics used.

Search Phase – the first step of graph query processing in which the set of maximal matching is found depending on semantics used.

Filter Phase – the second step of graph query processing in which the solution set is obtained (and from which output variables are given as a final answer).

Filter Constraints – the filtering parameters of the second phase. For examples, name = "Cohen", seniority > 18, t! (the teacher node exists) etc.

Incomplete Answer – answer in which not all the nodes in the graph query are matched. This is an important feature for the heterogeneous SSD model. This achieved using different semantics in both the search and the filter phases.

Server – the host on which query processing takes place. Invisible to the user, user submits a query request, which is sent to the server which, in turn, searches the database graph (located persistently at the server).

Client – user host from which the query parameters are submitted. Many clients can send query requests to the server simultaneously.

7.3 System Model

The following data flow diagram charts how data flows through a sequence of processing steps. The user or the program carries out these steps, denoted by rounded rectangles. The client process performs some of these steps while the server process performs others. Also, the data stores, denoted by rectangles, represent input/output files that the server program accesses. The server program initially retrieves the database files in order to find the given query matchings and finally the program writes the query answer to a log file. All these files are located at the server host/machine. The diagram is simply the first of many steps. It provides a high level abstraction for the system and it hides many of the details that are discussed in section 8, Software Design.

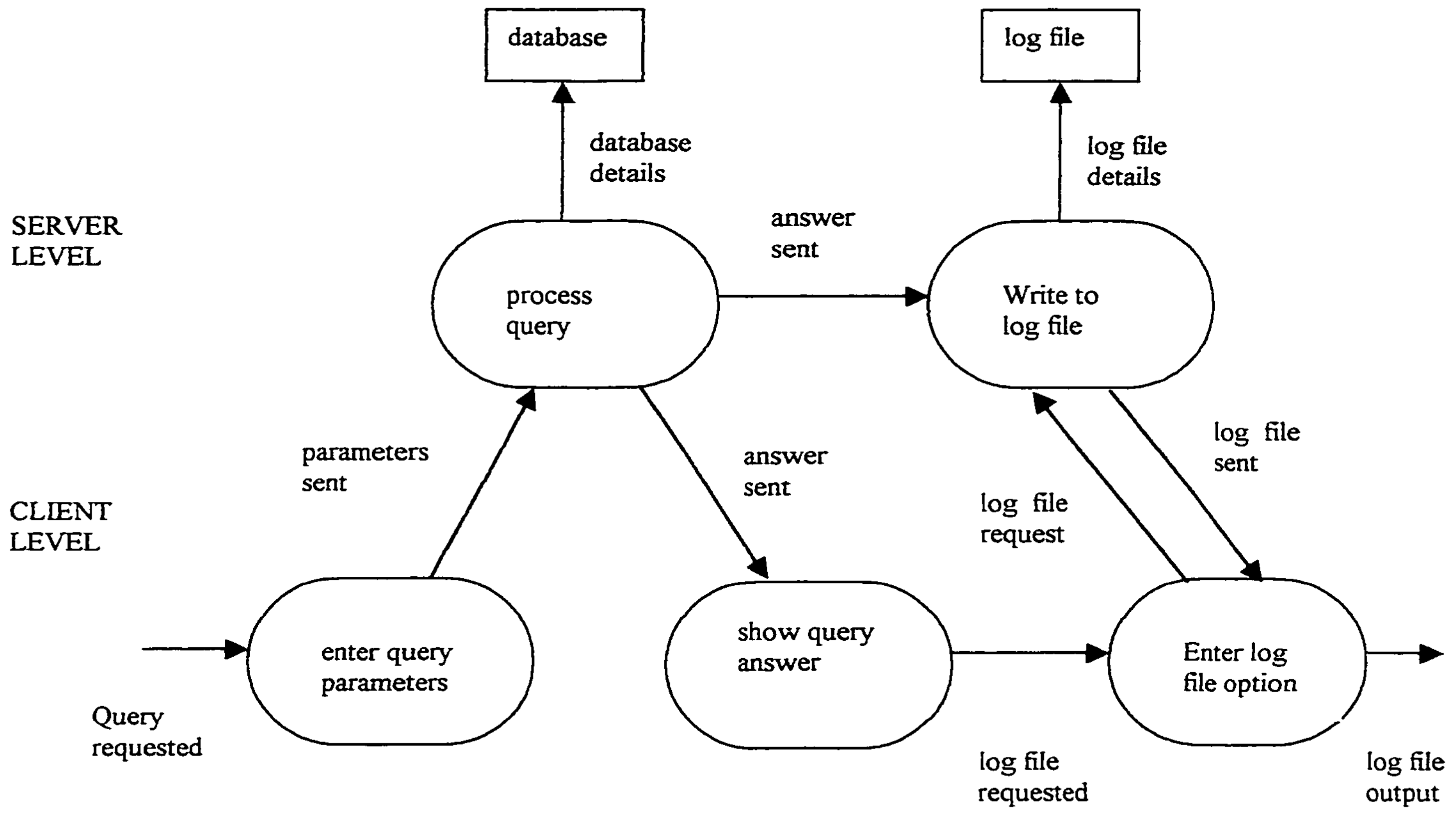


Figure 6: High-Level data flow diagram for GQL processing

7.4 Functional Requirements

R1: The system should enable the user to query the database using a graph query.
Rationale: the database is also given as a graph and the goal is to evaluate and analyze the proposed graph-oriented query language discussed in Part 1.

R1.1 The system should enable the user to select graph nodes in the query.
Rational: different query types allow analyzing different features.

R1.2 The system should enable the user to select the semantics under which the query is evaluated.
Rational: different semantics result in different answers to the same query.

R1.2.1 The system should enable the user to select no semantics.
Rational: Semantics is meaningless for tree queries.

R1.2.2 The system should enable the user to select OR semantics.
Rational: This semantics results in maximal answers.

R1.2.3 The system should enable users to select AND semantics.
Rational: The user might ask for specific search constraints.

R1.2.4 The system should enable users to select WEAK semantics.
Rational: This semantics enables special query meaning.

R1.3 The system should enable the user to enter filter constraints.
Rational: Users should be able to narrow down their search criteria.

R1.3.1 The system should enable the user to select an existence constraints.
Rational: The user should be able to enforce node variables.

R1.3.2 The system should enable the user to select a comparison constraints.
Rational: Users should be able to search for specific information.

R1.4 The system should inform the user of erroneous input.
Rational: Users might have entered invalid input/query parameters.

R1.5 The system should enable users to perform a query without selecting.
Rational: The program should provide for default query parameters.

R2: The system should provide the user with the query answers.
Rationale: the user can then verify the answers against the database.

R2.1: The system should provide users with answers in a table format.

Rationale: The examples in Part 1 of this document are given in a table format. It is clear and easy to understand and to verify.

R2.2: The system should inform the user if query result is empty.
Rationale: The program should provide clear messages for all cases.

R2.3: The system should provide the user with incomplete answers.
Rationale: one of the main features of the proposed graph query is the ability to support incomplete answers. Thus, this aspect has to be tested.

R3: The system should provide users with access to the database graph any time.
Rationale: The user should be able to see the database both before querying (to know what to query) and after query processing (to verify the results).

R4: The system should provide users with the query and its answers in a log file.
Rationale: The user should be able to analyze the query results off-line and compare many different query structures.

R4.1: The system should allow the user to download the log file any time.
Rationale: The user should have access to the answer log file any time.

R5: The system should enable users to restart the program at any time.
Rationale: User should be able to easily recognize the restart/home option and do so at any given time. This feature provides more user control.

R6: The system should enable users to exit the program at any time.
Rationale: User should be able to easily recognize the close/exit option and do so at any given time, which provides some degree of user control .

R7: The system should enable users to modify their graph query parameters.
Rationale: The system should support reversal of action and interface flexibility.

R8: The system should operate with consistent a sequence of actions.
Rationale: Repetitive operations, that results identical action for identical situations, help to achieve system ease-of-use and learnability.

R9: The system should provide users with informative feedback about the system
Rationale: System status gives the user control and helps in system learnability.

R10: The system should provide users with help about the system operation.
Rationale: User should be able to learn how to use the system without human help

R10.1 The help should explain the terminology used in the program.
Rational: Users should be able to understand the domain-specific terms.

R10.2 The help should explain the different options in selecting the query semantics and the related parameters.
Rational: Users should be aware of meaning of search/filter constraints.

R10.3 The help should provide a tutorial for performing a query.
Rational: Users should be able to understand the query input and output.

R11 The system should provide ways to contact the application developers.
Rationale: User might have questions or feedback concerning the application.

R12 The system should enable users to give feedback about the program.
Rationale: User feedback is needed to better examine, analyze and compare the graph-oriented query language with other query languages for SSD.

7.5 Non Functional Requirements

1.0 The program should be available on the World Wide Web.
Rationale: maximum feedback is needed in order to examine this domain-specific research area the program is exploring (query languages for semi-structured data)

2.0 The database graph should be stored persistently.
Rationale: modification and updates to the databases would be easily possible.

3.0 Experienced users should be able to use the system after 30min of training.
Rationale: system that is easy to use will attract more users, so more people can join in the discussion of the issues the program is dealing with.

4.0 After 30min training, the average number of errors made by an experienced user should not exceed 2 per hour.
Rationale: this usability goal proves that the system is easy to learn and use.

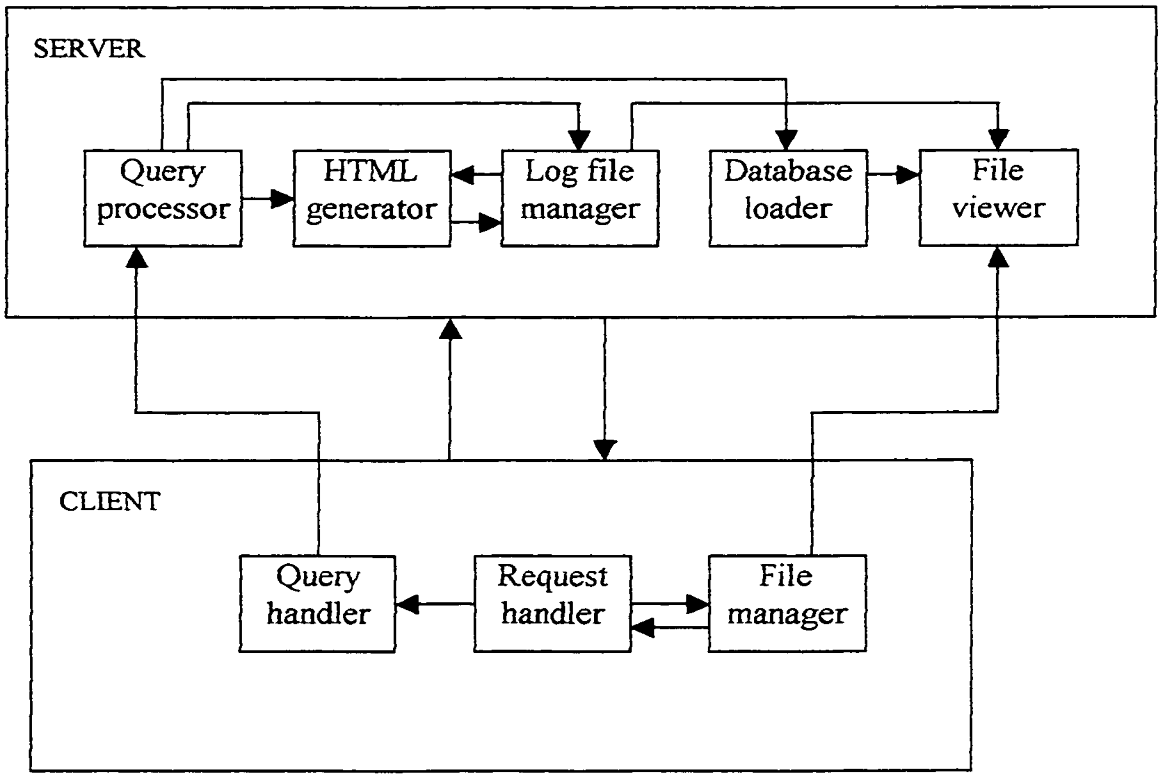
5.0 Time to restart the program after failure should not exceed 1 min.
Rationale: users might abandon the program altogether.

8 Software Design

This section is based on the requirement specifications described in the previous section and it serves as a basis for the next step of the actual implementation and testing. The design process includes several stages that are detailed in the following sections. First, general system architecture is given, in order to structure and organize the user requirements. This is achieved mainly by dividing the system into several sub systems. Then, a more detailed object-oriented model is designed which includes the main data structures and algorithms at the core of the program. A separate section is then dedicated to user interface design, which uses some heuristics that promote system learnability and ease of use. Finally, the last section discusses how the SSD database is stored. As it turns out, this issue remains an open one in the SSD study.

8.1 System Architecture

As mentioned earlier, the system can be decomposed into two major interacting entities: the client and the server. These constitute the main sub-systems. Each can be further decomposed as the following block diagram demonstrates. The following architectural block diagram represents an overview of the system structure. Each box represents a different sub system and in itself can include other boxes or sub-systems. The arrows show that data and/or control are passes from one sub system to another in the direction of the arrow.



[illegible]

The previous structural model did not, and should not, include control information. The following control model, a call return diagram, is concerned with the control flow between different sub systems. This control model uses a centralized control approach in which one sub system is designated as the system controller and has the responsibility of managing the execution of other sub systems. It may devolve control to other sub systems but it will expect to have the control returned to it. Each box represents a sub system that will be eventually implemented as a routine. Therefore, when a sub system requests a service from another one, it does so by calling the associated routine.

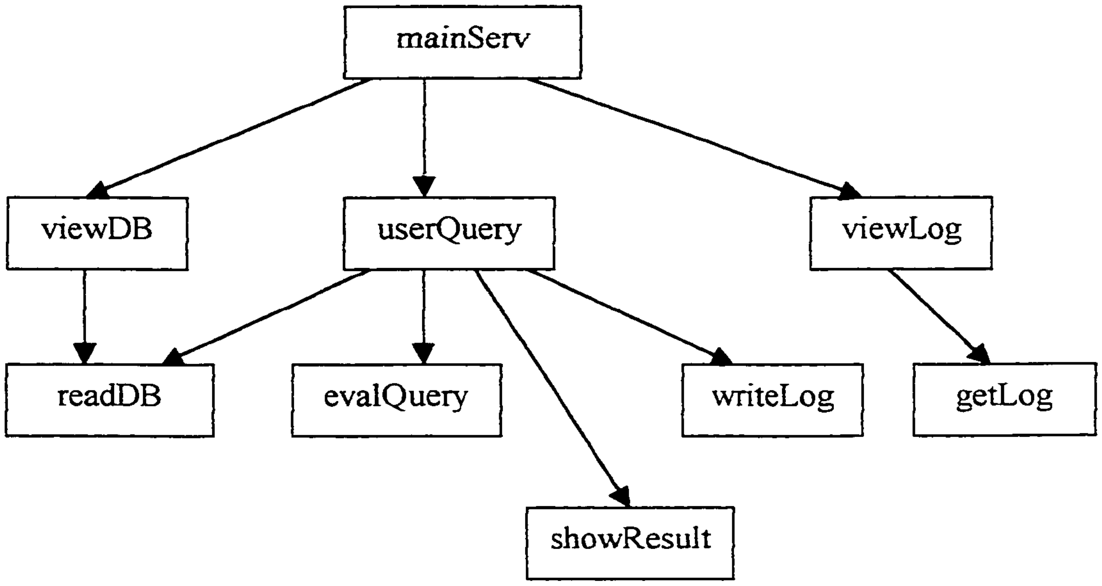


Figure 8: Call Return diagram for modeling system control flow

Each of the above routines belonging to a sub system can be further refined into sub routines. In particular, evalQuery is a complex routine that is implemented by the query processor sub system, and makes several other routine calls. It is not discussed at this point, since only an abstract high-level decomposition is given here. Instead, it is described in the next section, after specifying the system objects, using a service usage diagram.

8.2 Object-oriented Model

The design strategy used in modeling the system is object-oriented design (OOD). The OOD approach, whereby the identified requirements are implemented, views the system as a set of interacting objects, rather than as a set of functions sharing a global state. In OOD, each object is an independent entity with its own private state, data and operations or services, some of which are available to other objects.

Object-oriented systems are easy to maintain, as objects are independent. Changing the implementation of an object or adding to it new services should not affect other system objects. Therefore, OOD results in a more maintainable and loosely coupled system. By encapsulating the data and the operation allowed to manipulate it, objects become reusable components. This normally reduces implementation and testing costs. Encapsulation also entails information hiding, a principle idea in OOD according to which objects hide their internal data and operation from other objects. Again, this enhances maintainability, reusability and reduces software errors.

The object-oriented system modeling is done in several stages. First, the different classes are identified along with their attributes and operations. Second, these classes are organized into an aggregation hierarchy, which shows how objects are part-of other objects using association relationships and multiplicity information that specify the number of instances that participate in the relationship. Finally, object-use diagrams are constructed to show how objects of these classes interact for some main scenarios.
The following sections describe some data structure and algorithms that implement the above OOD.

8.2.1 Class Hierarchy

The major complex sub system that implements the graph-oriented query language is the query processor sub system. Since the graph-oriented query language was built upon the idea that the database has a graph format, one obvious class that could be used to initialize both the database object and the query object, is the *graph* class. A graph consists of many *node* objects and in particular the root node. The graph class uses two auxiliary classes, which are used in the query evaluation process: a *queue* and a *stack* classes. The result of the query evaluation process is matching/*pairs lists*, in which each query node is matched to a database node (or to null if corresponding database node does not exist). These relations are described in the following class diagram in figure 2.4.

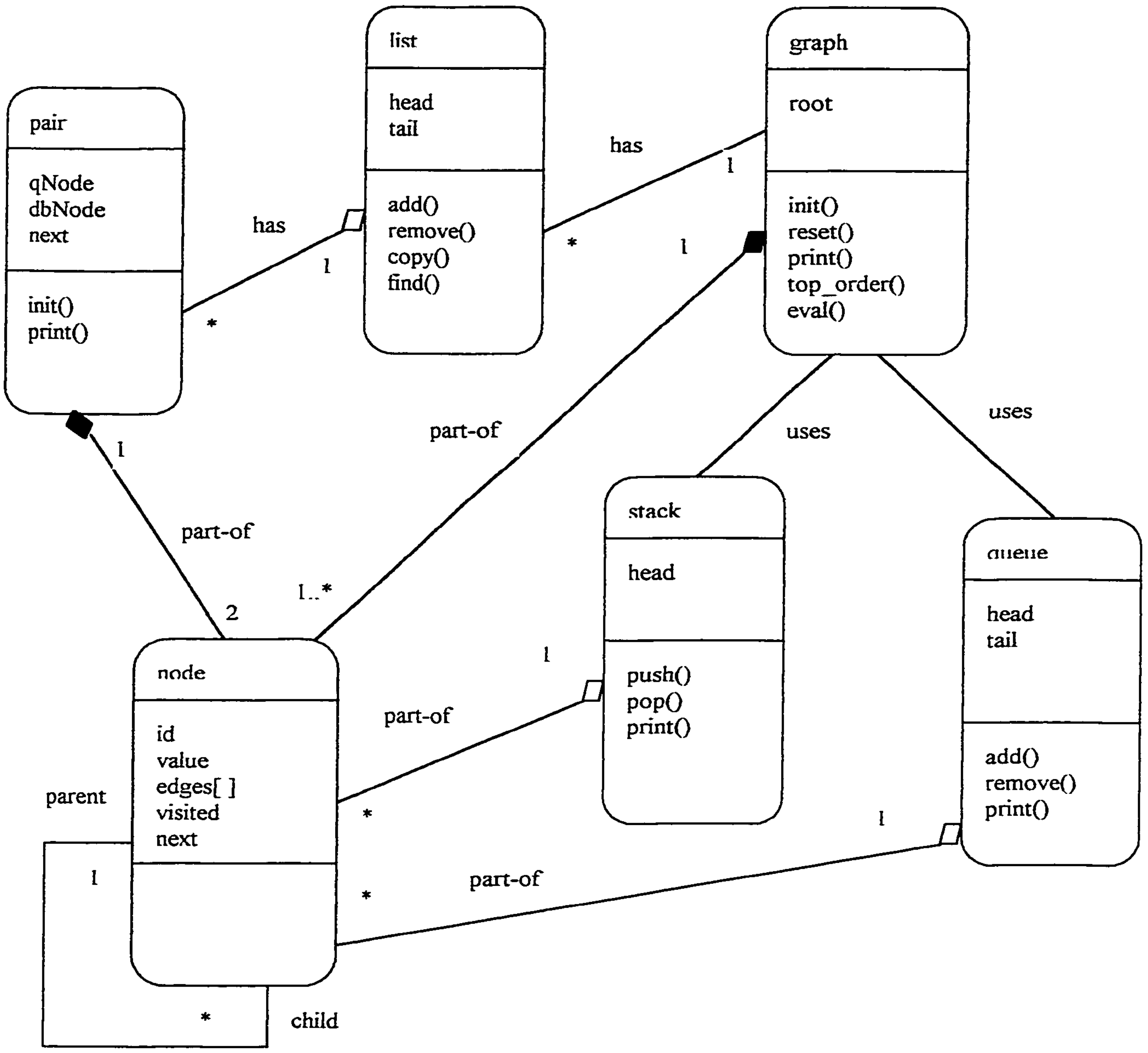


Figure 9: Class diagram for modeling the query processor sub system

8.2.2 Object Interactions

The following sequence diagram shows interactions between objects from a temporal standpoint. It focuses on message chronology as time progresses. The message type used here is synchronous broadcast for which the transmitter blocks and waits until the called object has finished processing the message. Objects are represented by rectangles and the messages exchanged are represented by horizontal arrows drawn from the sender to the recipient. The vertical rectangle represents object activation, that is, the time during which an object performs an action. In that case there is an implicit return at the end of the execution of the operation.

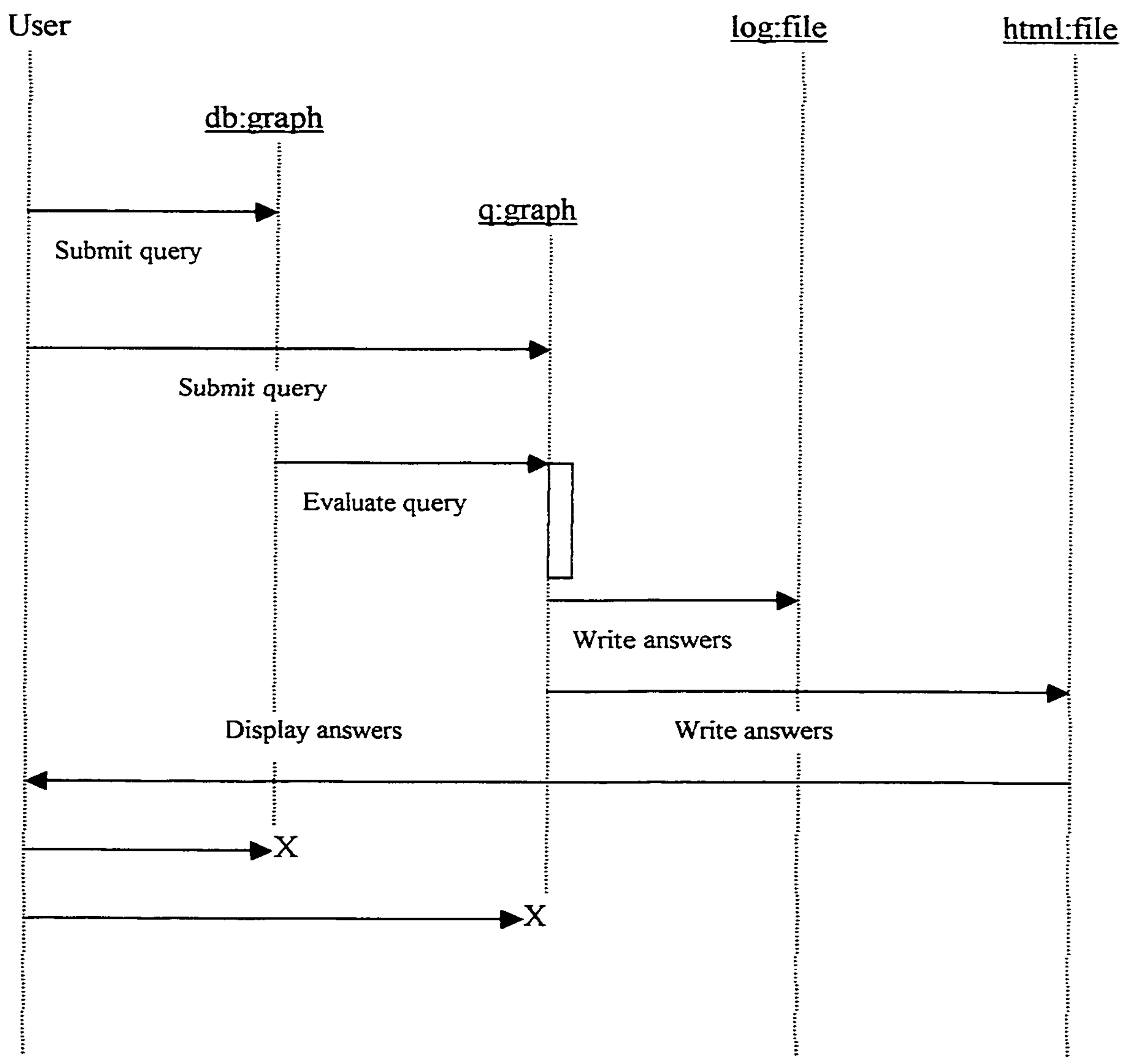


Figure 10: Sequence diagram for modeling object interaction

8.3 Data structures

The following are the data structure prototypes to be used in the implementation phase. Some of these structures are encapsulated in others in order to achieve information hiding and loosely coupled program, easy to maintain and modify.

```
class node{
    // member functions
    node();
    node(int id1, char* value1 = "");
    void setEdge(char* str, node* n);
    void visitAll();
    void show();
    // data members
    int id;
    char value[MAX_STR];                      // value in case of a leaf node
    char edgeName[MAX_EDGE][MAX_STR]; // array of outgoing edge labels
     node* edgePtr[MAX_EDGE];                 // pointers correspond to
edgeName
    bool visited;       // used in matchings functions
    node* pNext;        // used in stack and queue structures
}

class list{ // list of pairs

        class pair{ // pair of matching
                // member functions
                 pair();
                void print();
                // data members
                node* qNode;
                node* dbNode;
                pair* pNext;
        }
    // member functions
    list();
    void add(node* qNode1, node* dbNode1 = 0);
    void remove();
    void copy(list* l);
    pair* find(int id);
    bool empty() { return !pHead; };
    void print();
    // data member
    pair* pHead;
    pair* pTail;
}
```

```
class graph {

        // stack used for searching a node in a graph //
        class stack{
        stack() { pHead = 0; };
                void push(node* n);
                node* pop();
                void print();
                // data member
                node* pHead;
        };

        // queue used in topological sort //
        class queue{
                queue() { pHead = 0; pTail = 0; };
                void add(node* n);
                node* remove();
                bool empty() { return !pHead; };
                void print();
                // data member
                node* pHead;
                node* pTail;
        };

        // member functions
        graph();
        node* find(int id); // used in constructing the db graph
        void topOrder(node** arr);
        void reset();
        void print();
        void eval(graph* q, list** l);
        void evalOr(graph* q, list** l);
        void evalWeak(graph* q, list** l);
        void evalAnd(graph* q, list** l);
        void evalStrong(graph* q, list** l);
        // data members
        node* root;
};
```

8.4 Algorithms

The algorithm which is the main focus of the program, and the most complex, is the matching algorithm eval(), which evaluates the given query graph against the db graph.
After eval() is called the maximal matchings are reduced to the matchings under a given semantics. Semantics is meaningful only for dag queries as explained in section 4.1.1, Tree Queries. eval() returns an array of pointers to lists, where each list contains pairs of matching (binding of a query node to a database node).

The eval() algorithm calls first the toplogicalOrder() function, described below,in order to arrange the query nodes in a topological sort before searching the database graph.

```
void topologicalOrder(node** arr){

  queue openQ, closeQ;
  node* ptr;

  reset();
  openQ.add(root);

  while (!openQ.empty()){
    ptr = openQ.remove();
    closeQ.add(ptr);
    for(int i = 0; ptr -> edgePtr[i] && i < MAX_EDGE; i++)
        if (!ptr -> edgePtr[i] -> visited){
            openQ.add(ptr -> edgePtr[i]);
            ptr -> edgePtr[i] -> visited = true;
        }
  }// end while

  // reverse the closeQ order to get the topological order
  for(int i = 0; i < MAX_NODE; i++) arr[i] = 0;
  for(int i = 0; !closeQ.empty() && i < MAX_NODE; i++) arr[i] =
closeQ.remove();
}
```

```
void eval(graph* q, list** match){

  node* order[MAX_NODE]; // the query nodes in topologigal order
  int nextMatch = 0;  // next position in match[] to add a newly created list
  int lastMatch;  // last position in match[] before new lists were added
  pair* pairPtr;
  int edgeNum; // num of child edges with the same name
  bool found;  // indicates if order[n]'s child edge was found

  toplogicalOrder(order);  // arrange query nodes in a topological order
  for (int i = 0; i < MAX_LIST; i++) match[i] = 0;

  // assign the db root to the query root
  list* l = new list();
  l -> add(q -> root, root);
  match[nextMatch++] = l;

  // for each node n in the topological order
  for(int n = 0; order[n] && n < MAX_NODE; n++){
        // for each edge e of the above n
        for(int e = 0; order[n]->edgePtr[e] && e < MAX_EDGE; e++){
              // for each matching m in the matching lists
              for(int m = 0; m < lastMatch; m++) {
                 pairPtr = match[m] -> find(order[n] -> id);
                          if (pairPtr){
                             found = false;
                   // for each child edge of the of n's db node
                             for(int c = 0;      pairPtr->dbNode->edgeName[c]; c++)
                                 if(!strcmp(pairPtr->dbNode, order[n])){
                                          found = true;
                                          edgeNum++;
                                          if (edgeNum == 1)
                                             match[m] -> add(order[n], pairPtr->dbNode);
                                          else { // more than one edge with same name
                                            list* newList = new list(*copyList);
                                           newList -> add(order[n], pairPtr->dbNode);
                                           match[nextMatch++] = newList;
                                           }
                              } // end if (strcmp())
                            if (!found) match[m] -> add(order[n]->edgePtr[e], 0);
                          } //if (pairPtr)
              } //end for(int m)
        } //end for(int e)
   } // end for(int n)
}
```

8.5 UI Design

This section analyzes the user interface (UI) requirements with respect to the target user. The GQL processor will be available on the web and thus, the UI has to generally fit the look-and-feel of this environment. However, the type UI elements needed may vary depending on the target user and the UI requirements. Therefore, first, the user profile is determined. Knowing the type of user who is going to use the application is important in order to design an effective user interface which is easy to learn and to use. Second, the UI requirements are defined, which describe the UI elements needed from a functionality point of view. Finally, in order to fine-tune the UI design, some usability heuristics are provided. In section 9, Verification and Validation, a usability test is proposed in order to verify that indeed the UI is easy to use and to learn.

8.5.1 User Profile

User Characteristics: The GQL application will be used by users in the academic field, users that are interested in the study of query languages for SSD.

User's characteristics:
- Motivation: high.
- Attitude: positive.
- Cognitive style: analytical and intuitive.

Design goals:
- Easy to use: consistent interfaces "look and feel".
- Easy to learn: similar functionality for similar interface elements.
- Powerful: include many design features with many options.

Physical Environment: private with low level of noise. Since the hardware is simply a PC, there are no design implications as far as input and output devices, glare control and the physical location in which the computer is placed.

Knowledge and experience factors: the web site is useful for a small range of users who are interested in the study of databases, and in particular, SSD. Clearly, most users are computer literate with a lot of experience with Internet applications.

Knowledge and experience characteristics:
- Education: university level.
- Native Language: English.
- Computer Literacy: usually high.
- Application Experience: high level.
- System Experience: moderate level.

Implications for UI design:
- Informative messages: no need for extensive status and help messages
- Error prevention: simple help option with small tutorial will suffice.
- Efficient commands: database search should easily be performed.
- Familiar interface: similar to other web based applications

Task Factors – the UI considerations related to operating the GQL web site.

Task characteristics:
- Frequency of use: low frequency, less than two days a month.
- Primary training: none, neither needed nor required.
- System use: Discretionary. It is the user's decision to use the product.

UI design goals:
- Easy to learn and remember: short term memory rule (5-7 items/interface)
- Simple Interface: no fancy graphics.

8.5.2 UI Requirement

The following are requirements specific to user interface functionalities. Some appear in the higher-level requirements specified in section 7.4.

1.0 The system will have a main menu from which the user can select next step.
2.0 The system will have a welcome screen, which explains the general purpose of the application and how to start using it.
3.0 The system will have a query interface with a button to submit the query.
4.0 The system will enable the user to select search criteria (semantics) from a pre-defined list in order to minimize user errors
5.0 The system will enable the user to select filter criteria (constraints) from a pre-defined list in order to minimize user errors
6.0 The system will have a help screen
  6.1 The system will have a FAQ list (Frequently Asked Questions)
  6.2 The system will have a step-by-step tutorial
7.0 The system will enable the user to restart the application at any time.
8.0 The system will clearly display the query results in a table format
9.0 The system will enable the user on-line and off-line view of query results.

8.5.3 Interface Guidelines

In order to have a well-designed user interface it is important to take into account both the user profile discussed in section 8.5.1 and the UI design principles discussed below. The application's interface has to meet some basic usability factors: (1) effective: users are able to perform the identified tasks in section 7.4, (2) efficient: users can perform tasks easily and in a timely manner, (3) fun: interfaces are natural, intuitive, predictable and let the user be in control (4) safe: interfaces provide reversal of actions and eliminate errors and confusion.

Since the application is meant to be used as a web site over the Internet, the user interface design also considers differences between Web and GUI design such as the Back button being part of the browser and not the application, implementation of link as navigation tools, minimizing graphic images due to network delays and consistency with web site format in order to enhance familiarity and predictability. Other UI guidelines include the following:

Consistency - the different screens/pages should have the action buttons in the same location, shape, color and font. The command operates in a consistent manner and the same sequence of action is used in similar situations.
Familiarity - the interface design should allow users to build on previous knowledge such as using hyperlinks and familiar buttons.
Simplicity - the user interface should be simple and straightforward. Too many graphics and advanced options would distract users from performing their frequent tasks. The basic functions should be immediately apparent. The STM (short term memory) rule of 7 ± 2 items should be respected.
User Control - the system should allow the user to freely navigate through the web site pages and search for the desired information. A feeling of stability is also established using the Home button, which allows the user to restart the application at any time.
Encouragement - the interface should be designed to make user actions predictable and reversible. Users are encouraged to initiate actions and feel confident to explore them, knowing that they can try an action, see the result and undo the action if needed.
Versatility - the system should support alternate interaction techniques allowing users to choose between input devices (mouse or keyboard) and allowing shortcuts, abbreviations, accelerator keys etc.
Error Handling - the system should be designed to avoid user errors. When such errors happen, it should provide clear and simple error messages. The system also provides on-line help in order to eliminate error and confusion.
Informative Feedback - the system should offer feedback, using messages. For every action there should be feedback, especially for infrequent critical actions.

8.6 Database Storage

The SSD database is stored in separate text files, each corresponding to a database edge. Each file is a binary relation as explained in section 3.1, Syntax. Thus, each file contains a list of pair of nodes representing the two nodes that the edge connects. The first node in each pair is the source node and the second one is the destination node. If the second destination node is a leaf node, a value corresponding to that leaf node might appear next to it. For example, the file "name.txt" for the database shown in section 2.1 includes a pair such as {3 6 chemistry} meaning that there is an edge called "name" from node 3 to node 6 and since node 6 is a leaf node it has a value "chemistry".

9 Verification and Validation

Usability evaluation methods (system testing) vary, depending on the system characteristics, the target user and the available human resources. This section presents the evaluation process of the GQL processor. It is based on the user profile and the system requirements presented earlier. After the evaluation objectives are discussed, the evaluation procedure is set up. Before the proposed evaluation takes place it is recommended to verify the requirement specifications from section 7.4 as a preliminary step.

The usability evaluation procedure, designed and detailed in this section, is based on both actual user testing and expert inspection (heuristic evaluation). It seems that combining empirical tests and inspections achieves the best results in detecting usability defects since it involves both actual performance of end-users and the expertise of usability specialists [NM94]. These two usability techniques complement one another and work well together in terms of detecting usability defects. The reason is that heuristic evaluation, which is done by an expert, better detects major/high-level problems with the application whereas testing normally better detects minor or low-level problems.

9.1 Evaluation objectives

Quantitative Goals

- Percentage of tasks completed correctly without assistance should be at least 75%.
- Percentage of tasks completed correctly the first time should be at least 75%.
- Time to perform a query successfully after 20 min should be less than 2 min.
- Number of errors should be 2 or less per task.
- Average number of help calls should be 2 or less per task.
- Average number of negative comments should be 2 or less per task.
- Time to perform a search effectively should be less than 2 min.

Qualitative Goals

- Achieve a satisfaction rate 3.75 (75%) or greater, on a 5-point satisfaction scale.
- Number of suggestions to improve the system should be 3 or less.
- Percentage of critical errors should be in the range of 0%-10%
- Achieve an evaluator rating 3.75 (75%) or greater, on a 5-point satisfaction scale.

9.2 Evaluation procedure

Setup
The following details the evaluation plan in terms of its structure, participants and evaluation measures. The actual tests, performed by both the evaluators and end-users are presented later. Thus, the GQL usability evaluation process is divided into two main parts:
<u>Expert Evaluation</u> - the evaluator examines the systems interface twice, once with a focus on a flow and once with a focus on individual dialog elements.
<u>End user Testing</u> - end-users are presented with a list of tasks to perform while interacting with the application. A monitor takes notes using a log table. The user tasks have to be completed within a given time.

Measures
This section of the test plan provides an overview of the types of measures that will be collected during the test, both in terms of performance and preference. The precise usability objectives are specified in section 4. However, the following list of measurements is the basis of the Tasks Test and the Satisfaction Questionnaire, and therefore, reinforces the evaluation process.
<u>Performance Measures:</u>
MCT - Max Completion Time - the maximal time for completing a task (min)
SCT - Successful Completion Time - time the user can complete a task (min).
EN - Errors Number - the number of error performed during one task.
RT - Recovery Time - the time it takes the user to recover from an error (min).
MRT - Maximal Recovery Time – the time allowed recovering from error (min).
NIS - Number of Incorrect Selections - the number of times the user selected an incorrect/inappropriate option/selection during one task.
NHC - Number of Help Calls - the number of times monitor/system help needed.
NNC - Number of Negative Comments - the number of times the user has expressed negative comments or mannerism.
Average Completion Time (ACT): factored SCT average of all tasks of one user.
Average Errors Number (AEN): factored EN average of all tasks of one user.
Average Recovery Time (ART): factored RT average of all tasks of one user.
Average Incorrect Selections (AIS): factored NIS average of all tasks of one user.
Average Help Calls (AHC): factored NHC average of all tasks of one user.
Average Negative Comments (ANC): factored NNC average of all tasks of one user.

<u>Preference Measures:</u> (1 to 5 rating scale where 1 is the worst and 5 is the best)
Degree of ease of use.
Degree of ease of learning.
Degree of satisfaction.
Degree of usefulness of the product.
Type of error (Simple, Possible, Critical).

Inspector Evaluation Form

Use the scale below to rate the following statements (circle only one number):

1	2	3	4	5
Strongly disagree	Disagree agree	Sometimes	Agree agree	Strongly

(1) Consistency - the different screens always have the action buttons in the same location, shape, color and font. The command operates in a consistent manner and the same sequence of action is used in similar situations.

1 2 3 4 5

(2) Familiarity - the interface design allows users to build on previous knowledge such as using hyper links, familiar search options and familiar web concepts.

1 2 3 4 5

(3) Simplicity - the user interface is straightforward. There are not too many graphics and advanced options that would distract users from performing their frequent tasks, and the basic functions are immediately apparent. The STM rule of 7 ± 2 item is respected.

1 2 3 4 5

(4) User Control - the system allows the user to freely navigate through the web site and search for the item or information desired. A feeling of stability is also established using the Home button allowing the user to restart the application at any time.

1 2 3 4 5

(5) Encouragement - the interface is designed to make user actions predictable and reversible. Users are encouraged to initiate actions and feel confident to explore knowing that they can try an action, see the result and undo the action if the result is unacceptable.

1 2 3 4 5

(6) Versatility - the system supports alternate interaction techniques by allowing users to choose between input devices (mouse or keyboard) and by allowing shortcuts, abbreviations, accelerator keys etc.

1 2 3 4 5

(7) Error Handling - the system is designed to avoid user errors, and when they happen provide a clear simple error message. The system also provides on-line help in order to eliminate error and confusion.

1 2 3 4 5

(8) Informative Feedback - the system offers feedback using messages and the status line. For every action there should be feedback, especially for infrequent critical actions.

1 2 3 4 5

User Test Form

The following tasks could be given to representative end-user. Each participant gets the same test but the tasks are arranged in different order. Before each task the participant is given various instructions concerning the task such as requirements, scenario, starting time and finishing time. The test monitor records the task related information as in the Monitor Log Table.

Task 1 (Simple Search)
Task code and name: S1, search with no semantics.
Task Description: Find all department names in the university.

Task 2 (Advanced Search)
Task code and name: S2, search using AND semantics.
Task Description: Find staff members that are both course teachers and lab instructors.

Task 3 (Simple View)
Task code/name: M1, view query results.
Task Description: After a given search is performed view the log file.

Task 4 (Advanced View)
Task code/name: M2, view and verify query results.
Task Description: After search is performed check the results vis-à-vis the database.

Task 5 (Simple Select)
Task code/name: L1, select a query with chairman edge.
Task Description: Find chairmen that also teach course.

Task 6 (Advanced Select)
Task code/name: L2, perform a query whose result is not a leaf node.
Task Description: Find if there are labs in the math department.

Task 7 (Simple Help)
Task code/name: H1, use the help option.
Task Description: Find and verify the tutorial example from the Help option.

Task 8 (Advanced Query)
Task code/name: Q2, teacher node existence with negation.
Task Description: Find all courses that do not have teachers.

Monitor Log Table

The following table contains the information the monitor records while supervising the users. The table contains eight rows, which represent the eight tasks presented to the user (see User test). The table contains ten columns, which represent each task's information. The table heading acronyms are explained below.

Task No	Task Code	MCT	SCT	EN	MRT	RT	NIS	NHC	NNC
1	S1	2 min			1 min				
2	S2	3 min			3 min				
3	M1	2 min			1 min				
4	M2	3 min			3 min				
5	L1	2 min			2 min				
6	L2	3 min			3 min				
7	H1	4 min			4 min				
8	Q2	4 min			4 min				

Table 5: Monitor log table

MCT - Maximal Completion Time - the maximal time to complete a task (min).
SCT - Successful Completion Time - the time a user completes a task (min).
EN - Errors Number - the number of errors performed during one task.
RT - Recovery Time - the time the user recovers from the last error (min).
MRT - Maximal Recovery Time - the maximal time to recover from error (min).
NIS - Number of Incorrect Selections - the number of times the user selected an incorrect/inappropriate option/selection during one task.
NHC - Number of Help Calls - number of times monitor/system help were needed
NNC - Number of Negative Comments - the number of times the user has expressed negative comments or mannerism.

User Satisfaction Questionnaire

Use the satisfaction scale below to rate the following statements (circle only one):

1	2	3	4	5
Strongly disagree	Disagree	Sometimes agree	Agree	Strongly Agree

1. The application is easy to use and I did not have major difficulties.
   1 2 3 4 5
2. The application is enjoyable to use and I had fun using it.
   1 2 3 4 5
3. I did not have many errors performing the tasks.
   1 2 3 4 5
4. The error messages that were given by the system were clear and helpful.
   1 2 3 4 5
5. Once I learned how to use one screen, it was easy to use the other screens.
   1 2 3 4 5
6. The system is generally interesting and I enjoyed exploring on my own.
   1 2 3 4 5
7. The application provides shortcuts and accelerator to speed up operations.
   1 2 3 4 5
8. I could perform all the operations I wanted with few or no errors.
   1 2 3 4 5
9. It was easy to cancel or undo operations and go back to the previous step.
   1 2 3 4 5
10. I would recommend this web site to many of my collegues.
    1 2 3 4 5

Please write in your own words your general impression of the application and how you would improve it in terms of both interface presentation and available operations.

10 Conclusion

The main feature missing from the proposed GQL is regular path expressions and the ability to reach an arbitrary depth in the database graph. One solution, mentioned in section 6 "Criticism and Future Research", is to allow path variable to range over edges. Other query languages for SSD use this concept in an SQL-like format. Incorporating regular expressions into a GQL would require revisiting the suggested algorithms.

The visual query graph is helpful in analyzing the GQL for research purposes, such as for establishing weak and strong matchings and for query optimization. However, from a practical point of view (see www.cs.concordia.ca/~grad/elkada/project) only users with domain-specific knowledge can query a SSD model using such a graph format. Therefore, some mechanisms are needed to internally convert conventional UI search elements (edit boxes, list boxes, radio buttons etc.) to a graph representation.

Finally, the proposed GQL is yet another step in exploring semi-structured query languages. The paper's intention was not to develop a concrete GQL but to explore how such a language allows for incomplete answers. In this respect, the objectives set at the beginning were achieved. The GQL model provides several mechanisms for dealing with incomplete answers such as partial matching, weak search and filter constraints, semantics with different levels of binding etc. An approach that combines a graph-oriented QL and other SSD-QLs aspects is needed for a practical GQL implementation.

11 References

[Abi97] Serge Abiteboul, "Querying semi-structure data", In F.N. Aftrati and Ph. Kolaitis, editors, Proc. 6th International Conference on Database Theory, Volume 1186 of Lecture Notes in Computer Science, pages 1-18, Springer-Verlag, (Delphi Greece), 1997.

[ABS99] Serge Abiteboul, Peter Buneman, Dan Suciu, "Data on the Web – from relational to semistructured data and XML", Morgan Kaufmann Publishers, (California USA), 1999.

[DFFLS98] Alin Deutsch, Mary Frenandez, Daniela Florescu, Alon Levy, Dan Suciu,"A Query Language for XML", ACM Press, 1998.

[KNS99] Yaron Kanza, Werner Nutt and Yehushua Sagiv, "Queries with Incomplete Answers over Semistructured Data", ACM Press, (Philadelphia USA), 1999.

[Mai86] D.Maier. A logic for objects. In Workshop on "Foundations on Deductive Database and Logic Programming", pages 6-26, 1986.

[NM94] J. Neilsen & R.L.Mack, "Usability Inspection Methods", John Wiley and Sons, New York (New York, USA), 1994.

[Reu94] J.Reuben, "Hand book of usability testing - How to Plan, Design and Conduct effective tests", John Wiley and Sons, (California USA), 1994.

[Som97] Ian Sommerville, "Software Engineering", Addison-Welsley, Harlow (England) fifth edition 1997.

[Ull89V1] J. Ullman, "Principles of Database and Knowledge-Base systems", Vol 1:The New Technologies. Computer Science Press, New York (New York, USA), 1989.

[Ull89V2] J. Ullman, "Principles of Database and Knowledge-Base systems", Vol 2:The New Technologies. Computer Science Press, New York (New York, USA), 1989.

12 Source Code

```
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


//////////////////////////////////////////////////////////////
// Global Constants
//////////////////////////////////////////////////////////////
const MAX_STR  = 20; // maximal string length
const MAX_EDGE = 5;  // maximal outgoing/incoming edges
const MAX_NODE = 40; // maximum nodes in a query graph
const MAX_LINE = 80; // maximal string line length
const MAX_LIST = 20; // maximal matching lists
int debug = 1;

//////////////////////////////////////////////////////////////
// Global Variables
//////////////////////////////////////////////////////////////
FILE *logFile, *debugFile;  // output files
char fileLine[MAX_LINE];    // output line
char params[MAX_NODE];      // node variables
bool chairEdge;            // chairman edge selected
char semantics, filter1, filter2; // query constraints


//////////////////////////////////////////////////////////////
// A node represents an element in the semi-structured data graph
// Each node has an array of outgoing edges (name and pointer)
// which represent the sub elements name and address
//////////////////////////////////////////////////////////////
class node{
   public:
   //  member functions
   node();
   node(int id1, char* value1 = "");
   void setEdge(char* str, node* n);
   void visitAll();
   void copy(node* n);
   void print();
   void printAll();
   // data members
   int id;
   char value[MAX_STR];               // value in case of a leaf node
   char edgeName[MAX_EDGE][MAX_STR]; // array of outgoing edge labels
   node* edgePtr[MAX_EDGE];          // pointers correspond to edgeName
   bool visited;
   node* pNext;
};


//////////////////////////////////////////////////////////////
// pair of matchings between a query node and a database node
//////////////////////////////////////////////////////////////
class pair{
   public:
   // member functions
   pair(node* qNode1, node* dbNode1 = 0);
   // data members
   node* qNode;
   node* dbNode;
   pair* pNext;
};
```

```
/////////////////////////////////////////////////////////////////
// list used in constructing the pairs of matchings
/////////////////////////////////////////////////////////////////
class list{

   public:
   list() { pHead = pTail = 0; };
   list(const list& l);
   void add(node* qNode1, node* dbNode1 = 0);
   void remove();
   void copy(list* l);
   pair* find(int id);
   bool empty() { return !pHead; };
   void print(char ch);
   // data member
   pair* pHead;
   pair* pTail;
};


///////////////////////////////////////////////////////////////////
// Graph structure is used for storing the semi-structured database,
// the user queries and the query results generated by the program.
///////////////////////////////////////////////////////////////////
class graph {

   // stack used for searching a node in a graph //
   class stack{
      public:
         stack() { pHead = 0; };
         void push(node* n);
         node* pop();
         void print();
         // data member
         node* pHead;
   };

   // queue used in topological sort //
   class queue{
      public:
         queue() { pHead = 0; pTail = 0; };
         void add(node* n);
         node* remove();
         bool empty() { return !pHead; };
         void print();
         // data member
         node* pHead;
         node* pTail;
   };

   public:
      // member functions
      graph(){ root = new node(1); };
      node* find(int id);
      void topOrder(node** arr);
      void reset();
      void print();
      void eval(graph* q, list** l);
      void evalOr(graph* q, list** answer);
      void evalAnd(graph* q, list** answer);
      void evalWeak(graph* q, list** answer);
      // data members
      node* root;
};
```

```
/***************************************************************/
/*******        GLOBAL FUNCTIONS PROTOTYPE       ******************/
/***************************************************************/
graph* readDB();
graph* query();
void readFile(char* edgeName, graph* db);
void openFiles();
void closeFiles();
void writeFile(char* str, char ch);
void getParams();
void checkParams();
bool isTree(char* params);
void filter(list** answer);
void showLists(char* params, list** l, int step);
void printLists(list** answer, int step);
char **getcgivars();
char x2c(char *what);
void unescape_url(char *url);
void terminate(char* message);



/***************************************************************/
/*******              THE MAIN FUNCTION          ******************/
/***************************************************************/
int main() {

   list* answer[MAX_LIST];

   openFiles();

   // read SSD graph from text files
   graph* db = readDB();

   // get and check the query params
   getParams();
   checkParams();

   // create the query graph from params
   graph* q = query();

   // step 1: search phase
   db -> eval(q, answer);
   printLists(answer, 1);
   showLists(params, answer, 1);

   // step 2: filter phase
   filter(answer);
   printLists(answer, 2);
   showLists(params, answer, 2);

   closeFiles();

   return 0;
}



/***************************************************************/
/*******        GLOBAL FUNCTIONS DEFINITION      ******************/
/***************************************************************/
```

```
///////////////////////////////////////////////////////////////////////
// Read the database and return a graph with the semi-structured data
// Data is stored in text files as described in the documentation
///////////////////////////////////////////////////////////////////////
graph* readDB(){

     graph* db = new graph();

     writeFile("readDB(): start of function.\n", 'd');

     readFile("department", db);
     readFile("course", db);
     readFile("lab", db);
     readFile("chairman", db);
     readFile("teacher", db);
     readFile("instructor", db);
     readFile("name", db);
     readFile("first", db);
     readFile("last", db);
     readFile("seniority", db);

     writeFile("readDB(): end of function.\n", 'd');
     db -> print(); // print the db graph into debug file

     return db;
}


/////////////////////////////////////////////////////////////////////////
// Read the given edge file name into the given database graph
/////////////////////////////////////////////////////////////////////////
void readFile(char* edgeName, graph* db){

   FILE* f;
   char* fileName = new char[MAX_STR];
   char* strLine = new char[MAX_LINE];
   int sourceId, destId;
   char value[MAX_STR];
   node *pNode, *destNode;

   writeFile("readFile(): start of function.\n", 'd');

   if (!strcmp("department", edgeName)) strcpy(fileName, "dept");
   else if (!strcmp("seniority", edgeName)) strcpy(fileName, "senior");
   else if (!strcmp("instructor", edgeName)) strcpy(fileName, "inst");
   else strcpy(fileName, edgeName);

   strcat(fileName, ".txt");
   if (((f = fopen(fileName, "r")) == NULL))
         terminate("error in readFile(): file missing or cannot be open");

   while (!feof(f)){
      strcpy(strLine, "");
      strcpy(value, "");
      sourceId = destId = 0;
      fgets(strLine, MAX_LINE, f);
      sscanf(strLine,"%d %d %s", &sourceId, &destId, &value);
      if (sourceId){ // input line has non white char
            pNode = db -> find(sourceId);
            if (pNode){
               destNode = db -> find(destId);
            if (!destNode) destNode = new node (destId, value);
               pNode -> setEdge(edgeName, destNode);
            } // end if pNode
      } // end if sourceId
   } // end while
```

```
    sprintf(fileLine, "readFile(): file %s was read.\n", fileName);
    writeFile("readFile(): end of function.\n", 'd');
    fclose(f);
}


///////////////////////////////////////////////////////////////////////////////
// constructor - initializes the graph as a QUERY graph according to params
// the full query graph consist of the nodes variables udncbpvwxyz.
///////////////////////////////////////////////////////////////////////////////
graph* query(){

    graph* q = new graph();

     node* department = new node(101);
     node* deptName = new node(102);
     node* course = new node(103);
     node* lab = new node(104);
     node* courseName = new node(333);
     node* labName = new node(444);
     node* person = new node(105);
     node* first = new node(777);
     node* last = new node(888);
     node* seniority = new node(999);

    writeFile("query(): start of function.\n", 'd');

     q -> root -> id = 100;
     q -> root -> setEdge("department", department);
     q -> root -> print();


     if (params[2] == 'n') department -> setEdge("name", deptName);
     if (params[3] == 'c') department -> setEdge("course", course);
     if (params[4] == 'b') department -> setEdge("lab", lab);
     if (params[5] == 'p'){
          if (params[3] == 'c') course -> setEdge("teacher", person);
          if (params[4] == 'b') lab -> setEdge("instructor", person);
          if (chairEdge) department -> setEdge("chairman", person);
     }
     if (params[6] == 'v') course -> setEdge("name", courseName);
     if (params[7] == 'w') lab -> setEdge("name", labName);
     if (params[8] == 'x') person -> setEdge("first", first);
     if (params[9] == 'y') person -> setEdge("last", last);
     if (params[10] == 'z') person -> setEdge("seniority", seniority);

    writeFile("query(): end of function.\n", 'd');
    q -> print(); // print the query graph into debug file

     return q;
}


///////////////////////////////////////////////////////////////////////////////
// Eliminate some of the lists according to the filter constraints (global vars)
// filter1 is the existance constraint and filter2 is the comparison constraint
///////////////////////////////////////////////////////////////////////////////
void filter(list** match){

    int id;
    char str[MAX_STR];
    pair* p;

    writeFile("filter(): enter function.\n", 'd');

    // filter1 (existance constraint) specifies the node variable
    // that must be bound to db element, otherwise eliminate match
```

```
id = 0;
   switch(filter1){
      case 'n':id = 102;
               break;
      case 'c':id = 103;
               break;
      case 'b':id = 104;
               break;
      case 'p':id = 105;
               break;
      case 'v':id = 333;
               break;
      case 'w':id = 444;
               break;
   case 'x':id = 777;
               break;
      case 'y':id = 888;
               break;
      case 'z':id = 999;
               break;
} // end switch
if (id)
   for(int i = 0; match[i] && i<MAX_LIST; i++) {
      p = match[i] -> find(id);
      if (!p->dbNode) match[i] = 0; // p must exist
   }


// filter2 (comparison constraint) specifies the node variable
// that must be bound and eaual to db element, otherwise eliminate match
id = 0;
   switch(filter2){
      case 'c':id = 102;
           strcpy(str, "Chemistry");
               break;
      case 'm':id = 102;
           strcpy(str, "Math");
               break;
      case 's':id = 102;
           strcpy(str, "CS");
               break;
      case 'd':id = 777;
           strcpy(str, "David");
               break;
      case 'r':id = 777;
           strcpy(str, "Ruth");
               break;
   case 'h':id = 888;
           strcpy(str, "Ben-Yisahy");
               break;
      case 'b':id = 888;
               break;
} // end switch
if (id)
   for(int i = 0; i<MAX_LIST; i++) {
      if (!match[i]) continue; // some were eliminated in filter1
      p = match[i] -> find(id);
      if (p && p->dbNode){
         sprintf(fileLine, "filter(): p->dbNode->value: %s.\n", p->dbNode->val
ue);
         writeFile(fileLine, 'd');
         sprintf(fileLine, "filter(): str: %s.\n", str);
         writeFile(fileLine, 'd');
         if(strcmp(p->dbNode->value, str)) match[i] = 0;
      }
      else match[i] = 0;
   } // end for(int i)
```

```
    writeFile("filter(): end of function.\n\n", 'd');
}


///////////////////////////////////////////////////////////////////////////////////
// Create, open and initialize "debug.txt" to which execution results are written
// and log.txt to which query results are written.
////////////////////////////////////////////////////////////////////////////
void openFiles(){

    if (debug){
          if ((debugFile = fopen("debug.txt", "wb")) == NULL)
             terminate("Error opening  debug file. \n");
          writeFile("openFiles(): debug file was opened. \n", 'd');
    }

    if ((logFile = fopen("log.doc", "wt")) == NULL)
          terminate("Error opening log file. \n");
    writeFile("openFiles(): log file was opened. \n", 'd');

    printf("Content-type: text/html\n\n");
    printf("<html>\n");
    printf("<head> </head>\n");
    printf("<body bgcolor=FFFFFF>\n");
}


//////////////////////////////////////////////////////////////////////////
// close all output files
//////////////////////////////////////////////////////////////////////////
void closeFiles(){

    if(debug) fclose(debugFile);
    fclose(logFile);
    // put ending tags of html screen
    printf("</body>\n") ;
    printf("</html>\n") ;
    writeFile("\ncloseFiles(): all files were closed.\n", 'd');
}


//////////////////////////////////////////////////////////////////////////
// Write the string into the file given by ch. ch=d/l means debug/log
//////////////////////////////////////////////////////////////////////////
void writeFile(char* str, char ch){

     int len = strlen(str);

    if (ch == 'l') fwrite(str, 1, len, logFile);
     else if (ch == 'd' && debug) fwrite(str, 1, len, debugFile);
}


/////////////////////////////////////////////////////////////////////////////
// Write the query answers (given in l) to the html file in a table form
/////////////////////////////////////////////////////////////////////////////
void showLists(char* params, list** l, int step){

    char temp[10] = "";
    int ans = 1;

    writeFile("showLists(): start of function.\n", 'd');

    if (step ==1) {
       printf("<BR><FONT SIZE=+3 COLOR=0000FE>Query Results:</FONT><BR>\n");
       printf("To save the query results select Log file from the main menu<BR>\n"
);
          printf("To perform another query select Query from the main menu<BR>\n");
          printf("To view the query parameters click the Back button on your browser<
BR><BR>\n");
          printf("<BR><FONT SIZE=+2 COLOR=0000FF>The match phase results are:</FONT><
BR><BR>\n");
          }
     else printf("<BR><FONT SIZE=+2 COLOR=0000FF>The filter phase results are:</FON
T><BR><BR>\n");

     // before start displaying make sure not all answers are empty
     bool found = false;
      for(int i = 0; !found && i<MAX_LIST; i++)
        if (l[i]) {
           found = true;
           break;
        } // end if

     if (!found) {
        printf("<B>The query result in this phase is empty.\n");
        printf("<BR>try another query or see tutorial from Help option\n");
        return;
     }


     printf("\t<TABLE BORDER=1 CELLPADDING=12>\n");

      // write the node variables in bold in the first row
     printf("\t<TR>\n\t\t<TD>.</TD>\n");
      for (int i = 0; params[i] &&  i<MAX_NODE; i++){
        if (params[i] == '*') continue; // next iteration
           sprintf(fileLine, "\t\t<TD><B>%c</B>", params[i]);
        if(params[i] == 'n') strcat(fileLine, "<BR><B>dept</B>");
        else if(params[i] == 'v') strcat(fileLine, "<BR><B>course</B>");
        else if(params[i] == 'w') strcat(fileLine, "<BR><B>lab</B>");
        else if(params[i] == 'x') strcat(fileLine, "<BR><B>first</B>");
        else if(params[i] == 'y') strcat(fileLine, "<BR><B>last</B>");
        else if(params[i] == 'z') strcat(fileLine, "<BR><B>seniority</B>");
        else strcat(fileLine, "<BR>.");
        strcat(fileLine, "</TD>\n");
        printf(fileLine);
     }
      printf("\t</TR>\n\n");

     // write each answer list on a separate row
      for(int i = 0; i < MAX_LIST; i++){
        if (!l[i]) continue; // if empty matching go to next

        pair* ptr = l[i] -> pHead;
        sprintf(fileLine, "\t<TR>\n\t\t<TD><B>answer%d</TD></B>\n", ans);
        printf(fileLine);
        strcpy(fileLine, "");
        while (ptr){
           strcpy(fileLine, "\t\t<TD>");
              if (ptr->dbNode){
                 strcpy(temp, "");
                 sprintf(temp, "%d", ptr -> dbNode -> id);
                 strcat(fileLine, temp);
              }
              else strcat(fileLine, "--");
           strcat(fileLine, "<BR>");
           if (ptr -> dbNode && ptr -> dbNode -> value)
              strcat(fileLine, ptr -> dbNode -> value);
              strcat(fileLine, ".");
              strcat(fileLine, "</TD>\n");
              printf(fileLine);
              strcpy(fileLine, "");
```

```
            ptr = ptr -> pNext;
         } // while(ptr)
      ans++;
      strcpy(fileLine, "\t</TR>\n\n");
      printf(fileLine);
   } // end for(int i)

   printf("\t</TABLE><BR><BR>\n");

   writeFile("showLists(): end of function.\n\n", 'd');
}



//////////////////////////////////////////////////////////////////////////
// Read the database and return a graph with the semi-structured data
// Data is stored in text files as described in the documentation
//////////////////////////////////////////////////////////////////////////
void printLists(list** answer, int step){

   writeFile("printLists(): start of function.\n", 'd');

   if (step == 1) {
      writeFile("\nThe query parameters are:\n=======================\n\n", 'l'
);
      sprintf(fileLine, "node variables: %s\n", params);
      writeFile(fileLine, 'l');
      sprintf(fileLine, "chairEdge is: %d\n", chairEdge);
      writeFile(fileLine, 'l');
      sprintf(fileLine, "semantics is: %c\n", semantics);
      writeFile(fileLine, 'l');
      sprintf(fileLine, "filter1 is: %c\n", filter1);
      writeFile(fileLine, 'l');
      sprintf(fileLine, "filter2 is: %c\n", filter2);
      writeFile(fileLine, 'l');
      writeFile("node codes: u=100, d=101, n=102, c=103 b=104 p=105\n", 'l');
      writeFile("            v=333, w=444, x=777, y=888 z=999 \n\n\n", 'l');
      writeFile("Query Results:\n=============\n\n", 'l');
      writeFile("The match phase evaluation results are:\n", 'l');
   }
   else writeFile("The filter phase evaluation results are:\n", 'l');
    writeFile("=========================================\n", 'l');

   // before start printing make sure not all answers are empty
   bool found = false;
    for(int i = 0; !found && i<MAX_LIST; i++)
      if (answer[i]) {
         found = true;
         break;
      } // end if

   if (!found) {
      writeFile("\nThe query result in this phase is empty.\n", 'l');
      return;
   }

   for(int i = 0; i < MAX_LIST; i++) {
      if (!answer[i]) continue; // if empty matching go to next
      sprintf(fileLine, "list %d: ", i);
      writeFile(fileLine, 'l');
      answer[i] -> print('l');
    }
   writeFile("\n\n", 'l');

   writeFile("printLists(): end of function.\n\n", 'd');
}
```

```
///////////////////////////////////////////////////////////////////////////
// Display the given message and exit the program
///////////////////////////////////////////////////////////////////////////
void terminate(char* message){

   if (debug) writeFile(message, 'd');
   exit(0);
}


///////////////////////////////////////////////////////////////////////////
// get the parameters from the form in param.html and verify them
///////////////////////////////////////////////////////////////////////////
void getParams(){

    int i, v;
    char **cgivars ;

    writeFile("\ngetParams(): start of function.\n", 'd');

    cgivars = getcgivars() ; // get the CGI vars into list of strings

    i = v = 0;
    strcpy(params, "*****************");
    while (cgivars[i] && i < MAX_NODE)
      switch (cgivars[i][0]){
            case '1':// nodes variables
                  i++;
                  params[v++] = cgivars[i++][0];
                  break;
         case '2':// chairman edge
                  i++;
                  if (cgivars[i++][0] == 'n') chairEdge = false;
                  else chairEdge = true;
                  sprintf(fileLine, "getParams(): chairEdge is: %d\n", chairEdge)
;
                  writeFile(fileLine, 'd');
                  break;
         case '3':// semantics
                  i++;
                  semantics = cgivars[i++][0];
                  sprintf(fileLine, "getParams(): semantics is: %c\n", semantics)
;
                  writeFile(fileLine, 'd');
                  break;
         case '4':// filter1
                  i++;
                  filter1 = cgivars[i++][0];
                  sprintf(fileLine, "getParams(): filter1 is: %c\n", filter1);
                  writeFile(fileLine, 'd');
                  break;
         case '5':// filter2
                  i++;
                  filter2 = cgivars[i++][0];
                  sprintf(fileLine, "getParams(): filter2 is: %c\n", filter2);
                  writeFile(fileLine, 'd');
                  break;
            default:terminate("getParams(): switch case not found.\n");
      }

   if (params[0] == 'a') { // default case
      strcpy(params, "udncbpvwxyz");
      semantics = 'o';
   }
   sprintf(fileLine, "getParams(): params are: %s\n", params);
   writeFile(fileLine, 'd');

   /** Free anything that needs to be freed **/
   for (i=0; cgivars[i]; i++) free(cgivars[i]) ;
   free(cgivars) ;
   writeFile("getParams(): end of function.\n\n", 'd');
}


////////////////////////////////////////////////////////////////////////
// verify the node variables and arrange them in the order: udncbpvwxyz
// if any of these characters are missing place an asterik instead
// if the parameters are invalid send an error message and exit.
////////////////////////////////////////////////////////////////////////
void checkParams(){

   bool check = true;
   char temp[MAX_NODE];

    writeFile("checkParams(): start of function.\n", 'd');

    strcpy(temp, params);
    strcpy(params, "****************");

    // arrange in the topological order: udncbpvwxyz
    for (int i = 0; temp[i] && i < MAX_NODE; i++)
         switch (temp[i]){
             case 'u':params[0] = temp[i];
                    break ;
             case 'd':params[1] = temp[i];
                    break ;
             case 'n':params[2] = temp[i];
                    break ;
             case 'c':params[3] = temp[i];
                    break ;
             case 'b':params[4] = temp[i];
                    break ;
             case 'p':params[5] = temp[i];
                    break ;
             case 'v':params[6] = temp[i];
                    break ;
             case 'w':params[7] = temp[i];
                    break ;
             case 'x':params[8] = temp[i];
                    break ;
             case 'y':params[9] = temp[i];
                    break ;
             case 'z':params[10] = temp[i];
                    break ;
      }

   sprintf(fileLine, "checkParams(): params are: %s\n", params);
   writeFile(fileLine, 'd');

   // check that selected nodes are consistent
    if (params[0]!='u' || params[1]!='d') check = false;
    if (params[2]=='n' || params[3]=='c' || params[4]=='b' || params[5]=='p')
          if (params[1]!='d') check = false;
    if (params[6]=='v' && params[3]!='c') check = false;
    if (params[7]=='w' && params[4]!='b') check = false;
    if (params[8]=='x' || params[9]=='y' || params[10]=='z')
       if (params[5]!='p') check = false;
    if (params[5]=='p' && params[3]!='c' && params[4]!='b' && !chairEdge) check =
 false;


    sprintf(fileLine, "checkParams(): for node variables check is: %d\n", check);
```

```
     writeFile(fileLine, 'd');

    if (!check) {
       printf("<B>Error in selecting the query parameters</B>\n");
          printf("<BR><BR>Please verify that the selected nodes create a graph\n")
;
          printf("<BR><BR>Click again the Query option from the Main Menu or \n");
          printf("<BR><BR>Click the Help option and follow the Tutorial link\n");
        terminate("\ncheckParams(): invalid node variables.\n");
    }

     if (!isTree(params) && semantics =='0') {
       printf("<B>Error in selecting the semantics parameter</B>\n");
          printf("<BR><BR>The selected nodes create an acyclic graph so semantics
must be selected\n");
          printf("<BR><BR>Click again the Query option from the Main Menu or \n");
          printf("<BR><BR>Click the Help option and follow the Tutorial link\n");
        terminate("\ncheckParams(): invalid semantics parameter.\n");
    }

    if (filter1 != '0'){
       char* found = strchr(params, filter1);
       if (!found) {
             printf("<B>Error in selecting the first filter parameter</B>\n");
             printf("<BR><BR>The selected filter does not exist in the selected qu
ery nodes\n");
             printf("<BR><BR>Click again the Query option from the Main Menu or \n
");
             printf("<BR><BR>Click the Help option and follow the Tutorial link\n"
);
             terminate("\ncheckParams(): invalid semantics parameter.\n");
       }
    }

     writeFile("checkParams(): end of function.\n\n", 'd');
}


//////////////////////////////////////////////////////////////////////////
// Return true if the graph query is a tree graph or false otherwise
// i.e. non tree graph with a node that has more than one incoming edge.
// The only such node in the full query graph udncbvwp123xyz is p.
//////////////////////////////////////////////////////////////////////////
bool isTree(char* params){

   bool tree = true;
   int count = 0;

   writeFile("\nisTree(): enter function.\n", 'd');

   if (params[5] == 'p'){
         if (params[3] == 'c') count ++;
         if (params[4] == 'b') count ++;
         if (chairEdge) count ++;
      if (count > 1) tree = false;
   }

   sprintf(fileLine, "isTree(): bool var tree is %d.\n", tree);
   writeFile(fileLine, 'd');
   writeFile("isTree(): end of function.\n\n", 'd');
   return tree;
}

//////////////////////////////////////////////////////////////////////////
// Convert a two-char hex string into the char it represents
//////////////////////////////////////////////////////////////////////////
```

```
char x2c(char *what) {

   register char digit;

   digit = (what[0] >= 'A' ? ((what[0] & 0xdf) - 'A')+10 : (what[0] - '0'));
   digit *= 16;
   digit += (what[1] >= 'A' ? ((what[1] & 0xdf) - 'A')+10 : (what[1] - '0'));

   return (digit);
}


//////////////////////////////////////////////////////////////////////
// Reduce any %xx escape sequences to the characters they represent
//////////////////////////////////////////////////////////////////////
void unescape_url(char *url) {

    register int i,j;

    for(i=0,j=0; url[j]; ++i,++j) {
        if((url[i] = url[j]) == '%') {
            url[i] = x2c(&url[j+1]) ;
            j+= 2 ;
        }
    }
    url[i] = '\0' ;
}


//////////////////////////////////////////////////////////////////////
// Read the CGI input and place all name/val pairs into list.
// Returns list containing name1, value1, name2, value2, ... , NULL.
//////////////////////////////////////////////////////////////////////
char **getcgivars() {

   register int i ;
    char *request_method ;
    int  content_length;
    char *cgiinput ;
    char **cgivars ;
    char **pairlist ;
    int  paircount ;
    char *nvpair ;
    char *eqpos ;

   writeFile("getcgivars(): start of function.\n", 'd');

    // Depending on the request method, read all CGI input into cgiinput
    request_method= getenv("REQUEST_METHOD") ;

    if (!request_method)
        terminate("getcgivars(): request_method is NULL.\n") ;

    if (!strcmp(request_method, "GET") || !strcmp(request_method, "HEAD")) {
        cgiinput= strdup(getenv("QUERY_STRING")) ;
    }
    else if (!strcmp(request_method, "POST")) {
         if (strcasecmp(getenv("CONTENT_TYPE"), "application/x-www-form-urlencode
d"))
                 terminate("getcgivars(): Unsupported Content-Type.\n") ;
         if ( !(content_length = atoi(getenv("CONTENT_LENGTH"))) )
                 terminate("getcgivars(): No Content-Length sent with POST reque
st.\n") ;
         if ( !(cgiinput= (char *) malloc(content_length+1)) )
                 terminate("getcgivars(): Could not malloc for cgiinput.\n") ;
         if ( !fread(cgiinput, content_length, 1, stdin))
             terminate("Couldn't read CGI input from STDIN.\n");
            cgiinput[content_length]='\0' ;
        }
        else terminate("getcgivars(): unsupported REQUEST_METHOD\n") ;

    // Change all plusses back to spaces
    for(i=0; cgiinput[i]; i++) if(cgiinput[i] == '+') cgiinput[i] = ' ' ;

    // First, split on "&" to extract the name-value pairs into pairlist
    pairlist= (char **) malloc(256*sizeof(char **)) ;
    paircount= 0 ;
    nvpair= strtok(cgiinput, "&") ;
    while (nvpair) {
        pairlist[paircount++]= strdup(nvpair) ;
        if (!(paircount%256))
            pairlist= (char **) realloc(pairlist,(paircount+256)*sizeof(char **))
;
        nvpair= strtok(NULL, "&") ;
    }
    pairlist[paircount]= 0 ;     // terminate the list with NULL

    // Then, from the list of pairs, extract the names and values
    cgivars= (char **) malloc((paircount*2+1)*sizeof(char **)) ;
    for (i= 0; i<paircount; i++) {
        if (eqpos=strchr(pairlist[i], '=')) {
            *eqpos= '\0' ;
            unescape_url(cgivars[i*2+1]= strdup(eqpos+1)) ;
        }
        else unescape_url(cgivars[i*2+1]= strdup("")) ;
        unescape_url(cgivars[i*2]= strdup(pairlist[i])) ;
    }
    cgivars[paircount*2]= 0 ;    // terminate the list with NULL

    // Free anything that needs to be freed
    free(cgiinput) ;
    for (i=0; pairlist[i]; i++) free(pairlist[i]) ;
    free(pairlist) ;

    writeFile("getcgivars(): end of function.\n", 'd');
    return cgivars ;
}


/*****************************************************************/
/*******            node CLASS FUNCTIONS         ****************/
/*****************************************************************/


//////////////////////////////////////////////////////////////////
// constructor - create an empty node with all values zero or null
//////////////////////////////////////////////////////////////////
node::node(){

   id = 0;
   strcpy(value, "");
   for (int i = 0; i < MAX_EDGE; i++) edgeName[i][0] = 0;
   for (int i = 0; i < MAX_EDGE; i++) edgePtr[i] = 0;
   visited = 0;;
   pNext = 0;
}


///////////////////////////////////////////////////////////////////
// constructor - default of value string is null (for internal node)
// all edge names and pointers are initially null
///////////////////////////////////////////////////////////////////
node::node(int id1, char* value1){
    id = id1;
    strcpy(value, value1); // default value is null (func header)
    for (int i = 0; i < MAX_EDGE; i++) edgeName[i][0] = 0;
    for (int i = 0; i < MAX_EDGE; i++) edgePtr[i] = 0;
    visited = false;
    pNext = 0;
}

/////////////////////////////////////////////////////////////////
// set the node's 2 edges arrays in the next available position
// to the edge's name and and its corresponding edge (node) pointer
/////////////////////////////////////////////////////////////////
void node::setEdge(char* str, node* n){

    int i;

    for(i = 0; edgePtr[i] && i < MAX_EDGE; i++);
    strcpy(edgeName[i], str);
    edgePtr[i] = n;
}

/////////////////////////////////////////////////////////////////
// copy the id and value of current node into the given node n
/////////////////////////////////////////////////////////////////
void node::copy(node* n){

    n -> id = id;
    strcpy(n -> value, value);
}


/////////////////////////////////////////////////////////////////
// Reset the node's visited data member (and its descendnts)
// to false recursively, so that it can be used for more queries
/////////////////////////////////////////////////////////////////
void node::visitAll(){

    visited = false;
    for(int i = 0; edgePtr[i] && i < MAX_EDGE ; i++)
       edgePtr[i] -> visitAll();
}


/////////////////////////////////////////////////////////////////
// display the screen the node information
/////////////////////////////////////////////////////////////////
void node::print(){

    sprintf(fileLine, "\nid: %d\n", id);
    writeFile(fileLine, 'd');
    strcpy(fileLine, "value: ");
    if (strcmp("", value)) strcat(fileLine, value);
    else strcat(fileLine, "no value");
    strcat(fileLine, "\n");
    writeFile(fileLine, 'd');

    writeFile("outgoing edges:\n", 'd');
    if (edgePtr[0]){
          for(int i = 0; edgePtr[i] && i < MAX_EDGE ; i++){
             sprintf(fileLine, "\t %d: %d %s %s\n",
                     i, edgePtr[i] -> id, edgeName[i], edgePtr[i] -> value);
             writeFile(fileLine, 'd');
       }
    }
    else writeFile("no outgoing edges\n", 'd');
    writeFile("\n", 'd');
```

```
}


//////////////////////////////////////////////////////////////////////
// display the screen the node information
//////////////////////////////////////////////////////////////////////
void node::printAll(){

   if (visited) return;

    sprintf(fileLine, "id: %d\n", id);
    writeFile(fileLine, 'd');
    strcpy(fileLine, "value: ");
    if (strcmp("", value)) strcat(fileLine, value);
    else strcat(fileLine, "no value");
    strcat(fileLine, "\n");
    writeFile(fileLine, 'd');

    writeFile("outgoing edges:\n", 'd');
    if (edgePtr[0]){
         for(int i = 0; edgePtr[i] && i < MAX_EDGE ; i++){
            sprintf(fileLine, "\t %d: %d %s %s\n",
                      i, edgePtr[i] -> id, edgeName[i], edgePtr[i] -> value);
            writeFile(fileLine, 'd');
      }
   }
    else writeFile("no outgoing edges\n", 'd');
    writeFile("\n", 'd');

    for (int i = 0; edgePtr[i] && i < MAX_EDGE; i++) edgePtr[i] -> printAll();

   visited = true;
}


/*******************************************************************/
/*******           graph CLASS FUNCTIONS          ******************/
/*******************************************************************/


////////////////////////////////////////////////////////////////////////////
// Resets all visited data members in the database graph to false
// Used in order to procees a new query
////////////////////////////////////////////////////////////////////////////
void graph::reset(){

   root -> visitAll();
}

////////////////////////////////////////////////////////////////////////////
// Sorts the graph nodes in a topological order
////////////////////////////////////////////////////////////////////////////
void graph::topOrder(node** arr){

   queue openQ, closeQ;
    node* ptr;

    writeFile("graph::topOrder(): start of function.\n", 'd');

    reset();
    openQ.add(root);

    while (!openQ.empty()){
      ptr = openQ.remove();
      closeQ.add(ptr);
      for(int i = 0; ptr -> edgePtr[i] && i < MAX_EDGE; i++)
            if (!ptr -> edgePtr[i] -> visited){
                openQ.add(ptr -> edgePtr[i]);
                ptr -> edgePtr[i] -> visited = true;
            }
    } // end while

    // reverse the closeQ order to get the topological order
     for(int i = 0; i < MAX_NODE; i++) arr[i] = 0;
     for(int i = 0; !closeQ.empty() && i < MAX_NODE; i++) arr[i] = closeQ.remove()
;

     writeFile("graph::topOrder(): topological order: ", 'd');
     for(int i = 0; arr[i] && i < MAX_NODE; i++){
          sprintf(fileLine, " %d", arr[i] -> id);
          writeFile(fileLine, 'd');
     }

     writeFile("\ngraph::topOrder(): end of function.\n\n", 'd');
}


//////////////////////////////////////////////////////////////////////
// Returns a pointer to the node which has the given id.
// Returns null if the given id was not found in the graph.
//////////////////////////////////////////////////////////////////////
node* graph::find(int id){

    bool found = false;
    node *nAns = 0, *pNode;
    stack* s = new stack();

     s -> push(root);
     while(!found && s -> pHead){ // while not found and stack not empty
          pNode = s -> pop();
       if (pNode -> id == id) {
          found = true;
          nAns = pNode;
       }
       for(int i = 0; !found && pNode -> edgePtr[i] && i < MAX_EDGE; i++)
          if (pNode -> edgePtr[i] -> id == id) {
              found = true;
                 nAns = pNode -> edgePtr[i];
          }
          else s -> push(pNode -> edgePtr[i]);
    } // end while

    return nAns;
}


//////////////////////////////////////////////////////////////////////////////
// Evaluate the given query graph against the db graph for maximal matchings
// Return an array of pointers to lists, where each list contains matchings
// (in topological order) of query id and its corresonding database id
//////////////////////////////////////////////////////////////////////////////
void graph::eval(graph* q, list** match){

    node* order[MAX_NODE]; // the query nodes in topologigal order
    int nextMatch = 0;  // next pos in match[] to add a new created list
    int lastMatch;         // last pos in match[] before new lists were added
    pair* pairPtr;
    int  edgeNum; // num of child edges with the same name
    bool found;   // indicates if order[n]'s child edge was found

    writeFile("graph::eval(): start of function.\n\n", 'd');

    // arrange query nodes in topological order (t.o)
    q -> topOrder(order);
    // initialize all matchings lists to null
    for (int i = 0; i < MAX_LIST; i++) match[i] = 0;

     // assign the db root to the query root
     list* l = new list();
     l -> add(q -> root, root);
     match[nextMatch++] = l;

     // for each node n in order (t.o)
     for(int n = 0; order[n] && n < MAX_NODE; n++){
          // for each edge e of n
          for(int e = 0; order[n]->edgePtr[e] && e < MAX_EDGE; e++){


                lastMatch = nextMatch;
          // for each matching m of match[]
                for(int m = 0; m < lastMatch; m++) {

                    pairPtr = match[m] -> find(order[n] -> id);
                if (pairPtr){
                        edgeNum = 0;
                    list* copyList = new list(*match[m]); // list before changes
                    found = false;
                        // for each child edge of the of n's db node
                    for(int c = 0; pairPtr->dbNode && pairPtr->dbNode->edgeName[c] &&
c < MAX_EDGE; c++){
                          if(!strcmp(pairPtr->dbNode->edgeName[c], order[n]->edgeName[
e])){
                          found = true;
                                edgeNum++;
                          if (edgeNum == 1)
                                    match[m] -> add(order[n]->edgePtr[e], pairPtr->dbNo
de->edgePtr[c]);
                          else { // more than one edge with same name
                                    list* newList = new list(*copyList);
                              newList -> add(order[n]->edgePtr[e], pairPtr->dbNode->edg
ePtr[c]);

                              match[nextMatch++] = newList;
                          }
                              if (debug){
                                    writeFile("\nThe matching lists are:\n", 'd');
                                 for(int i = 0; i < nextMatch; i++) match[i] -> print('
d');

                              writeFile("\n", 'd');
                          } // end if (debug)
                          } // end if (strcmp())
                        } // end for (int c)
                    if (!found) match[m] -> add(order[n]->edgePtr[e], 0);
                } //if (pairPtr)

                } //end for(int m)
        } //end for(int e)
    } // end for(int n)

   writeFile("graph::eval(): end of matching.\n\n", 'd');

    if (!isTree(params)) {
         if (semantics == 'o') evalOr(q, match);
         else if (semantics == 'a') evalAnd(q, match);
    }

   writeFile("graph::eval(): end of function.\n\n", 'd');
}


////////////////////////////////////////////////////////////////////////
```

```
// evaluate the query results under OR semantics. For each list first remove
// any null bindings, then remove duplicates and finally create another list
// for more that one person node before removing it from match
//////////////////////////////////////////////////////////////////////////////
void graph::evalOr(graph* q, list** match){

    int dbId;
   pair* p;
   list* add[MAX_LIST];
   int addIndex = 0;

   writeFile("\n\ngraph::evalOr(): start of function.\n", 'd');

   // initialize all matchings lists to null
   for (int i = 0; i < MAX_LIST; i++) add[i] = 0;

   // remove all null matchings except first
   // first is the t.o position and must not be removed
    for (int i = 0; match[i] && i<MAX_LIST; i++){
      pair* pFirst;
      bool first = true;

      for (int j = 0; (p = match[i] -> find(105)) && j < 4; j++){
         if (first) {
            pFirst = p;
            first = false;
         }
         else {
            if ((p -> dbNode) && !(pFirst->dbNode))
               pFirst->dbNode = p->dbNode;
            else match[i] -> remove();
         } // end if (first)
         p -> qNode->id = 99;
      } // end for (int j)

      for (int j = 0; (p = match[i] -> find(99)) && j < 4; j++)
         p -> qNode->id = 105;
   } // end for (int i)

   // remove duplicate matchings (no null ones)
    for (int i = 0; match[i] && i<MAX_LIST; i++){
      bool first = true;
      for (int j = 0; (p = match[i] -> find(105)) && j < 4; j++){
         if (first) {
            dbId = p -> dbNode -> id;
            first = false;
         }
         else if (p->dbNode->id == dbId) match[i] -> remove();
         p -> qNode->id = 99;
      }
      for (int j = 0; (p = match[i] -> find(99)) && j < 4; j++)
         p -> qNode->id = 105;
   }

   int i;
   // create additional lists for non null non duplicate matchings
    for (i = 0; match[i] && i<MAX_LIST; i++){
      bool first = true;
      for (int j = 0; (p = match[i] -> find(105)) && j < 4; j++)
         if (first) {
            p -> qNode->id = 99;
            first = false;
         }
         else {
            dbId = p->dbNode->id;
            match[i] -> remove();
                list* newList = new list();
            match[i] -> copy(newList);
            add[addIndex] = newList;
            p = add[addIndex] -> find(99);
            p->dbNode->id = dbId;
            node* n = find(dbId);
            // update cildren
            for (int c = 0; n->edgePtr[c] && c<MAX_EDGE; c++)
               if ((p = add[addIndex] -> find(777)) && !strcmp("first", n->edgeNa
me[c])){
                  p->dbNode->id = n->edgePtr[c]->id;
                  strcpy(p->dbNode->value, n->edgePtr[c]->value);
               }
            for (int c = 0; n->edgePtr[c] && c<MAX_EDGE; c++)
               if ((p = add[addIndex] -> find(888)) && !strcmp("last", n->edgeNam
e[c])){
                  p->dbNode->id = n->edgePtr[c]->id;
                  strcpy(p->dbNode->value, n->edgePtr[c]->value);
               }
            for (int c = 0; n->edgePtr[c] && c<MAX_EDGE; c++)
               if ((p = add[addIndex] -> find(999)) && !strcmp("seniority", n->ed
geName[c])){
                  p->dbNode->id = n->edgePtr[c]->id;
                  strcpy(p->dbNode->value, n->edgePtr[c]->value);
               }
            addIndex++;
         } // end if (first)
   }

   for (int index = 0; index < addIndex; index++)
      match[i+index] = add[index];

   for (int i = 0; match[i] && (p = match[i] -> find(99)) && i < MAX_LIST; i++)
      if (p) p -> qNode->id = 105;

   writeFile("graph::evalOr(): end of function.\n\n", 'd');
}


////////////////////////////////////////////////////////////////////////
// evaluate the query under AND semantics. For each list, all person
// nodes (with id=105) must be matched/bound to the same database node
// such duplicate pairs are removed from the current match list
////////////////////////////////////////////////////////////////////////
void graph::evalAnd(graph* q, list** match){

   int dbId;
   bool valid;
   pair* p;

   writeFile("graph::evalAnd(): start of function.\n", 'd');

    for (int i = 0; match[i] && i<MAX_LIST; i++) {

      valid = true; // assume list is AND semantics valid
      dbId = 0;     // assume first db match does not exist

      // find first query node with id=105 and check if it has a db match
      pair* pFirst = match[i] -> find(105);
      if (pFirst->dbNode) dbId = pFirst->dbNode->id;
      else valid = false;
      pFirst->qNode->id = 99; // dummy value, so next 105 can be found

      // find second query node with id=105 (must exist for dag)
      pair* pSecond = match[i] -> find(105);
      if (!pSecond->dbNode) valid = false;
      else if(pSecond->dbNode->id != dbId) valid = false;
      match[i] -> remove(); // remove anyway, only 1 copy in answer
```

```
        // find third query node with id=105 (not always exist) and remove it
        pair* pThird = match[i] -> find(105);
        if (pThird){
           if (pThird->dbNode && pThird->dbNode->id != dbId) valid = false;
           match[i] -> remove(); // remove anyway
        }

        if(!valid) {
           pFirst->dbNode = 0; // null, does not satisfy AND semantics
           p = match[i] -> find(777);  // reset p node children to null
           if (p) p->dbNode = 0;
           p = match[i] -> find(888);  // reset p node children to null
           if (p) p->dbNode = 0;
           p = match[i] -> find(999);  // reset p node children to null
           if (p) p->dbNode = 0;

        }

        pFirst->qNode->id = 105; // restore original value of query id

    } // end for(int i = 0)


    writeFile("graph::evalAnd(): end of function.\n\n", 'd');
}


//////////////////////////////////////////////////////////////////
// display all graph nodes, starting with the root, recursively
//////////////////////////////////////////////////////////////////
void graph::print(){

    writeFile("\nGRAPH DATA IS: \n", 'd');
    writeFile("************** \n\n", 'd');
    root -> printAll();
}



/*******************************************************************/
/*******        graph::queue CLASS FUNCTIONS       ******************/
/*******************************************************************/

//////////////////////////////////////////////////////////////
// add a node always at the BEGINING of the queue
//////////////////////////////////////////////////////////////
void graph::queue::add(node* n){

    if (!pHead) pTail = n;
    n -> pNext = pHead;
    pHead = n;
}


//////////////////////////////////////////////////////////////
// Remove a node always from the END of the queue
//////////////////////////////////////////////////////////////
node* graph::queue::remove(){

    node* pNode = pTail;

    if (!pHead)
       terminate("queue::remove() - empty queue, cannot remove\n");

    if (pHead -> pNext){ // more than one node in the queue
       node* ptr = pHead;
```

```
        while (!(ptr -> pNext == pTail)) ptr = ptr -> pNext;
        pTail = ptr;
        ptr -> pNext = 0;
    }
    else pHead = pTail = 0; // exactly one node in the queue

    return pNode;
}



/*******************************************************************/
/*******        graph::stack CLASS FUNCTIONS      ******************/
/*******************************************************************/

///////////////////////////////////////////////////////////////
// add a node always at the BEGINING of the stack
///////////////////////////////////////////////////////////////
void graph::stack::push(node* n){

    n -> pNext = pHead;
    pHead = n;
}

///////////////////////////////////////////////////////////////
// Remove a node always from the BEGINING of the stack
///////////////////////////////////////////////////////////////
node* graph::stack::pop(){

    node* pNode = pHead;

    if (pHead) pHead = pHead -> pNext;
    else terminate("stack::pop() - empty stack, cannot remove\n");

    return pNode;
}



/*******************************************************************/
/*******            pair CLASS FUNCTIONS          ******************/
/*******************************************************************/

///////////////////////////////////////////////////////////////
// Contructor of the pair class, id2 has 0 default value
///////////////////////////////////////////////////////////////
pair::pair(node* qNode1, node* dbNode1){

    qNode = qNode1;
    dbNode = dbNode1;
    pNext = 0;
}



/*******************************************************************/
/*******            list CLASS FUNCTIONS          ******************/
/*******************************************************************/

///////////////////////////////////////////////////////////////
// Copy contructor - return copy of the given list
///////////////////////////////////////////////////////////////
list::list(const list& l){

    pHead = pTail = 0;
    if (l.pHead){
        pair* ptr = l.pHead;
```

```
          do {
                add(ptr -> qNode, ptr -> dbNode);
             ptr = ptr -> pNext;
          } while  (ptr);
       }
    }


    ///////////////////////////////////////////////////////////////////
    // add a pair always at the END of the matching list
    // add a copy the query node since multiple matcings might exist
    ///////////////////////////////////////////////////////////////////
    void list::add(node* qNode1, node* dbNode1){

       node *copyNode = new node();
       qNode1 -> copy(copyNode);
       pair* p = new pair(copyNode, dbNode1);

       if (pHead) pTail -> pNext = p;
       else pHead = p;
       pTail = p;
    }


    //////////////////////////////////////////////////////////////////////
    // Remove the first pair with id=105 from the list. This duplicate
    // pair must exist in the list because of the calling function
    //////////////////////////////////////////////////////////////////////
    void list::remove(){

       pair *ptr, *prev = 0;
       bool  found = false;

       ptr = pHead;

       while  (!found)
          if  (ptr->qNode->id == 105)  found = true;
          else  {
                prev = ptr;
                ptr = ptr -> pNext;
          }

       if(!found) terminate("list::remove() - id 105 not found");
       else  {
             prev -> pNext = ptr -> pNext;
          delete ptr;
       }
    }


    //////////////////////////////////////////////////////////////////////
    // Find the pair in the list which has the given query id
    // Return a pointer to this pair in the list or null if not found
    //////////////////////////////////////////////////////////////////////
    pair* list::find(int id){

       pair *ptr, *pReturn = 0;

       ptr = pHead;
       while  (ptr && !pReturn){
             if  (ptr -> qNode -> id == id) pReturn = ptr;
          ptr = ptr -> pNext;
       }

       return pReturn;
    }
```

```
////////////////////////////////////////////////////////////////////////
// copy the current list into the given list l. copy only the nodes id
// and value by calling node::copy(). only for semantics purposes
////////////////////////////////////////////////////////////////////////
void list::copy(list* l){

   if (!pHead) terminate("error list::copy(): the list to copy is empty");

   pair* ptr = pHead;
   do {
      node *qNode  = new node();
      node *dbNode = new node();
      ptr -> qNode -> copy(qNode);
      if (ptr -> dbNode) ptr -> dbNode -> copy(dbNode);
      else dbNode = 0;
      l -> add(qNode, dbNode);
      ptr = ptr -> pNext;
   }
   while (ptr);
}


////////////////////////////////////////////////////////////////////
// write file all pair ids in the list. ch=l/d means log/debug file
////////////////////////////////////////////////////////////////////
void list::print(char ch){

   char temp[10] = "";
   pair* ptr = pHead;

   if (!ptr) writeFile("\n\n empty list \n\n", ch);
   while (ptr){
         sprintf(fileLine, "(%d,", ptr -> qNode -> id);
      if (ptr -> dbNode){
            if (ptr -> dbNode -> id < 10) strcat(fileLine, " ");
         strcpy(temp, "");
         sprintf(temp, "%d", ptr -> dbNode -> id);
         strcat(fileLine, temp);
      }
      else strcat(fileLine, "--");
      strcat(fileLine,") ");
      writeFile(fileLine, ch);
      ptr = ptr -> pNext;
   }
   writeFile("\n", ch);
}
```

Java Source Code

```
import java.awt.*;
import java.applet.*;
import java.lang.*;
import java.util.*;


/////////////////////////////////////////////////////////////////
// A node represents an element in the semi-structured data graph
// Each node has an array of outgoing edges (name and pointer)
// which represent the sub elements name and address
////////////////////////////////////////////////////////////////

public class node extends Applet {

      int MAX_STR = 80;
      int MAX_EDGE = 8;
      int MAX_NODES = 15;
      int d = 25;       // diameter of node's circle
      int xText= 12;  // x coordinate of text with respect to circle
      int yText = 15; // y coordinate of text with respect to circle

      int xNode; // x coordinate of the node
      int yNode; // y coordinate of the node
      int xMax;  // max distance between first and last child nodes
      int yDiff; // distance between parent/child nodes

      String id;     // mandatory for each node
      String value; // for leaf node
      int edgeNum;  // number of edges a node has


      String edgeName[] = new String[MAX_EDGE]; // children edge labels
      String edgeId[] = new String[MAX_EDGE];    // children ids
correspond to edgeName
      String edgeValue[] = new String[MAX_EDGE]; // children values
correspond to ids

      class childCoor {
            String id;
            int x;
            int y;
      };

      childCoor child[] = new childCoor [MAX_NODES];


      //////////////////////////////////////////////////////////////////
      // node class constructor
      //////////////////////////////////////////////////////////////////

      public node(String paramsNode){

            // get the node's data: id, value and all its edges
            StringTokenizer params = new StringTokenizer(paramsNode,
",");
            id = params.nextToken(); // get node's id
            value = params.nextToken(); // get node's value

            edgeNum = MAX_EDGE;
            // get node's children info
            for (int i = 0; i < MAX_EDGE; i++){
                edgeName[i] = params.nextToken();
                if (edgeName[i].compareTo("empty") == 0){
                      edgeNum = i;
                      break;
                }
                edgeId[i] = params.nextToken();
                edgeValue[i] = params.nextToken();
          }
          for (int i = 0; i < MAX_NODES; i++)
                child[i] = new childCoor();

    } // end of node()
    ////////////////////////////////////////////////////////////////////


    ////////////////////////////////////////////////////////////////////
    // return the x or y coordinate of the given id node
    ////////////////////////////////////////////////////////////////////
    int find(String id, char coordinate){

          int answer = 0;
          boolean foundNode = false;
          int i;

          for (i = 0; !foundNode && i < MAX_NODES; i++)
                if (child[i].id.compareTo(id) == 0) foundNode = true;
          if (foundNode)
                if (coordinate == 'x') answer = child[--i].x;
                else answer = child[--i].y;
          return answer;

    } // end int find()
    ////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////
    //
    ////////////////////////////////////////////////////////////////////
    public void paint (Graphics g, int paramsDraw[]) {

          // get the node's drawing parameters
          xNode = paramsDraw[0]; // the node's X coordinates
          yNode = paramsDraw[1]; // the node's Y coordinates
          xMax  = paramsDraw[2]; // max distance between first and
last child nodes
          yDiff = paramsDraw[3]; // vertical distance between parent
and child node

          int xMin= xNode - xMax/2; // first child node
          int xDiff = 0;  // distance between 2 child nodes (0 if one
child)

          int xPos, yPos; // coordinates of current node child
          int x0Line, y0Line, x1Line, y1Line; // edge line coordinates
          int xEdge, yEdge; // edge name coordinates

          g.setColor(Color.black);
          Font f = new Font("Ariel", Font.BOLD, 10);
          g.setFont(f);

          // display the given node
          g.drawOval(xNode, yNode, d , d);
```

```
            g.drawString(id, xNode+xText, yNode+yText);

            // display the node's chlidren
            if (edgeNum == 1) xMin = xNode;
            else xDiff = xMax / (edgeNum - 1);
            yPos = yNode + yDiff; // xPos is modified inside loop
            x0Line = xNode + (d/2);
            y0Line = yNode + d;
            y1Line = yPos; // x1Line is modified inside loop
            yEdge = yNode + (yDiff - 10);
            int childIndex = 0;
            for (int i = 0; i < edgeNum; i++){
                  xPos = xMin+ (i * xDiff);
                  g.drawOval(xPos, yPos, d, d);
                  g.drawString(edgeId[i], xPos+xText, yPos+yText);
                  x1Line = xPos + d/2;
                  g.drawLine(x0Line, y0Line, x1Line, y1Line);
                  xEdge = xPos + xText;
                  g.drawString(edgeName[i], xPos, yNode+yDiff-10);
                  if (edgeValue[i].compareTo("val") != 0)
                        g.drawString(edgeValue[i], xPos-10, yPos+d+10);
                  child[childIndex].id = edgeId[i];
                  child[childIndex].x = xPos;
                  child[childIndex++].y = yPos;
            }

      } // end of void paint()
      ////////////////////////////////////////////////////////////////////

} // end of class node
```

```
import java.awt.*;
import java.applet.*;
import java.lang.*;
import java.util.*;


public class query extends Applet
{
      int MAX_NODES = 15;
      int totalNodes = MAX_NODES;
      String paramsNode[] = new String[MAX_NODES];
      node nodes[] = new node[MAX_NODES];


      //////////////////////////////////////////////////////////////////
      // read applet parameters
      //////////////////////////////////////////////////////////////////

      public void init(){

            for (int i = 0; i < MAX_NODES; i++){
                  paramsNode[i] = getParameter("node" + i);
                  if (paramsNode[i] == null){
                        totalNodes = i;
                        break;
                  }
                  //System.out.println("i is: " + i);
                  //System.out.println(paramsNode[i]);
                  nodes[i] = new node(paramsNode[i]);
            }
      } // end of void init()
      //////////////////////////////////////////////////////////////////



      //////////////////////////////////////////////////////////////////
      // paint the query graph, uses node.paint
      //////////////////////////////////////////////////////////////////
      public void paint (Graphics g){

            g.setColor(Color.white);
            int d = 25;
            int nodeNum = 0;
            int edgeNum = 0; // num of edges of the current node
            int paramsDraw[] = new int[4];
            int xNode = 200; // node's X coordinates
            int yNode = 20;  // node's Y coordinates
            int xMax = 160;  // max distance first and last child
            int yDiff = 50; // distance between parent/child nodes
            int xDiff = 0;  // distance between 2 child nodes (0 if one
child)

            // draw the root node u
            paramsDraw[0] = xNode;
            paramsDraw[1] = yNode;
            paramsDraw[2] = xMax;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            // drwa node d (department)
            paramsDraw[0] = nodes[0].find(nodes[nodeNum].id, 'x');
            paramsDraw[1] = nodes[0].find(nodes[nodeNum].id, 'y');
            paramsDraw[2] = xMax;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            // draw node c (course)
            paramsDraw[0] = nodes[1].find(nodes[nodeNum].id, 'x');
            paramsDraw[1] = nodes[1].find(nodes[nodeNum].id, 'y');
            paramsDraw[2] = xMax;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            // draw node l (lab)
            paramsDraw[0] = nodes[1].find(nodes[nodeNum].id, 'x');
            paramsDraw[1] = nodes[1].find(nodes[nodeNum].id, 'y');
            paramsDraw[2] = xMax;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            // draw node t (teacher)
            paramsDraw[0] = nodes[2].find(nodes[nodeNum].id, 'x');
            paramsDraw[1] = nodes[2].find(nodes[nodeNum].id, 'y');
            paramsDraw[2] = xMax/2+50;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            g.drawString("teacher", paramsDraw[0]-40 , paramsDraw[1]-
20);
            g.drawString("instructor", paramsDraw[0]+30 , paramsDraw[1]-
20);
            g.drawString("chairman", paramsDraw[0], paramsDraw[1]-50);
            g.drawLine(nodes[1].xNode+(d/2), nodes[1].yNode+d, nodes
[1].xNode+(d/2), nodes[4].yNode);

            int nXPos = nodes[0].xNode-xMax+(d/2);
            int nYPos = nodes[1].yNode + 40;

            g.drawOval(nXPos, nYPos, d, d);
            g.drawString("n", nXPos+10, nYPos+15);

            g.drawLine(nodes[1].xNode+(d/2), nodes[1].yNode+d, nXPos+
(d/2), nYPos);
            g.drawString("name", nXPos+d, nYPos-d/3);

      } // end void paint
      ///////////////////////////////////////////////////////////////////


}
```

```
import java.awt.*;
import java.applet.*;
import java.lang.*;
import java.util.*;


public class db extends Applet
{
      int MAX_NODES = 40;
      int totalNodes;
      String paramsNode[] = new String[MAX_NODES];
      node nodes[] = new node[MAX_NODES];


      ////////////////////////////////////////////////////////////////////
      // read applet parameters
      ////////////////////////////////////////////////////////////////////

      public void init(){

            int totalNodes = MAX_NODES;
            for (int i = 0; i < MAX_NODES; i++){
                  paramsNode[i] = getParameter("node" + i);
                  if (paramsNode[i] == null){
                        totalNodes = i;
                        break;
                  }
                  //System.out.println(paramsNode[i]);
                  nodes[i] = new node(paramsNode[i]);
            }
      } // end of void init()
      ////////////////////////////////////////////////////////////////////


      ////////////////////////////////////////////////////////////////////
      // paint the database graph, uses node.paint
      ////////////////////////////////////////////////////////////////////
      public void paint (Graphics g){

            int d = 25;
            int nodeNum = 0;
            int edgeNum = 0; // num of edges of the current node
            int paramsDraw[] = new int[4];
            int xNode = 320; // node's X coordinates
            int yNode = 20;  // node's Y coordinates
            int xMax = 450;  // max distance first and last child
            int yDiff = 85; // distance between parent/child nodes

            // draw the root node
            paramsDraw[0] = xNode;
            paramsDraw[1] = yNode;
            paramsDraw[2] = xMax;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            // draw 3 department nodes
            for (int i = 1; i < 4; i++) {
                  paramsDraw[0] = nodes[0].find(nodes[nodeNum].id, 'x');
                  paramsDraw[1] = nodes[0].find(nodes[nodeNum].id, 'y');
                  int dXPos = paramsDraw[0];
                  int dYPos = paramsDraw[1];
```

```
                paramsDraw[2] = xMax/3;
                paramsDraw[3] = yDiff;
                nodes[nodeNum++].paint(g, paramsDraw);
            }

            // draw department's id=3 sub nodes
            for (int i = 1; i < 3; i++) {
                paramsDraw[0] = nodes[1].find(nodes[nodeNum].id, 'x');
                paramsDraw[1] = nodes[1].find(nodes[nodeNum].id, 'y');
                paramsDraw[2] = xMax/8-20;
                paramsDraw[3] = yDiff;
                nodes[nodeNum++].paint(g, paramsDraw);
            }

            // draw department's id=4 sub nodes
            for (int i = 1; i < 3; i++) {
                paramsDraw[0] = nodes[2].find(nodes[nodeNum].id, 'x');
                paramsDraw[1] = nodes[2].find(nodes[nodeNum].id, 'y');
                paramsDraw[2] = xMax/8-20;
                paramsDraw[3] = yDiff;
                nodes[nodeNum++].paint(g, paramsDraw);
            }

            // draw department's id=5 sub nodes
            for (int i = 1; i < 4; i++) {
                paramsDraw[0] = nodes[3].find(nodes[nodeNum].id, 'x');
                paramsDraw[1] = nodes[3].find(nodes[nodeNum].id, 'y');
                paramsDraw[2] = xMax/8-20;
                paramsDraw[3] = yDiff;
                nodes[nodeNum++].paint(g, paramsDraw);
            }

            g.drawLine(nodes[3].xNode+d/2, nodes[3].yNode+d, nodes
[7].xNode+d/2, nodes[7].yNode);
            g.drawString("course", nodes[7].xNode+50, nodes[7].yNode-
20);


            // draw course's id=9 sub node
            paramsDraw[0] = nodes[4].find(nodes[nodeNum].id, 'x');
            paramsDraw[1] = nodes[4].find(nodes[nodeNum].id, 'y');
            paramsDraw[2] = xMax/8-20;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            // draw lab's id=10 sub node
            paramsDraw[0] = nodes[5].find(nodes[nodeNum].id, 'x');
            paramsDraw[1] = nodes[5].find(nodes[nodeNum].id, 'y');
            paramsDraw[2] = xMax/8-20;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            // draw courses's id=11 sub node
            paramsDraw[0] = nodes[6].find(nodes[nodeNum].id, 'x');
            paramsDraw[1] = nodes[6].find(nodes[nodeNum].id, 'y');
            paramsDraw[2] = xMax/8-20;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            // draw courses's id=12 sub node
            paramsDraw[0] = nodes[7].find(nodes[nodeNum].id, 'x');
            paramsDraw[1] = nodes[7].find(nodes[nodeNum].id, 'y');
            paramsDraw[2] = xMax/8-20;
            paramsDraw[3] = yDiff;
            nodes[nodeNum++].paint(g, paramsDraw);

            g.drawLine(nodes[8].xNode+d/2, nodes[8].yNode+d, nodes
[14].xNode+d/2, nodes[14].yNode);
            g.drawString("instructor", nodes[14].xNode+25, nodes
[14].yNode-30);

            // draw chairman edge from node3(id=5) to node14(id=23)
            g.drawLine(nodes[3].xNode+d/2, nodes[3].yNode+d, nodes
[7].xNode+d*2, nodes[7].yNode);
            g.drawLine(nodes[7].xNode+d*2, nodes[7].yNode, nodes
[14].xNode+d/2, nodes[14].yNode);
            g.drawString("chairman", nodes[14].xNode+10, nodes[7].yNode+
25);

            // draw chairman edge from node1(id=3) to node11(id=17)
            g.drawLine(nodes[1].xNode+d/2, nodes[1].yNode+d, nodes
[4].xNode+d*2, nodes[4].yNode);
            g.drawLine(nodes[4].xNode+d*2, nodes[4].yNode, nodes
[11].xNode+d/2, nodes[11].yNode);
            g.drawString("chairman", nodes[4].xNode+30, nodes[4].yNode+
40);

      } // end void paint
      ///////////////////////////////////////////////////////////////////


} // end class db
```

Interfaces

Main Menu

Home

Query

Database

Log File

Document

Contact

Help

Welcome to the GQL Processor
Graph-oriented Query Language processor

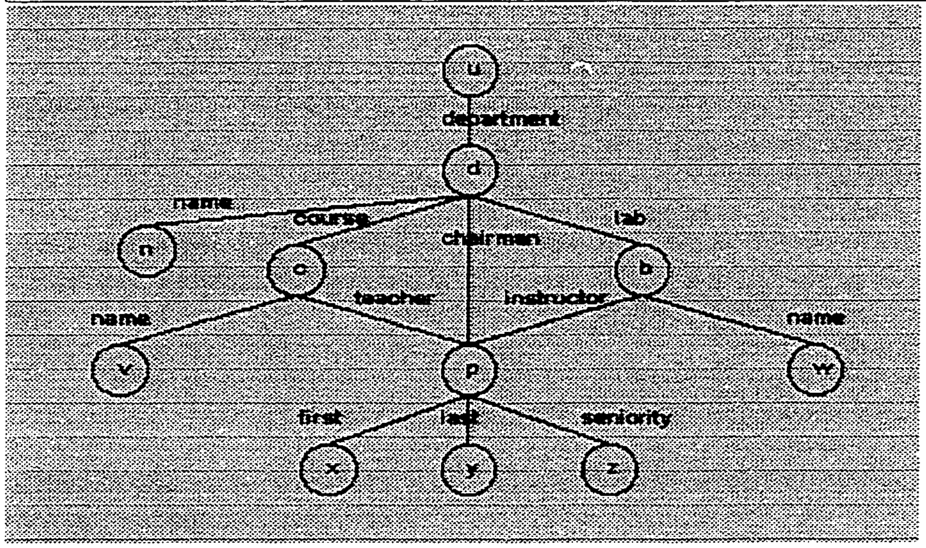
Instructions: To start using the application, please select an option from the main menu on the left. In order to use the application effectively and to understand how to perform valid meaningful queries it is **highly recommended** that first time users use the tutorial from the "Help" option in the main menu on the left. In order to simplify interaction with the application, it is recommended that you first use it without filter constraints (default option). At any time, you can restart the application by selecting the "Home" option from the main menu.

Purpose: This web site explores the practical aspects of a graph-oriented query language (GQL) for semistructured data (SSD). SSD is a schemaless data model in which there is no separation between the data and the metadata. It allows representation of heterogeneous data, information sharing and information integration. SSD is a data model represented as graph and a GQL seems to be a natural way to a query SSD graph. GQL is one way to query SSD and the project documentation discusses and analyzes this topic from both the theoretical and the practical points of view. For more select the "Help" option from the main menu.

Context: This web site is a product of a M.Cs graduation project and is part of the database group at Concordia University. It is based on: Yaron Kanza, Werner Nutt and Yehushua Sagiv, "Queries with Incomplete Answers over Semistructured Data", 1999, (Philadelphia USA) ACM Press. Since this is a theoretical research paper, the main focus of the project is the theoretical aspect (that can be found in the accompanying documentation) rather than the practical aspect provided here. However, such a partial implementation is important in order to analyze the proposed GQL, to compare it to other SSD query languages and to suggest new ways to improve it. To learn more about me and the people who guided and assisted me, select the "Contact" option from the main menu.

Users: This web site and the acoompanying research paper are targeted to a computer science person who wishes to learn what the SSD model is, and what the available query languages are by which this data model can be queried. The research paper explains why GQL is an appropriate query language for SSD, in what ways it fails to meet the SSD requirements and hence, how it can be improved to meet the SSD needs. For the full discussion of these issues, select (and download) the "Document" option from the main menu. The GQL web site is especially interesting for people who specialize in databases, the SSD model in particular. I gladly welcome any questions or comments. For feedback, you are encouraged to select the "Contact" option from the main menu.

Query Guide:



Query Parameters:

To make multiple selections, hold the Ctrl key while clicking.
To cancel a selection, click again while holding the Ctrl key.

Query nodes: [✓ all] u d n (nodes u and d should be selected)

chairman edge: [✓ No] Yes (in case nodes d and p were selected)

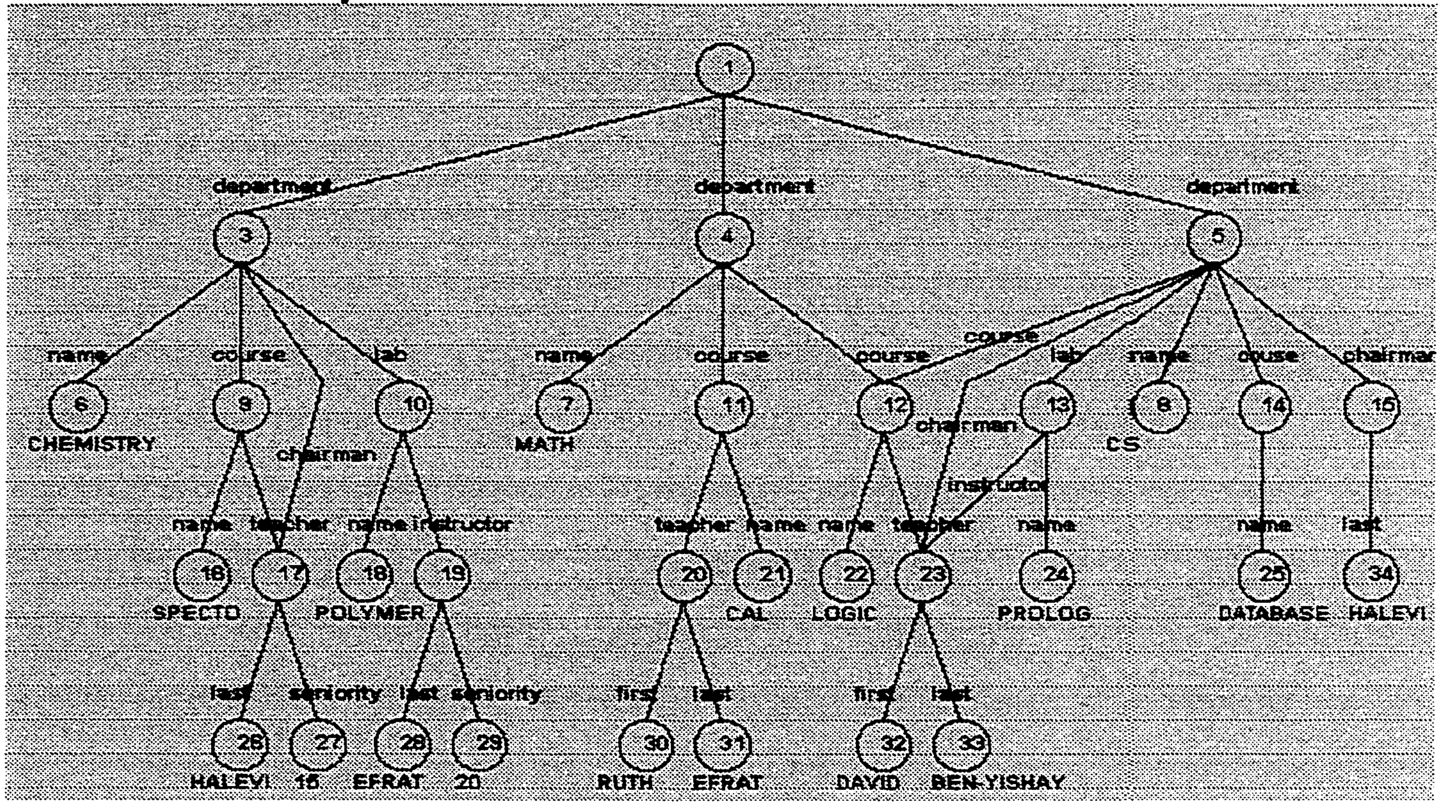
Query semantics: [✓ none] OR (in case node p has more than 1 edge)

Filter constraints: [✓ none] !n (dept name) !c (course) !b (lab) [✓ none] n=Chemistry n=Math n=CS

(existance) (comparison)

[illegible] [illegible]

The Hebrew University Dtabase



To save the recent query's input and output click here

Project Document

The project document discusses the graph-oriented query language (GQL) proposal in [KNS99] from both theoretical and practical points of view. It is accordingly divided into two part: (1) review and analysis of the GQL proposal and (2) implementation and discussion of the GQL operation. You can download a Miscrosoft Word copy of the document by clicking **here**. The following is the document's table of contents.

Table of Contents

[KNS99] Yaron Kanza, Werner Nutt and Yehushua Sagiv, Queries with Incomplete Answers over Semistructured Data,1999, (Philadelphia USA) ACM Press.

Please send your feedback / questions/ comments to:

Jimmy Elkada: elkada@cs.concordia.ca

My home page: www.cs.concordia.ca/~grad/elkada

My Supervisor: Prof. Grahne Goste grahne@cs.concordia.ca

My Advisor: Alex Thomo thomo@cs.concordia.ca

FAQ: (Frequently Asked Questions)

TUTORIAL: (the steps for using the application)

FAQ: (Frequently Answered Questions)

1. What is semi-structured data (SSD)?
SSD has recently emerged as an important topic of study in database. This data model enables heterogeneous data sources such as the Web to be treated as databases. Such sources cannot be constrained by a fixed schema. Also, due to information sharing and information integration in both the commercial and the academic fields, it is desirable to have an extremely flexible format for data exchange between disparate data sources. SSD is represented as labeled-edge graph in which nodes represent data objects and edges represent data attributes. Select "View database" from the main menu to see a SSD graph.

2. What is a query language for SSD?
A SSD query language is a programming language by which the semistructured database model can be searched and queried. Any SSD query language should maintain certain features such as path expressions, expressive operations, precise semantics, composionality, structure conscious and verbose language. Some current SSD query languages are XML-QL Lorel and UnQL which all use SQL like format. Unlike these query languages, this project explores a graph-oriented format as a query language.

3. What is a graph-oriented query language (GQL) for SSD?

In the proposed GQL, the query takes on a graph format. It seems that a graph-oriented query model is more natural and intuitive for the graph-oriented database model. In this model, the query is a graph whose nodes are variables that need to be matched by database values, according to given filter constraints.

4. Where can I get more information about SSD and GQL?
More information about SSD and GQL is provided in the project document. In this document, you can find analysis, summary and examples of both topics. For an even more detailed discussion refer to the "References" section of the project document.

5. How does the GQL work behind the scenes?
In the proposed GQL, queries are evaluated in two phases: (1) the search phase and (2) the filter phase. In phase 1, the database is searched for the pattern specified in the query graph (by the node variables). The result is a list of possible answers (pairs of query nodes and corresponding database nodes). These are called maximal matchings since they represent the maximal possible information that can be obtained from the given query. In phase 2, these maximal matchings are filtered in order to output only those matches that are needed according to the filter constraints given by the user.

6. How do I query the database?
To perform a query, simply select the "Query" option from the main menu and then enter the query parameters. If you just click the "Submit" button without entering any parameters, the default query is all query nodes under OR semantics.

7. How do I view and verify the query results?
After you click the "Submit" button, the query will be processed. The results will be displayed on the screen. At that point, you can select the "Database" option from the main menu to verify the results. The query results can be then viewed again by selecting the "Log file" option from the main menu. You can even download and save the query parameters and results for future reference.

8. What are the different query semantics?
For a tree query graph, all three semantics OR, AND and WEAK give the same answers. For an acyclic query, that has some nodes with more than one *incoming* edge, the query semantics depends on which one of these incoming edges must exist in the database in order for the node to be in the matching set.
- OR semantics - at least one of the query's node incoming edges must exist in the database in order for that node to be included in the answer.
- AND semantics - all of the query's node incoming edges must exist in the database in order for that node to be included in the answer.
- WEAK semantics - a query's node incoming edge and the source node of that edge must exist in the database in order for that node to be included in the answer.

9. What are filter constraints?
Filter constraints are restrictions on the query parameters that reduce the matching set to the solution set, such as the constraint department="Math". STRONG and WEAK constraint satisfaction and the three types of filter constraints are exaplained in the document. For simplicity, this project deals only with STRONG satisfaction in which all of the selected filter constraints must be satisfied.

10. How does GQL query translate into natural language?
There is no direct translation such as in SQL or XML-QL. In GQL, the graph query specifies desired path(s). In this sense, GQL is not programmer-friendly and certainly not user-friendly. Simple Internet users are not normally familiar with notions such as graph structures, paths, nodes and edges. However,

the intuitive graph format compensates in visualizing how the query relates to the database but not in the natural language translation. Also, see next answer.

11. What are the advantages/disadvantages of a GQL?
Select the "Document" option from the main menu and view section 6 "Criticism and Future Research" for a detailed discussion.

12. Where can I find the documentation for this project?
Select the "Document" option from the main menu. You can then view the table of contents of the present project documents and download it.

13. What does the project's document include?
The project document discusses the GQL proposal in *[KNS99] from both theoretical and practical points of view. It is accordingly divided into two parts: (1) review and analysis of the GQL proposal and (2) implementation and discussion of the GQL operation. To view the document's table of contents select the "Document" option from the main menu on the left.

14. Who are the people behind this project and web site?
As mentioned in the web site's home page, this site is part of my masters degree graduation project (Major Report) at Concordia University, Montreal, Canada. The project's title is "Graph-oriented Query Language for Semistructured Data". Select the "Contact" option from the main menu to learn more about me and the people who assisted me in this project.

15. How do I send questions, comments or feedback?
Your comments and suggestions are important for the future existence of this web site. Select the "Contact" option from the main menu to send any feedback and to learn more about me and the people who assisted me in this project.

TUTORIAL: (strongly recommended for first time users)

Note: (1) In any step, to make multiple selections hold the Shift key while clicking the Ctrl key and to cancel a selection click the Ctrl key again.
(2) Each step is explained later (click tutorial example link after step 7).

Step1: select "Query" option
To perform a query first select the "Query" option from the main menu on the left. You will then see the screen split into two parts. The upper part is the query guide, which is the largest query graph the user can select. The lower part is the area where users can enter the query graph criteria (Note that for simplicity and time constraints, users cannot create their own graph although more sophisticated user interface allowing that would be appropriate).

Step2: select query nodes
Select from the list box the graph nodes. Note that these selection create path patterns that can be then matched to the database. Nodes u and d should be selected in order to create meaningful queries.

Step3: select chairman edges
If you have selected node p, then you could specify chairman edge as one of its incoming edges. If node p

has more than one incoming edge, then the query graph becomes an acyclic graph instead of tree graph. In this case the next parameter, query semantics, is meaningful.

Step4: select query semantics
In case node p is selected and it has more than one incoming edge, the query graph becomes acyclic. Query semantics should be selected. The default semantics is Or semantics which gives the largest matchings set. The meaning of the three possible semantics is explained in FAQ#8.

Step5: select filter constraints
Select from the two types of constraints in order to narrow your search. This step is optional. The default is no filter constraints at all. For simplicity only STRONG satisfaction of two simple filter constraint types is implemented. The meaning of filter constraints is explained in FAQ#9.

Step6: click the "Submit" button
To send your query for processing, click the "Submit" button. The query answers will be then displayed on the screen and will be available from the "Log file" option in the main menu.

Step7: verify the query results
In order to understand the evaluation process and analyze the results you should verify what nodes are returned in the solution set. By selecting the "Database" option in the main menu, you can review the database SSD graph. You can also select the "Log file" option from the main menu in order to view and/or save the query parameters and its solution.

Tutorial example

---

*[KNS99] Yaron Kanza, Werner Nutt and Yehushua Sagiv, "Queries with Incomplete Answers over Semistructured Data",1999, (Philadelphia USA) ACM Press.

Example: Query Input and Output

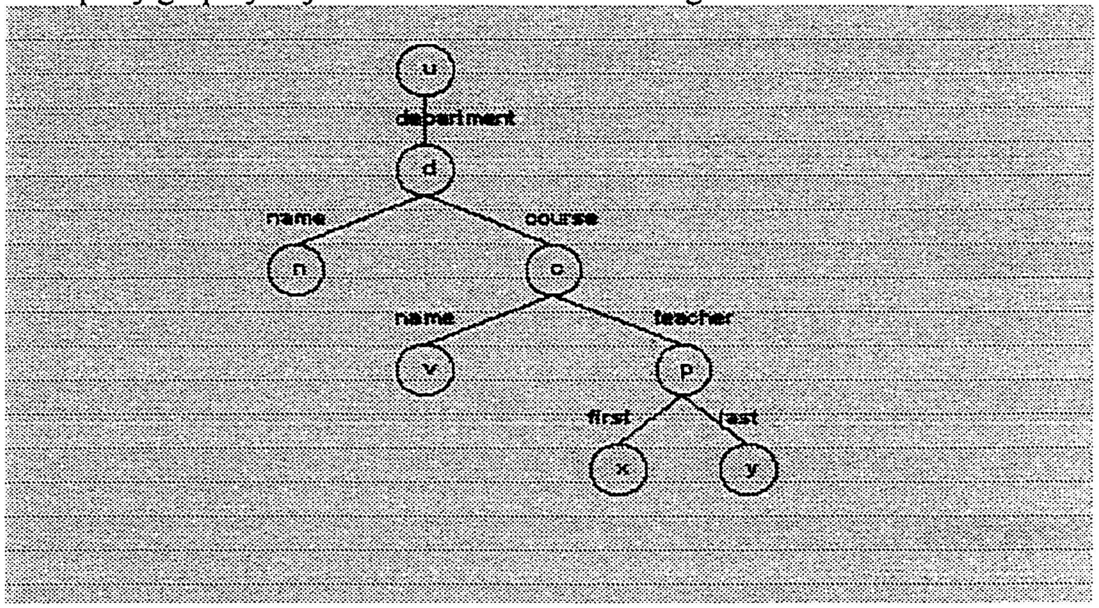
Note: If at any point the graph drawing is fuzzy (unclear), go to the middle of that graph and click the "Refresh" button on your browser

step1: select "Query" option
First select the "Query" option from the main menu.

step2: select query nodes
Suppose you select nodes u, d, n, c, v, p, x, y.

step3: select query edges
Nodes p and c were selected so "teacher" edge is added automatically to the query graph. In this step you can add an additional edge "chairman" by selecting "Yes" and then the query graph become a dag instead of a tree. For simplicity, suppose you select "No" for chairman edge in this step.

The query graph you just selected is the following:



step4: select query semantics
Node p was selected but with only one incoming edge so ignore this step. This is because for tree graphs all three semantics give the same result.

step5: select filter constraints
This step is optional, but suppose you want an existance constraint !x (node x, first name, must exist) and an object comparison constraint n="MATH". Thus select these from the two list boxes of the Filter constraints parameters. At this point the query can be phrased in natural language as follows:
"give me teachers' first and last names in the Math department (where first name exists)"

step6: "submit" the query
Click the "Submit" button to send the query paramrters for processing. If you wish to renter the parameters click the "Clear" button. Few seconds later you will get the query solutions (filtered matching sets in a form of a table). For the above query you will get the following table:

	u .	d .	n dept	c .	v course	p .	x first	y last
answer 1	1 .	4 .	7 Math	11 .	21 Cal	20 .	30 Ruth	31 Efrat
answer 2	1 .	4 .	7 Math	12 .	22 Logic	23 .	32 David	33 Ben-Yishay

step7: verify the query results

You can verify that in the Math department there are two matchings that satisfy the above query. In this case the existance filter constraint has no effect becuase both matchings have first name in the Math department.

If the above query did not have the comparison constraint n="Math", then the answer would be as below (note that the database course is not included because it does not have a teacher node and thus the existance constraint !x is not satisfied) :

	u .	d .	n dept	c .	v course	p .	x first	y last
answer 1	1 .	3 .	6 Chemistry	9 .	16 Specto	17 .	-- .	26 Halevi
answer 2	1 .	4 .	7 Math	11 .	21 Cal	20 .	30 Ruth	31 Efrat
answer 3	1 .	4 .	7 Math	12 .	22 Logic	23 .	32 David	33 Ben-Yishay
answer 4	1 .	5 .	8 Cs	12 .	22 Logic	23 .	32 David	33 Ben-Yishay