IMPLEMENTING VISUAL QUERIES AND PRESENTATIONS WITH BLOBS

XUEDE CHEN

A THESIS

IN

THE DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE AT
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

DECEMBER 2004

Your file Votre référence
ISBN: 0-494-04441-1
Our file Notre référence
ISBN: 0-494-04441-1

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Xuede Chen**

Entitled: **Implementing Visual Queries and Presentations with Blobs**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair

______________________________ Examiner

______________________________ Examiner

______________________________ Supervisor

Approved by ______________________________.
Chair of Department or Graduate Program Director

______ 2004 ______________________________.
Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

ABSTRACT

Implementing Visual Queries and Presentations with Blobs

Xuede Chen

Diagrammatic Query Tool (DQT) is a visual data query and presentation interface that helps better exploit and understand large, complex collections of data.

This thesis describes the design and implementation of the graphical user interface of DQT with focus on blob support for query construction, blob support for result visualization, automatic translation of query results into DOT format, and presentation of query results with clarity.

The implementation of Blobs enables DQT to support the full representative capability of hygraphs and the complete set of Graphlog. With Blob support, users can generalize collections of relationships between one entity and one set of related entities, which makes the construction of queries and visualization of query results more clear, and therefore enables the users' to recognize and discover interesting patterns more easily and intuitively.

Graphviz is used to layout query results. Query results must be in the DOT format for Graphviz to produce layouts. Automatic translation of query results into DOT-format graphs has been implemented. This work makes DQT usable by users who do not have knowledge of TGL and DOT.

Clarity of result layout and presentation has been achieved by assigning layout properties at the schema level, providing a legend graph for the query result graph and ranking objects in the result graphs.

Acknowledgements

Many thanks to my supervisor, Dr. Butler, for his great support and advice on the thesis work, as well as his understanding during the past several years.

Special thanks to Ms. Yue Wang for her solid foundation work and for her wonderful help during the thesis project.

Thanks to Dr. Grogono and Dr. Rilling for their valuable comments on the thesis that help to improve its quality.

Thanks to Ms. Halina for her help and reminders that I have received during the years. Thanks to the Graduate Program Directors and all the instructors and staff who have taught and/or helped me during my studies at Concordia.

Table of Contents

List of Figures

1. Introduction

1.1 Motivation

Data query and presentation play a fundamental role in helping users solve complex, information-intensive problems in scientific, engineering and business applications. However, the database query languages, such as SQL, are difficult for non-database-experts to use. Untrained users often find database query interfaces frustrating and even trained database users frequently have difficulty analyzing the results of queries. Despite about 30 years of research in this area, these problems still persist. Making databases easier to use, and thereby more accessible, is an important issue today and will become more important as database technology becomes faster, cheaper, and more powerful [1, 2].

Graphical visualization of data queries and presentations is more intuitive and more effective for users who are domain-specific experts but not technical database experts. Scientific visualization has been very successful in enhancing the ability of people to understand quantitative multi-dimensional data sets [3, 4]. Graph drawings are one of the best ways to present technical information and are particularly appropriate for showing relations between objects. Research has recently been active in graph-based browsing and querying of relationships of abstract, structured data [5, 6, 7, 8, 9, 10, 11, 12, 13, 14].

A visualization formalism based on graphs has been developed at the University of Toronto, which includes the *hygraph* data model and the *GraphLog* query language [8, 9, 15]. Hygraphs are graphs augmented with *blobs*. In hygraphs, nodes represent entities and are labeled by identifiers and attributes (encoded as first-order form); edges represent relations and are labeled by path expressions on relationships that optionally carry additional arguments (encoded as literals). A blob relates to a containing node with a set of contained nodes; it replaces all the edges that would otherwise connect the container node of the blob with each of the nodes contained in the blob. In short, a hygraph has nodes that represent objects, and has both edges and blobs to represent relationships among those objects. GraphLog is a graph query language using hygraphs as the underlying data structure. GraphLog supports aggregate operators (in particular, recursive aggregation or transitive closure) and path expressions (similar to regular expressions). Therefore it is more expressive than SQL. In GraphLog, there are two kinds of queries: *filter* and *define*. Filter queries request a subset of the database facts to be retrieved; define queries create new relationships from the existing data. The visualization formalism was implemented as the Hy+ system [9]. Hy+ allows users to compose graphical queries in GraphLog and view the query results in hygraphs. However, the Hy+ system has not been deployed in real applications although the underlying visualization formalism is sound and powerful. One of the major reasons is that Hy+ is based on an old Smalltalk environment.

A project has been established at Concordia University to design and implement a practical diagrammatic query tool, called DQT [16]. The project has been motivated by the evolution of technologies, the case study of the Know-It-All framework, and the demand of real applications: (1) The art of design of graphical user interfaces (GUI) has evolved much since Hy+ was implemented in early 1990's. Java is becoming more and more popular in the development of user interfaces. Java, as a pure object-oriented programming language, offers the benefits of platform independence and look-and-feel support, while absorbing the advantages of C++ constructs and the Smalltalk environment [17]. We are interested to design and implement a Hy+-like system based on GUI design principles and the Java platform. (2) The Know-IT-All project, which has been underway at Concordia University since 1997, is investigating methodologies for development, application, and evolution of frameworks. A concrete framework for database management systems is being developed as a case study for the research on methodologies. A diagrammatic query and visualization user interface is required for the case study. (3) Bioinformatics is the driver for many of the discoveries from genomics. Data management, access, and mining are at the heart of bioinformatics. In order to enable scientist to more easily query and better visualize bioinformatics data, a diagrammatic query and presentation tool is required by the research project on bioinformatics at Concordia University.

1.2 DQT Overview

The Diagrammatic Query Tool (DQT) is a visualization and query tool for databases based on the hygraph and GraphLog formalisms. It uses the hygraph data structure and the GraphLog query language. The tool can help users better exploit and understand large, complex collections of structural data. DQT GUI uses the TGL Translator to work with the CORAL deductive engine and the underlying (relational, object-relational, or object) database systems. The system architecture is shown as Figure 1.

DQT provides users with a graphical interface to allow users to open and view schemas, to edit and submit GraphLog queries, and to browse the results of queries. The

TGL Translator is responsible for translating the user queries in GraphLog into logic programs acceptable to CORAL, and translating the results back to hygraph. The TGL Translator provides services for DQT GUI.

CORAL is a deductive engine [18]. CORAL evaluates logic programs against the underlying databases to deduce the facts that satisfy the conditions and rules defined by the logic programs. The underlying databases for CORAL could be relational. In our case, the underlying database platform is the MySQL database system.

The work in this thesis focuses on the design and implementation of the DQT GUI.

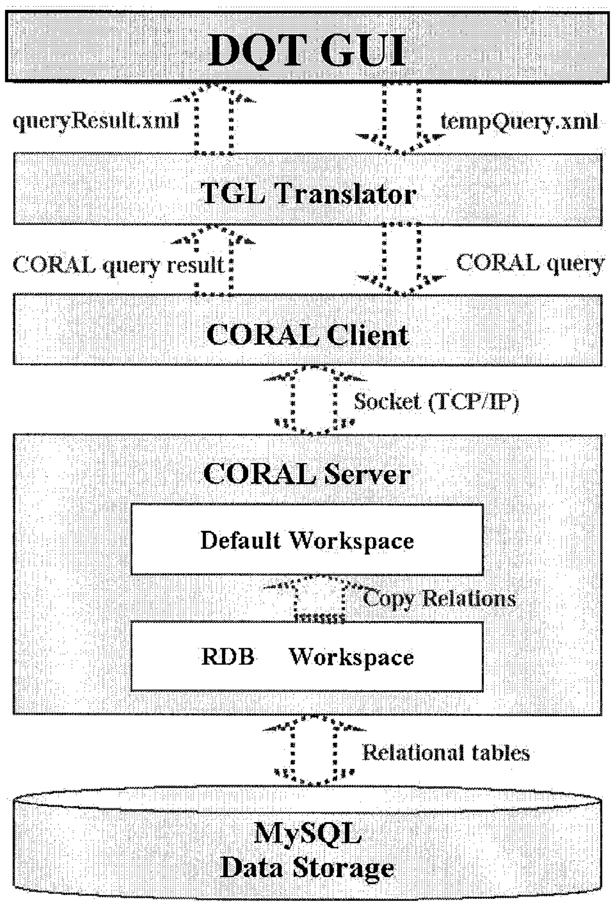


Figure 1: DQT: Diagrammatic Query Tool for Graph databases [23]

1.3 DQT in the Know-It-All Framework

Know-It-All is an object-oriented framework for database management systems [16]. The Know-It-All framework has three sets of aims: (1) to research methodologies and models for framework development, application, and evolution; (2) to develop a framework for database management systems, supporting a variety of data and knowledge models, integration of different paradigms, and heterogeneous databases; (3) to apply the Know-It-All framework to advanced applications for bioinformatics. For all the three aims, graph databases and graph query languages are one of the most important paradigms in the framework

DQT and GraphLog views will be integrated into the Know-It-All framework for databases. DQT will be an important user interface to enable users to query databases in GraphLog and browse the results of queries in the form of hygraphs. GraphLog and hygraphs will be implemented as a VIEWDB subclass of relational, object, or object-relational databases. CORAL is used as the deductive engine between DQT and the underlying databases for the moment, but this will be de-coupled when the intended hygraph databases and facilities are in place. Then DQT will evaluate queries in GraphLog directly against the hygraph databases (see Figure 2).

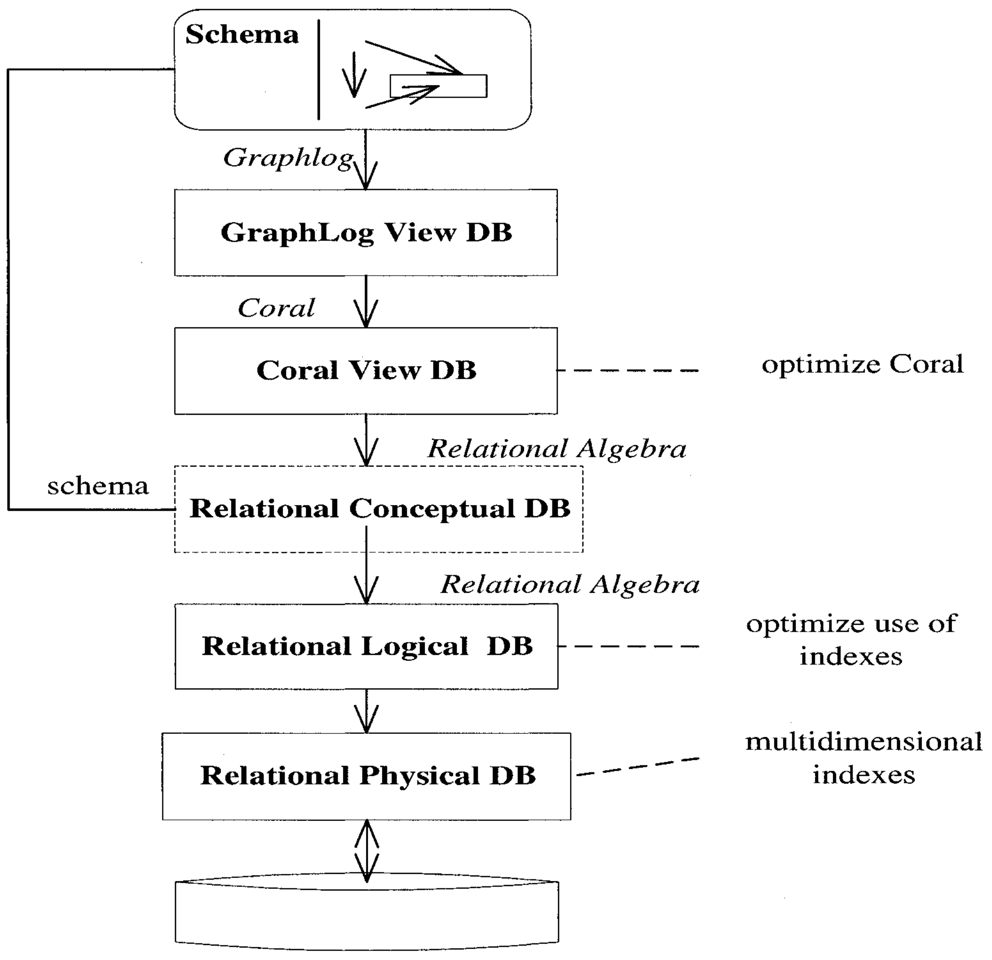


Figure 2: DQT as a User Interface in the Know-It-All Framework

1.4 Intended Applications to Bioinformatics

Data management, access and mining are at the heart of bioinformatics. Most data access today in genomics is provided by point-and-click interfaces on icons for canned queries, or by filling in forms for parameterized sets of canned queries, or by SQL-like textual notations for more advanced or flexible querying. These are solutions tailored to the underlying database technology, rather than solutions tailored to the scientists.

Untrained users often find database query interfaces frustrating and even trained database users frequently have difficulty analyzing the results of queries. Diagrams are a more intuitive way for scientists to pose queries to relational, object-relational, and object databases. Diagrammatic queries are particularly appropriate for interactions as found in databases for metabolic pathways, protein-protein interactions, and gene regulations.

In addition, while relational databases are the accepted standard within industry [19], there has been considerable research into deductive databases and graph databases to extend the capabilities of relational databases. Deductive databases allow a view, called the *intentional database*, to be defined using logical rules, and allow logical queries against the view. Since the rules allow recursive definitions, the resulting expressive power of the query language is greater than ordinary relational databases. Graph query languages are even more expressive, while having the very important property of a visual representation.

DQT is a GraphLog-based diagrammatic query tool. It is intended to apply DQT to bioinformatics. It allows for a broader range of queries, from very simple queries to the very complex ones. It also presents the query results in graphs that are easier comprehendible by the genetic scientists than SQL-like languages or form-based queries. DQT will be an elegant tool for data mining in bioinformatics.

1.5 The State of DQT Development

The DQT project has been developed incrementally since initiated by Dr. Butler in Spring 2001. The goal setting, requirements analysis, and architectural design have been conducted [16], and a simple GUI prototype has been developed during Summer 2001 [20]. The development of the CORAL interface and GraphLog translation component, which bridges the DQT GUI component and CORAL system, has been completed by 2003 [21, 22]. And a working version of DQT without Blob support or result translation has been available and applied to visual queries of a graph database for genomics by early this year [23]. So far, the DQT GUI allows an end user to import a database schema, define new relations and construct queries by drawing graphs that conform to GrapgLog, but consist of nodes and edges only, no blobs. The GUI will capture the user's query along the new definitions if any in TGL format and route them to TGL Translator, in turn, to CORAL, for evaluation. Then query results are manually translated into graphs in the Graphviz DOT language so that the query results can be visualized by calling Graphviz, and presented to the user as images.

The DQT GUI currently does not support blobs, which are the essential difference of hygraphs from ordinary graphs and important parts of GraphLog. And DQT currently does not provide automated translations of query results from TGL into Graphviz dot graph – currently this must be done manually and limited to maximum three nodes in the results. In addition, it is also desired to integrate Graphviz into DQT to make the query and presentation process more streamlined, therefore friendlier to users.

1.6 The Thesis Work

The thesis is to enhance the functionality of DQT by providing blob support in query construction and result presentation, implementing automated translation of query results into Garphviz DOT graph, and streamlining the query and presentation process.

Blobs, as hygraphs' extension to graphs, are generalizations of edges and can be used to cluster related nodes together. A blob in a hygraph represents a relation between a node, called the container node, and a set of other nodes, called the contained nodes. A blob replaces all the edges that would otherwise connect the container node of the blob with each of the nodes contained in the blob; that is, it modularizes the set of contained nodes and their associations with the container node. Blobs can help reveal interesting characteristics of data while avoiding distractions and irrelevancy, and provide effectiveness and clarity in construction of graphical queries and visualization of query results; therefore help the users discover interesting patterns more easily and intuitively. Blob support in DQT is very important for the project on genomics. One of the major tasks in this thesis is to provide blob support in DQT, including supporting users to construct new relations and queries with blobs, capturing the user queries and translate them into TGL, and formulizing query results with blobs and visualizing the results using Graphviz. This work will enable DQT to have a full support for Hygraph and GraphLog paradigm.

Graphviz has been chosen for layout of the query results. To make it work, DQT must provide the functionality of automatically translating query results in TGL format into the dot graph format, which is the format Graphviz takes as input. It is unlikely for non-Graphviz and non-TGL experts to use DQT without this functionality. This thesis will enhance DQT's functionality and streamlines the query and presentation process with automated translations and integration of Graphviz into DQT.

The thesis will summarize the background knowledge and context in Chapter 2, present the design and implementation of constructing new relations and queries with blob support in Chapter 3 and the approach of visualizing blobs with Graphiviz in Chapter 4, and describe the automatic translation of query results for visualizations in Chapter 5. The conclusions will be drawn in Chapter 6.

2. Background

DQT is a diagrammatic query tool based on *GraphLog* and *hygraph*. DQT captures and translates user queries into TGL (Transferable Graphical Language), and visualize the query results with *Graphviz*. Java Swing and Java 2D are used for the implementation of DQT. In this chapter, we summarize the required background knowledge.

2.1 GraphLog and Hygraph

GraphLog is a graph query language based on hygraphs [8]. It was originally developed in the Hy+ system at the University of Toronto and has shown many advantages over other paradigms. The language is very suitable for querying and visualizing structural data [15].

A hygraph is a *hy*brid between Harel's *higraph* [24] and directed *hy*pergraphs [25] (and hence the name); it extends the notion of a graph by incorporating *blobs* in addition to edges. In hygraphs, nodes represent entities and are labeled by identifiers and attributes (encoded as first-order form); edges represent relations and are labeled by path (regular) expressions on relationships that optionally carry additional arguments (encoded as literal). A blob relates a containing node with a set of contained nodes; it replaces all the edges that would otherwise connect the container node of the blob with each of the nodes contained in the blob. Blobs are diagrammatically represented by a closed curve that is associated with the container node and that encloses the contained nodes. Figure 3 is a hygraph for the NIH class hierarchy; Figure 4 shows a partial hygraph of the hierarchy and its representation by blobs. In short, a hygraph has nodes that represent objects, and has both edges and blobs to represent relationships among those objects. For completeness, here is a formal definition of hygraphs [15].

Definition 1: A *hygraph H* is a septuple

$$(N , L_N , v , L_E , E , L_B , B)$$

where: N is a finite set of *nodes*; L_N is a set of *node labels*; v, the *node labeling function*, is a function from N to L_N that associates with each node in N a label from L_N; L_E is a set of *edge labels*; $E \subseteq N \times N \times L_E$ is a finite set of *labeled edges*; L_B is a set of *blob labels*; and $B \subseteq N \times 2^N \times L_B$ is a finite set of *labeled blobs*.

A restriction is placed in the labeled blob relation B to ensure that there is only one tuple (n, N, l) in B with the same values for the *container node n* and the blob label l (i.e., the container node and blob label values functionally determine the value of the set of *contained nodes N*, so B can be considered as a function $B : N \times L_B \rightarrow 2_N$). □

GraphLog is a graph query language using hygraphs as the underlying data structure. A GraphLog query is a finite set of hygraphs. There are two kinds of queries: *show* and *define*. A *show query* requests a subset of the database facts to be retrieved; a *define query* creates a new relationship from the existing data. Formally, GraphLog *define queries* are hygraphs with no isolated nodes having the following properties:

(i) the nodes are labeled by terms,

(ii) each edge and blob is labeled by a literal (either an atom or a negated atom) or by a closure literal, which is simply a literal s followed by the positive closure operator, denoted s^+, that can only appear between nodes labeled by sequences of the same length, and

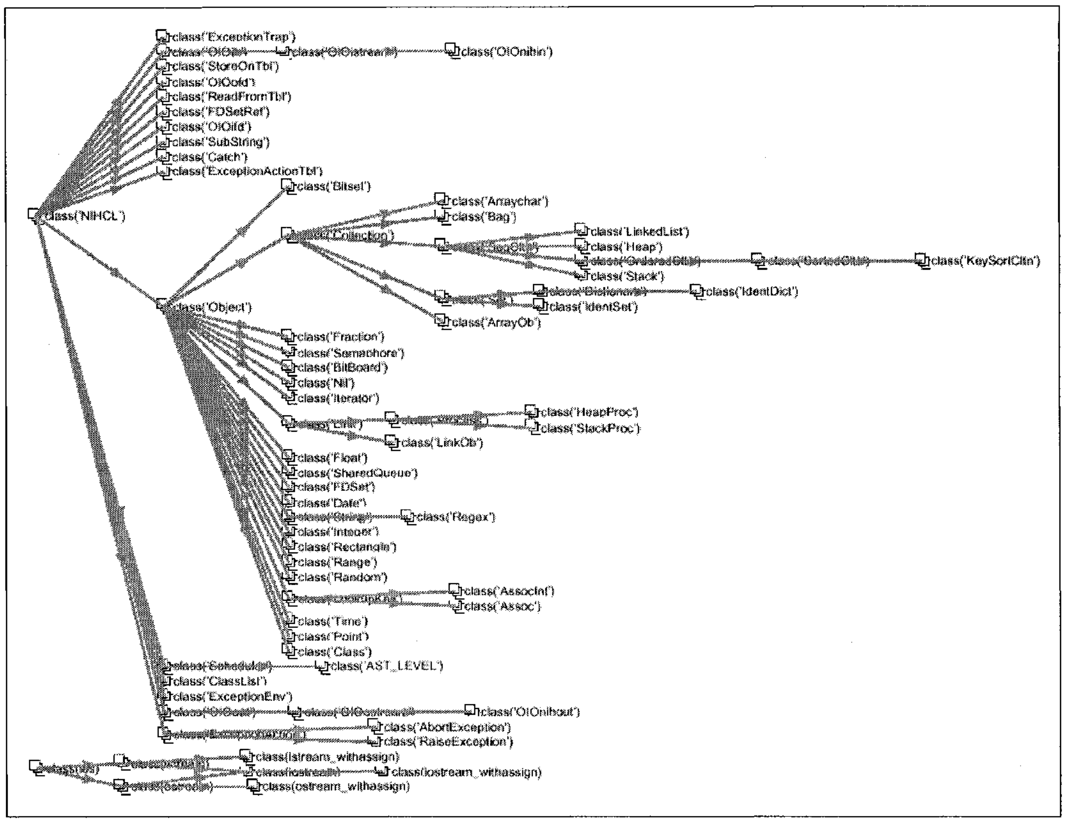


Figure 3: Hygraph for the NIH Class Hierarchy [15]

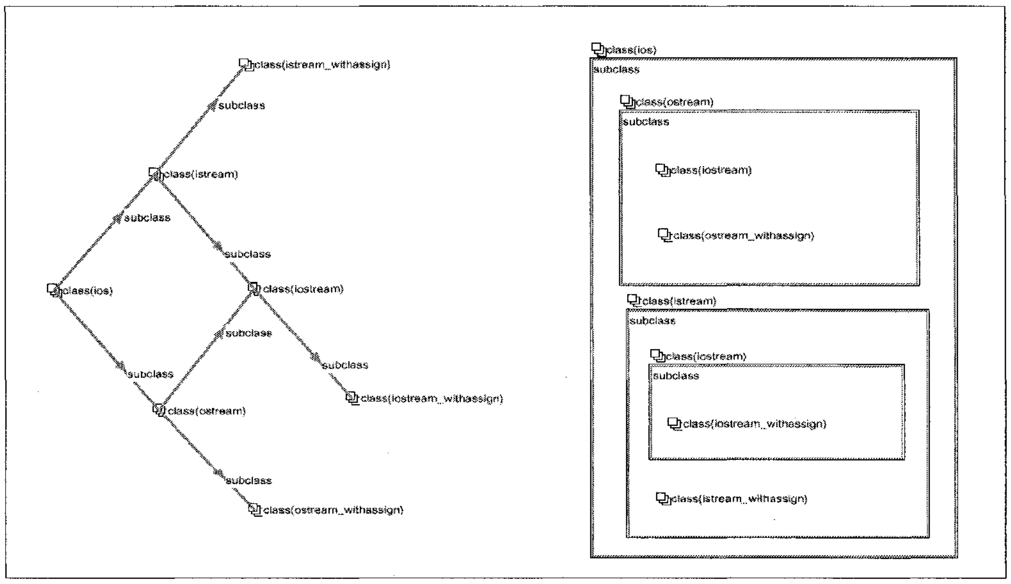


Figure 4: Blob Representation of a Hygraph [15]

(iii) there are one or more *distinguished* edges and blobs (drawn thicker), which can only be labeled by positive non-closure literals.

In addition to the usual operators for positive and Kleene closure, optional (i.e., the operator ? denoting zero or one occurrence), alternation, and concatenation, two new ones are defined: *inversion* reverses the direction of the edge or blob labeled by the regular expression, and *negation* negates the predicate defined by its argument. GraphLog *show queries* are analogous to *define queries*, except that:

(i) nodes can also be distinguished (and they have a special unary predicate associated with them, hence isolated nodes are allowed), and

(ii) non-negated path regular expressions can label distinguished edges and blobs.

To summarize the syntax of GraphLog, a term is either a constant, a variable, an anonymous variable (an underscore), or a function *f* applied to a number of terms. Nodes are labeled by terms. Edge or blob labels are expressions generated by the following grammar [15]

$$E \rightarrow E \mid E;\ E{\cdot}E;\ -E;\ \neg E;\ (E);\ E+;\ E^*;\ E?;\ S$$

where S is any literal of the form $p(t_1, \ldots, t_n)$ and t_i, $1 \leq i \leq n,$ are terms.

A simple GraphLog query that displays the class hierarchy is shown in Figure 5. The pattern consists of a thick edge labelled `subclass` between two nodes labeled `class(C1)` and `class(C2)`, enclosed in a box labeled `showGraphLog`. The meaning of the show pattern is: match all facts of the form `subclass(C1,C2)` and display them as edges. The symbols `C1` and `C2` starting with capital letters are used (in

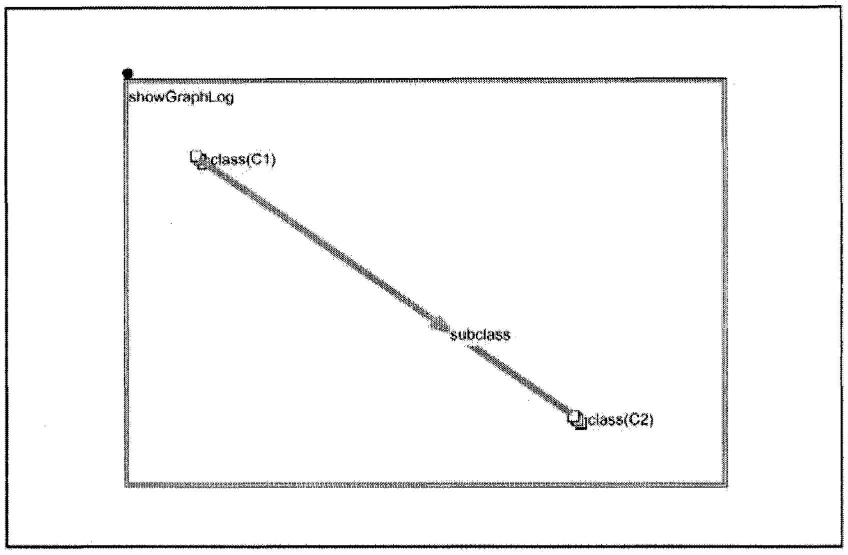


Figure 5: Displaying the Class Hierarchy [15]

the logic programming tradition) to denote variables. Consequently, all possible subclass edges are displayed, producing the visualization of the NIH class hierarchy shown in Figure 3.

An example of a define query is presented in the leftmost box in Figure 6. The pattern consists of two nodes labeled `class(C1)` and `class(C2)` and two edges connecting them, labeled `subclass+` and `all_subclass`. Thickness is used to distinguish edges, therefore the edge labeled `all_subclasses` is known as a *distinguished edge.* The meaning of the define query is: First, match the transitive closure of the subclass relation, as indicated by the non-distinguished edge labeled `subclass+` (where + is the closure operator). Second, for each pair of classes in the transitive closure of the subclass relation create a new edge labeled `all_subclasses` between them. An `all_subclasses` edge directly connects each class to all of its subclasses (both direct and indirect). These newly defined edges are then considered to be part of the current database. The show box to the right of Figure 7 is to display the `all_subclasses` edges just defined that originate at `class('Collection')`.

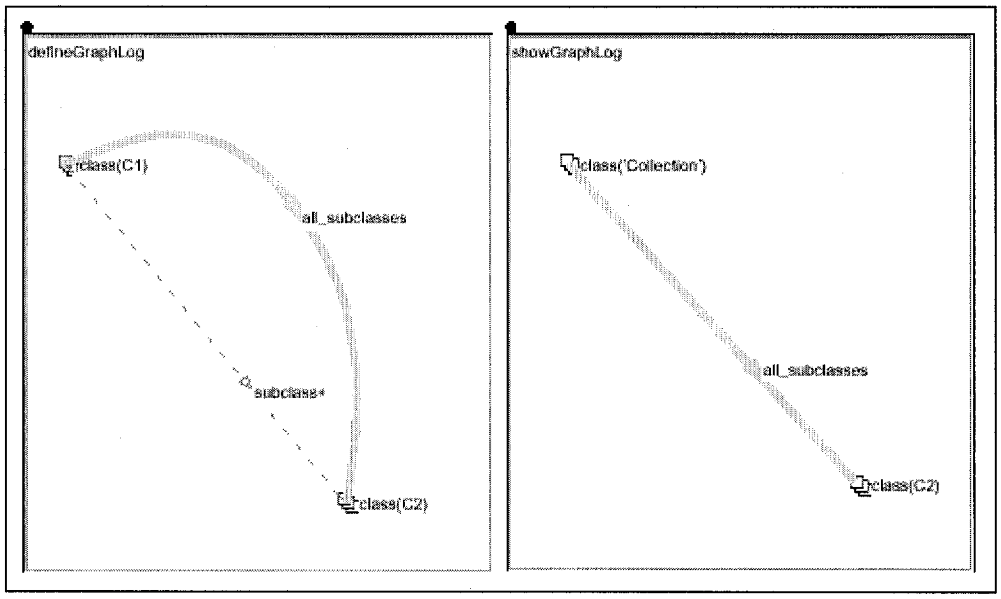
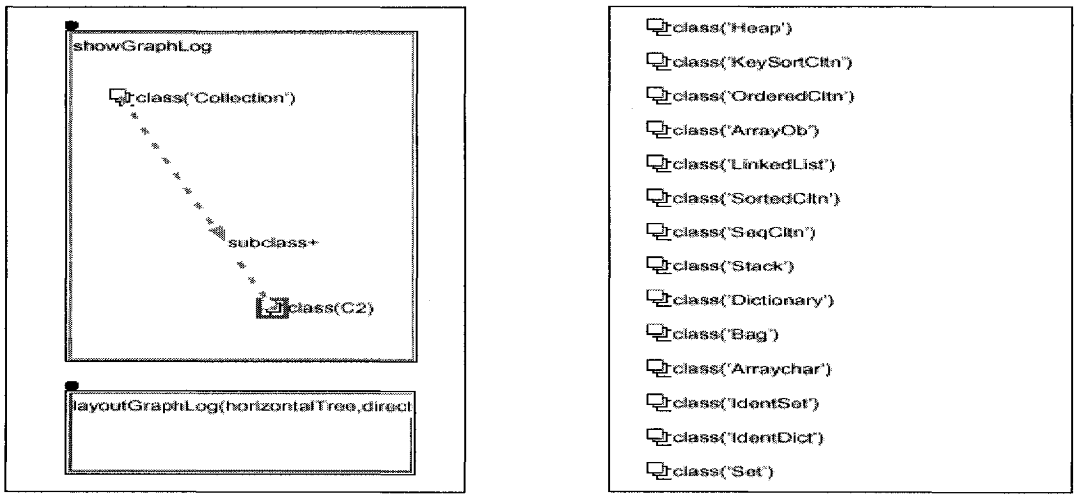


Figure 6: Defining and Showing `all_subclasses` of `Collection` [15]

The show pattern in Figure 7 has a *distinguished node* (labeled `class(C2)`), but the edge is not thicker. This illustrates how distinguished elements in show patterns are used to identify which objects from the matched pattern should be displayed in the answer, while leaving out the non-distinguished portions (in this case the remaining node and edge). Consequently, the result is simply a list of classes.

GraphLog has higher expressive power than SQL; in particular, it can express, with no need for recursion, queries that involve transitive closures or similar graph traversal operations. The language is also capable of expressing first order aggregate queries as well as path expressions [15].



```
class('Heap')
class('KeySortCltn')
class('OrderedCltn')
class('ArrayOb')
class('LinkedList')
class('SortedCltn')
class('SeqCltn')
class('Stack')
class('Dictionary')
class('Bag')
class('Arraychar')
class('IdentSet')
class('IdentDict')
class('Set')
```

Figure 7: Obtaining the Set of Subclasses of `Collection [15]`

2.2 Hy+ System

Hy+ allows users to compose graphical queries in GraphLog and view the query results in hygraphs [9]. Figure 8 shows a screen shot of a Hy+ session. The top leftmost window (labeled Hy+) is used to control the Hy+ environment and open other windows. One of them is a Hy+ File List, from which the user can open different editors on the contents of files. The window labeled File Editor shows six facts from the NIH database mentioned above. The same facts are displayed in the bottom left window by a Hy+ browser that has extensive facilities for interactively editing hygraphs. Hy+ assigns colors to edges based on the predicates in the edge labels and different icons to nodes based on the functions labeling the nodes. The Hy+ Palette Editor and Hy+ Icon Editor windows in the bottom right corner of the screen let the user select the colors corresponding to the predicates from a palette and pick icons for the functors by grabbing an image from anywhere in the screen. Color-coding relations and assigning different icons to nodes based on the objects they represent, are capabilities that Hy+ makes directly available to end users from customizing the hygraph visualizations to the semantics of the application.

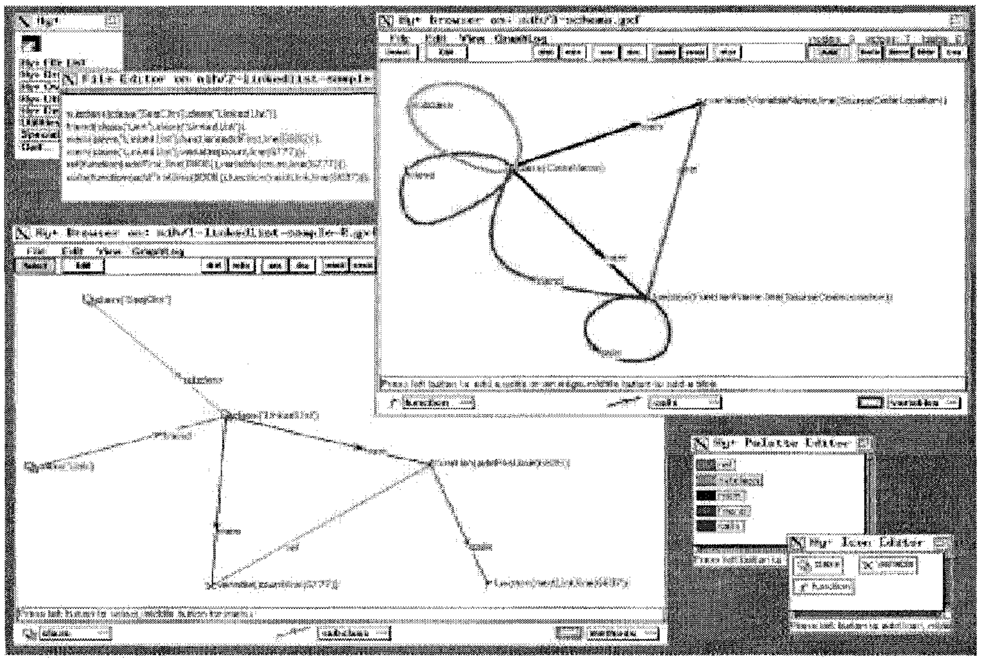


Figure 8: A Hy+ Screen for the NIH Scenario [15]

An overview of the Hy+ system architecture is given in the diagram in Figure 9. In Hy+, visualizations are based on a graphical formalism that allows comprehensive representations of databases, queries, and query answers to be interactively manipulated. Hy+ accepts graph queries represented by *GraphLog*, translates them into logic programs suitable for execution by one of the backend engines: the logic programming language LDL and the deductive language *CORAL*, and presents the results in *hygraph*. Hy+ are implemented by twelve categories of classes in Smalltalk.

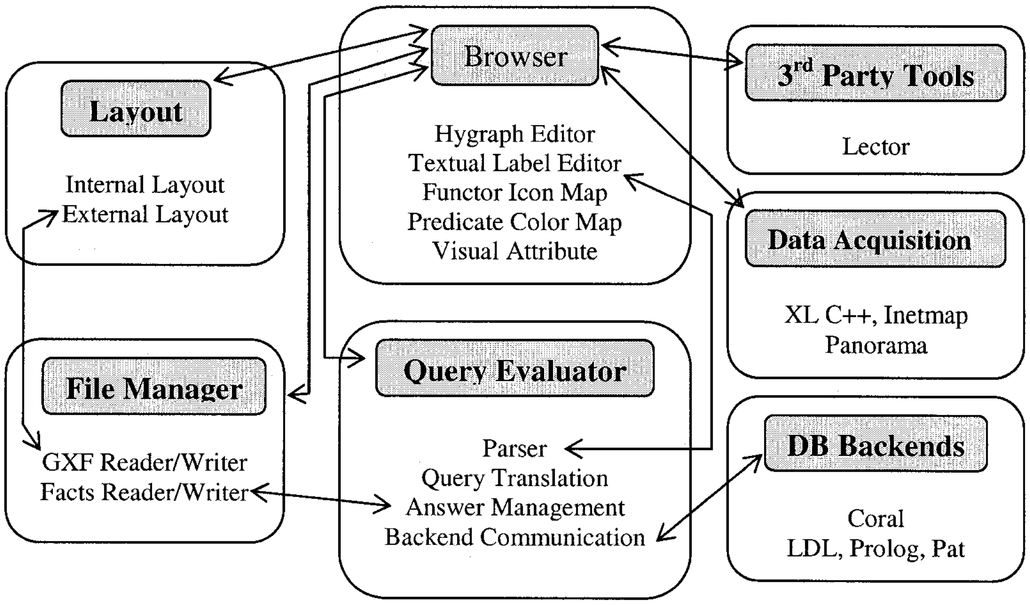


Figure 9: Hy+ Architecture Overview [15]

Hy+ has implemented the hygraph and GraphLog visual formalism of structural data and demonstrated its applications to large scale software engineering, network management, and distributed and parallel debugging. Graphlog is more expressive than SQL. However, the old Smalltalk environment might have limited further applications of Hy+ system to other areas.

2.3 Other Systems for Diagrammatic Queries

Several other database languages or systems have also used graphs as their underlying data structure and have aimed to provide a uniform approach to representing, querying and, possibly, updating both schema and data [10, 11, 12, 13, 14]. An overview of the systems, along with the Hy+ system, is presented in Figure 10. Among these systems or languages, the Hyperlog language [14] and its hypernode model are more comparable to Graphlog and hygraphs of Hy+ [8, 9].

System or Language	**Data Model**	**User Query**	**Query Evaluation**	**Result Presentation**
GOOD	Single flat directed graph	Graphs	Graphs	Graphs
G-Log	Single flat directed graph	Graphs	Rule-based	Graphs
Gql	Single flat directed graph	Graphs	Textual language	Textual output
Hyperlog	Hypernodes (nested graphs)	Template set	Rule-based	Hypernode set
Hy+ or GraphLog	Hygraphs (augmented graphs with blobs)	Hygraph patterns	Logic program	Hygraphs

Figure 10: Overview of Data Visualization Systems

GOOD, G-Log, and Gql regard a database as a single flat directed graph. Nested graphs are not allowed in these languages. In GOOD [10], queries are graphs, which match subgraphs of the database graph, and programs consist of sequences of patterns. G-Log queries are also graphs, but the query evaluation is based on ordered rules [12]. Gql provides a graphical representation of queries, which are translated into a textual database language for evaluation [11]. The output to a Gql query is the textual output of the underlying textual query, rather than a set of graphs.

Hyperlog [14] allows nested graphs as hypernodes, while Hy+ uses blobs for that. In Hyperlog, a query is a set of templates, which are translated into a set of literals of three predicates, *hypernode(-), node(-,-),* and *edge(-,-,-),* and are to be matched against a selected domain of the database. The result of the query is a set of hypernodes obtained by applying each match for the query to its set of templates. A Hyperlog program is a set of rules. The evaluation of program consists of repeatedly matching the bodies of its rules against the current database state and updating this state with the information inferred until no more new information is inferred.

Comparatively, GraphLog is a more powerful, intuitive diagrammatic query language.

2.4 Java Swing and Java 2D

Java Swing is a graphical user interface component kit, part of the Java Foundation Classes (JFC) integrated into Java 2 platform. Swing simplifies and streamlines the development of applications by providing a complete set of user-interface elements written entirely in the Java programming language [17].

One of the most important capabilities of the Swing toolkit is its *pluggable look and feel* (PL&F) -- a feature that it lets developers choose the appearance and behavior (or the look and feel) of the windowing components. Swing's look and feel standards promote flexibility and ease of use in cross-platform applications. The Swing toolkit provides a default set of look-and-feels, which includes three basic looks: Cross Platform (Java/Metal), CDE/Motif (Sun), and Windows (Win32). (A Mac L&F for Macintosh systems is also available, as a separate download). With Swing's PL&F capabilities, developers can explicitly specify which look will be used, or get the actual class name and make the application have the native look and feel for whatever platform the user runs the program on. If a program does not specify a look and feel or the specified look and feel is not available on the user's computer system, Swing's default behavior is to use the Java (Metal) look and feel. Furthermore, Swing also allows developers to create their own customized Swing components – or even complete sets of customize Swing components – that can have any kind of appearance and behavior that the developer can dream up.

Swing components are lightweight. Swing components do not use any platform-specific implementation. Instead, Swing creates its components using pluggable look-and-feel modules that are written from scratch and do not use any platform-specific code at all. Consequently, Swing components use fewer system resources and produce smaller and more efficient applications than their heavyweight AWT counterparts. Some of the Swing components used by the DQT GUI are listed in Figure 11.

Java 2D extends the graphics and imaging classes defined by `java.awt`, with an API (Application Programming Interface) for two-dimensional graphics, as part of JFC and part of Java 2. The Java 2D API is a set of classes for advanced 2D graphics and imaging, encompassing line art, text, and images in a single comprehensive model. The Java 2D API provides

- A uniform rendering model for display devices and printers
- A wide range of geometric primitives, such as curves, rectangles, and ellipses and a mechanism for rendering virtually any geometric shape
- Mechanisms for performing hit detection on shapes, text, and images

- A compositing model that provides control over how overlapping objects are rendered
- Enhanced color support that facilitates color management, and
- Support for printing complex documents

in a flexible, full-featured framework for developing richer user interfaces, sophisticated drawing programs and image editors. The Java 2D API also enables the creation of advanced graphics libraries, such as CAD-CAM libraries and graphics or imaging special effects libraries, as well as the creation of image and graphic file read/write filters.

When used in conjunction with the Java Media Framework and other Java Media APIs, the Java 2D APIs can be used to create and display animations and other multimedia presentations. The Java Animation and Java Media Framework APIs rely on the Java 2D API for rendering support.

Java 2D is used in DQT for GraphLog drawing and editing.

Component	Description
`UIManager`	This class keeps track of the current look & feel and its defaults.
`JFrame`	A window with a title bar, a border, a content pane, and an optional menu bar.
`JInternalFrame`	Special-purpose container which looks like a frame and has much the same API, but must appear within another window.
`JBorderLayout`	The default layout manager for JFrame, which arranges the components into five areas: North, South, East, West and Center.
`JSplitPane`	A container which is horizontally and vertically split.
`JScrollPane`	A general-purpose container which provides scroll bars around a large or growable component.
`JTabbedPane`	A container that contains multiple components but shows only one at a time. The user can easily switch between components.
`JGridBagLayout`	A layout manager that arranges components into rows and columns, and allows that each component size varies and components are added in any order.
`JMenuBar`	A class for managing a menu bar.
`JMenu`	A class for managing menus.
`JMenuItem`	A class for managing menu items.
`JToolBar`	A holder of a group of components in a row or column.
`JButton`	An area that triggers an event when clicked.
`JTree`	A class that provides a tree view of hierarchical data.
`JTextField`	A class that display information & handles user input.
`JTextArea`	A class for displaying and/or editing text.
`JDialog`	A that .
`JCheckBox`	Implementation of a check box that can be selected or deselected.
`JComponent`	The base class for all Swing components except top-level containers.
`JDesktopPane`	A container for a multiple-document interface or a virtual desktop.
`JLabel`	A display area for a short text string or an image, or both.
`JPanel`	JPanel is a generic lightweight container

Figure 11: Swing Components Used in DQT

2.5 TGL and TGL Translator

TGL stands for Transferable Graphic Language [21]. TGL is an XML format that defines the communication protocol between the DQT GUI layer and TGL Translator layer. The GUI captures user queries in TGL Query Structure (see Figure 12). The TGL Translator transforms a TGL-formatted query, which is received from the GUI layer, to a CORAL query program and submit it to CORAL system for evaluation. The TGL Translator is also responsible to transform the CORAL query result into TGL Result Structure (Figure 13) and then pass the TGL-formatted result to the upper GUI layer.

```
<!ELEMENT graphlog((defineGraphlog+, showGraphlog*)|
                   (defineGraphlog*, showGraphlog+))>
<!ELEMENT defineGraphlog(include*, distinguished-define, content)>
<!ELEMENT showGraphlog(include*, ID, distinguished-show, content)>
<!ELEMENT distinguished-define(node|edge|blob)>
<!ELEMENT distinguished-show((node+,edge*,blob*)|
                     (node*,edge+,blob*)|(node*,edge*,blob+))>
<!ELEMENT content(node*, edge*, blob*)>
<!ELEMENT node(ID, entity)>
<!ELEMENT entity(name, field*)>
<!ELEMENT edge(ID, predicate, fromNodeID, toNodeID)>
<!ELEMENT blob(ID, predicate, outerNodeID, innerNodeID+)>
<!ELEMENT ID(#PCDATA)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT field(#PCDATA)>
<!ELEMENT predicate(#PCDATA)>
<!ELEMENT fromNodeID(#PCDATA)>
<!ELEMENT toNodeID(#PCDATA)>
<!ELEMENT outerNodeID(#PCDATA)>
<!ELEMENT innerNodeID(#PCDATA)>
<!ELEMENT include(#PCDATA)>
```

Figure 12: TGL Query Structure [21]

```
<!ELEMENT showGraphlogReturn(result+)>
<!ATTLIST showGraphlogReturn ID CDATA>
<!ELEMENT result(node*, edge*, blob*)>
<!ELEMENT node(field+)>
<!ATTLIST node ID CDATA>
<!ELEMENT field(#PCDATA)>
<!ATTLIST field pos CDATA>
<!ELEMENT edge(fromNode, toNode)>
<!ATTLIST edge ID CDATA>
<!ELEMENT fromNode(#PCDATA)>
<!ATTLIST fromNode ID CDATA>
<!ELEMENT toNode(#PCDATA)>
<!ATTLIST toNode ID CDATA>
<!ELEMENT blob(outerNode, innerNode+)>
<!ATTLIST blob ID CDATA>
<!ELEMENT outerNode(#PCDATA)>
<!ATTLIST outerNode ID CDATA>
<!ELEMENT innerNode(#PCDATA)>
<!ATTLIST innerNode ID CDATA>
```

Figure 13: TGL Result Structure [21]

The complete details of TGL and TGL Translator can be found in [21].

2.6 Graphviz

Graphviz is an open source graph drawing and visualization software package [33]. Graphviz tools run stand-alone, but can also be extended to create interfaces to external databases and systems. It has been applied to hundreds of projects.

The Graphviz package provides a set of graph visualization tools, along with related user interfaces, stream filters and libraries. It comes with the following viewers:

- dotty - a vintage customizable Unix/X windows viewer that has subsequently been ported to Microsoft Windows.
- tcldot - a TCL/TK scripting language extension for Graphviz
- WebDot. - a tcldot scripted WWW service for graphs in HTML documents. There is also a simplified version written in perl.
- Grappa - a Java package for graphs with full Java graph data structures
- ZGRViewer - an SVG-based zooming graph viewer for large graphs.
- Mac OS X graphviz

One of the unifying themes of Graphviz is the *DOT* language for describing attributed graphs and subgraphs. An abstract grammar for the DOT language is shown as Figure 14. Terminals are shown in bold font and non-terminals in italics. Literal characters are given in single quotes. Parentheses (and) indicate grouping when needed. Square brackets [and] enclose optional items. Vertical bars | separate alternatives.

An *id* is any alphanumeric string not beginning with a digit, but possibly including underscores, a number, any quoted string possibly containing escaped quotes, or an HTML string (<...>). Note that in HTML strings, the content must be legal XML, so that the special XML escape sequences for ", &, <, and > may be necessary in order to embed these characters in attribute values or raw text. Both quoted strings and HTML strings are scanned as a unit, so any embedded comments will be treated as part of the strings.

An *edgeop* is -> in directed graphs and -- in undirected graphs.

The language supports C++-style comments: `/* */` and `//`.

```
graph : [strict] (digraph | graph) [id] '{' stmt-list '}'
stmt-list : [stmt [';'] [stmt-list ] ]
stmt : attr-stmt | node-stmt | edge-stmt | subgraph | id '=' id
attr-stmt : (graph | node | edge) attr-list
attr-list : '[' [a-list ] ']' [attr-list]
a-list : id ['=' id] [','] [attr-list]
node-stmt : node-id [attr-list]
node-id : id [port]
port : ':' id [ ':' compass_pt ] | ':' compass_pt
compass_pt : (n | ne | e | se | s | sw | w | nw)
edge-stmt : (node-id | subgraph) edgeRHS [attr-list]
edgeRHS : edgeop (node-id | subgraph) [edgeRHS]
subgraph : [subgraph [id]] '{' stmt-list '}' | subgraph id
```

Figure 14: Grammar of the DOT Language

Semicolons aid readability but are not required except in the rare case that a named subgraph with no body immediate precedes an anonymous subgraph, because under precedence rules this sequence is parsed as a subgraph with a heading and a body.

Complex attribute values may contain characters, such as commas and white space, which are used in parsing the DOT language. To avoid getting a parsing error, such values need to be enclosed in double quotes.

The DOT language supports many useful attributes for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes. Figure 15 shows a simple example graph in DOT file format and its layout output by *dot*.

```
digraph G {
      size ="4,4";
      main [shape=box]; /* this is a comment */
      main -> parse [weight=8];
      parse -> execute;
      main -> init [style=dotted];
      main -> cleanup;
      execute -> { make_string; printf}
      init -> make_string;
      edge [color=red]; // so is this
      main -> printf [style=bold,label="100 times"];
      make_string [label="make a\nstring"];
      node [shape=box,style=filled,color=".7 .3 1.0"];
      execute -> compare;
}
```

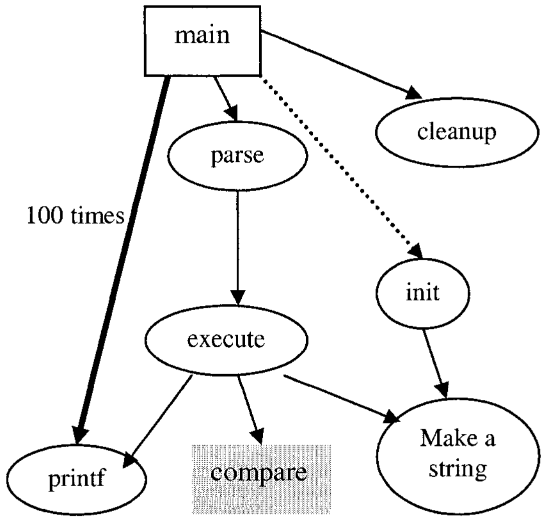


Figure 15: A Simple Graph in DOT Format and Its Layout [34]

Graphviz has several graph layout programs in the families of hierarchical layouts of trees and DAGS (directed acyclic graphs), and virtual physical (``spring model") layouts of undirected graphs.

- dot - makes ``hierarchical" or layered drawings of directed graphs. The layout algorithm [35] aims edges in the same direction (top to bottom, or left to right) and then attempts to avoid edge crossings and reduce edge length.

- neato - make ``spring model" layouts. neato uses the Kamada-Kawai algorithm [36], which is equivalent to statistical multi-dimensional scaling.
- twopi - radial layout.
- circo - circular layout. Suitable for certain diagrams of multiple cyclic structures.

The layout programs take descriptions of graphs in DOT language, and make diagrams in several useful formats such as images and SVG for web pages, Postscript for inclusion in PDF or other documents; or display in an interactive graph browser. It also supports GXL, an XML dialect.

Our project will utilize the Graphviz dot language and layout programs to layout the hygraph query results for visual presentations.

2.7 The University Database for Case Study

A university database is used for illustrations and examples in the thesis. The database captures the entities and teaching/studying relationships in a condensed university community. The database schema is shown in Figure 16. The relational database schema is mapped to CORAL relations and the relational database instance is mapped to CORAL ground facts. The corresponding CORAL relations are shown as Figure 17 and the instance is shown in Figure 18.

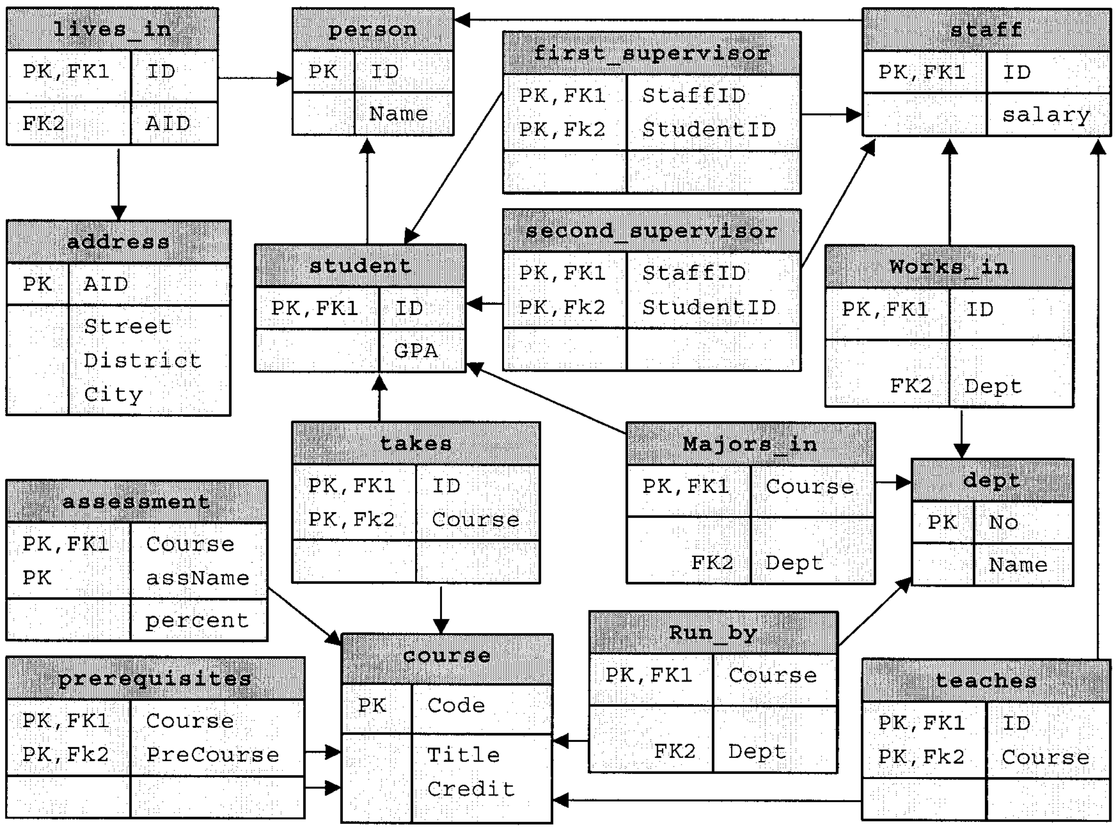


Figure 16: The University Database Schema

```
person(ID, Name).
staff(ID,Salary).
student(ID, GPA).
dept(No, Name).
course(Code, Title, Credit).
address(AID, Street, District, City).
works_in(ID, Dept).
teaches(ID, Course).
majors_in(ID, Dept).
takes(ID,Course).
run_by(Course,Dept).
prerequisites(Course,PreCourse).
assessment(Course,assName,Percent).
lives in(ID, AID).
first supervisor(StaffID, StudentID).
second supervisor(StaffID, StudentID).
resides(Dept, AID).
```

Figure 17: The System Scheme File shared with lower layer system

```
course(comp218, "Fundamentals of C++ Programming", 3).
course(comp248, "Introduction to Programming",3).
course(coen60, "Software Regular ",4).
course(comp651, "DB4", 4).
course(comp646, "Computer Networks and Protocols", 4).
course(eSL207, "English as Second Language 207", 1).

address(addr001, "Maisonuvue Street", "center ville", "Montreal").
address(addr002, "Hillhead Street", "Hillhead", "Montreal").
address(addr003, "University Avenue", "Kelvinside", "Montreal").
address(addr004, "Lincoln Street", "Dowanhill", "Glasgow").
```

```
person(cs0001, "Eric Atwood").
person(cs0002, "Larry Nabil").
person(ce0001, "Tony Herry").
person(css001, "Steve Losa").
person(ce1005, "Lisa Joey").
person(4881177, "Steve Johnson").
person(3345167, "Marry Sabin").
person(3511786, "Hossa Gosta").
person(3788947, "Anrew Li").
person(4125785, "Jenny Liu").
person(cs0003, "Linda Smith").
person(cs0004, "Bob Campbell").

staff(cs0001, 2500).
staff(cs0002, 1800).
staff(ce0001, 4801).
staff(css001, 3400).
staff(ce1005, 1685).
staff(4125785, 3400).
staff(cs0003, 2300).

student(4881177).
student(3345167).
student(3511786).
student(3788947).
student(4125785).
student(css001).

dept(cs, "Computing Science").
dept(gm, "John molson business").
dept(artG, "Art Gallery").
dept(edu, "Education").
dept(eg, "Engineering").

works_in(cs0001, cs).
works_in(cs0002, cs).
works_in(ce0001, eg).
works_in(css001, cs).
works_in(ce1005, eg).
works_in(4125785, gm).
works_in(cs0003, cs).

teaches(cs0001, comp676).
teaches(cs0002, comp218).
teaches(ce0001, coen60).
teaches(css001, comp646).
teaches(ce1005, coen61).
teaches(cs0003, comp248).
teaches(ce0001, eSL207).

run_by(comp646, cs).
run_by(comp218, cs).
run_by(comp248, cs).
run_by(eSL207, edu).
```

```
prerequisites(comp248, comp218).
prerequisites(comp646, comp248).
prerequisites(comp651, comp646).

majors_in(4881177, cs).
majors_in(css001, cs).
majors_in(3345167, gm).
majors_in(3511786, eg).
majors_in(3788947, edu).
majors_in(4125785, cs).

takes(4881177, comp646).
takes(4125785, comp218).
takes(4881177, eSL207).
takes(4125785, eSL207).
takes(4881177, comp248).
takes(4125785, comp248).
takes(3345167, comp248).
takes(3788947, comp248).

assessment(comp646, "mid-term1", 0.25).
assessment(comp646, "mid-term2", 0.25).
assessment(comp646, "assign", 0.25).
assessment(comp646, "final", 0.25).
assessment(comp651, "mid-term", 0.25).
assessment(comp651, "project", 0.25).
assessment(comp651, "final", 0.5).
assessment(coen60, "ass1", 0.33).
assessment(coen60, "ass2", 0.34).
assessment(coen60, "final", 0.33).
assessment(comp218, "mid-term", 0.5).
assessment(comp218, "final", 0.5).
assessment(comp248, "final", 1).
assessment(eSL207,  "ass1", 0.1).
assessment(eSL207, "ass2", 0.1).
assessment(eSL207, "ass3", 0.1).
assessment(eSL207, "ass4", 0.1).
assessment(eSL207, "final", 0.6).

lives_in(cs0001, addr002).
lives_in(cs0002, addr003).
lives_in(4881177, addr001).
lives_in(4125785, addr004).
lives_in(3511786, addr002).

first_supervisor(cs0001, 4881177).
first_supervisor(cs0002, 3511786).
first_supervisor(cs0003, 3788947).
first_supervisor(cs0003, 4125785).

second_supervisor(cs0004, 4125785).
second_supervisor(css001, 4881177).
second_supervisor(cs0004, 3788947).
```

Figure 18: The University Database Instance

3. Supporting Blobs in Visual Queries

In this chapter, we compare visual queries with and without blobs, describe how to define blobs, and how to define new relations and queries with blobs, as well as the implementation of blob support in query constructions.

3.1 Visual Queries with and without Blobs

Blob is a new notion incorporated into graph by hygraph. A blob in a hygraph represents a relation between a node, called the container node, and a set of other nodes, called the contained nodes. A blob replaces all the edges that would otherwise connect the container node of the blob with each of the nodes contained in the blob; it is diagrammatically represented as a labeled rectangle block associated with the container node that encloses the contained nodes.

In effect of visualization, a blob modularizes the set of contained nodes and their associations with the container node. Blobs can help reveal interesting characteristics of data while avoiding distractions and irrelevancy, and provide effectiveness and clarity in construction of graphical queries and visualization of query results; therefore help the users discover interesting patterns more easily and intuitively. Let's see one example query: "*Return all the students who take courses taught by Linda Smith.*"

Without using blobs, the query can be expressed in Graphlog shown in Figure 19.

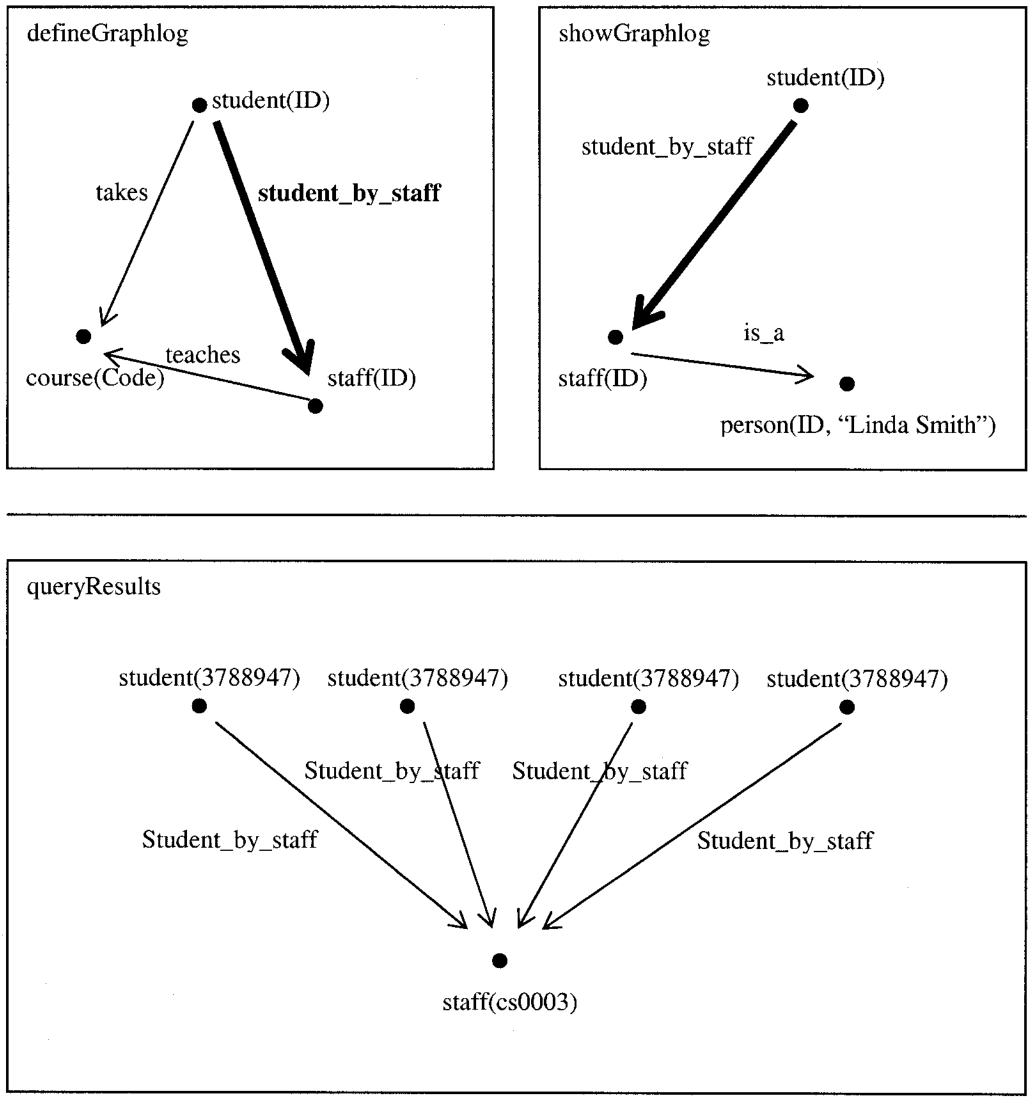


Figure 19: Example Query and Results without Using Blobs

It can be seen that for each student taught by Linda Smith, an edge appears with the corresponding label. Imagine that how the look will be if Linda is teaching 100 students! Using a blob can dramatically improve the picture.

Figure 20 shows the Graphlog that implement the same query and its corresponding presentation with a blob. Much more clear and intuitive now!

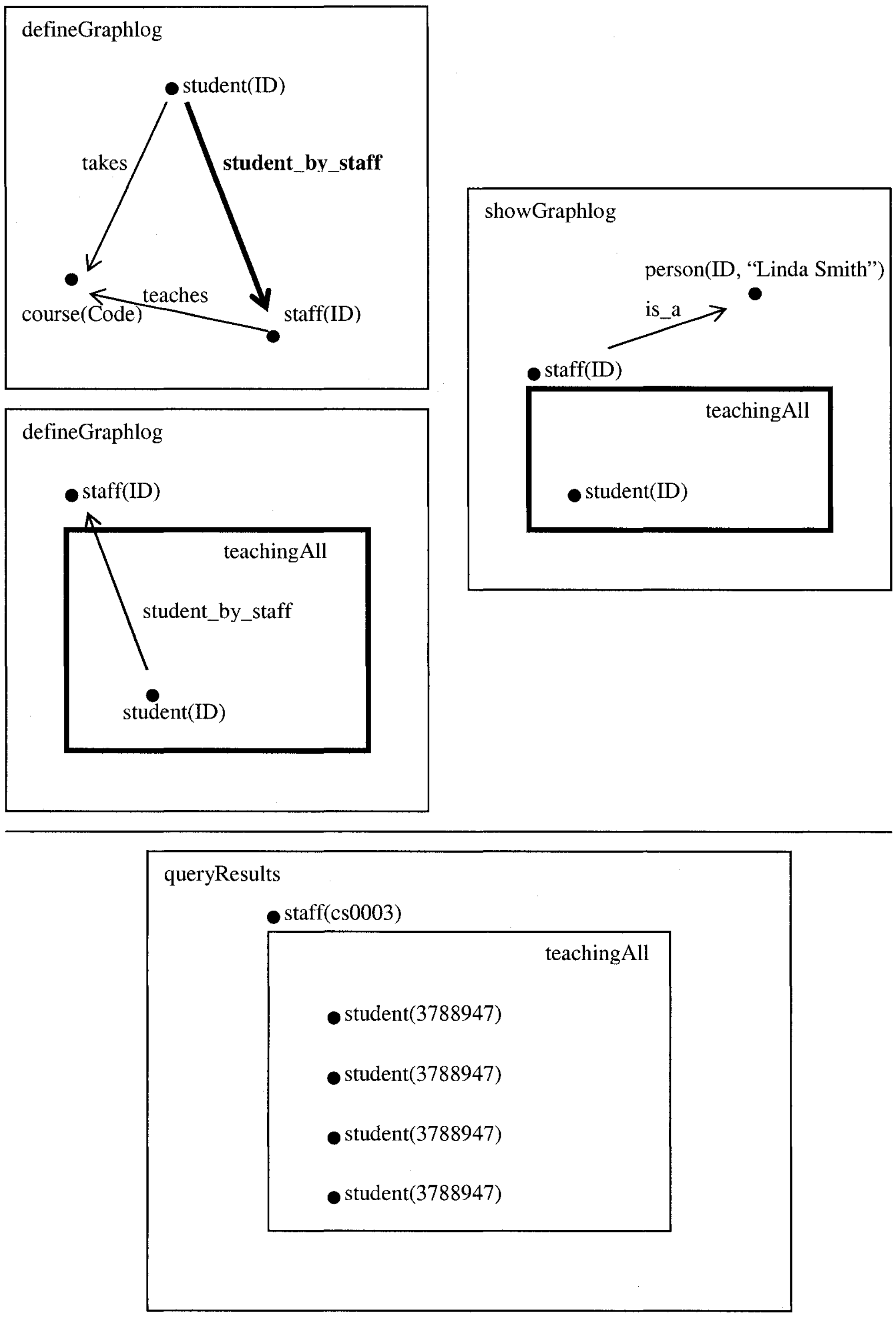


Figure 20: Example Query and Results with a Blob

3.2 Defining Blobs with DQT

A blob associates a container (or outer) node to a collection of contained (or inner) nodes. To facilitate visually defining blob, we have considered two alternative designs, including drag-and-drop and drawing-and-select.

Drag-and-Drop: The basic idea is to pre-define a blob template in the current graph editor and ask users to drag objects from the schema tree and drop it into the blob template. The major advantage of this approach is the potential of automatic population of some of the object properties, like object names – note that users still need to manually edit the object properties, like specifying what fields are involved in the definition or query. The disadvantages are the requirements for the capability of distinguishing entities from relationships in the schema tree to make the automatic property population work and the complexity of its implementation.

Point-and-Select: The approach allows users to point and click on a node to specify the container node, and then drag a rectangle to select the contained nodes. The advantages of this approach are the consistency of the methods for defining nodes, edges and blobs by point-and-click and drawing, and the artful joy of drawing. The major disadvantage is that users have to manually edit the object names, but it does not affect the usability much – users have to do that for nodes and edges any way.

The selected design is the *Point-and-Select* approach for its consistency with current drawing without blob support and its independency on the schema tree, with considering the complexity of implementation. The selected approach has been implemented in DQT.

To define a blob, a user need to select the *Create a Node* mode and draw all the involved nodes first (A screenshot is shown as Appendix A). Optionally, the user can then select *Create an Edge* mode to draw edges (Appendix B) - users can draw the edges even after the blob gets drawn; then the user can select the *Create a Blob* mode to draw the blob by the way of point-and-click on the target container node and dragging a rectangle to cover all the node to be contained, see Appendix C. At any time after an object has been drawn, the user can switch to the *Select an Object* mode to edit the properties of objects; however, the user can also choose to edit the properties after all the objects have been drawn, which eliminates the needs for switching between the working modes. Importantly, the blob to be defined must be specified as *Distinguished* by checking on the corresponding check box in the property dialog. Visually, the blob rectangle will be drawn in bold to represent it as distinguished. Appendix D shows a screenshot for the newly defined blob "teachingAll"

3.3 Defining New Relations and Queries with Blobs

With referencing existing definitions of blobs, a user can define new relations and queries more easily. Referencing a blob, such as "teachingAll" defined above, is just like referencing another kind of user-defined relations, such as "student_by_staff". Users just need to draw the blob with the user-defined blob name in the new relations or queries, and then press the Define or Execute icon for submission.

Appendix E shows the Graphlog query that returns all students taught by Linda Smith. To execute the query, users just need to push the execute icon. The DQT will translate the Graphlog query into TGL (shown in Figure 21) for evaluation by the engines at lower layers, and visualize the results, including using the defined blobs, when it receives the query results.

```
<graphlog>
 <showGraphlog>
   <id>tempQueryResult</id>
   <distinguished-show>
     <blob>
       <id>BID0002</id>
       <predicate>Blob_0</predicate>
       <outerNodeID>NID0000</outerNodeID>
       <innerNodeID>NID0001</innerNodeID>
     </blob>
   </distinguished-show>
   <content>
     <node>
       <id>NID0000</id>
       <entity>
         <name>staff</name>
         <field>ID</field>
       </entity>
     </node>
     <node>
       <id>NID0001</id>
       <entity>
         <name>student</name>
         <field>ID</field>
       </entity>
     </node>
     <node>
       <id>NID0002</id>
       <entity>
         <name>person</name>
         <field>ID</field>
         <field>"Linda Smith"</field>
       </entity>
     </node>
     <edge>
       <id>EID0_2</id>
         <predicate>is_a</predicate>
         <FromNodeID>NID0000</FromNodeID>
         <ToNodeID>NID0002</ToNodeID>
     </edge>
   </content>
 </showGraphlog>
</graphlog>
```

(a) tempQuery.xml

```
<graphlog>
 <defineGraphlog>
   <distinguished-define>
     <blob>
       <id>BID0001</id>
         <predicate>teachingAll</predicate>
         <outerNodeID>NID0000</outerNodeID>
         <innerNodeID>NID0001</innerNodeID>
     </blob>
   </distinguished-define>
   <content>
     <node>
       <id>NID0000</id>
       <entity>
         <name>staff</name>
         <field>ID</field>
       </entity>
     </node>
     <node>
       <id>NID0001</id>
       <entity>
         <name>student</name>
           <field>ID</field>
       </entity>
     </node>
     <edge>
       <id>EID1_0</id>
       <predicate>student_by_staff</predicate>
       <FromNodeID>NID0001</FromNodeID>
       <ToNodeID>NID0000</ToNodeID>
     </edge>
   </content>
 </defineGraphlog>
</graphlog>
```

(b) teachingAll.xml

Figure 21: Blobs in TGL

3.4 Implementation

A blob has one container node and one or more contained nodes, and generalize the edges between the container node and the contained nodes, and may involve other edges among the contained nodes and other nodes. To warrant the consistency of a graph and efficiency of processing, blob support is implemented based on the *Flyweight design pattern*. The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost [33]. We treat a node as a flyweight; all the nodes, captured as a NodeList, form a pool of flyweight objects, which are referenced by blobs.

The Blob class captures what is the container node, what are the contained nodes, the enclosure rectangle, and other properties, such as if the blob is distinguished (see Appendix F). The Blob class does not copy all the details of the nodes involved; instead, it simply makes reference to the node Ids. And the Blob class does not memorize the edges inside; instead, it uses the normal edge handling and does special checking during the blob-related event handling. This can greatly reduce memory usage, simplify the implementations, and, most importantly, avoid inconsistency in the event of changes.

Mouse events are handled for blob creations if the current mode is *Create a Blob*, and for blob property editions if the current mode is *Select an Object* and a mouse double-click has been done right on the blob rectangle. Both the creating a new blob and the editing blob property events are initiated by a proper mouse-down event within the GraphCanvas of the GraphEditor. The relevant pieces of code are listed as Appendix G.

A blob rectangle gesture is drawn during a mouse-drag event if a container node has been set (see Appendix H). A blob rectangle keeps changing during mouse dragging events until the mouse button is released. A mouseUp event occurs when the mouse button is released. Then, the mouseUp handler (See Appendix I for details) will capture all the information and insert a new blob into the blob holder, which is organized as a *Hashtable* and is a part of the current Graph object.

Several attributes and methods have been incorporated into the Graph class and GraphCanvas class so that they can adopt and take care of blobs.

The TGLGraph class has been augmented to translate Graphlog with blobs into TGL a query structure and to compute the metadata for visualizing the query results. See Appendix J for details. Methods also have been added into GraphEditor class so that it can handle mode selection for *Create a Blob* when selected, generate TGL and populate the definition of a blob into the schema tree when the *Define* button (icon) is pressed, generate TGL queries and translate query results into the Graphviz dot language for presentations (the details will be described in Chapter 4).

4. Visualizing Blobs in Query Results

A blob modularizes the set of contained nodes and their associations with the container node and provides a flexible mechanism for clustering information. Using blobs in visualization of query results can help reveal interesting characteristics of data while avoiding distractions and irrelevancy; therefore help the users discover interesting patterns more easily and intuitively.

In this chapter, we describe how to visualize blobs using Graphviz.

4.1 Cluster and Its Visualization in Graphviz

Graphviz provides graph layout generators (such as dot and neato), which can read and layout the graphs expressed in the DOT language. DOT describes three kinds of objects: graphs, nodes, and edges, with various attributes. The main (outermost) graph can be a directed (digraph) or undirected graph. A graph can contain nested subgraphs; a subgraph defines a subset of graph objects, such as nodes, edges, and nested subgraphs.

A cluster is a subgraph placed in its own distinct rectangle of the layout. A subgraph is recognized as a cluster when its name has the prefix `cluster`. A cluster is laid out separately, and then integrated as a unit into its parent graph, with a bounding rectangle drawn about it. If the cluster has a label parameter, this label is displayed within the rectangle. There can be clusters within clusters. Clusters at the same level are drawn in non-overlapping rectangles. Figure 22 shows an example of cluster layouts and the corresponding graph files in the DOT language.

```
digraph G {
   subgraph cluster0 {
         node [style=filled,color=white];
         style=filled;
         color=lightgrey;
         a0 -> a1 -> a2 -> a3;
         label = "process #1";
   }

   subgraph cluster2 {
         node [style=filled];
         b0 -> b1 -> b2 -> b3;
         label = "process #2";
         color=blue;
   }

   start -> a0;
   start -> b0;
   a1 -> b3;
      b2 -> a3;
   a3 -> a0;
   a3 -> end;
   b3 -> end;

   start [shape=Mdiamond];
   end [shape=Msquare];
}
```

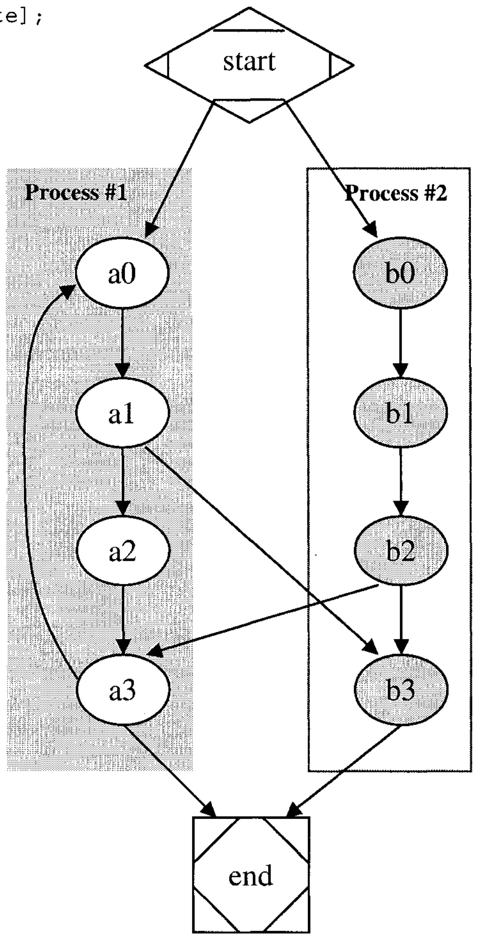


Figure 22: Process Diagram with Clusters

Labels, *font* characteristics and the *labelloc* attribute can be set for a cluster as they would be for the top-level graph, though cluster labels appear above the graph by default. For clusters, the label is left-justified by default; if labeljust="r", the label is right-justified. The *color* attribute specifies the color of the enclosing rectangle. In addition, clusters may have *style*="filled", in which case the rectangle is filled with the color specified by *fillcolor* before the cluster is drawn.

Graphviz's cluster construct and its layout treatments provide a way to visualize blobs in hygraph query results.

4.2 Visualizing Blobs using Graphviz

Blob instances may appear in the query result if any blobs have been referenced or defined in the corresponding query.

To visualize blobs, we design the following representation, called `Visual Blob Pattern`, that visualizes the blob's container node as a normal Graphviz node and all its contained nodes along with the edges between them as a Graphviz cluster labeled with the blob label, and connect the container node to the cluster with a "dot-inv" arrow (Figure 23).

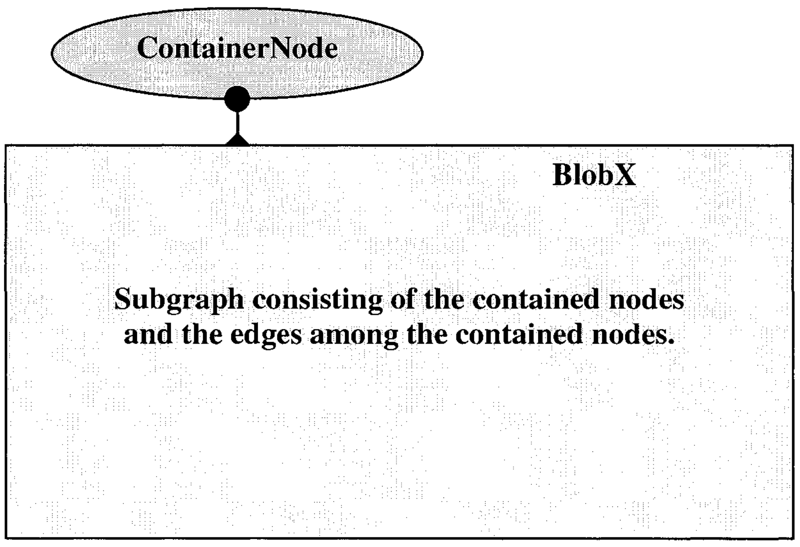

ContainerNode
BlobX
Subgraph consisting of the contained nodes and the edges among the contained nodes.

Figure 23: The Visual Blob Pattern

The visual blob pattern is formulated in the DOT language as shown by Figure 24. We add a special *invisible* node, called *dummy inner node*, into the blob cluster to enable a connection between the container node and the cluster in the way of independent of any real inner nodes, with ignoring any other edges between the container node and each of the actual contained nodes. By setting `compound=true`, we can make the connection clipped by the blob boundary rectangle to achieve the effect shown by Figure 23. By setting tailport=s and headport=n, we make the connection between the container node to the blob cluster always from top to bottom.

```
digraph G {
  Compound=true; // to allow edges from/to a cluster
  // other parts of the result graph
        :
  // Begin of Blob, named BlobX
  subgraph cluster_BlobX {
        label="BlobX";
        style=filled;
        fillcolor=lightgrey; // the fill color will be set using metadata
        // subgraph of the blob's contained nodes and edges
              :
  }
  theContainerNode -> theDummyInnerNode
        [ style=bold, arrowsize=2, arrowhead=inv, arrowtail=dot,
          len=0.1, tailport=s, headport=n, lhead=cluster0 ];
  // End of BlobX


  // Other part of the query result graph
        :
}
```

Figure 24: Represent the Visual Blob Pattern in DOT

Next, we show a query of the University database: "Return all the students who take course run by the department of Computer Science as a blob with the department number as the container node." The result blob is shown in Figure 25.

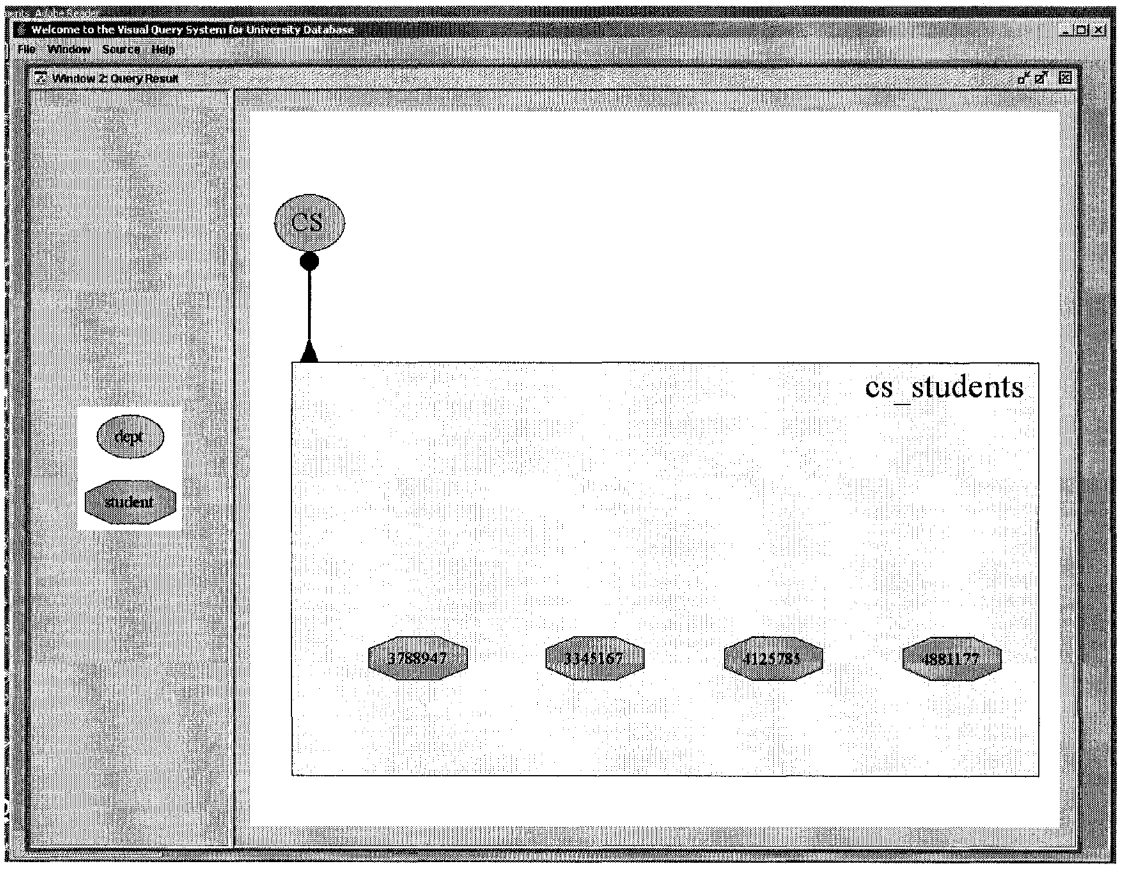


Figure 25: Visualization of A Blob

4.3 Implementation

DQT receives the results of queries in TGL format. TGL is based on XML. DQT parses query results using SAX, the *Simple API for* XML, to extract the necessary information for presentations of the results. After the XML result file gets parsed, DQT contructs any blobs from the result set into the visual blob pattern using DOT language. Appendix K lists the section of code which constructs blob instances in DOT language following the visual blob pattern..

5. Translating and Presenting Query Results

Originally, DQT supports graph queries consisting of only nodes and edges, and parses XML-format query results consisting of only nodes and edges. It does not provide automated translations of query results from TGL into Graphviz dot graph. Therefore, to layout and visualize a query result, a person has to manually translate it into a DOT graph before calling Graphviz for presentation of the result. DQT was far from usable by application subject experts who lack expertise in Graphviz or DOT language.

As part of this thesis, we have implemented the automated translations of query results from TGL into Graphviz dot graph in an integrated way, in addition to providing blob support for visual queries and presentations.

This chapter is to describe the design and implementation of the automated translations of query results.

5.1 Graphviz Java API

Graphviz Java API is not a part of Graphviz, but third-party free software under the GNU Lesser General Public License. With the API, we can construct DOT graphs and call Graphviz in Java by a more streamlined and better readable way.

In the method, `visualizeResult()`, for automatic translation of query results, we have used the Graphviz Java API for better readability of the source code.

5.2 Presenting Query Results with Clarity

A query result may contain tens, hundreds, and even thousands of objects, such as nodes, edges and blobs. Different objects should be presented in different visual effects for distinguishing from each other, while the objects belonging to the same class, such as the same entity, the same relationship or the same blob, should be visualized with some kind of common effects for grouping the same kinds. The combinations of shapes and colors are used for classifying node-like objects; the combinations of line styles and colors are used for distinguishing and grouping edge instances; the blob names (or labels) along with colors are used for visually identifying blob instances. The properties are assigned at the schema object level based on the executed query and captured as a property map, which consists of three vectors, `dotNodeVector`, `dotEdgeVector`, and `dotBlobVector`. At this stage, the assignment is simply based on the order of graph objects of each kind, and the predefined lists of node shapes, node colors, edge styles, edge colors, and blob colors (Figure 26). When the number of object classes exceeds the number of properties of some kind, the assignment starts over from the beginning of the list again. Since the size of the lists of associated properties are different, start-over should not occur with two lists at the same time, given that properties are assigned at the schema object class level and the number of object classes in a query is limited. The property map will be consulted when assigning properties to any instance objects during the procedure of translating the query result and constructing the result graph in DOT language.

```
// Xuede: generate DOT graph property maps for visualizations
//
private void generatePropertyMaps()
{
  String nodeShape[] = { "ellipse", "octagon", "hexagon", "house", "diamond",
                         "parallelogram", "trapezium", "doubleoctagon" };
  String nodeColor[] = { "plum", "goldenrod", "coral", "lightpink", "orange",
                         "mediumpurple", "limegreen", "lightcyan", "lightblue",
                         "lightgray", "orchid", "tan" };

  String edgeStyle[] = { "solid", "dashed", "dotted", "bold" };
  String edgeColor[] = { "black", "red", "blue", "green",
                         "magenta", "brown", "cyans", "orange" };

  String blobColor[] = { "cornsilk", "wheat", "lightyellow", "olivedrab",
                         "aquamarine", "springgreen", "lightskyblue", "beige"};

  int i;

  dotNodeVector = tglGraph_.getDOTNodeVector();
  dotEdgeVector = tglGraph_.getDOTEdgeVector();
  dotBlobVector = tglGraph_.getDOTBlobVector();

  System.out.println("NodeV.size() = " + dotNodeVector.size() +
                   "; EdgeV.size() = " + dotEdgeVector.size() +
                   "; BlobV.size() = " + dotBlobVector.size());

  DOTNode dotNode;
  for ( i=0; i<dotNodeVector.size(); i++ ) {
    dotNode = (DOTNode) dotNodeVector.elementAt( i );

    dotNode.setShape( nodeShape[i%8] );
    dotNode.setFillColor( nodeColor[i%12] );

    System.out.println( "dotNode[" + i + "]: id=" + dotNode.getId() +
                        " Name=" + dotNode.getName() );
  }

  DOTEdge dotEdge;
  for ( i=0; i<dotEdgeVector.size(); i++ ) {
    dotEdge = (DOTEdge) dotEdgeVector.elementAt( i );

    dotEdge.setStyle( edgeStyle[i%4] );
    dotEdge.setColor( edgeColor[i%8] );

    System.out.println( "dotEdge[" + i + "]: id=" + dotEdge.getId() +
                        " Name=" + dotEdge.getName() );
  }

  DOTBlob dotBlob;
  for ( i=0; i<dotBlobVector.size(); i++ ) {
    dotBlob = (DOTBlob) dotBlobVector.elementAt( i );

    dotBlob.setFillColor( blobColor[i%8] );

    System.out.println( "dotBlob[" + i + "]: id=" + dotBlob.getId() +
                        " Name=" + dotBlob.getName() );
  }

}
```

Figure 26: Generating Property Map

With many objects in a graph, node and edge labels may make the graph unreadable. To achieve clarity, we illustrate the node class names and edge labels along with their source and destination classes in a legend graph, instead of displaying them for each instance of the node classes and edges in the main result graph. Only those node classes and edges that have instances in the result graph are shown in the legend graph The legend graph and the result graph are presented side by side in a Java Swing split pane (JSplitpane). The legend graph, in the left pane, shows only the entities and relationships at the schema level, when the query result graph is shown at the instance level in the right pane. An instance node in the query result graph is mapped to their schema entity, visualized as a node with entity name inside in the legend graph, by the same shape and color; an instance edge in the result graph is associated to its labeled relationship in the legend graph by the same line style and color. An example is shown in Figure 27. The example is based on an artificially made result file in TGL format for testing and showing complex translation and layout.

Users can move the divider to enlarge one pane while reducing the other one; user can totally hide the legend graph by reducing the legend pane to the minimum.

By this design, all the information for the query results are preserved and presented, and the result graph are visualized with greater clarity.

The information for building a legend graph is captured during the translation of the query results in TGL to the result graph in DOT. More details will be given in next section.

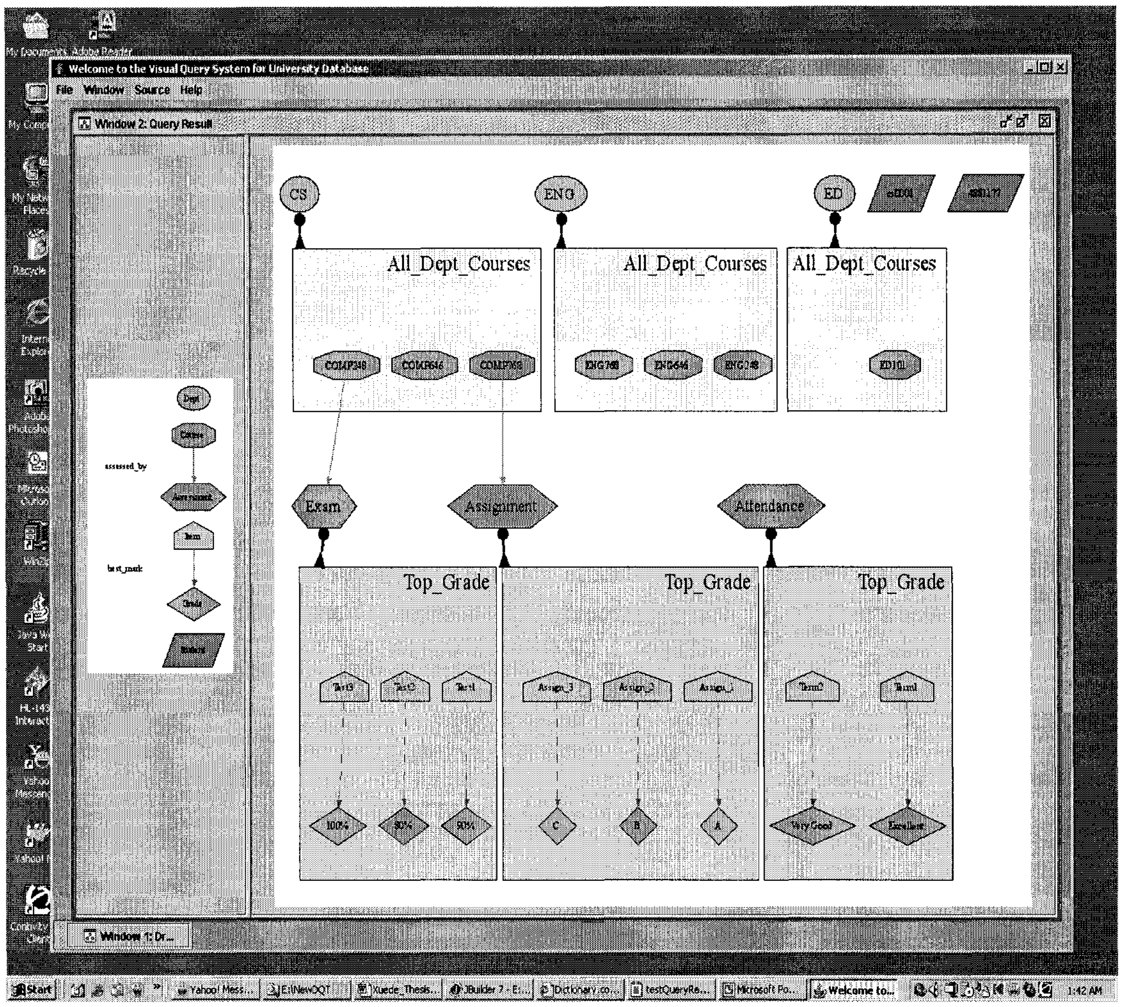


Figure 27: Visualizing Query Result with Legend

5.3 Automating Translation of Query Results into DOT Graphs

DQT receives a query result set in TGL format – XML format. We must translate it into a graph defined in the DOT language so that Graphviz can take it for layout processing. In addition, various properties, including shapes, styles, and colors etc, should be properly assigned to every graph objects (nodes, edges, and blobs) for better effects of visualization.

The translation task is processed in two phases. The first phase is, simply by means of the SAX XML Parser, to parse the TGL (XML) format result file into the internal result structure consisting of three vectors `nodeVec`, `edgeVec` and `blobVec`, which capture all the nodes, edges, and blobs respectively. These are attributes of the internal TGLTranslator class. An object of TGLTranslator is created and its parseXML() method is called by the visualizeResult() method in GraphEditor. See Note 1 in Appendix L for the implementation.

The second phase of translation is to construct the query result graph and the legend graph in DOT language, and call Graphviz for layout. Since blobs are subgraphs from Graphviz' layout point of view, blobs must be processed first to ensure that the nodes contained by any blobs are placed into the right subgraphs. Then edges are processed. Orphan nodes are processes at last.

Blob Processing: Using the DOT language and the Graphviz Java API, an instance of the visual blob pattern is defined for each blob in the result set. For each blob instance, a special subgraph, *cluster*, is opened with the properties from the property map; then all the contained (or inner) nodes are placed within the cluster; a dummy inner node is added before the cluster is closed after all the contained nodes have been processed. Then the container node is added to the graph, but outside of the blob cluster, with the properties from the property map, and the connection is added between the container node and the cluster (the dummy inner node, actually) with the properties defined by the visual blob pattern (See Note 2 in Appendix L). All the blob instances of the same blob class, through their container nodes, are defined with the same rank, which give guidance to Graphviz for layout (See Note 3 in Appendix L).

Edge Processing: Each edge instance, along with its nodes, is added to the dot graph with the properties from the property map. (See Note 4 in Appendix L.)

Node Processing: All the node instances are then added to the dot graph with the properties from the property map. (See Note 5 in Appendix L.)

After all the blobs, edges and nodes have been processed and added into the result graph, Graphviz is called (see Note 6 in Appendix L) to make a layout for the presentation of the result.

Legend Processing: During the procedure of processing blobs, edges and nodes, all the information needed to build the legend graph has been collected. A separate DOT graph for legend is built with the legend properties. A separate layout of the legend graph is made by calling Graphviz again. The implementation is shown at the last section of Appendix L (Note 7).

6 Conclusions

In this thesis, we have provided blob support in DQT. With this capability, DQT users can define new blobs and include blobs in graph queries by drawing. With usage of Blobs in graph queries, query results can include blobs. Using Blobs in query results greatly increases the modularization and abstraction of the result graphs, which helps reveal interesting characteristics of data while avoiding distractions and irrelevancy, and provides effectiveness and clarity in construction of graphical queries and visualization of query results; therefore helps the users discover interesting patterns more easily and intuitively. We have designed the visual blob pattern for visualizing blobs in query results.

We have also implemented the automatic translation of TGL query results into DOT graphs and automatic generation of layouts of the result graphs. With this enhancement, DQT automatically constructs the result graphs in the DOT format and makes a layout by calling Graphviz. This makes DQT practically usable by any users who are familiar with the application domains.

Careful considerations and design measures have been taken to obtain best presentations of query results. As described in Section 5.2, we use combinations of colors and shapes to distinguish different kinds of nodes; combinations of colors and line styles to identify different relationships, and different filling colors to highlight the difference of blob categories. And instead of duplicating schema information and labels in the result graphs, we separate the instance level graphs from their schema level, and present the named entity class with names (each of which may have many nodes as its instances) and relationships with labels (each of which may have many occurrences of edges in the result graph) in the legend graph with mapping properties beside the result graph. With this design, entity class names and edge labels are not necessary to appear in the result graph presentation, which avoids blacking out the result graphs for large and complex result sets.

There are still some future work for further enhancing DQT's functionality and usability. Currently, the blob support capability provided by the thesis work is limited to only one blob that can be defined from a container node (there is no such a limitation from the visualization side). This limitation can be removed by processing blob rectangles in other directions – currently DQT process only draws a blob rectangle from top-left to bottom-right. Another limitation is that nested blobs are not processed since the underlying TGL specification does not support nested blobs, although users can draw nested blobs. To support nested blobs, TGLTranslator must be enhanced with the capability to parse queries with nested blobs and convert them into Coral programs. Also it may be desired to support a more complicated case in which one node participates in multiple blobs.

In addition, drawing graph queries by drag-and-drop (drag an object from the schema tree and drop it into the graph editor) is of interests since it can reduce users' work to draw or edit a graph and offer the potential of automating the naming of graph objects. To implement this approach, the schema tree must be organized in the way that clearly distinguishes node entities from edge relationships, and includes all the primary keys and foreign keys to enable automatic labeling with lower requirements for users to make corrections.

The DQT tool has not been tested for scalability with large result sets. And user trials are needed to receive feedback for improvements.

During the project, several knowledge areas and methodologies have been intensively studied to achieve an effective, elegant design. By doing the project, my background and skills have been greatly enhanced in the areas of database systems and query languages, framework and interface development, GUI design principles, object-oriented design and Java programming. By participating in the active discussions and presentations led by Dr. Butler, my knowledge has been broadened. What I have learned will definitely help in the future.

References

[1] M. Stonebraker, R. Agrawal, U. Dayal, E. Neuhold, and A. Reuter. DBMS re-search at a crossroads: The Vienna update. *Proc. of the 19th International Conference on Very Large Data Bases*, pages 688-692, Dublin, Ireland, August 1993.

[2] Aiken, A.; Chen, J.; Lin, M.; Spalding, M.; Stonebraker, M.; Woodruff, A. The Tioga-2 database visualization environment. *Proceedings: Data Issues for Data Visualization, IEEE Visualization '95 Workshop*, Atlanta, GA, USA, 28 Oct. 1995). Edited by: Wierse, A.; Grinstein, G.G.; Lang, U. Berlin, Germany: Springer-Verlag, 1996. p. 181-207.

[3] H. McCormick, T. A. DeFanti, and M.D. Brown, "Visualization in Scientific Computing", *SIGGRAPH Computer Graphics,* vol. 21, no. 6, pp.30-42, Nov. 1987.

[4] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. Van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization", *IEEE Computer Graphics and Applications*, vol. 9, no. 4, pp.30-42, July 1989.

[5] G. Butler, Database technology for pathways. *In Workshop on Computation of Biochemical Pathways and Genetic Networks,* Logos Verlag, Berlin, 1999, ISBN 3-89722-093-8, pp. 89-95.

[6] G. Butler, E. Bornberg-Bauer, G. Grahne, F. Kurfess, C. Lam, J. Paquet, I. Rojas, R. Shinghal, L. Tao, A. Tsang. The BioIT projects: Internet, database and software technology applied to bioinformatics, *Proceedings of SSGRR'2000*, Suola Superiore G. Reiss Romoli SpA, Coppoto, Italy

[7] S. Card, G. Robertson, and J. Mackinlay, "The Information Visualizer, An Information Workspace", In *Proceedings of the Conference on Computer Human Interaction,* pp.181-188, 1991

[8] M.P. Consens and A.O. Mendelzon, "Graphlog: A Visual Formalism for Real Life Recursion", *Proc. ACM Symposium Principles of Database Systems*, pp.511-516, 1993.

[9] M.P. Consens, F.Ch. Eigler, M.Z. Hasen, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista, "Architecture and Applications of the Hy+ Visualization System", *IBM Systems Journal,* vol. 33, no. 3, pp.458-476, 1994.

[10] M. Gyssens, J. Paredaens, and D.V. Van Gucht, "A Graph-Oriented Object Model for Database End-User Interfaces", *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp.24-33, 1990.

[11] A. Papantonakis and P.J.H. King, "Syntax and Semantics of Gql, A Graphical Query Language", *J. Visual Language and Computing*, vol.6, no.1, pp.3-36, 1995.

[12] J. Paredaens, P. Peelman, and L. Tanca, "G-Log: A Declarative Graphical Query Language", *Proc. Second Int'l Conf. Deductive and Object-Oriented Databases (DOOD),* pp.108-128, 1991.

[13] A. Poulovassilis and C. Small, "A Functional Programming Approach to Deductive Databases", *Proc. 17the Very Large Data Base Conf.,* pp.491-500, 1991.

[14] A. Poulovassilis and S.G. Hild, "Hyperlog: A Graph-Based System for Database Browsing, Querying, and Update", *IEEE Transaction on Knowledge and Data Engineering,* vol. 13, no. 2, pp.316-333, March/April 2001.

[15] M.P. Consens, "Creating and Filtering Structural Data Visualizations using Higraph Patterns", *Ph.D Thesis, University of Toronto,* February 1994.

[16] Greg Butler, Ling Chen, Xuede Chen, A. Gaffar, Jinmiao Li, Lugang Xu. The Know-It-All Project: A Case Study in Framework Development and Evolution, *Domain Oriented Systems Development: Perspectives and Practices.* K. Itoh, S. Kumagai, T. Hirota (eds), 101-118, Taylor & Francis, UK, 2002. ISBN: 0415304504.

[17] H.M. Deitel and P.J. Deitel, *Java: How to Program*, Prentice Hall, 1999.

[18] R. Ramakrishnan, D. Srivastava, S. Sudarshan, P. Seshadri, "The CORAL Deductive System", *VLDB Journal,* Vol.3, No.2, pp.161-210, 1994.

[19] A. Silberschatz, H.F. Korth, and S. Sudarshan, *Database System Concepts,* 4th Edition, McGraw-Hill, 2001.

[20] Ling Chen, User Interface Design for a Diagrammatic Query Tool, Major Report, Dept. of Computer Science, Concordia University, 2001.

[21] Liqian Zhou, Graphlog: its represntation in XML and Translation to CORAL, M*aster's Thesis*, Dept. of Computer Science, Concordia University, 2003.

[22] Guang Wang. Linking CORAL to MySQL and PostgreSQL. *Master's Thesis*, Dept. of Computer Science, Concordia University, 2004.

[23] Greg Butler, Guang Wang, Yue Wang, Liqian Zhou, A Graph Database with Visual Queries for Genomics, to appear in The Third Asia-Pacific Bioinformatics Conference, Singapore, 17-21 Jan, 2005.

[24] D. Harel, "On Visual Formalisms", *Communications of the ACM,* Vol.31, No.5, pp.514-530, 1988.

[25] C. Berge, *Graphs and Hypergraphs,* North-Holland Publishing Company, 1973.

[26] D. Hix and H.R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process,* New York, New York: John Wiley & Sons, Inc., 1983.

[27] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction,* 3rd Edition, Reading, MA: Addison-Wesley Publishing Co., 1997.

[28] J. Erlandson and J. Holm, "Intelligent Help Systems", *Information and Software Technology*, Vol.29, No.3, pp.115-121, 1987.

[29] M.M. Gardinar and B. Christie, editors, *Applying Cognitive Psychology to User-Interface Design*, John Wiley, Chichester, 1987.

[30] D. J. Mayhew, *Principles and Guidelines in Software User Interface Design,* Englewood Cliffs, NJ: Prentice Hall, 1992.

[31] R.L. Solso, and H.H. Johnson, *An Introduction to Experimental Design in Psychology: A Case Approach*, 4th Edition, New York, NY:Harper & Row, 1989.

[32] E.Gamma, R. Helm, R. Johnson, *Design Patterns*, Addison Wesley, 1995.

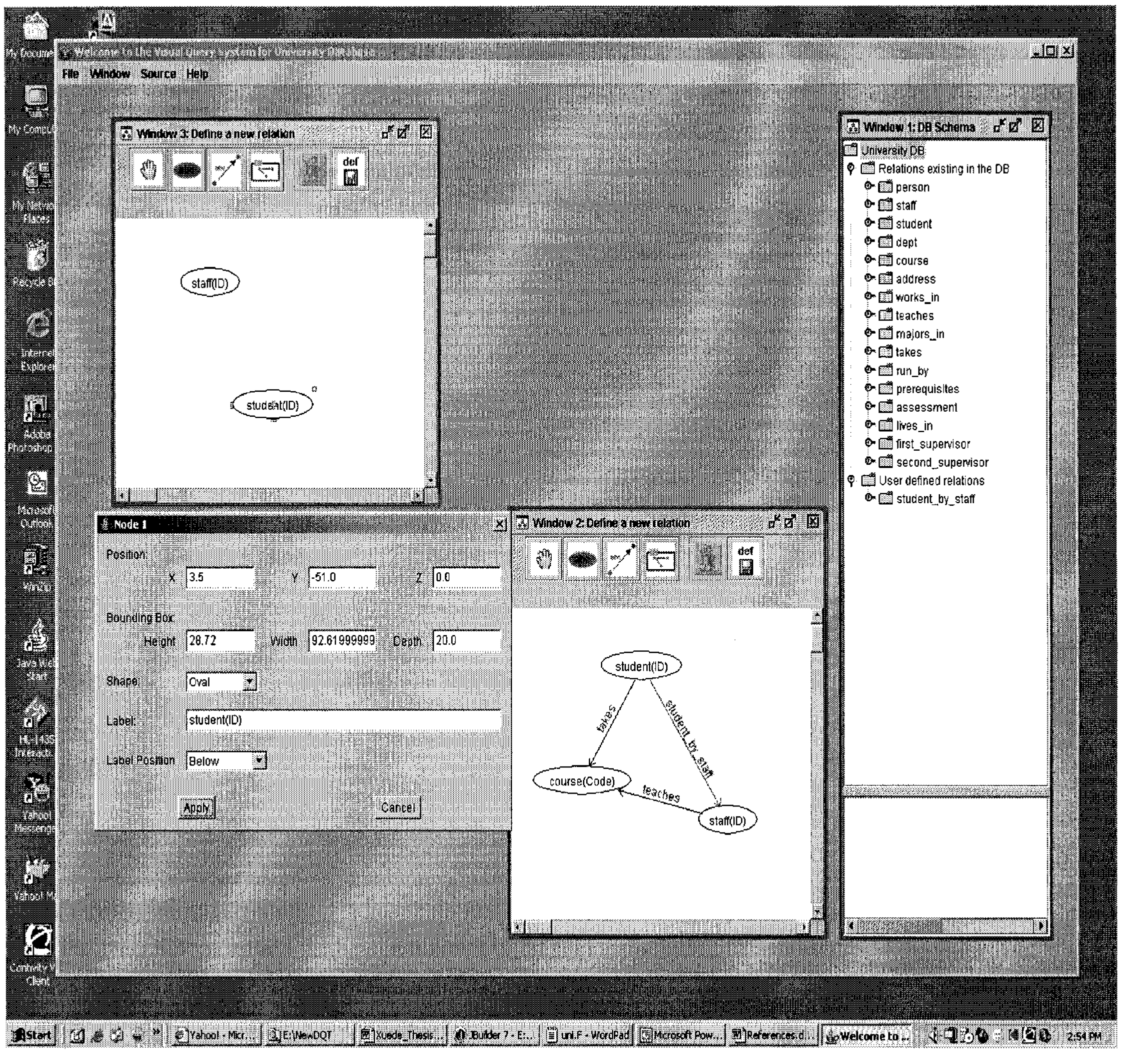
[33] E. Gansner, E. Koutsofios, S. North, *Drawing graphs with dot,* **http://www.graphviz.org/Documentation.php**, 2002.

[34] *A list of applications of Graphviz,* **http://www.graphviz.org/Resources.php**

[35] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo., A Technique for Drawing Directed Graphs. *IEEE Trans. Sofware Eng.*, 19(3):214–230, May 1993.

[36] T. Kamada and S. Kawai, An algorithm for drawing general undirected graphs, *Information Processing Letters*, 31(1):7–15, April 1989.

Appendix A Creating Nodes and Editing Node Properties



Node 1

Position:
X 3.5 Y -51.0 Z 0.0

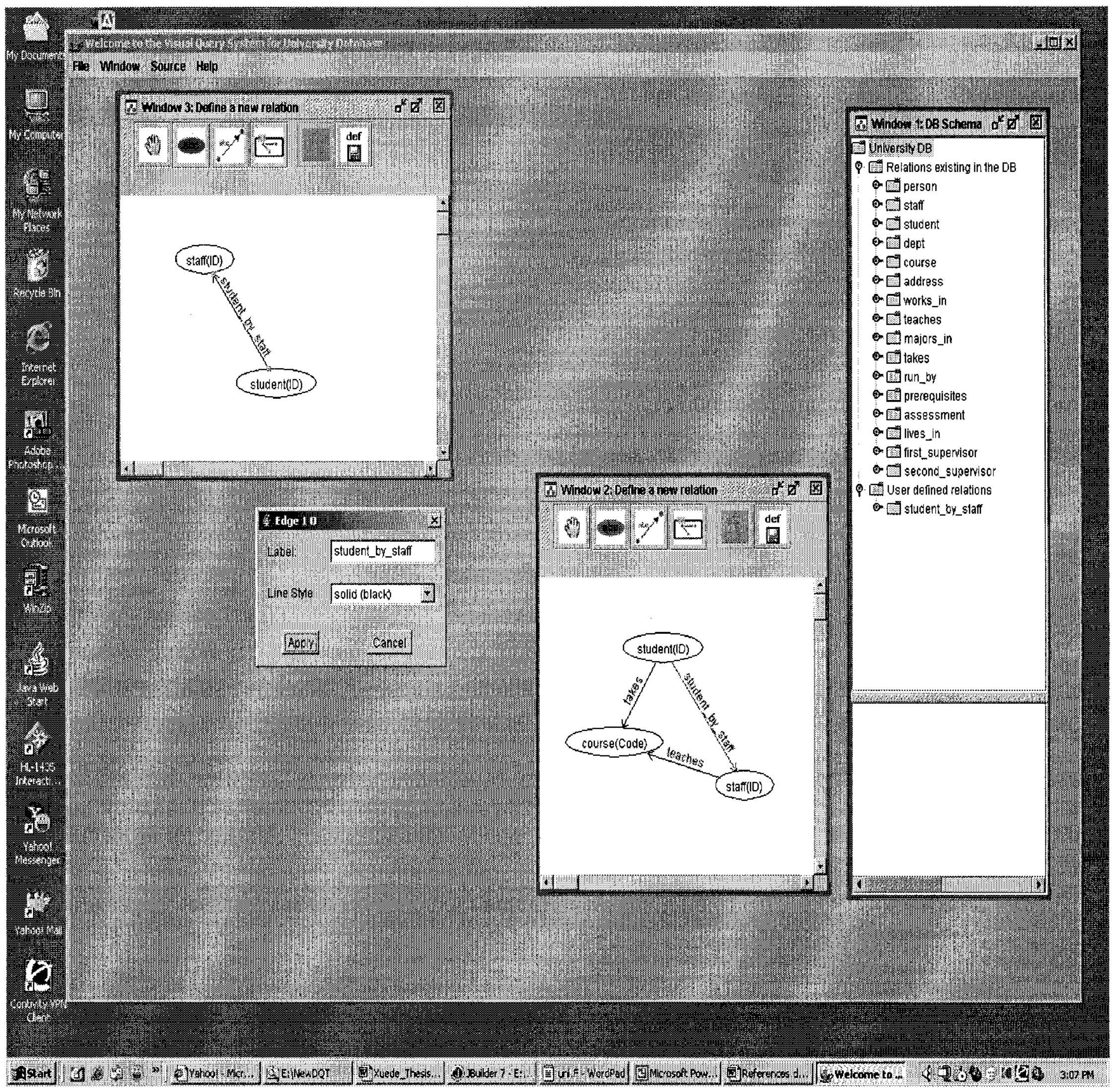
Bounding Box:
Height 28.72 Width 92.61999999 Depth 20.0

Shape: Oval

Label: student(ID)

Label Position: Below

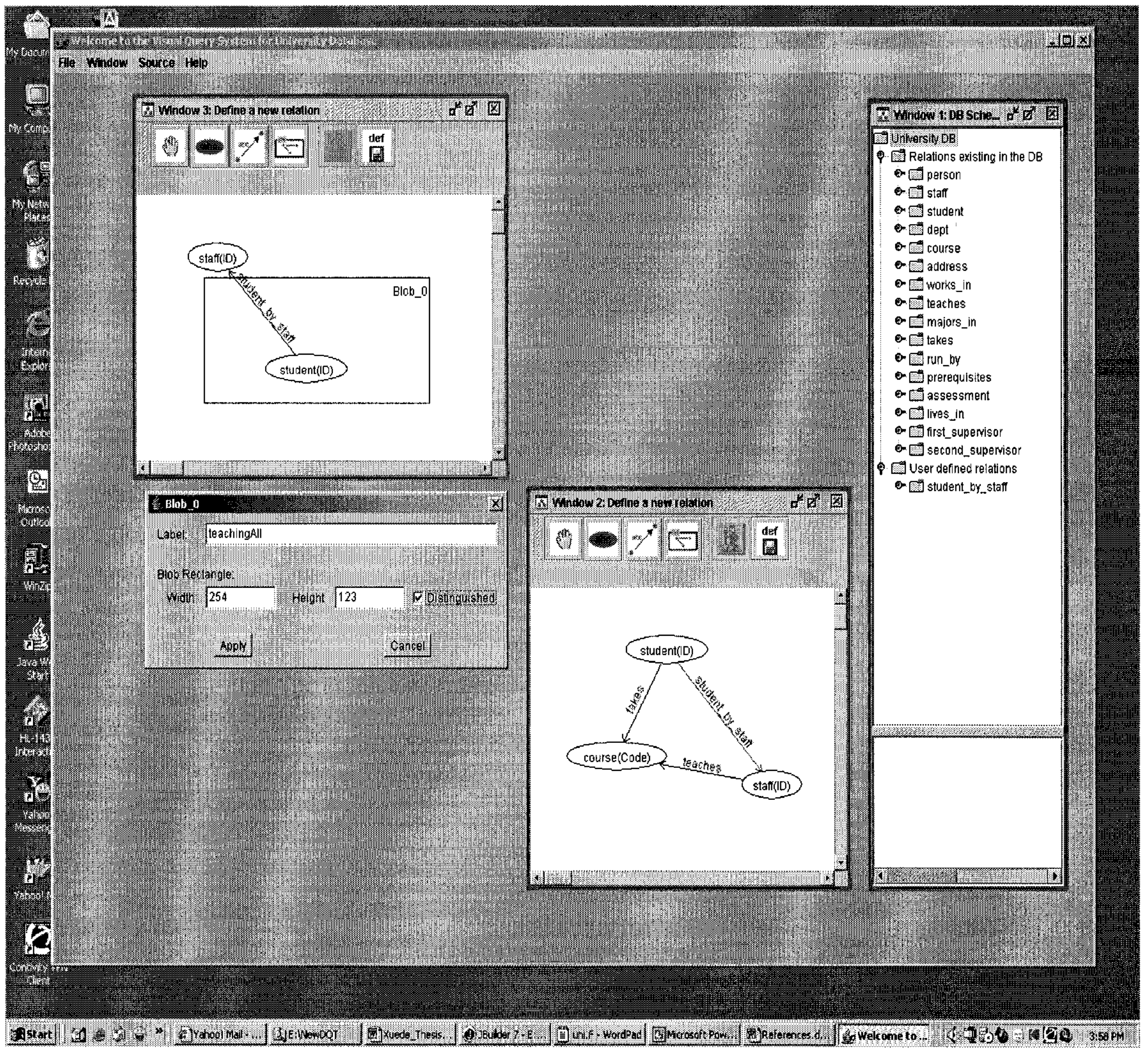
Appendix B Creating Edges and Editing Edge Properties



Label: student_by_staff

Line Style: solid (black)

Appendix C Creating a Blob and Editing Blob Properties



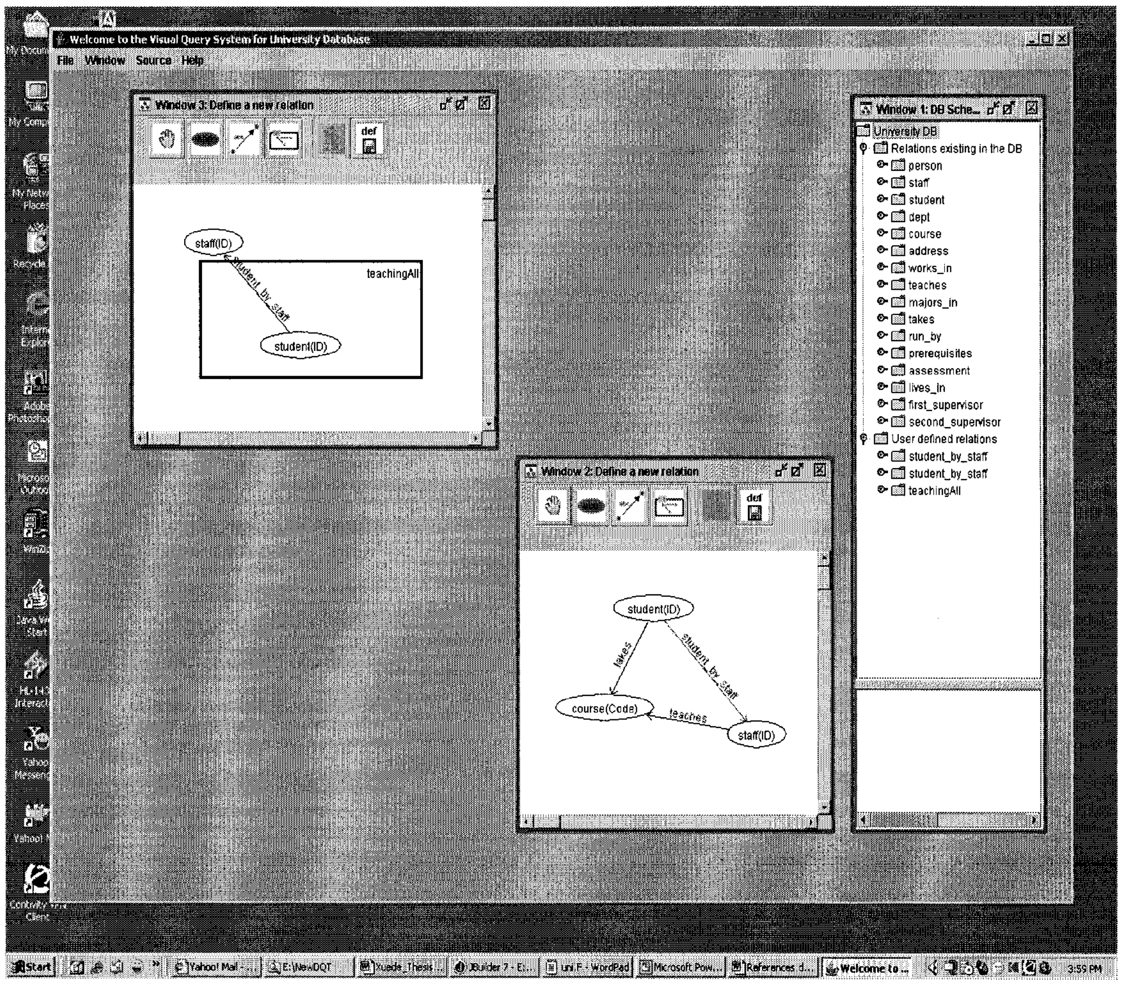
Blob_0

Label: teachingAll

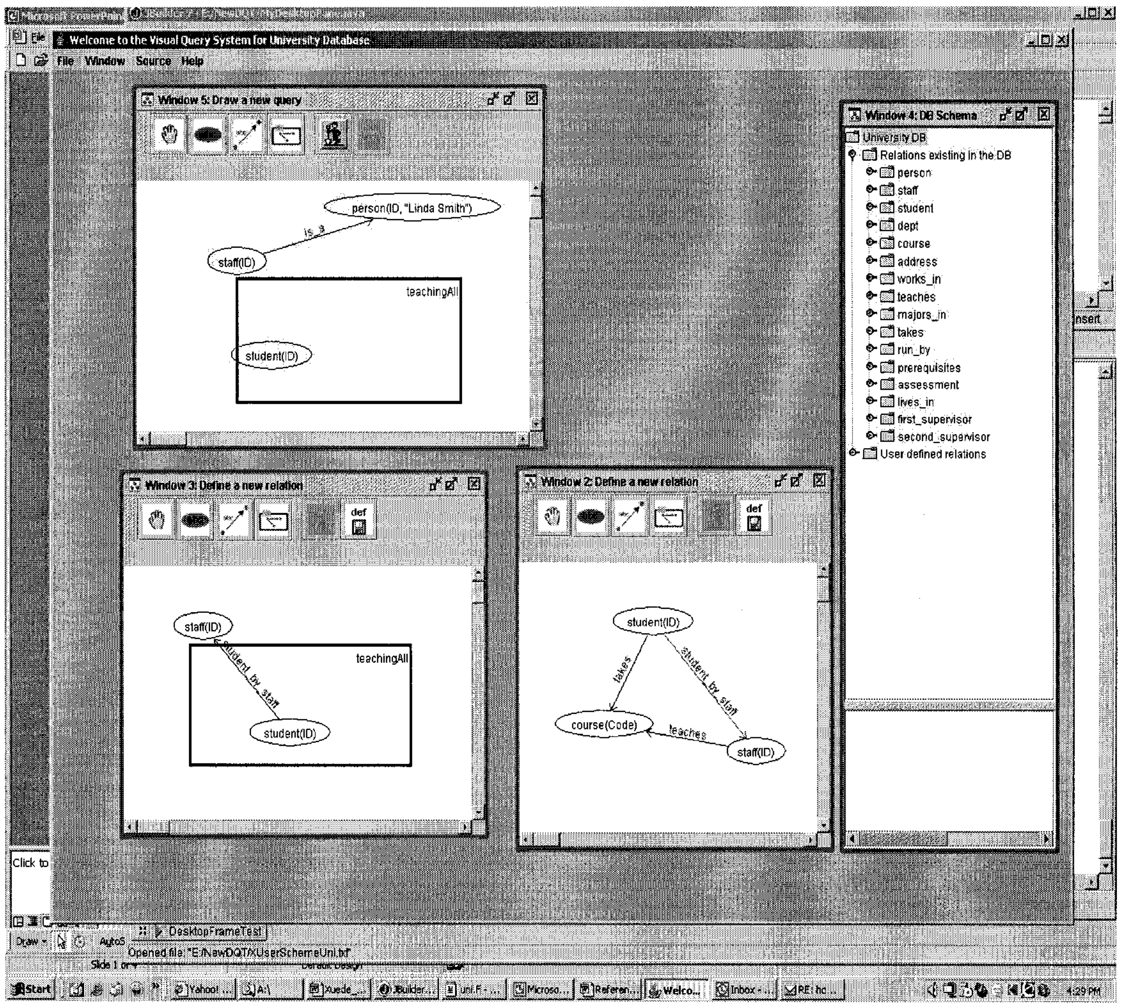
Blob Rectangle:

Width: 254 Height: 123 [✓ Distinguished]

Appendix D The "teachingAll" Blob (the Example in Figure 20)



Appendix E Query with Blob (for the Example in Figure 20)



Appendix F Blob Class – Attributes, Constructor, and Drawing Method

```
public class Blob
{
  private String label_;
  private boolean dummy_ = false;

  protected Node containerNode_;
  protected NodeList containedNodeList_;
  protected Rectangle blobRect_;
  protected boolean distinguished_ = false;

  public boolean selected_ = false;

  public Blob(Node container, NodeList containedList,
          Rectangle rect, String label, boolean dummy)
  {
    containerNode_ = container;
    containedNodeList_ = containedList;
    blobRect_ = rect;

    dummy_ = dummy;
    if ( dummy )
      label_ = new String("Blob_XYZ");
    else label_ =label;
  }

  public void draw(Graphics graphics, int quality, int which_gr)
  {
    graphics.setColor(Color.Balack);

    // draw the blob rectangle
    graphics.drawRect(blobRect_.x, blobRect_.y,
                      blobRect_.width, blobRect_.height);

    // make the blob rectangle thicker if it's distinguished
    if ( distinguished_ ) {
      graphics.drawRect(blobRect_.x - 1, blobRect_.y - 1,
                        blobRect_.width + 2, blobRect_.height + 2);
      graphics.drawRect(blobRect_.x + 1 , blobRect_.y + 1,
                        blobRect_.width - 2, blobRect_.height - 2);
    }

    // Draw label.
    if( quality > 0 && label_ != null && label_.length() > 0 )
    {
      FontMetrics fm = graphics.getFontMetrics();

      int x, y;
      x = (int)(blobRect_.x + blobRect_.width - fm.stringWidth(label_) - 2);
      y = (int)(blobRect_.y + fm.getHeight() + 2);

      graphics.drawString(label_,  x, y);
    }

  }

 ...... // Other methods are omitted.

}
```

Appendix G mouseDown: Initiation of Blob Creation and Proerty Edtion

```
public boolean mouseDown(Event e, int x_in, int y_in)
{
  // System.out.println("mouse down event in Graph_Canvas.");

  // finishing edge drawing if appropriate
  … … //omitted: edge finishing code

  // Determine which action to take.
  … … //omitted

  // Taking the action determined above
  if(currentMouseAction_ == 1) {  // create a new node
    … … // omitted
  } else if(currentMouseAction_ == 2) { // create a new edge
    … … // omitted
  } else if(currentMouseAction_ == 5) {  // Xuede: Create a new blob

    newContainerNode_ = null;

    if(selectedNode_ != null || selectedEdge_ != null) {
      unselectItems();
      paintOver();
    }

     // initiate a new blob if a container node has been selected
    if((newContainerNode_ = findNearestNode_(x_in, y_in, false)) != null) {
      DDimension bbox = newContainerNode_.getBoundingBox();

      containedNodeList_ = new NodeList();

      blobSelectX_ = x_in;
      blobSelectY_ = y_in + (int) bbox.height / 2;
      blobSelectX2_ = -1;  // First time flag.
    }

  } else if(currentMouseAction_ == 3) { // Select object.

    selected_ = NONE_;

    // Xuede: Blob Support
    if(selectedBlob_ != null && e.clickCount == 2)
    {
      //open the Property Dialog and get any changes
      setBlobProperties(false);

      return false;
    }

    … … // omitted

  }

  return false;
}
```

Appendix H mouseDrag: Keep Resizing the Blob Rectangle until mouseUp

```
public boolean mouseDrag(Event e, int x_in, int y_in)
{
... ... // omitted

  if(currentMouseAction_ == 5 && newContainerNode_ != null) // Xuede:
  {

    // the container node has been selected already (set in mouseDown)
    if ( x_in - blobSelectX_ > 0 && y_in - blobSelectY_ > 0 ) {

      // dragging a blob rectangle
      blobSelectX2_ = x_in;
      blobSelectY2_ = y_in;
      drawSelectRect_(blobSelectX_, blobSelectY_,
                      blobSelectX2_, blobSelectY2_);

    }

  }

... ... // omitted

  return false;
}
```

Appendix I mouseUp: Finish Blob Drawing and Capturing

```
public boolean mouseUp(Event e, int x_in, int y_in)
{

   ... ... // omitted

   // Xuede: Finish creating a blob - capturing contents, then paintOver!
   if (currentMouseAction_ == 5 && newContainerNode_ != null
           && ( blobSelectX2_ - blobSelectX_ > 0
           && blobSelectY2_ - blobSelectY_ > 0 ) )
  {

    double x1, y1, x2,y2;
    x1 = (double)Math.min(blobSelectX_, blobSelectX2_);
    y1 = (double)Math.min(blobSelectY_, blobSelectY2_);
    x2 = (double)Math.max(blobSelectX_, blobSelectX2_);
    y2 = (double)Math.max(blobSelectY_, blobSelectY2_);

    // Capture the blob rectangle
    Rectangle blobRect = new Rectangle( (int) x1, (int) y1,
                                        (int) (x2 - x1), (int) (y2 - y1) );

    // Selecting all the nodes and edges enclosed by the blob rectangle
    multiSelect_( x1, y1, x2, y2 );

    // Get all the contained nodes in the blob rectangle
    Node tmpNode;
    for( tmpNode = graph_.firstNode();
         tmpNode != null; tmpNode = graph_.nextNode(tmpNode) ) {
      if ( tmpNode.getId() != newContainerNode_.getId() ) {

        DPoint3 nodePos = tmpNode.getPosition3();

        nodePos.transform( viewTransform_ );
        if( nodePos.x >= x1 && nodePos.x <= x2
            && nodePos.y >= y1 && nodePos.y <= y2 ) {
          containedNodeList_.addNode( tmpNode );
        }
      }
    }

    Blob newBlob = new Blob( newContainerNode_,
                             containedNodeList_,
                             blobRect,
                             new String( "Blob_" + Integer.toString(
                                          newContainerNode_.getIndex() ) ),
                             false );
    graph_.insertBlob( newBlob );

    // create the blob visually
    paintOver();

  }

  ... ... // omitted

  return false;

}
```

Appendix J Translation of Blobs into TGL

```
public class TGLGraph {

  int type_; // EDGE type or Node type; Xuede: if (type_ == 12) it's BLOB type;

  … … // omitted: more definitions

  // Xuede: For Blob support
  public String blob_toString(){

    Blob thisBlob;

    if (objectValue_ == null)
      return "";
    else thisBlob = (Blob) objectValue_;

    String retstr = padding(2)+"<blob>\n";

    // add blob ID
    retstr += padding(3)+"<id>BID" + thisBlob.getLabel() + "</id>\n";

    // add blob label as predicate
    String blob_label = thisBlob.getLabel(); //default blob value
    if (! (thisBlob.getLabel().equals(""))) {
      blob_label = thisBlob.getLabel();
      if (blob_label.equals(new String(">")) || blob_label.equals(new String("<")))
        blob_label = "&" + blob_label;
    }
    retstr += padding(3)+ "<predicate>" + blob_label + "</predicate>\n";

    // add blob's outer node
    retstr += padding(3)+ "<outerNodeID>NID" +
              (thisBlob.getContainerNode().getId()) + "</outerNodeID> \n";

    // add blob's inner nodes
    NodeList innerNodeList = thisBlob.getContainedNodeList();

    Node node;
    for ( node = innerNodeList.firstNode();
          node != null; node = innerNodeList.nextNode(node) )
      retstr += padding(3)+"<innerNodeID>NID"+node.getId()+"</innerNodeID> \n";

    retstr += padding(2)+ "</blob>";

    return retstr;
  }

  … … // omitted: more methods


  public void divide_Content_Show(){

     … … // omitted

    TGLGraph obj;
    for(obj = this; obj != null; obj = obj.next_){

      if (obj.type_ == 10){
         … … // omitted: edge processing
      } else if (obj.type_ == 11){
         … … // omitted: node processing
      } else if (obj.type_ == 12) {  // Xuede: Blob Support

        Blob tempBlob = (Blob) obj.objectValue_;
        if (tempBlob.getDistinguished() == true) { // distinguished blob toShow
          toShow.addElement(obj);
          if ( (this.graph_type_).equals("defineGraphlog")){
            newRelName = tempBlob.getLabel();
          }
        } else content.addElement(obj);
      }
    }

  }

  public Vector define_toString() {

    … … // omitted: definitions

    while ( !toShow.isEmpty() && toShow.size()> counter){
      TGLGraph obj = (TGLGraph) toShow.elementAt(counter);
      if (obj.type_ == 10){
         … … // omitted: edge processing

      } else if (obj.type_ == 12) { // Xuede: Blob Support
        System.out.println("OBJ is a blob " +   obj.objectValue_.toString());
        Blob thisBlob = (Blob) obj.objectValue_;
        predicate_name = thisBlob.getLabel();
        new_rel.addElement(predicate_name);

        String containerNode = (thisBlob.getContainerNode()).getLabel();
        new_rel.addElement(containerNode);

        NodeList containedNodeList = thisBlob.getContainedNodeList();
        Node node = null;
        for ( node = containedNodeList.firstNode();
              node != null; node = containedNodeList.nextNode(node) ) {
          new_rel.addElement( node.getLabel() );
        }
      }
      counter++;
    }
    return new_rel;
  }

}
```

Appendix K Visualizing Blobs in the DOT Language and Visual Blob Pattern

```
private void visualizeResult() {

  // Omitted: definitions of variables
        :

   // 1. parse TGL XML result into internal data structures (3 vectors)
   //
   TGTranslator result = new TGTranslator();
   result.parseXML("E:/NewDQT/QueryResult.xml");

   // 2. construct the result graph & legend in dot, with object properties
   //
   GraphViz gv = new GraphViz();
   gv.addln(gv.start_graph());
   gv.addln( "fontsize=28;" );
   gv.addln( "compound=true;" );
   gv.addln( "size=\"9,8\"; ratio=fill; center=true;" );

  // first: go over the blob vector to make a cluster for each blob
   for ( i=0; i<result.blobVec.size(); i++ ) {
     tglBlob = (TGLBlob)(result.blobVec.elementAt(i)); // get a blob

     // get the blob's name, color, etc
     blobId = tglBlob.getBID();
     for (j=0; j<dotBlobVector.size(); j++ ) {
       dotBlob = (DOTBlob)(dotBlobVector.elementAt(j));
       if ( dotBlob.getId().equalsIgnoreCase( blobId ) ) {
         blobName = dotBlob.getName();
         blobColor = dotBlob.getFillColor();
         break;
       }
     }
     if ( j == dotBlobVector.size() )
       System.out.println("BID could not be found ??? " + blobId );

     // get the outer node
     outerNode = tglBlob.getOuterNode();
     outerNodeId = outerNode.getNID();
     outerNodeText = outerNode.getText();

     // make outer node rank lists
     for (j=0; j<outerNodeRankVector.size(); j++ ) {
       outerNodeRank = (Rank)(outerNodeRankVector.elementAt(j));
       if ( outerNodeRank.getObjectId().equalsIgnoreCase( blobId ) ) {
         outerNodeRank.addNode( "\"" + outerNodeText + "\";" );
         break;
       }
     }
     if ( j == outerNodeRankVector.size() )
       outerNodeRankVector.addElement( new Rank( blobId,
                                       "\""+outerNodeText+"\"" ) );

     // remove any node which is a duplicate of the outer node
     removeDuplicatedNode( outerNodeId, outerNodeText, result );

     // get the outer node's name, shape and color
     for (j=0; j<dotNodeVector.size(); j++ ) {
       dotOuterNode = (DOTNode)(dotNodeVector.elementAt(j));
       if ( dotOuterNode.getId().equalsIgnoreCase( outerNodeId ) ) {
         outerNodeName = dotOuterNode.getName();
```

```
    outerNodeShape = dotOuterNode.getShape();
    outerNodeColor = dotOuterNode.getFillColor();
    break;
  }
}
if ( j == dotNodeVector.size() )
  System.out.println("Outer NID could not be found ?" + outerNodeId );

// draw the outer node
gv.addln( "\"" + outerNodeText + "\"" +
          " [fontsize=22, shape=" + outerNodeShape +
          ", style=filled, fillcolor=" + outerNodeColor + "];" );

// make legend rank lists - add the outer node in if not there yet
for (j=0; j<legendNodeRankVector.size(); j++ ) {
  legendNodeRank = (Rank)(legendNodeRankVector.elementAt(j));

  if ( legendNodeRank.getObjectId().equalsIgnoreCase( outerNodeId ) )
    break;
}
if ( j == legendNodeRankVector.size() ) {
  legendNodeRankVector.addElement( new Rank( outerNodeId, outerNodeName,
                                   outerNodeShape, outerNodeColor ) );
}

// open a new dot cluster for this blob
gv.addln( "subgraph cluster" + i + " {" );
gv.addln( "style=filled;" );
gv.addln( "fillcolor=" + blobColor + ";" );
gv.addln( "labeljust=\"r\"; label=" + blobName + ";" );

// get an inner nodes' name, shape and color
// and then add this node into the dot graph
innerNodeList = tglBlob.getInnerNodeList();
for ( k=0; k<innerNodeList.size(); k++ ) {

  // get an inner node
  innerNode = innerNodeList.getTGLNodeAt(k);
  innerNodeId = innerNode.getNID();
  innerNodeText = innerNode.getText();

  // remove any node which is a duplicate of the inner node
  removeDuplicatedNode( innerNodeId, innerNodeText, result );

  // get the node's properties & add into the dot graph
  for (j=0; j<dotNodeVector.size(); j++ ) {
    dotInnerNode = (DOTNode)(dotNodeVector.elementAt(j));
    if ( dotInnerNode.getId().equalsIgnoreCase( innerNodeId ) ) {
      innerNodeName = dotInnerNode.getName();
      innerNodeShape = dotInnerNode.getShape();
      innerNodeColor = dotInnerNode.getFillColor();

      gv.addln( "\"" + innerNodeText + "\"" + " [shape=" +
                innerNodeShape + ", style=filled, fillcolor="
                + innerNodeColor + "];" );

      // Add the inner node into the legend list if not there yet
      int n;
      for (n=0; n<legendNodeRankVector.size(); n++ ) {
        legendNodeRank = (Rank)(legendNodeRankVector.elementAt(n));

        if (legendNodeRank.getObjectId().equalsIgnoreCase(innerNodeId))
          break;
          }
          if ( n == legendNodeRankVector.size() ) {
            legendNodeRankVector.addElement( new Rank( innerNodeId,
                  innerNodeName, innerNodeShape, innerNodeColor ) );
          }

          // remove the edge between the outerNode and the innerNode
          removeBlobEdge( outerNodeId, outerNodeText,
                          innerNodeId, innerNodeText, result );
          break;
        }
      }
      if ( j == dotNodeVector.size() )
        System.out.println("Inner NID could not be found?" +innerNodeId );
    }

    // define a dummy inner node for the dummy outer-inner connection
    dummyInnerNodeText = blobName + i;
    dummyInnerNodeShape = "point";
    dummyInnerNodeColor = blobColor;
    gv.addln( "\"" + dummyInnerNodeText + "\"" + " [style=invis, shape=" +
             dummyInnerNodeShape + ", color=" + dummyInnerNodeColor + ",
             style=filled, fillcolor=" + dummyInnerNodeColor + "]" );
    gv.addln( "}" ); // close the cluster for this blob

    // add the outer node and link it to the dummy inner node (the blob)
    gv.addln( "\""+outerNodeText+"\"" + " -> " + "\""+dummyInnerNodeText+"\""
       + " [style=bold, arrowsize=2, arrowhead=inv, arrowtail=dot, len=0.1, "
             + " tailport=s, headport=n, lhead=cluster" + i +"];" );
    gv.addln();
  } // loop for blobs

// Omitted: code for visualizing edges and then nodes
      :

// close the dot graph
gv.addln(gv.end_graph());

// Omitted: code for constructing the legend graph & calling Graphviz
      :
```

Appendix L Automated Translation and Visualization of Query Results

```
// Xuede: generate DOT graph, then call Graphviz to get visualization layout
//
private void visualizeResult() {

  int i, j, k;

  Vector outerNodeRankVector = new Vector();
  Rank outerNodeRank = null;

  Vector legendNodeRankVector = new Vector();
  Rank legendNodeRank = null;

  Vector legendEdgeVector = new Vector();
  LegendEdge legendEdge = null;

  DOTBlob dotBlob;
  DOTEdge dotEdge;
  DOTNode dotNode, dotOuterNode, dotInnerNode;

  TGLBlob tglBlob;
  TGLEdge tglEdge;
  TGLNode tglNode, outerNode, innerNode, fromNode, toNode;
  TGLNodeList innerNodeList;

  String blobId = "", blobName = "", blobColor = "", rankSet[];
  String outerNodeId = "", outerNodeName = "", outerNodeText = "",
         outerNodeShape = "", outerNodeColor = "";
  String innerNodeId = "", innerNodeName = "", innerNodeText = "",
         innerNodeShape = "", innerNodeColor = "";
  String dummyInnerNodeText="",dummyInnerNodeShape="",dummyInnerNodeColor="";

  String edgeId = "", edgeName = "", edgeColor = "", edgeStyle = "";
  String fromNodeId = "", fromNodeName = "", fromNodeText = "",
         fromNodeShape = "", fromNodeColor = "";
  String toNodeId = "", toNodeName = "", toNodeText = "",
         toNodeShape = "", toNodeColor = "";

  String nodeId = "", nodeName = "", nodeText = "",
         nodeShape = "", nodeFillColor = "";

  // 1. parse TGL XML result file into internal data st[illegible]ures (3 vectors)
  //
  TGTranslator result = new TGTranslator();
  result.parseXML("E:/NewDQT/testQueryResult.xml");

  //a.parseXML("/mnt/jeeves0/repository/Graphlog/result/tempQueryResult.xml");

  // 2. construct the result graph & legend in dot language
  //
  GraphViz gv = new GraphViz();
  gv.addln(gv.start_graph());
  gv.addln( "fontsize=28;" );
  gv.addln( "compound=true;" );
  gv.addln( "size=\"9,8\"; ratio=fill; center=true;" );

  // First: Go over result.blobVec
  for ( i=0; i<result.blobVec.size(); i++ ) {
    tglBlob = (TGLBlob)(result.blobVec.elementAt(i)); // get a blob

    // get the blob's name, color, etc
```

Note 1:
Parse the TGL result file.

```
blobId = tglBlob.getBID();
for (j=0; j<dotBlobVector.size(); j++ ) {
  dotBlob = (DOTBlob)(dotBlobVector.elementAt(j));
  if ( dotBlob.getId().equalsIgnoreCase( blobId ) ) {
    blobName = dotBlob.getName();
    blobColor = dotBlob.getFillColor();
    break;
  }
}
if ( j == dotBlobVector.size() )
  System.out.println("BID could not be found ??? " + blobId );

// get the outer node
outerNode = tglBlob.getOuterNode();
outerNodeId = outerNode.getNID();
outerNodeText = outerNode.getText();

// make outer node rank lists
for (j=0; j<outerNodeRankVector.size(); j++ ) {
  outerNodeRank = (Rank)(outerNodeRankVector.elementAt(j));
  if ( outerNodeRank.getObjectId().equalsIgnoreCase( blobId ) ) {
    outerNodeRank.addNode( "\"" + outerNodeText + "\";" );
    break;
  }
}
if ( j == outerNodeRankVector.size() )
  outerNodeRankVector.addElement(new Rank(blobId, "\""+outerNodeText+"\""));

// remove any node which is a duplicate of the outer node
removeDuplicatedNode( outerNodeId, outerNodeText, result );

// get the outer node's name, shape and color
for (j=0; j<dotNodeVector.size(); j++ ) {
  dotOuterNode = (DOTNode)(dotNodeVector.elementAt(j));
  if ( dotOuterNode.getId().equalsIgnoreCase( outerNodeId ) ) {
    outerNodeName = dotOuterNode.getName();
    outerNodeShape = dotOuterNode.getShape();
    outerNodeColor = dotOuterNode.getFillColor();
    break;
  }
}
if ( j == dotNodeVector.size() )
  System.out.println("Outer NID could not be found ? " + outerNodeId );

// draw the outer node
gv.addln( "\"" + outerNodeText + "\"" +
          " [fontsize=22, shape=" + outerNodeShape +
          ", style=filled, fillcolor=" + outerNodeColor + "];" );

// make legend rank lists - add the outer node in if it is not there yet
for (j=0; j<legendNodeRankVector.size(); j++ ) {
  legendNodeRank = (Rank)(legendNodeRankVector.elementAt(j));

  if ( legendNodeRank.getObjectId().equalsIgnoreCase( outerNodeId ) )
    break;
}
if ( j == legendNodeRankVector.size() ) {
  legendNodeRankVector.addElement( new Rank( outerNodeId, outerNodeName,
                                   outerNodeShape, outerNodeColor ) );
}

// open a new dot cluster for this blob
gv.addln( "subgraph cluster" + i + " {" );
```

```
//gv.addln( "orientation=portrait;" );
//gv.addln( "rankdir=LR;" );
gv.addln( "style=filled;" );
gv.addln( "fillcolor=" + blobColor + ";" );
gv.addln( "labeljust=\"r\"; label=" + blobName + ";" );

// get an inner nodes' name, shape and color
// and then add this node into the dot graph
innerNodeList = tglBlob.getInnerNodeList();
for ( k=0; k<innerNodeList.size(); k++ ) {

  // get an inner node
  innerNode = innerNodeList.getTGLNodeAt(k);
  innerNodeId = innerNode.getNID();
  innerNodeText = innerNode.getText();

  // remove any node which is a duplicate of the inner node
  removeDuplicatedNode( innerNodeId, innerNodeText, result );

  // get the node's properties & add into the dot graph
  for (j=0; j<dotNodeVector.size(); j++ ) {
    dotInnerNode = (DOTNode)(dotNodeVector.elementAt(j));
    if ( dotInnerNode.getId().equalsIgnoreCase( innerNodeId ) ) {
      innerNodeName = dotInnerNode.getName();
      innerNodeShape = dotInnerNode.getShape();
      innerNodeColor = dotInnerNode.getFillColor();

      gv.addln( "\"" + innerNodeText + "\"" +
                " [shape=" + innerNodeShape + ", style=filled, fillcolor="
                + innerNodeColor + "];" );

      // make legend rank lists - add the inner node in if not there yet
      int n;
      for (n=0; n<legendNodeRankVector.size(); n++ ) {
        legendNodeRank = (Rank)(legendNodeRankVector.elementAt(n));

        if ( legendNodeRank.getObjectId().equalsIgnoreCase(innerNodeId) )
          break;

      }
      if ( n == legendNodeRankVector.size() ) {
        legendNodeRankVector.addElement(new Rank(innerNodeId, innerNodeName,
                                         innerNodeShape, innerNodeColor));
      }

      // remove the edge between the outerNode and the innerNode
      removeBlobEdge( outerNodeId, outerNodeText,
                      innerNodeId, innerNodeText, result );

      break;
    }
  }
  if ( j == dotNodeVector.size() )
    System.out.println("Inner NID could not be found? " + innerNodeId );
}

// define a dummy inner for the dummy outer-inner connection
dummyInnerNodeText = blobName + i;
dummyInnerNodeShape = "point";
dummyInnerNodeColor = blobColor;
gv.addln( "\"" + dummyInnerNodeText + "\"" +
          " [style=invis, shape=" + dummyInnerNodeShape +
          ", color=" + dummyInnerNodeColor + ", style=filled, fillcolor="
```

Note 2:
Create a dummy connection for a blob.

```
                  + dummyInnerNodeColor + "]" );

        gv.addln( "}" ); // close the cluster for this blob

        // add the outer node and link it to the dummy inner nodes, i.e., the
blob.
        gv.addln( "\""+outerNodeText+"\"" + " -> "+"\""+dummyInnerNodeText+"\""+
                  " [style=bold,arrowsize=2,arrowhead=inv,arrowtail=dot,len=0.1,"+
                  " tailport=s, headport=n, lhead=cluster" + i +"];" );
        gv.addln();

      } // loop for blobs

      // define ranks for all the blobs
      for (j=0; j<outerNodeRankVector.size(); j++ ) {
        outerNodeRank = (Rank)(outerNodeRankVector.elementAt(j));
        gv.addln( "{ rank=same; " + outerNodeRank.getNodeList() + " }" );
      }

      /** Second: Go over result.edgeVec
       *
       */
      for ( i=0; i<result.edgeVec.size(); i++ ) {
        tglEdge = (TGLEdge)(result.edgeVec.elementAt(i)); // get an edge

        // get the edge's name, color, etc
        edgeId = tglEdge.getEID().trim();
        for (j=0; j<dotEdgeVector.size(); j++ ) {
          dotEdge = (DOTEdge)(dotEdgeVector.elementAt(j));

          if ( dotEdge.getId().trim().equalsIgnoreCase( edgeId ) ) {
            edgeName = dotEdge.getName().trim();
            edgeStyle = dotEdge.getStyle().trim();
            edgeColor = dotEdge.getColor().trim();
            break;
          }
        }
        if ( j == dotEdgeVector.size() )
          System.out.println("EID could not be found ??? " + edgeId );

        // get the from node
        fromNode = tglEdge.getFromNode();

        fromNodeId = fromNode.getNID().trim();
        fromNodeText = fromNode.getText().trim();

        // get the to node
        toNode = tglEdge.getToNode();

        toNodeId = toNode.getNID().trim();
        toNodeText = toNode.getText().trim();

        // remove any node which is a duplicate of the fromNode or the toNode
        removeDuplicatedNode( fromNodeId, fromNodeText, result );
        removeDuplicatedNode( toNodeId, toNodeText, result );

        // get the from node's name, shape and color
        for (j=0; j<dotNodeVector.size(); j++ ) {
          dotNode = (DOTNode)(dotNodeVector.elementAt(j));

          if ( dotNode.getId().trim().equalsIgnoreCase( fromNodeId ) ) {
            fromNodeName = dotNode.getName().trim();
            fromNodeShape = dotNode.getShape().trim();
```

Note 3:
Rank the blobs.

```
    fromNodeColor = dotNode.getFillColor().trim();
    break;
  }
}
if ( j == dotNodeVector.size() )
  System.out.println("FromNID could not be found ??? " + fromNodeId );

// get the to node's name, shape and color
for (j=0; j<dotNodeVector.size(); j++ ) {
  dotNode = (DOTNode)(dotNodeVector.elementAt(j));

  if ( dotNode.getId().trim().equalsIgnoreCase( toNodeId ) ) {
    toNodeName = dotNode.getName().trim();
    toNodeShape = dotNode.getShape().trim();
    toNodeColor = dotNode.getFillColor().trim();
    break;
  }
}
if ( j == dotNodeVector.size() )
  System.out.println("toNID could not be found ??? " + toNodeId );

// add the edge into the dot graph
gv.addln( "\""+fromNodeText+"\"" + " -> " + "\""+toNodeText+"\"" +
          " [style=" + edgeStyle + ", color=" + edgeColor + "];" );
// replace the above line with the one below if want to show edge labels
//" [label="+edgeName+", style="+edgeStyle+", color="+edgeColor+"];" );
gv.addln( "\"" + fromNodeText + "\"" + " [shape=" + fromNodeShape +
          ", style=filled, fillcolor=" + fromNodeColor + "];" );
gv.addln( "\"" + toNodeText + "\"" + " [shape=" + toNodeShape +
          ", style=filled, fillcolor=" + toNodeColor + "];" );

// make Legend Edge lists
for (j=0; j<legendEdgeVector.size(); j++ ) {
  legendEdge = (LegendEdge)(legendEdgeVector.elementAt(j));
  if ( legendEdge.getEdgeName().equalsIgnoreCase( edgeName )
       && legendEdge.getFromNodeName().equalsIgnoreCase( fromNodeName )
       && legendEdge.getToNodeName().equalsIgnoreCase( toNodeName ) )
    break;
}
if ( j == legendEdgeVector.size() ) {
  legendEdgeVector.addElement(new LegendEdge(fromNodeName, toNodeName,
                                       edgeName, edgeStyle, edgeColor));
}

// make legend rank lists - add the from/to node in if not there yet
int n;
for (n=0; n<legendNodeRankVector.size(); n++ ) {
  legendNodeRank = (Rank)(legendNodeRankVector.elementAt(n));

  if ( legendNodeRank.getObjectId().equalsIgnoreCase( fromNodeId ) )
    break;
}
if ( n == legendNodeRankVector.size() ) {
  legendNodeRankVector.addElement(new Rank(fromNodeId, fromNodeName,
                                         fromNodeShape, fromNodeColor));
}

for (n=0; n<legendNodeRankVector.size(); n++ ) {
  legendNodeRank = (Rank)(legendNodeRankVector.elementAt(n));

  if ( legendNodeRank.getObjectId().equalsIgnoreCase( toNodeId ) )
    break;
```

Note 4:
Add an edge.

```
  }
  if ( n == legendNodeRankVector.size() ) {
    legendNodeRankVector.addElement(new Rank(toNodeId, toNodeName,
                                           toNodeShape, toNodeColor));
  }

} // loop for edges

/** At Last: add all the orphan nodes still left in result.nodeVec
 *
 */
gv.addln( "{ rank=same; " );
for ( i=0; i<result.nodeVec.size(); i++ ) {
  tglNode = (TGLNode)(result.nodeVec.elementAt(i)); // get a node

  // get the node's name, color, etc
  nodeId = tglNode.getNID().trim();
  nodeText = tglNode.getText();
  for (j=0; j<dotNodeVector.size(); j++ ) {
    dotNode = (DOTNode)(dotNodeVector.elementAt(j));

    if ( dotNode.getId().trim().equalsIgnoreCase( nodeId ) ) {
      nodeName = dotNode.getName().trim();
      nodeShape = dotNode.getShape().trim();
      nodeFillColor = dotNode.getFillColor().trim();
      break;
    }
  }
  if ( j == dotNodeVector.size() )
    System.out.println("NID could not be found ??? " + nodeId );

  // add the node into the dot graph
  gv.addln( "\"" + nodeText + "\"" + " [shape=" + nodeShape +
            ", style=filled, fillcolor=" + nodeFillColor + "];" );

  // make legend rank lists - add the all orphan nodes in if not there yet
  int n;
  for (n=0; n<legendNodeRankVector.size(); n++ ) {
    legendNodeRank = (Rank)(legendNodeRankVector.elementAt(n));

    if ( legendNodeRank.getObjectId().equalsIgnoreCase( nodeId ) )
      break;
  }
  if ( n == legendNodeRankVector.size() ) {
    legendNodeRankVector.addElement(new Rank(nodeId, nodeName,
                                          nodeShape, nodeFillColor));
  }

} // loop for nodes
gv.addln( "}" );  // close node rank

// close the dot graph
gv.addln(gv.end_graph());
System.out.println(gv.getDotSource());

// 3. call graphviz to get the result layout
//
File resultOut = new File("E:/NewDQT/Graphviz/result.gif");
gv.writeGraphToFile(gv.getGraph(gv.getDotSource()), resultOut);

// 4. make legend graph & layout for the result g
//
GraphViz legend = new GraphViz();
```

Note 5:
Add nodes.

Note 6:
Call Graphviz to layout the result graph.

Note 7:
Construct the Legend graph.

```
    legend.addln(legend.start_graph());
    legend.addln( "fontsize=28;" );
    legend.addln( "size=\"1.6, 8\"; ratio=fill; center=true;" );
    legend.addln( "rankdir=LR;" );

    // define ranks for all the blobs
    legend.addln( "{ rank=same; " );
    for (i=legendNodeRankVector.size()-1; i>=0; i-- ) {
      legendNodeRank = (Rank)(legendNodeRankVector.elementAt(i));

      String legendNodeName = legendNodeRank.getNodeList();
      String legendNodeShape = legendNodeRank.getNodeShape();
      String legendNodeFillColor = legendNodeRank.getNodeFillColor();

      legend.addln("\""+legendNodeName.trim()+"\"" + " [shape="+legendNodeShape
                  +",style=filled,fillcolor="+legendNodeFillColor +"];" );
    }
    legend.addln( "}" ); // close rank


    // add edges into legend
    for (i=0; i<legendEdgeVector.size(); i++ ) {
      legendEdge = (LegendEdge)(legendEdgeVector.elementAt(i));

      legend.addln( "\""+legendEdge.getFromNodeName().trim()+"\"" +" -> "+
                  "\"" + legendEdge.getToNodeName().trim() + "\"" +
                  " [label=" + legendEdge.getEdgeName() +
                  ", labelfloat=true, labeldistance=0.2, style=" +
                  legendEdge.getEdgeStyle() +
                  ", color=" + legendEdge.getEdgeColor() + "];" );
    }


    // close the dot graph for the Legend
    legend.addln(legend.end_graph());
    System.out.println(legend.getDotSource());

    // layout the Legend
    File legendOut = new File("E:/NewDQT/Graphviz/legend.gif");
    legend.writeGraphToFile(legend.getGraph(legend.getDotSource()), legendOut);

  }
```