QUICK - A Graphical User Interface to Genome Multidatabases

Wang Chiew Tan
Ke Wang
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore, 119260
tanwangc@iscs.nus.sg, wangk@iscs.nus.sg

Limsoon Wong
Institute of Systems Science
Heng Mui Keng Terrace,
Singapore, 119597
limsoon@iss.nus.sg

Abstract

Formulating queries to access multiple databases can be a formidable task, especially when many terms from various databases and complex constraints are involved. To specify a multidatabase query, the user usually has to search through documents for exact database terms and learn the multidatabase language. This report presents QUICK (QUery Interface to CPL-Kleisli), a graphical user interface to multiple databases. CPL (Collection Programming Language) is a high-level multidatabase language built on top of an open query system Kleisli. QUICK allows users to handle overwhelming information from different data sources in an intuitive and uniform manner. The query specification is reduced to specifying user's terms in his/her own world, selecting paths and specifying constraints in a graph. Through the terms entered by the user, QUICK is able to reduce the scope of the views and paths to only those that are relevant to answering the user's query. Furthermore, QUICK is able to automatically generate a CPL query that corresponds to the user's intent. Additional graphical functions are provided for the user to fine-tune the query generated.

Keywords. application, database language, genomic data, multidatabase, system, user interface

1 Introduction

A multidatabase system is a distributed system that acts as a front end to many autonomous DBMSs and a global layer above the autonomous DBMSs through a global schema or a multidatabase language. The global user can access information from multiple sources in the multidatabase system in a single straightforward request. However, the multitude of information

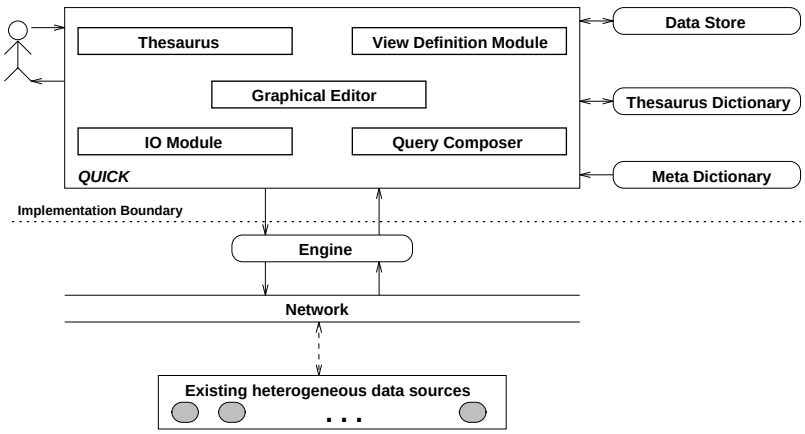

Figure 1: Overall architecture of the system.

available in multidatabase systems often impedes the user from quickly formulating a query. One reason is that the user often has to search through numerous manuals or documents for exact database terms in order to precisely specify a query. For example, it is difficult to be sure that employee identification number is termed "emp_id" and not "emp-id" or "employee_id" in the multidatabase. Although this problem also exists for single database systems, the magnitude of information in multidatabase environments makes it a more immediate problem. For instance, the schema documentation of GDB [9] (Genome Data Base), a collection of databases providing human genome information, is well over 300 pages. In addition, multidatabase users are usually occasional users, in the sense that they use their home databases most of the time and access multidatabases only occasionally. As such, it is unreasonable to require multidatabase users to provide exact database terms. Some form of help should be given to reduce the user's effort in query specification.

The traditional textual query formulation requires syntactic and semantic knowledge of the language. A large number of graphical user interfaces exist for single database systems which make query specification more user-friendly. However, the issue of graphical user interfaces is not well-addressed in multidatabase systems, see related work in Section 2. In this paper, we present a prototype system QUICK (QUery Interface to CPL-Kleisli) to address this issue. The CPL [6] (Collection Programming Language) is a high level multidatabase language that can handle nested relations and structured files. QUICK is built on top of an open query system called Kleisli [17] and is a graphical query interface to multiple autonomous and heterogeneous databases. The purpose of QUICK is to minimize the effort of end users in formulating queries for multidatabase systems. QUICK allows fast query formulation even with sporadic users having neither sufficient knowledge of query languages nor extensive prior knowledge of database structures. QUICK is written in Tcl 7.4/Tk 4.0 and it can be executed in any unix environment with X-Windows system and Tcl 7.4/Tk 4.0 installed. To run a query, CPLTCL, a variant of CPL for interfacing with TCL, is required to be installed.

Figure 1 shows the overall architecture of the system. QUICK is running on the Engine module, CPLTCL in our implementation, that executes the query sent by QUICK. To replace CPLTCL by another multidatabase language, only the Query Composer and Meta Dictionary

need to be replaced. The Thesaurus Dictionary provides a synonym mapping between user terms and database terms. The Meta Dictionary contains the schema information about views and frequently used predicates between views in the form of graphs. We consider general predicates that are not necessarily join. The Meta Dictionary can only be modified by the DBA. The editing of graphs during a particular user session have no effect on the Meta Dictionary. Instead, the editing result is saved onto a separate user file kept in the Data Store. The user can retrieve this file in a later session. Within the QUICK, there are five main modules. The Thesaurus is responsible for extracting corresponding database terms for the user specified terms. The View Definition is responsible for extracting the subgraph that contains the database terms returned by the Thesaurus. The Graphical Editor is the core module which supports essential graphical functions. The IO module is responsible for accessing session files in Data Store. Finally, the Query Composer generates a well-formed CPL query from a graphical specification.

There are three layers in the use of this system – the Thesaurus layer, Graph layer, and CPL layer. A user can enter the system at any of these layers. An expert user may like to enter the system at the lowest CPL layer, by directly formulating a query in CPL but in the comfort of the graphical environment. A naive user may like to enter from the Thesaurus layer or Graph layer. The Thesaurus layer is good for users with minimal knowledge of databases and when only user terms are known. To map user terms to corresponding database terms, an interactive confirmation by the user may be needed and certain context information such as description of database terms and containing views and databases will be available to help with the confirmation. Based on confirmed database terms, the View Definition module retrieves the relevant portion view and schema definitions from the Meta Dictionary and presents it to the user in the form of a subgraph. Then the user proceeds to formulate queries using graphical interface functions provided by QUICK. The user can also choose to skip the Thesaurus layer and work with the entire graph or retrieve a subgraph by manually removing the irrelevant nodes — the Graph Layer. This is suitable for users with some knowledge of the underlying databases.

Though the use of QUICK in the paper is based on an example from the Human Genome Project, QUICK is a generic interface for general multidatabase applications. For a new application running on CPL, only new Meta Dictionary and Thesaurus Dictionary need to be created (by the DBA); for a new application running on a multidatabase language rather than CPL, the Query Composer also needs to be substituted. The rest of the system, as shown in Figure 1, remains unchanged. The rest of this paper is organized as follows. In Section 2, we review related interface work on multidatabase systems. In Section 3, we describe the example biological databases used in this paper. In Section 4, we show how a multidatabase query can be formulated with our prototype system QUICK using some biological databases. In Section 5, we present the coherence theory which is the basis of the correctness of our Query Composer. The conclusion is given in Section 6.

2 Related Work

Most multidatabase research projects have emphasized on schema conflict resolution, query optimization, query processing and concurrency control. Query formulation for multidatabase systems has been predominantly textual. For example, see [15, 19, 18]. Most aim to perform

query transformation and optimization, but have much neglected the graphical user interface aspect of multidatabase systems.

Graphical user interface, which has evolved from textual languages to the WIMP (Windows, Icons, Menus, Pointing devices) metaphor, is a mature research area for single database systems. A large number of graphical user interfaces already exist, mostly based on the popular relational databases (for example, SUPER [24] and CANDID [23]). There has been a growing number of graphical user interfaces for object-oriented databases, e.g., [16, 20, 22, 1]. Most of these interface work focus on schema editing operations and navigations, and translation from a graphical specification into a query program is mostly on a single database system based on the relational model.

The multidatabase graphical user interface in [25] uses the object-oriented approach to structure its databases. Standard retrieval functions are available for accessing objects. However, the user is still responsible to explicitly specify which databases, object classes and operations to invoke. Each instance of a class or attribute is returned in a separate window, which could be a problem if there are many instances to be returned. In our approach, the thesaurus helps naive users to immediately scope down to relevant views and relationships. Join and selection conditions can be added dynamically and the results are returned in a nested relation in one window.

3 Biological Data Sources

In this paper, the query specification by QUICK will be demonstrated through an example based on two biological data sources, namely, GDB (Genome Database) and GenBank [13]. GDB is a Sybase relational database that supports biomedical research, clinical practice and professional and scientific education by providing human gene mapping information. It is an international collaboration sponsored by biomedical funding agencies worldwide. GenBank is a genetic sequence database which is a collection of all known DNA sequences. We will be accessing GenBank via Network Entrez which is a retrieval program for a specially formatted text file that contains all information of GenBank in a certain release. An average record in this source has over 12 levels of nesting and over 150 different kinds of subobjects. These databases are part of the Human Genome Project whose primary aim is to identify all genes in the human genome and to sequence 3 billion bases of DNA that comprise the human genome. These biological sources are chosen for two reasons. First, it is an open research problem to integrate these genome databases well (see [14]). Second, databases in the Human Genome Projects are among the most complex and diverse information sources in the world. The study on such applications will have general implications on a total solution.

Important genomic data exist in a number of distributed databases, e.g., GDB is hosted at The John Hopkins University at Baltimore, Maryland, whereas GenBank is located at National Center for Biotechnology Information which is part of National Library of Medicine which is in turn part of National Institute of Health at Washington, D. C. They also exist in a number of different formats. For example, GDB is a Sybase relational database and GenBank is a data source consisting of structured files in the ASN.1 (Abstract Syntax Notation) format. These are just two of the databases in the Human Genome Project. It was noted in [3] that structured files in various formats are adopted in preference over database management systems for the

following reasons. Biological data are generally complex structures in its natural form. It can include sequential data as well as deeply nested record structures. It can also contain a number of data types not supported in conventional databases, such as lists and variants which may also be deeply nested. Hence it is difficult to represent these data using the relational model. Besides, structured files can also be easily accessed from programs written in languages such as C which in turn makes them easily available for a variety of platforms as well as special retrieval packages. The constant need of database restructuring[1] makes object-oriented databases an unsuitable choice. Therefore, many genomic data are best represented as free text or structured text files. At the same time, there is also a need to integrate these structured files with traditional databases.

Genomic data are not only diverse in type, but also large in size. A typical genetic database can consist of hundreds of tables and thousands of database terms. GDB version 5.5 contains close to 400 tables and approximately 1,300 database terms. A typical genetic query usually requires joins spanning relations in several distributed databases. Currently, there are many special-purpose HTML forms available on the internet such as the Entrez Browser [8], Query Forms for searching data in GDB [10, 11, 12]. As these interface tools only put certain aspects of the data online, they do not allow flexible access to all the data available. Although powerful query languages such as CPL offer flexible access to these data sources, they often require strong syntax and semantic knowledge of the language before a query can be formulated. QUICK is one step towards a more flexible and friendly interface to multiple sources in such applications.

4 An Application with QUICK

Since CPL is the target query language of QUICK, we give a brief introduction to it.

CPL uses the comprehension syntax [4]. A CPL query has the form

$$\{ e \mid GF_1, ..., GF_n \}$$

called a *comprehension*. GF_i is of the form `\y <- R` or is a condition such as "y.#name = z.#name". e is an expression for the result to be returned. A CPL query can be read in a way similar to tuple calculus: "The set of all e such that GF_1, ..., GF_n."

XXX: Please thoroughly explain y, z, R, #, and ˙

For example, in the expression

```
{(#name:m, #matric_no:n) | (#name:\m,#matric_no:\n,...) <- STUDENT};
```

where \m is a simple pattern that matches anything and binds it to "m". Subsequent references to "m" will use this binding. The same goes for `\n` . "..." matches anything. # marks labels or attribute names. Thus, the above expression matches each tuple in STUDENT partially.

XXX: does the above computes the projection of STUDENT on name and matric_no? If so, explicitly say so.

A "primitive" is analogous to the concept of a function in programming languages. For example, the primitive below adds one to its argument, and the statement "addone(8);" invokes the primitive and the returned result will be 9.

[1] Database restructing occurs when new experimental techniques are developed or new generalization and higher order biological laws are discovered.

```
primitive addone == \x => x + 1.
```

In CPL, the construct for sending a request e to a server N has the form

```
\cpl{process} $e$ \cpl{using} $N$.
```

Comments are indicated by "!".

A Real Application

Consider the following four views in Figure 2 derived from databases GDB and GenBank. View GDB-locus contains the locus summary information. View GDB-object_genbank_eref contains the class description of genes. View GDB-locus_cyto_location contains the information about chromosomes. View GENBANK-entrez_summary contains the GenBank summary. The first three views are related through identifiers locus_id and object_id, and the second and fourth views are related through attributes genbank_ref and accession. Relating attributes share a common domain and are links for specifying across-view queries. The CPL definitions of views, created by the DBA in the Meta Dictionary, are given below. In our convention, each view name has the form "DBname-Viewname". So, GDB-locus means the view locus derived from database GDB. It is also possible that a view is derived from more than one database. GENBANK-entrez_summary is one such example. It contains information from both GDB and GenBank but the main information (summary) comes from GenBank. Therefore, we still call it GENBANK-entrez_summary. By leaving out "DBname", the user will be completely unaware of the location of each view. However, for clarity, we include database names as part of node names in the display.

The following are the CPL definitions of the first three views in Figure 2. These definitions are kept in the Meta Dictionary and are automatically referenced by QUICK through involved database terms. That is, the user does not need to input them at all.

```
primitive GDB-locus ==
  { (#locus:
        (locus_id: x.#locus_id,
         #locus_loc_summary: x.#locus_loc_summary,
         #locus_name: x.#locus_name,
         #locus_symbol: x.#locus_symbol)) |
     \x <- process "select * from locus l where 1=1" using gdb };

XXX: ! one line comment for the above. explain what is locus 1 and 1=1?

primitive GDB-object_genbank_eref ==
  { (#object_genbank_eref:
        (#genbank_ref: x.#genbank_ref,
         #object_class_key: x.#object_class_key,
         #object_id: x.#object_id)) |
     \x <- process "select * from object_genbank_eref o where 1=1" using gdb };

XXX: ! one line comment for the above
```

View Name	Attributes	Attribute Description
GDB-locus	locus_id	unique internal identifier for a locus.
	locus_loc_summary	summary listing of the cytogenetic location for the locus.
	locus_name	HGM-approved name given to describe this locus.
	locus_symbol	HGM-approved symbol given to describe this locus.
GDB-object_genbank_eref	genbank_ref	GenBank accession number, that is, an external identifier.
	object_class_key	number key indicating the type of database object. For locus object, this number is 1.
	object_id	unique internal identifier for this object.
GDB-locus_cyto_location	loc_cyto_band_end	cytogenetic band at the qter end of the locus location
	loc_cyto_band_start	cytogenetic band at the pter end of the locus location.
	loc_cyto_chrom_num	chromosome number on which the locus is located.
	locus_id	unique internal identifier for a locus.
GENBANK-entrez_summary	accession	accession number.
	uid	internal identifier used by GenBank
	title	name for this accession.

Figure 2: The views of each node in the subgraph of Figure 4

```
primitive GDB-locus_cyto_location ==
  { ( #locus_cyto_location:
        (#loc_cyto_band_start: x.#loc_cyto_band_start,
         #loc_cyto_band_end: x.#loc_cyto_band_end,
         #loc_cyto_chrom_num: x.#loc_cyto_chrom_num,
         #locus_id: x.#locus_id)) |
     \x <- process "select * from locus_cyto_location l where 1=1" using gdb };

XXX: ! one line comment for the above
```

The last of the four views in Figure 2 is more complicated. For each GDB accession in the GDB database, the view retrieves the entrez_summary in the GenBank database through the network entrez. This provides a cross reference between GDB and GenBank on nucleic acid entries. The condition "object_class_key = 1" specifies human genes.

```
primitive GENBANK-entrez_summary == {
   ( #entrez_summary: ( #accession: x, #uid: u, #title: t ) ) |
      ( #accn: \x ) <-
         process "select distinct accn = o.genbank_ref
                  from object_genbank_eref o
                  where o.object_class_key = 1"
```

```
      using gdb,
   ( #title: \t, #uid: \u, ... ) <-
      process ( #db: "na",
                #args: "-D",
                #path: "Seq-entry",
                #select: "accession " ^ x )
      using entrez };

XXX: ! one line comment here
```

Suppose that we wish to answer the following query: *Retrieve the GenBank summary and locus summary information about human genes on chromosome 20p13.*

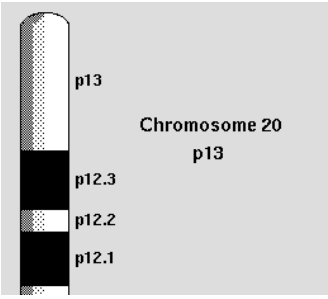

Figure 3: Part of chromosome 20 showing the p13 band.

The steps to specify the query using QUICK are described below.

Step 1. Specify the user terms. The user enters terms in his/her own world through the Thesaurus module. For the above query, user terms are entered through the following SQL-like statement

```
SELECT summary,locus,human genes, chromosome
WHERE chromosome=20 and band start=p13
```

where "locus", "human genes", "summary", "chromosome", "band start" are user terms, which are mentioned, explicitly or implicitly, in the query either as the data to be retrieved or as the constraint of such data. If the user further knows that these information are contained in databases "GDB" and "GenBank", he/she may enter these database names in a "WITHIN" sub-statement. Both "WHERE" and "WITHIN" are optional.

There are important differences between a standard SQL statement and the above SQL-like statement. First, all terms in the above statement are user terms, not database terms. Second, the above statement does not have the "FROM" sub-statement because the user is not required to know the views or nodes containing the required data. In other words, through the above statement the user specifies what is wanted in his/her own terms as if there were a universal relation containing all data items. It is the job of the Thesaurus modules to map the user terms to database terms, with some degree of interaction with the user, and to decide and retrieve views containing the required data, which are nodes in graphs contained in the Meta Dictionary. For this example, the Thesaurus module will retrieve (i) the node GDB-locus (containing the locus summary information) due to user terms "locus" and "summary", (ii) the node GDB-locus_cyto_location (containing the cytogenetic location information for locus objects) due to user terms "locus", "band start" and "chromosome", (iii) the node GDB-object_genbank_eref (containing an attribute for restriction to human genes) due to user term "human genes". Since node

GENBANK-entrez_summary provides a cross reference between GDB and GenBank on nucleic acid entries and contains information about locus, it is also returned. Corresponding database terms in these nodes and their information are presented to the user for confirmation. The confirmed database terms are then passed to the View Definition module which will extract a subgraph to be displayed on the Graphical Editor.

Step 2. Edit the retrieved subgraph. The subgraph retrieved is displayed in Figure 4. A list of pre-defined predicates is associated with an edge. The CPL definitions of views for nodes and definitions of predicates for edges can be examined by clicking on the edge or node, such as in Figure 6 and Figure 7. Frequently used predicates on edges are maintained in the Meta Dictionary. However, predicates can be added or deleted and unwanted nodes can be deleted during a user session. Such updates are local to the separately stored session file; the underlying graphs in the Meta Dictionary remain unchanged. From the displayed subgraph the user will select edges and predicates on edges to compose the query. For the above query, three selected predicates are shown in Figure 5. When the graphical editing is completed, the query is formulated with the click of a button. A name can be given to a saved query for later references; in our case, ENTREZ-OBJECT-CYTO-LOCUS, which is the concatenation of names of views involved.

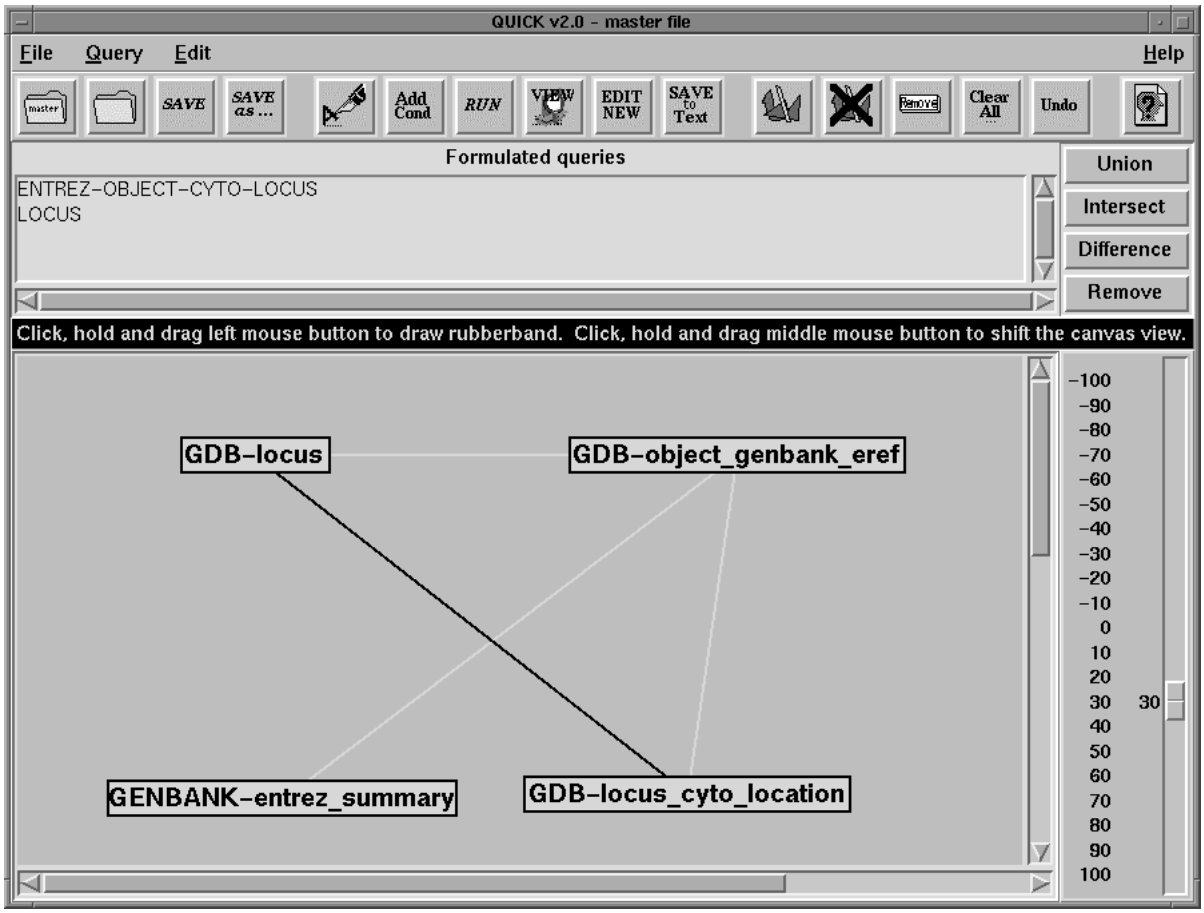

Figure 4: A display of a subgraph in Graphical Editor

Step 3. Specifying additional conditions. The additional conditions that the chro-

Selected edge	Selected predicates with definitions in CPL

GDB-locus
GDB-object_genbank_eref
	JOIN_locus_genbank
	
	`primitive JOIN_locus_genbank == (\L,\R) =>`
	`L.#locus.#locus_id = R.#object_genbank_eref.#object_id`
	`andalso R.#object_genbank_eref.#object_class_key = 1;`

GDB-object_genbank_eref
GDB-locus_cyto_location
	JOIN_object_cyto
	
	`primitive JOIN_object_cyto == (\L,\R) =>`
	`L.#object_genbank_eref.#object_id =`
	`R.#locus_cyto_location.#locus_id andalso`
	`L.#object_genbank_eref.#object_class_key = 1;`

GENBANK-entrez_summary
GDB-object_genbank_eref
	JOIN_entrez_summary_object_genbank
	
	`primitive JOIN-entrez_summary_object_genbank ==`
	`(\L, \R) => L.#entrez_summary.#accession =`
	`R.#object_genbank_eref.#genbank_ref;`

Figure 5: The selected edges and selected predicates

mosome is No. 20, that cytogenetic band at the pter end of the locus location is "p13", and that only human genes are of interest, are specified by selecting the "Add Condition" function in the "Query" menu on the upper-left part of the window. This window displays the query in the familiar SQL format. For example, Figure 9 shows the selection of four attributes from the intermediate view ENTREZ-OBJECT-CYTO-LOCUS formulated in Step 2, with additional conditions appended to the WHERE portion of the SQL window. The resulting CPL query, which is generated automatically by QUICK, is shown below.

```
! The name convention of intermdiate views: the concatenation of the
! names of individual views involved.

! Connects to databases GDB and Entrez (in GENBANK) with 1 connection each.

connect-to-gdb(1);
connect-to-entrez(1);

! CPL definitions of nodes and predicates involved are produced here but
! not shown. They can be found in on Page 7 and Figure 5.
!   :
!   :

! Intermediate views ------------------------------------------------------

! The first intermediate view between GENBANK-entrez_summary and
! GDB-object_genbank_eref generated by the Query Composer.
```

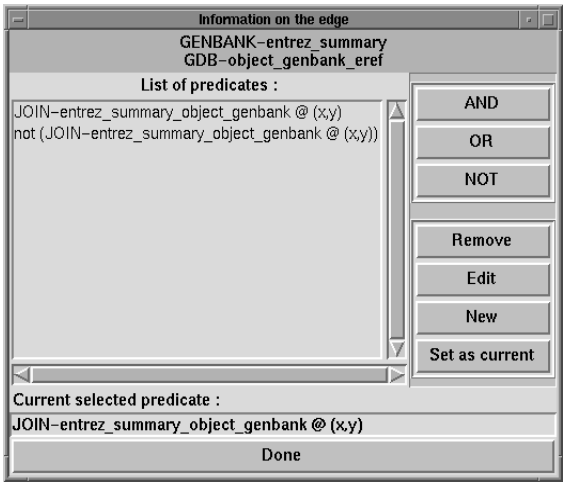

Figure 6: Information on the edge, by double clicking the edge.

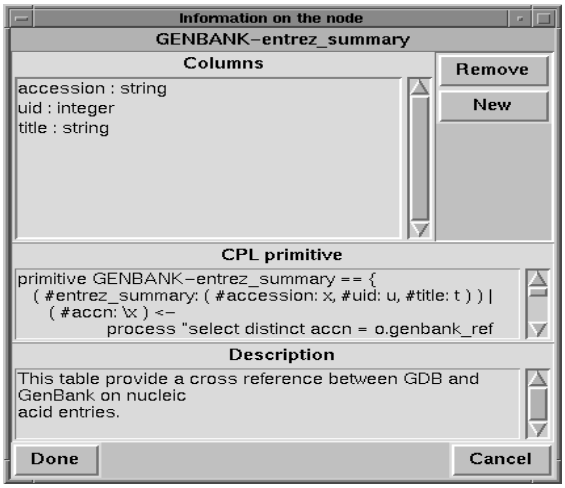

Figure 7: Information on the view, by double clicking the node.

```
primitive ENTREZ-OBJECT == {
     ( #entrez_summary: x.#entrez_summary,
       #object_genbank_eref: y.#object_genbank_eref ) |
     \x <- GENBANK-entrez_summary,
     \y <- GDB-object_genbank_eref,
     JOIN-entrez_summary_object_genbank(x,y)
};


! The second intermediate view between ENTREZ-OBJECT and
! GDB-locus_cyto_location generated by the Query Composer.

primitive ENTREZ-OBJECT-CYTO == {
     ( #entrez_summary: x.#entrez_summary,
       #object_genbank_eref: x.#object_genbank_eref,
       #locus_cyto_location: y.#locus_cyto_location ) |
     \x <- ENTREZ-OBJECT,
     \y <- GDB-locus_cyto_location,
     JOIN_object_cyto(x,y)
};

! The third intermediate view (also the final view) between
! ENTREZ-OBJECT-CYTO and GDB-locus generated by the Query Composer.
!
! User specified conditions are inserted in this final view.
! #locus_cyto_location.#loc_cyto_band_start = "p13" is to specify that
! only p13 band is required.
! #locus_cyto_location.#loc_cyto_chrom_num = "20" is to specify that
! only chromosome 20 is required.

primitive ENTREZ-OBJECT-CYTO-LOCUS == {
```

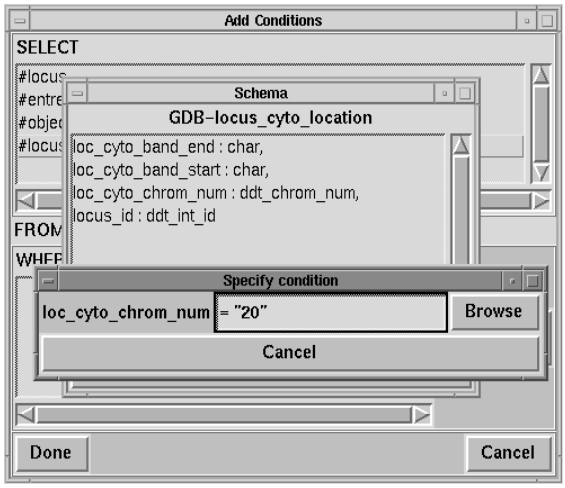

Figure 8: Adding the first condition.

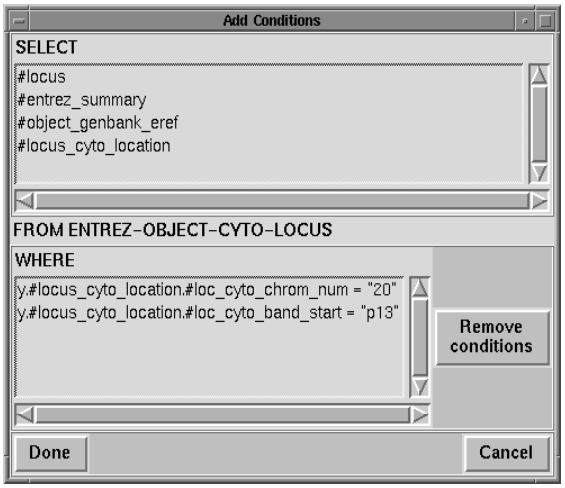

Figure 9: All conditions added.

```
    ( #locus: x.#locus,
      #entrez_summary: y.#entrez_summary,
      #object_genbank_eref: y.#object_genbank_eref,
      #locus_cyto_location: y.#locus_cyto_location ) |
    \x <- GDB-locus,
    \y <- ENTREZ-OBJECT-CYTO,
    JOIN_locus_genbank(x,y),
    y.#locus_cyto_location.#loc_cyto_band_start = "p13",
    y.#locus_cyto_location.#loc_cyto_chrom_num = "20"
};

! The invocation statement.

ENTREZ-OBJECT-CYTO-LOCUS;
```

The CPL engine comes with an optimizer which will automatically avoid materializing intermediate views as well as migrating joins, selections and projections on GDB to the remote server (see [26, 21, 3]). Therefore, QUICK only needs to generate queries that are clear and easy to understand.

Step 4. Getting the result. The generated query can be executed with the click of a button. The result is returned in a window as a nested relation. An example of the result is shown below.

```
{ ...
  (locus: (locus_name: "centromere protein B (80kD)",
           locus_symbol: "CENPB",
           locus_id: 58,
           locus_loc_summary: "20p13"),
  entrez_summary:
          (accession: "X55039",
```

```
          uid: 29860,
          title: "Human hCENP-B gene for centromere
                  autoantigen B (CENP-B)"),
  object_genbank_eref:
          (genbank_ref: "X55039",
          object_class_key: 1,
          object_genbank_id: 66951,
          object_id: 58),
  locus_cyto_location:
          (loc_cyto_band_start: "p13",
          loc_cyto_band_end: "",
          locus_id: 58,
          loc_cyto_chrom_num: "20"))
  ... }
```

In the above example, the user's effort of specifying a query is reduced to three steps, that is, specify user terms, select nodes or edges at the confirm of their on-line definitions, and specify additional conditions. Thesaurus and browsing capabilities help the user to quickly and easily recall exact terms. Any (generic) joins between nodes will be depicted graphically. Most predicates needed for join conditions will probably be already available or it can be easily incorporated into the graph. With QUICK, minimal knowledge of underlying databases and CPL are required because the process of transacting a graphical specification to a CPL query is automated.

Other than the features shown above, a great deal of efforts are made towards a user-friendly graphical editor. Here are a few of them. The top menu bar of Figure 4 contains options "File", "Query", "Edit", and "Help". QUICK allows to save the current session and retrieve a saved session for further editing work in a later time. The "File" menu contains commands for saving and retrieving a session file. After the user finishes the graphical specification of the query, he/she may choose to translate the specification to a named CPL query, or edit the query manually within QUICK, or run the query. These functions are contained in the "Query" menu. Named queries can be further manipulated using the set functions "Union", "Intersect" and "Difference" displayed on the right side of Figure 4. The menu "Edit" contains additional graphical functions. Among them, "Abstract" and "Unabstract" allow the user to collect a set of nodes together under one node or reverse the operation, thus enabling the user to hide away irrelevant portions for a while and put them back on the screen when required; "Zoom In" and Zoom Out" zoomed in or out a portion of a graph in a separate window; "Clear All Selection" cancels selection of nodes, edges, and predicates; "Undo" undoes the most recent graph operation. Nodes and edges can be moved around or deleted and the graph can be scaled in size to a level conformtable to the user. The "Help" option will bring up the help information on various topics in the form of hypertext links. Finally, most frequently used functions in "File", "Query", "Edit" menus are explicitly displayed below these menus so that only a single click is needed to use these functions.

5 The Coherence of Graphs

So far, the user selects nodes and edges without concerning the order of selection and the Query Composer processes selected edges in an order that may or may not correspond to the user selection order. A fundamental question is whether the query result depends on the order of processing edges. This question arises because predicates on edges are general, not necessarily natural join predicates. For example, we can define "is-blast-homolog-of" as a predicate which executes the NCBI BLAST[2] (National Center for Biotechnology Information Basic Local Alignment Search Tool) program to check whether a given gene is homologous[3] to another. For general predicates, the query result may depend on the processing order of edges, therefore, a subgraph alone does not specify a unique query. We propose the notion of "coherence" to capture the independence of the result with respect to the order of processing edges. First, we need a formal definition of "graphs".

Definition 5.1 (Graph) *A graph G consists of a finite set of nodes V and edges E, such that the following are satisfied:*

1. *A set is attached to each node which is a view defined on some relations. The set has the type $\{ (d_1 : r_1, \ldots, d_n : r_n) \}$, where each d_i is a view name and each r_i is the set of attribute names under the view d_i. The order of d_i's are not important. A* basic *node has the form $\{ (d : r) \}$.*

2. *An edge connects two nodes only.*

3. *An edge cannot connect two identical nodes. By renaming of nodes this can be satisfied.*

4. *An edge is visible if and only if both its end nodes are visible.*

5. *An edge has an orientation, that is, the left and right nodes of an edge are precisely defined so that any predicate on it is oriented with respect to this edge.*

6. *An edge has a predicate $p : s \times t \rightarrow B$ which takes the left and right nodes of type $\{s\}$ and $\{t\}$ as arguments and returns a boolean value.*

7. *Each view name d_i appears at most once in G. This property can always be satisfied by renaming of view names.*

Two nodes may have multiple edges connecting them. The user can select more than one edge between two nodes and specify a relationship among the selected edges, i.e., conjunctively, disjunctively, ... etc. In our implementation, however, there is at most one edge between any two nodes so as to make the graph look neater. However, this causes no loss of generality because predicates associated with multiple edges can be represented by a predicate associated with one edge. For instance, suppose that between nodes A and B there are three edges that have

[2]BLAST is a heuristic search algorithm employed by BLAST programs. The BLAST programs are tailored for sequence similarity searching - for example to identify homologs to a query sequence.

[3]Two sequences are homologous if they are 'similar'.

predicates p_1, p_2, p_3, respectively, and suppose that the user selects $(p_1 \wedge p_2) \vee p_3$ relationship among these predicates. Then the three edges between A and B can be replaced with one edge that has predicate $(p_1 \wedge p_2) \vee p_3$. Edge is associated with a list of predicates and only the selected predicate is used for the edge.

Graph Transformations

Assume that a subgraph and associated predicates are selected by the user. The Query Composer generates the query based on the following graph transformation. At each step, one edge in the selected subgraph is reduced to one node. For example, in Figure 10 the reduction on edge connecting $X1$ and $X2$ will result in a new node N being formed. The set attached to N is the one that satisfies the selected predicate $P12$ on edge $(X1, X2)$. The edges ($X3$,$X2$) and ($X1$,$X3$) now become ($X3$,N) and (N,$X3$), respectively. Figure 11 shows how the properties of a graph in Definition 5.1 are preserved by one step transformation. Assume that $V11$ is the view attached to node $X1$ with attributes $A1$ to Ai. $V21$ and $V22$ are views attached to node $X2$ with attributes $B1$ to Bj and $C1$ to Ck respectively. Views $V11, V21$ and $V22$ are attached to the new node N. The attribute names remain attached to the respective view names. Thus, the type of the set attached to N remains in the form { ($d_1 : r_1$, ..., $d_n : r_n$) }, where each d_i is a view name and each r_i is the set of attribute names under the view d_i. Therefore, the transformation preserves all properties of a graph required in Definition 5.1 and the result is also a graph.

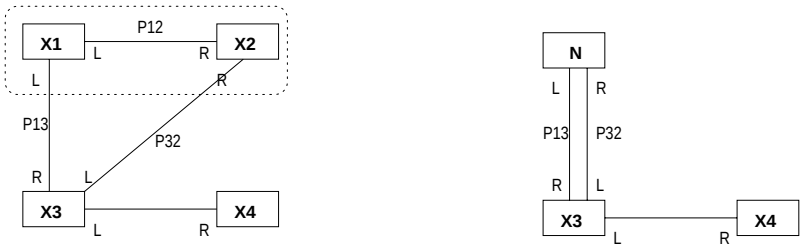

Figure 10: One transformation. L and R denote left and right ends of an edge

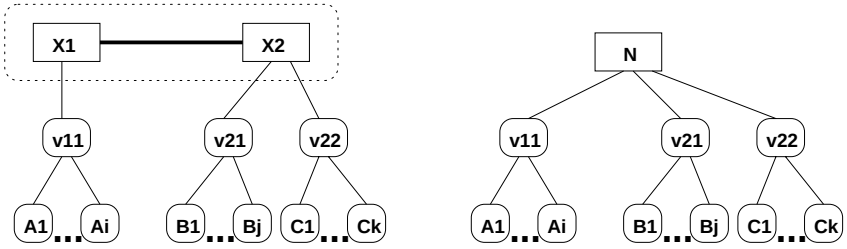

Figure 11: The new node type after one transformation

Definition 5.2 (Transformation) *Given a graph $G(V, E)$ and an edge $(X1, X2)$ in G, G is* transformed *into a graph $G'(V', E')$ with respect to $(X1, X2)$ if*

1. *The two nodes are discarded and a new node is created, that is,*

$$V' = V \setminus \{X1, X2\} \cup \{N\},$$

where N is the new node.

2. *The set attached to N is*

$$\{\ (s,\ t) \mid s \in S1,\ t \in S2,\ P_{12}(s,t)\ \},$$

where $S1$ is the set attached to $X1$ and its type is $\{\ (\ d_1 : r_1,\ ...,\ d_n : r_n\ )\ \}$, $S2$ is the set attached to $X2$ and its type is $\{\ (\ d_{n+1} : r_{n+1},\ ...,\ d_m : r_m\ )\ \}$, P_{12} is the predicate selected for edge $(X1, X2)$.

3. *The set of new edges E' are obtained from E as follows. Remove edges connecting to $X1$ or $X2$. For each edge that has only one end connecting to $X1$ or $X2$ in E, add corresponding edge which connects to the new node N instead of $X1$ or $X2$. That is,*

$$\begin{aligned} E' = E \setminus \{\ &(Xi,\ Xj) \mid i \in \{\ 1,\ 2\ \} \text{ or } j \in \{\ 1,\ 2\ \}\ \} \cup \\ &\{\ (N,\ Xi) \mid (X1,\ Xi) \in E \text{ or } (X2,\ Xi) \in E \text{ and } i \notin \{\ 1,\ 2\ \}\ \} \cup \\ &\{\ (Xj,\ N) \mid (Xj,\ X1) \in E \text{ or } (Xj,\ X2) \in E \text{ and } i \notin \{\ 1,\ 2\ \}\ \} \end{aligned}$$

4. *Predicates attached to edges remain unchanged.*

Let Φ denote the above transformer. We denote the above graph $G'(V', E')$ by $\Phi_G(X1, X2)$.

Theorem 5.1 *The graph transformer Φ defined above is commutative and associative. That is,*

1. $\Phi_G(X1, X2) = \Phi_G(X2, X1)$, *and*

2. $\Phi_{\Phi_G(X1,X2)}(X12, X3) = \Phi_{\Phi_G(X2,X3)}(X1, X23)$, *where $X12$ is the new node that results from $\Phi_G(X1, X2)$ and $X23$ is the new node that results from $\Phi_G(X2, X3)$.*

We omit the straightforward proof from the definition of transformation. Let $\rightarrow_\Phi$ denote one step transformation and $\rightarrow_\Phi^*$ denote zero or more step transformation.

Definition 5.3 (Coherency) *A graph G is* coherent *with respect to a graph transformer Φ if whenever $G \rightarrow_\Phi G_1$ and $G \rightarrow_\Phi G_2$ then there exists G' such that $G_1 \rightarrow_\Phi^* G'$ and $G_2 \rightarrow_\Phi^* G'$.*

The following corollary follows immediately from this definition.

Corollary 5.1 *If a graph G is coherent with respect to a graph transformer, the attached set and type for the final graph containing only one node are uniquely defined. In other words, the final result is independent of the order of applying transformations.*

Luckily, every graph as defined in Definition 5.1 is coherent with respect to the above graph transformer, as shown below.

Theorem 5.2 *A graph is coherent with respect to the above graph transformer* Φ.

The proof of this result proceeds by a case analysis. Suppose that the original graph is G and it can be transformed to $G1$ and $G2$ in one step. There are three cases to consider. The first case is when $G1 = \Phi_G(X1, X2)$ and $G2 = \Phi_G(X2, X1)$. Then $G1 = G2$ follows by commutativity of Φ. The second case is when $G1 = \Phi_G(X1, X2)$ and $G2 = \Phi_G(X2, X3)$. Let $G1' = \Phi_{G1}(X12, X3)$, where $X12$ is the new node in $G1$. Let $G2' = \Phi_{G2}(X1, X23)$, where $X23$ is the new node in $G2$. Then $G1' = G2'$ follows by associativity of Φ. The third and final case is when $G1 = \Phi_G(X1, X2)$ and $G2 = \Phi(X3, X4)$, with $X1$, $X2$, $X3$, and $X4$ distinct. Let $G1' = \Phi_{G1}(X3, X4)$ and $G2' = \Phi_{G2}(X1, X2)$. It is straightforward to verify using the definition of Φ that $G1' = G2'$. This completes the case analysis.

Since a graph is finite, regardless of the order of processing edges, for each connected component of n nodes there will be a final resulting node whose attached set and type are defined below

Attached set:

$$\{\ (s_1, ..., s_n) \mid s_1 \in S1, ..., s_n \in Sn, P_{ij}(s_i, s_j) = TRUE, 1 \leq i < j \leq n\ \},$$

where Si is the set attached to node i and $P_{ij}(s_i, s_j)$ is TRUE if there are no edges between nodes i and j, otherwise $P_{ij}(s_i, s_j)$ is the predicate selected between nodes i and j taking into account their orientations.

Attached type:

$$\{\ (\ d_1^1 : r_1^1, ..., d_1^{m_1} : r_1^{m_1}, d_2^1 : r_2^1, ..., d_2^{m_2} : r_2^{m_2}, ..., d_n^1 : r_n^1, ..., d_n^{m_n} : r_n^{m_n}\ )\ \},$$

where d_i^j is a view attached to node i and r_i^j is the set of attributes attached to the view d_i^j. Since the result set is uniquely defined, it does not matter which edge is chosen first and processed next. See Figure 12 for an illustration.

Two implications follow from Corollary 5.1 and Theorem 5.2: First, the query result of a graph is unique and thus well-defined. Second, the query optimizer of CPLTCL engine has the freedom to rewrite the join order of a CPL query generated by QUICK for the purpose of optimizing query processing. This lays down a sound theoretical basis for our interface design and query optimization.

6 Conclusion

A preliminary prototype interface to multiple autonomous heterogeneous databases, called QUICK, is implemented. QUICK translates a graphical specification into well-formed CPL primitives that can be run on the CPL-Kleisli system. Graphical functions are supported to allow the generated query to be enhanced or modified to suit specific needs. Though the use of QUICK is based on an example from the Human Genome Project, the interface is generic in the sense that for a new application running on CPL, only new Meta Dictionary and Thesaurus Dictionary need to be created; for a new application running on a multidatabase language rather other CPL, the Query Composer also need to be substituted. The rest of the system remains unchanged. Clearly, the study on such applications will have general implications on a total solution.

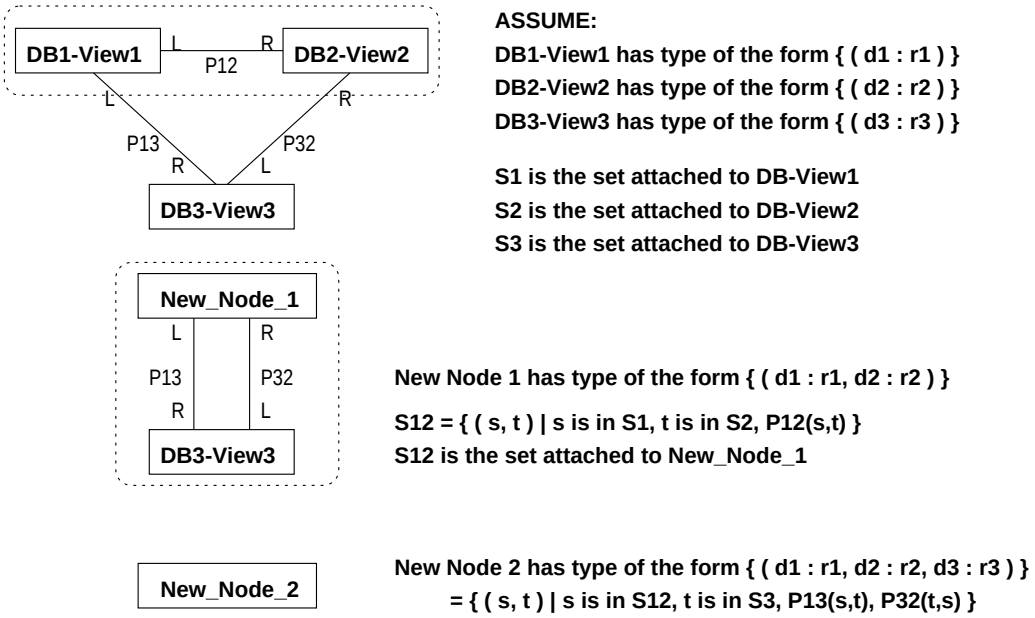

Figure 12: An illustration of two step transformation

Like other interface work, QUICK does not have much formal theory due to the nature of this type of work. However, several principles of designing a user-friendly interface have been addressed, that is, no requirement of the user to know the exact database terms, no requirement of the user to learn a new multidatabase language, a graphical representation is generally preferred to a textual one, etc. These principles serve the foundation for the interface design.

Many experiments and runs show that QUICK has been very efficient in helping the user to formulate a correct query quickly. To understand this better, think what a naive user of a multidatabase may do without QUICK. First, the user has to find the exact database terms and databases for the data items interesting to him. In the case of the biological data sources (in particular, GDB version 5.5), this means that the user has to search through over 300 pages of schema documents, which contains close to 400 tables and approximately 1300 database terms, and follow links among tables. In most cases the search is very slow because most multidatabase users are occasional users and are not familiar with structures of remote databases. The second painful step is to learn a multidatabase language for specifying the query. This language is usually different from the user's home database languages. After the query program is written and submitted for compilation, the user realizes there are a lot of syntax errors in his/her program. So the third step is to debug the query program. Lastly, the query is run. The user examines the result and finds the result is not something that he expects. Then the cycle of the four steps restarts, until the user is satisfied with the result.

On the other hand, QUICK helps the user in every of the above four steps. With QUICK, the user does not need to look through the 300 pages document; instead, the user inputs the data items in his/her own familiar terms. The Thesaurus will do the search and find the exact database terms. Relevant database schemas are presented to the user in a graphical form

with detailed explanation available at a click of button. Then the user constructs the query through selecting nodes and edges, which are automatically translated into CPL programs. The automatic translation is important because the user does not have to learn a new language and deal with syntax errors. Restarting the cycle means simply editing a saved file graphically.

In all experiments with QUICK performed, the gain over the conditional textual query specification is obvious. For example, a user with some domain knowledge in the biological application needs hours, sometime days, to search for the data sources, learn the CPL language and specify a query correctly, whereas the same user needs less than half an hour to specify the same query. The key point is that multidatabases are usually very large and diverse in format and location. For such applications, an on-line graphical interface with an intelligent search mechanism (such as the Thesaurus) is probably the only solution.

As the future work, we plan to conduct more intensive and broad experiments with QUICK on large and real world applications. We also intend to study and extend the expressiveness of queries that can be formulated in QUICK. The ultimate goal of QUICK is to provide a user-friendly interface for querying large heterogeneous databases such as genome databases.

References

[1] John Boyle, John E. Fothergill, and Peter M.D. Gray. Design of a 3D User Interface to a Database. In *Proceedings of the 2nd International Workshop on Interfaces to Database Systems*, pages 127–142, Lancaster University, July 1994.

[2] M.W. Bright, A.R. Hurson, and Simin H. Pakzad. A Taxonomy and Current Issues in Multidatabase Systems. *Computer*, 25(3):50–60, March 1992.

[3] Peter Buneman, Susan Davidson, Kyle Hart, Chris Overton, and Limsoon Wong. A Data Transformation System for Biological Data Sources. In *Proceedings of 21st International Conference on Very Large Data Bases*, pages 158–169, Zurich, Switzerland, August 1995.

[4] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[5] Hock Chuan Chan. Graphical Entity Relationship Query Languages. Technical Report TRA1/94, Department of Information Systems and Computer Science, National University of Singapore, Kent Ridge, Singapore 0511, Jan 1994.

[6] The Collection Programming Language Home Page
URL=http://www.cis.upenn.edu/~khart/cpl/cpl.html.

[7] Susan Davidson, Chris Overton, and Peter Buneman. Challenges in Integrating Biological Data Sources. *Journal of Computational Biology*, 2(4), 1995. In press.

[8] Entrez Browser
URL=http://www3.ncbi.nlm.nih.gov/Entrez/index.html.

[9] The Genome Database
URL=http://gdbwww.gdb.org/.

[10] GDB Quick Search
URL=http://gdbwww.gdb.org/gdb/shortcuts.html.

[11] GDB Query Forms for search Public Data
URL=http://gdbwww.gdb.org/gdb/queryPublic.html.

[12] GDB Query Forms for search Public Data
URL=http://gdbwww.gdb.org/gdb/queryPrivate.html.

[13] The National Center for Biotechnology Information
URL=http://www.ncbi.nlm.nih.gov/.

[14] Nathan Goodman. Research Problems in Genome Databases. In *PODS*, May 1995.

[15] J. Grant, W. Litwin, N. Roussopoulos, and T. Sellis. Query Languages for Relational Multidatabases. *The Int'l Journal on Very Large Data Bases*, 2(2):153–171, April 1993.

[16] Yong S. Jun and Suk I. Yoo. A Graph-based Graphical User Interface for Object-Oriented Databases. In *Proc. of the 1994 Int'l Conf. on Object-Oriented Information System. of Data*, pages 238–251, London, England, 1994.

[17] The Kleisli Project
URL=http://sdmc.iss.nus.sg/kleisli/kleisli/kleisli.html.

[18] R. Krishnamurthy, W. Litwin, and W. Kent. Interoperability of Heterogeneous Databases with Schematic Discrepancies. In *Proc. First Int'l Workshop. on Interoperability in Multidatabase Systems*, pages 144–151, 1991.

[19] W. Litwin and A. Abdellatif. An Overview of the Multi-Database Manipulation Language MDSL. *Proc. of the IEEE*, 75(5):621–632, May 1987.

[20] Simon Monk. A Graphical User Interface for Schema Evolution in an Object-Oriented Database. In *Proc. of the 2nd Int'l Workshop on Interfaces to Database Systems.*, pages 185–196, Lancaster University, 1994.

[21] Anne Ngu, Lingling Yan, and Limsoon Wong. Heterogeneous Query Optimization using Maximal Subqueries. In *Proceedings of 3rd International Symposium on Database Systems for Advanced Applications*, pages 413–420, Taejon, Korea, April 1993.

[22] Martin H. Rapley and Jessie B. Kennedy. Three Dimensional Interface for an Object Oriented Database. In *Proc. of the 2nd Int'l Workshop on Interfaces to Database Systems.*, pages 143–167, Lancaster University, 1994.

[23] M. Schneider and C. Trepied. Extensions for the graphical query language CANDID. In *Proc. of IFIP 2nd Working Conf. on Visual Database Systems*, pages 185–199, North-Holland, Netherlands, 1991.

[24] Stefano Spaccapietra and Zahir Tari. Super : A comprehensive approach to Database Visual Interfaces. In *Proc. of IFIP 2nd Working Conf. on Visual Database Systems*, pages 365–380, North-Holland, Netherlands, 1991.

[25] Elizabeth R. Towell and William D. Haseman. Implementation of an Interface to Multiple Databases. *Journal of Database Management*, pages 13–21, Spring 1995.

[26] Limsoon Wong. Normal Forms and Conservative Properties of Query Languages over Collection Types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 26–36, Washington, D. C., May 1993.