Project Report

Minirel[1] frontend and utilities

with a QBE[2] ->SQL[3] Parser

Advisor
Michael Carey and **Raghu Ramakrishnan**
Department of Computer Science
University of Wisconsin, Madison

submitted by

Daniel M. Wong
Department of Computer Science
University of Wisconsin, Madison

May 12, 1995

1 Minirel is a relational DBMS developed by database courses at University of WIsconsin, Madison
2 QBE stands for Query by Example, and is developed by IBM at Yorktown Heights for QMS.
3 SQL evolves from a language named SEQUEL, and is developed by IBM at San Jose for System R.

Contents

I. Introduction

Database Management Systems have been among the most active research areas in computer science academia. Regrettably, there has been little work done on frontend systems. Early implementations of graphical user interfaces(GUI) suffered from lack of both development tools, and cross-platform unified system support. One of the earliest graphical query languages is IBM's QBE, developed for their QMS system. Using special keys as command inputs certainly would seem a little obscure to many database users nowadays. Recent popularity of C++, Tcl/Tk and X Window System has allowed us to take advantage of this open environment to easily build a cross-platform GUI. Therefore, in this project, we intended to study the design and implementation of QBE for IBM's QMS developed in the early 80's, and to experiment with translating the QBE queries into SQL statements using these new tools.

The QBE -> SQL translator prototype is built for Minirel, in which we are responsible for the frontend tools and utilities. The goal of the project is to assemble together a multiuser DBMS code base for future database course projects. Realizing that GUI has become an indispensable part of complete database system offerings, we decided to build a GUI with QBE as the query frontend. For various reason of conveniences, QBE queries are translated into equivalent SQL statements for submitting to the Minirel DBMS for processing.

II. Project overview

Our part of the Minirel project mainly focuses on the implementation of Minirel's frontend system, in particular, a presentation manager and a query manager. Presentation manager manages all of frontend's GUI while the query manager translates the entered QBE queries into equivalent SQL statements. Basically, presentation manager accepts user inputs and requests, and then responds by invoking appropriate commands to process them. Query manager responds upon query request to translate QBE queries into SQL statements. The result is returned to the presentation manager. Figure 1 shows an overview of the structure of Minirel's frontend system.

Overview of Minirel frontend system

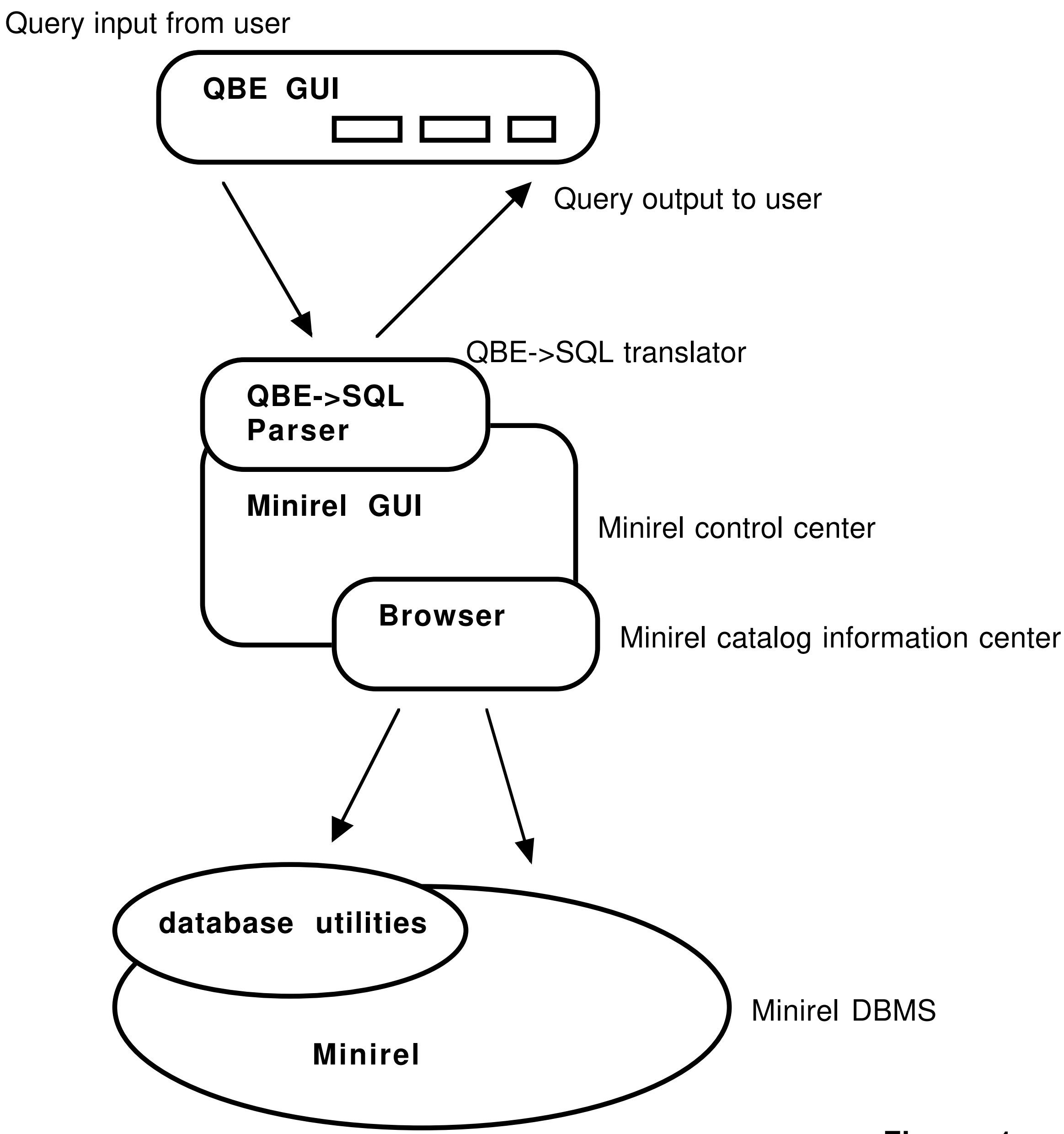


Figure 1.

The GUI[4] and the query translation engine are entirely coded in Tcl/Tk, while the utilities and some of the underlying functionalities of the browser are written in C++. Syntax of the query semantics is based upon the original QBE. The entire translation engine is structured as an independent application used by Minirel's frontend system. In terms of size, the Tcl/Tk code of the frontend, with proper documentation, consists of ~5.5k lines of Tcl/Tk code, and the C++ code for the Tcl/Tk/C++ integration and utility functions is ~1.5k. The breakdown of the Tcl/Tk code is approximately ~2k lines for parser, ~1k lines for the communication modules and GUI, and ~2k lines of Tk utility library borrowed from a Tk public archive.

III. Minirel GUI and utilities

A. Overview

Minirel's GUI is the sole mean for users to perform queries. Our goal on this part is to provide a friendly user environment. In addition to the basic menu to access the query manager, the GUI includes a database browser in which users can navigate the available catalog information, i.e. information on specific databases, relations, or attributes, by playing point and click. Database utilities can also be accessed through the GUI as well, though they are actually structured as separate UNIX command line programs executable from any shell.[5]

B. External interface

From users' point of view, the GUI itself is the interface. Figure 2 is a screen capture of the Minirel's main GUI. Users inputs by typing over the keyboard or drags through the menus. From the Minirel DBMS's point of view, it is not aware of any external interface provided by the GUI. This is because the Minirel process is indeed run as a X window application[6] . The internal event loop within the application detects the user actions and invoke the associated commands. Minirel DBMS plays a passive role. Even for the utilities, the DBMS only takes SQL statements and processes them.

---

4 As a brief note, in QBE, users query by giving examples of the values the resulting tuple(s) contain.

5 Indeed, the GUI runs the utilities by spawning it as another UNIX process.

6 Details of the Minirel process structure should probably appear in the report by system group.

Never in a case any GUI modules are being invoked by the Minirel DBMS[7].

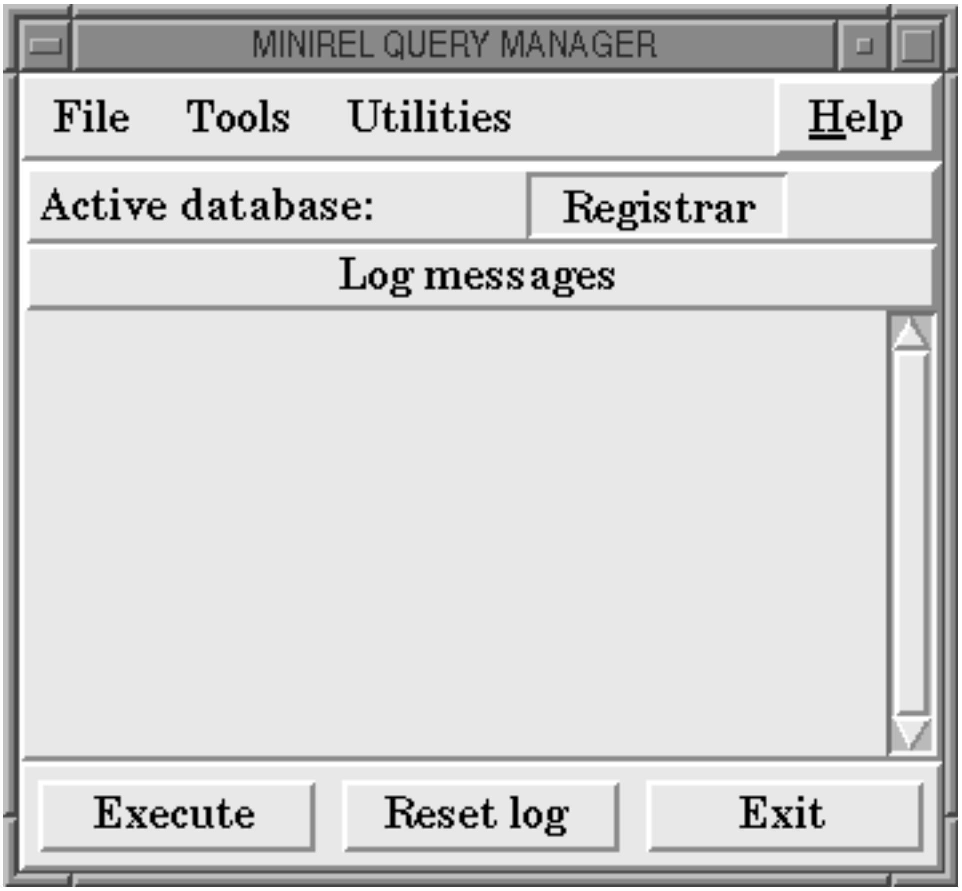


Figure 2. Main GUI

Internally, each individual component of the frontend system does have its own public interface. Notably, each set of GUI and the integration layer do provide a fairly complete set of public interfaces. The following

[7] In principle, when the application is registering with the X server, it does need to provide an interface for the process to initialize the GUI. In this case, in addition to the Tcl/Tk build-in Tcl_Init () functions, we only have a single function, Init_GUI (), which deals with initializing the GUI with the integration layer.

sections attempts to go through each of them:

The design of Minirel's GUI evolves as Tcl/Tk skill matures and application environment changes. Eventually, it has been decided that minimal amount of information should be displayed to avoid user distraction, and to simplify all the synchronization between display information and internal representation of information, i.e. name of the database in use, or name of the relation opened. Altogether, there are four major GUI's, main GUI, browser GUI, utility GUI, and QBE GUI, designed and implemented for the corresponding components of the frontend system.

The layout of the main window's GUI is very straight forward. Referring back to Figure 2, our screen capture of the GUI, the main window is labeled 'Minirel Query Manager.' Buttons have self-explanatory labels on them. Along the top of the main window there is a menu bar with three menu options: 'File','Tool,' and 'Utility.' Each of these menu options represent a functional group of the frontend system. Opening and closing of a database are commands universally[8] placed under 'File.' Query editor and condition box are tools for doing query, thus the commands for enabling them are under 'Tool.' 'Utilities' submenu gives access to all the database utilities that Minirel supports, including creating a database, deleting a database, creating a relation, deleting a relation, adding an index, dropping an index, inserting a record, deleting a record, and bulkloading. Other than the buttons and the menu bar, there is another label indicating the current active database, and a log window displaying the messages[9] generated by Minirel as a result of any user issued commands.

Browser GUI presents to the users all available catalog information of a database. A sample state of browser GUI is shown in Figure 3. It contains three major components, a database listbox, a relation listbox, and an attribute listbox. Depending on user selections, each listbox dynamically updates its list of entries. The small index-type menu bar at the bottom left of the browser allows the user to tell what is the index type built for a specific attribute.

Utility GUI basically builds on top of listbox components of the browser. Instead of running as a tool for browsing catalog information, as is so in the browser, listboxes are used as tools for input to specify

[8] In almost all the commercial GUI application, this is true.

[9] Results of queries are not included here.

target for their utility operation, e.g. which database to delete when calling dbdelete. Customized utility GUI is provided for each specific utilities as needed. Figure 4 is a sample of the customized utility GUI when adding a relation.

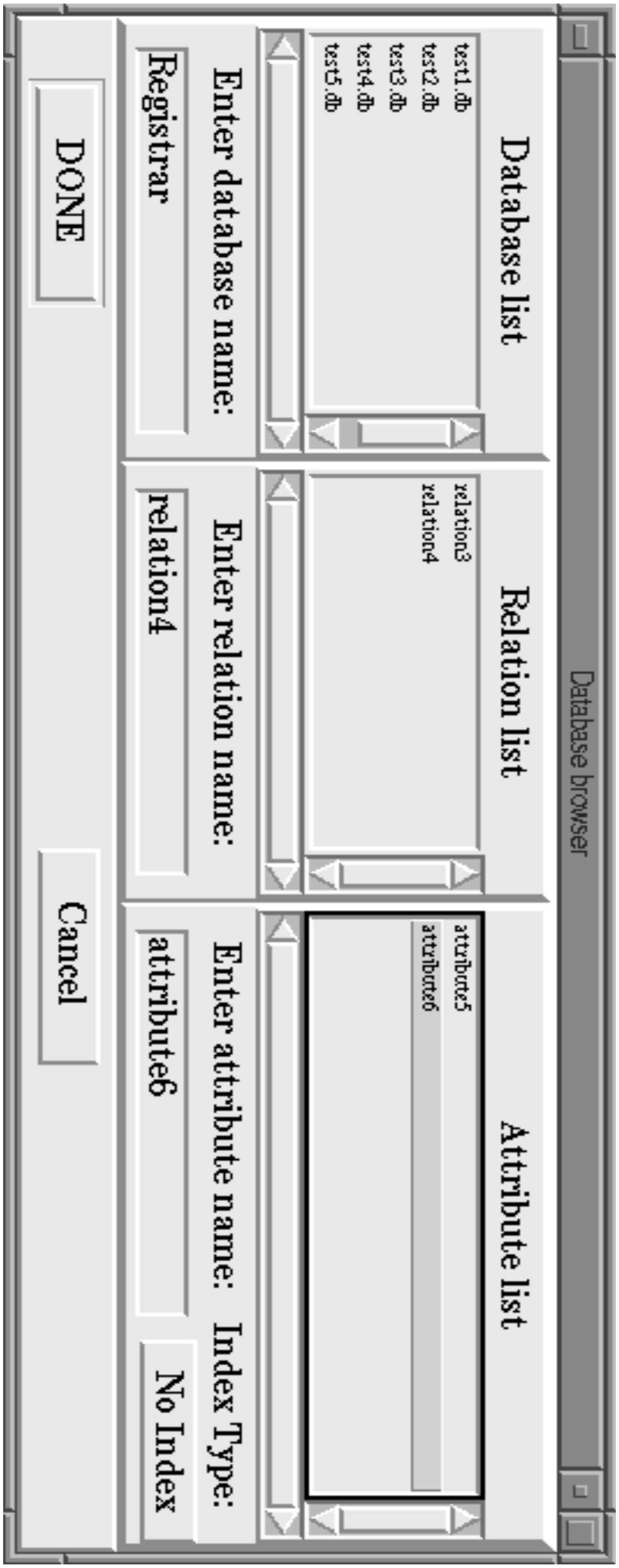


Figure 3. Browser GUI

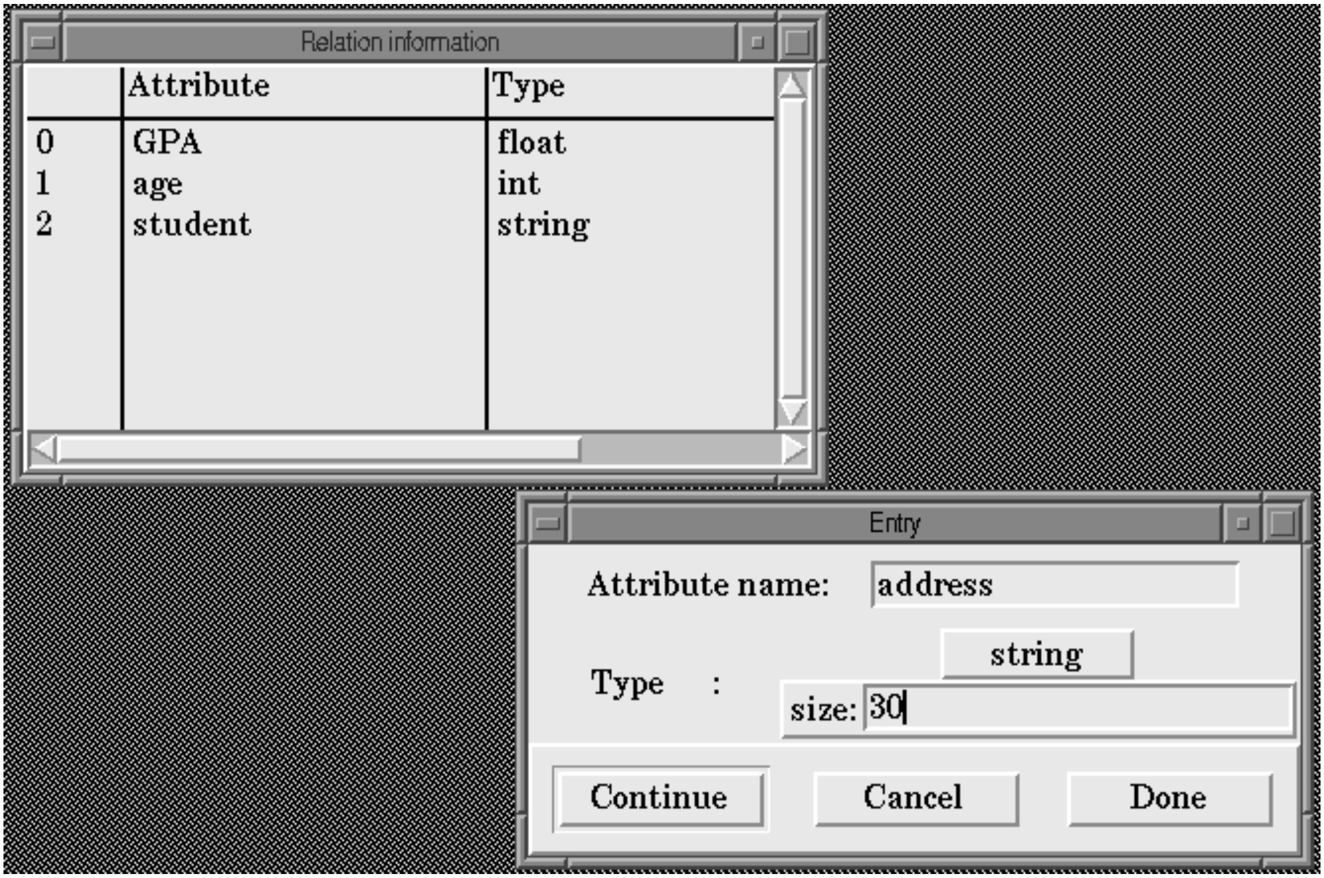


Figure 4. Utility GUI for adding a relation

Simple QBE GUI is provided to the query manager for entering QBE queries. Largely following the specification of QMF, in each relation table there are the standard command column, attribute columns, optional condition box, and optional unnamed columns. Less obvious features would be arbitrary number of row entries and unnamed columns. In addition to the mentioned must-have QBE features, there are also enhancements over the original QBE. For instance, users can now configure their table skeletons through point and click, and arbitrary attributes in any relation tables can be turned on or off. Most importantly, we support a query editor. Before the SQL statement is submitted to the DBMS, it can be edited manually in anyway. Figure 5 is shows a sample of QBE GUI with condition box and query editor enabled. The details on the merit of this query editor will be discussed later.

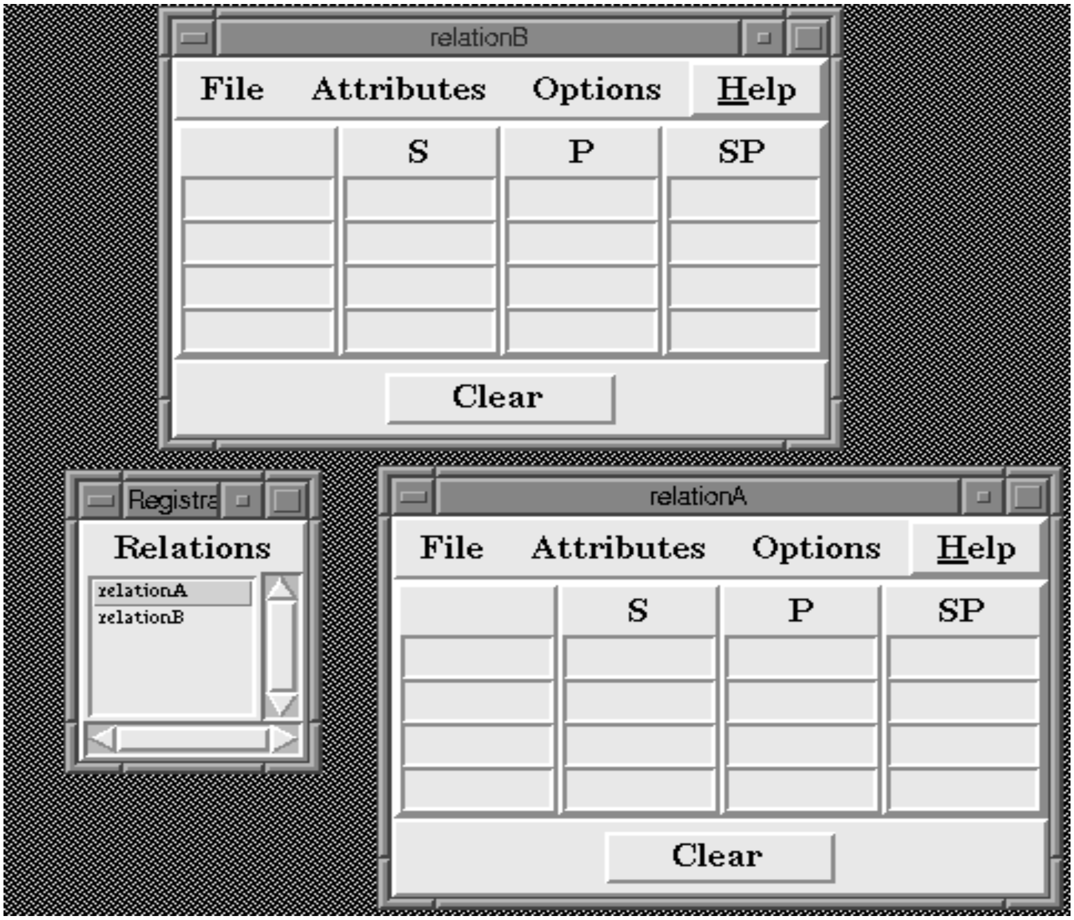


Figure 5. QBE GUI

Aside from the GUI, frontend system work also includes building of an integration layer for Tcl/Tk/C++, and database utilities. Though they only require a moderate amount of implementation and not much room for design, for the sake of completeness, we will briefly describe in general the design of all of them.

Integration of Tcl/Tk/C++ code was fairly straight forward, once the template file was located. A set of initialization routines were written to register new Tcl commands to the Tcl interpreter. A new class Browser is built to manage the catalogs in order to handle the necessary information request from the browser, or specifically from each of the listboxes. In short, there is a set of new Tcl commands built for the routines in class Browser so that the Tcl scripts can use them. Figure 6 and 7 are the header file used for the class Browser and the header file for declaring the new Tcl routines, respectively. It is very similar to the class maincatalog.[10] At time of implementation, class maincatalog was not designed yet, thus class Browser was implemented to provide a similar set of functionalities.

There are two main routines for integrating Minirel DBMS and GUI layer into the extended Tcl/Tk libraries, which contain all necessary commands to interpret the Tcl/Tk scripts. These two routines, Init_Minirel() and Init_GUI(), are defined in gui.h, as shown in Figure 7. They contain the necessary routines to initialize the entire DBMS and frontend, respectively. Note that Init_Minirel() is used by database utilities as well because Minirel database utilities run on separate UNIX address space.

As mentioned, Minirel database utilities are standalone UNIX executables.[11] They can be viewed as a special instance of the Minirel DBMS, executing on one single command only. They are needed because current implementation of parser and planner do not yet support DDL commands. They can run from UNIX shell or the Minirel frontend system. There is really no external interface from these utilities. Syntax for using these utilities are in the appendix section.

---

10 See system architecture group for details on this class.

11 Minirel processes do share some data structures, such as share buffer and share memory. For details, refer to the report by system architecture group.

```
# include "maincatalog.h"
# include "util.h"
# include "db.h"

#ifndef BROWSER_H
#define BROWSER_H

#define WORKINGDIR "/afs/cs.wisc.edu/u/grad/dmw/cs784/proj"

// Browser class only exists as an instance of a utility operation
class Browser {

 private:
 // internal bookkeeping
 // char DBname[MAXLENGTH];
 // char RELname[MAXLENGTH];
 // char ATTRname[MAXLENGTH];

 // some state info.
 Error   error;
 DB*     OpenedDB ;
 RelCatalog* OpenedRelCat ;
 AttrCatalog* OpenedAttrCat ;
 IndexCatalog* OpenedIndexCat ;

 public:
 // Open DB, RelCat, and Attr Cat
 Browser();

 // clean up all opened stuffs, if any
 ~Browser();

 // return list of databases available
 Status GetDBList(char* dblist);

 // return list of relations in the opened db
 Status GetDBInfo(char* dbname, char** rellist, int &dbcnt, int MAXCOUNT);

 // return list of attr. in the opened relation
 Status GetRELInfo(char* attrname, char** attrlist, int &relcnt, int MAXCOUNT);

 // return specific info on an attr.
 Status GetATTRInfo(char* relname, char* attrname,int &cnt,IndexType *indexrec);

 // prints record in an opened relation
 Status Print();

 // return pointer to current DB, relations, attributes, respectively
 DB* CurrentDB() { return OpenedDB;}
 RelCatalog* CurrentREL() { return OpenedRelCat;}
 AttrCatalog* CurrentATTR() { return OpenedAttrCat; }
 IndexCatalog* CurrentINDEX() { return OpenedIndexCat;}

 Status ResetStates();
};
#endif BROWSER_H
```

Figure 6. Class Browser in browser.h

```
#include "tclExtend.h"
#include "catalog.h"
#include "util.h"
#include "browser.h"
#include "db.h"
#include "error.h"

#ifndef GUI_H
#define GUI_H

// initialize browser
int
Init_Browser(ClientData  clientdata, Tcl_Interp *interp,int argc,char* argv[]);

// clean up browser
int
Close_Browser(ClientData  clientdata, Tcl_Interp *interp,int argc,char* argv[]);

// set of tclcreate_cmd extensions to tcl
int MR_CreateCmd(Tcl_Interp* interp) ;

// calling misc functions to init GUI
int Init_GUI(Tcl_Interp* interp) ;

// init all managers
Status Init_Minirel();

// open the db
int DBopen(ClientData  clientdata, Tcl_Interp *interp, int argc,char* argv[]);

// close the db
int DBclose(ClientData  clientdata, Tcl_Interp *interp, int argc,char* argv[]);

int ExecuteQuery(ClientData  clientdata, Tcl_Interp *interp,
int argc,char* argv[]);

int getRELinfo(ClientData  clientdata, Tcl_Interp *interp,
int argc,char* argv[]);

int getDBinfo(ClientData  clientdata, Tcl_Interp *interp,
int argc,char* argv[]);

int getATTRinfo(ClientData  clientdata, Tcl_Interp *interp,
int argc,char* argv[]);

// get relation info without browser
int QRELinfo(ClientData  clientdata, Tcl_Interp *interp,int argc,char* argv[]);

// define some global variables
#define GLOBAL_DB       db
#define GLOBAL_ERROR    error
#define GLOBAL_RELCAT   relCat
#define GLOBAL_ATTRCAT  attrCat
#define GLOBAL_INDEXCAT indCat

#endif GUI_H
```

Figure 7. New Tcl commands in gui.h

C. Internal design

For the GUI, given that Tk is used as the development tool, we feel obligated to explore the features of Tcl/Tk to manage the widgets. For simplicity, within each component of the GUI there is a set of simple global variables that represent the internal status of the frontend. For example, all shared variables inside the main window, such as the current directory, the name of database in use, and the status of the browser are all maintained as global instances. There are some other global information maintained by the translator, and it will be described in the corresponding section. The GUI really does not require any special data structure, since no processing is done. The translator does have major data structures and it will be discussed in the corresponding section also.

Browser GUI requires a little more logic to maintain the internal representation of the state of the database. For example, users are allowed to navigate between relations and attributes in any order. Since only one database can be opened at a time, care must be taken to refresh information when switching between databases.[12] Listboxes in browser are linked[13] together in the sense that they can directly access each other's information

As a note, some of the display blocks are built on the fly, such as the toggling of attributes in QBE GUI, and the entry for string sizes and hash sizes. Those features require the GUI to dynamically restructure the display arrangement on the fly. Fortunately, by redrawing the corresponding frame, Tk supports this reconfiguration feature. Just that more state variables, in the form of associative arrays, are maintained to ensure consistency of state information.

About the database utilities, each of them is structuraly a separate Minirel process. Their only difference with the Minirel DBMS that the frontend runs on is that they do not do queries. Each utility basically perform three actions: invokes Init_Minirel () to initialize the different managers in Minirel, instantiates DB and maincatalog class, and makes the appropriate function call supported by the maincatalog. There are times, such as when adding relations, the utilities have to do packaging. But then

[12] This feature is disabled later, for simplification of the running environment.

[13] The location of these listboxes is static, since at most one instance of browser is allowed for each Minirel process.

that’s it. Relatively, the work for the frontend GUI and catalog interface were more substantial then the implementation.

D. Retrospective

As seen from the discussion above, much of the work involved in implementing the GUI was the layout of the external interface. The entire frontend was original structured to allow multi-database operations. There exists a small amount of careful and somewhat tricky coding for ensuring consistency. It is unfortunate that some of those features are disabled now due to realization of some design constrains. On the other hand, it is readily extensible. Code for the disabled features will remain in the code base for reference. There is a fair amount of effort spent on the external interface of the catalogs. As a result, the first version of the catalog interface did not become available until a week before the final exam. took place. The good news of building a maincatalog class came a day after the new catalog interface first became available in a public directory. And that simplifies the interaction with the catalog class significantly.[14]

In retrospect, the design of the frontend could be greatly improved by building a super database class that maintains all global instances in a Minirel process, similar to maincatalog class, which is basically a collection of macro’s to hide the catalog interface. Indeed, as mentioned before, the class Browser does a very similar function with maincatalog, except that it is specifically built to support browser operations only. We hesitated to design the super database class because doing so would require a major overhaul of how all lower layers access global information. Still, as it turns out, we could not escape the ill-fate of using global variables. The simple mismatch in DB instance (pointer or not pointer) has prevented us from compiling the utilities and test the integration[15] , while the next update of code deals with multi-user, which is probably not a good test case, because we have to wait until all integration is done before testing. We needed a two phase testing plan: one for single user mode, another for multi-user mode.

The work done could also be moderately reduced if the catalog interface has been frozen early. There was the concept that the catalog interface would be little different from the one in old Minirel, just little changes

[14] All database utilities use the maincatalog class then. But class Browser is kept.

[15] Unless we provide another version of all code that deals with the DB instance, and as the frontend, we deal a lot with this instance.

here and there. Thus work on coming up with the final catalog interface was not started until the GUI were in place and integration of Tcl/Tk/C++ needed the interface to implement the utilities. Actually this is partly due the probability that the old design was not very elegant.[16] The initial implementation of utilities was a therefore little messy, because of the need to deal with several catalogs, and it was based on the old minibase interface.

In current implementation, GUI of utilities is integration into the frontend's utilities. A better approach would be to separate the GUI of utilities altogether. That is, instead of spawning a UNIX process to take the arguments as inputs to perform the operation, we can spawn a GUI application and let the utility's GUI take care of taking inputs from users. This way the utilities are much more independent, and most importantly we can hide the restriction that one process can only work on a single database, because then we will have several Minirel processes running, each performing the necessary operation and then exits. By the same token, the browser should be made a separate process. Then we would not have to disable the database listbox. Too bad we learned of the restriction too late.

IV. QBE->SQL translator

A. Overview

As part of the frontend system, the query manager interacts with users for inputting queries. Upon request, the translator builds an entry list that contains all entry information, and then constructs the SQL statements by parsing these entry lists. Generated SQL statements will be forwarded to Minirel's DBMS for processing. Output of the query will then be returned to presentation manager for displaying to users.

There are several objectives in building this new translator: depending on the point of view, the ideal GUI raises users' productivity, provides an easily portable and maintainable frontend across all platforms, and incurs minimal resource overhead. Another less ideal perspective is to allow the programmer to gain hands on experience with the newly available tools and analyze the resulting improvement in developing applications, as well as experimenting with some parsing techniques on QBE queries. Our primary concern is to build a simple query processing frontend that stretches the

[16] All methods are public, which makes the programmer wonder how much work each method does.

ideas of QBE by experimenting some simple enhancements to the interface in order to improve user friendliness aspect and possibly some functionality, while maintaining the notion of KISS. The frontend system would allow users to query by simply supplying constrains on the relation table(s). Since QBE GUI has been briefly described, we will focus our attention on the translator, especially its architecture and its translation algorithm.

External interface

As a an application used by the Minirel frontend, the translator has a very simple set of API. Figure 8 is a list of Tcl/Tk modules that consists of the API to initialize several components of the translator:

```
# initialize the query editor
Init_QueryEditor {}

# initialize the condition box
Init_ConditionBox {}

# initialize the relation tables
Init_Tables {}

# translate the entered QBE queries into SQL and execute the query
ProcessQueries {}

# translate the entered QBE queries into SQL and display the query
ReloadQueries {}
```

Figure 8. QBE GUI interface modules

Once the QBE GUI is initialized with other GUI's when the Minirel is started. Whenever user opens a relation in a database, main GUI will invoke Init_Tables {}, which, as part of the initialization procedure, in turn starts the QBE GUI for that particular relation. Users can request the main GUI to enable the query editor or condition box at any time, and it will respond by calling Init_QueryEditor {} or Init_ConditionBox {}, respectively. Processing the input is invoked by calling ProcessQueries {} at the main GUI where the main component of the translator is located. ReloadQueries {}, in which translated SQL statements is only returned but not executed, is only invoked by query editor.

C. Design layout

Current implementation of the parser, because of the lack of tools, or actually the difficulties in error detection, presents a best effort parser. In other words, valid QBE's are parsed into corresponding SQL statements correctly (if it is not on the unsupported list/in doubt list), and limited sets of invalid QBE's will be parsed into a valid SQL statement as well. For example, if P. is specified on the command column, and I. and D. are specified within the same row in a relation table, but on the relation attributes, the parser will simply ignore these embedded invalid commands and generate a project SQL statement as if these I.'s and D.'s are never entered. Invalid commands embedded within the attribute columns are not detected as well. The translator works by looking for specific information in the entries, rather than understanding the them. Later in the detailed description of translator design these unusual characteristics will become clear.

D. Translator architecture

The translator is distributed in the sense that it consists of three major components: a query editor, local parsers, and a base. Figure 10 gives an overview of how these components are linked together.

Query editor

The function of the query editor is to allow the user to edit the SQL statement, possibly generated from the base according to the information gathered from the local parsers. Upon user request, the edited query is submitted to the DBMS for execution. Therefore it is optionally the last stage of the translation process. Note that there is no checking done on the edited query because building another SQL parser to check the query would be quite cumbersome and also repeating the Minirel parser's task. No state information is kept, except the address of the base where requests are forwarded to.

Local parser

Each relation table is handled by a separate UNIX process. At each relation table there is a local parser. Depending on the query command found, the local parser preprocesses the entries and format them in a way that the corresponding query command's processing modules at the base can easily utilize the information. Specifications of the data format for each query command is given in the appendix. State information maintained includes table skeleton configuration parameters (i.e. #rows, # unnamed columns, etc.), the name of the relation it represents, the address of the base, and the list of attributes.

Base

The base is the main translation engine, doing parsing, variable substitution, and string formatting to generate the SQL statements. Overview of the translation algorithm is shown in Figure 10. Outlines and functionality of each major modules are in appendix. But the dataflow figures should be self-explanatory. In terms of state information, the base maintains a list of opened relations, condition box status, query editor status and other misc. status information when processing requests.

Overview of translator architecture

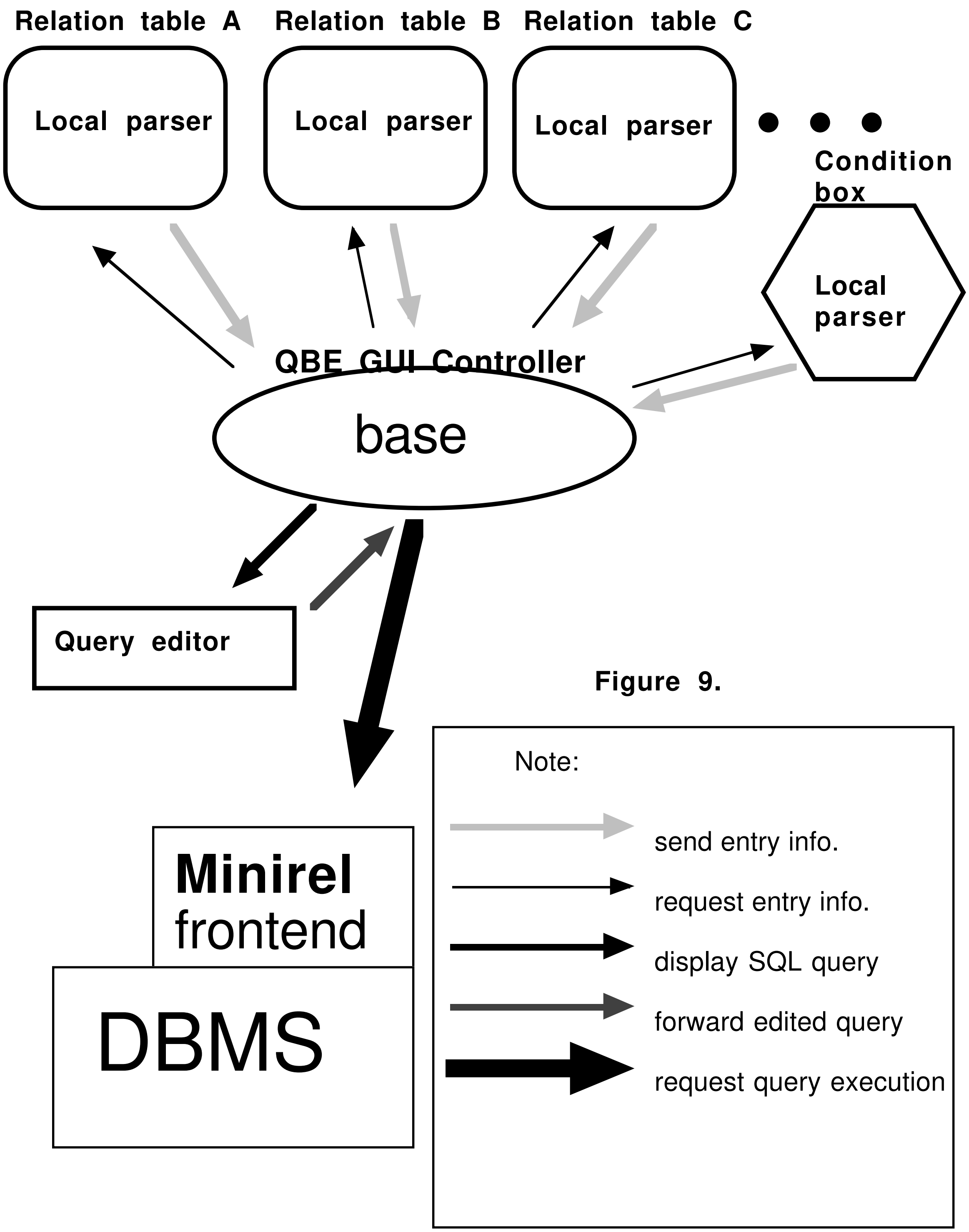


Figure 9.

Communication protocol

Current implementation of the parser components relies on build in Tk and X protocol do handle event notification, data delivery, and security; since both areas are not a major concern for this project, we recommend the interested reader to check out references on Tk and X protocol specifications. As a remark, there is also a small set of communication routines for synchronization between the base and relation tables, i.e. to notify the base that the user has closed a relation table.

Translator data structure

Since all data structures in Tcl are represented in terms of lists, the data structure of the translation algorithm is in terms of lists too, e.g. information is passed around in lists, and are kept in lists. For simplicity, there is no attempt to derive a generic data structure for the translator. Each query command basically has its own data structure and data format. Use of common data structure is limited to the communication layer between base and relation tables, and the basic variable replacement mechanism, which will be described in later sections.

In QBE, each box is an entry. NULL entries are kept for completeness[17], but they do not have any effect on the generated query because the current parser can not extract any useful information from them. Each row in a relation table is associated with a unique alias of the relation. For example, all attributes in row#1 is associated with relation A1 while all attributes in row#2 is associated with relation A2, if the rows are in the relation table for relation A. In terms of logic programming, each row can be viewed as a tuple variable while variables within each tuple variable are subgoals. The mission of the parser is to resolve these subgoals by matching them with an attribute. The rationale is, one of the instances of the variable must be simply associating itself with an attribute. Identifying such an instance would allow us to replace all appearances of the variable by its associated attribute. Once constrains are all bound to constants or attributes, it is just a matter of extracting the relation information, attribute list, and some embedded commands (i.e. aggregate commands) from the data and form the SQL query.

[17] It is possible that future extension of the parser might require the knowledge of NULL input. Current implementation of the parser requires that user explicitly type in NULL for NULL values

Translation algorithm

This section contains an in-depth treatment of how DML queries are parsed. Outline of the translation algorithms for other commands, together with the reference on specific data format used can both be found in appendix section.

The parser consists of a small set of generic routines, modules that are commonly used, and customized routines, modules that are called by a specific query command only. Figure 11 and Figure 12 detail the dataflow of how different query command modules processes the entry information. DUe to space limit, we will only discuss translation the query command P., which actually represents the entire set of DML, including selection, projection, union, negation, and join, here.

The parsing stage begins when the user requests the generation of a query at the base. The base responses by broadcasting a request to every opened relation table to send entry information. Different components go through the following different steps.

Local parsing at relation tables

Here is part of the local parser algorithm, as shown in Figure 13:

0) Read the command column to check for P., if found, it is a special case to project all attributes in a relation table, format the entry as if a P. is placed at every column

1) Read the command column to check for I. or D., if found, call corresponding GetInsertEntryCmd/GetDeleteEntryCmd, and send the result to the base

2) If there is no info.in command column, read in all the available attribute column entries pack them in the form of <attributes values> pair for each entry, empty ones have <attribute {}>;

3) check for U. in the list of <attribute values> pairs,
if found, call correspond GetUpdateEntryCmd, send result to base,
If not found, just pack the entry info with command UNKNOWN. to the base, and let it decide

Note: The brief outline of the algorithm used in each specialized GetXXXEntryCmd is in the appendix section. Further detailed documentation will be on line with the source code.

Overview of the parsing algorithm

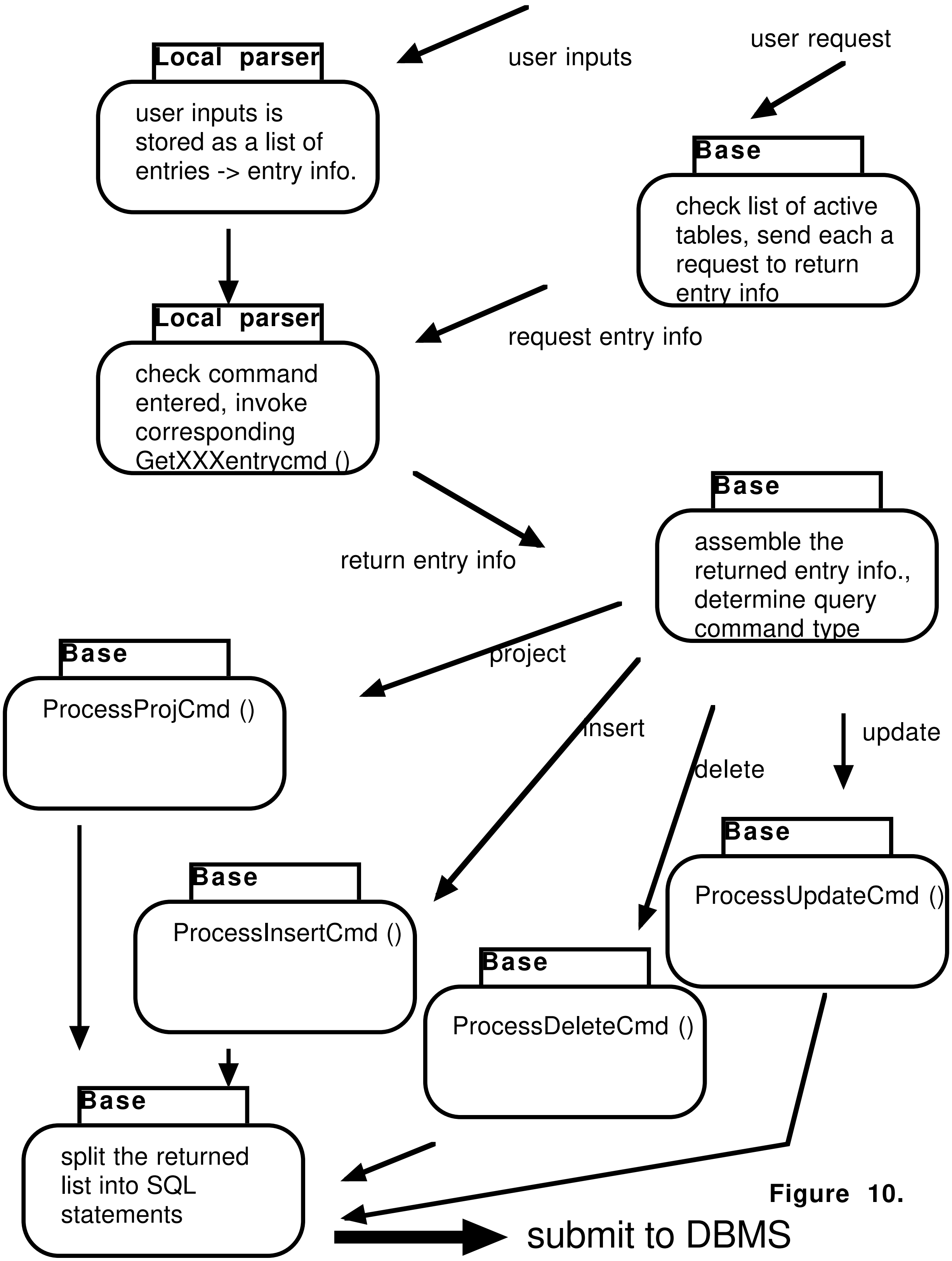


Figure 10.

Central processing at the base:

The base is basically the main translator. Upon user request, the base begins the query generation process by proceeding through the following stages:

1) Request stage:

   The base gathers entry information from all opened relation table by sending each active table a send data request, as described above.

2) Receive stage:

   The base arranges the received information by placing the ones with commands to the beginning of the list. As a result, entry lists with UNKNOWN. as commands are placed at the end. This is needed because entries list with command UNKNOWN. do not contain commands, These entries only help in forming the queries. The stage ends when all local parsers have responded with their respectively entry info.

3) Parse stage:

   After collecting all the commands from all opened relation tables, the base processes the command list by checking if there is incompatible commands[18], then determine the command to be performed and invokes the corresponding ProcessXXXCmd routine with the entry lists gathered.

4) Translate stage:

   Depending on the query command, this lengthy stage can be very different, let's examine the processing of DML queries in detail. Notice that in case that other query command is found, the corresponding GetXXXEntryCmd will be invoked instead.

Mechanism for translating DML queries

ProcessProjCmd - processing DML queries

This routine begins by checking if the DML requires an union. This can be determined by checking if command P. is found at more than one row in a relation table. If so, the answer is yes, no otherwise.

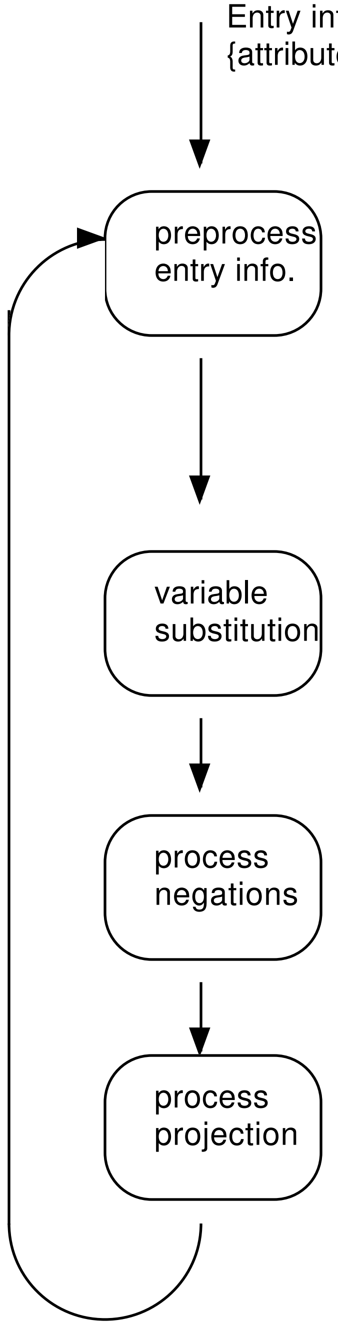
If a union is not required, then it proceeds through the following steps:

1) Do the variable substitutions,
2) Process any negation commands,
3) Extract projection attributes,

[18] current implementation assumes that there will be command from only one relation table, so the checking is optional now

Overview of ProcessProjCmd dataflow

Entry info: {attribute1 entry1} {attribute2 entry2} {attribute3 entry3} {attribute4 entry4} {attribute5 entry5} {attribute6 entry6} {attribute7 ...



1) check if the query requires a union

2) if it does, preprocess entry info. as if there are independent projection queries by duplicating the entry info., except the projection attributes, to each projection query; then process each query independently, join the resulting SQL statements with the SQL keyword **union**

3) if it does not, proceed with processing the projection query

1) scan through the entry info; if find pair in the form {attribute variable}, hash the pair into a string table;

2) scan through the entry info again, replace occurrences of variables by its corresponding attribute entry in the string table

1) extract relation name, attribute name, and constrains for the negated row

2) form a **not exist** clause using the information from step1, i.e. **not exists select** <attributes names> **from** <relation name> **where** <constrains>

1) extract relation name, attribute name, and constrains for the projection row

2)form a SQL query using the information from step1, i.e. **select** <attributes names> **from** <relation name> **where** <constrains>

3) append result from process negation

Figure 11.

4) Extract selection constrains,
5) Extract condition box entries,
6) Extract aliases of relation used for negation and projection,
7) Perform string manipulation to put the information together to form a valid SQL statement

Among the above steps, variable substitutions and processing of negation commands might need further explanation. Figure 11 contains a detail dataflow of ProcessProjCmd. Basically, variable substitution is based on the observation that in a valid QBE, for each variable, there must be a single entry that associates itself with a specific attribute. Following the assumption that we will have a correct input, all the <attribute entry> pairs are scanned once to build the hash table to associate each variable with an attribute using the above characteristic. Then a second scan would do the replacement of each occurrence of the variables with its associated entry in the string table. Processing of negation requires a little more work. A subquery is generated using the same translation technique as generating a projection/selection query, except that we assume no nested negation nor union within this subquery, thus avoiding recursive invoking of modules. This step ends with the formation of a not exists subclause.

In the case that a union is required, for each row with a P. command in it, the base would discard other rows' projection information, perform the above parsing steps, and join the result of each parsing together by inserting a keyword 'union' in between.

The projection algorithm described above can be appropriately extended to handle insert, delete and update commands. Interested readers are advised to read the appendix section for details on processing these query commands. Figure 12 also illustrates the dataflow schema of these commands.

Overview of Process[Insert/Delete/Update]Cmd dataflow

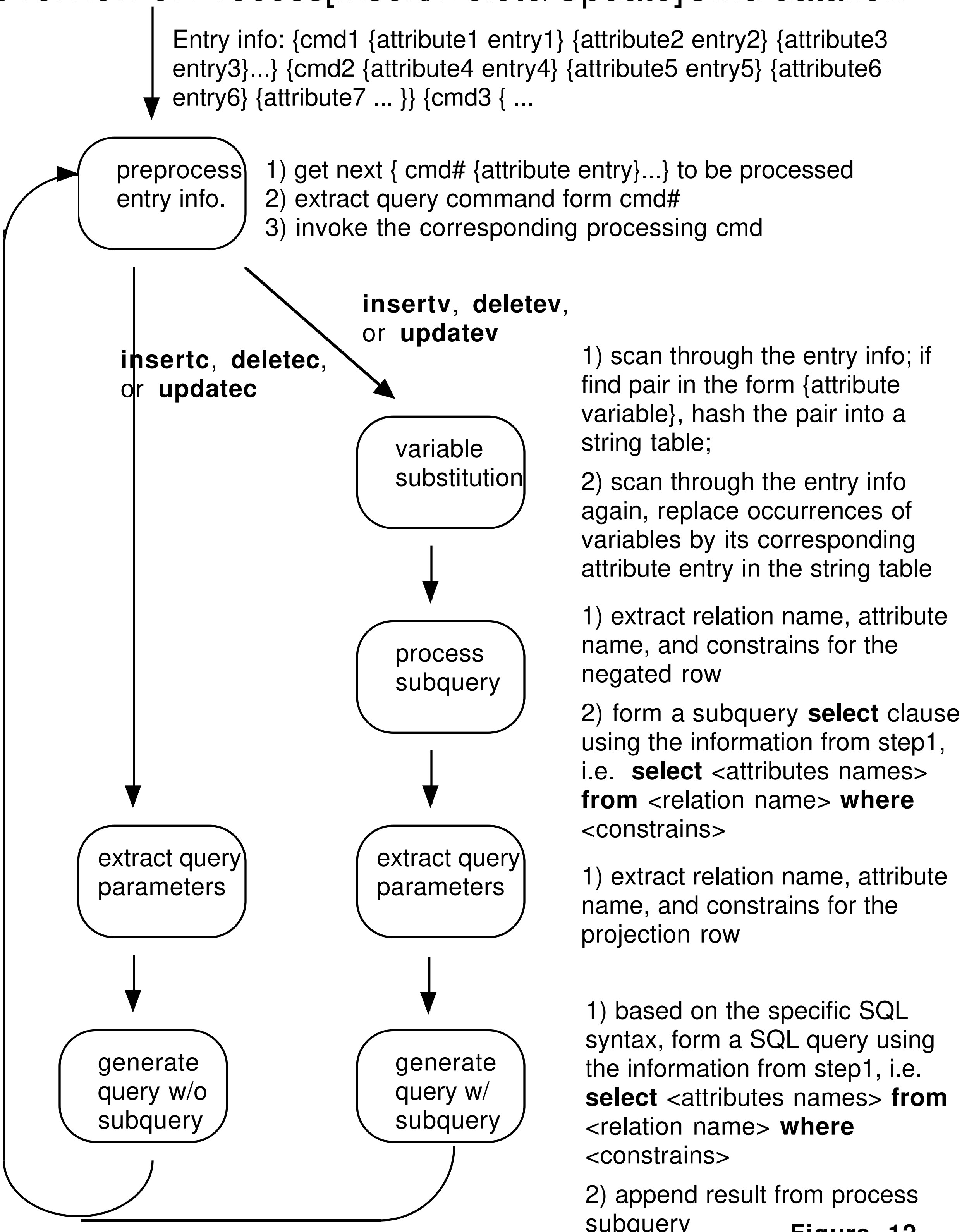


Figure 12.

Note: cmd has to be one of the following labels: **insertc**, **insertv**, **deletec**, **deletev**, **updatec**, and **updatev**

Implementation status

The following is a summary of the features that current implementation of the parser support/not support:

Supported features

1) P. with any combination of constants, variables, unions, negation, using unnamed columns,and/or condition box entries.
2) I.D.U. multi-statement, constants and/or variables, limited form of subquery, as mentioned above.
3) G., group by clause, a sample implementation for a simple extension of the parser to handle aggregate commands and qualifiers.
4) limited error checking

Not supported features:

1) aggregate functions – i.e. SUM. AVE. MIN. MAX. CNT.
2) qualifiers – i.e. UN. ALL. AO. DO.
3) union and negation in subqueries of insert, delete, and update queries
4) queries that require nested select's
5) total error detection

Comments

Before waving our hands to these neglected/incomplete features in QBE, let us constructively suggest how these features can be extended into the current implementation of the parser.

1) aggregate functions – can extend the parser by adding a preprocessing stage to replace these key words with corresponding SQL keywords and put a bracket enclosing the next item in the entry

2) qualifiers – can extend the parser by setting various flags in the projection processing routines to generate an additional keyword in the SQL statement, since basically quantifiers are simply additional keywords in the SQL statements at a predefined location

3) union and negation in insert, delete, and update subqueries – require some modification of structure as mentioned in the section above

4) nested select's – we are not aware of any valid QBE queries that require nested select's in corresponding SQL statements yet, but we are not ready to disprove it either

5) total error detection – it basically requires ad-hoc ways to check every entry and then conflicting entries. Additionally, each entry input must be processed. Due to time constrain, this

feature is incomplete. This is also not the main purpose of the project. We will leave this to future work section, if desired.

Obviously, adding each additional feature is very easy tasks, since all necessary tools are in placed, just the bookkeeping of new status variables must be carefully maintained.

Analysis of development tools:

The author's experience with using Tcl/Tk to develop this entire parser engine and GUI frontend has been very positive. Unified data structure, or lists-only-structures simplified the tasks on providing a consistent external interface for every major module. We also found that wishx, the Tcl/Tk interpreter, is especially helpful in learning and debugging Tk applications.

String based data types have made Tcl almost the ideal tool for doing this query translation parser. The Tcl technique being explored in this project is mainly the use of associate array, which makes data referencing a lot easier and faster. All variables are represented as strings, allowing the programmer to run from the nightmare of keeping all those indexing pointers. Moreover, in the extended version of Tcl, Tclx, there are a lot more very commonly used list and string manipulation routines, saving the programmer a week or so to write those basic utilities. Packers in Tk provides intelligent placement of widgets, making the complicated X window protocol almost transparent to programmer. Writing short GUI in Tcl/Tk is definitely an enjoyable experience.

There are also negative aspect of these oversimplification our development is providing us. When a feature is not supported, such as highlighting multiple listbox, it is very hard for users to do tear down each layer of the tool to identify the changes needed. More important, with Tcl, the tasks of doing a decent layout of the module, such as data abstraction or encapsulation, if it can be done, becomes extremely critical, since the language has very limited support for structuring. The string only data type and list only data structure certainly have limitations. And programmers have to work around this by copying data around, and putting special markers to serve as flags. Pass by reference requires knowledge of the scope of the variable. Thus often in the implementation, for simplicity, data is being passed by value and copied around. In addition to that, the lack of a real debugging tool for Tcl/Tk force programmers to rely on the interactive interpreter to carefully use primitive techniques to track down error sources.

Retrospect

Current implementation of the parser is heavily influenced by continuous evolvement of design decisions. Essentially, the overall structure is a result of continuous extensions, or generalization of simple string manipulation tools implemented. Thus a brief history of the work done would be very helpful in understanding the motivation of the parser structure.

The project started in mid February, when design proposal was to be drafted. After some planning and background readings, a proposal was turned in. The original proposal included a data structure in C, intended for building the hash table for the variable substitution stage. As soon as the author has realized there is a build-in hash table in Tcl, it has been decided that the entire project is to be done in Tcl, with the integration to C code for generating test data. A code review of Vivek’s GUI for Garlic was completed first in order to check if there is any special technique that would aid in parsing queries. Unfortunately, the Garlic project has a different priority of implementation. Therefore the essential routine, variable substitution for join operations, in QBE was actually not supported at the time the code review was done. The tabular form of GUI in QBE was distinct from the multimedia interface Garlic provides. Thus work on the GUI is started from scratch.

Since early March, there is intense programming to get the code base for QBE GUI completed in order to start working on the translator. Later on, part of this GUI was actually extended to be Minirel’s frontend GUI. At the late March, most GUI and communication protocol, after a little trouble[19] , was completed. The first goal of the project was to parse simple select’s with only constants as entry input, then simple projects and joins. This is when the variable substitution module was built, and it required minimal modification from then on, because the routine is a little complicated and with many details in it to take care of special cases such as empty entries, end of row markers, etc.

The initial GUI looked a like the GUI provided by Paradox[20] , in the

19 X server does not clean up dead wishx entries. Thus after every crash, as it happened very often when testing the Tk code, the wishx , when trying to contact dead widgets, will prompt communication error.

20 Paradox is a database application program by Microsoft

sense that projection commands are issued by clicking checkboxes at the corresponding attribute column. But the author later learned that straight QBE syntax were to be followed. That changed most of the modules implemented, except variable substitution. As soon as simple DML queries were tested, the attention of work was switched to parsing inserts, deletes and updates for entries with constants only. The DDL query commands require similar parsing techniques, except they have distinct SQL statements as output. Modules are largely borrowed from the ones implemented from the DML queries.

The second round of implementation involves doing DDL with subqueries. Unfortunately, codes are *copied* from DML routines, rather then *called* by those DDL routines. This is where it starts getting ugly. As a result of developing parsing routines for each query command independently, the code base for doing similar routines started to diverge, and each set of modules were making different assumptions on the data format. It often required a moderate amount of changes to modules whenever a new feature is added somewhere. This is the reason DML supports negation and union now, but not those routines for DDL queries. The final round of implementation involves handling union and negation for DML queries, and multiple SQL statements for DDL commands. Current source code, as a result of stages of modification, also contains code for routines that were previously used.

Fortunately, in spite of the fall back, the code base still appears very modular. In fact, this design approach has its strength too. Future people working on the code will be able to work on a specific command at a time. It is indeed very hard to identify the set of routines that all query commands would share the same pattern.

The main weakness of the parser is the error detection. Since new features are being added and added, there is no reason to do error checking before the entire parser, along with the all the planned feature is completed[21] . Also, size of source code suffers from using slowing evolving modules. A better overall design can possibly simplify the application development. For example, in some cases, such as generating subqueries in insert and delete, the DDL modules should have simply been able to call a more general version of ProcessProjCmd to form the subquery, instead of developing sets of smaller modules for each query command to handle it. It is not possible now because the many minor syntax difference of these DDL queries in SQL make it impractical to try to build a module

[21] There is a fair amount of error checking once the type of query command is determined.

general enough to build subquery for every query command.

V. Accomplishments

The project is all about months of intense and focused implementation work, from the time the proposal was turned in, to finalizing the GUI and the QBE translator. At each stage of the project we improve upon our old design and attempt to attain a better looking GUI and more versatile QBE to SQL translator. After every change of the project status we attempt too the integration. Indeed, this was when we found out some of the name conflicts and type conflicts. Every best effort was made to make sure the frontend system works along with other groups from the begin to end. We would just let the completed work speak for the amount of work we have done. Starting from scratch (Vivek's code was thought to be structurally too far off from the QBE's GUI design), and with a set of Tk library routines, the project now, with documentation, is altogether ~7k lines. For comparison, we are aware that 2 years ago, in Prof. Yannis's CS784 there was a QBE project that required a 3 person team to support a subset of QBE. Then again they were writing the entire application in C. We want to emphasize the hard effort and convenient programming tools we have.

VI. Testing

Testing for QBE to SQL translator and Minirel frontend GUI were done separately. Sample QBE test queries[22] are drawn from the two QBE references.[23] These sample test queries serve to test most forms of QBE queries. Additional test cases are also generated[24] . Regarding the Minirel frontend, several dummy layers, i.e. buffer manager, heap file and catalog layers, have been built to test the integration with the Tcl code. Indeed, only the browser requires a fair amount of testing with the GUI because all utilities are built as command line UNIX tools and are thus very simple to write and debug using gdb.

VII. Conclusion

The project is undoubted an excellent team work experience with other groups. We have suggested some ways to possibly improve the project –

[22] All sample queries from the references are used.

[23] Refer to reference section for book titles

[24] The complete set of test cases will be handed in as supplment to the report before the demonstration.

basically, to enforce more checkpoints and to come up with better design approach.

To the translator, more generalization could have been done to save some work, by observing that that the constant case of DDL can be a special case of select distinct query. By making all DDL contain subqueries, we could have save the work to implement another 3 sets of dataflow modules for the special case of queries that only contain constants as inputs.

If Zloof were to design QBE again, embedded command, such as U. would probably be replaced by a menu command. This is more user friendly and can substantially ease the parsing process, because then there is no need to extract embedded commands from the entries.

VIII. Future work

The QBE frontend is almost completed. Two major add on's will make it a complete product. One would be to extend the current implementation to support aggregate commands. This should be quite trivial. Second would be to identify queries that the IBM QMF's QBE would support, but not this QBE, especially in case of nested select's. If such a query is found, to support recursive generation of select clauses would probably require changes in the structure of the translator.

IX. Acknowledgement

The author would like to thank Professor Raghu Ramakrishnan and Vivek for their comments and ideas on the QBE GUI and parser implementation. We would also like to thank Professor Carey for inputs on frontend work. System group has undoubtedly provided the much needed utility support. This we have to thank Peter Feakins for his hard work. Finally we have to thank to entire class for working so hard to make this project work and our CEO happy.

X. Reference

-Introduction to Database System - QBE and QBF
-Introduction to Tcl/Tk
-QBE manual, IBM QMS Reference
-Jim Melton and Alan R. Simon, Understanding the new SQL
-PESTO, the GUI for Garlic

Appendix

•*Utilities command references*

- dbcreate <database name>
- dbdelete <database name>
- relcreate <database name> <relation name> <attribute datatype size> ...
  datatype = string, int, real, or null
- reldelete <database name> <relation name>
- indexcreate <database name> <relation name> <attribute indextype size>..
  indextype = _None, Hash, or B_Tree
- recinsert <database name> <relation name> <attribute value> ...
- recdelete <database name> <relation name> <attribute value>
- bulkload <database name> <relation name> <file name>

•*Format of <attribute value>*

attribute = (relation name(N)).(attribute name) N = row#
value = raw entry in the corresponding box

•*Format of <entry info>:*

- P., or {} found:
  <cmd, [<attribute, values>] >
  cmd = P. or UNKNOWN.
- I., D., or U. found
  <insert, list of <insertcmd,list of <attribute values> pairs>
  insertcmd = insertc, or insertv
  deletecmd = deletec, or deletev
  updatecmd = updatec, or updatev

•*Outline of GetXXXEntryCmd (XXX denotes command found)*

- GetInsertEntryCmd

Read the entries row by row

Determine the type of insert by scanning each row to see if there is any variable involved:

1) If not: tagged insertc to the data list, send to base
2) If yes, tagged insertv to the data list, send to base

The result of each row is appended together and sent to the base

- GetDeleteEntryCmd

Determine the type of delete by scanning each row to see if there is any variable involved:

1) If not: tagged deletec to the data list, send to base
2) If yes, tagged deletev to the data list, send to base

The result of each row is appended together and sent to the base

- GetUpdateEntryCmd

Similar concept to GetInsertEntryCmd and GetDeleteEntryCmd, except that it must identify the attribute to be updated, and determine the type of update by scanning each row to see if there is any variable involved:

1) If not: tagged updatec to the data list, send to base
2) If yes, tagged updatev to the data list, send to base

The result of each row is appended together and sent to the base

•*Outline of ProcessXXXCmd (XXX denotes the command found)*

- ProcessInsertCmd

From the local parser, the base obtains a list of insert queries, tagged with insertc or insertv command, therefore, for each insert query:

1) If tagged with insertc, invoke ProcessInsertConstantCmd
2) If tagged with insertv, invoke ProcessInsertVariableCmd

ProcessInsertConstantCmd

This module handles with inserts that do not require subquery, given the entry lists

1) Extract all constants
2) Extract all attributes
3) Extract name of the relation to do insertion
4) Perform string manipulation to put the information together to form a valid SQL insertion query

ProcessInsertVariableCmd

Sometimes insert commands require subqueries to allow users to combine projection and selection query and insert commands into one step. Local parser will conclude that insertion requires a subquery as soon as it finds a variable in the row where a insert command, I., is found. The module is basically a combination of ProcessProjCmd and ProcessInsertConstantCmd, except that for simplicity, no union nor negation is supported in the subquery part (it should be a very easy modification of this module to incorporate the union and negation, and it will be done if time permits)

1) Extract update attributes
2) Variable substitution
3) Subquery generation following the case of projection
4) Repeat the 4 steps in the case that only constants are used

- ProcessDeleteCmd/ProcessUpdateCmd

Formation of delete and update queries, conceptually, are very similar.

1) Specified constrains are first extracted
2) Formation of a subquery in case variables are found
3) Eliminate the aliasing of relations,
4) Put the pieces together

Essentially there are only differences in command names used, i.e. deletec and deletev, instead of updatec and updatev, and the syntax of the SQL statement. Interested readers are advised to compare the section of delete modules with update modules for understanding the differences.