A Simple Interface for Non Standard Knowledge Systems (SINKS)

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirement for the Degree

Master of Science

By

Harini Rao

December 2003

Harini Rao

harini12@yahoo.com

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Christopher Pollett

Dr. Archana Sathaye

Dr. Suneuy Kim

APPROVED FOR THE UNIVERSITY

Abstract

Deductive database systems combine a declarative style for formulating queries and constraints with efficient and reliable database technology for mass-memory data storage. They have the ability to use a logic programming style for expressing deductions concerning the contents of a database. Currently, most of the deductive-style databases such as NAIL!, LDL, CORAL, XSB, etc. usually act as front-ends to a more traditional relational database. It might make it easier to deploy a deductive database system as a back end for a relational database since, then only, the designer of the database needs to understand the non-standard knowledge system and the application programmers can use the SQL they know and love. The purpose of this project is to develop an interface system whereby non-standard knowledge systems such as XSB can make their resources available as a backend to a more traditional relational database, Oracle.

Table of contents

Page

1.Introduction

The relational data model is the most well-known and widely used data model. Some of the major advantages of this model are flexibility, data independence, simplicity, sound theoretical background and set processing. However, the expressive power and functionality of relational database query languages are limited compared to that of logic programming languages. Relational languages also do not have built-in reasoning capabilities. Conversely, logic programming is programming by description and is based on mathematical logic. It uses logic to represent knowledge and uses deduction to solve problems by deriving logical consequences. But, logic-programming techniques are not alone enough for managing large, shared, and reliable data collections.

Deductive databases are a result of the integration of relational databases and logic programming techniques. They combine the benefits of both the approaches, such as reasoning capabilities, recursion of logic programming, and declarative querying, and efficient secondary storage access of relational data model. They also provide means for expressing disjunction and negation and query processing is also much simpler and easier in deductive databases. However, one impediment to their more widespread acceptance is that most database application programmers are familiar with relational database programming, and are less or even totally unfamiliar with programming in a logic-based or other non-standard paradigms.

Currently, most of the deductive-style databases such as NAIL!, LDL, CORAL, and XSB, usually act as front-ends to a more traditional relational database. It might make it easier to deploy a deductive database system as a back end for a relational database since, then only, the designer of the database needs to understand the non-standard knowledge system and the application programmers can use the SQL they know and love.

The purpose of this project is to develop an interface system whereby non-standard knowledge systems such as XSB can make their resources available as a back-end to a more traditional relational database, Oracle. The Oracle–XSB interface is a subsystem that allows Oracle users to access XSB databases. This interface allows users to access the facts stored in the deductive database XSB from Oracle's environment as though they existed as tables. It allows users to write explicit SQL statements to be passed to the interface to retrieve data from XSB and all database accesses to be done on the fly.

This report is organized as follows. Section 1 (Introduction) provides information on the related research and background of this project. Section 2 gives a brief overview of deductive databases, Stable models and SModels. Section 3 covers XSB and tabling in XSB. Section 4 narrates the innovative features of the interface and challenges encountered during its development. Section 5 (Design) illustrates the architecture and the design pattern used for this project. Section 6 discusses implementation details of the

Oracle-XSB interface. Section 7 presents some of the potential applications of the interface. Section 8 discusses future enhancements and Section 9 concludes with the potential of the interface suggested.

2.Deductive Databases

A deductive database system is a database system that includes capabilities to define deductive rules, which can deduce or infer additional information from the facts that are stored in a database. Facts and rules are the two main specifications used by a deductive database. A fact is an expression that can be interpreted as true or false. Rules are sentences, which define the data. They are used to derive new and complex information from stored data and to specify integrity constraints on objects. These deductive rules cannot modify the database and are therefore called as passive rules.

Deductive rules are similar to relational views in some ways in that, they specify virtual relations that are not actually stored but that can be formed from the facts by applying inference mechanisms based on the rule specifications. However, deductive rules may include recursion and hence may sometimes yield virtual relations that cannot be defined in terms of typical relational views.

The meaning of rules can be interpreted in two ways: proof theoretic and model theoretic. In the proof theoretic interpretation of rules, facts and rules are considered as true statements or axioms. Facts are ground axioms, which contain no variables, and which are given to be true whereas rules are deductive axioms, which can be, used to construct proofs from existing facts. This interpretation gives a procedural or computational approach for computing answers to a Datalog Query. In the second type of interpretation called as the model-theoretic interpretation, given a finite or infinite domain of constant values, every possible combination of these values as arguments is assigned to a predicate. Then it is determined whether the predicate is true or false. If this is done for every predicate, it is called an interpretation of the set of predicates.

An interpretation is called a model for a specific set of rules if those rules are always true under that interpretation. i.e., a model of a program is a set of facts such that for any rule, replacing body literals by facts in the model results in a head fact that is also in the model. A model is called a minimal model for a set of rules if any fact cannot be changed from true to false and a model is still obtained. A minimal model is a model such that no subset is a model. In the presence of negated literals, a program may not have a minimal model.

Eg: p(a) :- ~p(b)

has two minimal models {p(a)} and {p(b)}

In other words, the minimal model is the smallest set of atoms that characterize a positive logic program. Stable models is a similar notion for programs that contain negation.

2.1 Stable Models

Stable model semantics were proposed by Gelfond and Lifschitz to express logic programs without negation. The stable model semantics for a logic program P can be calculated as follows: It is assumed that all the atoms in P are ground atoms by replacing each rule containing variable by all its ground instances. A residual program P_M (for any set M of atoms from P) is then calculated by deleting

i) each rule that has a negative literal ~G in its body with G in M, and

ii) all negative literals in the bodies of the remaining rules.

Since P_M is now negation-free, it has a unique minimal Herbrand model and if this model is the same as M, then M is said to be the stable model of P. P can have more than one stable model.

For example, consider the following non-stratified program,

a:-~b,c.
b:-d.
b:-~a.

c.

There are two stable models of this program: in one, a and c are true, and in another, b and c are true. The residual program for the above program is

a:-~b.

b:-~a.

c.

The following program demonstrates a method for calculating the Stable Models**.** The mechanism used to calculate Stable Models for a given program is as follows: All the rules of the given program are stored in a text file. The number of literals occurring in each of these rules is calculated and stored in an array.

```
do
{
i = fin.read () ;
j = fin.read () ;
if (i != -1)
{
  c = (char) i;
  j = i + 1;
  g = (char) j;
  if (c != g && c != '~' && c != ':' && c != ',')
```

```
   {
     a[k] = c;
     if(a[k] != '\n' && a[k] != '\r')
                    k++;
   }
}
}
while (i != -1) ;
for (k = 0; k < a.length; k++)
{
if (a[k] != '\n')
  if (a[k] != '\r')
  if (a[k] != 0)
l++ ;
}
```

Depending on the number of literals, assumptions are made.

```
for(int k = 0; k < l; k++)
{
int g = i%2;
if(g==1)
temp[k] = true;
else
temp[k] = false;
i = i/2;
```

```
}
```

Each of the rules in the program is then read and stored in an ArrayList.

```
do
{
i = fin.read () ;
if (i != -1)
{
c = (char) i;
if (c != '\n')
{
if (c == ':')
s = s+"=" ;
else
if (c == '~')
s = s+"!" ;
     else
if (c == ',')
s = s+"&" ;
   else
s = s+c ;
}
else
{
al.add (s) ;
s = new String() ;
```

```
}
}
else
al.add(s);
}
while (i!=-1) ;
```

The truth-values in each assumption are then applied to the literals in each rule and a reduced program is calculated by removing all negated literals in the rules.

```
for(int k = 0; k < l; k++)
{
if(k>=p)
break;
if (b[k] == '=')
{
if (b[k+1] != '!')
{
if(v.contains(String.valueOf(b[k+1])))
gy = v.indexOf(String.valueOf(b[k+1]));
t = temp1[gy] ;
String s2 = b[k-1]+":"+b[k+1];
if(t)
al1.add(s2);
if(v.contains(String.valueOf(b[k-1])))
```

```
ge = v.indexOf(String.valueOf(b[k-1]));
temp2[ge] = t;
}
else
{
if(v.contains(String.valueOf(b[k+2])))
gy = v.indexOf(String.valueOf(b[k+2]));
t = temp1[gy];
String s2 = b[k-1]+":"+".";
if(!t)
al1.add(s2);
if(v.contains(String.valueOf(b[k-1])))
ge = v.indexOf(String.valueOf(b[k-1]));
temp2[ge] = !t;
}
}
if (b[k] == ',')
{
if (b[k+1] != '!')
{
if(v.contains(String.valueOf(b[k+1])))
gy = v.indexOf(String.valueOf(b[k+1]));
t = temp1[gy] ;
String s2 = b[k-1]+"&"+b[k+1];
if(t)
al1.add(s2);
```

```
if(v.contains(String.valueOf(b[k-1])))
ge = v.indexOf(String.valueOf(b[k-1]));
temp2[ge] = t;
}
else
{
if(v.contains(String.valueOf(b[k+2])))
gy = v.indexOf(String.valueOf(b[k+2]));
t = temp1[gy];
String s2 = b[k-1]+".";
if(!t)
al1.add(s2);
if(v.contains(String.valueOf(b[k-1])))
ge = v.indexOf(String.valueOf(b[k-1]));
temp2[ge] = !t;
}
}
}
```

A check is made to see if any more variables can be derived from the rules in the reduced program and if so, the derived program is calculated.

```
for(int sk = 0; sk < p; sk++)
{
if (change)
```

```
break;
if (b[sk] == ':')
{
if (b[sk+1] == '.')
{
if(al1.size() > 1)
{
st = String.valueOf(b[sk-1]);
change = true;
}
else
{
change = true;
if(v.contains(String.valueOf(b[sk-1])))
ge = v.indexOf(String.valueOf(b[sk-1]));
temp3[ge] = t;
}
}
str = st;
if(str.equals(String.valueOf(b[sk+1])))
{
if(v1.contains(String.valueOf(b[sk-1])))
ge = v1.indexOf(String.valueOf(b[sk-1]));
temp3[ge] = t;
}
}
```

```
}
```

and then another check is made to see whether the assumption made initially is same as the result.

```
for(int ty = 0,tx = 0; ty < l && tx < l; ty++,tx++)
{
if (temp[ty] == temp3[tx])
gu++;
}
```

If the truth values of the literals in the derived program and the original assumption are same, then that assumption is a stable model for the given program.

```
if (gu >=l)
{
System.out.println("Stable Models:");
for(int qs = 0; qs < l; qs++)
{
System.out.println(v.elementAt(qs)+" is "+temp[qs]);
}
}
else
System.out.println("This is not a Stable Model");
```

This process is repeated for all the assumptions made.

2.2 Smodels

Smodels system was developed by Niemela and Simons to compute stable models of logic programs. It is implemented in C++ language and contains two modules namely smodels and parse. smodels implements the well-founded and stable model semantics for ground programs and parse computes and produces a set of ground instances of a normal program. Smodels system uses a bottom-up backtracking search strategy with a powerful pruning method to implement stable model semantics. This strategy can be implemented to work in a linear space thus making to possible to use stable model semantics in applications where the resulting programs contain a large number of stable models. One such scenario is described below.

The following program creates random Smodels knowledge bases and uses the Smodels system to generate stable models. This program takes parameters from the user such as number of the rules, number of the variables, probability for a literal to exist in a rule and the probability of that literal being non-negative. It also takes another parameter namely the number of programs to be generated so that the user can simultaneously test many programs with the Smodels package to generate Stable Models.

do

```
{
String s1 = br.readLine();
a[i] = s1;
i++;
}
while (i < 5);
```

Depending on the number input by the user, the program automatically generates the variables and stores them in an arraylist.

```
int size = Integer.parseInt(a[1]);
ArrayList b = new ArrayList(size);
for (int k = 0; k < size; k++)
{
String var = "v"+String.valueOf(k);
b.add(var);
}
```

It then cycles through the variable list and generates each rule by calculating the probabilities of those variables to appear in that specific rule.

```
for (int k = 0; k < size; k++)
{
if (k != j)
{
```

```
double pe = Math.random();
double pp = Math.random();
int pe1 = (int) (pe*10);
int pp1 = (int) (pp*10);
int problit = Integer.parseInt(a[2]);
int probposlit = Integer.parseInt(a[3]);
if (pe1 <= problit)
{
if (pp1 <= probposlit)
      {
if (y > 0)
{
sfile = sfile+", "+b.get(k);
}
else
{
sfile = sfile+b.get(k);
}
y++;
}
else
{
if (y > 0)
{
sfile = sfile+", not "+b.get(k);
}
```

```
else
{
sfile = sfile+"not "+b.get(k);
}
y++;
}
}
else continue;
}
}
sfile = sfile+".";
```

When all the rules are generated, they are stored in a text file in a program format.

```
if (sfile.length() == 5)
{
c = sfile.charAt(0);
sfile1 = String.valueOf(c);
char d = sfile.charAt(1);
sfile1 = sfile1 + String.valueOf(d);
char e = sfile.charAt(4);
sfile1 = sfile1 + String.valueOf(e);
}
out.write(sfile1);
```

```
}
out.write("compute all { not fail }.");
out.close();
```

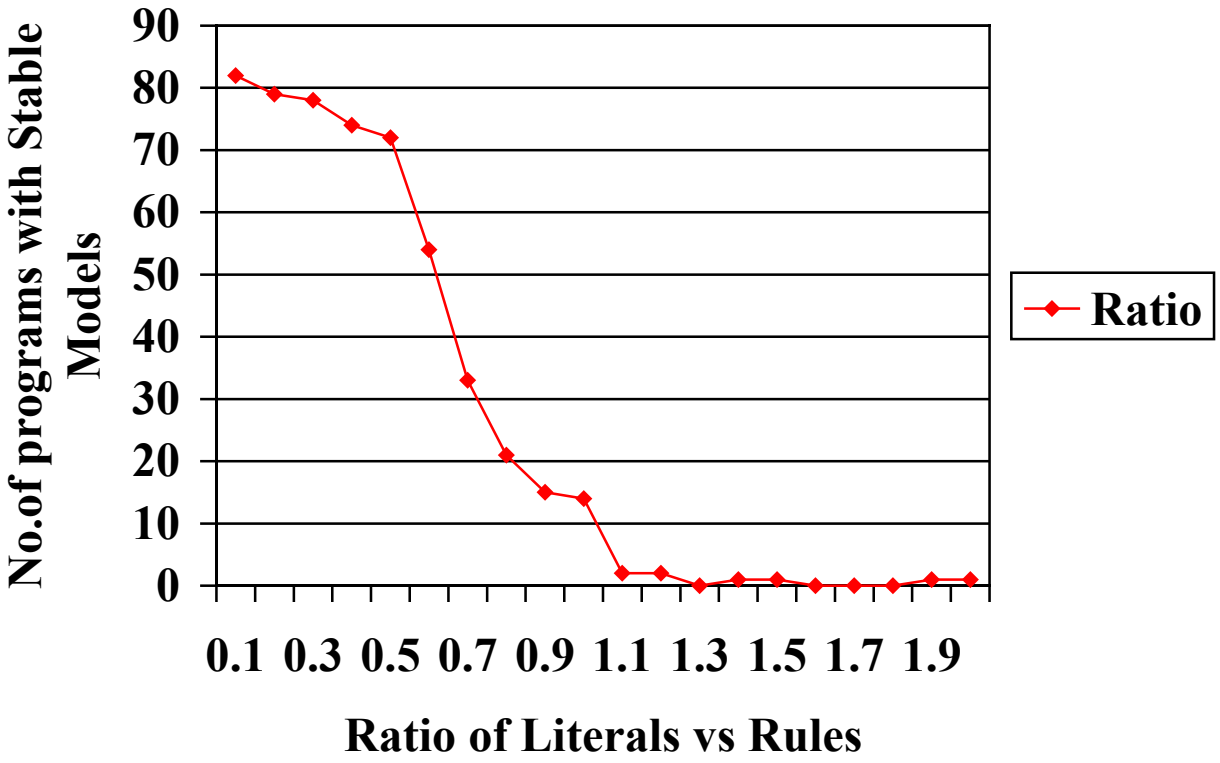
This process is repeated for a number of times specified by the user.

The text file, which contains all the required number of programs, can be used for testing with Smodels package. The following paragraph examines the behavior of the above program in various scenarios and checks whether any threshold phenomena hold. The performance of the above program is examined in two scenarios namely, when the number of rules is more than the number of the variables and when the number of variables is more than the number of rules to be generated. Each of these scenarios is checked by generating a fixed the number of programs, say in this case, a 100 and the number of programs with stable models is calculated in the above two scenarios.

Case 1: When the number of rules is more than the number of variables. In this case, the number of rules is 50, number of variables generated is varied from 5 to 50. The probability for a literal to exist is 0.3 and probability for that literal to be positive is 0.2. The number of programs generated by the above program is 100 and stable models are calculated for these generated programs using the Smodels package.

Case 2: When the number of variables is more than the number of rules generated. In this case, the number of rules is 50, number of variables generated per program is varied from 55 to 100. The probability for a literal to exist is 0.3 and probability for that literal to be positive is as 0.2. The number of programs generated by the above program is 100 and stable models are calculated for these generated programs using the Smodels package.

A graph is plotted to observe the relationship between the number of programs with stable models and the ratio of literals and rules in the generated programs. It is observed from the above graph, that the number of programs with stable models gradually decreases as the ratio of the number of literals and rules increases and becomes almost constant between the values 1.0 and 2.0. A significant decrease is noted in the number of programs having stable models when the number of literals is increased from 25 to 30. As the number of literals increases and becomes almost twice the number of rules generated, the number of programs with stable models almost remains constant and varies only slightly between the values 0 and 1.



All the observations made above hold true when the above two cases are repeated for different values of the probabilities for a literal to exist and for it be positive in the rules generated.

3.XSB

XSB is a Logic Programming and Deductive Database system for Unix and Windows. It is being developed at the Computer Science Department of Stony Brook University. XSB uses a goal-directed resolution strategy called SLG resolution to solve SLD's problems of lack of finiteness and redundant computation. Prolog uses SLD resolution strategy that employs depth-first search through trees and thus might loop infinitely. On the contrary, SLG stores answers to certain queries in a table and rather than expanding the repeated subgoals, it resolves the subgoals against answer clauses in the table. Thus, it avoids looping infinitely and non-terminating evaluations for programs with finite models. To handle general negation efficiently, SLG uses a scheduling strategy and delay mechanisms.

XSB uses tuple-at-a-time evaluation strategy and evaluates stratified queries much faster than current bottom up implementations. This gives it an advantage in object-oriented applications where information can be kept in complex terms rather than distributed across many relations. An advantage of the top-down approach used by XSB is that its efficiency relies less on the rewriting techniques and on alternate control strategies than as usual in the bottom-up approaches.

3.1 Tabling in XSB

XSB system provides two methods for evaluating predicates: Prolog-style evaluation and tabled resolution. By reusing partial answers to the query, tabled resolution makes it possible for a user to successfully evaluate programs with negation. XSB provides the programmer the choice to specify what calls should be tabled by using a compiler directive, such as:

: - table proc/2.

This example requests that all calls to the procedure proc that has two arguments should be tabled. Predicates that have such declarations in a given program are called as tabled predicates. Tabling is also called as memoization or lemmatization and is discussed at length in the following paragraph.

Consider the following Prolog program

```
superior(X,Y) :- supervise(X,Y).

superior(X,Y) :- supervise(X,Z), superior(Z,Y).
```

together with the query ?-superior(1,Y). This program has a simple, declarative meaning: Y is a superior of X if Y is a supervisor of X or if Y is a superior of a supervisor of X. Prolog, however cannot compute the answer to this query as it enters an infinite loop. The

inability of Prolog to answer such queries, which arise frequently, comprises one of its major limitations as an implementation of logic.

A number of approaches have been developed to solve programs like superior by reusing partial answers to the query superior(1,Y). One of them is tabling, where the implementation keeps track of all calls to tabled predicates, or tabled sub goals. Whenever a new tabled subgoal is called, a check is first made to see whether it is in the table. If so, the sub goal is resolved against answers in the table. If not, it is entered into the table and resolved against program clauses. When an answer to a tabled subgoal is derived, a check is made against the table to see if the answer is there. If the answer is not in the table, the answer is added and scheduled to be returned to all instances where the subgoal has been called. If the answer is already in the table, the evaluation simply fails and backtracks to generate more answers.

The following program demonstrates XSB's tabling features.

```
:- table superior/2.
superior(X,Y) :- supervise(X,Y).
superior(X,Y) :- supervise(X,Z), superior(Z,Y).

supervise(jennifer,ahmad).
supervise(james,jennifer).
```

```
supervise(franklin,john).
supervise(franklin,ramesh).
supervise(franklin,joyce).
supervise(jennifer,alicia).
supervise(james,jennifer).
supervise(james,franklin).
```

This program returns answers to the queries like: ?- superior(X,jennifer) which calculates all the superiors of jennifer in the example, using the tabling procedure explained above and displays them one per line. It also returns answers to queries like ?- superior(james,ahmed) which essentially check whether james is the superior of ahmed or not.

4.Oracle – XSB Interface

One of the goals of XSB is to provide an implementation engine for both logic programming and for data-oriented applications such as data mining and so it provides a number of interfaces to other software systems, such a C, Java, Perl, ODBC, SModels, and Oracle. The purpose of this project is to develop an interface between Oracle and an XSB system and so some of the interfaces mentioned above such as the C-XSB interface are utilized in the design of this project.

4.1 Innovations

The Oracle-XSB interface is completely different from the XSB-Oracle interface provided with the XSB system, in both design and implementation perspectives. It uses a novel approach to access XSB from Oracle using C procedures as the link between the two environments. It employs relatively new and advanced technologies like updatable views, instead of triggers in its design.

The Oracle-XSB interface will allow facts in the deductive database XSB to be accessed from Oracle's environment as though they existed as tables. It will allow users to write SQL statements to access data from XSB and does not require the user to be familiar with programming logic-based systems. This system is innovative in that it will lower how much a user needs to know compared to the present situation in order to add fuzzy or rule-based information to a relational database. Adding such table that could be useful to limit the scope of queries and potentially improve the speed of their evaluation.

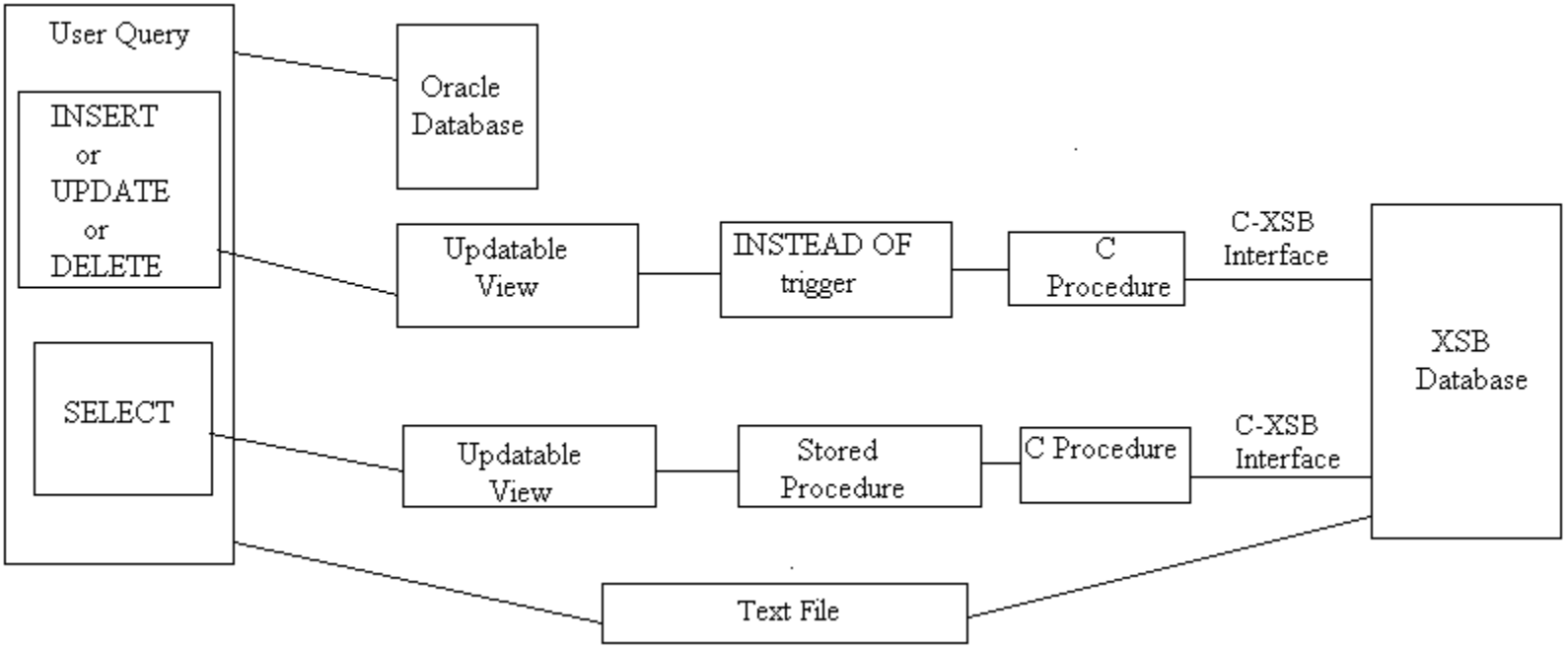
4.2 Challenges

Several challenges were encountered during the development process of the interface both in the design and the implementation stages. Translation of SQL queries into prolog queries is complex. Retrieval of data without updating it posed a complex design challenge as triggers can only be invoked by the DML statements (INSERT, UPDATE, or DELETE). Stored procedures were the alternative used for the retrieval of data without updating it (for a SELECT query). Accessing external C procedures from Oracle also posed a challenge, as some of the libraries needed are not provided in the required directory. They had to be copied into the appropriate location for Oracle to access them. Calling XSB from C also posed a challenge, as the existing C-XSB interface provided with the XSB system created an executable file. This .exe file must be manually invoked to access XSB from C and as one of the objectives of this project is to hide all the design and implementation complexities from the user, this process needed to be automated. There was no mechanism provided in C to return the data from XSB to Oracle. Though popen system call could be used to accomplish this in Unix systems, this is not possible in Windows and hence has to be implemented using text files.

5.Design

The Oracle–XSB interface is a subsystem that allows Oracle users to access XSB databases. This interface allows facts in the deductive database XSB to be accessed from Oracle's environment as though they existed as tables. It allows the users to write explicit SQL statements to be passed to the interface to retrieve data from XSB and all database accesses will be done on the fly. XSB can be directly linked into C programs in UNIX and XSB can be linked into C programs through a Dynamic Link Library (DLL) interface in a Windows-based system. Similarly, C functions can be made callable from XSB either directly within a process, or using a socket library. XSB can access external data in a variety of ways: through an ODBC interface, through an Oracle interface, or through a variety of mechanisms to read data from flat files. Hence the interface proposed above uses all the above features along with updatable views, instead of triggers and stored procedures as discussed in the following paragraphs.

Data Flow Diagram of the Oracle-XSB interface



5.1 Updatable Views

A view is a virtual table whose contents is defined by a query and is derived from other tables called base tables. Base tables can be actual tables or might be previously defined views. Similar to a real table, a view consists of a set of named columns and rows of data. The rows and columns of data in the view come from tables referenced in the query defining the view and are produced dynamically when the view is referenced. All operations performed on a view affect the base table of the view. Views support the

operations like query, update, insert into, and delete from views, just as standard tables. Views can be used to establish additional levels of security for table data by restricting access to a predetermined set of rows and/or columns of a table, derive information from base table data, and simplify application development.

A view is not modifiable if its view query contains joins, set operators, aggregate functions, GROUP BY, CONNECT BY, START WITH clauses or DISTINCT operator. If a view query contains pseudo columns or expressions, the corresponding view columns are not updatable. To overcome these obstacles Oracle provides INSTEAD OF triggers.

5.2 INSTEAD OF Triggers

Triggers are procedures that are stored in the database and implicitly executed, or fired, when the table with which it is associated has an insert, delete or update operation performed on it (or, in some cases, against a view) or when database system actions occur. These procedures can be written in PL/SQL or Java or they can be written as C callouts. The events that fire a trigger include DML statements that modify data (INSERT, UPDATE, or DELETE), DDL statements, system events such as startup, shutdown, and error messages, and user events such as logon and logoff. Triggers can be used to enforce complex business rules, security authorizations or to automatically

perform additional actions required by business rules. There are different types of triggers:

- Row Triggers and Statement Triggers
- BEFORE and AFTER Triggers
- INSTEAD-OF Triggers
- Triggers on System Events and User Events

An Instead of trigger is a trigger that tells Oracle how to process a DML operation (INSERT, UPDATE, or DELETE) performed on a view. Unlike other types of triggers, Instead of triggers fire instead of the triggering statement. In other words, an Instead of trigger tells Oracle to execute the body of trigger instead of performing the actions that invoked the trigger. By default, Instead of triggers are activated for each row. These triggers can also be defined on object views in addition to the standard relational views. However, they cannot be defined on base tables. Since views can be a combination of more than one base table, Instead of triggers can be used to perform DML operations directly on the underlying tables, a functionality that was previously not available.

5.3 Stored Procedures

A procedure is a subprogram that performs a specific action. It can take parameters and be invoked. A procedure has two parts: the specification and the body. The procedure body has three parts: an optional declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and subprograms. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains handlers that deal with exceptions raised during execution. A stored procedure is a named group of SQL statements previously created and stored in the server database. Stored procedures reduce network traffic and improve performance along with security. Additionally, stored procedures can be used to help ensure the integrity of the database.

6.Implementation

This interface provides the users, the capability to access data in XSB environment through SQL queries. The facts stored in the XSB database are represented in the Oracle environment by using updatable views. When the user wants to access and update the data in the database, he types in an SQL query. If the data is present in the Oracle database in the tables, the query is processed and results are returned. However, if the

data exists as facts in the XSB database, the user can access it by invoking the instead of triggers on the updatable views, which in turn, call the C procedures, which then interact with XSB and returns the results through the text files to the Oracle environment. Since triggers can only be invoked by the DML statements (INSERT, UPDATE, and DELETE), it will be not possible for the user to retrieve the data without updating it. This problem can be resolved and the SELECT operation can be accomplished by using the stored procedures, similar to the triggers as above. The only difference between the triggers and the stored procedures is that triggers are implicitly fired by Oracle whereas stored procedures have to be explicitly called by the user.

The Oracle-XSB interface includes the following sub components.

- A View Level Interface that interacts with external C Procedures to translate SQL queries into Prolog clauses.
- The C-XSB interface (provided with XSB) that allows a C program to provide queries for XSB to evaluate and to retrieve back the answers.

6.1 External Routines

An external routine is a third-generation language procedure stored in a dynamic link library (DLL), registered with PL/SQL, and called by the DBA to perform special-purpose processing. At run time, PL/SQL loads the library dynamically, and then calls the routine as if it were a PL/SQL subprogram. An external routine is a procedure or function written in a language other than PL/SQL, but callable from a PL/SQL program. This is done by publishing the external routine to PL/SQL through a PL/SQL-callable entry point that maps to the actual external code. A PL/SQL program then calls the wrapper, which in turn invokes the external code. In order to call an external C routine, the following steps must be completed:

- The routine must be coded and compiled into a shared library.
- The Net8 parameter files must be configured and the listener must be started.
- A library data dictionary object must be created to represent the operating system library.
- The external routine must be published by creating a PL/SQL wrapper that maps the PL/SQL parameters to C parameters.

Coding the routine

The first step is to write the routine. Once this file is created, it must be compiled into a shared library. The shared library will be created in the current directory.

Configuring the Net8 Listener

The listener needs to be configured only once. Once it is set up and running, the extproc process will be automatically spawned as needed. Configuring the listener requires two files-listener.ora and tnsnames.ora. Once these files are shared, the listener can be started. The default location for Net8 configuration files will vary depending on the operating system used. The file listener.ora specifies the parameters for the listener. The file tnsnames.ora is used to specify Net8 connect strings.

The SID specified in the CONNECT_DATA clause of both tnsnames.ora and listener.ora must be the same. For Oracle 8i, the installer will configure these files automatically to use PLSExtproc as the SID name. The ADDRESS clause including the protocol must be the same for both files as well. The listener is then started (and stopped) using the lsnrctl utility. This utility is launched from the operating system prompt and has a character mode interface similar to SQL*Plus.

Creating the Library

A library is a data dictionary object that contains information about the operating system location of the shared library on disk. Libraries are created using the DDL command as follows:

```
CREATE LIBRARY library_name {IS | AS}

'operating_system_path';
```

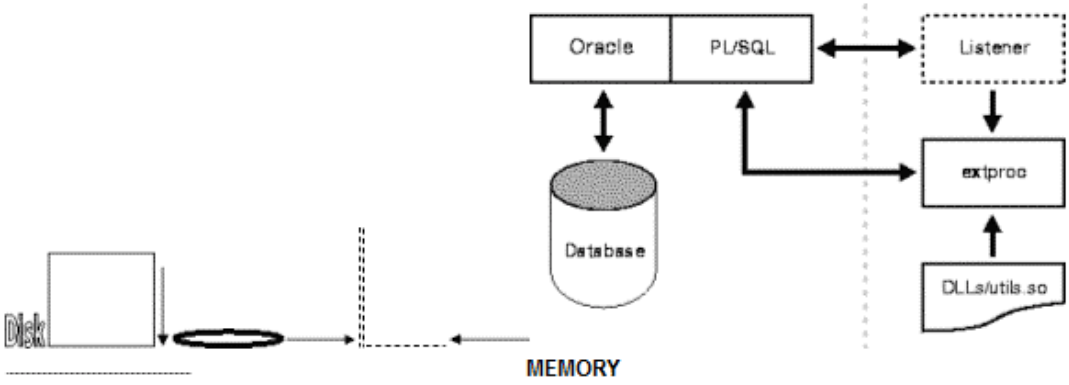
where library_name is the name of the new library, and operating_system_path is the complete path including directory, of the shared library on the file system. In order to create a library, the user needs to have CREATE LIBRARY system privilege and for other users to access it, EXECUTE privilege should be granted on it by the database administrator.

Publishing the Routine

In order to call an external routine from PL/SQL, it must be published. This is done by creating a PL/SQL wrapper, which serves several purposes: It maps the PL/SQL parameters to C parameters, it serves as a placeholder so that calling procedures can determine dependencies, and it tells PL/SQL the name of the external library. A wrapper consists of a subprogram specification including the parameters, if any, followed by an external clause or the AS LANGUAGE clause. The wrapper can be a stand-alone

procedure or function, or part of a package or type body.

Architecture of external routines



(Source: http://www.databasejournal.com/features/oracle/article.php/10893_1549161_2)

The following steps summarize how PL/SQL calls an external routine:

- PL/SQL alerts the Listener process
- PLSQL in turn starts a session-specific agent named extproc
- Listener hands over control to extproc

- PLSQL passes the name of the DLL or SO, the name of external Routine and any parameters to extproc
- Extproc invokes the disk
- It loads the specific DLL/SO in the memory for execution
- Extproc returns the results to PLSQL

Once the External Routine is complete, the extproc does not terminate, it remains active in the memory. The extproc is killed when the Oracle Session is terminated.

Using all the components described above, an equivalent Prolog clause is generated for the SQL query specified by the user. The external C procedure then invokes the C-XSB interface provided with the system to pass the query to XSB and retrieves the results. The following sections describe the architecture of the C-XSB interface.

6.2 Calling XSB from C

Several functions are provided by XSB that act as means of communication between C and XSB. These functions can be called from C and they allow a C program to initialize and interact with XSB just as a subroutine. They pass commands or queries to the XSB system, have them executed and retrieve the results. Some of these C routines use XSB-

specific C-type definition for variable-length strings and others permit the users to directly manipulate the XSB data structures to construct queries and retrieve answers. The following paragraphs describes some of these routines:

*int xsb_init(int argc, char *argv[]):* This function is used for initializing XSB and is the first function that must be called before any other functions. It takes two parameters, namely an integer argc and a vector argv [] where argc is the count of the number of arguments in the argv vector. The first argument in the argv vector, argv[0] should be the absolute or relative path name of the directory where XSB is installed. The second argument, argv[1] should be the -n flag which tells XSB to act as a subroutine to a calling C routine and not to start the read-eval-print top loop. This function returns a value 0 if initialization is completed and 1 if an error is encountered.

int xsb_command(): This is a C-callable function that passes a command to XSB. Before calling this function, any active query should be closed and the xsb term representing the command in XSB's register 1 constructed, by the caller (using the c2p * and p2p * routines). xsb_command invokes the command represented in register 1 and returns 0 if the command succeeds and 1 if it fails. In either case it resets register 1 back to a free variable. If there is an error, it returns 2.

*int xsb_command_string(char *command):* This function is similar to the above function but passes the command as a string to XSB. The string consists of a term that can be read by the XSB reader and should be terminated by a period. As discussed previously, no other query can be active and must be closed before calling this function.

int xsb_query(): This function is used for passing a query to XSB. Before calling xsb_query, any previous query should be closed. Unlike an XSB command which either fails or succeeds, an XSB query could return multiple data answers and this function is used for returning the first such data answer to the caller. xsb_next could be then used to get the consequent answers. Similar to the xsb_command routine, the calling program should construct the xsb_query in register 1 before calling this function. The answers to the query can be found in two locations, in XSB's register1 and register2. If the query has no answers (i.e., just fails), register 1 is set back to a free variable and 1 is returned. If the query has at least one answer, the variables in the query term in register 1 are bound to those answers and xsb_query returns 0. Furthermore, register 2 is bound to a term whose main functor symbol is ret/n, where n is the number of variables in the query. The main sub fields of this term are set to the variable values for the first answer. (These fields can be accessed by the functions p2c *, or the functions xsb_var *.). xsb_next could be then used to get the consequent answers.

*int xsb_query_string(char *qry):* This function is similar to the above function but passes the query as a string to XSB. The string consists of a term that can be read by the XSB reader and should be terminated by a period. As discussed previously, no other query can be active and must be closed before calling this function. Contrary to xsb_query routine, the answers to the query can be found only in register 2. There are other variants of this routine namely xsb_query_string_string which returns its answer as a string and xsb_query_string_string_b which makes it easier for non-C callers such as Visual Basic or Delphi to access XSB functionality.

*int xsb_get_last_answer_string(char *buffer, int bufferln, int *answerln):* This function is used when the buffer provided to accommodate the computed answer is not large enough. A larger buffer should be allocated and immediately after the previous failed call, the user should call this routine to retrieve the correct answer that had been saved.

int xsb_next(): This routine is used for returning answers to the calling program if the query has multiple data answers. It should be called after xsb-query and only if it returns a 0. It rebinds the query variables in the term in register 1 and rebinds the argument fields of the ret/n answer term in register 2 to reflect the next answer to the query. It returns 0 if an answer is found, and returns 1 if there are no more answers and no answer is returned. On a return of 1, the query has been closed.

int xsb_close_query(): This function is used for closing a query, before retrieving all the answers to the query. The answers that are not retrieved are not computed, as XSB is a tuple-at-a-time system. The calling program can call another query, only after the previous query has been explicitly closed using xsb_close_query or after retrieving all the answers to the previous query.

int xsb_close(): This function is used for completely closing the connection to XSB. Once the connection is closed, no other calls can be made including calls to xsb_init.

6.3 Using the Interface

6.3.1 Connecting to and disconnecting from XSB

This interface does not need to be loaded manually and also does not require the user to explicitly start and close XSB. However, the environment variables ORACLE SID and ORACLE HOME have to be set, for Oracle to access the external C procedures.

6.3.2 View Level Interface

This Oracle-XSB interface has diverse capabilities. It provides users the capability to combine queries from different database systems such as XSB and Oracle and retrieve the data. The queries in Oracle and XSB can be merged similar to nesting the queries in

Oracle. For example, when a user types in a query, the system parses the query, detects which tables belong to XSB and which belong to Oracle, then determines which columns belong to which tables, retrieves the data from the corresponding tables, and then combines and displays the results. However, the interface also provides users the capability to retrieve data from either XSB or Oracle or from both. To access the data in XSB, the view level interface translates a complex database query into a combination of one or more Prolog rules, which are then executed taking advantage of the query processing ability of the database system.

The definition of a simple join view between the two database predicates emp and dept in XSB is shown below:

Assuming the table declarations:

emp(ename,job,sal,comm,deptno).

dept(deptno,dname,loc).

the SQL statement:

SELECT empno,comm,hiredate,dname from dept,emp

generates the query,

emp(EMPNO,_,_,_,HIREDATE,_,COMM,_),dept(_,DNAME,_).

and the results on the oracle end:

Answer

****

7934	565	1-jun-1980	research
7934	565	1-jun-1980	sales
7934	565	1-jun-1980	operations
7844	_	_	research
7844	_	_	sales
7844	_	_	operations
7780	_	_	research
7780	_	_	sales
7780	_	_	operations
7834	_	15-jan-1985	research
7834	_	15-jan-1985	sales
7834	_	15-jan-1985	operations
7782	_	9-jun-1981	research
7782	_	9-jun-1981	sales
7782	_	9-jun-1981	operations
7369	_	17-dec-1980	research
7369	_	17-dec-1980	sales
7369	_	17-dec-1980	operations
7499	300	20-feb-1981	research

7499	300	20-feb-1981	sales
7499	300	20-feb-1981	operations
7521	500	5-may-1970	research
7521	500	5-may-1970	sales
7521	500	5-may-1970	operations
7566	_	2-apr-1981	research
7566	_	2-apr-1981	sales
7566	_	2-apr-1981	operations
7390	0	10-oct-1960	research
7390	0	10-oct-1960	sales
7390	0	10-oct-1960	operations

****

A more complicated example,

The SQL statement:

SELECT dname,empno,mgr,hiredate, FROM dept,emp, where deptno=20');

generates the rule,

equalto(DNAME,EMPNO,MGR,HIREDATE) :-

dept(X,DNAME,_),emp(EMPNO,_,_,MGR,HIREDATE,_,_,X), X = 20.

and the results:

```
                Answer
*****************************
research  7844
research  7369  7902    17-dec-1980
research  7566  7839     2-apr-1981
*****************************
```

As mentioned above, the Oracle-XSB interface provides users the capability to retrieve data either from XSB or Oracle or from both. The examples described above show how a user could access the data in XSB databases. However, if a user wants to retrieve the data from both Oracle and XSB environments, it can be accomplished by invoking a stored procedure as follows:

```
begin
    Select_From('SELECT empno,colmpos,colmname FROM emp,sampletable');
end;
```

```
emp(EMPNO,_,_,_,_,_,_,_).
```

is the equivalent XSB query generated, results are then computed using the query processing ability of the XSB database system, combined with the results from the Oracle environment and then displayed.

Answer

****

7934

7844

7780

7834

7782

7369

7499

7521

7566

7390

****

Oracle Results

****

1 col1

2 col2

3 col3

4 col4

5 col5

5 col5

6 col6

7 col7

8 col8

9 col9

****

A more complicated example,

```
begin
    Select_From('SELECT empno,colmpos,colmname FROM emp,sampletable WHERE
oraempno=empno');
end;
```

Answer

****

7934 roy

7844 keeth

7780 russell

7834 george

7782 clark

7369 smith

7499 allen

7521 ward

7566 jones

7390 kit

****

Oracle Results

****

1 col1

2 col2

3 col3

4 col4

5 col5

5 col5

6 col6

7 col7

8 col8

9 col9

****

The above example shows how the results from one database system such as XSB can be used to retrieve the results from another system, Oracle.

6.3.3 Insertions and deletions of rows

The Oracle-XSB interface allows the users to add new facts and rules to the database. In addition, the user can modify as well as delete the facts and rules from the database.

For example assuming the table declarations:

emp(ename,job,sal,comm,deptno).

dept(deptno,dname,loc).

To insert a new row to a table, a stored procedure has to be called which then invokes an instead of trigger as follows:

```
begin
    Insert_into('emp','#empno#ename#job#mgr#hiredate#sal#comm#deptno#');
end;
```

The SQL statement,

INSERT INTO emp VALUES(7390,'KIT','CFO',7342,'10-OCT-1960',56344,0,20);

adds a new fact to the database as emp(7390,kit,cfo,7342,10-oct-60,56344,0,20).

Before update or delete operations can be performed on the database, the corresponding stored procedures have to be called as above and they take the column name/names (on which these operations have to be performed) as their arguments.

For example, to delete a fact from a table based on a condition, the stored procedure has to be called as follows:

```
begin
    Delete_from('emp','sal');
end;
```

The SQL statement,

```
DELETE FROM emp WHERE sal = 1680;
```

then deletes the fact from the emp table.

The update operation is performed similar to the delete operation described above. For example, to update a row of a table, based on a condition, the stored procedure has to be called as follows:

```
begin
```

```
    Update_to('emp','#empno#ename#','#sal#');
end;
```

The columns based on which the row is identified and the column that has to be modified are each separated by delimiters.

The SQL statement,

```
UPDATE emp SET sal = 50000 WHERE empno = 7782 AND ename = 'CLARK';
```

then modifies the salary of employee in the emp table.

However, the condition is limited to simple comparisons in both delete and update operations.

6.3.4 Limitations

As the user can retrieve data both from Oracle and XSB database systems, the data retrieval time is considerably large when compared to the traditional relational database systems. Currently, this interface does not provide the equivalent implementations for all the clauses used for merging queries in Oracle.

7.Applications

Deductive database systems can be used in a variety of application domains like scientific modeling, financial analysis, decision support, language analysis, and parsing. They are best suited for applications in which a large amount of data must be accessed and complex queries must be supported. However, one of the significant problems with the prevalence of the deductive databases is that there is no immediately obvious way of expressing updates within a logical framework and it is necessary to be able to update and modify a database. The interface proposed above provides the users with this functionality and hence there are many potential applications to this system especially in the emerging fields of biotechnology and genetic research.

The following scenario illustrates a sample application of the Oracle-XSB interface. To predict the structure of proteins and the function of amino acid sequence, comparative analysis of protein three-dimensional structures has to be performed. Conventionally, these analyses have been performed with traditional programming languages like Fortran or with relational database management systems using SQL query language to query the data. However, both Fortran and RDBMS are limited by the inflexibility of the data storage format and the query language. In the case of Fortran, it requires considerable effort to not only code the question but also to answer the follow-up query. Conversely, the RDBMS systems are not flexible enough to represent the complex patterns of the

analyses and the naturally sequential protein structural data. Hence a much better solution is to represent the protein structures using logic-programming rules in Prolog and to use deductive databases such as XSB for storing the large data collections.

However, in order to take advantage of the benefits discussed above, the user needs to learn Prolog and even seasoned "C" or Fortran programmers usually find this a barrier. Hence the Oracle-XSB interface was developed so that the users and application programmers can access knowledge based systems just as they would access a traditional relational database system like Oracle by using the Structured Query Language (SQL) they know and love.

Another scenario examines the presence of two database systems: One a knowledge based system such as the described above and the other a traditional relational database. The Oracle-XSB interface not only provides the application programmers the capability to access each database system separately but also both the database systems simultaneously. Thus it reduces both time and effort for application programmers and improves data retrieval times in complex and heterogeneous data storage systems.

8.Future Enhancements

Since the data retrieval time is considerably large, there are plans to develop a faster implementation of the interface. Creating a virtual drive and allocating some memory in RAM to store the text file instead of the hard disk (currently used) can accomplish this. To generate more efficient queries and programs, there are also plans to provide equivalent implementations for all the clauses needed for merging queries in Oracle. Using Java external procedures instead of C as the intermediary means to communicate between Oracle and XSB also considerably speeds the data retrieval process. Java has in built packages to return the data from XSB to Oracle environment, while no such mechanism is provided in C. There are also plans to extend the interface to other databases such as Microsoft SQL Server and Postgres.

8.1 Microsoft SQL Server

Microsoft's SQL Server 2000 supports complex types of INSERT, UPDATE, and DELETE statements that reference views. INSTEAD OF triggers can be defined on views that tell SQL Server how to process a DML operation (INSERT, UPDATE, or DELETE) performed on a view and on hence on the underlying base tables. SQL stored procedures are also supported by SQL Server 2000. Hence the above design proposed for the Oracle – XSB interface could also be extended for the SQL server database with the same building blocks.

8.2 Postgres

Postgres is a database management system that extends the traditional relational data model with various additional mechanisms to support object management and knowledge management. These mechanisms include abstract data types such as user-defined operators and procedures, relation attributes of type procedure, and attribute and procedure inheritance. Hence Postgres system can be classified as an object-relational database.

Postgres also has many other features, such as query language procedures, views, rules, and triggers along with interfaces to programming languages like C. It is possible to call C functions as trigger actions and libpq is the C application programmer's interface to Postgres. Hence the proposed design of the XSB backend for Oracle could also be extended for the Postgres database using triggers to call the C procedures, which then access the XSB environment and retrieve the results.

9.Conclusion

Deductive databases are logic programming systems designed for applications with large quantities of data and offer the prospect of a more intelligent way of handling data, by applying deductive rules to capture the complexity of information. They are more powerful and sophisticated than relational systems, in that rules can be stored along with raw facts. Deductive databases have a wide range of applications and in various domains such as in data mining, data cleaning, stock market analysis, web searching, and GUI generation. However, there is a significant lack of deductive database systems developed for commercial purposes even though many powerful prototypes are available.

The goal of this project is to develop an interface system whereby non-standard knowledge systems such as XSB can make their resources available as a backend to a more traditional relational database like Oracle. It uses a novel approach to access XSB from Oracle and is completely unlike the XSB-Oracle interface provided with the XSB system in both design and deployment aspects. This interface aims to reduce the time when data needs to be retrieved from both XSB and Oracle. It is necessary to be able to update and modify a database, but there is no immediately obvious way of expressing updates within a logical framework. This interface provides a method to perform updates on the logical databases and provides equivalent of the basic relational INSERT, DELETE and UPDATE operations. However it does not handle complex queries

involving joins. It is only limited to handling queries with SELECT-FROM, and SELECT-FROM-WHERE clauses.

Bibliography

Foundations of Databases. Abiteboul, Hull, Vianu. Addison Wessley. 1995.

Fundamentals of Database Systems 3rd. Ed. ElMasri and Navathe. Addison-Wesley. 2000.

Jaffar and Mahar. Constraint Logic Programming: A survey. Journal of Logic Programming. pp503-581. 1994.

I . Niemela and P. Simons. SModels - an implementation of the stable model and well-founded semantics for normal logic programs. In Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning, volume 1265 of Lecture Notes in Artificial Intelligence, pages 420-429, Dagstuhl, Germany, July 1997.

P. Simons. Extending and Implementing the Stable Model Semantics. Doctoral dissertation. Research Report 58, Helsinki University of Technology, Helsinki, Finland, April 2000.

M . Gelfond, V . Lifschitz. The Stable Model Semantics For Logic Programming
In Proceedings of the Fifth International Conference on Logic Programming, 1988.

Online References

SMODELS. http://xsb.sourceforge.net/

The XSB Research Group. http://xsb.sourceforge.net/

The Oracle Technology Network. http://otn.oracle.com/