A Simple Interface for Nonstandard Knowledge Systems (SINKS)

CMPE 297 Report

Harini Rao

<u>Advisor:</u> Dr. Chris Pollett

1. Introduction

Deductive databases are an extension of relational databases, utilizing logic programming rules for more complex data modeling. Logic overcomes the limitations of query languages in the definition of complex views and constraints, allowing for recursive definitions.

Deductive databases are more beneficial than the traditional relational databases as they provide means for expressing disjunction and negation information. 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, 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. Unless the non-standard system is quite weak, this will generally be stronger than the relational system by itself.

For my master's project, I intend to develop an interface system whereby non-standard knowledge systems can make their resources available as a backend to a more traditional relational database. The particular deliverable I am working towards is an XSB system backend for the Oracle system. To design the interface, a strong foundation is required in logic programming and so this entire semester has been devoted to learning about nonstandard knowledge representation systems, the deductive database XSB, its various features, and the interfaces it supports.

This document is organized as follows. Section 2 introduces the basics of XSB. Section 3 gives a brief overview of deductive databases. Section 4 covers stable models and a procedure to calculate stable models while Section 5 discusses tabling feature in XSB. Section 6 takes a look at the Smodels system and illustrates a procedure for creating random Smodels knowledge bases and using the system to generate stable models. Section 7 suggests a design for the proposed Oracle –XSB interface and Section 8 concludes with the potential of the interface suggested and the future work to be done.

2. XSB

XSB is a Logic Programming and Deductive Database system for Unix and Windows. It is being developed at The Computer Science Department, Stony Brook University. It offers an alternative approach to creating a deductive database system and extends Prolog's SLD resolution in two ways: 1) adding tabling to make evaluations finite and non-redundant on datalog, and 2) adding a scheduling strategy and delay mechanisms to treat general negation efficiently. The resulting strategy is called SLG resolution.

XSB evaluates stratified queries much faster than current bottom up implementations. The most significant difference between the XSB implementation and conventional deductive database implementations is XSB's tuple-at-a-time evaluation strategy. This gives it an advantage in object-oriented applications where information can be kept in complex terms or distributed across many relations. In addition its syntax, which is based on HiLog extended with Prolog operators, offers a useful means of knowledge representation for object-based schemas.

In addition to providing all the functionality of Prolog, XSB contains several features not usually found in Logic Programming systems, including

- A variety of indexing techniques for asserted code, along with a novel transformation technique called unification factoring that can improve program speed and indexing for compiled code
- A number of interfaces to other software systems, such a C, Java, Perl and Oracle.
- Extensive pattern matching libraries, which are especially useful for Web applications.
- Preprocessors and Interpreters so that XSB can be used to evaluate programs that are based on advanced formalisms, such as extended logic programs, Generalized Annotated Programs and F-Logic.
- Source code availability for portability and extensibility.

3.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.

A deductive database system uses two main types of specifications: facts and rules. Facts are specified in a manner similar to the way relations are specified in relational databases, and rules are similar to relational views to some extent. In a deductive database system, rules are specified through a declarative language. They specify virtual relations that are not actually stored but they can be formed from the facts by applying inference mechanisms based on rule specifications. The main difference between rules and views is that rules may involve recursion and hence may yield virtual relations that cannot be defined in terms of standard relational views.

There are two main alternatives for interpreting the theoretical meaning of rules: 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. 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. In other words, the minimal model is the smallest set of atoms that characterize a positive logic program. Stable models is an analogous notion for programs that contain negation.

4.Stable Models

Stable models are one of the most popular semantics for non-stratified programs. The idea behind the stable model semantics for a ground program P can be seen as follows. Each negative literal not L in P is treated as a special kind of atom called an *assumption*. To compute the stable model, a guess is made about whether each assumption is true or false, creating an assumption set, A. Once an assumption set is given, negative literals do not need to be evaluated. Rather an evaluation treats a negative literal as an atom that succeeds or fails depending on whether it is true or false in A.

For example, consider the simple, non-stratified program

```
writes_manual(terry):- ~writes_manual(kostis),has_time(terry).
writes_manual(kostis):- ~writes_manual(terry),has_time(kostis).
has_time(terry).
has_time(kostis).
```

There are two stable models of this program: in one writes_manual(terry) is true, and in another writes_manual(kostis) is true.

The residual program for the above program is

writes_manual(terry):- ~writes_manual(kostis).
writes_manual(kostis):- ~ writes_manual(terry).
has_time(terry).
has_time(kostis).

The following program demonstrates a method for calculating the Stable Models**.**

```
/**
* StableModel.java - a simple class for generating Stable Models.
*  @author  Harini Rao
*  @version 5.0, 04/12/2002
*/

import java.io.*;

import java.util.*;

/**
*  This class contains an application that can be used to generate
*  Stable Models for a given program. It is run from the command line
*  as:
*
*   java StableModel.java text1.txt
*
*  Here text1.txt is the name of a file containing the rules of the
*  program.
*/

public class StableModel

{

/**
* Calculates StableModels for the rules of a given program which are
* stores in a file.
*
* @param args A variable array of type String.
* @exception IOException.
*
*/
public static void main (String args[]) throws IOException

{

      FileInputStream fin1;

      try
```

```
                        //Check for the existence of the file.
      {

            fin1 = new FileInputStream (args[0]);
      }

      catch (FileNotFoundException e)

      {

            System.out.println ("File Not Found");

            return;
      }


      String filename = args[0];

      char a[] = countVariables(filename);

      fin1.close();
}

/**
*  Calculates the number of the variables.
*
* @param filename A variable of type String.
* @return A char array data type.
* @exception e If the file is not present.
*
*/


public static char[] countVariables (String filename)

{

      int l = 0;

      int i;

      char a[] = new char[35] ;

      char c;

      char g;

      int j = 0;

      int k = 0;

      FileInputStream fin;

      try
      {
            fin = new FileInputStream (filename) ;
```

```
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) ;

int ft = 0;

for (k = 0; k < a.length; k++)

{
      if (a[k] != '\n')

                  if (a[k] != '\r')

                            if (a[k] != 0)

                                    l++ ;

}

j = 0;

Vector v = new Vector(20);

fin = new FileInputStream (filename);

do

{

 i = fin.read ();
```

```
            if (i != -1)

            {
                  c = (char) i;

                  if ((c >= 48 && c <= 57) || (c >= 97 && c <= 122))

                        {
                              if(!v.contains(String.valueOf(c)))

                                    v.addElement(String.valueOf(c));

                        }

            }

            }

            while (i!=-1) ;

            // Make assumptions for calculating stable model.

            for (int x = 0; x < l * l; x++)

            {
                  boolean temp[] = makeAssumption(l,x);

                  //Store the rules of the program into an array list.

                  ArrayList al = storeRules(l,filename);

                  //Calculate stable model from the rules.

                  calculateStableModel(v, al, temp, l);

            }

            return a;

      }

      catch (Exception e)

      {

      System.out.println (e);

      return null;

      }

}

/**
*  Makes assumptions to calculate stable models.
*
* @param l A variable of type Integer.
* @param i A variable of type Integer.
* @return A Boolean array data type.
*
*/


public static boolean[] makeAssumption(int l, int i)

{

      int y = 0;

      boolean temp[] = new boolean[l] ;

      for(int k = 0; k < l; k++)

      {
            int g = i%2;

            if(g==1)

                  temp[k] = true;

            else

                  temp[k] = false;

            i = i/2;
      }

      System.out.println();

      System.out.println("Assumption Made:");

      for(int rf = 0; rf < l; rf++)

      System.out.println(temp[rf]);

      return temp;

}

/**
*  Stores the rules in the file to an ArrayList.
*
* @param l A variable of type Integer.
* @param filename A variable of type String.
* @return An ArrayList data type.
* @exception e If the file is not present.
*
*/

public static ArrayList storeRules(int l, String filename)

{

      int i;
```

```
char c;

String s = new String ();

ArrayList al = new ArrayList ();

FileInputStream fin;

try

{
      fin = new FileInputStream (filename) ;

      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) ;

            return al;

      }

      catch (Exception e)

      {
            System.out.println (e) ;

            return null;

      }


}

/**
*  Calculates StableModels for the given program by applying the rules
to the assumptions.
*
* @param al A variable of type ArrayList.
* @param temp A variable array of type Boolean.
* @param l A variable of type Integer.
*
*/

public static void calculateStableModel(Vector v, ArrayList al, boolean
temp[], int l)

{

      boolean temp1[] = new boolean[l];

      boolean temp2[] = new boolean[l];

      boolean temp3[] = new boolean[l];

      for(int gr = 0;gr < l; gr++)

      temp1[gr] = temp[gr];

      int sl=0;

      ArrayList al1 = new ArrayList ();

      do
```

```
{
      if(sl >= al.size())

            break;

      if(sl > 0)

      {

            for(int pe = 0; pe < l; pe++)

            temp1[pe] = temp2[pe];

      }

      int yu = 0;

      int gk = 0;

      String s1 = String.valueOf(al.get(sl));

      int p = s1.length();

      char b[] = new char[p] ;

      //Retrieve the contents of the ArrayList

      b = s1.toCharArray() ;

      boolean t = true;

      boolean rv;

      int gy = 0;

      int ge = 0;

      /* Apply the truth values in the assumption to the rules and
      calculate the reduced program.*/

      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;

                  }

            }

      }

      for(int rt=0;rt<l;rt++)

      {
            if(rt!=ge)

             {

                  temp2[rt]=temp1[rt];

             }
      }


      sl++;

}

while(sl < l);


if(al1.size() == 0)
```

```
{
      System.out.println("This is not a Stable Model");

}
else
{
      for(int g = 0; g < al1.size();g++)
      System.out.println(al1.get(g));
      boolean change = false;
      sl = 0;
      String str = new String();
      String st = new String();
      Vector v1 = new Vector();
      for(int gy = 0; gy < l; gy++)
      {
            v1.add(String.valueOf(temp2[gy]));
      }
      /*System.out.println("Temp[2] Contents:");
      for(int gy = 0; gy < l; gy++)
      {
            System.out.println(v1.get(gy));
      }*/
      do
      {
            change = false;
            if(sl>0)
            {
                  for(int pe = 0; pe < l; pe++)
                  temp2[pe] = temp3[pe];
            }
```

```
int yu = 0;

int gk = 0;

String s1 = String.valueOf(al1.get(sl)) ;

int p = s1.length();

char b[] = new char[p] ;

//Retrieve the contents of the ArrayList

b = s1.toCharArray() ;

boolean t = true;

boolean rv;

int gy = 0;

int ge = 0;

/*Calculate the derived program from the reduced
program.*/

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;

                  }

            }

      }

      for(int rt=0;rt<l;rt++)

      {

            if(rt!=ge)

             {
                  temp3[rt]=temp2[rt];

            }

      }

      sl++;

      }

      while(sl<al1.size());

      /*Check if the assumption made initially is same as the
      result */

      int gu = 0;

      for(int ty = 0,tx = 0; ty < l && tx < l; ty++,tx++)

      {
            if (temp[ty] == temp3[tx])
```

```
                                gu++;
            }



            //Display the Stable models.

            if (gu >=1)

            {

                  System.out.println("This is a StableModel");

                  System.out.println("                      ");

                  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");

      }

}

}
```

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. Then the number of literals occurring in these rules is calculated and stored in an array. Depending on the number of literals, assumptions are made. Each of the rules in the program is read and stored in an ArrayList. 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. Then it checks to see if any more variables can be derived from the rules in the reduced program. If so, the derived program is calculated and a check with the original assumption is made. 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. This process is repeated for all the assumptions made.

5.Tabling in XSB

XSB implements a unique feature called as tabling. Tabling is one of the two ways of evaluating predicates in XSB and is also sometimes called as memoization or lemmatization. The basic idea of tabling is as follows: the same procedure call is never made twice: the first time a call is made, all the answers the system returns are remembered, and if it is ever made again, the previously computed answers are used to satisfy the later request. In XSB the programmer indicates what calls should be tabled by using a compiler directive, such as:

```
 :- table np/2.
```

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

Consider the following Prolog program

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
```

together with the query ?-ancestor(1,Y). This program has a simple, declarative meaning: Y is an ancestor of X if Y is a parent of X or if Y is a parent of an ancestor 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 ancestor by reusing partial answers to the query ancestor(1,Y).One of them is tabling, where the implementation keeps track of all calls to tabled predicates, or tabled subgoals. 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.

6. Smodels

The Smodels system is an implementation of the well-founded and stable model semantics for normal logic programs. It can be used as either as a C++-library that can be called from user programs or as a stand-alone program together with a suitable front-end. It extends the normal logic programs by adding new special rule types:

- constraint rules
- choice rules
- weight rules

Smodels programs are written using standard logic programming notation. That is, the programs are composed of atoms and inference rules. An atom represents a claim about the problem universe and it may be true or false. Inference rules are used to represent the relationships between atoms. An answer to a problem is a set of atoms, called a stable model, that tell which atoms are true. A Smodels program may have one, none, or many stable models.

Smodels system includes two modules: smodels, an efficient implementation of the stable model semantics for normal logic programs, and lparse, a front-end that transforms user programs into form that smodels understands.

Smodels works with variable-free programs that are quite cumbersome to generate by hand. Lparse adds variables to the accepted language and generates a variable-free simple logic program that can be given to smodels. Lparse also implements several other semantics (classical negation, partial stable models) by translating them into normal logic programs.

The following program creates random Smodels knowledge bases and uses the Smodels system to generate stable models.

```
/**
*  Smodels.java - a simple class for generating rules for Smodels system.
*  system.
*  @author  Harini Rao
*  @version 3.0, 04/26/2002
*/

import java.io.*;

import java.util.*;

import java.lang.*;

/**
*  This class contains an application that can be used to automatically
*  generate rules for SModels System. It generates a text file with the required number of
programs to test with the SModels system.
*
*  This application is run from the command line as:
*  java SModels
*
*   The generated text file can be checked for generating Stable Models using Smodels
*   system as:
*   lparse text4.lp | smodels
*
*/
public class Smodels

{

/**
* Automatically generates rules for a program which are then stored in a file.
*
* @param args A variable array of type String.
* @exception IOException.
*
*/
```

```
public static void main (String args[]) throws IOException

{
        char c;

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        String a[] = new String[6];

        System.out.println("Enter the number of rules, number of  variables");

        System.out.println("Enter the probabilities for a literal to exist, to be positive and
enter the number of programs:");

        int i = 0;

        int j = 0;

        do

        {

                String s1 = br.readLine();

                a[i] = s1;

                i++;

        }

        while (i < 5);

        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);

        }
```

```
int w = 0;

if (Integer.parseInt(a[0]) < Integer.parseInt(a[1]))


                w = Integer.parseInt(a[0]);

else

                w = Integer.parseInt(a[1]);

int o = Integer.parseInt(a[4]);

for ( int z = 0; z < o; z++)

{

        String filename = "text"+z+".lp";

        System.out.println("Filename:"+filename);

        FileOutputStream fout;

        fout = new FileOutputStream(filename);

        Writer out = new OutputStreamWriter(fout, "UTF8");

                for (j = 0; j < w; j++)

                {

                        String g = String.valueOf(b.get(j));

                        System.out.print(g);

                        String sfile = g;

                        System.out.print(" :- ");

                        sfile = sfile+":-";

                        int y = 0;

                        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);

        System.out.print(", "+b.get(k));

        }

        else

        {

        sfile = sfile+b.get(k);

        System.out.print(b.get(k));

        }

        y++;
```

```
                    }

                    else

                    {

                    if (y > 0)

                    {

                    sfile = sfile+", not "+b.get(k);

                    System.out.print(", not "+b.get(k));

                    }

                    else

                    {

                    sfile = sfile+"not "+b.get(k);

                    System.out.print("not "+b.get(k));

                    }

                    y++;

                    }

                    }
                    else    continue;
          }
}
sfile = sfile+".";

System.out.print(".");

System.out.println();

String sfile1 = sfile;

if (sfile.length() == 5)

{
```

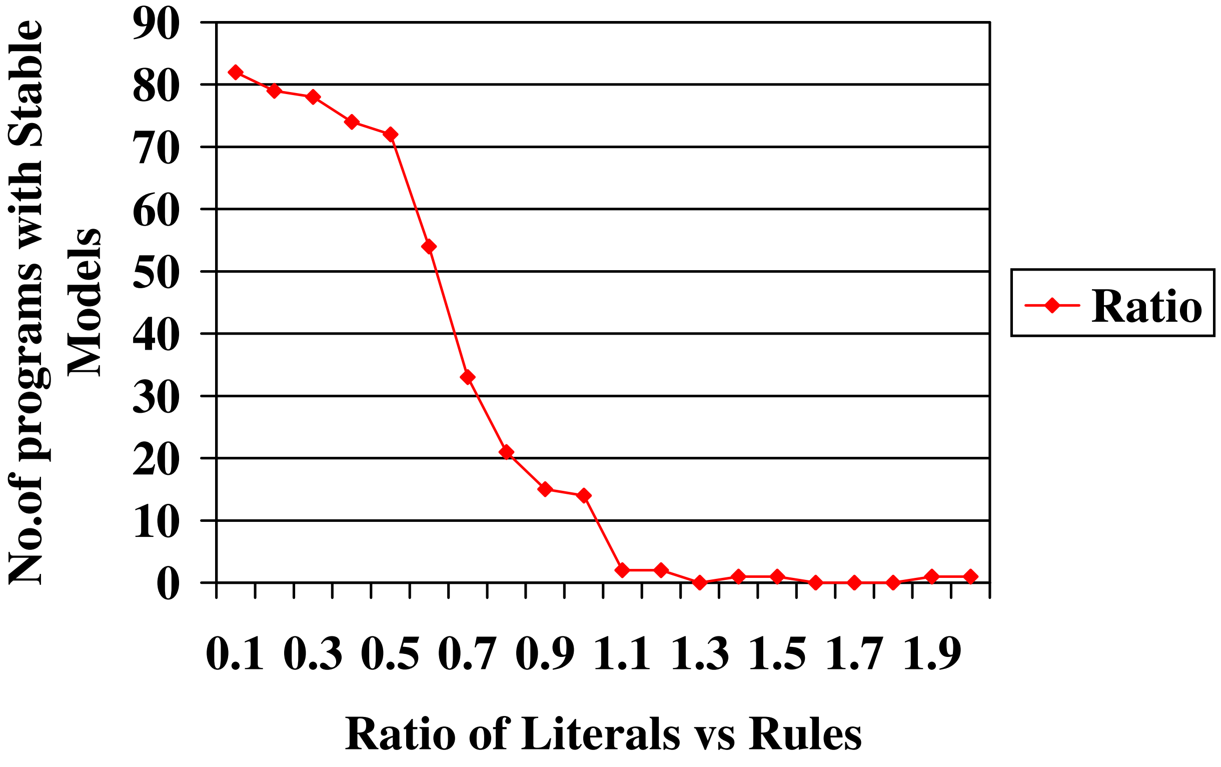
```
                                        c = sfile.charAt(0);

                                        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();
          }
}

}
```

The above program takes the following parameters from the user such as the 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 a 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.

It cycles through the variable list and generates each rule by calculating the probabilities of those variables to appear in that specific rule. When all the rules are generated, it stores this program to a text file. This process is repeated for the user-specified number of times.

The text file, which contains all the required number of programs, can be used for testing with Smodels package that generates Stable Models for the programs stored in the text file. The following paragraph examines the behavior of the above program in various scenarios and checks whether any threshold phenomena hold. The perfomance of the above program is examined mainly in the 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 3 and probability for that literal to be positive is 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 3 and probability for that literal to be positive is as 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.

7. Oracle – XSB Interface

The purpose of this project is to develop an interface between Oracle and an XSB system such that the XSB system can make its resources available as a backend to the traditional relational database, Oracle.

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. The interface involves updatable views, instead of triggers and stored procedures and so these are briefly discussed in the following paragraphs.

7.1 Updatable Views

A view is a basic schema object in an Oracle database that is an access path to the data in one or more tables. In other words, a view is a logical representation of another table or combination of tables. It derives its data from the tables on which it is based. These tables are called base tables. Base tables can be actual tables or might be views themselves. All operations performed on a view actually 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 are commonly used to establish additional levels of security for table data, derive information from base table data, and simplify application development.

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

7.2 INSTEAD OF Triggers

Triggers are procedures that execute implicitly when an INSERT, UPDATE, or DELETE statement is issued against the associated table (or, in some cases, against a view) or when database system actions occur. These procedures can be written in PL/SQL or Java and stored in the database, or they can be written as C callouts.

The 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 special type of trigger that tells Oracle how to process a DML operation (INSERT, UPDATE, or DELETE) performed on a view. These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement. The trigger performs update, insert, or delete operations directly on the underlying tables. Multitable object views can be made updatable by using an INSTEAD OF trigger--that will intercept SQL commands to update the view and take appropriate action. INSTEAD-OF triggers can also be used to make any conventional relational view updatable.

The events that fire a trigger include:

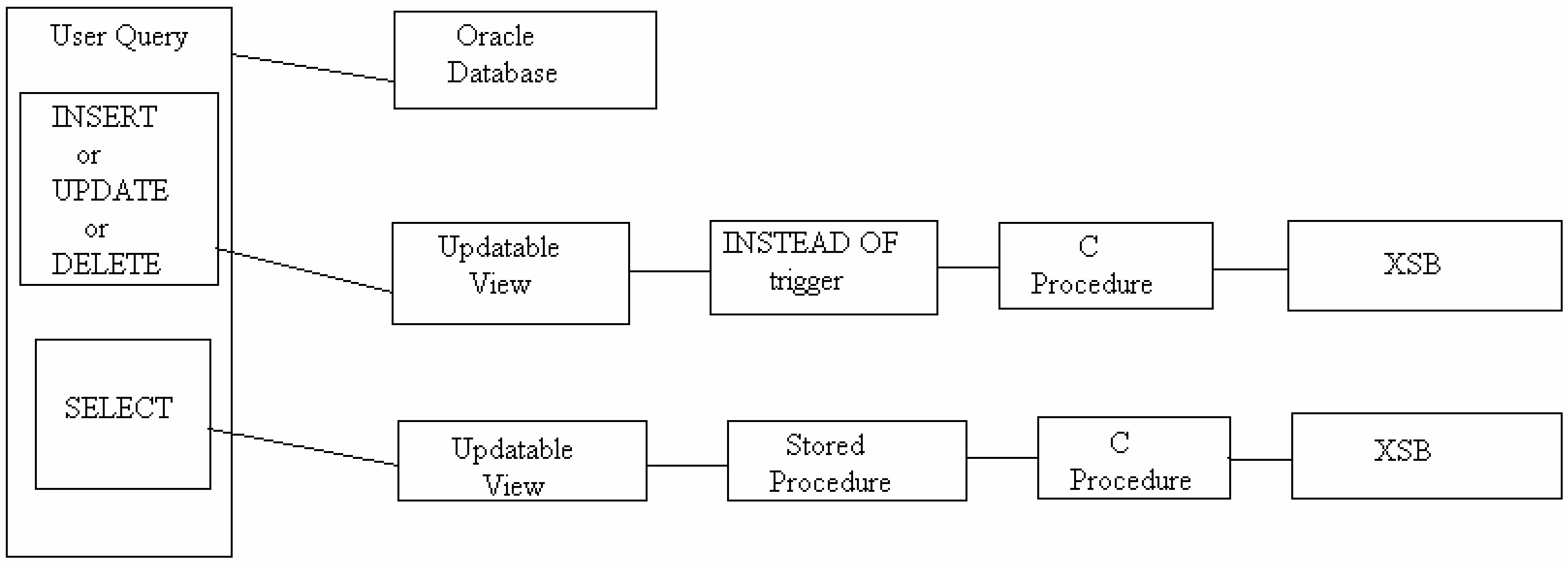
- DML statements that modify data (INSERT, UPDATE, or DELETE)
- DDL statements
- System events such as startup, shutdown, and error messages
- User events such as logon and logoff.

7.3 Stored Procedures

A procedure is a subprogram that performs a specific action. It is a subprogram that 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 that have been previously created and stored in the server database. Stored procedures accept input parameters so that a single procedure can be used over the network by several clients using different input data. Stored procedures reduce network traffic and improve performance. Additionally, stored procedures can be used to help ensure the integrity of the database.

The basic design of the Oracle-XSB interface at the conceptual level looks some thing like this:



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 using updatable views, which invoke instead of triggers, which in turn, call the C procedures, which then interact with XSB. Since triggers can only be invoked by the DML statements (INSERT, UPDATE, or DELETE) it will be not possible for the user to retrieve the data without updating it using the triggers. This can be resolved by using the stored procedures in oracle, which can be called from inside the updatable views and they interact with the C procedures similar to the triggers. 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.

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 a view to specify the individual updates that must be performed against the base tables to support the INSERT, UPDATE, or DELETE statement. Also, partitioned views support INSERT, UPDATE, and DELETE statements that modify multiple member tables referenced by the view. SQL server also supports SQL stored procedures. So the above design proposed for the Oracle – XSB interface could also be extended for the SQL Server database with the same building blocks.

Postgres

Postgres is a database management system, originally developed at the University of California at Berkeley. Views in Postgres are implemented using the rule system. The query rewrite rule system is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query optimizer for execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions.

Postgres has various client interfaces such as Perl, Tcl, Python and C. It is also possible to call C functions as trigger actions and libpq is the C application programmer's interface to Postgres.. The user can currently specify BEFORE or AFTER on INSERT, DELETE or UPDATE of a tuple as a trigger event.

Hence the proposed design of the XSB backend for Oracle could also be extended for the Postgres database using triggers in the views to call the C procedures, which interface with the XSB environment.

8. Conclusion

There are many potential applications to this system as for instance tables can be added for fuzzy or rule-based information to the relational database that could be used to limit the scope of queries and potentially improve the speed of their evaluation. The design for the interface suggested above, is still at the conceptual level. It might undergo several significant changes until the final system is developed. So the goal to be accomplished in the next semester is to elucidate and enhance the design of the interface and employ this design to develop a complete and working system. Though my master’s project is to develop XSB back end for an Oracle database, the future work would involve extending the XSB back end to all the relational databases in general.

Appendix

Preliminary Bibliography

[AHV95] Foundations of Databases. Abiteboul, Hull, Vianu. Addison Wessley. 1995.

[EN00] Fundamentals of Database Systems 3rd. Ed. ElMasri and Navathe. Addison-Wesley. 2000.

[JM94] Jaffar and Mahar. Constraint Logic Programming: A survey. Journal of Logic Programming. pp503-581. 1994.

[SN97] 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.

[S00] P. Simons. Extending and Implementing the Stable Model Semantics. Doctoral dissertation. Research Report 58, Helsinki University of Technology, Helsinki, Finland, April 2000.

Online References

[S02] SMODELS. http://xsb.sourceforge.net/

[X02] The XSB Research Group. http://xsb.sourceforge.net/