Implementation and Performance of Transaction Logic in Prolog

by

Samuel Y. K. Hung

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Implementation and Performance
of
Transaction Logic in Prolog

for the degree of Master of Science, 1996

by

Samuel Y. K. Hung

Department of Computer Science

University of Toronto

Abstract

Transaction Logic is a recently proposed logic which accounts for state changes in logic programs and databases. With this increase in functionality, it can be used in a wide class of applications such as in logic programming, graph theory, and AI.

The contributions of this thesis are twofold. First, we present a Prolog implementation of the serial-Horn fragment of Transaction Logic. We developed four prototypes, which range from simple but slow, to fast but complex. The functionality of the implementation includes queries, backtrackable updates, serial execution of transactions, bulk updates, and procedural constructs, such as iterative loops. Second, we exercise and test our implementations on various graph problems. We test several different Transaction Logic prototypes implemented on top of several different Prologs. The results effectively compare the query and update performance of different Prologs. They also show that with some Prologs, Transaction Logic programs achieve efficiency comparable to programs with destructive updates in procedural programming languages.

Acknowledgments

I would like to begin by expressing my sincere thanks to my supervisor, Anthony Bonner, for his guidance, advice, and assistance for preparing this thesis. I feel so honour to work with him. The completion of this work would not have been possible without him sharing his time, experience, and knowledge with me.

A special thank to my second reader, Alberto Mendelzon, for his time and effort on reviewing this thesis. I also appreciate the help from Eric Harley on numerous technical problems.

Last, but not least, without the lifelong encouragement and support of my parents, none of this would ever have been possible.

Contents

List of Figures

Chapter 1

Introduction

Transaction Logic (abbreviated $\mathcal{TR}$) is a recently proposed predicate logic which extends first-order logic with transaction specifications to account for update-related phenomena. It was developed by Bonner and Kifer to account for state changes in logic programs and databases [$BK95$]. $\mathcal{TR}$ has a natural model theory and a sound and complete proof theory as well as a procedural semantics for a "Horn" version. Unlike other logics, it can specify and execute procedures that update and permanently change a database or a logic program. To be more specific, transactions can be defined as logic programs because the "Horn" version of $\mathcal{TR}$ has has both a procedural and a declarative semantics.

Like procedural languages, $\mathcal{TR}$ can use sequential operators to specify transactions by combining simple actions into complex ones. It supports a wide range of functionality in different areas including database queries and updates, bulk updates, transaction definition, deterministic and non-deterministic transactions, conditions on actions, etc. With the increase in expressiveness and functionality, $\mathcal{TR}$ can be used in a wide class of applications, especially in relational and deductive databases, logic programming and AI.

As described above, like classical logic, $\mathcal{TR}$ has a "Horn" version, called serial-Horn $\mathcal{TR}$. Execution of transactions in serial-Horn $\mathcal{TR}$ can be considered as an extension of classical query processing methods. Although much research has been done on query processing in recent years, researchers in this area still do not pay much attention to the execution of updating transactions. For example, Prolog executes simple updates using the non-logical operators, "*assert*" and "*retract*", in conjunction with other logical operators.

Transaction Logic was developed as a general logic to solve a wide range of update-related problems. It can extend the theory of logic programming to include updates as queries with the maintenance of semantic consistency. In AI, it suggests a logical account of planning. By storing nodes and edges of a graph into predicates, $\mathcal{T}R$ can be used to solve a lot of graph problems, such as connected component problem, and Hamiltonian cycle problem. The capabilitity of undoing operation can be used not only in solving graph problems, but also in simulating circuit designs.

In this thesis, we present an implementation[1] of serial-Horn $\mathcal{T}R$ in Prolog. Readers are assumed to be familiar with Prolog. Four prototypes have been implemented; they range from simple but slow, to fast but complex. The programs define transaction directives of serial conjunctions, backtrackable updates and bulk updates, as well as some iteration methods in a declarative fashion. Both transaction base and database of $\mathcal{T}R$ are stored in Prolog database. Transaction rules are specially treated so that they are used in transaction queries only. In addition, database tuples are not stored directly in Prolog database. A database predicate must be declared before it is created or called. The declaration informs the system that some attributes are defined as the primary key. Indexing on predicates can also be specified in the declaration. Although our implementation was developed on top of XSB Prolog, it can also be run on other Prologs with some simple adjustments. Comparisons between the performance of the implementation in several Prolog systems, including XSB Prolog, Quintus Prolog, and Sicstus Prolog, have been made, and the tests in chapter 5 shows some of the strength and the weakness of each systems.

Thesis Outline

The thesis is organized as follows:

Chapter 2 provides an overview of Transaction Logic, general $\mathcal{T}R$ and serial-Horn $\mathcal{T}R$ programs, the model theory, and the notion of executional entailment.

Chapter 3 develops applications of $\mathcal{T}R$. They exploit features of $\mathcal{T}R$ such as transaction definition and execution, consistency maintenance, and bulk updates. We will also see how $\mathcal{T}R$ handles planning in the "*blocks world*".

Chapter 4 describes our implementation of $\mathcal{T}R$. It discusses the features of $\mathcal{T}R$ together with brief explanations of how they are implemented. The implementation includes four

[1] The implementation is available in the directory *pub/goku/transaction_logic* by anonymous ftp to *db.toronto.edu*.

distinct prototypes which range from simple but inefficient (Prototype 1), to efficient but complex (Prototype 3), as well as providing a special version for transactions that do not backtrack through updates. Although our implementation is developed in XSB Prolog, this chapter shows how to modify the prototypes for other Prologs.

Chapter 5 compares the performance of Prototype 3 and Prototype 4 of the implementations in XSB Prolog, Sicstus Prolog and Quintus Prolog. The tests include graph problems such as finding connected components, all pairs shortest paths, Hamiltonian cycle, and clique. Each test exercises different features of the implementations.

Chapter 6 concludes the work and suggests future directions for research.

Chapter 2

Transaction Logic Programming

Transaction Logic was designed as a general logic capable of solving a wide range of update-related problems. It supports a wide variety of functionality in several areas. The functionality includes database queries and views, unification and rule-base inference, updating of information, serial execution, and non-deterministism. This chapter outlines the general semantics of Transaction Logic. A complete development can be found in [$BK95$].

2.1 Overview of Transaction Logic Programming

From the user's point of view, using $\mathcal{TR}$ is very similar to using Prolog or using a relational database system. Users can specify rules, ask queries, or perform updates. Each of these transactions can be viewed as a query with possible side effects on the database. Unfortunately, updates in Prolog are not logical. If the execution fails after an update is performed, the update cannot be undone. So, although executions in Prolog can be backtracked, the database does not roll back to its initial state if updates are involved. In other words, database consistency cannot be guaranteed if transactions fail.

2.2 Syntax

Transaction Logic extends first-order logic with a single binary operator, $\otimes$, called *serial conjunction*. It is used to compose transactions and formulate queries. Informally, the formula, $a \otimes b$, means "First execute transaction a, and then execute transaction b." It has a dual operator called *serial disjunction*, $\oplus$, defined by the equivalence $a \oplus b = \neg(\neg b \otimes \neg a)$.

Formally, transaction formulas are defined recursively as follows:

1. An atomic transaction formula is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol, and $t_1, \ldots, t_n$ are function terms as in first-order logic.

2. If ϕ and ψ are transaction formulas, then so are the following expressions:

   - $\phi \vee \psi,\ \phi \wedge \psi,\ \phi \otimes \psi,\ \phi \oplus \psi, and\ \neg\phi.$
   - $(\forall\ \mathrm{X})\phi$ and $(\exists\ \mathrm{X})\phi$, where X is a variable.

2.3 Elementary Operations

Transaction Logic comes with a parameter: a pair of oracles, called the *data oracle* and the *transition oracle*, which specify elementary database operations. The data oracle specifies a set of primitive database queries, and the transaction oracle specifies a set of primitive database updates. These oracles are not fixed because any pair of oracles can be "plugged into" a $\mathcal{T}R$ theory.

The data oracle specifies the semantics of states. It is a mapping, O^d, from state identifiers to sets of first-order formulas. On the other hand, the state transition oracle, O^t, defines elementary state transitions. It is a function that maps pairs of database states into sets of ground atomic formulas, or elementary transitions. For a relational database, a state $\mathbf{D}$ is a set of ground atomic formulas, and $O^d(\mathbf{D}) = \mathbf{D}$. The insertion and deletion of atoms with predicate symbol p is represented by the predicates $p.ins$ and $p.del$, respectively. Formally, a transition oracle for these updates is defined as follows: $p.ins(x) \in O^t(\mathbf{D_1}, \mathbf{D_2})$ iff $\mathbf{D_2} = \mathbf{D_1} \cup p(x)$ and $p.del(x) \in O^t(\mathbf{D_1}, \mathbf{D_2})$ iff $\mathbf{D_2} = \mathbf{D_1} - p(x)$. More examples of oracles can be found in $[BK95]$.

2.4 Serial-Horn Transaction Logic Program

Transaction Logic has a Horn-like fragment that allows users to program transactions by rule definitions. A serial-Horn $\mathcal{T}R$ program consists of 2 parts: transaction base, and database.

- The transaction base is a finite set of serial Horn rules specifying user-defined transactions.

- The database is a set of first-order logic formulas. It is updatable by the transaction base.

A serial Horn rule is a transaction formula of the form $p \leftarrow \phi$, where p is an atomic formula and ϕ is a serial goal (a transaction formula of the form $b_1 \otimes\ b_2 \otimes\ \cdots \otimes\ b_n$, where each b_i is an atomic formula, and $n \geq 0$). As in classical logic, it is equivalent to $p \vee \neg\phi$, and it means "to execute p, it is sufficient to execute ϕ". This interpretation provides a subroutine facility for $\mathcal{TR}$, and allows us to write transaction programs. Like Horn-clause logic, in the rule $p(X_1, X_2, \ldots, X_n) \leftarrow \phi$, the symbol p acts as the transaction name, and the arguments $X_1, \ldots, X_n$ act as the input parameters, and ϕ is the transaction definition.

2.5 Model Theory

The formal semantics of $\mathcal{TR}$ is based on states and transaction execution paths. The set of states is a set of first-order classical formulas which comes from the data and transition oracle. A transaction execution path is a finite sequence of states. When a transaction is executed, the database may change, from an initial state, through some number of intermediate states, to a final state. A path is any of such finite sequence of states. In $\mathcal{TR}$, truth of an execution is defined on paths, not states. For example, $ins : a\ \otimes\ ins : b$ is not equivalent to $ins : b\ \otimes\ ins : a$ because their execution paths, $< D,\ D + \{a\},\ D + \{a, b\} >$ and $< D,\ D + \{b\},\ D + \{a, b\} >$ are not the same.

In general, if the path has length 1, then the transaction is a query; if the path has length 2, then the transaction is an elementary update; and if the path has length greater than 2, then the transaction is a composite update. The path structure distinguishes a state s from the path $< s >$ of length 1. The state s represents the formulas stored in the database, and the path $< s >$ represents formulas derived by combining the database and the transaction base. The path structure is defined here:

Definition 2.5.1 *(Path Structure)* Let L be a language of $\mathcal{TR}$ with set of function symbols F. A path structure, M, over L is a triple $< U, I_F, I_{path} >$ where U is the domain of M, I_F is an interpretation of function symbols in L, and I_{path} is a mapping that assigns a first-order semantic structure to a path.

The truth or falsity of a transaction formula is evaluated from on paths of a structure, as follows:

Definition 2.5.2 *(Satisfaction)* Let $M =< U, I_F, I_{path} >$ be a path structure, let π be an arbitrary path, and let v be a variable assignment (i.e., a mapping $V \mapsto U$, which takes a variable as input, and returns a domain element as output). Then:

1. $M, \pi \models_v p(t_1, \ldots, t_n)$ if and only if $I_{path} \models^c_v p(t_1, \ldots, t_n)$, for any atomic formula $p(t_1, \ldots, t_n)$, where "$\models^c_v$" denotes classical satisfication in first-order predicate calculus.

2. $M, \pi \models_v \neg\phi$ if and only if it is not the case that $M, \pi \models_v \phi$.

3. $M, \pi \models_v \phi \wedge \psi$ if and only if $M, \pi \models_v \phi$ and $M, \pi \models_v \psi$.

4. $M, \pi \models_v \phi \vee \psi$ if and only if $M, \pi \models_v \phi$ or $M, \pi \models_v \psi$.

5. $M, \pi \models_v \phi \otimes \psi$ if and only if $M, \pi_1 \models_v \phi$ and $M, \pi_2 \models_v \psi$ for some split[1] $\pi_1 \circ \pi_2$ of path π.

6. $M, \pi \models_v (\forall X)\phi$ if and only if $M, \pi \models_\mu \pi$ for every variable assignment μ that agrees with v everywhere except X.

7. $M, \pi \models_v (\exists X)\phi$ if and only if $M, \pi \models_\mu \pi$ for some variable assignment μ that agrees with v everywhere except X.

A model, M, of a $\mathcal{T}R$-formula ϕ is a path structure such that $M \models \phi$ if and only if $M, \pi \models \phi$ for every path π. A path structure is a model of a set of formulas if and only if it is a model of every formulas in the set. Further details of the model theory can be found in $[BK95]$.

2.6 Execution Entailment

Let P be a transaction base. Let ϕ be a transaction formula, and let D_0, D_1, $\ldots$, D_n be a sequence of databases states. Then, the following statement

$$P,\ D_0,\ D_1,\ \ldots,\ D_n \models \phi$$

[1] $\pi_1 \circ \pi_2$ is a split of path π if and only if $\pi = < \mathbf{D_1}, \ldots, \mathbf{D_n} >$, $\pi_1 = < \mathbf{D_1}, \ldots, \mathbf{D_i} >$ and $\pi_2 = < \mathbf{D_i}, \ldots, \mathbf{D_n} >$ for some i $(1 \leq i \leq n)$.

is true if and only if $M, < D_0,\ D_1,\ \ldots,\ D_n > \models\ \phi$ for every model M of P.

For any transaction base P, any sequence of database states $D_0,\ \ldots,\ D_n$, and any transaction formulas α and β, the following are properties of execution entailment:

1. If $P,\ D_0,\ \ldots,\ D_n \models\ \alpha$ and $P,\ D_0,\ \ldots,\ D_n \models\ \beta$ then $P,\ D_0,\ \ldots,\ D_n \models\ \alpha\ \vee\ \beta$.

2. If $P,\ D_0,\ \ldots,\ D_i \models\ \alpha$ and $P,\ D_i,\ \ldots,\ D_j \models\ \beta$ then $P,\ D_0,\ \ldots,\ D_j \models\ \alpha \otimes \beta$.

3. If $\alpha \leftarrow\ \beta$ is in P, $D_0,\ \ldots,\ D_n \models \beta$ then $D_0,\ \ldots,\ D_n \models \alpha$.

4. If $O^t(D_0,\ D_1) \models^c \alpha$ then $P,\ D_0,\ D_1 \models\ \alpha$.

5. If $O^d(D_0) \models^c \alpha$ then $P,\ D_0 \models\ \alpha$.

In 4 and 5, α is a first-order formula and "$\models^c$"denodes classical entailment.

Chapter 3

Applications

Since $\mathcal{TR}$ formulas can capture the features of database and knowledge-base systems, a wide variety of interesting and useful formulas in this area can be constructed in $\mathcal{TR}$. These include transaction definition and execution, ad hoc queries, consistency maintenance, bulk updates, non-determinism, sampling, etc. In this chapter, we will describe some of these applications. We shall also see how $\mathcal{TR}$ can be used in solving graph theory problems and to simulate circuits.

The examples in Section 3.1-3.4 are adapted from [$BK95$]. To start with our discussion, we shall consider some simple database transactions first.

3.1 Update Consistency

In many database systems, when updating one relation, it is often to make additional updates to other relations in order to maintain the consistency. For instance, in a banking system, when a certain amount of money is transferred from one account to another, same amount of money should be deducted from one account and added to the other. In normal logic programming, since update operations are non-backtrackable, if the execution fails after the first update, the database would be in an inconsistent state because the previous update cannot be undone. On the other hand, consistency can be maintained easily in $\mathcal{TR}$ by simple transaction formulas.

For example, suppose a school has a database of students, courses, and professors. This database includes the following base relations:

$$takes(Student, Course)$$
$$course\_size(Course, N)$$
$$courses\_taken(Student, Total)$$

Using elementary updates, we can define two procedures by which a student drops a course and a teacher is relieved of teaching a course, respectively. When a student drops a course, in order to ensure database consistency, the number of students taking the course and the total number of courses taken by the student must be decremented. This is accomplished by the following transaction rules, adapted from [$BK95$]:

$$drop(Student, Course) \leftarrow del : takes(Student, Course) \otimes decr\_class(Course) \otimes decr\_total(Student). \tag{3.1}$$

$$decr\_total(Student) \leftarrow del : courses\_taken(Student, N) \otimes N_1 \ is \ N - 1 \otimes ins : course\_taken(Student, N_1). \tag{3.2}$$

$$decr\_class(Course) \leftarrow del : course\_size(Course, N) \otimes N_1 \ is \ N - 1 \otimes ins : course\_size(Student, N_1). \tag{3.3}$$

The first rule defines the procedure for dropping a course. The last two rules define subprocedures for decrementing the number of courses taken by the student and the current course size. If either of the base predicates is missing, which causes the procedure to fail, the whole execution will be undone and will leave the database in its original state.

3.2 Circuit Design

Horn Logic can conveniently simulate physical systems without internal state such as combinational digital circuits. In such circuits, the output depends on the input only. The input of the circuit can be stored in the database, and then a logic program can compute the output.

However, for systems with internal state, such as counters and flip-flops, the output depends on the inputs and on the internal state of the systems. It is natural to simulate the internal states by the database because it can guarantee the persistence of internal states. Also, the internal states can be queried and changed by database transactions. Unfortunately, since classical logic programming and deductive databases do not provide a clean theory of updates, logic programmers cannot represent such systems in this natural

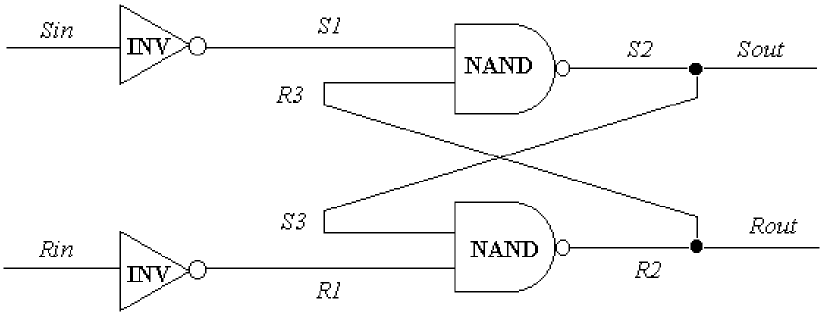


Figure 3.1: Circuit of a R-S flip-flop

way.

A digital circuit can be represented as a collection of logic gates connected by wires. Each gate has input ports and an output port. Each port is attached to a wire. Each wire is connected to two ports. There is a device called "*split*" which divides a wire into two. It has an input port and two output ports, which have same value. Using the split device, the output of one gate can be propagated to many other gates. Also, each wire has a name and a value of 0 or 1. A base predicate $val(N, V)$ means the wire with name N has the value V.

A gate is the smallest element of a circuit, and we use a predicate called *gate* to represent it. There are different versions of this predicate which have different arities to represent different numbers of input and output ports. Gates with two inputs and one output, such as and-gate, or-gates, nand-gates, etc., are represented by $gate(Type, In_1, In_2, Out)$, where *Type* is name of this kind of gate, In_1, In_2 are the names of the input ports, and *Out* is the output port. Similarly, $gate(inv, In, Out)$ represents an inverter which has one input and one output. To represent the split device, we use the predicate $split(In, Out_1, Out_2)$. The R-S flip-flop in Figure 3.1 is represented by the following database:

$gate(inv, sin, s1)$ $gate(nand, s1, r3, s2)$ $split(s2, s3, sout)$
$gate(inv, rin, r1)$ $gate(nand, r1, s3, r2)$ $split(r2, r3, rout)$
$val(sin, 0)$ $val(s1, 1)$ $val(s2, 1)$ $val(s3, 0)$ $val(sout, 0)$
$val(rin, 0)$ $val(r1, 1)$ $val(r2, 0)$ $val(r3, 1)$ $val(rout, 1)$
$dangling(sout)$ $dangling(rout)$

The S-R flip-flop has two input lines, *sin* and *rin*, and two output lines, *sout* and *rout*. The outputs of two NAND gates cross-connect to each other's inputs. Such feedback gives the flip-flop an internal state, and therefore, the outputs of the flip-flop depend on its current internal state. We assume there is just one flip-flop in the circuit, and so $sout$ and $rout$ are dangled. Initially, the flip-flop is holding the state 0, with both input lines sin and rin are low, the output line $sout$ is low, $rout$ is high.

To simulate this circuit, we first have to define the function of each gate. We use a predicate called *table* to store the truth table of the gates. For gates with two inputs, we use $table(Type, In_1, In_2, Out)$, where $Type$ is the type of the logic gate, In_1 and In_2 are the two input values, and Out is the output value. Thus, the truth table of a NAND gate is represented as follows:

$$table(nand, 0, 0, 1) \qquad table(nand, 0, 1, 1)$$
$$table(nand, 1, 0, 1) \qquad table(nand, 1, 1, 0)$$

Like the predicate gate, an inverter has one input and one output. The predicate table thus has a different arity for the inverter.

$$table(inv, 0, 1) \qquad table(inv, 1, 0)$$

The truth tables can be stored in the transaction base rather than in the database. (However, if we store the truth tables in the database, they may be corrupted by users.) To simulate the logic circuit, apart from the behavior of the gates, we also need to describe the dynamic changes of the circuit. If the input line of one gate changes, the value of the output line may change and eventually more lines will be affected. Therefore, a change in value of a line is considered as a basic event or action. The predicate $setline(Line, Val)$ is defined to set the value of a line with name $Line$ to Val. For example, $setline(S1, 1)$ sets the value of the line $S1$ to 1. Like most events, $setline$ does not just set the value of a line. It may also trigger other events, which may trigger other events, etc., leading to a cascade of triggered events. This cascade may eventually reach a stable state (when the new value and the old value of a line are the same, or the propagation ends at a dangling tuple) , or the circuit may oscillate forever. The following rules, taken from [$BK95$], capture the dynamic behavior of logic gates:

$$setline(Line, NewVal) \leftarrow val(Line, NewVal). \tag{3.4}$$

$$\begin{aligned} setline(Line, NewVal) \leftarrow\ & val(Line, OldVal) \otimes \\ & NewVal \neq OldVal \otimes \\ & del : val(Line, OldVal) \otimes ins : val(Line, NewVal) \otimes \\ & propagate(Line). \end{aligned} \tag{3.5}$$

$$propagate(In) \leftarrow dangling(In). \tag{3.6}$$

$$\begin{aligned} propagate(In) \leftarrow\ & split(In, Out_1, Out_2) \otimes val(In, Val) \otimes \\ & setline(Out_1, Val) \otimes setline(Out_2, Val). \end{aligned} \tag{3.7}$$

$$\begin{aligned} propagate(In) \leftarrow\ & gate(inv, In, Out) \otimes val(In, Val1) \otimes \\ & table(inv, Val_1, Val_2) \otimes setline(Out, Val_2). \end{aligned} \tag{3.8}$$

$$\begin{aligned} propagate(In) \leftarrow\ & [gate(Type, In_1, In_2, Out) \vee gate(Type, In_2, In_1, Out)] \otimes \\ & val(In1, Val_1) \otimes val(In_2, Val_2) \otimes \\ & table(Type, Val_1, Val_2, Val_3). \end{aligned} \tag{3.9}$$

These rules define two recursive predicates, setline and propagate. The predicate $setline(Line, NewVal)$ defines the action of setting the value of a line. *Rule* 3.4 terminates the recursion if the old value is the same as the new value. *Rule* 3.5 represents the case when the old value and the new value are different. The rule changes the value of the line, and propagates the effect of the change to other lines by invoking the predicate propagate.

Similarly, *Rule* 3.6 terminates the propagation when the line is an open end (i.e., a dangling line). *Rule* 3.7 represents the function of the split device, which divides a line into two. This rule sets the value of both output lines to the same value as the input line. The final two rules, *Rule* 3.8 and *Rule* 3.9, encode the behaviors of logic gates by consulting the corresponding truth table and setting the output line accordingly.

To illustrate the behavior of these rules, we will show their use in simulating the R-S flip-flop of *Figure 3.1.* In the example, the initial values of the input lines, *sin* and *rin*, are both 0, and the output lines, *sout* and *rout*, have values 1 and 0, respectively.

If the user issues the command $? - setline(sin, 0)$, nothing should happen because the value of *sin* is already 0. In the simulation, the event terminates immediately via *Rule* 3.4.

But, if the user issues the command $?-setline(sin,1)$, by *Rule* 3.5, the value of sin will change to 1, and a new subgoal $propagate(sin)$ is issued. The execution propagates the changes of states to $s1$, $s2$, etc. The propagation terminates at the dangling wire $sout$ and at $r3$ because the new value matches the old one. The entire simulation is summarized by the following series of subgoals, adapted from [$BK95$]:

$?-setline(sin,1)$	
$?-propagate(sin)$	(value of sin change to 1)
$?-setline(s1,0)$	
$?-propagate(s1)$	(value of s1 change to 0)
$?-setline(s1,1)$	
$?-propagate(s2)$	(value of s2 change to 1)
$?-setline(sout,1) \otimes\ setline(s3,1)$	(s2 split to sout and s3)
$?-propagate(sout) \otimes\ setline(s3,1)$	(value of sout change to 1)
$?-setline(s3,1)$	(sout dangled)
$?-propagate(s3)$	(value of s3 change to 1)
$?-setline(r3,0)$	

The last subgoal sets the value of $r3$ to 0, which is the same as the original value of $r3$, and terminates the simulation. A detailed explanation can be found in [$BK95$].

3.3 Bulk Updates

A prominent feature of database languages is bulk updates. It is a basic function to insert or delete a set of tuples in database languages such as SQL. However, such updates are not as simple in most logic programming languages. In logic programming languages, updates are typically based on the insertion and deletion of single tuples. However, database languages like SQL evaluate a query first, and then insert the resulting set of tuples into a relation, or delete the set from a relation.

Like procedural database languages, $\mathcal{TR}$ provides elementary state transitions to capture the behavior of bulk updates at the lowest level. In $\mathcal{TR}$, relational assignment, $:=$, is defined so $p(X,Y) := q(X,Y)$ copies the whole extent of q (source formula) to p (destination predicate) overwriting the contents of p, where p is a base predicate. In this example, p and q will contain the same data after the assignment. So, the transition oracle contains the following formulas:

$$p(X,Y) := q(X,Y) \in O^t(\{q(1,1), q(2,2)\},\ \{p(1,1), p(2,2), q(1,1), q(2,2)\})$$
$$p(X,Y) := q(X,Y) \in O^t(\{p(1,1), q(1,1), q(2,2)\},\ \{p(1,1), p(2,2), q(1,1), q(2,2)\})$$
$$p(X,Y) := q(X,Y) \in O^t(\{p(3,3), q(1,1), q(2,2)\},\ \{p(1,1), p(2,2), q(1,1), q(2,2)\})$$

The source of the assignment needs not necessary to be a base predicate. It can be a derived predicate or a query. For instance, suppose the database contains the following predicates and rules:

$$\begin{array}{ll} r(1) & r(3) \\ p(1,1) & p(2,2) \\ q(X,Y) :- r(X), p(X,Y) & \end{array}$$

In order to be in serial Horn $\mathcal{TR}$, the source of the assignment should be defined by a database formula, rather than by a transaction formula. Therefore, in the above example, q is defined as a derived database predicate, rather than as a transaction formula. This is indicated by the syntax of the rule for q, which uses ":-" instead of "$\leftarrow$". With the above database, $q(1,1)$ is derived from p and r. So, after the assignment of $p(X,Y) := q(X,Y)$, the relation p contains $p(1,1)$ only. As mentioned above, we can use a query instead of a predicate in the source of the assignment. For instance, the assignment $p(X,Y) := r(X),\ p(X,Y)$ can produce the same outcome as $p(X,Y) := q(X,Y)$.

Note that the destination predicate should contain no free variables. Variables in the destination predicate must also appear in the source predicate or be unified before the assignment takes place. For example, "$X = 1 \otimes\ p(X,Y) := q(X,Y)$" is equivalent to "$p(1,Y) := q(1,Y)$", which removes all entries of $p(X,Y)$ in the database, and copies all $q(1,Y)$ tuples to $p(1,Y)$. On the other hand, free variables may appear in the source. For instance, in the assignment, $r(X) := p(X,Y)$, Y is a free variable which means that for every X, there exists Y such that $p(X,Y)$ is in database. So, the following transitions are defined in transaction base:

$$r(X) := p(X,Y) \in O^t(\{p(1,1), p(2,2)\},\ \{r(1), r(2), p(1,1), p(2,2)\})$$
$$r(X) := p(X,Y) \in O^t(\{p(1,1), p(2,2), r(3)\},\ \{r(1), r(2), p(1,1), p(2,2)\})$$
$$r(X) := p(X,Y) \in O^t(\{p(1,1), p(1,2), p(2,2)\},\ \{r(1), r(2), p(1,1), p(1,2), p(2,2)\})$$
$$r(X) := p(X,Y) \in O^t(\{p(1,1), p(1,2), p(2,2), r(3)\},\ \{r(1), p(1,1), p(1,2), p(2,2)\})$$

The last two formulas show that any duplicate predicates created in the assignment are discarded.

Examples

Some complex transactions can easily be expressed by relational assignment. For instance, the transaction, "Increase the balance of all savings account by adding 5% interest, and retrieve all persons who have an account balance less than \$100", can be expressed as follows.

$$
\begin{aligned}
update(Name, NewBal, savings) &\leftarrow account(Name, Bal, savings) \otimes && (3.10)\\
&\quad NewBal is Bal \times 1.05. &&\\
update(Name, Bal, Type) &\leftarrow account(Name, Bal, Type) \otimes Type \neq savings && (3.11)\\
retrieve(Name) &\leftarrow account(N, B, T) := update(N, B, T) \otimes && (3.12)\\
&\quad account(Name, Bal, Type) \otimes &&\\
&\quad Bal < 100. &&
\end{aligned}
$$

The first two rules compute the updated content of account, which is temporarily stored in the relation update. When the transaction $? - retrieve(Name)$ is invoked, the base predicate account in the database will be modified by the assignment and the query will return the specified user names. Actually, we do not need the rules for the temporary predicate update. Instead, we can redefine *retrieve* as follows:

$$
\begin{aligned}
retrieve(N) \leftarrow\ & account(N, NewB, savings) := (account(N, B, savings), NewB is B \times 1.05) \otimes \\
& account(N, Bal, Type) \otimes Bal < 100. \qquad (3.13)
\end{aligned}
$$

With the ability to compute queries and assign the output to a base relation, this kind of generalized bulk update has most of the power of bulk updates in SQL. As a special case, bulk insertion and bulk deletion can be expressed by relational assignment.

3.4 Planning

In $\mathcal{T}R$, action-definitions and planning strategies can be defined in the transaction base. There is no need to include any special mechanisms because both types of knowledge can be represented as serial Horn rules.

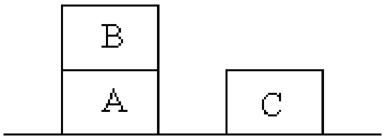
This section illustrates an approach to planning called "*script-based planning*". The presentation and examples are adapted from [BK95]. This approach describes actions and plans by a set of "*scripts*" which suggest ways to achieve goals. The line differentiating between action definition and planning strategy is fuzzy. Actions are specific and low-level, whereas plans are abstract and high-level. We will use a simple example from the blocks world to demonstrate the use of $\mathcal{TR}$ in planning problem.

The following rules define a hierarchy of planning strategies to simulate movements of a robot arm in the blocks world. There are two possible states of a block; $on(x, y)$, which says that block x is on top of block y, and $isclear(x)$, which says that nothing is on block x. These rules define four actions that can change the state of a block. Note that each action has pre-conditions. If the pre-conditions are not satisfied, the action fails.

$$
\begin{aligned}
stackTwoBlocks(X, Y, Z) &\leftarrow move(Y, Z) \otimes move(X, Y). && (3.14)\\
move(X, Y) &\leftarrow on(X, Y). && (3.15)\\
move(X, Y) &\leftarrow pickup(X) \otimes putdown(X, Y). && (3.16)\\
pickup(X) &\leftarrow isclear(X) \otimes on(X, Y) \otimes del : on(X, Y) \otimes && (3.17)\\
&\quad ins : isclear(Y).\\
putdown(X, Y) &\leftarrow X \neq Y \otimes isclear(Y) \otimes ins : on(X, Y) \otimes && (3.18)\\
&\quad del : isclear(Y).
\end{aligned}
$$

The most basic robot actions, $pickup(X)$ and $putdown(X, Y)$, mean "pick up block X" and "put down block X on top of block Y", respectively. They are defined in terms of the elementary functions ins and del, which change the state of the blocks. The remaining rules combine these simple actions into complex ones. For example, $move(X, Y)$ means "move block X to the top of block Y" which is defined as "pick up block X, and put block X down on top of block Y"; $stackTwoBlocks(X, Y, Z)$ means "stack up block X on top of block Y and block Y on the top of block Z".

The planning problem is fairly simple. It assumes all blocks that are picked up are clear. If the blocks involved are not clear, the action fails. Consider the following example:



Database:	$on(blkB, blkA)$	$isclear(blkB)$
	$on(blkA, table1)$	$isclear(blkC)$
	$on(blkC, table2)$	

The command "$move(blkA, blkC)$" fails, as shown by the following trace of subgoals during execution:

$?- move(blkA, blkC)$

$?- pickup(blkA) \otimes\ putdown(blkA, blkC)$

$?- isclear(blkA) \otimes\ on(blkA, X) \otimes\ del : on(blkA, X) \otimes\ ins : isclear(X) \otimes$
$putdown(blkA, blkC)$

Since block A is not clear, the execution fails. However, the command $?- move(blkB, blkC)$ succeeds because both blocks are clear. The actual sequence of subgoals during execution is:

$?- move(blkB, blkC)$

$?- pickup(blkB) \otimes\ putdown(blkB, blkC)$

$?- isclear(blkB) \otimes\ on(blkB, X) \otimes\ del : on(blkB, X) \otimes\ ins : isclear(X) \otimes\ putdown(blkB, blkC)$

$?- on(blkB, blkA) \otimes\ del : on(blkB, blkA) \otimes\ ins(isclear(blkA) \otimes\ putdown(blkB, blkC)$

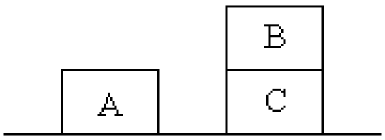
(X is unified with blkA)

$?- putdown(blkB, blkC)$ (on(blkB,blkA) is deleted and isclear(blkA) is inserted)

$?- blkB \neq\ blkC \otimes\ isclear(blkC) \otimes\ ins : on(blkB, blkC) \otimes\ del : isclear(blkC)$

!! succeed

All subgoals in the last step succeed. The state of the blocks is now:



Database:	$on(blkB, blkC)$	$isclear(blkA)$
	$on(blkA, table1)$	$isclear(blkB)$
	$on(blkC, table2)$	

The example above is too simplistic for many AI applications. It is not difficult to modify the example so that the robot can plan to clear a block before picking it up. However, some extra actions are needed to provide this flexibility. We introduce the predicate $achieve\_clear(X)$, which tries to clear the top of block X. Two predicates, $achieve\_stack(X, Y, Z)$ and $achieve\_on(X, Y)$, are defined using $achieve\_clear$. The former predicate stacks up blocks X, Y, Z even if their tops are not clear initially. The latter predicate finds and executes a plan to put block X on the top of block Y. With these meanings, $achieve\_clear$, $achieve\_stack$ and $achieve\_on$ can be defined as follows [$BK95$]:

$$achieve\_stack(X, Y, Z) \leftarrow achieve\_on(Y, Z) \otimes achieve\_on(X, Y). \tag{3.19}$$

$$achieve\_on(X, Y) \leftarrow achieve\_clear(X) \otimes achieve\_clear(Y) \otimes move(X, Y). \tag{3.20}$$

$$achieve\_clear(X) \leftarrow isclear(X). \tag{3.21}$$

$$achieve\_clear(X) \leftarrow on(Y, X) \otimes achieve\_clear(Y) \otimes clear(Z) \otimes Z \neq Y \otimes move(Y, Z). \tag{3.22}$$

The above rules are straightforward. For instance, in the last rule, in order to clear block X, anything on the top of block X is moved somewhere that is clear. However, this intuitive definition is not completely correct since achieving some subgoals may destroy previously achieved subgoals. For instance, in the rule for $achieve\_stack(X, Y, Z)$, after successfully putting block Y on top of block Z, the subgoal $achieve\_on(X, Y)$ may undo the effect of $achieve\_on(Y, Z)$, and so, after finishing execution, block Y may not be on the top of block Z. To overcome this problem, the postcondition $on(Y, Z)$ should be included in the rule as follows [$BK95$]:

$$achieve\_stack(X, Y, Z) \leftarrow achieve\_on(Y, Z) \otimes achieve\_on(X, Y) \otimes on(Y, Z). \tag{3.23}$$

So, after the first two subgoals, if block Y is not on the top of block Z, the system will backtrack and try other possible ways to achieve the stack. Note that the postcondition $on(X, Y)$ can be omitted because it is guaranteed by the subgoal $achieve\_on(X, Y)$.

The rest of this section describes another, more efficient way to avoid subgoal interaction, as proposed in [$BK95$]. Consider the second rule $achieve\_on(X, Y)$, the subgoal

$move(X, Y)$ guarantees that both block X and block Y are clear, and will backtrack if either of them is not clear. However, it is possible to avoid too much backtracking, and therefore, increase efficiency. Once a block is clear, we can lock the block to avoid putting anything on top of it again. This is achieved by introducing a base predicate $locked\_clear(X)$ which means the atom $clear(X)$ is locked. We can lock and unlock block X by inserting and deleting $locked\_clear(X)$ from the database. The modified rules for $achieve\_on$ are as follows [$BK95$]:

$$achieve\_on(X,Y) \leftarrow on(X,Y). \tag{3.24}$$

$$\begin{aligned} achieve\_on(X,Y) \leftarrow\ & \neg\, on(X,Y) \\ & achieve\_clear(X) \otimes\ ins: locked\_clear(X) \otimes \\ & achieve\_clear(Y) \otimes\ del: locked\_clear(X) \otimes \\ & move(X,Y). \end{aligned} \tag{3.25}$$

We also have to modify the definition of putdown to ensure the robot does not put a block onto a locked block [$BK95$].

$$\begin{aligned} putdown(X,Y) \leftarrow\ & clear(Y) \otimes\ X \neq\ Y \otimes\ ins: on(X,Y)) \otimes \\ & \neg locked\_clear(Y) \otimes\ del: clear(Y). \end{aligned} \tag{3.26}$$

In this way, once a block is cleared and locked, no other block can be put on top of it until it is unlocked. Unfortunately, However, this program is very inefficient for some anomalous situations, in which the stacking is obviously impossible. For example, the command $?- stackTwoBlocks(blkA, blkB, blkA)$ is clearly impossible in any blocks world, but the system is not clever enough to figure this out, and needs to search all possible moves exhaustively. There are many other possible optimizations, which we leave for future work.

3.5 Graph Algorithms

In computer science, engineering, mathematics, and many other areas, problems are represented by the relationships among data. Graphs are often used to model such relationships. Most graph algorithms are presented in procedural programming style, and they are difficult to translate into logic programs. As described earlier, $\mathcal{TR}$ integrates logic programming and procedural programming, so we can use it to implement graph algorithms conveniently. In this section, we shall see how algorithms for finding connected components, all pairs shortest path, Hamiltonian cycle, and maximum clique can be implemented in $\mathcal{TR}$. In these problems, the graph is stored in the database as two relations: $node(X)$ and $edge(X,Y)$. The former means that X is a node, and the latter means there is a directed edge from node X to node Y in the graph.

3.5.1 Polynomial Time Algorithms

Transaction Logic can express loops from procedural programming, which allows simple translation of an algorithm to a $\mathcal{TR}$ program. The following examples show the simple use of while loops and for loops in $\mathcal{TR}$ program. Detailed implementation are shown in section 4.3.8.

1. $forall\,[X,Y]\ in\ p(X,Y)\ do\ ins:q(X,Y)$
   select all possible values of X and Y such that $p(X,Y)$ is true, and insert $q(X,Y)$ into the database.

2. $with\,[Y]\ forall\,[X]\ in\ p(X,Y)\ do\ ins:q(X,Y)$
   with the value of Y imported from outside the loop, select all possible values of X such that $p(X,Y)$ is true, and insert $q(X,Y)$ into database.

3. $while\ p(X,Y)\ do\ q(X,Y)$
   call $q(X,Y)$ until $p(X,Y)$ is no longer true for any X or Y.

4. $with\,[Y]\ while\ p(X,Y)\ do\ q(X,Y)$
   with the value of Y imported from outside the loop, call $q(X,Y)$ until $p(X,Y)$ is no longer true for any X or Y.

The following examples show the use of $\mathcal{TR}$ in two P-time algorithms.

Example 1: Connected Components

A connected component of a graph G is a maximal connected subgraph. One method of finding a connected component is to pick an arbitrary vertex and then use depth first search to reach all vertices in the same component. This process continues until all vertices have been visited.

The following is a simple algorithm of finding connected component by depth first search:

findComponent:

mark all vertices v as unvisited;

for each vertex v do;

if v is not visited then dfs(v,v)

(i.e., start searching a new component with name v)

dfs(c,v): (c is the component name, and v is a vertex in the component)

record that node v is in component c;

mark v as visited;

for each unvisited vertex w such that edge(v,w) do

dfs(c,w)

A sample $\mathcal{TR}$ program for the above algorithm is:

$$\begin{aligned} findComponents &\leftarrow unvisited(P) := node(P) \otimes \\ &\quad\; while\ unvisited(V)\ do\ dfs(V,V). \end{aligned} \tag{3.27}$$

$$\begin{aligned} dfs(C,V) &\leftarrow ins : component(C,V) \otimes \\ &\quad\; del : unvisited(V) \otimes \\ &\quad\; with\ [C,V]\ while\ (edge(V,W),\ unvisited(W))\ do\ dfs(C,W). \end{aligned} \tag{3.28}$$

Note that the $\mathcal{TR}$ program is almost a direct translation of the algorithm.

Example 2: All-Pairs Shortest Paths

Another famous P-time problem is the all-pairs shortest paths program. In this problem, we have to construct a table that gives the shortest distance from any node to any other node in a graph.

Dijkstra's algorithm uses a "greedy" technique to solve the shortest path problem. It keeps a set of vertices whose shortest distance from the source is already known. At each step, it adds to the set a vertex with the next shortest distance to the source. Since the algorithm visits the vertices radially from the source, it eventually finds the shortest distance from the source to each vertex.

In an unweighted graph, Dijkstra's algorithm is equivalent to a breadth first search of the graph starting from the source. We can solve the all-pairs shortest path problem on such graphs by doing a breadth first search from each vertex in the graph. The rule bfs in the following program implements a modification of Dijkstra's algorithm. Starting from a source, v, it repeatedly finds all the unvisited neighbors of the "wave front", using the predicate $newfront$. A wave front is a list of vertices in the next level of search. In an unweighted graph, all vertices in the same wave front have the same distance from the source. For example, the first level of wave front from v is the neighbors of v (i.e., with distance 1 from v), and the following level of wave front is the neighbors of the previous wave front (i.e., with distance 2 from v). At each level of search, the program stores the distances from v to the nodes in the wave front are stored.

apsp:

> *for each vertex v do bfs(v)*

bfs(v):

> *record v as the current wavefront list;*
> *D = 0;*
> *assign distance from v to v to 0;*
> *repeat until the current wavefront list is empty do*
>
> > *repeat the wavefront list with all vertices in the next level*
> > *(i.e., unvisited neighbors of the vertices in the current wavefront list)*
> > *D := D+1;*
> > *assign the distances D from v to the vertices in wavefront list*

Here is the corresponding $\mathcal{TR}$ program:

$$apsp \leftarrow ins : tempD(0) \otimes \tag{3.29}$$
$$for\ [V]\ in\ node(V)\ do\ bfs(V).$$
$$bfs(V) \leftarrow front(X) := front(V) \otimes \tag{3.30}$$
$$del : tempD(\_) \otimes\ ins : tempD(0) \otimes$$
$$ins : dist(V, V, 0) \otimes$$
$$with\ [V]\ while\ newFront(V, \_)\ do$$
$$($$

$$
\begin{aligned}
&del : tempD(T) \\
&D \; is \; T + 1 \otimes \\
&ins : tempD(D) \otimes \\
&front(M) := newFront(V, M) \otimes \\
&dist(V, N, D) := (front(N), dist(X, Y, Z))
\end{aligned}
$$

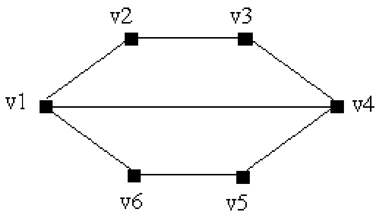
).

We also have to define the rule of $newFront(R, N)$. A vertex N belongs to the new wave front if it is unvisited and is a neighbor of one of the front vertices. A vertex is visited if its distance from the source has been calculated:

$$visited(R, N) \quad :- \quad dist(R, N, \_). \tag{3.31}$$

$$newFront(R, N) \quad :- \quad front(M), edge(M, N), not(visited(R, N)). \tag{3.32}$$

Since the rule newFront is involved in a relational assignment in the $\mathcal{TR}$ program, and relational assignment is defined for database predicates only, the above rules should be defined in the database instead of in the transaction base. We will explain the $\mathcal{TR}$ program on the following graph:



Starting with the source $v1$, it has distance 0 to $v1$ itself, and it is the only vertex in the wave front. The predicate newfront is defined as unvisited neighbors of the vertices in the wave front. In the first iteration, the neighbors of $v1$, $(v2, v4, v6)$, are found, and inserted into the predicate front as the next front wave. Also, all these nodes have distance 1 from the source. In the next iteration, the program finds the unvisited neighbors of $(v2, v4, v6)$, which are $(v3, v5)$. The vertices $v3$ and $v5$ have distance 2 from the source. Since there is no more unvisited vertices, the program terminates.

The database evolves as follows:

$Before\ iteration:$	$After\ first\ iteration:$	$After\ second\ iteration:$
$dist(v1, v1, 0)$	$dist(v1, v1, 0)$	$dist(v1, v1, 0)$
$front(v1)$	$dist(v1, v2, 1)$	$dist(v1, v2, 1)$
$tempD(0)$	$dist(v1, v4, 1)$	$dist(v1, v4, 1)$
	$dist(v1, v6, 1)$	$dist(v1, v6, 1)$
	$front(v2)$	$dist(v1, v3, 2)$
	$front(v4)$	$dist(v1, v5, 2)$
	$front(v6)$	$front(v3)$
	$tempD(1)$	$front(v5)$
		$tempD(2)$

3.5.2 NP-Complete Problems

The class NP has the characteristic that all known algorithms for solving NP-complete problems run in exponential time. Typically, these algorithms search a space of exponential size until a solution is found. Since $\mathcal{TR}$ program allows backtrackable updates, formulating such search problems is relatively easy in $\mathcal{TR}$.

Example 1: Hamiltonian Cycles

A Hamiltonian path is a path which visits each vertex in a graph exactly once. Similarly, a Hamiltonian cycle is a Hamiltonian path together with an edge connecting the first and the last vertices on the path.

One way to find Hamiltonian paths is to use depth first search to exhaustively check all possible paths until a Hamiltonian path is found. If there is an edge connecting the two ends, the path, together with the edge, is a Hamiltonian cycle.

$$Transaction\ Program:$$

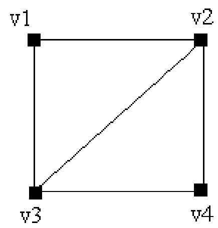
$$\begin{aligned} findhamo \quad \leftarrow \quad & unvisited(P) := node(P) \otimes \\ & del : unvisited(X) \otimes \\ & searchpath(X, X). \end{aligned} \tag{3.33}$$

$$\begin{aligned} searchpath(S,X) &\leftarrow edge(X,Y) \otimes \\ & del: unvisited(Y) \otimes \\ & ins: cycle(X,Y) \otimes \\ & searchpath(S,Y). \end{aligned} \tag{3.34}$$

$$\begin{aligned} searchpath(S,X) &\leftarrow \neg\, uninvisited(Y) \otimes \\ & edge(X,S) \otimes \\ & ins: cycle(X,S) \end{aligned} \tag{3.35}$$

The predicate *findhamo* first marks all nodes as unvisited. Then, it chooses a node as the start node and repeatedly searches for a Hamiltonian path. *Rule* 3.34 searches for an unvisited neighbor, adds the edge into the cycle, and then continues the search starting from the neighbor. *Rule* 3.35 defines the termination condition to be when all nodes are visited (which means a Hamiltonian path has been found), and there is an edge connecting the last node and the starting node (which converts the path to a cycle).

We use the following graph to illustrate the execution of the program:



1. Suppose, $v1$ is the starting node. *Rule* 3.33 marks all nodes as unvisited except $v1$. So, in addition to the graph itself, the database now contains:

   unvisited(v2), unvisited(v3), unvisited(v4)

2. Using *Rule* 3.34, the system selects $v2$ as the next node and stores $(v1, v2)$ as the first edge on the Hamiltonian cycle. In subsequent calls to *Rule* 3.34, edges $(v2, v3)$ and $(v3, v4)$ are chosen. The database now contains:

   cycle(v1,v2), cycle(v2,v3), cycle(v3,v4)

3. Since there are no unvisited adjacent nodes from $v4$, *Rule* 3.35 tests the termination condition. However, since there is no edge between $v4$ and $v1$, the termination condition fails and the program backtracks.

4. The program backtracks until $v3$ and $v4$ are unvisited. At this time, the database contains:

   unvisited(v3), unvisited(v4), cycle(v1,v2)

5. In *Rule* 3.34, rather than choosing $edge(v2, v3)$, the program unifies $edge(X, Y)$ to $edge(v2, v4)$, which becomes an edge of the growing path. In the next step, *Rule* 3.34 chooses the $edge(v4, v3)$ as the next edge of the path, and the database becomes:

   cycle(v1,v2), cycle(v2,v4), cycle(v4,v3)

6. Finally, the program terminates by choosing $edge(v3, v1)$ as the final edge in the cycle. Therefore, the Hamiltonian cycle is: $(v1, v2, v4, v3, v1)$.

Example 2: Clique

A $k - clique$ is a set of k vertices, each pair of which is connected by an edge, i.e., it is a complete graph with k nodes. It is a NP-complete problem to find a $k - clique$ in a graph, where k is part of the input to the problem.

Since a clique is a complete graph, there are $k(k-1)$ directed edges in a clique with k vertices. The following $\mathcal{TR}$ program specifies an algorithm that determines whether there is a *k-clique* in a graph:

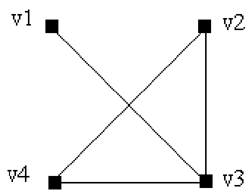
$$
\begin{aligned}
cliqueSize(K) \quad &\leftarrow \quad node(N) \otimes && (3.36)\\
&\qquad candidate(CN) := edge(N, CN) \otimes \\
&\qquad buildClique(K, N). \\
buildClique(1, \_). & && (3.37)\\
buildClique(K, N1) \quad &\leftarrow \quad K > 1 \otimes && (3.38)\\
&\qquad candidate(CN) := newCandidate(N1, CN) \otimes \\
&\qquad candidate(N2) \otimes
\end{aligned}
$$

$$K1\ is\ K-1 \otimes$$
$$buildClique(K1, N2).$$

Database rule:

$$newCandidate(N, CN) \quad :- \quad candidate(CN), edge(N, CN). \tag{3.39}$$

The predicate $cliqueSize$ arbitrarily picks a node N and marks all neighbor nodes as candidates for a clique containing node N. With this set of candidate nodes, the program starts to build a k-clique. In general, a candidate set holds those nodes which are neighbors of *all* previously visited nodes. The predicate $newCandidate(N, CN)$ is defined to find a new candidate node CN, which is the neighbor of N, from the current candidate set. Using $newCandidate$, the predicate $buildClique(K, N)$ builds the k-clique by shrinking the candidate set. Since all nodes in the candidate set are neighbors of the input node, N, if there is a k-clique in the original candidate set, there will be a $(k-1)$-clique in the new candidate set. If such clique is not found, it backtracks and starts with other nodes.



For example, in the above graph, the transaction query $?-cliqueSize(3)$ should hold because the graph has a clique of size 3, $(v2, v3, v4)$. The following shows the candidate sets and the subgoals of $buildClique$ made during the execution:

1. Suppose $v1$ is picked as the first node. Then the candidate set is initially: $\{v2\}$, and a subgoal $buildClique(3, v1)$, is invoked.

2. During the execution of the subgoal $buildClique(3, v1)$, the new candidate set is still $\{v2\}$ and the subgoal $buildClique(2, v2)$ is invoked.

3. Node v2 is the only atom left in the candidate set. Since the new candidate set is empty, $buildClique(2, v2)$ fails.

4. Since there is no alternative choice in $buildClique$, the process backtracks to the initial

state and picks $v2$ in *Rule* 3.36. There are two adjacent nodes and the candidate set now contains $\{v3, v4\}$.

5. Suppose $v3$ is chosen in the candidate set, the next subgoal is $buildClique(2, v3)$. Its adjacent node $v4$ is also in the candidate set and so the new candidate set becomes $\{v4\}$ and a new subgoal $buildClique(1, v4)$ is invoked.

6. The subgoal $buildClique(1, v4)$ succeeds immediately, and so the query $cliqueSize(3)$ succeeds.

As mentioned above, a candidate set holds the nodes which are neighbors of all previously visited nodes. For example, in step 4 above, the candidate set $\{v3, v4\}$ is the set of neighbors of $v2$. In step 5, the candidate set $\{v4\}$ is the joint neighbor set of previously visited, nodes $v3$ and $v2$. Therefore, these three nodes are vertices in the *3-clique*.

The complexity of $cliqueSize(K)$ grows exponentially in the number of nodes for an arbitrary graph and value K. However, if K is fixed, since the depth of recursive calls to $cliqueSize$ is at most K, the complexity of the program becomes polynomial with degree K.

Chapter 4

Implementation

In this chapter, we describe our implementation of $\mathcal{TR}$. The basic difference between Prolog and $\mathcal{TR}$ is the updates to the database. Prolog has a backtracking mechanism for queries only, while $\mathcal{TR}$ has a backtracking mechanism for updates too. In Prolog, updates are not undone during backtracking. For instance, the Prolog command "$assert(p(1))$, $fail$" asserts $p(1)$ into the database even though the execution fails. In $\mathcal{TR}$, as described in Section 2.1, if the query fails, the database is rolled back to its original state. In addition, features from procedural programming, like while-loop, for-loop, and if-then-else statements, can be expressed logically in $\mathcal{TR}$.

A $\mathcal{TR}$ program is a file containing a set of transaction rules. When it is loaded into our program, all rules pass through the interpreter, which translates transaction rules into Prolog rules, and are then asserted into the Prolog database. Four $\mathcal{TR}$ prototypes have been implemented, each testing a different implementation architecture. Prototype 1 is conceptually simple, while Prototype 3 is more complex and efficient. Since many applications do not require backtracking, Prototype 4 was developed as a special version of Prototype 3. It handles transactions without backtracking to increase speed.

The prototypes were designed to be used with XSB Prolog, because of its efficiency in handling large databases. However, with some simple adjustments, it runs on Quintus Prolog, Sicstus Prolog and C-Prolog.

4.1 Program Overview

4.1.1 Data Objects

Since the $\mathcal{T}R$ implementation is written in Prolog, its syntax for rules is similar to Prolog syntax. A term is either a constant, variable, or a compound term. Constants are either numbers or atoms. In most Prolog language, atoms usually have one of the following forms:

- Any sequence of alphanumeric characters (including '_'), starting with a lowercase letter.
- Any sequence from the following set of characters: + − * / \ ˆ < > = ' : . ? @ # &
- Any sequence of characters delimited by single quotes.
- Any of: ! ; []

Variables may be written as any sequence of alphanumeric characters (including '_') starting with either a capital letter or '_'.

The structured data objects of the language are compound terms. A compound term comprises a functor and a sequence of one or more terms called arguments. A functor is characterized by its name, which is an atom, and its arity, or number of arguments. Usually, when we need to refer explicitly to a functor, we will denote it by the form $Name/Arity$. For example, $p/2$ represents the predicate $p(X,Y)$.

4.1.2 Rules and Database Tuples

Unlike normal logic programming, there are two types of storage areas for $\mathcal{T}R$ rules: the transaction base and the database. The transaction base defines a set of transaction rules, while database stores a set of database rules and database tuples (ground atomic formulas).

An atomic formula is an expression of the form $p(t1, \ldots, tn)$, where p is a predicate symbol, and $t1, \ldots, tn$ are terms. In the implementation, a transaction formula is an atomic formula or a formula of one of the following forms:

- if ϕ and ψ are transaction formulas, so is $(\phi\ sc\ \psi)$. The infix operator, sc, called serial conjunction is used to compose transactions and formulate queries. In the formal theory of $\mathcal{T}R$ [BK95], the symbol $\otimes$ is used instead of sc.

- if ϕ is a transaction formula, α is a database clause, L is a list of variable names, then $(while\ \alpha\ do\ \phi)$ and $(with\ L\ while\ \alpha\ do\ \phi)$ are transaction formulas.
- if ϕ is a transaction formula, α is a database clause, and L_1, L_2 are lists of variable names, then $(forall\ (unique, possible)\ L_1\ in\ \alpha\ do\ \phi)$ and $(with\ L_2\ forall\ (unique,$ $possible)\ L_1\ in\ \alpha\ do\ \phi)$ are transaction formulas, where the flags *unique* and *possible* are optional.
- if ϕ and ψ are transaction formulas, α is a database clause, then $(if\ \alpha\ then\ \phi\ else\ \psi)$ is a transaction formula.

A database clause is either an atomic formula or a formula of the form ϕ, β where α and β are database clauses.

1. Transaction Rules:

   A transaction rule is either of the following:

   (a) an atomic formula ; or

   (b) $\phi \ \leftarrow \ \psi$, where ϕ is an atomic formula and ψ is a transaction formula.

2. Database Rules:

   A database rule is a normal Prolog rule, which has one of the following forms:

   (a) an atomic formula; or

   (b) $\phi\ :-\alpha$, where ϕ is an atomic formula and α is a database clause.

3. Database Tuples:

   During the execution of a $\mathcal{TR}$ program, database updates are backtrackable. Database tuples are implemented differently from database rules. They are not stored directly in the database; rather, to ensure backtrackability, they are specially treated. In the first place, the specification of a tuple should be informed by the a declare statement before the tuple is inserted. The declare statement contains the predicate name and arity, the number of arguments in the primary key, and an indexing specification. With this information, a tuple is broken down into two parts, the key of the tuple, and the value denoted by the key. For instance, *declare age/2 key 1* declares a tuple named age which has two arguments and the first one is the primary key. With this

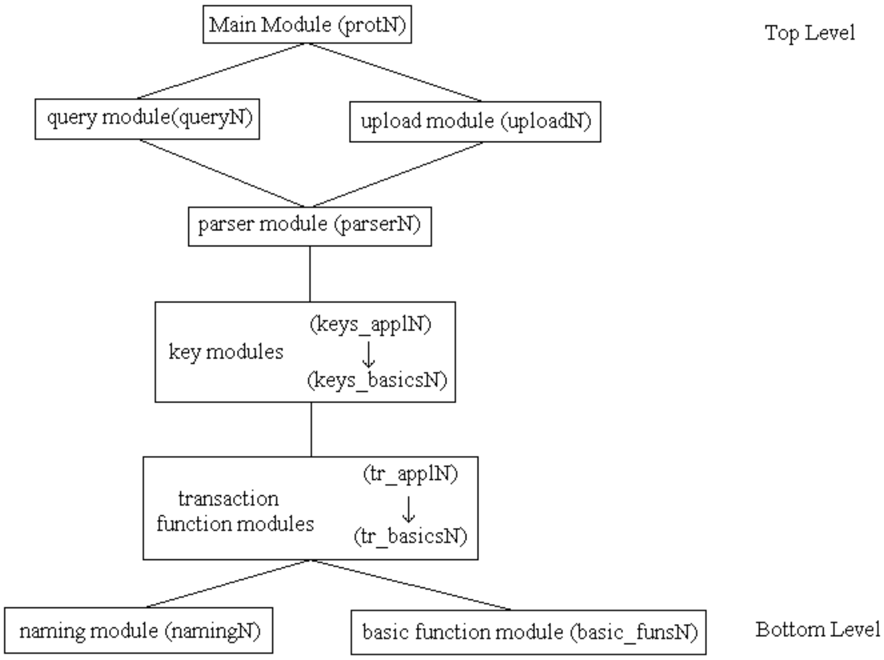


Figure 4.1: Hierarchy of Modules

declaration, the tuple $age(sam, 24)$ in database would be represented by storage of two base predicates, $key\_age(sam)$ and $value\_age(sam, 24)$, in database. More details of tuples and keys will be discussed in Section 4.3.

4.2 Modules

The implementation consists of 10 Prolog modules: a main module ($protN$), a parsing module ($parserN$), a loading module ($uploadN$), query module ($queryN$), two modules defining keys ($keys\_basicsN$ and $keys\_applN$), two modules defining elementary updates ($tr\_basicsN$ and tr_applN), a module for converting predicate names ($namingN$), and a module containing basic functions ($basic\_funsN$), where N is a number, 1, 2, 3, or 4, that represents the prototype number (to be discussed in Section 4.4). For instance, $prot3$ is a program for Prototype 3, and $upload1$ is a program for Prototype 1. Figure 4.2 shows a hierarchy of the modules. In this section, we give a brief description of the contents of each module.

4.2.1 Main Module (*protN*)

To start the $\mathcal{TR}$ implementation, users consult this main module, and then run the predicate *init/1*, which loads up all other modules and initializes their parameters.

4.2.2 Parsing Module (*parserN*)

This is a core module of the implementation. In this module, transaction formulas (queries or rule bodies) are parsed and translated into Prolog formulas. New rules may be added during the parsing phase.

4.2.3 Upload Module (*uploadN*)

It defines the directives *loadtrans/1* and *loadbase/1*. The former loads a transaction program into database, and the latter loads up a database program. When a transaction program is loaded, the translated Prolog code is saved in a file with the postfix "_tran" added to the original file name.

4.2.4 Query Module (*queryN*)

It defines the directive *trans/1* as an interface for transaction query. So, in order to perform a transaction ϕ, executes the query "$trans\ \phi$" in Prolog prompt.

4.2.5 Key Modules (*keys_basicsN* and *keys_applN*)

There are two modules for handling primary database keys, *keys_basicsN* and *keys_applN*. The first one defines basic operators for keys, i.e., *declare*, *create/1*, *destroy/1*, etc. The second defines operators for bulk updates.

4.2.6 Transaction Function Modules (*tr_basicsN* and *tr_applN*)

These modules are used in Prototype 4 only. The *tr_basicsN* module defines basic transaction operators like *ins/1*, *del/1*. The *tr_applN* module defines transaction operators for bulk updates.

4.2.7 Naming Module (*namingN*)

This module contains rules that transform predicate names. For instance, it contains rules outputing *key_age*, or *value_age*, etc., when the input is the predicate name *age*, as mentioned above.

4.2.8 Basic Function Module (*basic_funsN*)

This is the lowest-level module. Some of its rules may need changing when porting the $\mathcal{T}R$ prototypes to other Prolog interpreters.

4.3 General Features of the Implementation

4.3.1 Undo Operations

At first glance, a natural method of translating elementary updates (*ins* and *del*) from $\mathcal{T}R$ formulas to Prolog formulas is to simply replace them by the Prolog update operators (*assert* and *retract*). However, the resulting program would not work correctly. The reason is that Prolog does not undo updates during backtracking.

To get around this problem, we need to remember the tuples being updated, and undo the update when backtracking. If a predicate is asserted, then during backtracking, it must be retracted; and vice versa. Consider the following rules:

$$on\_backtracking(X). \tag{4.1}$$

$$on\_backtracking(X) \quad :- \quad call(X),\ !,\ fail. \tag{4.2}$$

When $on\_backtracking(X)$ is first called, this predicate is true no matter what argument X is, so nothing happens. During backtracking, the second rule is executed and X is called. With cut (!) and fail, this rule also fails but X is not undone. So, this predicate has no effect when it is executed the first time, but the argument X will be executed during backtracking. The following example shows how the insertion and deletion of tuples are undoable during backtracking.

$$ins(X) \quad :- \quad assert(X), on\_backtracking(retract(X)). \tag{4.3}$$

$$del(X) \quad :- \quad retract(X), on\_backtracking(assert(X)). \tag{4.4}$$

Consider the transaction query $?-ins(p(1))\ sc\ fail$. First, $p(1)$ is asserted. Then, the execution of $on\_backtracking(retract(p(1)))$ returns true without any side effect. During backtracking, $p(1)$ is retracted, i.e., the assert is undone. However, there is a weakness in this method, which will be discussed in Section 4.3.3.

4.3.2 Transaction Base and Database

In $\mathcal{T}R$, a predicate can be defined in two places, in the database and in the transaction base. During transaction processing, both sets of rules are used; but during query processing, only database rules are used. This difference is accounted for the logical semantics of $\mathcal{T}R$ [$BK95$].

In the implementation, database rules and transaction rules are both implemented as Prolog rules. To distinguish them, the $\mathcal{T}R$ compiler adds a special pre-condition to each transaction rule, e.g., the rule $a \leftarrow b_1\ sc\ b_2\ sc\ ....\ sc\ b_n$ is compiled into the following Prolog rule:

$$a \quad :- \quad tr,\ b_1,\ b_2,\ ....,\ b_n.$$

Thus, transaction rules can effectively be disabled by retracting the atom "tr" from the Prolog database.

$$r1. \tag{4.5}$$

$$r2 \quad \leftarrow \quad p(X)\ sc\ q(X). \tag{4.6}$$

$$r1 \quad :- \quad tr. \tag{4.7}$$

$$r2 \quad :- \quad tr,\ p(X),\ q(X). \tag{4.8}$$

$$r3. \tag{4.9}$$

$$r4 \quad :- \quad p(X)\ ,q(X). \tag{4.10}$$

Consider the above three pairs of rules, Rules 4.5 and 4.6 are transaction rules, Rule 4.7 and Rule 4.8 are their translations into Prolog, and Rule 4.9 and Rule 4.10 shows two Prolog rules with similar structures to Rules 4.5 and 4.6. For transaction processing, the flag "tr" is asserted into database. After the flag is asserted, the pre-condition tr in all transaction rules succeeds, so both transaction rules and database rules can be used in subsequent queries. To disable transaction rules, the flag "tr" is retracted from database. In subsequent queries, since the pre-condition tr in all transaction rules fails, only database rules can be used. For instance, if a Prolog database contains Rule 4.7, then the Prolog query "$?-r1$" fails but the query "$?-ins(tr), r1$" succeeds. Therefore, when a transaction ϕ is executed, it is converted into "$ins(tr),\ \psi,\ del(tr)$" where ψ is the Prolog translation of ϕ. Before ψ is executed, tr is asserted to activate transaction rules, and after the execution of ψ, tr is retracted to deactivate the transaction rules.

As discussed above, only transaction processing uses rules in the transaction base. Therefore, rules in database cannot use rules in transaction base. For instance, suppose a Prolog translation of a $\mathcal{TR}$ program contains the following rules:

$$\begin{array}{lcl} p(1) & :- & tr. \\ p(2). & & \\ q(X) & :- & p(X). \end{array}$$

Above prolog rules represent that the transaction base contains $p(1)$, and the database contains $p(2)$ and "$q(X) :- p(X)$". So, the database query $q(X)$ returns $X = 2$ only. However, if the above transaction processing method is used, then a transaction query $q(X)$ will be translated into "$ins(tr),\ q(X),\ del(tr)$". The query returns $X = 1$ and $X = 2$ because after tr is inserted, $q(X)$ can bind X to 1 and 2. However, since $q(1)$ is neither in transaction base, nor in database (only $q(2)$ is in database), the answer from this transaction query is incorrect. The reason for the failure is the use of transaction rule in a database rule. To avoid using transaction rules from a database rule in transaction query, a predicate $dbase(X)$ is defined as follows:

$$\begin{array}{lcl} dbase(X) & :- & tr,\ !,\ del(tr),\ X,\ ins(tr) \\ & & ; \\ & & X. \end{array} \tag{4.11}$$

The predicate $dbase(X)$ executes X directly if the tr flag is not in the database, or deletes the flag before executing X if it is in the database. In the latter case, after finished executing X, tr is inserted into database, so that other transaction process can continue to be executed. Also, instead of directly asserting database rules into Prolog database, we translate the rules

$p :- \psi$ into $p :- dbase(\psi)$ in Prolog, where p is an atomic formula and ψ is a database formula. Therefore, the database rule "$q(X) :- p(X)$" in the previous example becomes $q(X) :- dbase(p(X))$ in Prolog. Now, during transaction processing of $q(X)$, after the "tr" flag is inserted, with the rule $q(X) :- dbase(p(X))$, $dbase(p(X))$ is executed. The "tr" flag is then deleted before $p(X)$ is called. In this case, X binds to 2 only.

Apart from the addition of a "tr" flag, the compiling of a transaction rule also involves some parsings and translations. The serial conjunction operator "sc" is replaced with the conjunction operator ',' in Prolog (as shown in Rule 2 and Rule 4 in the above examples), because the evaluation of a clause body in Prolog executes serially from left to right. Other translations of particular predefined predicates will be discussed later in this section.

4.3.3 Delete Tags

In Section 4.3.1, although a backtrackable insert and delete was defined, the query "$? - del(p(X)), fail$" hangs with the following database in XSB Prolog:

$$p(1).$$
$$p(2).$$

When we analyse the above query execution, we notice that the following Prolog query has a similar set of subgoals produced during execution:

$$? - p(X),\ retract(p(X)),\ assert(p(X)),\ fail.$$

The execution also hangs. The reason is that during the first evaluation of $p(X)$, X is unified to 1. After the retraction of $p(1)$, the hash entry of $p(1)$ has been removed. Although $p(1)$ is asserted later during the backtracking, the hash table is changed. So, in the next unification, the program hangs due to an improper pointer. If $retract\_nr/1$ (which does not reclaim used space) is used in place of $retract/1$ in the above query, then the program goes into an infinite loop because $p(1)$ is put immediately back into the same place, which will then be retracted again (same result will be obtained if we replace $assert(p(X))$ by $assertz(p(X))$). The program does not notice that X has already been unified to 1 and failed. Although we can use $asserta$ (add the clause in the argument to the database before any other clause for the same predicate currently in the database), it will revert the order of the database. For instance, the database contains $p(1)$ and $p(2)$, and $p(1)$ is located before

$p(2)$. The query "$?-p(X),\ retract(p(X)),\ asserta(p(X)),\ fail$" first unifies X to 1, and retracts $p(1)$. Then, $p(1)$ is asserted (by $asserta(p(X))$) before any other p-tuples currently in the database, i.e., before $p(2)$. So, the order is still:

$$p(1).$$
$$p(2).$$

During the backtracking, X is unified to 2, and $p(2)$ is retracted. Then, $p(2)$ is re-asserted before $p(1)$ by $asserta(p(X))$, i.e.,

$$p(2).$$
$$p(1).$$

So, the order of p-tuples in the database is reversed. As mentioned before, the execution of "$?-del(p(X)),\ fail$" results in the same set of subgoals as "$p(X), retract(p(X)),$ $assert(p(X)),\ fail$". This shows why the deletion predicate $del(X)$ is defined improperly in Section 4.3.1.

To solve this problem, we notice that if a query has multiple answers, backtracking can cause problems, because when undoing an update during backtracking, the hash table can be corrupted. To overcome this, we never actually remove atoms from the database. Instead, we mark them as deleted, using delete tags. The insert tag and delete tag of a predicate $p(X_1, X_2, \ldots, X_n)$ are defined by $i\_p(X_1, X_2, ..., X_n)$ and $d\_p(X_1, X_2, \ldots, X_n)$, respectively. The insert tag represents the existence of the predicate in the database, and the delete tag means the predicate has been deleted from database. So, a predicate is in the database if its insert tag is in the database, but its delete tag is not. Similarly, a predicate is not in the database if both its insert tag and delete tag are in the database (i.e., it has been deleted), or neither of them are in the database, (i.e., it has never been inserted). Therefore, the insertion $ins\_p(X)$ of a predicate $p(X)$ is now defined by the following Prolog rules:

$$ins\_p(X) \quad :- \quad not(i\_p(X)),\ ins(i\_p(X)). \tag{4.12}$$
$$ins\_p(X) \quad :- \quad i\_p(X),\ del(d\_p(X)). \tag{4.13}$$

where *ins* and *del* are backtrackable updates as defined in Section 4.3.1.

The first rule means that in the insert tag, $i\_p(X)$, is not in the database, so the predicate has never been asserted, and we can assert the insert tag to represent that $p(X)$ is now in database. In the second rule, if the insert tag $i\_p(X)$ is in the database, then remove the delete tag $d\_p(X)$ from the database. In both cases, if the predicate $p(X)$ is successfully inserted, only $i\_p(X)$ and not $d\_p(X)$ is in the database.

Likewise, the deletion of $p(X)$ is defined as follows:

$$del\_p(X) :- \ i\_p(X), \ ins(d\_p(X)). \tag{4.14}$$

If a predicate has been inserted, its insert tags $i\_p(X)$ should be in the database. To delete the predicate, we mark it by inserting the delete tag $d\_p(X)$ into the database. If the delete tag already exists, the predicate has already been deleted, so the deletion fails.

As discussed above, a predicate is inserted if its insert tag but not its delete tag is in the database. So, $p(X)$ is defined by the rule:

$$p(X) :- \ i\_p(X), \ not(d\_p(X)). \tag{4.15}$$

For example, if $i\_p(1)$ and $i\_p(2)$ are in the database, and $d\_p(1)$ and $d\_p(2)$ are not, then $p(1)$ and $p(2)$ are true. Thus, the query "$?-del\_p(X),\ fail$" executes, $i\_p(X)$ unifies X to 1 and $d\_p(1)$ is asserted. During backtracking, $d\_p(1)$ is retracted. Then, $i\_p(X)$ unifies X is to 2, and $d\_p(1)$ is asserted and then retracted during backtracking. This execution does not hang because the hash table of $i\_p(X)$ is not changed during the execution. For the same reason, the order of answers to the query $p(X)$ is preserved.

4.3.4 The Declare Statement (Key and Indexing)

A key is a set of attributes that uniquely identifies a tuple. Every predicate has a key, which may be, in two extremes, the functor name only, or the whole tuple. In the former case, the relation contains one entry only, and the relation name uniquely identifies this entry. In the latter case, if any attribute is missing, then some tuples may be unable to identify. Therefore, the only possible key is the whole tuple. For instance, in some of our later examples, a predicate $count(X)$ is used to hold the current value of a counter. Since it has only one value at any given time, $count(X)$ is defined to have an empty key, and the predicate name itself uniquely identifies the value, X. Similarly, suppose a predicate $edge(X, Y)$ means there is an edge from nodes X to Y in a graph. Since each vertex may have several edges, we cannot declare the predicate with the first argument as key only. Instead, the key consists of both X and Y, i.e., the whole tuples.

Before any tuples are inserted into a table, the key information should first be provided.This is the function of the declare statement. A declare statement consists of 3 parts: the predicate name and arity, the key specification and index information. Here is a general declare statement:

$$declareName/N\ [key\ K\ [size\ M\ index\ L]]. \tag{4.16}$$

The first part is the predicate specification, which consists of the predicate name Name, and arity of the predicate N. For instance, $p/2$ represents a predicate with name p and arity 2, i.e., $p(X, Y)$. In the second part of a declaration statement, the number K means that the first K arguments of the predicate are the primary key. If K is not given, then by default, all the arguments form the primary key. The last part, index information is discussed later in this section.

During a declaration, the program first checks if a predicate with the same name and same number of arguments is already declared. It is not possible to have distinct predicates with same name and same number of arguments. However, predicates can have the same name and same key but different number of arguments. For instance, the two declarations $p/2$ *key* 1 and $p/3$ *key* 1 are allowed. A predicate is said to be in the database if both its key and the entry identified by the key, called the value of the key, are in the database. With the declaration $p/2$ *key* 1, the key of $p(X, Y)$ is denoted by $key\_2\_p(X)$, and the value is denoted by $value\_2\_p(X, Y)$. The number 2 in the names of the key and value predicates is the number of arguments of the declared predicate $p(X, Y)$. With the above declaration, the following rule defining $p(X, Y)$ from these two predicates is added to the Prolog database:

$$p(X, Y) : -key\_2\_p(X), value\_2\_p(X, Y). \tag{4.17}$$

A predicate with same name, same key but different number of arguments can also be declared. For instance, "*declare* $p/3$ *key* 1" causes the following rule to be added to the Prolog database:

$$p(X, Y, Z) : -key\_3\_p(X), value\_3\_p(X, Y, Z). \tag{4.18}$$

The predicate $key\_3\_p(X)$ represents the key for $p(X, Y, Z)$, whereas $key\_2\_p(X)$ represents the key for $p(X, Y)$.

Thus, when there is a call to $p(a, Y)$, the system first checks if $key\_2\_p(a)$ is in the database. If so, it returns the unique value Y identified by the predicate $value\_2\_p(a, Y)$. Thus, the insertion of a p-tuple is implemented as the insertion of two predicates: the key $(key\_2\_p(X))$ and the value $(value\_2\_p(X, Y))$.

The last part of a declare statement is the index information, i.e., *size M index L*. This part is also optional, so if it is not given, then no index is used for the predicate. The size M is the size of the hash tables for the key and the value predicates. This is usually the number of ground atoms that will be inserted. The second part of the index information is the specification of secondary indexes, L. It has the same syntax and same meaning as index argument specifications in XSB Prolog. In general, L can be an integer that indicates the argument on which to index, or a list of $IndexArg$. An $IndexArg$ is an integer or a sequence of integers separated by +, e.g., 1+2+3, which indicates that a single index is to be built using these combined arguments. As an example, "*declare* $p/3$ *key* 2 *size* 100 *index* [1+2, 2]" will construct two hash tables with 100 entries for the combination of the first and second argument of the key $(key\_3\_p(X, Y))$ and value $(value\_3\_p(X, Y, Z))$ predicates, and two other hash tables with 100 entries for the second argument of the key and value predicates, i.e., four hash tables in total. So, a call to $p/3$ will first check to see if both the first and second arguments are ground, and if so, use the index based on those arguments for the predicate $key\_3\_p(X, Y)$. Otherwise, it will see if the second argument is ground, and if so, use the index based on it. Otherwise, it will use no index and search through all the clauses for the predicate.

By default, if an index field is not specified, indexes will be built on $[1 + 2 + \ldots + k,\ 1 + 2 + \ldots + (k-1),\ \ldots,\ 1]$. So, "*declare* $p/4$ *key* 3 *size* 100" is equivalent to "*declare* $p/4$ *key* 3 *size* 100 index $[1 + 2 + 3,\ 1 + 2,\ 1]$".

Examples:

1. declare edge/2.

2. declare buffer/2 key 0.

3. declare account/3 key 2.

4. declare account/3 key 2 size 100.

5. declare account/3 key 2 size 100 index [1].

Example 1 declares a predicate called *edge* with 2 arguments, which together form the primary key. Therefore, $edge(v1, v2)$ and $edge(v1, v3)$ can both be stored in the database. This declare statement is equivalent to "*declare edge*/2 *key* 2". No indexes are created.

Example 2 declares a predicate called buffer with 2 arguments, with an empty key. So, $buffer(1, 2)$ and $buffer(3, 4)$ cannot appear in the database together. Since the tuple has only 1 entry, it needs no indexes.

Example 3 declares that a predicate account has 3 arguments, where the first two form a key. So, $account(John, savings, 1000)$ and $account(John, checking, 1000)$ can both reside in the database. However, the system cannot store both $account(John, savings, 1000)$ and $account(John, savings, 2000)$ in the database. Since the index information is not specified, no indexes are created.

Example 4 is similar to Example 3 except that it specifies hash tables of size 100 for the key and value predicates. Default indexes are built on $[1+2, 1]$, i.e., an index on arguments 1 and 2 combined, and another index on argument 1.

Example 5 shows a user-defined index specification. It specifies hash tables of size 100 on the first argument of the key and value predicates.

4.3.5 Key

In Section 4.3.3, we discussed the use of insert and delete tags. The reason for using tags is to compensate for the loss of hash table entries during backtracking and to preserve the original order of facts in the Prolog database. If there is more than more than one choice for a unification, then we have to use the delete tag method. Otherwise, if there is only one possible unification, we can use the simple method of undoing updates described in *Rule 4.3* and Rule 4.4, because there is only one possible hash entry.

Consider the following example, in which, the predicate *distance*/3 is declared by "*declare distance*/3 *key* 2". The following rule is added according to Section 4.3.4:

$$distance(X, Y, Z) :- key\_3\_distance(X, Y),\ value\_3\_distance(X, Y, Z). \quad (4.19)$$

With the query, $?-distance(X, Y, Z)$, the subgoal $key\_3\_distance(X, Y)$ is evaluated. Suppose this subgoal binds X, Y to a, b, then, $value\_3\_distance(a, b, Z)$ will unify Z to c, for example, so the tuple $distance(a, b, c)$ is finally returned. Since the key uniquely identifies a tuple, there is only one possible binding for $value\_3\_distance(a, b, Z)$. For this reason, we can use the simple undoing method, instead of the delete-tag method, for updating value predicates. On the other hand, queries on a key may have multiple answers, so we use the tag method for implementing key predicates.

Section 4.3.3 showed that when a predicate is updated by the tag method, the predicate is not stored directly in the Prolog database. Instead, it becomes a defined predicate, and its insert and delete tags are base predicates. Therefore, the key predicate, e.g., $key\_3\_distance(X, Y)$, is defined in terms of tags,

$$key\_3\_distance(X, Y) \ :- \ i\_key\_3\_distance(X, Y), \ not(d\_key\_3\_distance(X, Y)). \tag{4.20}$$

So, if the tuple $distance(a, b, 1)$ is in the $\mathcal{TR}$ database, then $i\_key\_3\_distance(a, b)$ and $value\_3\_distance(a, b, 1)$, but not $d\_key\_3\_distance(a, b)$, are in the Prolog database.

Updates to Non-key Attributes:

In transaction programs, updates often occur on non-key attributes, e.g., changes to the balance of bank account, or to the state of an object. Without keys, the whole tuple would have to use the tag method. So, a change in just one argument would require the insertion of a delete tag followed by the insertion of a new insert tag. With keys, a change in the non-key arguments results in a retraction of the value predicate and an assertion of the new value predicate. So, without using keys, the database will contain two more predicates (a delete tag of old tuple and an insert tag of new tuple) after such update. However, database contains same number of predicates after the update if keys are used. For the same reason, the key should be carefully selected. For example, if the $distance2$ predicate is declared with 3 key arguments, then the declaration generates the following Prolog rules:

$$
\begin{aligned}
distance2(X, Y, Z) \ &:- \ key\_distance2(X, Y, Z). && (4.21)\\
key\_distance2(X, Y, Z) \ &:- \ i\_key\_3\_distance2(X, Y, Z),\\
&\qquad not(d\_key\_3\_distance2(X, Y, Z)). && (4.22)
\end{aligned}
$$

This is an optimization: since the whole tuple of $distance2$ is the key, the value predicate is redundant and can be discarded.

Compare the predicates $distance$ and $distance2$ again. Suppose both $distance(a, b, 1)$

and $distance2(a, b, 1)$ are in the database. Intuitively, each atom means that the distance from a to b is 1. We increase the distance from 1 to 1000 stepwise. For the predicate distance, at each step, $value\_3\_distance(a, b, N)$ is retracted and the new predicate $value\_3\_distance(a, b, N + 1)$ is asserted. So, after 1000 steps, the database contains the atoms $i\_key\_3\_distance(a, b)$ and $value\_3\_distance(a, b, 1000)$ to indicate the new distance. However, for the predicate $distance2$, at each step, $d\_key\_3\_distance2(a, b, N)$ and $i\_key\_3\_distance2(a, b, N + 1)$ are asserted. So, after 1000 steps for both predicates, the database contains:

$i\_key\_3\_distance(a, b).$	$value\_3\_distance(a, b, 1000).$
$i\_key\_3\_distance2(a, b, 1).$	$d\_key\_3\_distance2(a, b, 1).$
$i\_key\_3\_distance2(a, b, 2).$	$d\_key\_3\_distance2(a, b, 2).$
$\vdots$	
$\vdots$	
$i\_key\_3\_distance2(a, b, 998).$	$d\_key\_3\_distance2(a, b, 998).$
$i\_key\_3\_distance2(a, b, 999).$	$d\_key\_3\_distance2(a, b, 999).$
$i\_key\_3\_distance2(a, b, 1000).$	

In the above example, there are two reasons to favour the declaration of $distance$ over $distance2$. Too see this, suppose that $distance2$ is used. Then, after execution, the database contains 999 pairs of insert and delete tags for the $key\_3\_distance2$ predicate, needlessly consuming space. In addition, if there is a query on $key\_3\_distance2(a, b, N)$, the subgoal $i\_key\_3\_distance2(a, b, N)$ will bind N to 1 followed by the failure of the next subgoal $not(d\_key\_3\_distance2(a, b, N))$. The execution will backtrack to bind N to 2, 3, etc., until N is 1000, where the binding succeeds. On the other hand, the query $key\_3\_distance(a, b, N)$ needs only two steps: verifying that $i\_key\_3\_distance(a, b)$ is true, and retrieving $value\_3\_distance(a, b, 1000)$ from database. Needless to say, the latter requires less search time.

A drawback of breaking a predicate into separate key and value predicates is that an insertion or deletion of a tuple requires updates to two predicates instead of one. Therefore, if there is a lot of creation and deletion of tuples, then we have to pay for it in terms of the extra updates on value predicates.

As discussed in the previous section, if a predicate is declared with index information, then the key and the value predicate will indexed accordingly. Since the key predicate is not a base predicate, the index information is transferred to its tag predicates which are base. For instance, "*declare distance*/3 *key* 2 *size* 100 *index* [1]" puts indexes on the first argument of $i\_key\_3\_distance/2$, $d\_key\_3\_distance/2$, and $value\_3\_distance/3$.

4.3.6 Create, Destroy and Set

In our implementation, three basic operations for a database tuple are *create*, *destroy*, and *set*. The predicate *create* inserts a tuple with a new key value. Similarly, *destroy* deletes the tuple; and, *set* changes the value of an existing tuple. The *set* operation is allowed only when the key already exists in the database. As discussed in Section 4.3.4 about vis-a-vis *declare*, predicates involving these operations should be preceded by the declare statement first.

For example, suppose $p/2$ is declared with the first argument is the key. Then,

- The operation $create(p(1,2))$ inserts a tuple $p(1,2)$ with $p(1)$ as the key. Thus, $i\_key\_2\_p(1)$ and $value\_2\_p(1,2)$ are asserted.
- The operation $destroy(p(1,2))$ effectively removes the atom $p(1,2)$ by asserting $d\_key\_2\_p(1)$ and retracting $value\_2\_p(1,2)$.
- The operation $set(p(1,3))$ changes the value of the atom with key $p(1)$ to 3. If the atom with key $p(1)$ originally had the value 2, then the operation retracts $value\_2\_p(1,2)$ and asserts $value\_2\_p(1,3)$. In this way, instead of destroying the whole tuple and re-creating a new one, we can use the *set* operator to modify the non-key attributes.

Each key has only one corresponding value. So, the operator *create* should check whether a tuple with same key has been previously created. If it has, the *create* operation fails. Also, consider the operator *set*. If the whole tuple is a key, then the *set* operation is invalid because there is no value to set.

4.3.7 Bulk Updates

A feature of database languages is the ability to perform bulk updates. For example, inserting a set of tuples from one relation into another is a basic SQL operation. However, in most logic programming languages, such operations are not defined. The reason is that, unlike SQL operations, most logical formulations of updates are based on the insertion and deletion of single tuples.

In order to capture such behavior, we shall define an elementary transition that performs bulk updates. An example of the use of relational assignments, which copy the contents of one relation to another, was shown in Section 3.3.

A bulk update formula usually consists of two parts: a source formula (S), and a destination predicate (D). A source formula is a database query which sets up the conditions for update, i.e., a tuple is update if S is true. The destination predicate is a database predicate where the update takes place. Also, the predicate should have been declared (like $create/1$, $destroy/1$, etc.) So, if S is a database query and D is a declared database predicate, then the following defines three types of bulk updates: bulk insert , bulk delete, and bulk assignment:

1. Bulk Insert ($bulk\_ins(S, D)$)

   All tuples from D that satisfies condition S are inserted, and if a variable is not bound when the bulk insertion takes place, it is considered to be a free variable if it is not specified in S. For instance, $bulk\_ins(p(X,Y), q(X))$ means for every value X, if $\exists Y\ p(X,Y)$ is true in the database, then insert $q(X)$.

   Examples: The following shows the database before and after a bulk insert, where the declarations of p, q, r are: $p/2$ *key* 2, $q/1$ *key* 1, and $r/2$ *key* 1:

   1. Database Before: $p(1,1)$, $p(2,2)$, $q(3)$.
      Execution: $bulk\_ins(p(X,Y), q(X))$
      Database After: $p(1,1)$, $p(2,2)$, $q(1)$, $q(2)$, $q(3)$.
      With X bound to 1 and 2, q(1), q(2) are inserted.

   2. Database Before: $p(1,1)$, $p(2,2)$, $q(3)$
      Execution: $bulk\_ins(p(X,2), q(X))$
      Database After: $p(1,1)$, $p(2,2)$, $q(2)$, $q(3)$
      Only $p(2,2)$ satisfies the source condition and so $q(2)$ is inserted.

   3. Database Before: $p(1,1)$, $p(1,2)$, $q(3)$
      Execution: $bulk\_ins(p(X,Y), q(X))$
      Database After: $p(1,1)$, $p(1,2)$, $q(1)$, $q(3)$

Atoms $p(1,1)$, $p(1,2)$ bound X to 1, and $q(1)$ is inserted. Duplicated entry of $q(1)$ is discarded.

4. Database Before: $p(1,1),\ p(1,2),\ p(2,2),\ q(1),\ r(3,3)$

   Execution: $bulk\_ins((p(X,Y),q(X)),r(X,Y))$

   Database After: $p(1,1),\ p(1,2),\ p(2,2),\ q(1),\ r(1,1),\ r(3,3)$.

   The evaluation of the source query, $(p(X,Y),q(X))$, returns $X = 1$, $Y = 1$, and $X = 1$, $Y = 2$, so, $r(1,1)$, $r(1,2)$ can be added to the database. However, since the first argument of r is the key, only one of $r(1,1)$ or $r(1,2)$, is added into database non-deterministically. In this case, $r(1,1)$ is added.

5. Database Before: $p(1,2),\ p(2,3),\ q(2)$

   Execution: $bulk\_ins((p(X,Y),q(X)),q(Y))$

   Database After: $p(1,2),\ p(2,3),\ q(2),\ q(3)$

   The evaluation of $(p(X,Y),q(X))$, returns $X = 2$, $Y = 3$, so, q(3) is added to the database.

6. Database Before: $p(1,1),\ p(2,2),\ q(2),\ r(3,3)$

   Execution: $q(X)\ sc\ bulk\_ins(p(X,Y),r(X,Y))$

   Database After: $p(1,1),\ p(2,2),\ q(2),\ r(2,2),\ r(3,3)$

   The binding of X to 2 affects the bulk insert. So, only r(2,2) is added to the database.

7. Database Before: $p(1,1),\ p(2,2),\ q(2),\ r(3,3)$

   Execution: $bulk\_ins(p(X,Y),r(X,Y))\ sc\ q(X)$

   Database After: $p(1,1),\ p(2,2),\ q(2),\ r(1,1),\ r(2,2),\ r(3,3)$

   Although the execution returns X to 2, the binding of X after the bulk insert does not affect the previous operation. So, both $r(1,1)$ and $r(2,2)$ are added.

2. Bulk Delete ($bulk\_del(D)$)

   All tuples satisfying D are deleted. For instance, $bulk\_del(p(X,Y))$ deletes all tuples of the form $p(X, Y)$.

Examples: The predicate $p(X, Y)$ is declared as p/2 key 1.

1. Database Before: $p(1,1),\ p(2,2)$

   Execution: $bulk\_del(p(X, Y))$

   Database After: (No $p/2$ tuples)

   All tuples of the form $p(X, Y)$ are deleted.

2. Database Before: $p(1,1),\ p(2,1),\ p(3,3)$

   Execution: $bulk\_del(p(X, 1))$

   Database After: $p(3,3)$

   All tuples of the form $p(X, 1)$ are deleted from database, i.e., $p(1,1)$, $p(2,1)$.

3. Bulk Assignment ($D := S$)

   All tuples from D are overwritten by those satisfying the condition S. Again, variables specified in S but not in D are considered to be free variables. For instance, $q(X) := p(X, Y)$ selects the first elements of $p(X, Y)$ and put them into the predicate $q(X)$. It is similar to $bulk\_del(q(X))$ followed by $bulk\_ins(p(X, Y), q(X))$.

   Examples: The following shows the database before and after an assignment. Note the difference between bulk insert and bulk assignment. In the former case, any tuple in the source predicate remains in the database, while in the latter case, all tuples initially in source predicate are deleted.

   1. Database Before: $p(1,1),\ p(2,2),\ q(3)$.

      Execution: $q(X) := p(X, Y)$

      Database After: $p(1,1),\ p(2,2),\ q(1),\ q(2)$

      With X bound to 1 and 2, $q(1)$, $q(2)$ are in the database after the assignment. Note the difference with bulk insert, where $q(3)$ remained in the database after the update.

   2. Database Before: $p(1,1),\ p(2,2),\ q(3)$

      Execution: $q(X) := p(X, 2)$

Database After: $p(1,1)$, $p(2,2)$, $q(2)$

Only $p(2,2)$ satisfies the source condition, i.e., $q(X)$ was deleted so the predicate q has extent $q(2)$ in the final database state.

3. Database Before: $p(1,1)$, $p(1,2)$, $q(3)$

   Execution: $q(X) := p(X,Y)$

   Database After: $p(1,1)$, $p(1,2)$, $q(1)$

   Atoms $p(1,1)$, $p(1,2)$ bind X to 1, so $q(1)$ is in new the database state. Duplicated entry of $q(1)$ has been discarded.

4. Database Before: $p(1,1)$, $p(1,2)$, $p(2,2)$, $q(1)$, $r(3,3)$

   Execution: $r(X,Y) := (p(X,Y), q(X))$

   Database After: $p(1,1)$, $p(1,2)$, $p(2,2)$, $q(1)$, $r(1,1)$

   The evaluation of the source query, $(p(X,Y), q(X))$, returns $X = 1$, $Y = 1$, and $X = 1$, $Y = 2$, so, $r(1,1)$, $r(1,2)$ can be added to the database. However, since the first argument of r is the key, only one of $r(1,1)$ or $r(1,2)$ is added to database non-deterministically. In this case, $r(1,1)$ is added.

5. Database Before: $p(1,2)$, $p(2,3)$, $q(2)$

   Execution: $q(Y) := p(X,Y), q(X)$

   Database After: $p(1,2)$, $p(2,3)$, $q(3)$

   The evaluation of $(p(X,Y), q(X))$, returns $X = 2$, $Y = 3$, and $q(3)$ is added to the database. Note that $q(2)$ is no longer in the database after the bulk update.

6. Database Before: $p(1,1)$, $p(2,2)$, $q(2)$, $r(3,3)$

   Execution: $q(X)\ sc\ r(X,Y) := p(X,Y)$

   Database After: $p(1,1)$, $p(2,2)$, $q(2)$, $r(2,2)$

   The binding of X to 2 affects the bulk insert. So, only r(2,2) is added to the database.

7. Database Before: $p(1,1)$, $p(2,2)$, $q(2)$, $r(3,3)$

   Execution: $r(X,Y) := p(X,Y)\ sc\ q(X)$

Database After: $p(1,1),\ p(2,2),\ q(2),\ r(1,1),\ r(2,2)$

Although the execution returns X to 2, the binding of X after the bulk insert does not affect the previous operation. So, both $r(1,1)$ and $r(2,2)$ are added.

4.3.8 Loops

In most logic programming languages, only recursive rules are allowed. Implementation of iterative loops is provided by procedural languages only. Since $\mathcal{TR}$ is a blend of declarative and procedural programming languages, we can include *while-loop* and *for-loop* implementations in the language.

Here is the general syntax of while-loops and for-loops:

[with *VList*] while *Cond* do *Body*

[with *VList*] forall *Vars* in *Cond* do *Body*

where *Cond* is a database query,called the loop condition,

and *Body* is a transaction formula,called the loop body,

and *VList* and *Vars* are lists of variables.

The optional prefix "with *VList*" is used when passing the bindings of variables into the loop or out from the loop, including those variables which are bound before the loop. Variables in *VList* are considered to have constant values during loop iteration. Examples are shown later in this section.

While Loops:

A while-statement consists of two parts: loop condition, and loop body. It repeatedly executes the *Body*. The iteration finishes when the loop condition fails. At each iteration, any variables in the loop condition may be bound to particular values, and these values are passed into the loop body and used there. For example, "$while\ p(X)\ do\ destroy(p(X))$" picks a value of X at each iteration and removes the tuple $p(X)$ from database. It is interesting to notice that the execution of a while-loop may be non-deterministic.

Examples:

1. Database Before: $p(1,1),\ p(2,2),\ p(3,3)$

Execution: $Y = 2$ *sc while* $p(X, Y)$ *do destroy*$(p(X, Y))$.

Database After: (*No* $p(X, Y)$ *tuples*)

With specifying "*with*[Y]" as prefix of the while loop, the binding, $Y = 2$, does not pass into the loop, the iteration may remove $p(1, 1)$ in the first iteration, $p(2, 2)$ in the next iteration, so that all $p(X, Y)$ tuples are destroyed.

2. Database Before: $p(1, 1)$, $p(2, 2)$, $p(3, 3)$

   Execution: $Y = 2$ *sc with* [Y] *while* $p(X, Y)$ *do destroy*$(p(X, Y))$

   Database After: $p(1, 1), p(3, 3)$

   The execution destroys all $p(X, 2)$ tuples, i.e., $p(2, 2)$ only, because the binding of Y passes into the loop by using the prefix "with [Y]" in the while-loop.

3. Database Before: $p(1, 1)$, $p(2, 2)$, $p(3, 3)$

   Execution: *while* $p(2, 2)$ *do destroy*$(p(X, Y))$

   The execution is non-deterministic because the loop condition only states that $p(2, 2)$ is false at the end of the loop. The execution may destroy $p(2, 2)$ immediately in the first iteration, leaving $p(1, 1)$ and $p(3, 3)$ in the database; or at the other extreme, it may destroy $p(1, 1)$ first, then $p(3, 3)$, and finally $p(2, 2)$. In this case, all $p(X, Y)$ tuples are removed.

4. Database Before: $p(1, 1)$, $p(2, 2)$, $p(3, 3)$

   Execution: *with* [X, Y] *while* $p(X, Y)$ *do destroy*$(p(X, Y))$

   This example is not too useful but worth to remember. In this case, the value of X, Y are bound in the first iteration. Suppose, X and Y unify to 1 in the first iteration, only $p(1, 1)$ is deleted in the iteration and the database after iteration will contain $p(2, 2)$ and $p(3, 3)$.

For Loops:

A for-statement consists of three parts: variable list, loop condition, and loop body. Each set of variables in *Vars* that satisfies the loop condition *Cond* are used in each loop iteration. So, the number of iterations is fixed when the iteration starts.

Examples:

1. Database Before: $p(1,1),\ p(2,2)$

   Execution: $Y = 2\ sc\ forall\ [X]\ in\ p(X,Y)\ do\ create(q(X))$.

   Database After: $p(1,1),\ p(2,2),\ q(1),\ q(2)$

   The loop first determines a set of values $[1,2]$ for X, from $p(X,Y)$, and from this set, it creates $q(X)$ at each iteration.

2. Database Before: $p(1,1),\ p(2,2)$

   Execution: $Y = 2\ sc\ with\ [Y]\ forall\ [X]\ in\ p(X,Y)\ do\ create(q(X))$.

   Database After: $p(1,1),\ p(2,2),\ q(2)$

   Y is bound to 2, so only $X = 2$ is chosen for iteration.

3. Database Before: $p(1,1),\ p(1,2)$

   Execution: $forall\ [X]\ in\ p(X,Y)\ do\ create(q(X))$

   Database After: $p(1,1),\ p(1,2),\ q(1)$

   Although two tuples, $p(1,1)$ and $p(1,2)$ bind X to 1, the loop iterates only once. If not, the iteration would fail because after $q(1)$ is created in the first iteration, $create(q(1))$ in the second iteration would fail.

While-loops and for-loops cannot be translated into simple Prolog goals. Instead, they are translated into recursive rules together with a predicate to start the recursion. For example:

1. $with\ [X]\ forall\ [Y]\ in\ p(Y)\ do\ create(q(X,Y))$

   is translated into

   $temp(Y)\ := \ p(Y),\ loop1(X)$

   and the following rules are created

   $loop1(X)\ :-\ destroy(temp(Y)),\ create(q(X,Y)),\ loop1(X).$

   $loop1(X)\ :-\ not(temp(Y)).$

2. $while\ alpha\ do\ body$

   is translated into

   $loop1$

and the following rules are created

$loop1 \ :- \ alpha, \ body, \ loop$

$loop1 \ :- \ not(alpha)$

Note that loop conditions are database queries which are applied to the database only, not including the transaction base.

4.3.9 If-then-else

The if-then-else statement provides branch selection within a transaction formula. The general structure of an if-then-else statement is: if *Cond* then *F1* else *F2*, where *Cond* is a database clause, and *F1* and *F2* are transaction formulas. Intuitively, it means that if *Cond* is true, then execute *F1*, otherwise execute *F2*. In other word, if *Cond* is true, execute *F1*, and if *Cond* is false, execute *F2*. The simplest way to translate the statement into $\mathcal{TR}$ is to define a predicate *if(C,T,E)* as follows:

$$if(C,T,E) \quad :- \quad call(C), call(T). \tag{4.23}$$

$$if(C,T,E) \quad :- \quad not(C), call(E). \tag{4.24}$$

So, a transaction "$if\ Cond\ then\ F1\ else\ F2$" can be translated into "$if(Cond, T, E)$" where T and E are Prolog interpretations of $F1$ and $F2$, respectively. For instance, $if\ p(1)\ then\ (q1\ sc\ q2)\ else\ (r1\ sc\ r2))$ can be translated into $if(p(1),\ (q1,\ q2),\ (r1,\ r2))$.

One might want to define this predicate with cut as follows:

$$\begin{aligned} if(C,T,E) \quad :- \quad & call(C), !, call(T) \\ & ; \\ & call(E). \end{aligned} \tag{4.25}$$

However, if T fails, it would be impossible to backtrack to C again, and then some other bindings of variables in C that cause T success may be missed. For example, suppose the database contains: $p(1), p(2), q(2)$, and the transaction "$if\ p(X)\ then\ destroy(q(X))\ else\ true$" is translated into "$if(p(X),\ destroy(q(X)),\ true)$". Using the first definition of $if/3$, $p(X)$ binds X to 1. However, in the next subgoal, $destroy(q(1))$ fails. In the backtracking $p(X)$ binds X to 2, and the execution succeeds by removing tuple $q(2)$ from the database. On the other hand, if the second definition is used, since the cut(!) in the definition disallows the backtracking to $p(X)$, the execution fails.

4.3.10 Query and Cleanup

Before a transaction formula is executed, it passes through a parser and is translated into a set of a Prolog subgoals. The directive *trans/1* is a prefix operator defined as an interface for transaction query. To perform a transaction ϕ, executes the query "*trans* ϕ" in Prolog prompt. For instance, *trans* $create(p(1))$ *sc* $create(p(2))$ creates tuples $p(1)$ and $p(2)$, but $trans(create(p(1)),\ create(p(2)))$ fails because $(create(p(1)),\ create(p(2)))$ is not a transaction formula.

The execution of *trans* consists of five parts: cleanup, parsing and translation, activation of transaction rules, execution of the transaction, and deactivation of transaction rules.

During the translation phase, a transaction query like $trans(p(X)\ sc\ create(q(X)))$ will be translated into the prolog query $p(X),\ create(q(X))$. Rules may also be created during the translation for the extra facilities provided by $\mathcal{TR}$. For example, *trans while* $p(X)$ *do* $destroy(p(X))$ may be translated into *loop1* where a new rule defining *loop1* is asserted to the database, as explained in Section 4.3.8.

As described in Section 4.3.3, although the tag method can solve the problems of updates during backtracking, it increases the consumption of database space significantly, especially if tuples are repeatedly created and destroyed over and over again. Since tuples with both an insert tag and a delete tag are regarded as non-existent in the database, both tags can be removed from the database after execution. This is the job of the cleanup phase. Similarly, temporary rules added in previous transaction queries, such as $loop1$ in the above example, are removed to release program space. Therefore, the cleanup phase has two parts: the removal of insert tags and delete tags, and the removal of rules created in previous transactions. It is carried out before a transaction is executed, and it releases space used by the previous transaction. Since Prolog allows users to backtrack after a query succeeds, if the cleanup phase takes place after executing the query, the cleanup phase itself may have to backtrack. For instance, suppose the database contains $p(1)$, $p(2)$, and the query *trans* $p(X)$ returns $X = 1$. At this point, users can backtrack to find the second answer, $X = 2$. If the cleanup phase took place after the transaction executes, then during such kind of backtracking, any rule or tag, which has been deleted in the cleanup phase, should be re-asserted again. Otherwise, the database is inconsistent during the backtracking.

As shown in Section 4.3.1, transaction rules are enabled by asserting the atom "tr" to the Prolog database, and can be disabled by retracting it. So, before executing the translated subgoals, the atom "tr" is asserted during the activation stage. After the execution stage, the atom is retracted during the deactivation stage.

4.4 Prototypes

We have implemented four different $\mathcal{TR}$ prototypes. In all prototypes, predicate declaration is a way of building a transaction base. Each prototype has its own features, and tests different implementation architectures. Prototypes 1, 2, and 3 are very similar, with the former being conceptual, and the latter being practical. In general, Prototype 1 is the slowest but simplest, and Prototype 3 is the fastest. Prototype 4 is a non-backtrackable version of Prototype 3 that optimizes the performance for problems that do not require backtracking.

4.4.1 Prototype 1

In each prototype, during the declaration of a predicate, rules are asserted to the Prolog database. Prototype 1 provides a simple conceptual implementation of $\mathcal{TR}$. During the declaration of predicates, only one special rule which defines the predicate with its key is added to the database. Since the emphasis is on conceptual simplicity, no optimization is performed. For example, when the following database predicate is declared, only one rule is added, as follows:

$$\begin{array}{l} declare\ p/2\ key\ 1. \\ \quad\downarrow \\ p(X,Y)\ :-\ key(p(X),2),\ val(p(X),[Y],2). \end{array}$$

Rather than using particular key and value predicates for each newly declared predicate, two general predicates $key/2$ and $val/3$ are defined for storing keys and values, respectively. With the general predicate $key(X,N)$, X is a function term representing key value of a predicate of arity N. For example, $key(p(X),2)$ is a key for $p/2$, and $key(q(X,Y),5)$ is a key for $q/5$. Likewise, in the general predicate $val(X,Y,N)$, X is a function term representing a key value of a predicate. Y is a list which stores the remaining values of the predicate, and N is the arity of the predicate. For example, the value predicates of above example are $value(p(X), [Y], 2)$ and $value(q(X, Y), [A, B, C], 5)$, respectively.

As discussed in section 4.3.3, the key of a tuple is stored using tags. Therefore, the key predicate $key(X, Y)$ is not stored as a base predicate in Prolog. Instead, there is a general rule defining $key/2$ in terms of its tags:

$$key(X, N) \quad :- \quad i\_key(X, N),\ not(d\_key(X, N)). \tag{4.26}$$

So, with the above declaration, the atom $p(1, 1)$ is stored as $i\_key(p(1), 2)$, $val(p(1), [1], 2)$ in database, and $p(2, 2)$ is stored as $i\_key(p(2), 2)$, $val(p(2), [2], 2)$ in database.

In Prolog, if the indexed argument is a predicate, then the name of the predicate is used for indexing. So, Prolog indexes the key on the predicate name, which is almost equivalent to no indexing because tuples are exhaustively searched in access time.

For this reason, access to the base predicates, $i\_key(X, Y)$, $d\_key(X, Y)$, and $val(X, Y, Z)$, requires an exhaustive search of the database, so search times are proportional to the number of predicates in the database. This linear search time for predicates explains why Prototype 1 is the slowest of the four. Since Prototype 1 is extremely slow, it is only suitable for small databases. Also, since there is effectively no indexing in Prototype 1, the index information in a declare statement has no effect.

Since this prototype is conceptual only, it does not have any optimization. The most noticeable example is when all arguments of a predicate are the primary key. For example:

$$\begin{array}{l} declare\ p/2\ key\ 2.\ \text{(or } declare\ p/2.\text{)} \\ \quad\quad \downarrow \\ p(X, Y)\ :-\ key(p(X, Y), 2),\ val(p(X, Y), [], 2). \end{array}$$

Since the value predicate does not contain any extra information, it can be omitted from the rule, thus preventing a needless search of the entire database. Such optimization will be performed in later prototypes.

4.4.2 Prototype 2

Unlike Prototype 1, Prototype 2 allows user-defined indexing on database tuples. Prototype 2 is a transition prototype between the conceptual version (Prototype 1) and the practical version (Prototype 3). It performs many runtime computations, as well as a little optimization on predicate definition.

In this prototype, each predicate has its own key predicate, which means that the predicate declaration also creates a rule for for accessing its corresponding key tags. During declaration, rules for the predicate and its key are created, as in the following examples,

$$declare\ p/2\ key\ 1.$$
$$\downarrow$$
$$p(X,Y)\ :-\ key\_2\_p(X),\ val\_2\_p(X,Y).$$
$$key\_2\_p(X) :-\ i\_key\_2\_p(X), not(d\_key\_2\_p(X)).$$

$$declare\ q/2.$$
$$\downarrow$$
$$q(X,Y)\ :-\ key\_2\_q(X,Y).$$
$$key\_2\_q(X,Y)\ :-\ i\_key\_2\_q(X,Y),\ not(d\_key\_2\_q(X,Y)).$$

As described in Section 4.3.4, if the declare statement contains index information, then the tags (e.g., $i\_key\_2\_p(X)$, $d\_key\_2\_p(X)$, $i\_key\_2\_q(X,Y)$, and $d\_key\_2\_q(X,Y)$) and the value predicate (e.g., $val\_2\_p(X,Y)$) are indexed accordingly, and this can improve the access time to predicates.

Unlike Prototype 1, simple optimizations are implemented in this prototype. If the whole tuple is the key, then the value field is unnecessary, so there is no need to include the value predicate in the predicate definition rule, i.e., $q(X,Y)\ :-\ key\_2\_q(X,Y).$ in above example. The removal of the value predicate in this case reduces the search time of the tuple but, of course, such optimization has no effect if the key a proper subset of the arguments, such as the above example for $p/2\ key\ 1$.

For instance, consider the above example. To insert $p(1,2)$ into database, in Prototype 1, $i\_key(p(1),2)$ and $value(p(1),[2],2)$ are inserted into database. These two base predicates are easily generated by decomposing $p(1,2)$ into $p(1)$, and [2], which are stored as the arguments of two general predicates, $i\_key/2$ and $value/3$. In Prototype 2, with the same update, two predicates $i\_key\_2\_p(1)$ and $value\_2\_p(1,2)$ are generated from $p(1,2)$. This processing requires a lot of run-time execution to create functor names for the key and value predicates, i.e., at run-time, the names $i\_key\_2\_p$ and $value\_2\_p$ are generated from the predicate name p. Procedures for generating predicate names are defined in the *naming* module. In this prototype, since the base predicates (the tags and value predicate) can be indexed on the attributes rather than the predicate name (like Prototype 1), there is a considerable gain in access time. For this reason, although updates in Prototype 2 require extra run-time computations, it is still more efficient than Prototype 1 because the loss in computation time is negligible comparing to the gain in search time.

In Prototype 2, since each predicate has its own set of key and value predicates, indexes can be used in this prototype. Indexing on the arguments in the key can speed up access to large databases. Note that indexing is applied to the tags (the insert tag and delete tag of the key), and the value predicate.

4.4.3 Prototype 3

Prototype 3 is designed for practical use. It is also the most recommended version among the three general prototypes. During the declaration of predicates, rules defining elementary transitions are constructed and added to the Prolog database. They include rules for creating or destroying a tuple, changing (setting) the value of a key, inserting or deleting key predicates, etc. For example, if $p/2$ is declared with 1 key attribute, then the following rules are constructed:

$$p(X,Y) \quad :- \quad key\_2\_p(X), val\_2\_p(X,Y). \tag{4.27}$$

$$key\_2\_p(X) \quad :- \quad i\_key\_2\_p(X), not(d\_key\_2\_p(X)). \tag{4.28}$$

$$create\_p(X,Y) \quad :- \quad ins\_key\_2\_p(X), insertbase(val\_2\_p(X,Y)). \tag{4.29}$$

$$destroy\_p(X,Y) \quad :- \quad del\_key\_2\_p(X), deletebase(val\_2\_p(X,Y)). \tag{4.30}$$

$$set\_p(X,Y) \quad :- \quad key\_2\_p(X), deletebase(val\_2\_p(X,\_)), insertbase(val\_2\_p(X,Y)). \tag{4.31}$$

$$ins\_key\_2\_p(X) \quad :- \quad not(i\_key\_2\_p(X)), insertbase(i\_key\_2\_p(X)). \tag{4.32}$$

$$ins\_key\_2\_p(X) \quad :- \quad i\_key\_2\_p(X), deletebase(d\_key\_2\_p(X)). \tag{4.33}$$

$$del\_key\_2\_p(X) \quad :- \quad key\_2\_p(X), insertbase(d\_key\_2\_p(X)). \tag{4.34}$$

where

$insertbase/1$ and $deletebase/1$ are predicates defined as backtrackable $assert$ and $retract$, same as $ins/1$ and $del/1$ described in Section 4.3.1.

Unlike Prototype 2, basic functions for a tuple like *create*, *destroy* and *set* are defined separately for each predicate. Therefore, during the executions of these functions, it is not necessary to perform a sequence of predicate transformations as in Prototype 2, and so such executions will be faster.

In Prototype 3 (and in Prototype 4), there are several implementations of for-loops. In general, there are three types of for-loops:

> [with *VList*] forall *Vars* in *Cond* do *Body*
> [with *VList*] forall unique *Vars* in *Cond* do *Body*
> [with *VList*] forall possible *Vars* in *Cond* do *Body*

The first type of for-loop is the most efficient. It does not remove duplicate answers to the query *Cond*. It also has limited backtracking capabilities. In general, users should be aware of duplicate answers which may cause the loop iteration to fail. For example, attempting to destroy a tuple which has been deleted in previous iteration will fail, causing the entire loop to fail.

The second type of for-loop is similar to the first except it eliminates duplicate query answers.

The third type of for-loop is the general for-loop used in Prototype 1 and Prototype 3. The option *possible* refers to possible orders for choosing values for the variables. Without backtracking, it should perform in the same behavior as the second type, i.e., with option *unique*. However, in backtracking, the first and second types of for-loop undo the all iterations of the loop. In this type, it will backtrack and iterate with other orders on the values of the loop variables.

Examples: Database contains: $p(1,1)$, $p(1,2)$, and $p(3,3)$

1. $forall\ [X]\ in\ p(X,Y)\ do\ create(q(X))$

   The loop fails because the loop variable Y binds to be 1, 1, 3. After $q(1)$ is created in the first iteration, the program fails when attempting to create it again.

2. $forall\ unique\ [X]\ in\ p(X,Y)\ do\ create(q(X))$

   The loop succeeds because duplicated answers for X are removed. So, $q(1)$ and $q(3)$ are successfully created.

3. $forall\ possible\ [X]\ in\ p(X,Y)\ do\ (not(q(1))\ sc\ create(q(Y))$

   Suppose the first iteration binds X to 1, $q(1)$ is created. However, in the second iteration, X is bound to 3, and the loop body "$not(q(1))\ sc\ create(q(X))$" fails. Using the first or the second type of loop, the entire loop fails. In this formula, however, the loop backtracks and tries other order on the values for the variables, i.e., binds X to 3 first, and then X to 1. In this way, $q(3)$ is created successfully in the first iteration. Then, in the second iteration, $not(q(1))$ sc $create(q(X))$ also succeeds because $q(1)$ was not in the database when this subgoal is called.

Examples of the implementation are as follows:

1. $with\ [X]\ forall\ [Y, Z]\ in\ p(Y, Z)\ do\ create(q(X, Y, Z))$
   is translated into
   $bagof([Y, Z], p(Y, Z), List),\ loop1(List, X)$
   and the following rules are created
   $loop1([[Y, Z]|List], X)\ :-\ create(q(X, Y, Z)),\ loop1(List, X).$
   $loop1([], X).$

2. $with\ [X]\ forall\ unique\ [Y, Z]\ in\ p(Y, Z)\ do\ create(q(X, Y, Z))$
   is translated into
   $setof([Y, Z], p(Y, Z), List),\ loop1(List, X)$
   and the following rules are created
   $loop1([[Y, Z]|List], X)\ :-\ create(q(X, Y, Z)),\ loop1(List, X).$
   $loop1([], X).$

3. $with\ [X]\ forall\ possible\ [Y]\ in\ p(Y)\ do\ create(q(X, Y))$
   is translated into
   $temp(Y)\ :=\ p(Y),\ loop1(X)$
   and the following rules are created
   $loop1(X)\ :-\ destroy(temp(Y)),\ create(q(X, Y)),\ loop1(X).$
   $loop1(X)\ :-\ not(temp(Y)).$

Prototype 3 will generally be faster than Prototype 2. However, there is a drawback in this prototype. If new predicates are continuously declared, there will be more and more non-erasable rules residing in the database, and this may affect the total usable executional space. Also, the accumulation of rules is not neat when viewing the contents of the Prolog translation of a $\mathcal{TR}$ transaction base.

4.4.4 Prototype 4

Some transaction programs do not backtrack through updates, e.g., many P-time graph algorithms. Prototype 4 optimizes the $\mathcal{TR}$ implementation for such problems.

Since updates have no backtracking in this prototype, there is no need to use the special tag method to store keys, because this method is implemented to handle update backtracking. Instead, we can directly use *assert* and *retract* in the implementation. However, in order to use the same declaration method, we still have to check that each key has a unique value when a new tuple is created. Checking is performed when inserting a key. Thus, if $p/2$ is declared with 1 argument as key, then "$create(p(1,2))\ sc\ create(p(1,3))$" would fail.

Moreover, since there is no backtracking, we can include the Prolog "*cut*" in some formulas to speed up execution. Also, some Prolog languages like, Quintus Prolog and Sicstus Prolog, provide optimizations on tail recursive rules, i.e., rules of the form:

$$Rule\_head \quad :- \quad Rule\_body,\ !,\ Rule\_head. \tag{4.35}$$

Therefore, in Prototype 4, loops can be translated to tail recursive rules, so that the performance of loops will be optimized by the Prolog interpreter. For example, in Prototype 4, the translations of $while\_loop$ and for_loop are:

1. $while\ Cond\ do\ Body$
   is translated into
   $loop$
   and the following rules are created
   $loop\ :-\ Cond,\ Body,\ !,\ loop.$
   $loop\ :-\ not(Cond).$

2. $with\ [X]\ forall\ [Y,Z]\ in\ p(Y,Z)\ do\ create(q(X,Y,Z))$
   is translated into
   $bagof([Y,Z],p(Y,Z),List),\ loop1(List,X)$
   and the following rules are created
   $loop1([[Y,Z]|List],X)\ :-\ create(q(X,Y,Z)),\ !,\ loop1(List,X).$
   $loop1([],X).$

3. $with\ [X]\ forall\ unique\ [Y,Z]\ in\ p(Y,Z)\ do\ create(q(X,Y,Z))$
   is translated into

$setof([Y,Z], p(Y,Z), List),\ loop1(List, X)$

and the following rules are created

$loop1([[Y,Z]|List], X)\ :-\ create(q(X,Y,Z)),\ !,\ loop1(List, X).$

$loop1([], X).$

Note that the last type of for-loop (*forall possible*) can be implemented exactly as the second type (*forall unique*) because there is no backtracking in Prototype 4. Also, with the use of "*cut*" in these interpreted rules, choice points can be released in each loop iteration, which releases space from the choice point stack.

4.5 Prolog Interpreters

The $\mathcal{TR}$ implementation was designed to run under XSB Prolog, which provides facilities for user-defined indexes on any combination of arguments of a predicate. When a $\mathcal{TR}$ program is running in other Prolog environments, some facilities which are provided by XSB Prolog, such as indexing, might not be available.

We have separated the implementation into a number of modules. To port the implementation from one Prolog interpreter to another, minor adjustments must be made to each module. To minimize and isolate the changes each module, the *basic_funs* module contains the most basic rules which may need to be modified. So, most changes will be to this module.

To illustrate the modifications, we will show the changes necessary to run the program in Quintus Prolog or in Sicstus Prolog. Also, we will briefly describe some special features provided by these interpreters which are used in our implementations.

4.5.1 XSB Prolog

XSB Prolog is a research-oriented Logic Programming system for Unix-based systems. It was designed to be used as a module-oriented Prolog system. A module contains one or more export declarations, which specify the set of visible predicates that can be used by other modules. It may also contain import declarations, which allow predicates defined and exported by other modules to be used in the current module. Export and import declarations have the following forms:

:- export $pred_1,\ pred_2,\ \ldots,\ pred_m.$

:- import $pred_1$, $pred_2$, ... , $pred_n$ from *module*.

where each $pred_i$ has the form $functor/arity$.

Using modules, it is easier for us to maintain the $\mathcal{TR}$ implementation when making changes.

Another useful feature provided by XSB Prolog is indexing. Most Prolog compilers put an index on the principal functor of the first argument of a predicate, but XSB Prolog allows a user to put indexes on any argument or combination of arguments by means of an index directive. This directive has the form:

$$:- index(Functor/Arity,\ IndexArgSpec,\ HashTableSize).$$

which allows users to specify the size of the hash table to use for indexing a predicate. In this directive, $IndexArgSpec$ is either an integer, indicating the argument to be indexed, or a list of $IndexArgs$. An $IndexArg$ may be an integer, indicating the argument to be indexed, or a sequence of up to three integers separated by $+$. The latter indicates that an index is to be built a combination of arguments.

As an example, the command $:- index(p/3,\ [1+2,1,3],\ 100)$ specifies an index on predicate $p/3$. With this declaration, three hash tables are created, and each of them has 100 entries. A call to $p/3$ first checks to see if both the first and second arguments are bound, and if so, uses an index based on these two values. Otherwise, it checks if the first argument is bound, and if so, uses an index based on it. If not, it checks if the fourth argument is bound, and if so, uses an index based on it. Otherwise, it uses no index, and searches through all the clauses for this predicate in the database. Users should be aware of the size of hash tables. For example, if $p(X)$ has 100 entries in database, then setting the hash table size to 10,000 can severely affect tuple access time.

If a predicate is called before it is defined, most Prologs will display error messages, and some will even abort the execution. In our implementation, when creating a tuple, we have to check if the tuple is already in the database. If it exists, then the creation fails. However, if this is the first time a tuple is created after its predicate is declared, then during the checking, some warning messages may appear to inform users the predicate has not been defined, or the execution may abort. To avoid such behavior, when a predicate is declared, all database predicates (i.e., all tag and value predicates) should be defined dynamically (i.e., so they can be retracted from the Prolog database). For example, in executing the declaration $p/2$ *key* 1, the tags $i\_key\_2\_p/1$ and $d\_key\_2\_p/1$ together with the value predicate $val\_2\_p/2$ are defined as dynamic predicates so that calls to $p(X,Y)$ can fail quietly. In the implementation, we define $dynpred(X)$ for this function. In XSB Prolog, since the built-in predicate *dynamic/1* performs the required function, $dynpred/1$ can defined by the $dynpred(X) \ :- \ dynamic(X)$. Since $dynamic/1$ may be a built-in predicate with other purpose in other Prologs, like Quintus Prolog, we do not use it directly in the implementation, so that we can minimize the changes in the implementation when porting it to other Prologs.

4.5.2 Sicstus Prolog

Sicstus Prolog was developed at the Sweden Institute of Computer Science. It consists of a Warren Abstract Machine emulator written in C, a library and runtime system written in C and Prolog, and an interpreter and a compiler written in Prolog. To run a $\mathcal{TR}$ program in Sicstus Prolog, the following changes should be made to the XSB implementation of $\mathcal{TR}$:

- Since Sicstus Prolog does not support modules, all export or import declarations from the source code should be removed from the code.
- Sicstus Prolog indexes a predicate according to the principal functor of the first argument in the head of the clause, and does not allow a user to define hash table size. So, rules for declaring the index information should be removed from the key_basics module.
- Sicstus Prolog does not provide any built-in predicate similar to $dynamic/1$ in XSB Prolog. So, a user-defined rule for $dynamic$ should be written. In our Sicstus implementation of $\mathcal{TR}$, the predicate $dynpred/1$ is defined in the $basic\_funs$ module as follows:

$$
\begin{aligned}
dynpred(X) \quad :- \quad & retract(X),\ !,\ assert(X) \\
& ; \\
& true.
\end{aligned}
\tag{4.36}
$$

In Sicstus Prolog, when $retract(Y)$ is called, Y is defined to be dynamic no matter whether the execution succeeds or not. So, the predicate $dynpred/1$ defines a predicate to be dynamic by intentionally retracting the predicate from database. For instance, when

$dynpred(p(X))$ is called, the system first defines $p(X)$ to be dynamic by trying to retract $p(X)$ from database. If $p(X)$ has a tuple in the database, $retract(p(X))$ retracts the tuple, and then re-asserts it without affecting the content of the database. If $retract(p(X))$ fails, the execution succeeds with the disjunction to the subgoal $true$.

4.5.3 Quintus Prolog

Like Sicstus Prolog, Quintus Prolog does not support modules, and therefore, import and export declarations should be removed from our XSB implementation of $\mathcal{TR}$ before porting it to Quintus. Like Sicstus Prolog, Quintus Prolog indexes predicates on the principal functor of the first argument, and does not allow users to define their own indexes and hash tables. Thus, all indexing predicates should also be removed.

Unlike Sicstus Prolog, if a predicate is non-dynamic, then $retract/1$ does *not* define the predicate to be dynamic. So, another rule defining $dynpred/1$ is used:

$$\begin{array}{lll} dynpred(X) & :- & predicate\_property(X, (dynamic)),\ !,\ true \\ & & ; \\ & & assert(X),\ retract(X). \end{array} \tag{4.37}$$

Quintus Prolog provides a useful pre-defined predicate $predicate\_property/2$ to check the status of a predicate. The goal $predicate\_property(X, (dynamic))$ returns true if X is defined as a dynamic predicate. If the predicate is dynamic, the execution succeeds. If the predicate is not dynamic, we deliberately assert the predicate into database, which defines the predicate dynamically. To keep the consistency, the predicate is removed immediately, so that the database is unchanged afterwards. Note that suppose $p(1)$ is already in the database, by definition, $p(X)$ is dynamic. The execution of $dynpred(p(X))$ will not cause $p(1)$ to be retracted because $predicate\_property(p(X), (dynamic))$ is true.

Chapter 5

Testing and Analysis of the Performance

In this chapter, we test the performance of the $\mathcal{TR}$ implementation under *XSB Prolog* (Version 1.4.1, 1994), *Quintus Prolog* (Version 3.1, 1994), and *Sicstus Prolog* (version 0.6, 1987). Tests on XSB Prolog and Quintus Prolog are carried out in Sun Sparc-5, 32MB Main Memory, 64MB Virtual Memory. Although some tests are carried out on Sicstus Prolog, since it is an older version and is running on a different machine (Sun Sparc-10, 64MB Memory), it is used as a reference only. In the following tests, most of them involve using graph algorithms, with the graph stored in the database. The tests are divided into several categories, each showing a different feature of the implementation. These categories are:

1. simple updates;

2. connected components, which tests the performance in large, growing databases in which many tuples are created and destroyed;

3. all pairs shortest paths problem, which shows the use of keys in defining tuples and is the first test with non-linear complexity;

4. counter, which tests the use of the "*set*" command and the performance of recursion depth on a small, fixed database; and

5. backtracking, which consists of two tests: Hamiltonian cycle and clique, which test backtracking and undoing of updates in small and large database, respectively.

Some notes on the tests:

1. All tests were performed as the only application running on the machines to avoid any possible effect from other processes running on same machine.

2. Each test was run once only, so possible variance in results from run-to-run is unknown.

3. A fresh Prolog session was started for each test.

4. Runtime was measured by executing $statistics/1$ (a standard Prolog predicate (before and after each transaction query. The predicate statistics is defined in most Prolog (including XSB, Quintus, Sicstus and C Prolog) which prints a lot of information on the use of memory, including cputime usage since the start of the Prolog session. The difference in cputime between two call to statistics is the runtime of the transaction query.

5. All tests on XSB Prolog started with the allocation of 30,000 Kbytes to local/global stack area and 15,000 Kbytes to trail/choice-point stack area, i.e., XSB is started with the command *xsb -i -m 30000 -c 15000.*

6. Tuples were declared with suitable size for reserved space. For example, in Section 5.1, reserved space for $p/1$ should be the number of tuples to be created in the test. These indexes have effect on tests in XSB Prolog only. For Quintus Prolog and Sicstus Prolog, predicates have indexes on the first argument by default.

In all graphs, results of Prototype 3 in XSB Prolog, Quintus Prolog, and Sicstus Prolog are labelled by XSB3, Quintus3, and Sicstus3 respectively. Results of Prototype 4 are labelled by XSB4, Quintus4, and Sicstus4 for the corresponding prototypes.

In the theory of algorithms, graphs are often represented as arrays, so that any node or edge can be accessed in constant time. Using the $\mathcal{TR}$ program, by declaring proper indexes on graph predicates, tuples can be retrieved in constant time. With this reason, graph problems can be run in expected time in XSB version of $\mathcal{TR}$.

All test results are plotted in two dimensions. We use three types of plots:

1. time versus number of nodes, edges, or components, etc.;

2. logarithm of time versus logarithm of number of nodes, edges or components etc., (*log-log* graph); and

3. time versus logarithm of number of nodes, edges or components, etc., (*log* graph).

The first type of plot gives a general picture of the complexity of a program. A straight line means the complexity is linear. The log-log graph is used to find the degree of the polynomial when t he complexity is polynomial time. For instance, suppose for an algorithm with polynomial complexity,

$$\begin{aligned} t &= n^p \\ log(t) &= p\ log(n) \end{aligned}$$

Then, if the graph of *log(t)* versus *log(n)* is a straight line, the complexity of the program is polynomial and the slope of the graph is the degree of polynomial. Finally, The *log* graph is used to find the degree of the exponent. For an algorithm with exponential complexity,

$$\begin{aligned} t &= a\ k^{bn} \\ log(t) &= log(a)\ +\ n\ b^{log(k)} \end{aligned}$$

Therefore, if the graph of *log(t)* versus n is a straight line, the complexity is exponential with the slope of the line being $b\ log(k)$ where b is the degree of exponent and k is the base of exponent. Some notes for the analysis:

1. Logarithm with base 10 is used in this chapter.

2. Only Prototype 3 and Prototype 4 are tested because Prototype 1 and Prototype 2 are designed for development purpose only.

3. Slopes of each graph are estimated using a least-square fit method, i.e.,

$$slope = \frac{\sum x\ \sum y\ - n\ \sum xy}{(\sum x)^2\ - n\ \sum x^2}$$

   where n is the number of points in the graph

4. In some of the tests, the complexity is measured in terms of the number of edges rather than the number of nodes. Since we have chosen the graphs with the number

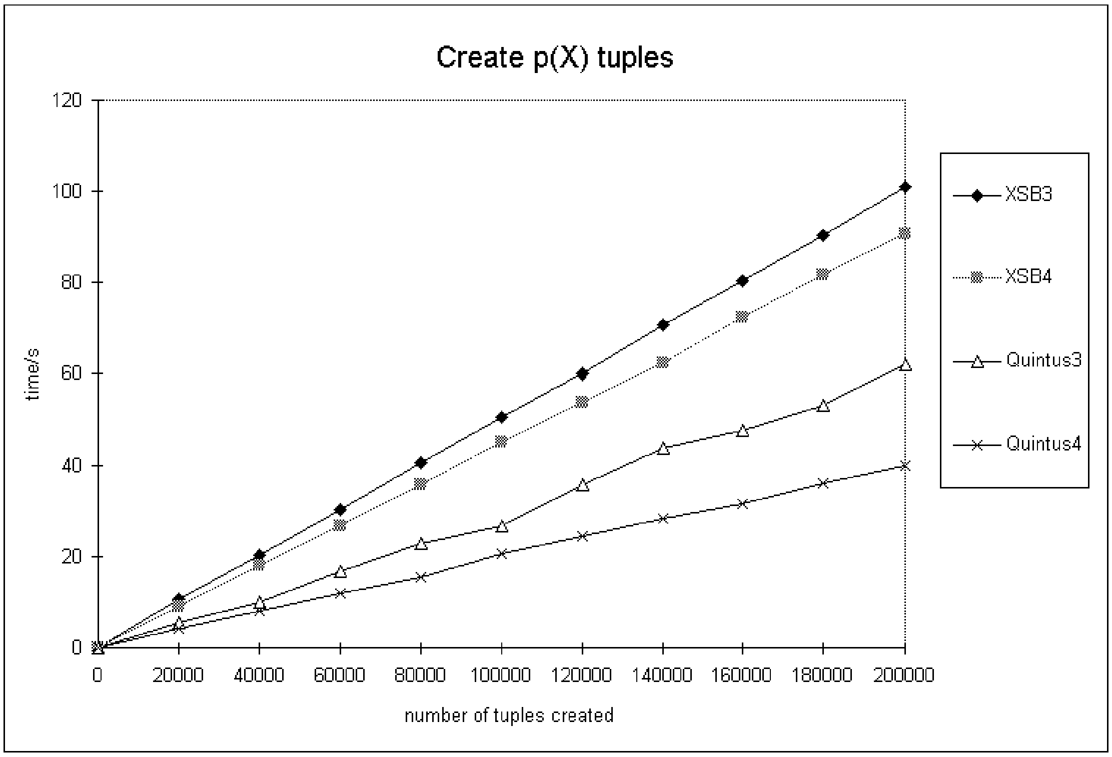


Figure 5.1: Graph for time required to create a number of p(X) tuples.

of nodes proportional to the number of edges, the complexity of a problem in terms of the number of edges is the same as in terms of the number of nodes.

5.1 Basic Updates

Insertions:

In $\mathcal{TR}$ programs, an insertion of a tuple is different from an assert in Prolog, since it takes an extra step to check if the tuple already exists in the database. To test the implementation with different prolog systems, we first compare their performance in inserting a number of tuples. We use the following $\mathcal{TR}$ program:

$$declare\ p/1\ size\ 20000.$$
$$test1(0).$$
$$test1(N) \leftarrow create(p(N))\ sc\ M\ is\ N-1\ sc\ test1(M).$$

If each tuple creation takes constant time, then the time for the execution of $test1(N)$ should be linear in N. *Figure 5.1* shows the performance of the program in Prototype 3 and

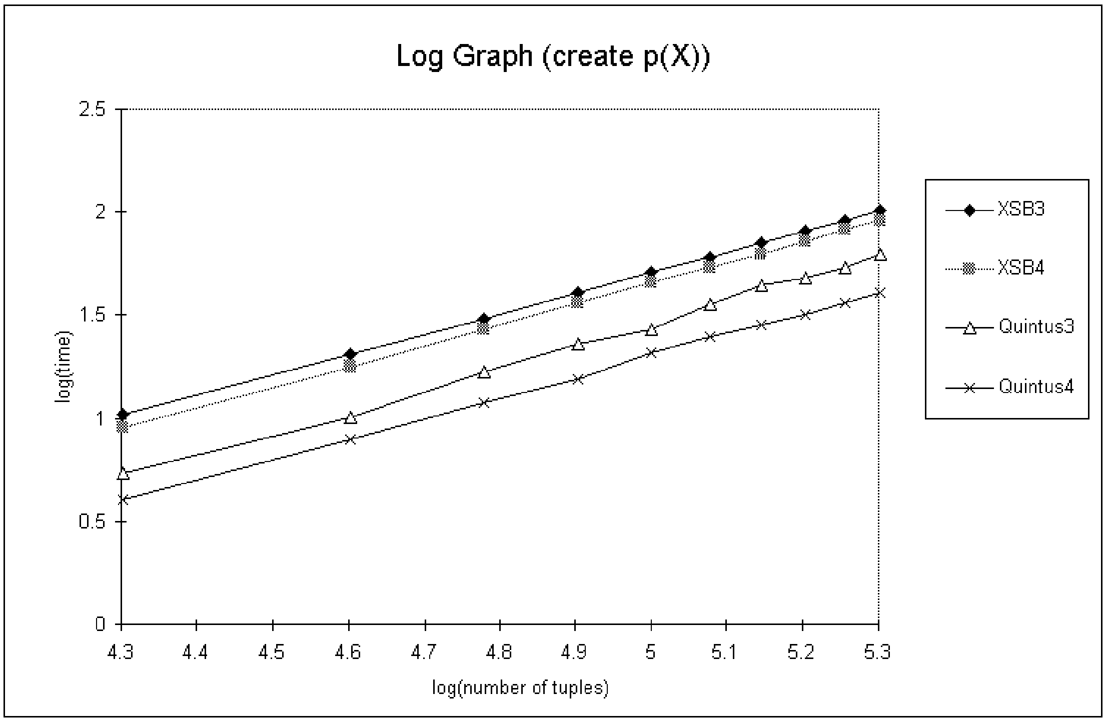


Figure 5.2: Log table of time required to create p(X) tuples

slope(XSB3) ≈ slope(XSB4) ≈ slope(Quintus3) ≈ slope(Quintus4) ≈ 1.0

Prototype 4. Since the graphs in the $log - log$ plot in *Figure 5.2* are linear, the complexity of all implementations is polynomial. In all cases, the slopes are approximately 1 and the degree of polynomial is 1, i.e., linear. Thus, the time required to create a single tuple is constant. Note that we did not perform this test on Sicstus Prolog because of its lengthy execution time. In both Sicstus prototypes, it takes over 30 minutes to create just 20,000 tuples!

Indexes:

The previous program creates a number of p-tuples in the database. As expected, the execution time increases linearly with the number of p-tuples. However, consider the following slightly different example for creating 20,000 binary tuples:

$declare\ p/2\ size\ 20000.$

$test2(0).$

$test2(N) \leftarrow\ create(p(1,N))\ sc\ M\ is\ N-1\ sc\ test2(M).$

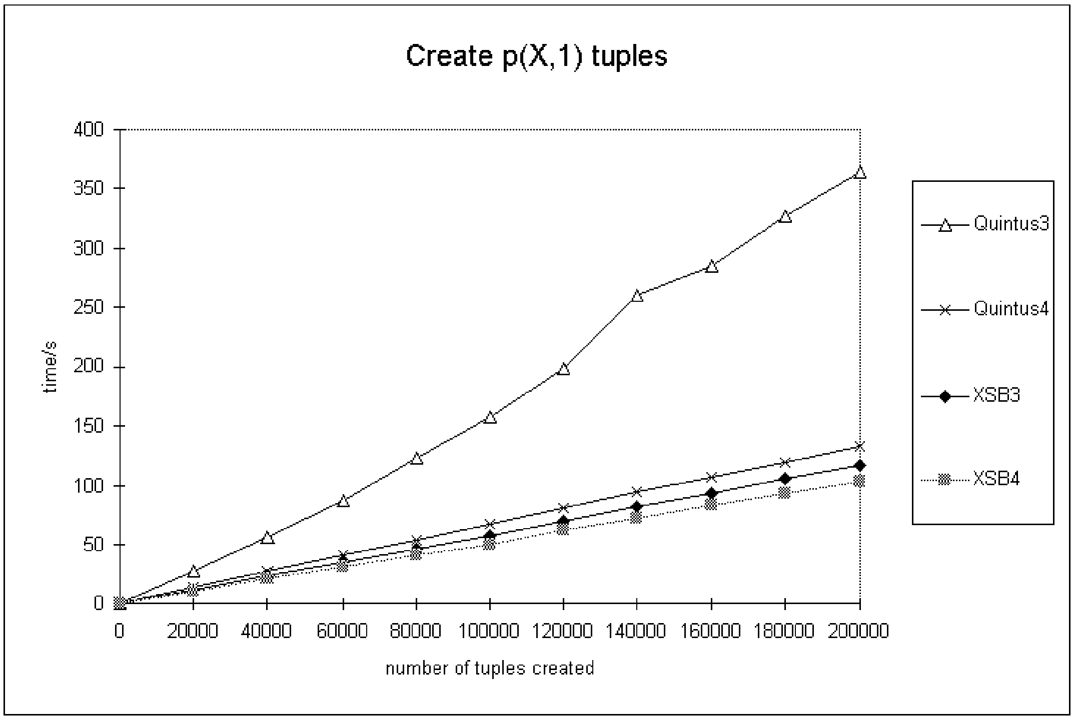


Figure 5.3: Graph of creating p(N,1) tuples.

The program works well in XSB Prolog. However, in Quintus Prolog, the program takes over half of an hour in both prototypes 3 and 4 to create just 20000 tuples! The poor results are due to the default index in Quintus Prolog, which is set on the first argument. Since the first argument of $p/2$ is always 1 in this example, searches in database are exhaustive in this case. If we interchange the arguments of $p/2$, we obtain the graph in *Figure 5.3* and *Figure 5.4*, then complexity is linear in Quintus Prolog as well.

Alternating Inserts and Deletes:

Apart from inserting tuples, we can also test the performance of the program by repeatedly creating and destroying a tuple. In the next example, there is at most one tuple exists in database at any time. However, we should still reserve considerable tuple space in Prototype 3 (but not in Prototype 4) because there are N insert tags and delete tags for the key inserted in the database (where N is the number of of tuples).

$declare\ p/1\ size$ 20000.

$test3(0).$

$test3(N) \leftarrow\ create(p(N))\ sc\ destroy(p(N))\ sc\ M\ is\ N-1\ sc\ test3(M).$

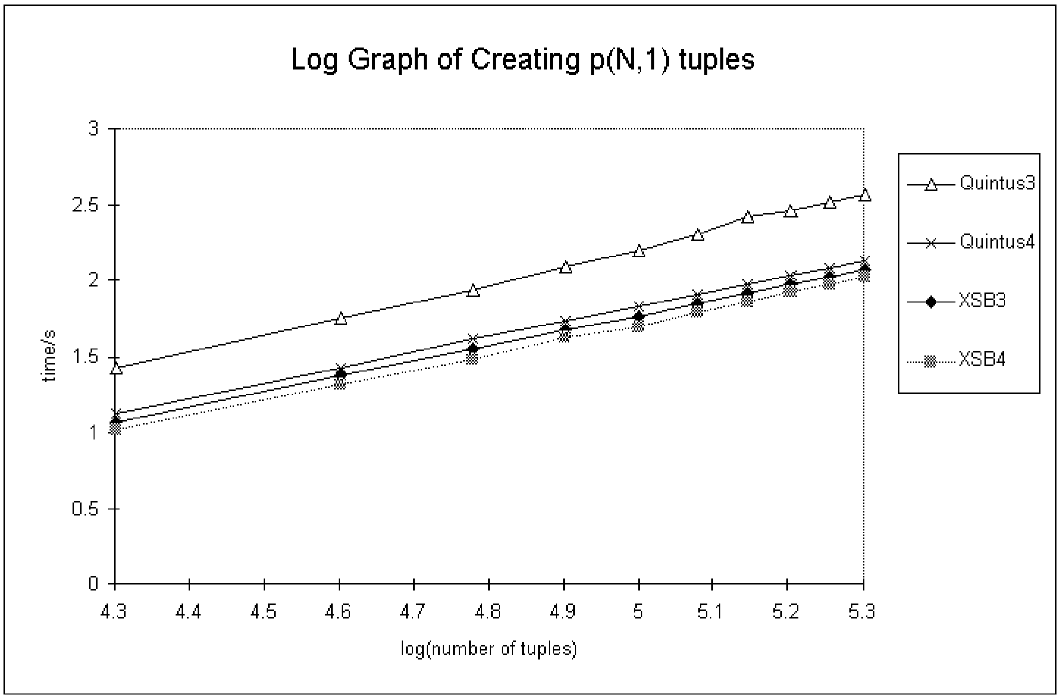


Figure 5.4: Log-log graph of creating p(N,1).

slope(XSB3) = 0.998 slope(Quintus3) = 1.152
slope(XSB4) = 0.994 slope(Quintus4) = 0.994

The result is shown in *Figure 5.5* and *Figure 5.6*. The difference between Prototype 3 and Prototype 4 is that in Prototype 4, tuples are removed from database immediately after they are created but in Prototype 3, a delete_tag on the key of the tuple is added to remove a tuple from database. So, in general, Prototype 4 should be faster because database size remains fairly constant (and small) throughout the execution.

Interestingly, the above transaction program has better performance in Prototype 3 than Prototype 4 on XSB prolog. This shows that it is faster to assert a predicate than to retract it from database. In the implementation, *retract*/1 is used to remove a predicate from database. But in XSB prolog, *retract*/1 reclaims program space every time it is executed. Another predicate *retract_nr*/1 is provided by XSB prolog which can remove a predicate from database without reclaiming program space. It has faster execution time, at the expense of no space reclamation.

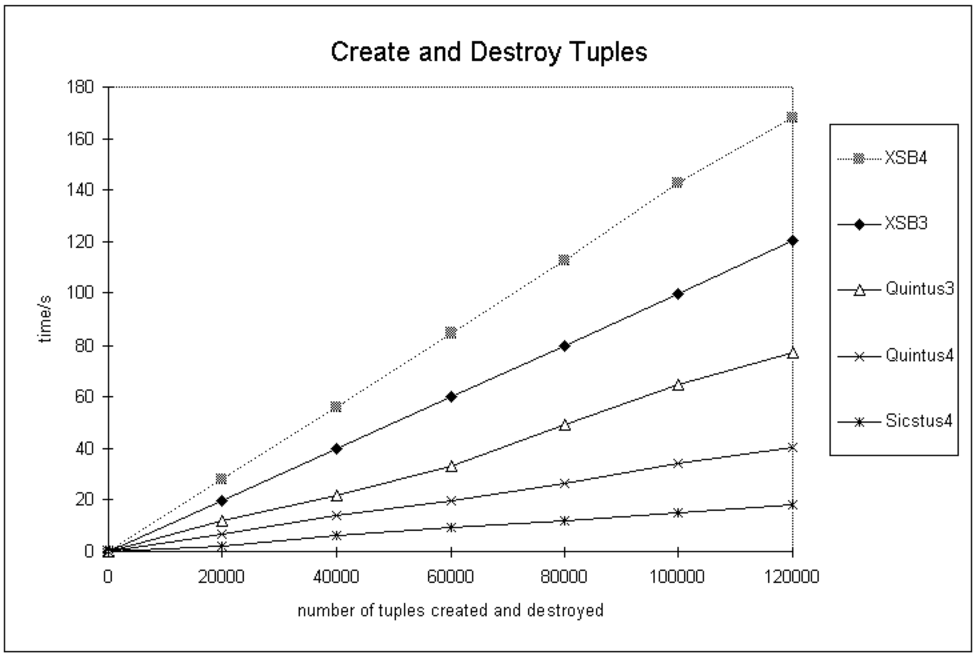


Figure 5.5: Results of repeatedly creating and destroying a tuple.

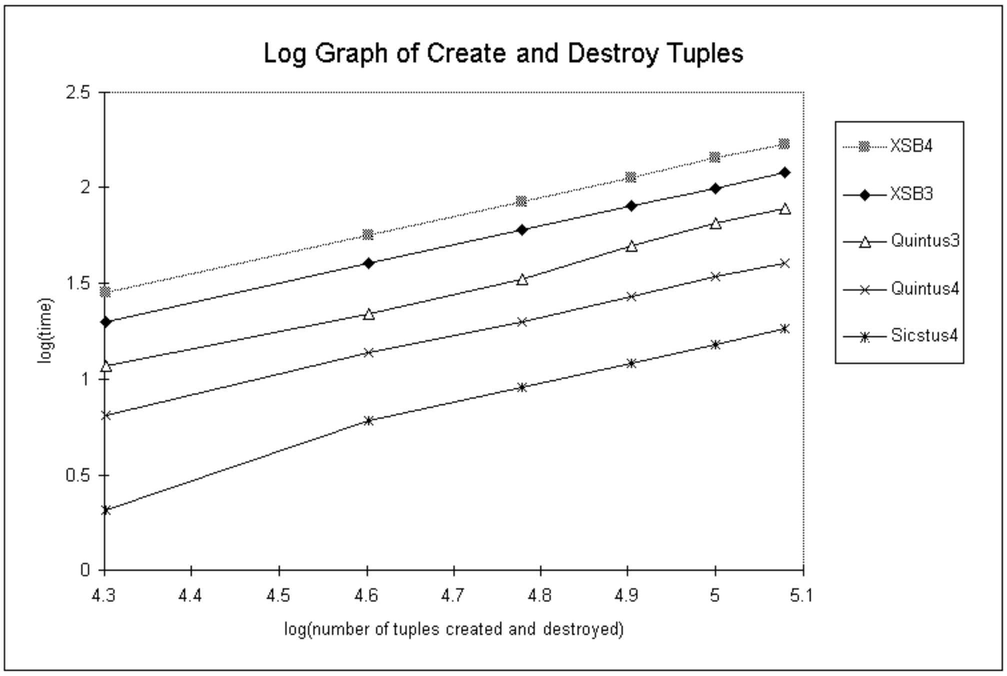


Figure 5.6: Log-log graph of repeatedly creating and destroying tuples.

slope(XSB3) = 1.004 slope(Quintus3) = 1.069
slope(XSB4) = 1.001 slope(Quintus4) = 1.014 slope(Sicstus4) = 1.19

Consider the results of Prototype 4 only, surprisingly, Sicstus Prolog has the best performance. There are three possible explanations: first, Sicstus prolog has optimizes tail recursion; second, retraction of tuples is the fastest in this Prolog; third, Sicstus Prolog has fast updates on small database. However, comparing the results of Prototype 3, Sicstus Prolog has the worst performance (which is not shown in the graph). The execution of $test3(20000)$ requires over 2000 seconds to complete whereas same executions on other Prologs require less than 20 seconds as shown in the graph. One can explain this result as the slow search time for large database in Sicstus Prolog (or more exactly, slow search time for predicates with large hash tables).

5.2 Bulk Updates

Bulk Insertion:

One of the new features provided by Transaction Logic the ability to perform bulk updates. It provides a convenient way to copy one set of tuples to another. The following example shows the performance of the program on bulk inserts from $p(X,Y)$ to $q(X)$. The tuple $p(X,Y)$ is declared with the first argument as key, and the whole tuple of $q(X)$ is declared as the key. So, if the database contains $p(1,1), p(2,1), p(3,1), \ldots, p(N,1)$, $bulk\_ins(p(X,Y), q(X))$ should create $q(1), q(2), q(3), \ldots, q(N)$ in the database.

Bulk Deletion:

Figure 5.7 shows that bulk insertion is linear in all cases. However, when considering bulk deletion, in *Figure 5.9*, the complexity is also linear but Prototype 4 performs much better than Prototype 3. The reason is that without the need for backtracking, the built-in directive $retractall/1$ is used to bulk-delete tuples. So, it is a subject for future work to develop a more efficient implementation of $bulk\_del/1$ in Prototype 3.

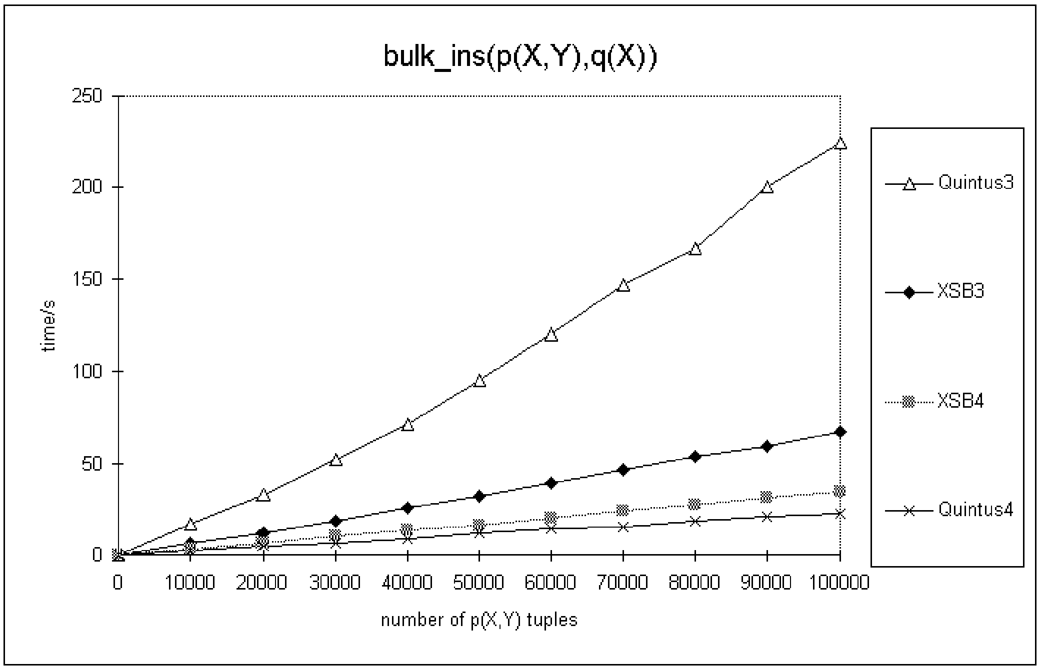


Figure 5.7: Graph of the performance of bulk_ins(p(X,Y),q(X)).

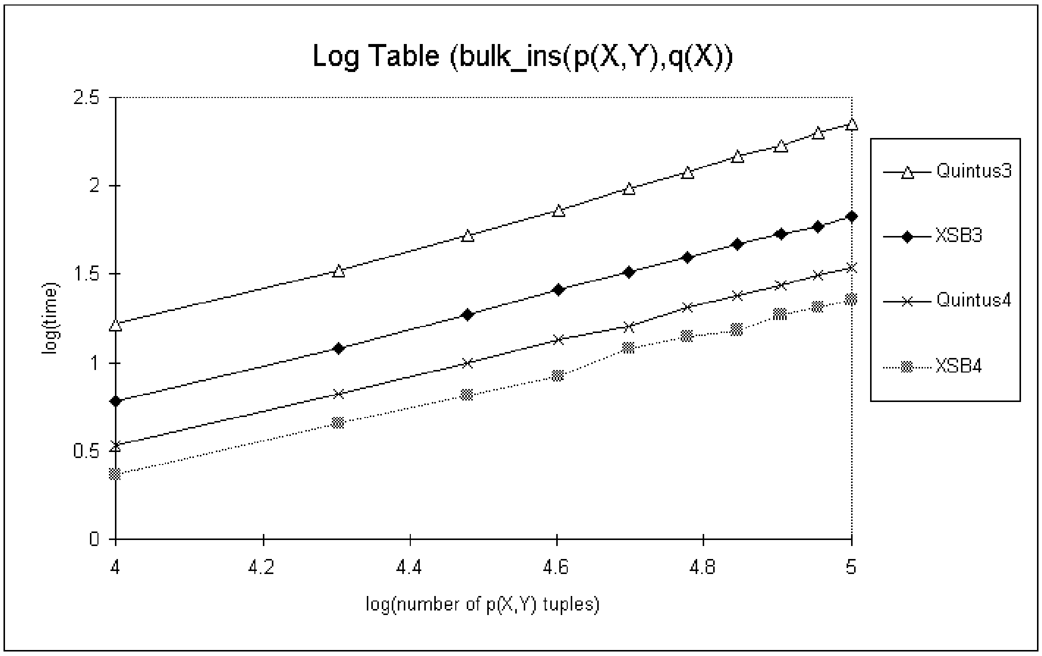


Figure 5.8: Log table of the performance on bulk_ins(p(X,Y),q(X)) with N p(X,Y) tuples.

slope(XSB3) = 1.05 slope(Quintus3) = 1.15
slope(XSB4) = 1.00 slope(Quintus4) = 1.00

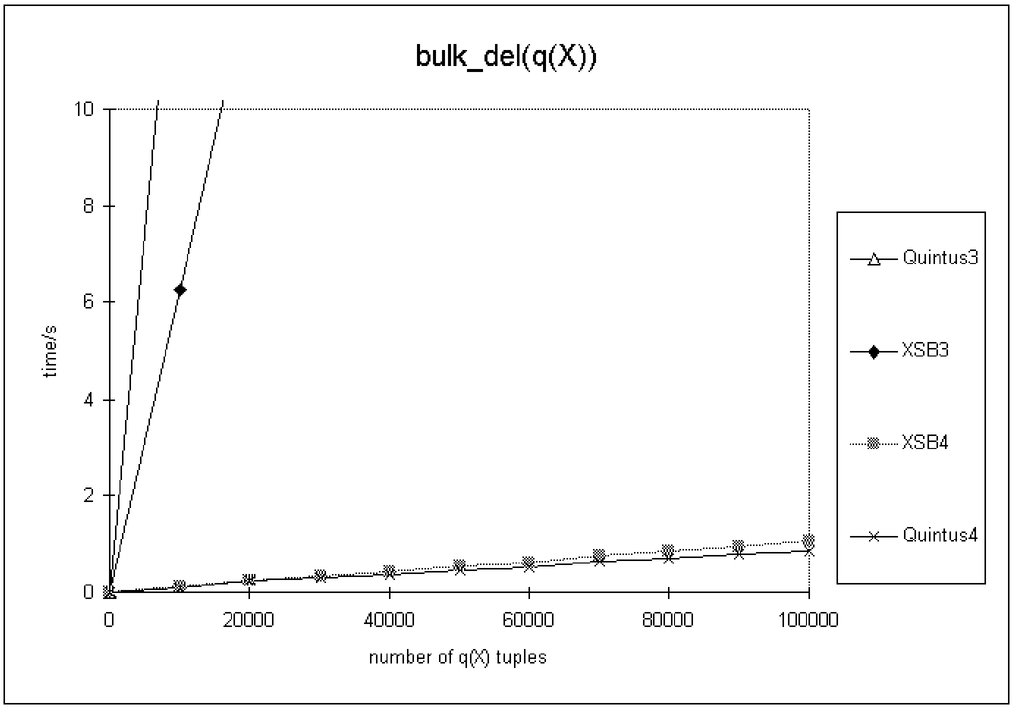


Figure 5.9: Graph of comparing the performance on bulk deletion of N q(X) tuples.

5.3 Connected Components

Having tested some basic database updates, we now test compare the performance of the $\mathcal{TR}$ implementation on some graph problems. The program for finding connected components uses a while_loop to repeatedly apply depth-first search to all unvisited nodes. During execution, each edge is traversed once, so the complexity should be linear in the number of edges, and thus linear in the number of nodes for the graphs used in our tests. It is interesting to note that the program exhaustively searches all the nodes and there are no choice points in the rules, so there is no backtracking during the execution. We can therefore use the Prototype 4 on this problem for faster speed.

$declare\ unvisited/1\ size\ 1000.$

$declare\ component/2\ size\ 1000.$

$$\begin{array}{ll} findComponents & \leftarrow unvisited(P)\ :=\ node(P)\ sc \\ & \quad while\ unvisited(N)\ do\ dfs(N,N). \\ dfs(N,N1) & \leftarrow create(component(N,N1))\ sc \\ & \quad destroy(unvisited(N1))\ sc \\ & \quad with\ [N,N1]\ while\ (edge(N1,N2),\ unvisited(N2))\ do\ dfs(N,N2). \end{array}$$

The above rules show the $\mathcal{TR}$ program for finding the connected components of a graph with 1000 nodes and a single component. The database for the graph is loaded explicitly, and space for the nodes and edges is reserved during the declaration.

The graph used in the test is just a simple cycle, i.e., a line in as the following figure:

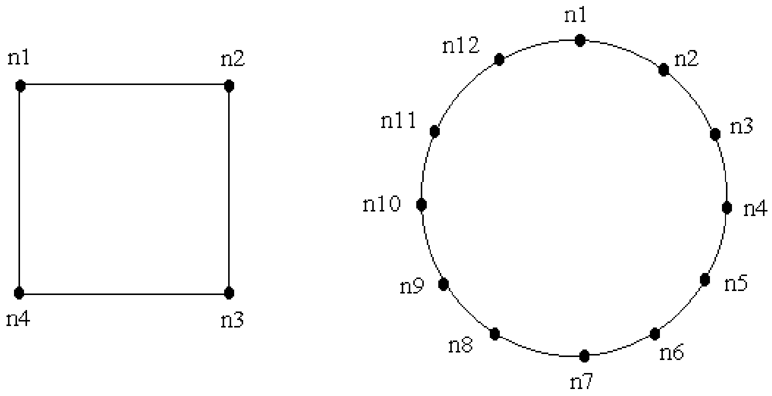


Figure 5.10: Cycles with 4 nodes and 12 nodes.

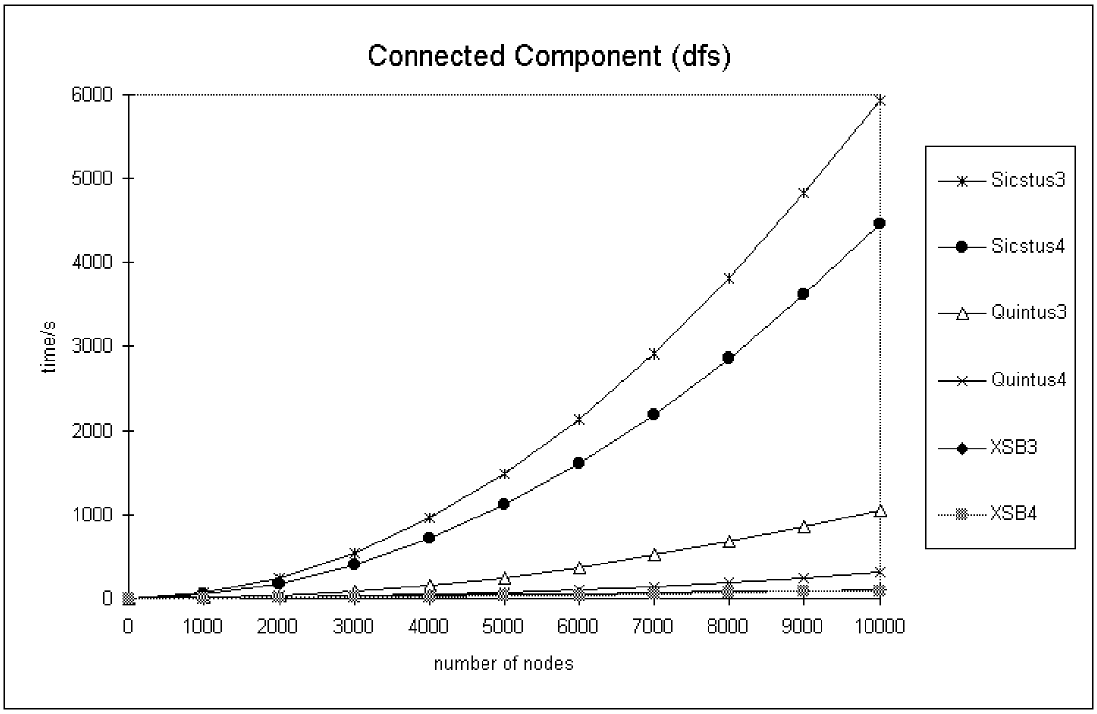


Figure 5.11: Graph comparing the performance for finding connected components.

The graph has only 1 component and the number of edges is proportional to the total number of nodes. It is clear from the program that each node is visited once during the execution. Since there is no backtracking involved in the execution, we can use Prototype 4 for the test. The program was tested by varying on the number of nodes, and the complexity of the program is expected to be linear.

The graph of time versus number of nodes, and the corresponding log-log graph, are plotted in *Figure 5.11* and *Figure 5.12*.

From *Figure 5.11*, as expected, the program is running faster in Prototype 4 than in Prototype 3. The improvement is not too significant because the declared predicates, *unvisited*/1 and *component*/2, use their arguments as a key. In this case, the cost of adding tuples to database in Prototype 3, is similar to that of Prototype 4. For this problem, the speed of each Prolog can be ranked from fastest to slowest as follows: XSB > Quintus Prolog > Sicstus Prolog.

Consider the graph in *Figure 5.12*, the complexity of Prototype 3 in XSB is linear, as we expected. However, the degree of polynomial is 1.3 in Prototype 4. One explanation for such poor behavior is that as the number of nodes increases, the database size increases

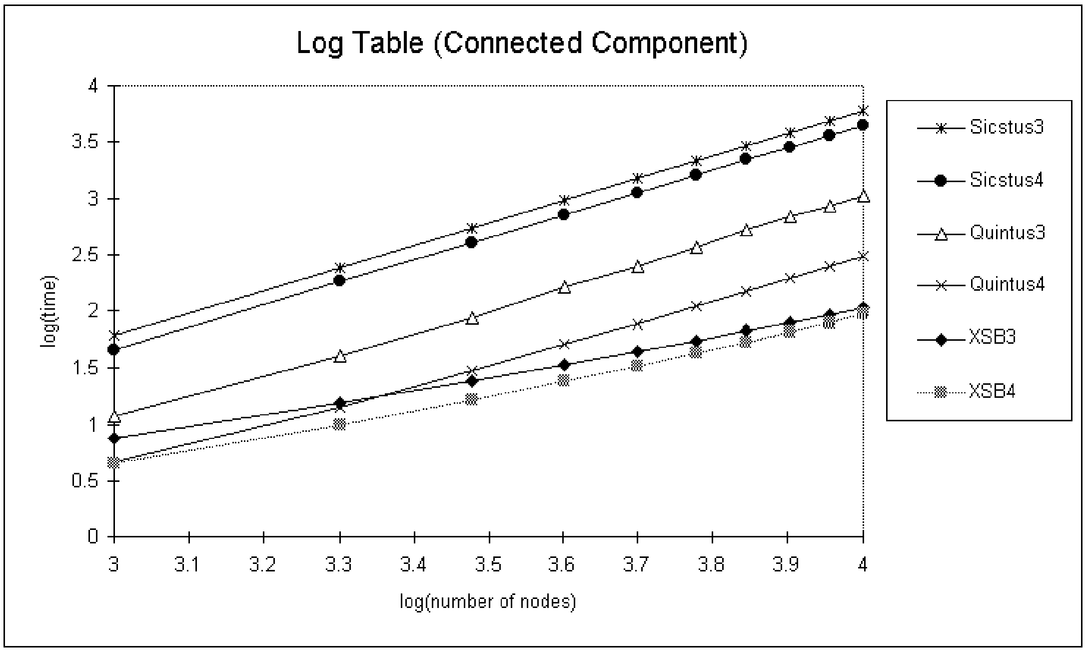


Figure 5.12: Graph of log(number of nodes) against log(time) for the connected component test.

slope(XSB3) = 1.157 slope(Quintus3) = 1.982 slope(Sicstus3) = 1.983
slope(XSB4) = 1.335 slope(Quintus4) = 1.834 slope(Sicstus4) = 1.989

linearly, but the time required for allocating trail points, check points, etc., does not increase linearly.

Apart from the slight defect of Prototype 4 in XSB Prolog, abnormal results also appear for both prototypes in Sicstus Prolog and Quintus Prolog. As shown in *Figure 5.11*, the complexity of the program becomes quadratic under these two languages. The reason is that since there is limited indexing in Sicstus Prolog and Quintus Prolog, the search times for some predicates are exponential. This is significant for the predicate "*component*/2". The first argument of component is regarded as the component name, and the second argument is a node in the component. In this example, since there is only 1 component, the number of tuples for component is same as the number of nodes, but the first argument of component is always the same. When the default index is set on the first argument for Sicstus Prolog and Quintus Prolog, the search time for the component predicate increases quadratically with the increase in the number of nodes.

In this case, there is a way to overcome the quadratic complexity by interchanging the arguments of *component*/2. The $\mathcal{TR}$ program then becomes:

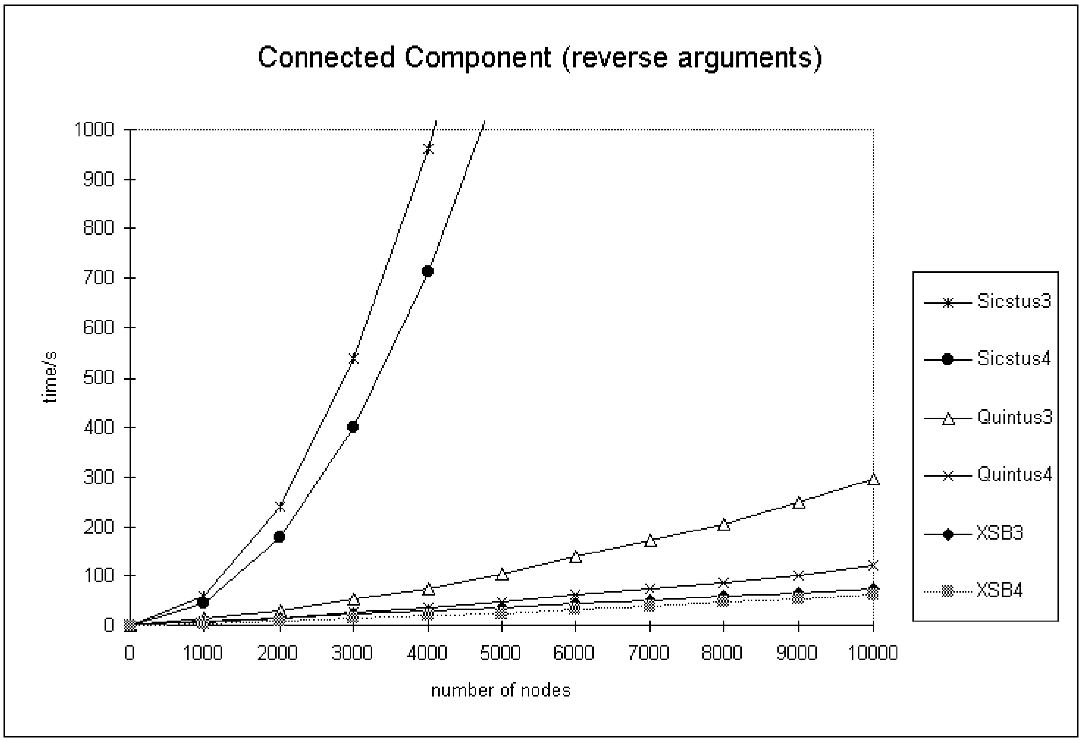


Figure 5.13: Graph comparing the performance of finding connected components after reversing the arguments in component/2.

$$
\begin{array}{lll}
\multicolumn{3}{l}{declare\ unvisited/1\ size\ 1000.}\\
\multicolumn{3}{l}{declare\ component/2\ key\ 1\ size\ 1000.}\\
findComponents & \leftarrow & unvisited(P)\ :=\ node(P)\ sc\\
 & & while\ unvisited(N)\ do\ dfs(N,N).\\
dfs(N,N1) & \leftarrow & create(component(N1,N))\ sc\\
 & & destroy(unvisited(N1))\ sc\\
 & & with\ [N,N1]\ while\ (edge(N1,N2),\ unvisited(N2))\ do\ dfs(N,N2).
\end{array}
$$

In this way, the indexing will be on the name of the node rather than the name of the component. Since each node is in one and only one component, the search time should be linear. The result of testing this program is plotted in *Figure 5.13* and *Figure 5.14*.

As shown in the above graphs, the performance of XSB Prolog is roughly the same in both versions of the problem. However, the performance in Quintus Prolog has improved a lot. In the second version, the complexity is much closer to linear (degrees of polynomial are 1.35 and 1.2 for Prototype 3 and Prototype 4, respectively). This means that the order

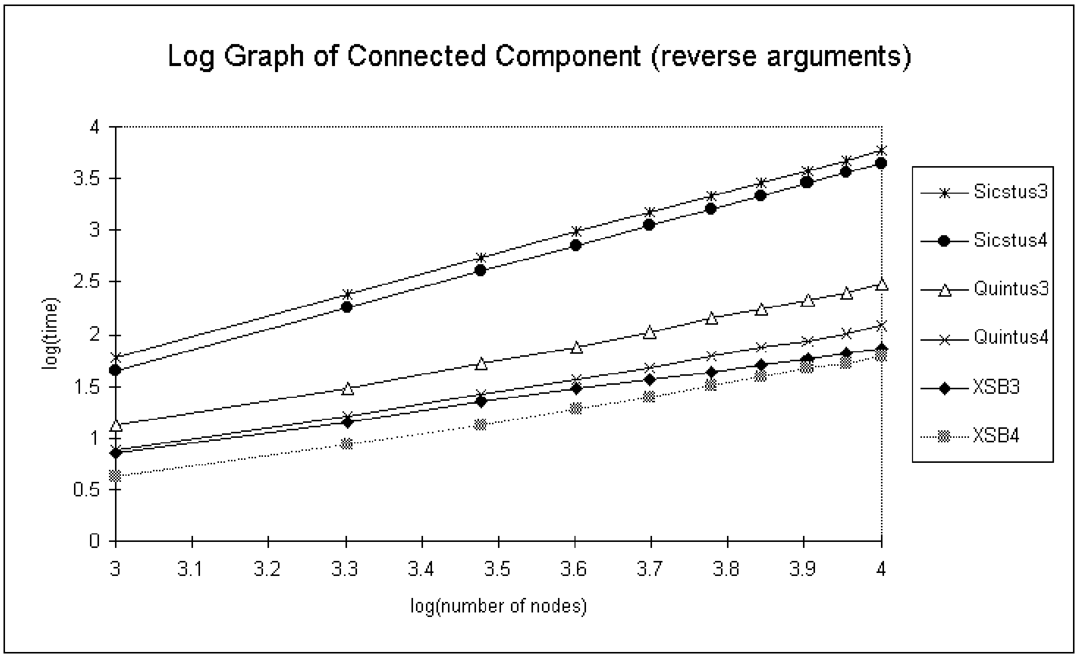


Figure 5.14: Graph of log(time) versus log(number of nodes) for the problem of finding connected components after reversing the arguments in component/2.

slope(XSB3) = 1.02	slope(Quintus3) = 1.352	slope(Sicstus3) = 1.985
slope(XSB4) = 1.181	slope(Quintus4) = 1.204	slope(Sicstus4) = 1.989

of arguments in storing tuples is important for running $\mathcal{TR}$ program in Quintus Prolog. Although Sicstus Prolog has a default index on the first argument, the complexity of the Second version running on Sicstus Prolog are still quadratic in both prototypes. The poor performance can be explained by Sicstus's slow tuple access time in large databases as shown in Section 5.1. Although the first argument of the predicate is indexed, the improvement is not significant enough to overcome the deficit of the increase in search time caused by the growth of the database.

One may complain that the graphs are too simple in the previous test, and doubt if the program will work well on more complex graphs. To address this issue, we have performed some tests on cycles of graphs with a clique-size of 4 (as in *Figure 5.15*).

Figure 5.16 and *Figure 5.17* shows the results of executing the second version of connected components program to a cycle of cliques with clique-size 4. As a result, the performance of the programs using the clique graphs is very similar to that of using the simple cycle.

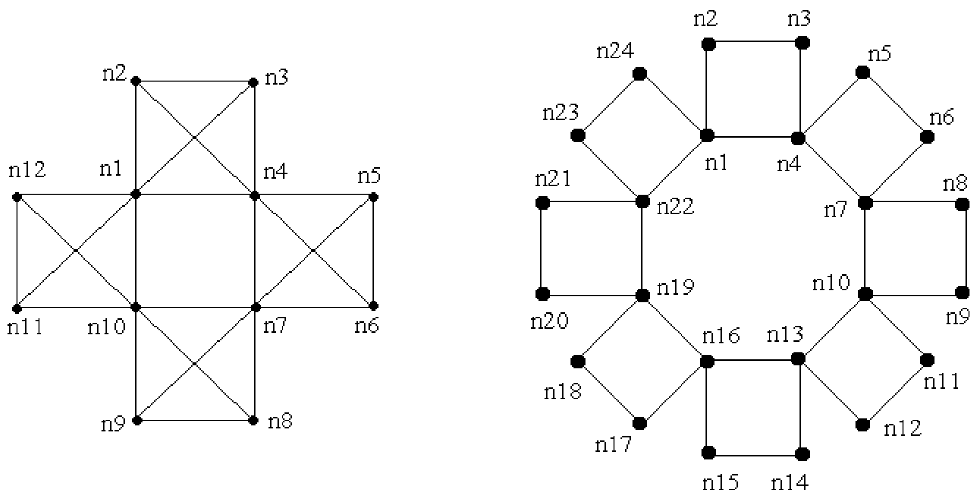


Figure 5.15: Cycles of 4 cliques and 8 cliques with clique-size 4.

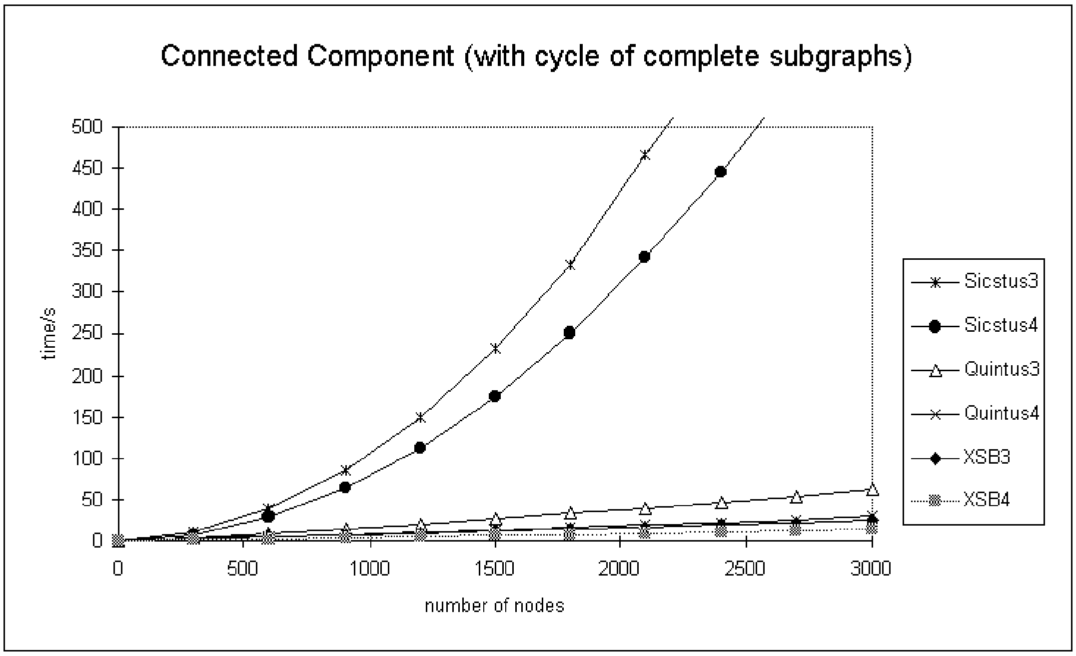


Figure 5.16: Performance of finding connecting components using a cycle of cliques with clique-size 4

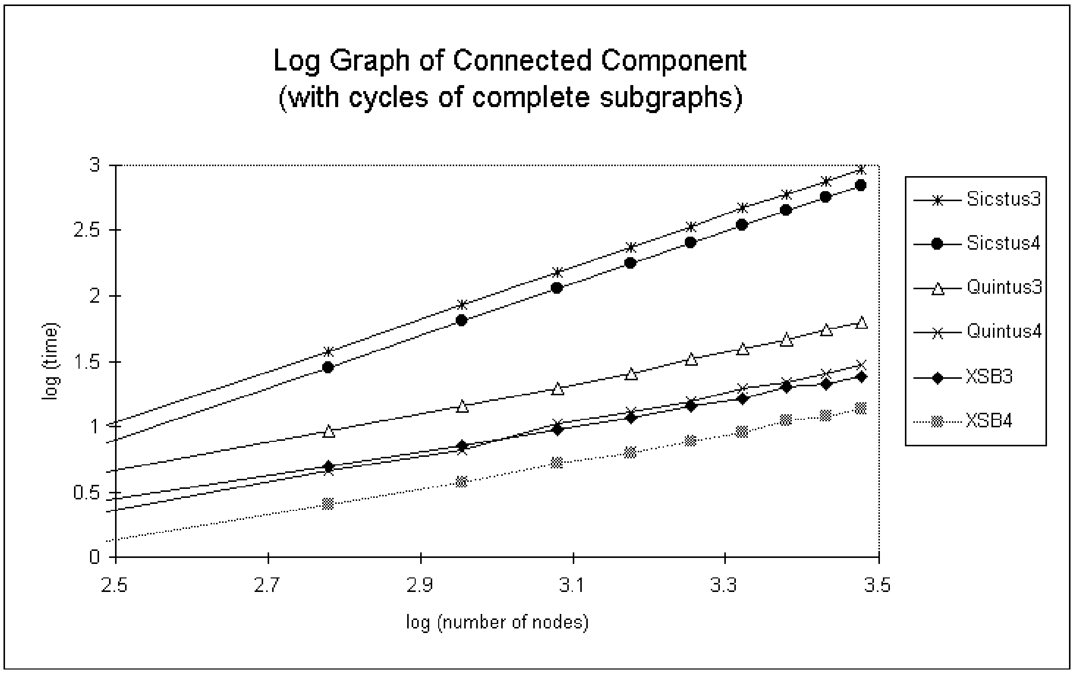


Figure 5.17: Log-log graph of finding connected components with cycle of cliques with size 4

slope(XSB3) = 0.968	slope(Quintus3) = 1.156	slope(Sicstus3) = 1.970
slope(XSB4) = 1.036	slope(Quintus4) = 1.138	slope(Sicstus4) = 1.986

5.4 All Pairs Shortest Paths Problem

This example shows the advantage of using keys. Intuitively, the $dist(X,Y,Z)$ predicate means that the minimum path length from node X to node Y is Z. Since each path has a unique minimum length, the predicate can be defined with the first two arguments as key, leaving the last argument as the value. We can use the *set* predicate to change these distance values. The program executes the minimum path algorithm n times. Since the minimum path algorithm searches each edge once, with the number of edge proportional to the number of nodes in our graph, the minimum path algorithm is of order of n. So, the complexity of the program should be on the order of n^2. Detailed explanation of the program can be found on Section 3.5.1.

```
declare tempD/1 key 0.
declare front/1.
declare dist/3 key 2 size 100.

apsp  ←create(tempD(0)) sc
          forall [R] in node(R) do bfs(R).

bfs(R)  ←bulk_del(front(X)) sc
          set(tempD(0)) sc
          create(front(R)) sc
          create(dist(R, R, 0)) sc
          with [R] while newFront(R, _) do
          (
                tempD(X) sc
                D is X + 1 sc
                set(tempD(D)) sc
                front(M)  :=  newFront(R, M) sc
                bulk_ins(front(N),  dist(R, N, D))
          ).
```

Note that since $newFront(\_,\_)$ is in the condition of a while loop, it must be defined in the database, rather than in the transaction base. Therefore, rules defining $newFront/2$

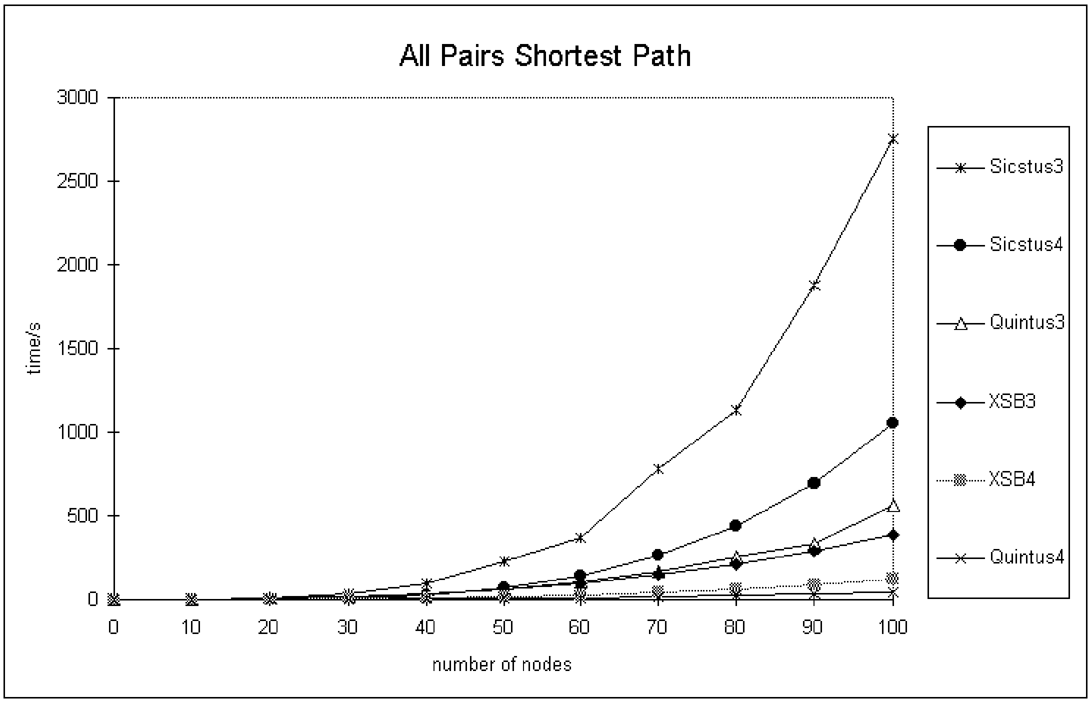


Figure 5.18: Result of All Pairs Shortest Path problem.

are not included in the above transaction program. These rules should be loaded into the database separately. The definition of $newFront/2$ is as follows:

$$newFront(R, N) \ :- \ front(M),\ edge(M),\ not(visited(R, M)).$$
$$visited(R, N) \ :- \ dist(R, N, \_).$$

In the worst case, to calculate the minimum path from one node to another, all edges are visited once. Therefore, the complexity is in the order of $e \times n$, or n^3 in worst case, where e is number of edges, and n is number of nodes. In our graphs, e is proportional to n, so the expected complexity is on the order of n^2.

Like the connected-components problem, the above program is tested on a sequence of graphs of increasing size, each consisting of a single cycle of nodes, Such graphs force the loop to iterate as much as possible. The test results are shown in *Figure 5.18* and the corresponding log-log graph is shown in *Figure 5.19*.

As expected, the complexity of the program is approximately quadratic when running under XSB Prolog. The complexity of the program under Sicstus Prolog has polynomial degree 1 higher than XSB Prolog, again because of the $O(n)$ access time to individual tuples.

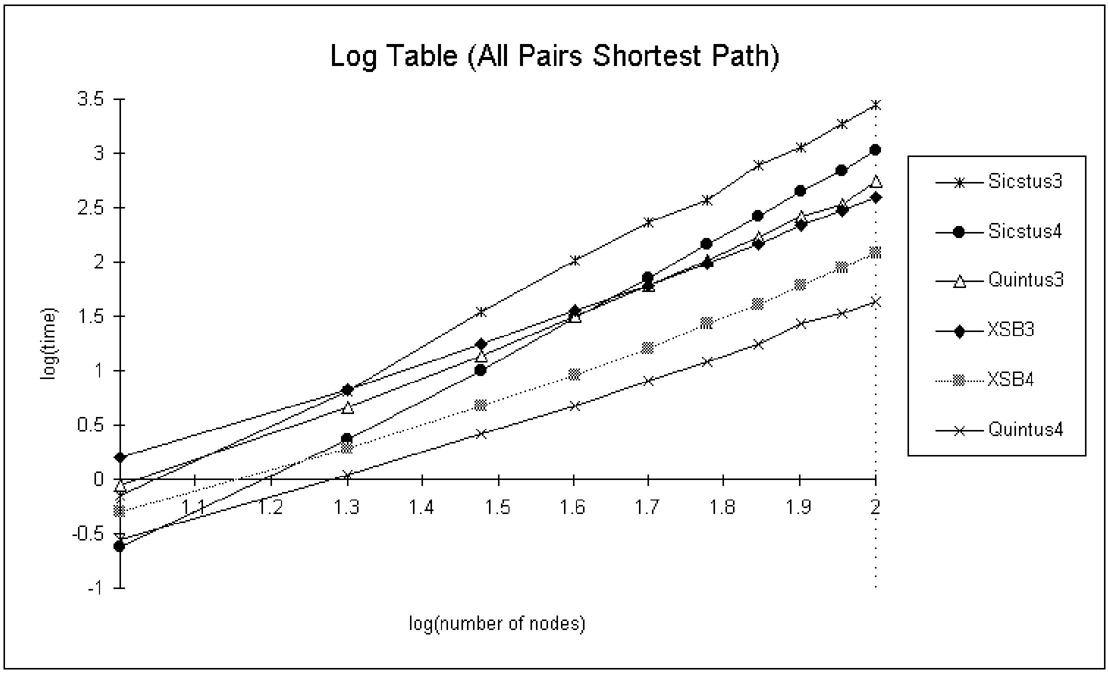


Figure 5.19: Log-log graph for All Pairs Shortest Path problem.

slope(XSB3) = 2.40	slope(Quintus3) = 2.79	slope(Sicstus3) = 3.62
slope(XSB4) = 2.40	slope(Quintus4) = 2.21	slope(Sicstus4) = 3.68

However, Quintus Prolog good complexity in this problem, which is also close to quadratic, especially in Prototype 4. One explanation for Quintus's good performance on this problem is that Quintus may be using larger hash tables for indexing predicates than Sicstus. So, when the number of nodes is not too large (less than 100), the increase in access time is negligible with the increase in number of nodes.

Figures 5.20 and *5.21* show the result of testing the program on cycles of cliques (as in Section 5.3). The program seems to perform better with clique graphs, since the results are close to quadratic in both XSB and Quintus Prolog.

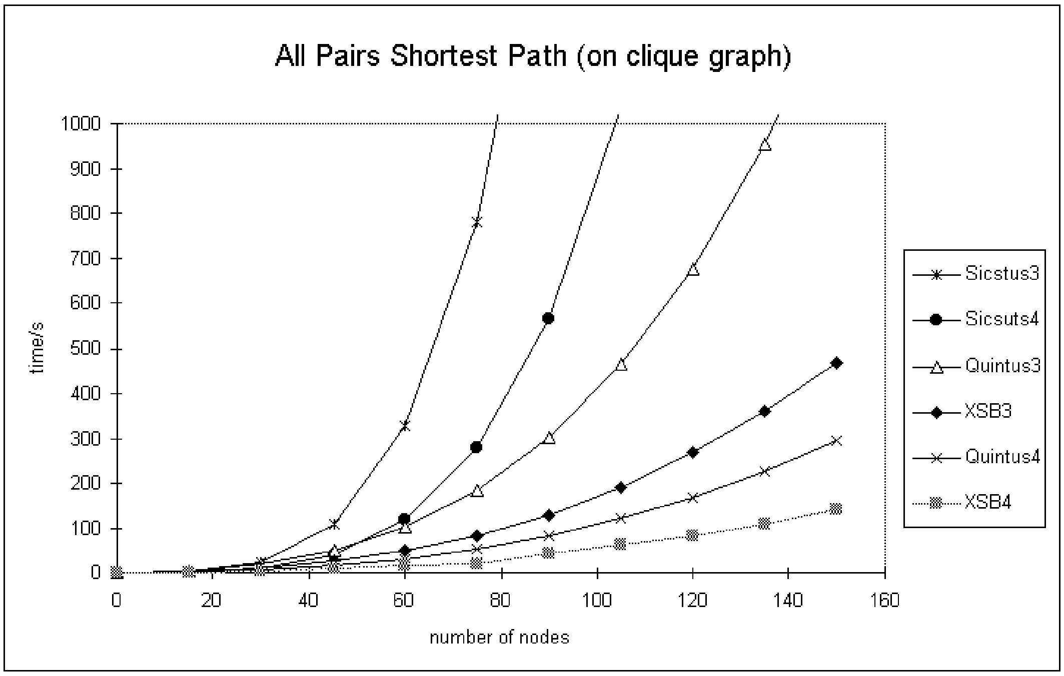


Figure 5.20: Performance of All Pairs Shortest Path programs on clique graphs.

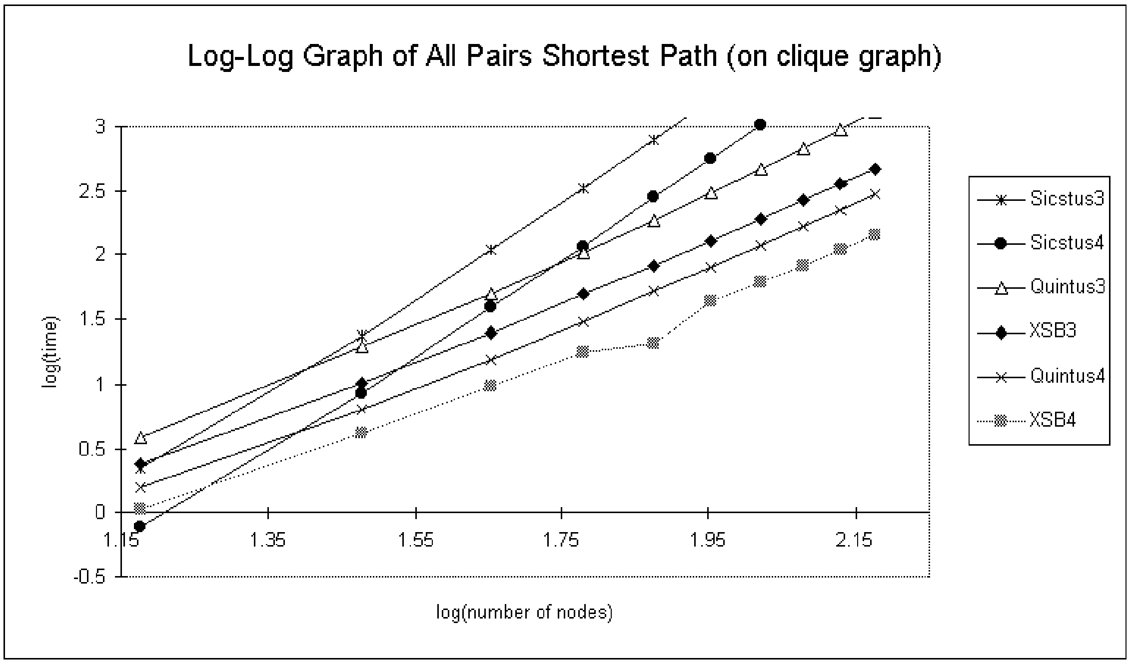


Figure 5.21: Log-log graph of All Pairs Shortest Path programs on Clique Graphs.

slope(XSB3) = 2.30 slope(Quintus3) = 2.53 slope(Sicstus3) = 3.67
slope(XSB4) = 2.11 slope(Quintus4) = 2.29 slope(Sicstus4) = 3.71

Floyd's Algorithm:

The All Pairs Shortest Path problem can also be solved using an algorithm called *"Floyd's Algorithm"*. The following shows a transaction program for a modified version of Floyd's algorithm:

```
declare dist/3 key 2 size 100.
apsp ←   dist(X,Y,Z) := distInit(X,Y,Z) sc
         forall [K] in node(K) do
         (
              with [K] forall [I,J,D] in newDist(I,J,D,K) do
              (
              if dist(I,J,X)
              then set dist(I,J,D)
              else create dist(I,J,D)
              )
         ).
```

And the following database rules define the predicates $distInit/3$ and $newDist/4$:

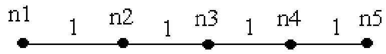
$$
\begin{aligned}
distInit(I,J,1) &\ :- \ edge(I,J).\\
distInit(I,I,0) &\ :- \ node(I).\\
newDist(I,J,D2,K) &\ :- \ dist2(I,J,D2,K), dist(I,J,D1), (D2 < D1).\\
newDist(I,J,D,K) &\ :- \ dist2(I,J,D,K), not(dist(I,J,\_)).\\
dist2(I,J,D,K) &\ :- \ dist(K,I,D1), dist(K,J,D2), (DisD1 + D2).
\end{aligned}
$$

The predicate $distInit(I,J,N)$ returns the initial distance between node I and node J. If I and J are the same, then N is 0. If there is an edge between I and J, then N is 1. The predicate $dist2(I,J,D,K)$ returns the distance D from node I to node J via node K. The predicate $newDist(I,J,D,K)$ returns the distance K from node I to node J, if $dist(I,J,K)$ is not in database or it is shortest that the distance recorded in database.

The transaction *apsp* first initializes all possible distance tuples from the graph using $distInit/3$. Then, repeat for every nodes K, use $newDist(I,J,D,K)$ to search for any new distance from node I to node J using node K as an intermediate node. If there is no distance recorded from node I to node J, then $dist(I,J,D)$ is created. Otherwise, change the distance from I to J into K because it is shorter than the distance currently stored in the database.

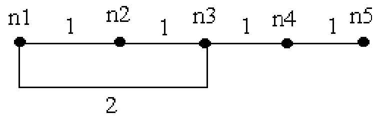
Example:

1. With the a "line" graph of 5 nodes, the initial distance is marked as shown in the graph (path with zero distance is omitted).

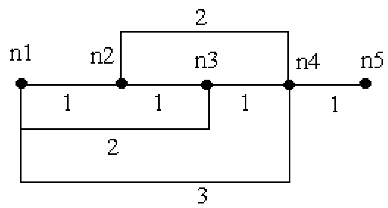


2. Suppose the program iterates by choosing nodes in the order $n1$, $n2$, $n3$, $n4$, $n5$. In the first iteration, no new distance is added.

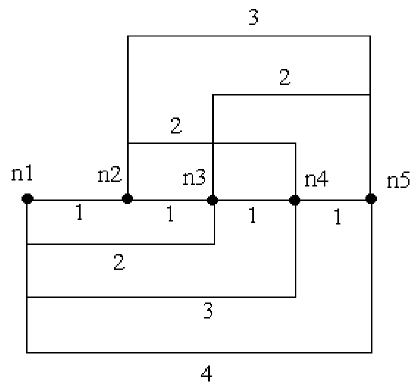
3. In the second iteration, distance between $n1$ and $n3$ is added because $dist(n1, n2, 1)$ and $dist(n2, n3, 1)$ is in the database.



4. In the next iteration, since $dist(n1, n3, 2)$ and $dist(n3, n4, 1)$ is in the database, the new distance between $n1$ and $n4$ is added. The tuple $dist(n2, n4, 2)$ is also added in this iteration.



5. In the next iteration, distances between $n3$ and $n5$, $n2$ and $n5$, $n1$ and $n5$ are added as shown in the figure.



6. In the last iteration, $n5$ is used but no more distance tuple is found.

Note: In the program, if $dist(X, Y, N)$ is added in an iteration, then $dist(Y, X, N)$ is also added.

Figures 5.23 and *5.24* show the performance of the program on "line" graphs of the following form:

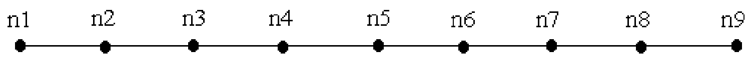


Figure 5.22: Line Graph with 9 nodes.

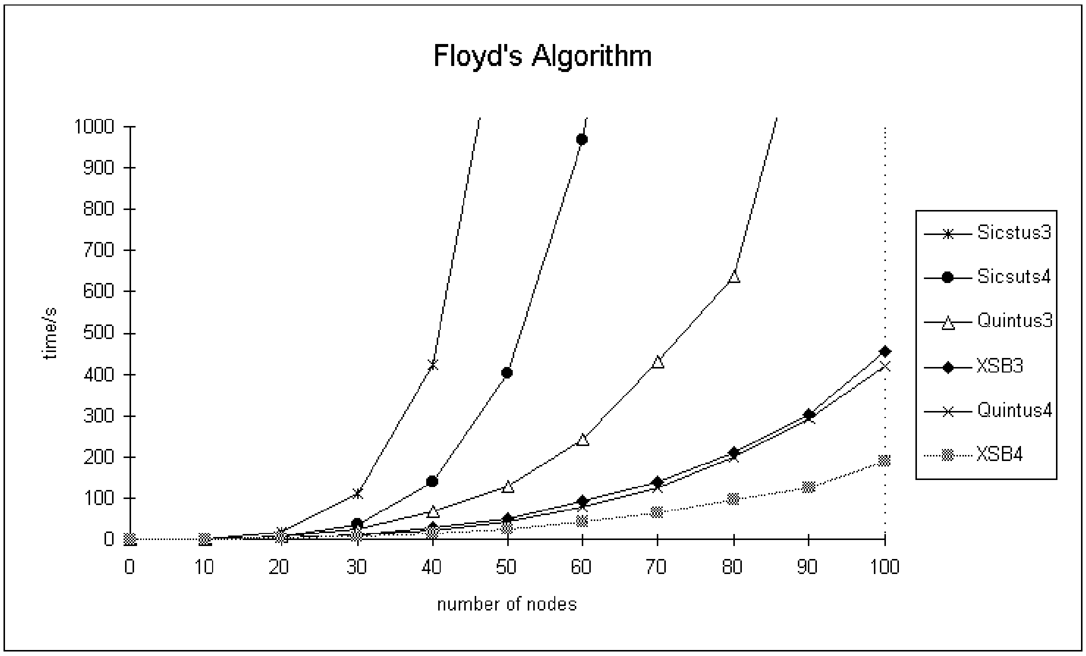


Figure 5.23: Performance of modified Flogd's algorithm on line graphs.

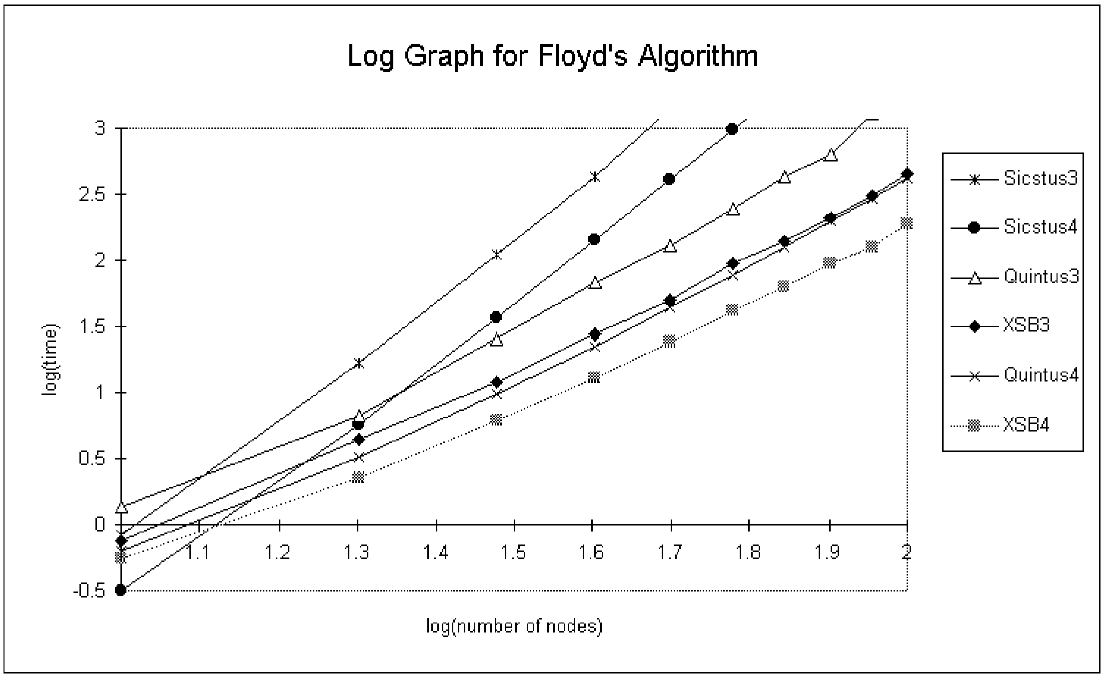


Figure 5.24: Log-log graph of modified Floyd's algorithm on line graphs.

slope(XSB3) = 2.77	slope(Quintus3) = 3.11	slope(Sicstus3) = 4.52
slope(XSB4) = 2.55	slope(Quintus4) = 2.84	slope(Sicstus4) = 4.52

5.5 Binary Counter

This program runs on a relatively small database. Also, unlike previous tests, the database size does not change during execution. Predicates for the counter program are defined as follows:

declare bit/2 key 1 size 13.

declare max/1 key 0.

declare pos/1 key 0.

declare num/1 key 0.

Intuitively, $bit(I, V)$ means that bit I of the counter has value V. Since each bit has just one value, I can be selected as the key. The remaining tuples can be thought of as global variables. The names of the predicates correspond to variable names, and each variables has just one value, so for each of these predicates, the predicate name alone is a key. The predicate $Max/1$ stores the maximum bit for the counter. $Pos/1$ stores the current position of a bit pointer during execution. $Num/1$ is a temporary predicate counting the number incremented in $count/1$.

$$
\begin{aligned}
initialize(M) \;\leftarrow\;& create\ num(0)\ sc \\
& create\ max(M)\ sc \\
& create\ pos(0)\ sc \\
& with\ [M]\ while\ (pos(N),\ N < M)\ do \\
& \qquad (create\ bit(N,0)\ sc\ N2\ is\ N+1\ sc\ set\ pos(N2))\ sc \\
& \qquad set\ pos(0).
\end{aligned} \tag{5.1}
$$

The predicate $initialize(N)$ initializes a N-bit counter by setting all bits to zero and initializing some parameters, such as maximum bit is N, number represents by counter is 0, etc.

$$zero(M) \;\leftarrow\; while\ bit(N,1)\ do\ set\ bit(N,0). \tag{5.2}$$

$$
\begin{aligned}
incr \;\leftarrow\;& set\ pos(0)\ sc \\
& while\ (pos(N),\ bit(N,1)) \\
& \qquad do\ (set\ bit(N,0)\ sc\ N2\ is\ N+1\ sc\ set\ pos(N2))\ sc \\
& pos(N)\ sc \\
& max(M)\ sc \\
& if\ N < M\ then\ set\ bit(N,1).
\end{aligned} \tag{5.3}
$$

The predicate $incr$ increments the counter by 1. Starting from the least significant bit, carry bits are propagating to more-significant bits. Propagation stops when carrying a bit into a bit whose value is 0.

$$
\begin{aligned}
count(M) \;\leftarrow\;& set\ num(0)\ sc \\
& with\ [M]\ while\ (num(N),\ N < M)\ do \\
& \qquad (incr\ sc\ N2\ is\ N+1\ sc\ set\ num(N2)).
\end{aligned} \tag{5.4}
$$

The predicate $count(M)$ increments the value of the counter by M. It uses $num/1$ to record the number of increments performed so far.

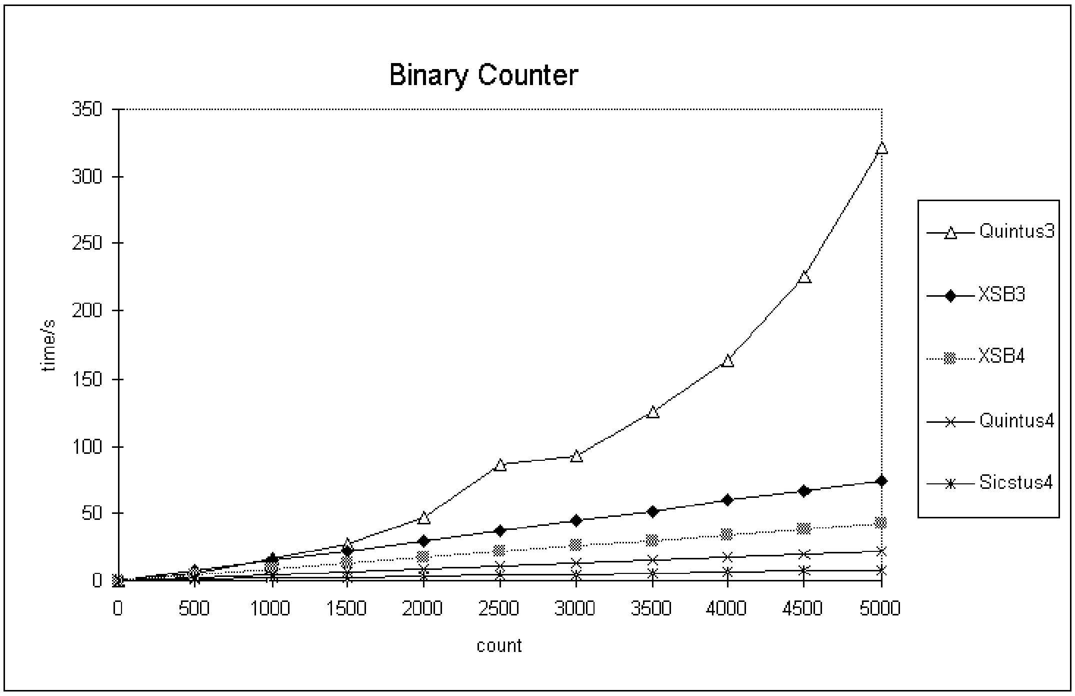


Figure 5.25: Result of counting a certain number in binary counter.

The tests are performed by using the counter to count from 0 to N. It is expected that the search time will increase linearly with N. *Figure 5.25* shows the result in plotting time against the count, N.

Apart from Prototype 3 in Quintus, the complexities match our expectations. Since the database size is small and none of the predicates has its key on more than one argument, the first argument each predicate is good enough for indexing. Thus, tuple access time in Sicstus Prolog is constant as it is in Quintus and XSB Prolog.

Unpredictably, the program runs fastest in Sicstus Prolog, which had the worst performance in the previous tests. Such good performance of Prototype 4 is due to the tail recursion optimization in Sicstus Prolog, so the interpreted tail recursive rule for while loop can be efficiently executed. Result also shows that in small databases, Sicstus Prolog is the best among these Prolog languages, whereas XSB Prolog is the best for large databases, because of its comprehensive indexing features. Unlike previous tests, Quintus Prolog performs very strangely in Prototype 3. For a polynomial complexity problem, the log-log graph should be linear. Since the result of Prototype3 in Quintus Prolog is non-linear, the program is running in superpolynomial complexity.

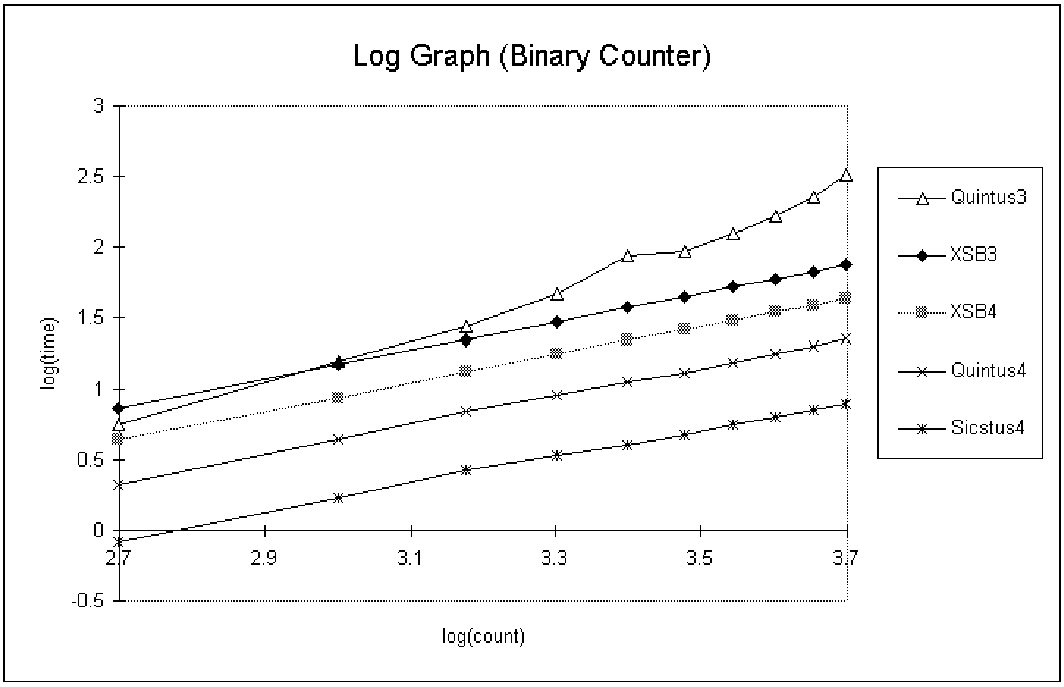


Figure 5.26: Log table of the the result in figure 5.5a.

slope(XSB3) = 1.01   slope(Quintus3) = 1.71
slope(XSB4) = 1.00   slope(Quintus4) = 1.01   slope(Sicstus4) = 1.00

Finally, by starting XSB Prolog with "*xsb -i -m 40000*", the counter program can count up to 50,000 in 9 minutes in Prototype 4, and Prototype 4 in Sicstus Prolog can count up to 1 million in 30 minutes!

5.6 Hamiltonian Cycle

The Hamiltonian-path problem is a well-known NP-complete problem. In the worst case, the complexity of Hamiltonian-cycle algorithms is exponential. For this reason, graphs for this test should be carefully selected, since it might take years to run the program.

In some special graphs, the Hamiltonian cycle can be found by chance without any backtracking, i.e., the program is lucky enough to find the right cycle in the first try. In this way, the performance of the program on different graphs is determined by luck, and it would be hard to analyze.

In the worst case, all paths in the graph must be checked. If we force the query to be false, i.e., by putting a *fail* after the query of the Hamiltonian cycle problem, all paths in the graph will be visited. In this way, we can exercise and test the backtracking capabilities of our $\mathcal{TR}$ implementations. The following program finds a Hamiltonian cycle in a graph.

declare unvisited/1 size 10.

declare start/1.

declare component/2 size 10.

findhamo	$\leftarrow$	*unvisited(P) := node(P) sc*
		destroy(unvisited(X)) sc ! sc
		searchpath(X).
searchpath(S,X)	$\leftarrow$	*edge(X, Y) sc*
		destroy(unvisited(Y)) sc
		create(component(X, Y)) sc
		searchpath(Y).
searchpath(S,X)	$\leftarrow$	*not(unvisited(Y)) sc*
		edge(X,S) sc
		create(component(X,S)).

Explanation of the above $\mathcal{TR}$ implementation was shown in Section 3.5.2.

The query "*findhamo sc fail*" explores all paths in the graph. Our tests are based on this query. Without selecting the graph carefully, the number of possible paths may grow exponentially in the number of nodes. For example, a complete graph has $n!$ Hamiltonian cycles, where n is the number of nodes. Therefore, a complete graph with 10 nodes will already have over 3 million possible cycles, which will take a very long time to search. In our tests, each graph is a cycle of C3 graphs, as in *Figure 5.27*.

Such graphs have just one Hamiltonian cycle. As described above, the Hamiltonian graph problem is a NPC problem, so the execution time should grow exponentially with the number of nodes. The expected complexity of the program is $2e$ because an addition of 1 edge doubles the possible number of search paths, the original search paths, and new paths resulted from including the new edge to the original paths. Again, since the number of edges are chosen to be proportional to the number of nodes, the complexity of the program should be $2^{O(n)}$.

Figure 5.28 shows the graph of time versus the number of nodes, and *Figure 5.29* shows a graph of $log(time)$ against number of nodes in order to deduce the degree of the exponential.

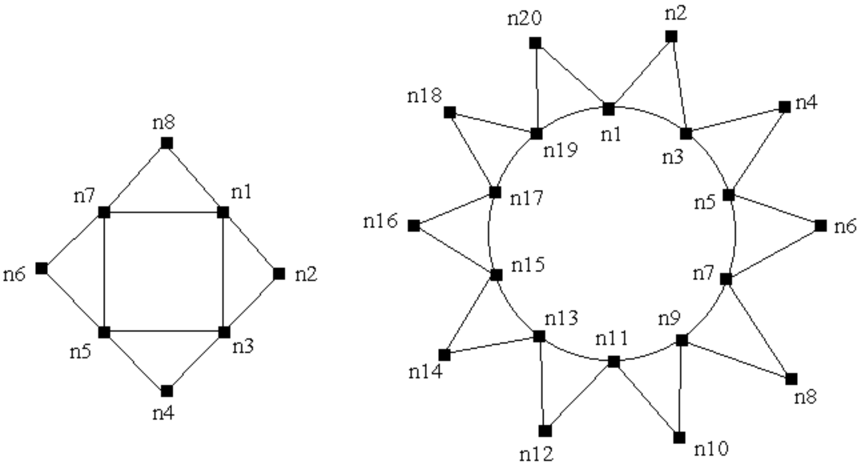


Figure 5.27: Cycles with 4 and 10 C3 Graphs.

As shown in the beginning of this chapter, the slope of log-n graph is the product of the degree of the exponential and the log of the base of the exponential. Since the base of exponential is 2, the degree of exponential is $slope/log(2)$. The slopes in *Figure 5.29* are approximately 1.42 in all three versions of Prototype 3:

$$\begin{aligned} degree\ of\ exponential &= slope/log(2) \\ &= 1.42/1og(2) \\ &\approx 1 \end{aligned}$$

Therefore, the program is running with the expected complexity, i.e., $2^{0(n)}$. Again, from these graphs, Sicstus Prolog is the fastest Prolog to run this program. This again shows the strength of Sicstus in small databases (note that there are only 24 nodes and 36 edges in the largest graph). However, it is disappointing that we cannot test the program in large graphs, because a 20 nodes graph takes over 5 hours to test due to the exponential complexity of the program.

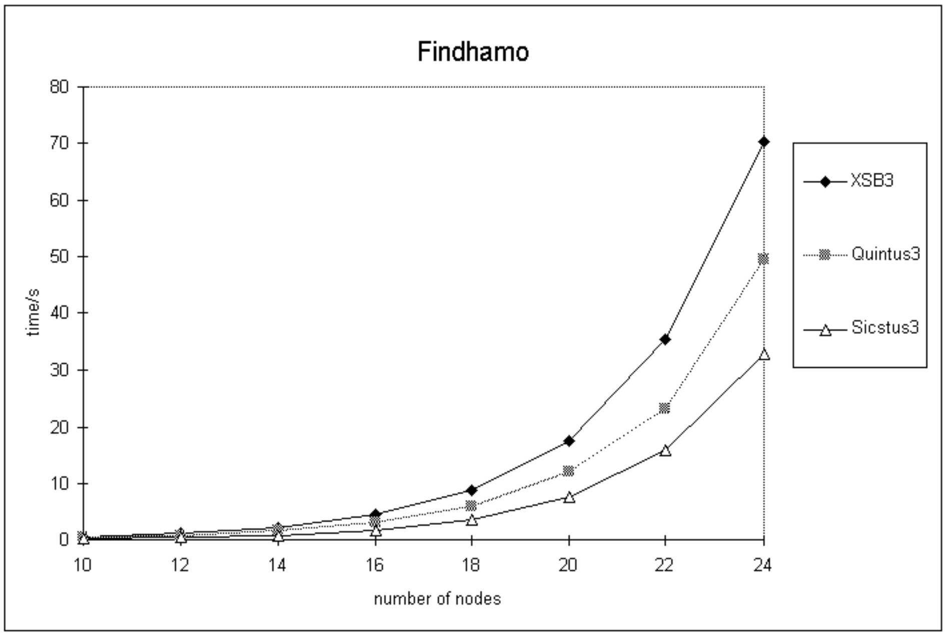


Figure 5.28: Graph of number of nodes versus time for executing Hamiltonian-cycle searching program.

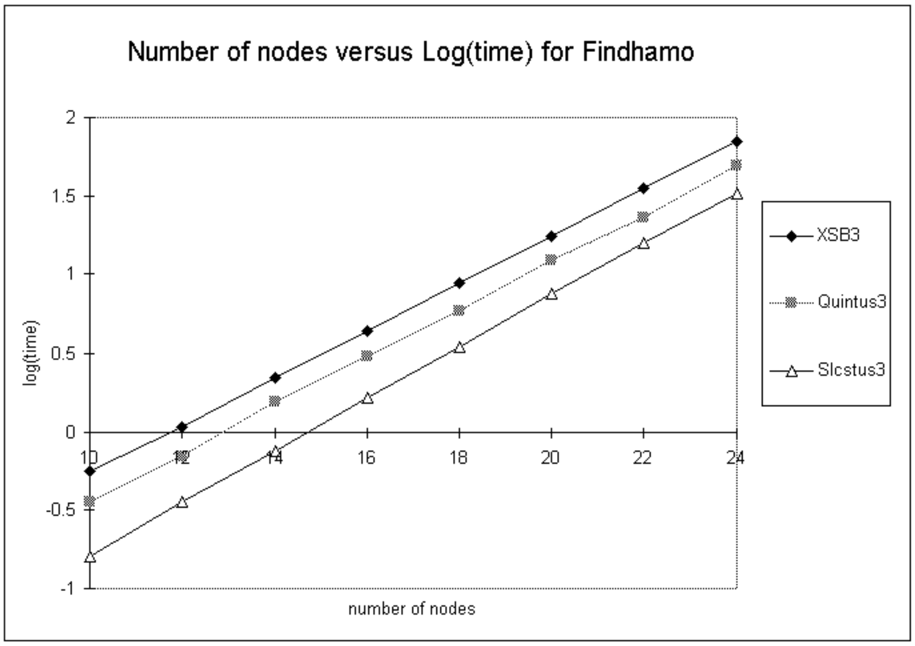


Figure 5.29: Log-n graph of number of nodes versus log(time) for finding Hamiltonian cycles.

slope(XSB3) = 1.42 slope(Quintus3) = 1.42 slope(Sicstus3) = 1.46

5.7 Clique

As discussed above, the Hamiltonian-cycle problem is suitable for small graphs only. To exercise the backtracking capabilities of $\mathcal{T}R$ on larger databases, this section uses the k-clique problem. The maximum clique size problem is a well-known NP-complete problem. However, it becomes a polynomial-time problem if the question is rephrased to that of finding a clique of size k. The following $\mathcal{T}R$ program (described in Section 3.5.2) succeeds if and only if the graph in the database has a clique of size k.

```
declare candidate/1.
@ Return true iff the graph has a clique of size K
cliqueSize(K)        ←    node(N) sc
                          candidate(CN) := edge(N,CN) sc
                          buildClique(K,N).

@ Attempts to build a clique of size K containing node N1 from candidate nodes.
buildClique(1,_).
buildClique(K,N1)    ←    K > 1 sc
                          candidate(CN) := newCandidate(N1,CN) sc
                          candidate(N2) sc
                          K1 is K-1 sc
                          buildClique(K1,N2).
```

The graphs in our tests are chains of complete graphs having 5 nodes each, as in *Figure 5.31*. Such graphs have maximum clique size of 5. If we ask the above program to find a clique of size 5, the execution should take constant time for all the graphs, because all nodes belong to a clique of size 5, and the first one will be chosen. However, when asking the program for a clique of size 6, then since there is no clique of size 6, all cliques will be exhaustively generated. The graphs are constructed so that the number of cliques is linear in N, the number of nodes. Thus, execution time should be linear.

Explanation of the complexity:

Consider the graph with cycle of K cliques of size 10 (similar to *Figure 5.30*). It is obvious from the program that before a clique with clique-size 10, cliques with clique-size 9, 8, ...,

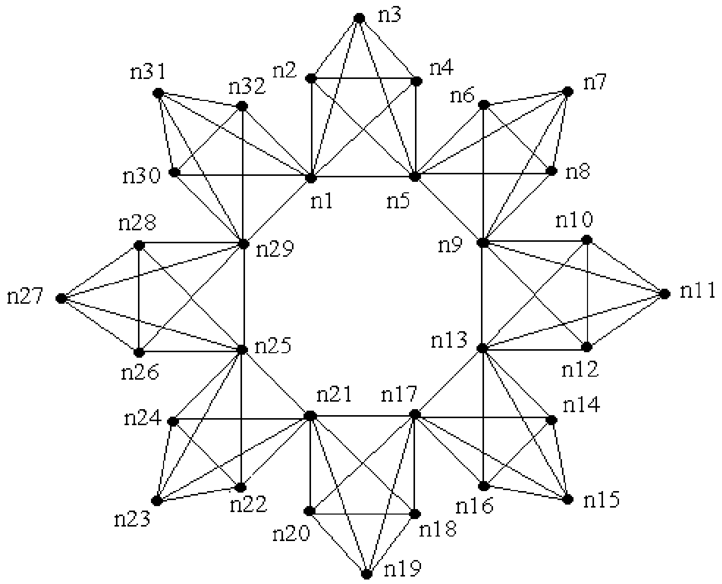


Figure 5.30: A graph of 8 cliques with size 5.

1 are found. Therefore, in the worst case, the complexity of finding a clique with clique-size 10 starting from a particular node N is equivalent to finding all cliques with node N which have clique-size less than 10. If a node is not at the intersection between two cliques, since it is in a complete subgraph with clique-size 9, there are in total of 9! cliques with clique-size less than 10. For nodes which are at the intersections, the possible number of cliques is 2*9!. So, if a graph has K cliques of clique-size 9, it has 9K nodes which is not in the intersection, and 9 nodes which is in an intersection. So, the worst case complexity of the problem of finding clique with clique size 10 is $9K * 9! + 2K * 9! = 10K * 9!$. (linear).

The program above is exponential in K, the requested clique size. However, by keeping K fixed, the execution time should increase linearly with the number of nodes in our graphs. The following tests use graph with a cycle of cliques with fixed maximum clique size. Again, since we have to test for the worst case of the program, we force the program to check all possible cliques by setting the requested clique size K to be one higher than the maximum clique size of the graph. Since the reason of running this program is to test the performance of backtracking in large database, we will use bigger graphs with small maximum clique size. *Figure 5.30* shows a graph of a cycle of cliques with clique-size 5. With this graph, we can use a cycle with 500 cliques (or 2000 nodes) for the test.

To force an exhaustive search of all cliques, the tests were run with the query $cliqueSize(6)$, i.e., to check if there is any clique with size 6, which is not there.

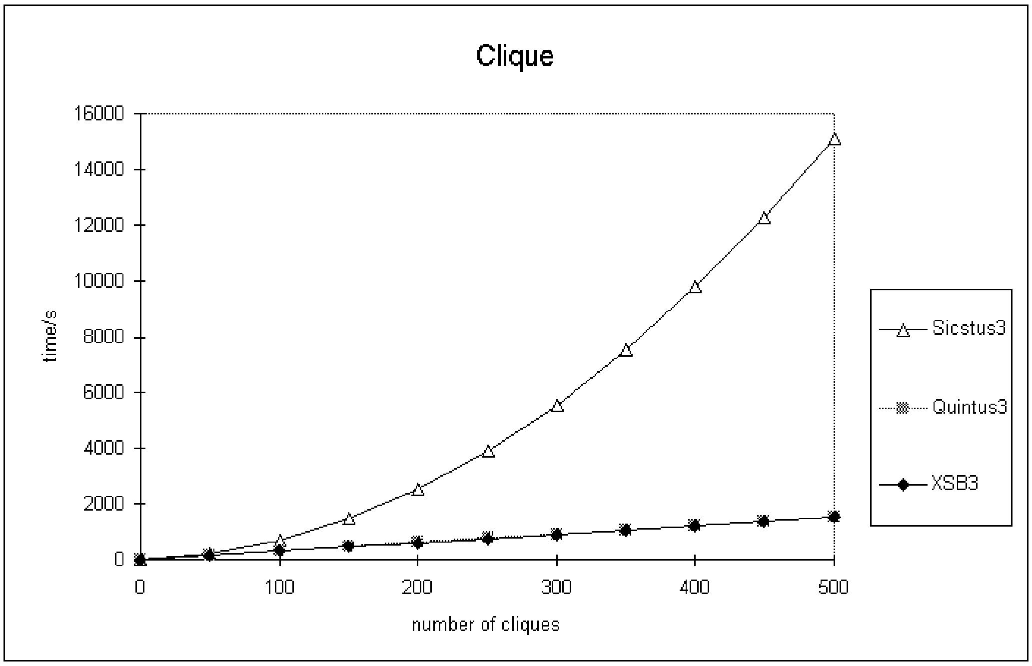


Figure 5.31: Performance on graphs with maximum clique size of 5.

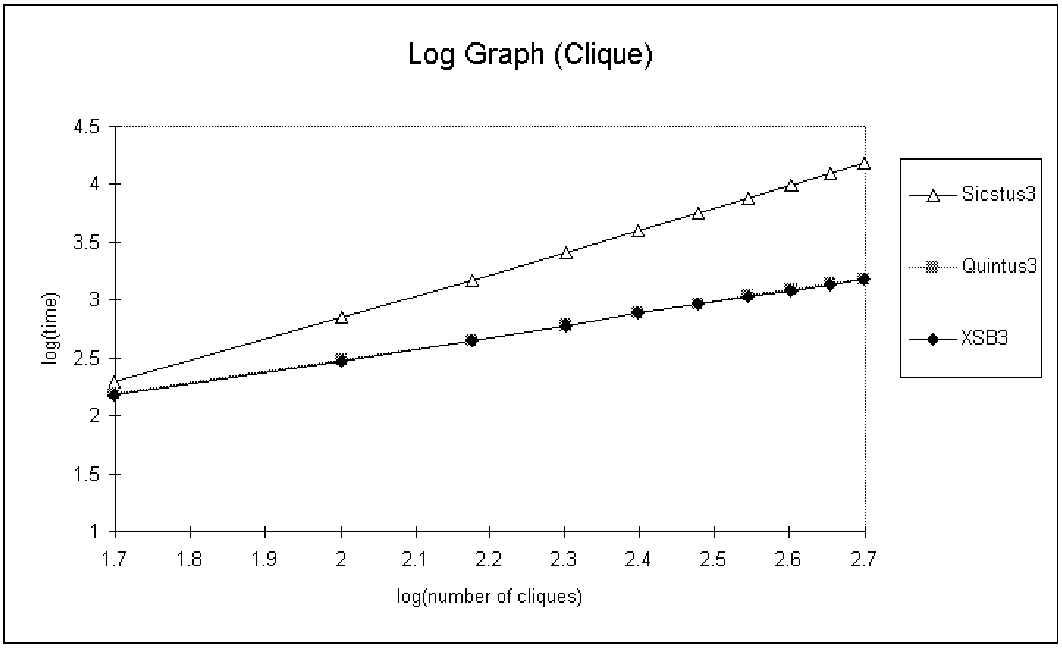


Figure 5.32: Log-log graph of log(number of cliques) versus log(time) for clique problem.

slope(XSB3) = 0.998 slope(Quintus3) = 0.991 slope(Sicstus3) = 1.887

Apart from Sicstus Prolog, the observed complexity is linear, which is what we expected. When the graph is small, the performance of Sicstus Prolog is very close to that of the others. However, as the size of the graphs increases, Sicstus Prolog becomes much slower than the others, and its complexity is quadratic. Again, this is the weakness of this language for large databases. In this example, the complexity of the program in Quintus Prolog is also linear because apart from the $edge/2$ predicate, none of the tuples is defined with more than one argument as a key. Although indexing is needed on both arguments, and the indexing of Quintus Prolog is provided on the first argument only, since each node has at most ten edges on it, access time is still linear for the $edge/2$ predicate.

Conclusions

The above tests show that among three Prologs, XSB Prolog performs in the most understandable way, and can achieve the expected complexities in most cases. On the other hand, although Sicstus Prolog showed its superiority on small databases, it did not achieve the expectedly complexity in tests with large databases. Quintus Prolog performs very unpredictably in many tests. Sometimes, it performs well, but sometimes it performs the worst without any obvious reason. For example, Quintus performs much better in the clique graph problem than in the Hamiltonian-cycle and connected-component problems. Also, although Prototype 3 on Quintus Prolog shows non-linearity in the counter test, but Prototype 4 on Quintus Prolog performs even better than XSB Prolog on the same test.

Chapter 6

Conclusions and Future Work

In the previous chapters, we described our implementation of Transaction Logic Programming in the XSB Prolog environment. The program captures the features in logic programming and database system by providing elementary updates as well as bulk updates. We also described four prototypes and showed how they can be run in other Prolog environments, like Sicstus Prolog and Quintus Prolog, with some minor adjustments. It captures the features in logic programming and database system by providing elementary updates as well as bulk updates.

Our implementation shows the basic prototypes of Transaction Logic programming. There are still many issues left open, especially optimization issues. This chapter suggests some possible directions for future work.

1. *Concurrent conjunction.* In the paper [$BK94$], Bonner and Kifer extends Transaction Logic with connectives for modeling the concurrent execution of complex actions. It allows interleaving of transaction formulas, and can be used in situations requiring synchronization and communications between atoms.

2. *More directives.* There are many built-in directives can be added to improve the ease of use of the system. For example, removing of a transaction rule, undeclaring a database tuple and destroying all related tuples, etc.

3. *Faster implementation on bulk updates.* Although tests showed that execution time for bulk updates is linearly dependent on the number of tuples in the database, the performance, particularly in $bulk\_del/1$, could be improved. In Prototype 4, $bulk\_del/1$

achieves reasonably good performance by sacrificing the backtracking feature.

4. *Write from scratch.* The current implementation is written in a Prolog environment. This is too restrictive because there are many Prolog interpreters, and each of them may provide different built-in directives. If the program is loaded onto a system which does not have the required Prolog interpreter, then users have to modify the program themselves before running it.

   It would be better if the program could be run directly from the UNIX environment by implementing Transaction Logic in C. Without the restrictions of a Prolog interpreter, the storage areas in Transaction Logic (transition oracle, transaction base and database) could be separated physically, rather than all residing in the Prolog database and conceptually distinguishing them by the format of the rules.

5. *Storage optimization (i.e. garbage collection).* Unlike Quintus or Sicstus Prolog, XSB Prolog has two different directives for removing predicates from the database, $retract/1$ and $retract\_nr/1$. In our implementation of $\mathcal{TR}$, $retract/1$ is used to remove a predicate from database. Section 5.1 showed that in XSB Prolog, the $\mathcal{TR}$ implementation takes longer to destroy a tuple than to create one. The reason is that anytime when a tuple is destroyed, the program releases the space used by the tuple. XSB Prolog provides a faster directive to retract a predicate, $retract\_nr/1$, which performs what $retract/1$ does except that it does not reclaim space used by the retracted clause. Using this directive, $\mathcal{TR}$ can destroy a faster tuple with the drawback that the system does not reclaim any program space for predicates which are no longer in the database, which can result in insufficient space for program execution later on. So, the system should reclaim the space automatically at suitable interval, such as after each transaction query, or after 100 updates.

   A better way to destroy a tuple is in a manner similar to $retract/1$ in Quintus Prolog (except that Quintus does not undo updates during backtracking). In Quintus Prolog, the predicate $retract/1$ may be used in a non-determinate fashion. It uses a choice point to keep the data structure of the retracted clause on an internal stack, for backtracking. The choice point is deleted when backtracking is no longer possible on that clause, e.g., by a cut (!), or when execution terminates. To avoid inconsistencies, the space of a retracted clause is reclaimed only when the system determines there is no choice point on the stack that could allow backtracking to the clause.

6. *Allowing compile mode or database mode.* In many Prolog interpreters, rules in a file can be inserted to the database either by compiling the file or by asserting the rules dynamically. In $\mathcal{T}R$, since all updates are performed in database only, rules in a transaction base could be compiled. With the compiled rules, execution of transaction rules can be more efficient and we can also use this method to distinguish transaction base and database physically. At present, this method does not work well in most Prolog interpreters because once a predicate is defined to be dynamic, compiled code defining the same predicate cannot be inserted into database. This is a problem because it is possible in Transaction Logic that a database tuple (stored dynamically) and a transaction formula (stored as compiled format) may define the same predicate.

Bibliography

[1] Alan Gibbons, Algorithmic Graph Theory. Cambridge University Press, 1991.

[2] Anthony J. Bonner and Michael Kifer, Transaction Logic Programming. *Technical Report CSRI-323*, Computer Systems Research Institute, University of Toronto, April 1995. [BK95]

[3] Anthony J. Bonner and Michael Kifer, Applications of Transaction Logic to Knowledge Representation *In Proceedings of the International Conference on Temporal Logic (ICTL), page 67-81*, July 1994.

[4] Anthony J. Bonner and Michael Kifer, Transaction Logic: An (Early) Expose In *Proceedings of the Workshop on Formal Methods in Databases and Software Engineering*, Pringer-Verlag, Workshops in Computing Series, May 1993.

[5] Barry Wilkinson, Digital System Design Prentice Hall 1987.

[6] David S. Warren, Konstantinos F. Sagonas and Terrance Swift, the XSB Programmer's Manual Version 1.4.1, SUNY at Stony Brook, 1994.

[7] Ivan Bratko, Prolog Programming for Artifical Intelligence Addison-Wesley, 1991.

[8] J. W. Llody, Foundations of Logic Programming. Springer-Verlag, 1987.

[9] Henry F. Korth and Abraham Silberschatz, Database System Concepts. McGraw-Hill, 1991.

[10] Sara Baase, Computer Algorithms, Introduction to Design and Analysis. Addison-Wesley, 1989.

[11] J.D. Ullman, Principles of Database and Knowledge Base Systems. Computer Science Press, 1989.

Appendix A

Tutorial

A.1 Getting started with Transaction Logic Programming

Our implementation of Transaction Logic Programming runs under Prolog environment. To run the implementation, at query prompt, loads (*consult*/1) the main module into Prolog and calls '*init*'. The predicate init loads all necessary files into Prolog and initializes the necessary parameters. The following shows the output when loading up Prototype 3 of implemenation in XSB Prolog.

```
XSB Version 1.4.1(94/11/21)
[sequential, single word, optimal mode]
?- [prot3].
[prot3 loaded]

yes
?- init.
[naming3 loaded]
[basic_funs3 loaded]
[tr_basics3 loaded]
[parser3 loaded]
[query3 loaded]
[upload3 loaded]
[keys_appl3 loaded]
[keys_basics3 loaded]

yes
```

The main modules of each module have the name $protN$ when N is the prototype number. For example, in the above example, $prot3$ is prototype 3.

A.2 Standard Predicates

Since this program is under the environment of Prolog interpreter, it is free to use those Prolog commands which have no side effect to the database.

A.2.1 Reading-in Programs

Transaction Logic is different from normal logic programming because of the special rules storing structure. Transaction rules have to be stored in transaction base whereas database rules and database tuples have to be stored in database. This means there are two types of program files: transaction program with all transaction rules, and database program with all database rules. Each program has to use the corresponding file handling predicate to read and compile.

- loadtrans(F)

  Reads transaction base from a file F. The file F should contains only transaction rules, and declaration statements of any undeclared database tuples used in the rules. The rules will pass through a parser and be compiled into a set Prolog rules. The compiler stores the rules into a new file with suffix *_tran* and loads them into the Prolog database. For instance, $loadtrans('mytr')$ reads the transaction program file *mytr*, stores the compiled rules into *mytr_tran* and loads these rules into the Prolog database.

- loadbase(F)

  Reads database file F from the current directory. Contrary to $loadtrans/1$, the file F should contains only database rules, database tuples, and a set of declaration statements. Note that database rules and transaction rules are treated differently.

A.2.2 Rules and Queries

Transaction Rule:

Like most logic formula, a transaction rule consists of two parts: a rule head and a rule body.

$$Head \leftarrow Body.$$
$$or$$
$$Head.$$

where *Head* is an atomic formula and *Body* is a sequence of transaction actions (or queries) connected by serial conjunction connectives (sc).

For example, suppose the transaction base consists of the following two rules:

$$add(0).$$
$$add(N) \leftarrow N > 0 \; sc \; N1 \; is \; N - 1 \; sc \; create(p(N)) \; sc \; add(N1).$$

Then, the transaction query *?- add(N)* would create 10 tuples (from *p(1)* to *p(10)*) in the database.

Database Rule:

Database rule has the same format as a Prolog rule.

$$Head \; :- \; Body.$$
$$or$$
$$Head.$$

Query:

To distinguish a transaction from a Prolog query, the user must put the transaction query as the argument of the Prolog predicate *trans/1*. For simplicity, "*?-*" is a Prolog query prompt, and "*?- trans*" is a transaction prompt. For example, suppose the transaction base and database contain the following:

transaction base	database
p(1)	p(2)

Then, $?-p(X)$ outputs $X = 2$ only, but $?-transp(X)$ outputs $X = 1$ and $X = 2$. So, rules in transaction base are invisible to Prolog queries. A transaction query is a sequence of goals (or updates), connected by serial conjunction, denoted sc. For example, $?-trans\ p(X)\ sc\ create(p(X))$ means if $p(X)$ is in the database (or transaction base), create the same tuple $p(X)$ to the database. In the above example, when X is bound to 1, since $p(1)$ is in the transaction base but not in the database, $create(p(1))$ succeeds.

A.2.3 Directives on Database Tuples

Declaration:

Database atoms represent tuples in a relation. Each relation has its own key. Each key determines a unique value. A declaration statement for a relation should be made before attempting to access it. The statement declares the arity of the relation, its key, and indexes. The format of a declare statement is:

declare $Name/NArgs$ key $Nkey$ size S indexing $IList$.

The statement declares a database relation with Name and number of arguments, $NArgs$. The first $Nkey$ arguments of the relation are the key, which uniquely identifies the values of the remaining arguments. The remaining part of the declaration statement is index specifications. With such declaration, the system would create a hash table with S entries for the arguments listed in $IList$. Note that in most Prologs other than XSB, this part of a declaration is neglected, because most Prolog interpreters do not allow user-defined indexing on predicates.

In the declaration, apart from the relation name and arity, if any of the remaining declaration is missing, the program uses some default settings:

- If the key field is missing, the program uses the whole tuple as the key.
- If the size and indexing fields are missing, the program does not define any special indexing.

- If the indexing field is missing, the program defines a hash table with S entries on the arguments of the key.

For example, *declare p/1* defines a relation *p(X)* with the relation itself be the key, and *declare q/2 key 1* defines a relation *q(X, Y)* with the first argument X be the key.

Updates:

After declaration, a relation can be accessed through queries and updates. There are three types of simple updates: *create*, *destroy* and *set*; and three types of bulk updates: *bulk insert*, *bulk delete* and *assignment*. The following examples use declarations of *p/1 key 1*, and *q/2 key 1*.

- create($Tuple$)

  Inserts the tuple $Tuple$ into database. The tuple specification should have been declared in advance. The predicate also fails if the key of $Tuple$ is already in the database. In other word, the key of tuple has already assigned a value. Note that all variables in $Tuple$ should have been unified before being created. For example, $trans\ create(p(1))\ sc\ create(p(2))$ creates two tuples, $p(1)$ and $p(2)$ to the database, but $trans\ create(q(1,1))\ sc\ create(q(1,2))$ fails.

- destroy($Tuple$)

  Removes the entry $Tuple$ from database. The process fails if $Tuple$ was not in the database. Arguments of $Tuple$ can be uninstantiated because the execution would unify the variables by choosing tuples in database. For example, $trans\ destroy(p(1))$ deletes $p(1)$ from database; $trans\ destory(q(X,Y))$ deletes one of the $q(X,Y)$ tuple from database.

- set($Tuple$)

  Change the value of the key identified by $Tuple$. The key of $Tuple$ should not be the tuple itself, otherwise, no value of the key can be set. For instance, if $q(1,1)$ is in the database, $trans\ set(q(1,2))$ succeeds, but if $q(2,2)$ is not in the database, $trans\ set(q(2,3))$ fails.

- bulk_ins($Cond$,$ToTuple$)

  adds all tuples to $ToTuple$ which satisfy the condition $Cond$. Also, it never fails except $ToTuple$ is not declared.

- bulk_del($Dtuple$)

  destroys all tuples satisfying $Dtuple$. Again, it never fails unless $Dtuple$ is not declared.

- $ToTuple := Cond$

  deletes all tuples in the relation of $ToTuple$ and overwrites them with those satisfies the condition $Cond$. For instance, $p(X) \;:=\; q(X,Y)$ selects the first elements of $q(X,Y)$, and then put in the predicate $p(X)$. So, if the database contains $q(1,1)$, $q(2,2)$, then $p(1)$ and $p(2)$ are created. Note that $Cond$ is a database formula and the assignment searches over the database only.

A.2.4 Loop and Selection

There are two types of loop iterations: while-loop and for-loop. It should be noticed that the conditions of loops are represented by database formulas, i.e., "$pred_1, pred_2, \ldots, pred_N$".

- [with $VList$] while $Cond$ do $Action$.

  repeatedly executes $Actions$ until $Cond$ fails. The list $VList$ contains variables that are exported into the loop and whose values cannot change from one iteration to the next. Comparing the following formulas:

  1. p(X) sc with [X] while q(Y) do r(X,Y).
  2. p(X) sc while q(Y) do r(X,Y).

  Suppose $p(1)$ is in the database. In the first formula, since X is defined to be a constant in the loop, it is bounded to 1 at every iterations. However, in the second formula, X is a free variable in the loop. It remains unbound at every call to $r(X,Y)$ in each iterations and can have different values in each iterations, e.g., $while\ q(Y)\ do\ (X = Y\ sc\ r(X,Y))$, where value of X in each iteration is unified to the value of Y at that occasions.

- [with $VList$] forall $Vars$ in $Cond$ do $Actions$.

  executes $Actions$ for all sets of distinct variables denoted by $Vars$ chosen from $Cond$. Similar to while-loop, $VList$ contains the variable list that are exported into the loop but the values cannot change from one iteration to another. On the other hand, $Vars$ contains the list of variables which are selected to test against the loop condition. For example, with $p(1)$ in the database, $p(X)$ sc $with$ $[X]$ $forall$ $[Y]$ in $q(X,Y)$ do $r(Y)$ executes $r(Y)$ for every Y in $q(1,Y)$. More details of for-loop can be found in Section 4.3.8.

The simplest selection statement is an if-then-ese statement. It has the following format:

if $Cond$ then $Action1$ else $Action2$

This exeuctes $Action1$ if $Cond$ is true, and executes Action2 otherwise.

A.3 Programming Examples

The following shows some simple transaction queries (or updates) and declaration. More examples can be found in benchmark tests.

1. Initially, the $\mathcal{TR}$ database is empty. A predicate q is defined with 1 arguments and the whole predicate is the key.

   ?- declare q/1.

2. The following steps create three tuples $p(1,1)$, $p(2,1)$ and $p(3,3)$. The first argument of $p(X,Y)$ is chosen as the key.

   ?- declare p/2 key 1.

   ?- trans create(p(1,1)) sc create(p(2,1)) sc create(p(3,3)).

3. With the above database, the following steps create a copy of tuples $q(X)$ from the first argument of $p(X,Y)$:

   ?- trans bulk_ins(p(X,Y),q(X)).

   Now, the database contains $p(1,1)$, $p(2,1)$, $p(3,3)$, $q(1)$, $q(2)$ and $q(3)$.

4. The tuples $p(1,1)$ and $p(2,1)$ can be removed by either of the following commands:

   ?- trans destroy(p(1,1)) sc destroy(p(2,1)).

   or

   ?- trans bulk_del(p(X,1)).

Now, the database contains $p(3, 3)$, $q(1)$, $q(2)$, $q(3)$.

5. The following assignment changes the extent of predicate p to $p(1, 1)$, $p(2, 1)$ and $p(3, 1)$:

   ```
   ?- trans p(X,1) := q(X).
   ```

   In this operation, contents of $p(X, Y)$ are removed first. Then, since X has 3 choices: 1,2 and 3, $p(1, 1)$, $p(2, 1)$ and $p(3, 1)$ are created into database.

6. Note: We can simulate the effect of $bulk\_del/1$ by using a while-loop. For example,

   ```
   ?- trans while p(X,1) do destroy(p(X,1)).
   ```

   has the same effect on the database as the updates in **4**, i.e., $p(X, 1)$ is destroyed at each iteration until there is no possible choice left.

7. We can also use a for-loop to simulate the bulk insert in **3**:

   ```
   ?- trans for [X] in q(X) do create(p(X,1)).
   ```

Appendix B

Benchmark Tests

This appendix provides a series of complex updates that test the correctness of a $\mathcal{TR}$ implementation. For each test, we show the top-level $\mathcal{TR}$ command, and the correct answers. In our implementation, the user interface to $\mathcal{TR}$ is similar to Prolog's.

1. Initially, the database is empty. The following commands declare and insert tuples on $p/2$, $q/2$ and $r/1$.

   ```
   ?- declare p/2 key 2.          @ declare p/2 with the whole tuple as the key.

   yes

   ?- trans create(p(1,a)) sc create(p(2,b)).

   yes
   ?- p(X,Y).

   X = 1
   Y = a;

   X = 2
   Y = b;

   no
   ```

?- declare q/2 key 1.	@ declare q/2 with the first argument as the key.
yes	
?- trans create(q(1,a)) sc create(q(2,b)).	
yes	
?- q(X,Y).	
X = 1	
Y = a;	
X = 2	
Y = b;	
no	
?- declare r/1.	@ declare r/1. No key field is indicated in the transaction, so the whole tuple is used as the key.
yes	
?- trans create r(1) sc create r(2).	
yes	
— ?- r(X).	
X = 1;	
X = 2;	
no	
?- trans create(p(1,c)).	p(1,c) successfully created.
yes	

?- trans create(q(1,c)).

no

@ create q(1,c) failed because the first argument is the key and q(1,a) is already in the database.

?- q(1,X).

X = a;

no

@ q(1,c) is not in the database.

---

2. Transaction rules and database rules can be added using *loadtrans/1* and *loadbase/1*, as follows. File 1 is a transaction program file, and File 2 contains database rules only. The following points should be noticed in this example:

   - Compare the difference in syntax between database rules and transaction rules.
   - All database predicates must be declared.
   - All interactive commands are database queries, unless enclosed inside trans (.), in which case, they are transactions.
   - These files are not serial Horn, since *p* occurs in the body of a database rule and the head of a transaction rule.

File 1:

a(X,Y) ¡- p(X,Y) sc r(X).

q(1,1).

q(1,2).

p(2,2).

r(1).

File 2:

declare q/2 key 1.

declare r/1.

b(X,Y) :- p(X,Y), r(X).

q(1,1).

p(1,a).

p(2,a).

r(2).

?- loadbase(file2).

yes

?- loadtrans(file1). yes	@ Base predicates in the transaction base do not require declare statement because they are derived, not stored, and are not restricted by the key of database predicate with same name and number of arguments.
?- r(X). X = 2; no	@ A database query considers data in database only.
?- trans r(X). X = 2; X = 1; no	@ A transaction query considers data in both database and transaction base.
?- b(X,Y). X = 2 Y = a; no	@ A database rule accesses predicate in the database only.
?- trans b(X,Y). X = 2 Y = a; no	@ Even in a transaction query, database rule still only consider database items only. So, r(1) in the transaction base is not considered by in the database rule.
?- a(X,Y). no	@ transaction rules are invisible to database queries.

?- trans a(X,Y).	@ A transaction rule considers data in both transaction base and database.
X = 2	
Y = 2;	
X = 1	
Y = a;	
X = 2	
Y = a;	
no	

---

3. Test of destroy and set. The database contains $p(1,a)$, $p(2,b)$, $p(1,c)$, $q(1,a)$, $q(2,b)$, $q(3,a)$, $r(1)$, $r(2)$, $r(3)$, where $p/2$ *key* 2, $q/2$ *key* 1, and $r/1$ *key* 1 are declared. The transaction base contains $r(4)$, $s(4)$.

?- trans destroy(r(4)).	@ Destroying q predicate in the transaction base is not allowed.
no	
?- trans destroy(s(4)). Undefined predicate/function: destroy_1_r/1 no	@ The destory operation is defined only on tuples declared in the database. Since no tuple about s is declared in the database, the destroy operation is undefined here.
?- trans destroy(p(1,c)).	@ removes p(1,c) from database
yes	

?- p(X,Y). X = 1 Y = a; X = 2 Y = b; no	@ Now, p(1,a) and p(2,b) are left in database.
?- trans destroy(p(X,Y)). X = 1 Y = a; X = 2 Y = b yes	@ We can select an atom to delete. First, p(X,Y) is unified to p(1,a), and delete it. Then, the deletion rolled back and delete p(2,b).
?- p(X,Y). X = 1 Y = a; no	@ p(2,b) was deleted, but p(1,a) was not.
?- trans destroy(q(X,Y)) sc write('aBc ') sc fail. aBc aBc aBc no	@ The execution deletes an atom from $q(X,Y)$, then printed the word "aBc", and then forced to fail. So, the number of times "aBc" shown indicated the number of times destroy(q(X,Y)) succeeded which is the same as number of backtracking, or the number of atoms of q(X,Y). Three "aBc" were shown, so q(X,Y) has three tuples.

?- q(X,Y). X = 1 Y = a; X = 2 Y = b; X = 3 Y = a; no	@ The extent and order of q(X,Y) was perserved.
?- trans destroy(q(X,Y)) sc destroy(r(Z)) sc write('aBc ') sc fail. aBc aBc aBc aBc aBc aBc aBc aBc aBc no	@ There are 3*3 choices for backtracking p and q, i.e., (X,Y,Z) are (1,a,1), (1,a,2), (1,a,3), (2,b,1), (2,b,2), (2,b,3), (3,a,1), (3,a,3), (3,a,3). So, 9 "aBc" are shown.
?- trans set q(1,b). yes	@ change the value of q(1,a) to q(1,b)
?- q(X,Y). X = 1 Y = b; X = 2 Y = b; X = 3 Y = a; no	

?- trans set q(4,a).

no

@ 'set' is not equivalent to 'create'. If the key is not already in the database, then 'set' fails.

?- trans set p(1,b).
Undefined predicate/function: set_2_p/2

no

@ 'set' is not defined on p/2 because the whole predicate is the key (i.e., there are no non-key attributes whose values can be set).

---

4. Test of bulk insertion and deletion. Database contains $p(1, a)$, $p(2, b)$, $p(3, a)$, $r(4)$. Transaction base contains $p(5, 5)$.

?- trans bulk_ins(p(X,Y),r(X)).

X = _1264924
Y = _1264740

yes

@ bulk insertion of p(X,Y) into r(X), so r(1), r(2) and r(3) are added to the database. Also, it copies data from the database only.

?- r(X).

X = 4;

X = 1;

X = 2;

X = 3;

no

?- trans bulk_ins(p(X,Y),q(X,Y)) sc fail.

no

@ Bulk insertion into q forced to fail and backtrack, so q(X,Y) should still contains no tuples.

?- q(X,Y).

no

?- trans bulk_del(p(X,a)). @ bulk deletion of p(X,a), so p(1,a) and p(3,a) should be deleted.

X = _1264776

yes

?- p(X,Y). @ p(2,b) should be the only p-tuple left.

X = 2

Y = b;

no

?- trans bulk_del(p(X,Y)) sc fail. @ A bulk deletion forced to fail and backtrack, so p(2,b) is still in the database after the execution.

no

?- p(X,Y).

X = 2

Y = b;

no

---

5. Bulk Assignment. Database contains $p(1,a)$, $p(1,b)$, $p(2,b)$, $q(3,c)$ and $r(b)$. Transaction base contains $p(1,1)$.

?- trans q(X,Y) := p(X,Y).

X = _1267740
Y = _1267748

yes

@ Since $q/2$ has the first argument as key, the assignment cannot copy both $q(1, a)$ and $q(1, b)$ to the database, so one of them is chosen non-deterministically.

?- q(X,Y).

X = 1
Y = a;

X = 2
Y = b;

no

@ Also, $p(1, 1)$ in the transaction base has no effect on the assignment.

?- trans q(X,Y) := (p(X,Y), r(Y)).

X = _1268180
Y = _1268188

yes

@ This test shows the use of a database formula $(q(X, Y), r(Y))$ rather than a single query in assignment. Since r has only one tuple in the database, i.e., $r(b)$, Y is bound to b, and so $p(1, b)$ and $p(2, b)$ are satisfied.

?- q(X,Y).

X = 1
Y = b;

X = 2
Y = b;

no

@ q(1,b) and q(2,b) were inserted.

6. Test of while loop and for loop. Transaction base contains $p(4,4)$ and database contains $p(1,a)$, $p(2,b)$, $p(3,a)$, and $s/2$ is declared with whole tuple as the key. The following cases are tested separately. Note that the transaction base predicate, $p(4,4)$ has no effect in loop conditions.

Case 1:

?- trans Y = a sc while p(X,Y) do destroy(p(X,Y)).

@ Although Y is bound to a, inside the while loop, it is still considered to be a free variable. So, p is empty after the execution.

Y = a

X = _1265180

yes

?- p(X,Y).

no

?- trans p(X,Y).

@ p(4,4) is still in transaction base.

X = 4

Y = 4;

yes

Case 2:

?- trans Y = a sc with [Y] while p(X,Y) do destroy(p(X,Y)).

@ The term "with [Y]" in the loop causes Y to be imported into the loop. Since Y is bound to 1, database atoms of the form $p(X,1)$ are destroyed during the execution of the loop.

Y = a

X = _1265312

yes

?- p(X,Y).

X = 2

Y = b;

no

Case 3:

?- trans Y = a sc forall [X,Y] in p(X,Y) do create(s(X,Y)).

@ As in case 1 above, Y is unified to a, but it is still a free variable inside the for-loop. So, the loop iterates on all pairs X,Y. So, when the loop terminates, s will contain the same information as p.

Y = a
X = _1265380

yes

?- s(X,Y).

@ X,Y unified with data in database only, so $s(4,4)$ was not created.

X = 2
Y = b;

X = 1
Y = a;

X = 3
Y = a;

no

Case 4:

?- trans Y = a sc with [Y] forall [X,Y] in p(X,Y) do create(s(X,Y)).

@ As in case 2 above, $Y = a$ is imported into the for loop, so X binds to 1 and 3 during loop iteration, and $s(1,a)$, $s(3,a)$ are created.

Y = a
X = _1265512

yes

?- s(X,Y).

X = 1

```
Y = a;

X = 3
Y = a;

no
```

---

7. Tests on different types of for-loop. The database contains $p(1, a)$, $p(2, b)$, $p(3, a)$.

```
?- trans forall [Y] in p(X,Y) do write('aBc ').     @ Y = a,b,a are chosen
aBc aBc aBc
Y = _1264920
X = _1264964

yes

?- trans forall unique [Y] in p(X,Y) do     @ Y = a,b are chosen
write('aBc ').
aBc aBc
Y = _1264936
X = _1264980

yes

?- trans forall possible [Y] in p(X,Y) do     @ Y = a,b are chosen
write('aBc ').
aBc aBc
Y = _1264936
X = _1264980

yes
```

?- trans forall [X,Y] in p(X,Y) do write('ac ') sc write('l ') sc fail. ac ac ac l no	@ Forced failure of for loop but no attemp to re-execute it in a different way.
?- trans forall unique [X,Y] in p(X,Y) do write('ac ') sc write('l ') sc fail. ac ac ac l no	@ Forced failure of for loop but no attemp to re-execute it in a different way
?- trans forall possible [X,Y] in p(X,Y) do write('ac ') sc write('l ') sc fail. ac ac ac l ac ac l ac ac ac l ac ac l ac ac ac l ac ac l no	@ Loop passed sixth times (six '1' printed). Note that sometimes there are 3 iterations to execute a loop, but sometimes there are only two, because the transaction does not backtrack to the first iteration. For example, the order p(X,Y) in the first succeeded loop is p(1,a), p(2,b), p(3,a). When it was forced to fail, it backtracks and the second and the third iteration only, i.e., exchange the order to p(1,a), p(3,a), p(2,b). So, only 2 'ac' printed in the second loop. In the next backtracking, the iteration backtracks to the first iteration, and the order is p(2,b), p(1,a), p(3,a). Eventually, all possible combinations are chosen.

---

8. Test on backtracking of for-loop and while-loop, database contains $p(1, a)$, $p(2, b)$, $p(3, a)$.

?- trans while p(X,Y) do destroy(p(X,Y)) sc write('aBc ') sc fail. aBc aBc aBc aBc aBc aBc no	@ Loop passed six times because there are 6 different orders to destroy $p(X,Y)$.
?- p(X,Y). X = 1 Y = a; X = 2 Y = b; X = 3 Y = a; no	@ p(X,Y) unchanged.
?- trans forall [X,Y] in p(X,Y) do destroy(p(X,Y)) sc fail. no	@ The for loop forced to fail. So, the destroy operations should be undone.
?- p(X,Y). X = 1 Y = a; X = 2 Y = b; X = 3 Y = a; no	@ p(X,Y) unchanged

9. Test on if-then-else. Database contains: $p(1,a)$, $p(2,b)$, $q(1)$, $q(2)$.

?- trans q(X) sc if p(X,Y) then write(Y) else write(X).
a
X = 1
Y = a;
b
X = 2
Y = b;

no

@ X unifies to 1 first, then the condition $p(1,Y)$ succeeds, unifying Y to a in the then-part, which writes a. The transaction backtracks to unify X to 2 and Y to b.

?- trans q(X) sc if p(X,Y) then write(Y).
a
X = 1
Y = a;
b
X = 2
Y = b;

no

@ Since in the above test, the else-part was not used, we can drop the else-part and obtain same result.

?- trans if p(1,1) then write(then) else write(else).
else
yes

@ Since p(1,1) is not in the database, else-part is executed.

?- trans if p(1,1) then write(then).

yes

@ If the condition fails, an if-then formula is still true.

---

10. Test of Nested Loops. Database contains: $p(1,1)$, $p(2,1)$, $p(2,2)$, $p(3,3)$, $q(1)$, $q(2)$.

Case 1:

?- trans forall [X] in q(X) do (with [X] while p(X,Y) do destroy(p(X,Y))).

X = _1265480

Y = _1265012

yes

?- p(X,Y).

X = 3

Y = 3;

no

@ In the forall loop, X is successfully unified to 1 and 2. So, all tuples of the form p(1,Y) and p(2,Y) are eventually deleted, leaving p(3,3) in database.

Case 2:

?- trans while q(X) do (write(X) sc with [X] forall [Y] in p(X,Y) do destroy(q(Y))).

122

X = _1265488

Y = _1265564

yes

?- q(X).

no

Explanation : Normally, this execution should fail because in the first iteration of the outer loop, X is bound to 1, and q(1) is destroyed in the forall loop. In the next iteration of the outer loop, X is bound to 2. The inner loop becomes forall [Y] in p(2,Y) do destroy(q(Y)), and Y is unified to 1 and 2 in the iterations. The iteration fails because destroy(q(1)) fails in the execution of this inner loop. Therefore, the program backtracks and chooses other order to executing the outer while loop. During backtracking of the outer loop, the program chooses to bind X to 2 first. Then, both q(1) and q(2) are destroyed in the forall loop. Since there is no q(X) left in the database, the while loop succeeds.

---

11. Test of Recursion Through Iteration.

    Database contains: $p(1,a)$, $p(1,b)$, $p(1,2)$, $p(2,a)$.

    Transaction base contains the rule:

    $a(X) \leftarrow\ with\ [X]\ forall\ [Y]\ in\ p(X,Y)\ do\ (write(Y)\ sc\ a(Y))$.

?- trans a(1).

ab2a

yes

Explanation : During execution of the loop, Y is unified successively to a, b, and 2. In the first iteration, a is printed and a(a) is called, i.e., forall [Y] in p(a,Y) do destroy(q(Y)). Since there is no p(a,Y) in database, the loop succeeds without any iteration. In the second iteration, b is printed and a(b) is called. Similar, there is no p(b,Y) in the database, and so a(b) succeeds without any updates to database. In the third iteration, 2 is printed and a(2) is called. Since p(2,a) is in database, the loop iterates, binding Y to a, so a is printed.
