Sabrina Vieira da Silva

Coupling Logic Programming with Relational Databases

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2005

Sabrina Vieira da Silva

Coupling Logic Programming with Relational Databases

Tese submetida à Faculdade de Ciências da
Universidade do Porto para obtenção do grau de Mestre
em Infomática

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2005

Acknowledgements

I would like to thank and express my sincere gratitude towards my supervisors, Michel Ferreira and Ricardo Rocha, both professionally and personally, for their advise, motivation, and support. Their suggestions, availability, needed criticism and inspiring words were always of great help in the development and conclusion of this thesis. I admire you both.

I am also grateful to "Projecto Matemática Ensino", to the coordinators and colleagues, for granting the time and support needed to accomplish this task. To Alexandra Bernardo and Ricardo Fernandes for their motivation and support during this master course.

To my mom and dad, for their unconditional love, support and trust. Thank you Daniel for being the best brother a girl can have. To my aunt for her time when I didn't have any. Thank you Evaristo for all the love, patience and understanding it took to help me through this. To all my friends who somehow knew, and were more certain than I was, that I could "pull this off".

To the outstanding soul of a native pacific northwest Indian whom I shall never forget. May your soul always accompany your footsteps on your long lost journeys.

To my Family

Abstract

The declarative nature of a logic programming language and the ultra-efficient data handling capabilities of database systems provides an outstanding reason for coupling the two systems. However, such a merge can be of a tenuous nature, often having to deal with certain obstacles. Coupling approaches between these two systems have seen a great amount of research, mainly on the strategies used to establish connections between them.

With this research we aim at developing three alternative coupling approaches for distinct forms of communication between both the systems. We propose an initial approach where tuples from the relational database are asserted as logic programming facts. The second approach consists in accessing database tuples through backtracking. The last approach transfers unification from the logic programming language to the relational database engine with the use of a Prolog to SQL translator .

The results in this thesis show that it is possible to couple a logic programming system with a relational database which combines the following three aspects: simplicity, performance and efficiency. This combination, together with the optimization obtained from view-level accesses and indexing can prove to be an interesting solution.

Resumo

A natureza declarativa das linguagens de programação lógica juntamente com as capacidades de armazenamento e manuseamento de informação das bases de dados relacionais, fornecem uma boa razão para integrar estes dois sistemas. Contudo, tal integração pode ser de difícil implementação. As abordagens para potenciar estes dois sistemas têm sido alvo de bastante investigação, principalmente no que diz respeito às possíveis estratégias para estabelecer a ligação entre eles.

Com este estudo, pretendemos desenvolver três abordagens diferentes para estabelecer a comunicação entre ambos os sistemas. Propomos uma abordagem inicial, onde os tuplos da base de dados relacional são assumidos como factos em Prolog. A segunda abordagem consiste em aceder ao tuplos recorrendo ao mecanismo de retrocesso dos sistemas lógicos. A última abordagem transfere o processo de unificação das linguagens de programação lógica para o motor da base de dados relacional, com o auxílio de um tradutor de Prolog para SQL.

Os resultados desta tese comprovam que é possível integrar as linguagens de programação lógica com um sistema de base de dados relacional que combine simplicidade, desempenho e eficiência. Esta combinação acrescida de mecanismos de indexação e de acessos ao nível de vista provam ser uma solução interessante.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Logic programming is a programming paradigm based on Horn Clause Logic, a subset of First Order Logic. The axiomatic knowledge of a logic program can be represented *extensionally* in the form of facts, and *intensionally* in the form of rules. Program execution tries to prove theorems (*goals*) and if a proof succeeds the variable bindings are returned as a solution. Relational databases can also be considered as a simpler First Order Logic model [23]. The axiomatic knowledge is now only represented extensionally in the form of database relations and the theorems to be proved correspond to (SQL) *queries*.

There are two main differences between a logic programming system and a relational database model. A first difference is the evaluation mechanism which is employed in logic systems and in relational database systems. Logic systems, such as Prolog, are based on a *tuple-oriented* evaluation that uses unification to bind variables with atomic values that correspond to an attribute of a *single tuple*. On the other hand, the relational model uses a *set-oriented* evaluation mechanism. The result of applying a relational algebra operation, such as projection, selection or join, to a relation is also a relation, which is a *set of tuples*. A second difference, is the expressive power of each language. While every relational operator can be represented as a logic clause, the inverse does not happen. Recursive rules cannot be expressed as a sequence of relational operators. Thus the expressive power of Horn clause systems is greater than that of the relational database model.

The main motivation behind a deductive database system, based on the marriage of

a logic programming language and a relational database management system, is the combination of the efficiency and safety of database systems in dealing with large amounts of data with the higher expressive power of logic systems. A logic programming language is a concise and intuitive way of specifying instructions to a database, and the deductive database aims at representing the extensional knowledge through database relations and the intensional knowledge through logic rules. Furthermore, a deductive database represents knowledge in a way closer to the human representation, using a *clausal* representation over which a model of deduction is applied to obtain, as needed, the extensional representation of such knowledge. For complex problems, this ability to represent knowledge *intentionally* can be crucial.

Coupling a logic system with a relational database system consists in the definition of an architecture connecting the two systems, to allow high-level querying and efficient data manipulation. The implementation of such deductive database systems follows the following four general methods [6]: **(i)** coupling of an existing logic system implementation to an existing relational database management system; **(ii)** extending an existing logic system with some facilities of a relational database management system; **(iii)** extending an existing relational database management system with some features of a logic language; and **(iv)** tightly integrating logic programming techniques with those of relational database management system's. The selected method for development in this thesis was the first alternative, since the deductive system can profit from future and independent developments of the logic system and of the relational database management system.

The level of integration between a logic system and a database system is described in the literature as *tight* or *loose* [72, 39]. As the complexity of the coupling architecture increases, the more tightly coupled the systems become. To fully preserve the unique features of both systems one must understand the motivations and possible achievements resulting from such a merge. To date, however, no full integration has gained a significant degree of acceptance, mainly because the usual link is often of a very tenuous nature, due to very significant differences between the two systems, that we will address in this thesis.

Quite obviously, the main tools in the construction of a coupling architecture between a logic system and a relational database system are the *application programming interfaces* (API's) of each system. Both Yap [13] and MySQL [19], the systems coupled

in this thesis, have a powerful C API, as C is there main development programming language. Another fundamental tool is a translator from the logic language, in this case Prolog, to the database system querying language, SQL. This thesis uses, as a number of other interfacing architectures, the fundamental work of Christoph Draxler on Prolog to SQL translation [18].

The existing literature on *how* to put these tools together in the implementation of a deductive database system is scarce. There are a number of possible alternatives that can be implemented. This thesis discusses, implements and evaluates some possible architectures for coupling logic and database systems.

1.1 Thesis Purpose

Although there is a vast literature regarding the subject of coupling a logic system with a relational database system, together with a number of existing systems based on this coupling technology, there are very few references in this vast literature that approach the implementation issues of such systems. Most of the research effort in deductive databases has been put on high-level problems, such as query languages, their semantics and evaluation algorithms for the specified queries. Low-level architectural problems, which are very relevant because of important differences between a logic system and a relational database system, are not explored in the literature.

Clearly, this lack of literature on these architectural implementation issues denotes an immature state of development of the current systems. Established technology follows a process of prototyping, where functionality is the main concern, a subsequent process of high-level optimization, and a final process where low-level optimization and alternative implementation architectures are explored. Deductive databases are entering this final process.

The main purpose of this thesis is to address different implementation architectures on the coupling of a logic system with a relational database system. Questions such as efficient translation of Prolog to SQL, representation of relational tuples in the logic system environment, exploration of indexing capabilities of database systems, management of side-effects of Prolog operators, such as the cut operator (`!`), over extensional predicates, are addressed in this thesis. The different approaches are

compared and their advantages and disadvantages are discussed.

For this discussion and evaluation we implemented different coupling architectures using the Yap Prolog compiler and the MySQL relational database management system. Few aspects are, however, strictly dependent on these two systems, and can be generalized to other logic systems and database management systems. The results obtained in this thesis clearly show the need for the implementation of some features in order to obtain efficiency, such as dynamic SQL generation, view-level transformations or use of indexing in database systems. These features are independent of the systems that are coupled.

A secondary purpose of this thesis is to contribute for the benchmarked evaluation of deductive database systems. The lack of such benchmark programs also reveals the immature state of development of current systems, with the absence of a tool that measures efficiency. Although the programs we present in Chapter 5 are far from allowing a relevant comparison of different deductive database systems, they represent a first step in this direction.

1.2 Thesis Outline

This thesis is structured in six major chapters that reflect the work developed. A brief description of each chapter is provided to better understand the sequence of ideas and contents.

Chapter 1: Introduction. The current chapter.

Chapter 2: Deductive Databases Systems. Provides an overview of Prolog and relational databases, along with the similarities and obstacles to integration. The motivations and possibilities for coupling the two systems are analysed on an access and coupling level, as well as various deductive database implementations.

Chapter 3: Development Tools This chapter presents in more detail the two main systems used in this thesis: the Yap Prolog system and the MySQL database management system. The main features of each system and their C language interface to build client programs are introduced and described. The Draxler's

Prolog to SQL compiler as a means to optimize the translation of queries between YAP Prolog and MySQL is also discussed in detail.

Chapter 4: Coupling Approaches. The three alternative approaches for coupling logic programming with relational databases are presented. A detailed step-by-step description of each approach is discussed along with their advantages an disadvantages. The three distinct approaches are: **(i)** asserting database tuples as Prolog facts; **(ii)** accessing database tuples by backtracking; and **(iii)** transferring unification to the database.

Chapter 5: Performance Evaluation. This chapter evaluates and compares the performance of each coupling approach. To improve performance, view-level and relational-level accesses, along with optional indexing, are also tested and studied.

Chapter 6: Concluding Remarks. Discusses the research, summarizes the contributions and suggests directions for further work.

Chapter 2

Deductive Databases Systems

This chapter provides an overview of Prolog and relational databases, along with the similarities and obstacles to the integration of both systems. To better understand the motivations and possibilities for coupling the two systems, we then discuss different techniques of integration and analyse several deductive database implementations.

2.1 Prolog and Relational Databases

This section provides some background to Prolog and relational databases while describing the motivation for their integration.

2.1.1 The Prolog Language

Prolog stands for "PROgramation en LOGic" [12]. It is an attempt to implement Colmerauer and Kowalski's idea that computation is controlled inference [36]. The motivation for Prolog is to separate the specification of *what* the program should do from *how* it should be done. This was summarized by Kowalski's motto [70]:

$$algorithm \ = \ logic \ + control$$

Prolog programs use the logic to express the problem and rely on the Prolog system to execute the specification.

Prolog is a logic programming language which allows users to specify both application knowledge (which is the program) and queries declaratively. The problem of how to solve a query using the knowledge is left to the interpreter. Prolog retrieves answers one by one using a *backtracking mechanism*. A pure Prolog program consists of facts and rules.

Prolog implements a subset of first order logic [38] known as Horn clause logic. A Prolog program is a set of relational rules of the form:

$$r_0(\overrightarrow{t_0}) : -r_1(\overrightarrow{t_1}), \ldots, r_n(\overrightarrow{t_n}).$$

where the r_i's are relational symbols and the $\overrightarrow{t_i}$'s are tuples of first order terms. These rules are called *definite clauses*, and the $r_i(\overrightarrow{t_i})$'s are called *atoms*. A clause is divided into two parts: the part to the left of the `:-` symbol, called the *head*, and the part to the right, called *body*, which is a conjunction of atoms, called *goals*. The symbol `:-` is read as *if*. The meaning of a rule is: "For all bindings of their variables, the terms $\overrightarrow{t_0}$ are in relation r_0 *if* the terms $\overrightarrow{t_1}$ are in relation r_1 *and* ... the terms $\overrightarrow{t_n}$ are in relation r_n." A rule where the number of goals in the body is 0 is called a *fact* and is written without `:-`, and a rule with one goal is called a *chain rule*. A rule with more than one goal is called a *deep rule*, and a rule with an empty head is called a *query*.

A *predicate* (or *procedure*) is defined by the set of clauses which have the same head relational symbol r_0 and the same arity in the tuple t_0 of head atom $r_0(\overrightarrow{t_0})$. A predicate is usually referred to by r_0/n where n is the arity of the tuple t_0.

Figure 2.1 illustrates an example of a Prolog program, used for list concatenation. The program consists of a single predicate, `append/3`, defined by two clauses, one being a fact and one being a chain rule. The letters `L`, `H`, `T` and `R`, represent variables, which always begin with a capital letter in Prolog.

```
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

Figure 2.1: The `append/3` example

Given a program P and a query `:-` $r_1(\overrightarrow{t_1}), ..., r_n(\overrightarrow{t_n})$., a program execution consists in finding for which assignments of values to variables the conjunction $r_1(\overrightarrow{t_1}), ..., r_n(\overrightarrow{t_n})$ is a logical consequence of the program P.

The computation process of Prolog is based on two mechanisms presented by Robinson [55]: *resolution* and *unification*. Resolution allows for several possible strategies. One important strategy is linear resolution, where a fixed clause keeps being transformed by resolving it against other clauses in a given set. A further restriction of linear resolution is *SLD-resolution* [37], where the fixed clause is a query, and a nondeterministic selection function is used to select the atom to resolve and to which clause to resolve against (SLD stands for Select Linear Definite).

Given a query, `:-` $q_1(\overrightarrow{s_1}), ..., q_n(\overrightarrow{s_n})$., SLD-resolution chooses an atom, $q_i(\overrightarrow{s_i})$, from the body of the query and chooses a definite clause from the set of clauses of the program whose head, $r_0(\overrightarrow{t_0})$, *unifies* with $q_i(\overrightarrow{s_i})$ through a valid assignment of values to the variables of both atoms. The atom $q_i(\overrightarrow{s_i})$ in the query is then replaced by the body of the clause selected, using the assignment of values to variables in every atom of the body, and the process is repeated. This process will end with *success* if the body of the query is empty (a fact is reached), or end with *failure* if there is no rule head that unifies with the selected atom [12].

For example, the Prolog program in Fig. 2.2 represents the knowledge about parent and ancestor relationship in a family:

```
parent(robert,janique).
parent(robert,daniel).
parent(janique,sebastien).

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

Figure 2.2: The `parent/2` and `ancestor/2` example

The first three clauses are facts and the last two are rules. The symbols `ancestor` and `parent` are predicates or relations, `robert`, `janique`, `madeleine` and `sebastien` are atoms and `X` and `Y` are variables. A fact of the form `parent(X,Y)` states that `X` *is a parent* of `Y`. The relation `ancestor` is recursively defined by the last two clauses. The first rule states that `X` *is an ancestor* of `Y` if `X` *is a parent* of `Y` and the second rules states that `X is an ancestor` of `Y` if `X` *is a parent* of some `Z` and `Z` is an ancestor of `Y`. An example of a query or goal to the above program to find out who are the descendants of `robert` is encoded as follows:

```
?- ancestor(robert,Y).
```

The Prolog interpreter will instantiate the variable `Y` to the first answer as follows:

```
Y = janique
```

Upon receiving a ';' from the user, the interpreter backtracks and the next answer is retrieved. This process can be continued until no more answers are found. Queries involving sets of answers are common in a typical database environment. This kind of set queries can be implemented in Prolog by using its predicates `setof/3` or `findall/3` [14]. For example, the following query retrieves all descendants of `robert` and returns the answers as a list, in `L`:

```
?- findall(Y,ancestor(robert,Y),L).
L = [janique,daniel,sebastien].
```

2.1.2 Relational Databases

A relational database [11] consists of a collection of relations or tables and their description, which is called a schema. The rows of the tables are called tuples and columns are called attributes. Each attribute has an associated domain of values. An example of an employee relation is shown in Fig. 2.3. The attributes are `Id`, `Name` and `Designation` and there are four tuples in the table.

Id	Name	Designation
1	John	programmer
2	Susan	software_eng
3	Tom	programmer
4	Brenda	project_leader

Figure 2.3: A database relation or table

Integrity constraints [15] are properties that must be satisfied by the data of a database. An example of an integrity constraint applied on the `employee` table could be that the identification field, `Id`, is the primary key. Another example of an integrity

constraint would be to restrict the values that the designation of an employee can take to the set of {`programmer, software_eng, project_leader`}. The schema for the `employee` table together with these constraints can be created in an Oracle or MySQL environment using its query language SQL as described in Fig. 2.4.

```
CREATE TABLE employee
(
  id INTEGER CONSTRAINT pk_emp PRIMARY KEY,
  name CHAR(16),
  designation CHAR(16) CONSTRAINT ck_desig
  CHECK (desig IN ('programmer','software_eng','project_leader'))
)
```

Figure 2.4: SQL expression for creating relation `employee`

A view [11] can be thought of as a mask overlaying one or more tables such that the columns in the view are found in one or more underlying tables or constructed using the columns of underlying tables. An example of a SQL expression to create a view on the table above, which extracts employees who are programmers is shown in Figure 2.5.

```
CREATE VIEW qualified_cc AS SELECT id, name
FROM employee
WHERE designation = 'programmer';
```

Figure 2.5: Creating a view in SQL

2.1.3 Similarities Between Prolog and Relational Databases

Much has been written over the similarities between logic based languages, such as Prolog, and relational databases, and there has been a sizable amount of effort exerted towards producing a practical combination between the two [26]. Such a combination combines the inference capabilities of Prolog with the ultra-efficient data handling capabilities of database systems. To date, however, no full integration has gained a significant degree of acceptance. The usual link between Prolog and a database, when it exists, is often of a very tenuous nature, usually being a simple interface between two independent systems.

Consider the Prolog program in Fig. 2.6. These clauses, added to the program of Figure 2.2, constitute a knowledge base about ancestor relationships which can be

queried. The structural similarity of the `parent/2` facts to tuples of a relational database should be fairly obvious. A predicate in Prolog (as in the rest of logic) is normally interpreted to be nothing more than a relation over some domain. What is perhaps more interesting, however, is that a Prolog rule, such as `father/2`, may be viewed as a join of the relations `male` and `parent` [61, 76, 26]. The Prolog predicate `father/2`, represented by a Prolog rule, would be, in the relational model, a derived relation, or view, over base relations [26]. Datalog, a language based upon Prolog, has been proposed and has gained acceptance as a relatively graceful language for defining and querying databases [69, 41].

```
male(robert).
male(daniel).
male(sebastian).

female(janique).

father(X,Y) :- male(X), parent(X,Y).
```

Figure 2.6: The `male/1`, `female/1` and `father/2` example

2.1.3.1 Motivations for Integration

There are good reasons for a union between Prolog and a relational database. One reason is that Prolog language is a concise and intuitive way of specifying instructions to a database [76, 26]. Compare, for example, the Prolog predicate and query:

```
p(X,Y):- q(X,Z), r(Z,Y).

?- p(a,Y).
```

to its SQL counterpart:

```
CREATE VIEW P AS SELECT Q.X, R.Y FROM Q, R WHERE Q.Z = R.Z;

SELECT Y FROM P WHERE X = a;
```

Though this example is very simple, many would argue that the Prolog rendition is easier to follow. The difference becomes more evident as the action to be performed grows more complex. Another reason for desiring some sort of integration is that

Prolog clauses are generally held in primary memory. This places a severe restriction on the project size to which Prolog can be reasonably applied [61]. Many who appreciate the general programming abilities of Prolog would prefer to have the large storage space of disk drives available to them. Furthermore, keeping data in primary memory limits access to data by multiple agents [30, 72]. Databases, in contrast, can store huge amounts of information on secondary storage such as disk drives, and almost universally allow concurrent access by multiple users. Unlike most database systems, Prolog makes little use of indexing schemes or other optimization techniques, making information retrieval, and therefore logical inference, relatively slow [61]. Database systems, in contrast, are extremely good at covering large quantities of information. Much effort has been exerted in devising clever ways to speed retrieval [41].

2.1.3.2 Obstacles to Integration

There are also differences between Prolog and relational databases, which serve as obstacles to integration and must be pointed out. While the average relational database manipulation language can be considered almost entirely declarative, Prolog has a strong procedural bent. This is the most important reason and obligatory result of Prolog being developed as a general purpose programming language. Database languages, being special purpose, need not be so encumbered [76, 6]. Furthermore, Prolog has a fixed built-in search mechanism (depth-first) and is littered with elements for performing actions irrelevant to logical inference [6].

Lesser points of conflict are as follows. First, the domains for relations in the database model are explicitly specified in a relational system. If the elements are not enumerated, their respective data types (integer, character, etc.) are at least specified [16]. Prolog, though it can be said to have a typing system of some sort, does not provide a means of specifying how predicate arguments are to be restricted [6]. Furthermore, attributes in a relational system are generally referenced by name rather than by position. In Prolog, no argument has a name. Also, values of attributes in a relational database are atomic, meaning that the tuples in a database table correspond only to atomic propositions containing no unbound variables. With very few exceptions, one cannot store a complex structure in a relation [16]. In contrast, Prolog predicate arguments can be as complex as one likes [41]. Another significant obstacle is the set-at-a-time strategy of database management systems that retrieve all the tuples of

a certain query at once in opposition to the tuple-at-a-time backtracking strategy of Prolog.

2.1.4 Prolog and Relational Algebra

A Prolog procedure consisting of ground unit clauses is considered the same thing as a database relation. A fact in the procedure is equivalent to a tuple in the relation, which consists of a set of attributes together with a value for each. A schema is essentially the relation name together with the attribute names and types. A relation is frequently represented as a table with the attributes appearing as column headings and a row representing a tuple. Table 2.1 shows a student relation with three attributes and two tuples.

Student Number -SNO-	**Surname -SName-**	**Subject enrolled in -SubjCode-**
324522	Wong	cs317
113540	Smith	cs383

Table 2.1: A student relation

A single tuple in this representation is simply a sequence of values, with the *ith* value being the value of the attribute appearing in the *ith* column heading. This tuple is represented as a unit clause by equating the relation name with the principal functor, and placing the value of the *ith* attribute in the *ith* argument position.

```
student(324522,'Wong',cs317)
student(113540,'Smith',cs383)
```

A relational algebra selection operation is represented as a Prolog query. For example, the selection operation $\sigma_{SNO=113540}(Student)$ gives rise to the following SQL query:

```
SELECT * FROM Student WHERE SNO = 113540;
```

which returns a new relation with the following single tuple:

```
<113540,Smith,cs383>
```

The corresponding Prolog query would be:

```
?- student(113540, Surname, Subject).
```

where the selection criterion from $\sigma_{SNO=113540}(Student)$ is expressed by the first argument position of the goal being instantiated to the integer 113540. More complex selection criteria are implemented by using built-ins predicates from Prolog. For example, if we wanted all students whose number is greater than 200000, we would have the SQL query:

```
SELECT * FROM Student WHERE SNO > 200000;
```

and the corresponding Prolog goal:

```
?- student(SNO,SName,SubjCode), SNO > 200000.
```

The Prolog interpreter would instantiate the variables to

```
SNO = 324522
SName = 'Wong
SubjCode = 'cs317'
```

If the user ask for further solutions, using the ';'operator at the top level, Prolog's backtracking would try to provide alternative solutions. This example brings up an important difference between Prolog and the relational algebra: Prolog's inference engine operates tuple-at-a-time, computing bindings of the query variables to terms occurring in a single clause of the procedure; while a database manager, the relational algebra's equivalent of an inference engine, identifies the subset of the relation while operating set-at-a-time to satisfy the selection conditions.

In Table 2.2, we present several examples of the representation of relational algebra operations, their SQL expression and their Prolog query. Besides the previously described `Student` relation, the `Subject` and `Lecturer` relations, characterized by the following predicates, are also used in Table 2.2:

```
subject(SubjCode,Title).
```

```
lecturer(LNO,SName,SubjCode).
```

Relational Algebra	SQL	Prolog
Π_{SUBJ}(Student)	SELECT SubjCode FROM Student	student(_,_,SubjCode).
Student $\triangleright\triangleleft_{SubjCode=SubjCode}$ Subject	SELECT * FROM Student, SubjCode WHERE Student.SubjCode= Subject.SubjCode	student(SNO,SName,SubjCode), subject(SubjCode,Title).
Student $\sigma_{Name="Wong"\wedge SNO>"1000"\wedge SubjCode=SubjCode}$ Subject	SELECT * FROM Student, Subject WHERE Student.SName="Wong" and Student.SNO>1000 and Student.SubjCode= Subject.SubjCode	student(SNO,"Wong",SubjCode), SNO>1000, subject(SubjCode, Title).
Student × Subject	SELECT * FROM Student, Subject WHERE Student.SNO=1000 and Subject.SubjCode="cs317" and Student.SubjCode="cs317"	student(1000,SName,cs317), subject(cs317,Title).
Intersection	SELECT * FROM Student, Lecturer WHERE Student.SNO=Lecturer.LNO and Student.SName= Lecturer.SName and Student.SubjCode= Lecturer.SubjCode	student(SNO,Sname,SCode), lecturer(SNO,Sname,SCode).
Union	SELECT * FROM Student, Lecturer	student(SNO1,Sname1,SCode1), lecturer(SNO2,Sname2,SCode2).
Subtraction	SELECT * FROM Student, Lecturer WHERE not(Student.SNO=Lecturer.LNO) and not(Student.SName= Lecturer.SName) and not(Student.SubjCode= Lecturer.SubjCode)	student(SNO,Sname,SCode), not lecturer(SNO,Sname,SCode).

Table 2.2: Relational algebra operations, SQL expressions and Prolog queries

2.2 Deductive Database Systems

We next briefly describe some of the most well-known deductive database systems along with the different techniques of integration between Prolog and relational databases.

This section in highly based on [54].

2.2.1 Historical Overview

Work in automated theorem proving and, later, logic programming was at the origins of deductive databases. Minker suggests that, in the early development of the field, Green and Raphael [27] were the first to recognize the connection between theorem proving and deduction in databases [43].

Early systems included MRPPS, DEDUCE-2, and DADM. MRPPS was an interpretive system developed at Maryland by Minker's group from 1970 through 1978, that explored several search procedures, indexing , techniques, and semantic query optimization. One of the first papers on processing recursive queries was [44]; it contained the first description of bounded recursive queries, which are recursive queries that can be replaced by non-recursive equivalents.

In 1977, a landmark workshop on logic and deductive databases, in Toulose, was organized by Gallaire, Minker and Nicolas that resulted in a book from several papers of the proceedings [23]. The workshop and the book brought together researchers in the area of logic and databases, and gave an identity to the field. The workshop was also organized in subsequent years, with proceedings that continued to influence the field.

Emden and Kowalski [71] showed, in 1976, that the least fixpoint of a Horn-clause logic program coincided with its least Herbrand model. This provided a firm foundation for the semantics of logic programs, especially deductive databases, since fixpoint computation is the operational semantics associated with deductive databases. One of the earliest efficient techniques for evaluating recursive queries in a database context was proposed by Henschen and Naqvi [29]. Earlier systems had used either resolution-based strategies not well-suited to applications with large data sets, or relatively simple techniques.

In 1984, with the initiation of three major projects, two in the U.S.A. and one in Europe, the area of deductive databases, and in particular, recursive query processing, became very active. Significant research contributions and the construction of prototype systems let to the Nail! project at Stanford, the LDL project at MCC in Austin,

and the deductive database project at ECRC.

2.2.1.1 ECRC

The research work at ECRC was led by J. M. Nicolas. The initial phase of research (1984-1987) led to the study of algorithms and the development of early prototypes [74, 73], integrity checking (Soundcheck by H. Decker) and a prototype system that explored consistency checking (Satchmo by R. Manthey and F. Bry) [7], a combination of deductive and object-oriented ideas (KB2 by M. Wallace), persistent Prolog (Educe by J. Bocca), and the BANG file system by M. Freeston [21]. A second phase (1988-1990) led to more functional prototypes: Megalog (1988-1990 by J. Bocca), DedGin* (1988-1989 by Vieille), EKS-V1 (1989-1990, also by Vieille).

2.2.1.2 LDL

The LDL project at MCC led to a number of important advances. By 1986, it was recognized that combining Prolog with a relational database was an unsatisfactory solution, and a decision was made to develop a deductive database system based on bottom-up evaluation techniques [68]. During this period, there were a number of significant research developments including the development of evaluation algorithms (work on semi-naive evaluation, magic sets and counting [4, 59, 58, 5]).

2.2.1.3 NAIL

The NAIL! (Not Another Implementation of Logic!) project was started at Stanford in 1985. The initial goal was to study the optimization of logic using the database-oriented "all- solutions" model. In collaboration with the MCC group, the first paper on magic sets [4] came out of this project, as did the first work on regular recursions [48]. The work on regular recursions was developed further in [49]. Many of the important contributions to coping with negation and aggregation in logical rules were also made by the project. Stratified negation [24], well-founded negation [25], and modularly stratified negation [57] were also developed in connection with this project.

2.2.1.4 Aditi

The Aditi project was initiated in 1988 at the University of Melbourne. The research contributions of this project include a formulation of semi-naive evaluation that is now widely used [3], adaptation of magic sets for stratified programs [2], optimization of right and left linear programs [34], parallelization, indexing techniques, and optimization of programs with constraints [33]. The work of the Aditi group was also driven by the development of their prototype system, which is notable for its emphasis on disk-resident relations. All relational operations are performed with relations assumed to be disk resident, and join techniques such as sort-merge and hash-join are used.

2.2.1.5 Coral

The CORAL project at U. Wisconsin, which was started in 1988, can also be traced to LDL. The research contributions included work on optimizing special classes of programs (notably, right and left linear programs) [49] (jointly with the Glue-Nail group), development of a multiset semantics for logic programs and optimizations dealing with duplicate checks [40], the first results on space-efficient bottom-up evaluation techniques [50], refinements of seminaive evaluation for programs with large numbers of rules [51], evaluation of programs with aggregate operations [65], arithmetic constraints [64], modular-stratified negation [52], and non-ground tuples [66].

2.2.1.6 XSB

XSB was developed at the State University of New York in 1992. XSB has a fundamental bottom-up extension, introduced through tabling (or memoing) [60], which makes it appropriate as an underlying query engine for deductive database systems. Because it eliminates redundant computation, the tabling extension makes XSB able to compute all modularly stratified datalog programs finitely and with polynomial data complexity. For non-stratified programs a meta-interpreter that has the same properties is provided. In addition XSB includes indexing capabilities greatly improved over those of standard Prolog systems [54].

2.2.1.7 Other Systems

Other Prolog systems like Sicstus [10], Quintus [62] and Ciao [8] have interfaces to relational databases. A higher level approach is exemplified by ProData [39], a library used in the LPA, SICStus, and Quintus implementations of Prolog. In [39], Lucas and Ltd call it a *transparent tight coupling*. Within ProData, all low-level database access functions are hidden from the user. It is a relation-level system, users specifying which database relations to be used as Prolog predicates. After such specification, database tuples are treated as Prolog facts. ProData is a standard of sorts. Besides the commercial systems using it, some effort was exerted to make the Ciao and XSB interfaces mimic it.

2.2.2 Techniques of Integration

According to Brodie and Jarke [6] there are four general methods of combining elements of Prolog and a relational database system:

1. Coupling of an existing Prolog implementation to an existing relational database system;

2. Extending Prolog to include some facilities of the relational database system;

3. Extending an existing relational database system to include some features of Prolog;

4. Tightly integrating logic programming techniques with those of relational database systems.

While the first three methods add, in some ways, features to pre-existing systems, the last may be viewed as building a system from scratch. Brodie and Jarke recommend this fourth alternative, saying that it is no more work than the second or third, and that the end result will be a more capable system.

2.2.2.1 Loose Coupling Versus Tight Coupling

Regarding coupled systems, the literature usually refers to systems of two types: *tight* and *loose*. The terms are used in various ways, however. Some authors appear to use the terms in reference to the underlying architectural connections between Prolog and the database system. Others appear to refer to the degree of integration from a programming point of view.

In [72], Venken defines a tight coupling to exist when Prolog and a database system are compiled together forming a single program. This matches the fourth architecture described by Brodie and Jarke [6]. It is natural to suppose that in such a system, there would be only one language involved in its manipulation. Lucas and Ltd [39] argue that a tight coupling exists *where external records are retrieved from the database and unified with Prolog terms as and when required*. With loose coupling, large chunks of information are fetched from a database into Prolog memory prior to being used by the program.

Another parameter, transparency, is specified where each database relation appears to be just another Prolog clause and can be invoked in normal Prolog fashion. This appears to be a programmatic consideration. In [16], Date defines loose coupling as providing a call level interface between the logic language and the database system; users would program both in Prolog and SQL, for instance *the user is definitely aware of the fact that there are two distinct systems involved*. This approach thus certainly does not provide the *seamless integration* referred to above. With tight coupling, *the query language includes direct support for the logical inferencing operations, thus the user deals with one language, not two*. Such uses of loose and tight coupling go against the usual meanings of the words in the computer industry, where systems are tightly coupled if they cannot function separately; they are loosely coupled if they can. Given this definition, all couplings of Prolog to databases since they connect independent systems via some software interface are loose couplings.

2.2.2.2 Relational-Level Access Versus View-Level Access

Regarding programmatic considerations, the most natural way of representing (and accessing) data stored in an external database for use in Prolog is simply to treat

relations in a database as predicates and treat their tuples as one would treat Prolog facts. Data in the database would be accessed in Prolog's normal depth-first search fashion. Importantly, with the exception of the routines needed to implement the transparent use of these database predicates, this method requires no changes to either Prolog or the database. Prolog gains the use of secondary storage and concurrent access and otherwise escapes unscathed. This is sometimes called relational access [18], or sometimes tuple-at-a-time access [47]. It is relational because only a single relation is involved in the query. It is tuple-at-a-time because generally only a single tuple is returned as a solution. The two terms are not quite interchangeable; a query involving one relation might return an entire set of solutions, and a query involving multiple relations could return solutions one-at-a-time. Prolog prefers to backtrack for further solutions rather than having them presented all at once. Relational access requires few changes to Prolog and the database, because it is easy to implement. However, it is very inefficient. Relation at a time access does not utilize any of the relational database's mechanisms for optimizing data retrieval. Relational databases are designed to take a complex query, determine an optimal plan for satisfying that query, and execute that plan. With relation at a time access, since queries are of the simplest possible variety, no optimization is possible.

The alternative to relation-level access is called view-level access. Here, a complex query is passed to the database system, and it is the database system which does all of the work in satisfying the query (importantly, it is not Prolog). Depending upon how it is implemented, solutions can be returned tuple-at-a-time or set-at-a-time. The improvement in performance using this method can be staggering. Solving a given problem might take a single call to the database system and less than a second for view access. Solving the same problem might take thousands of calls and many hours for relational access [20]. This is not surprising, for relational access is merely a variation of a depth-first search, which is a blind search. The drawback to view-level access is that it generally ruins the transparent use of the database. Queries to the database, if they are to be efficient, are generally isolated from the rest of the Prolog program upon being written.

In practice, almost all real world systems linking Prolog and a relational database system simply tack on a software interface between a pre-existing Prolog implementation and a pre-existing relational database system. In other words, the Prolog and

database systems are loosely coupled. An interface allows Prolog to query the database when needed, either by translating Prolog goals to SQL or by embedding SQL directly into the Prolog code. The database links allow Prolog varying degrees of control over databases. Some are very low level, meaning that the user must keep track of things such as connection handles and cursors. The benefit of this is greater control over program execution and more subtle access to databases. The drawback is that an inexpert programmer can easily write dangerous code.

2.2.3 Deductive Database System Implementations

We next summarize, in Tables 2.3 and 2.4, the main differences between some of the most well-known deductive database systems. These tables were adapted from [54]. Table 2.3 compares the following features:

Recursion. Most systems allow the rules to use general recursion. However, a few limit recursion to linear recursion or to restricted forms related to graph searching, such as transitive closure.

Negation. Most systems allow negated subgoals in rules. When rules involve negation, there are normally many minimal fixpoints that could be interpreted as the meaning of the rules, and the system has to select from among these possibilities one model that is regarded as the intended model, against which queries will be answered.

Aggregation. A problem similar to negation comes up when aggregation (sum, average, etc.) is allowed in rules. More than one minimal model normally exists, and the system must select the appropriate model.

Table 2.4 compares the following features:

Updates. Logical rules do not, in principle, involve updating of the database. However, most systems have some approach to specifying updates, either through special dictions in the rules or update facilities outside the rule system. Systems that support updates in logical rules have a "yes" in the table.

Name	Developers	Refs	Recursion	Negation	Agregation
Aditi	U. Melbourne	[25]	general	stratified	stratified
COL	INRIA	[1]	stratified	stratified	superset stratified
ConceptBase	U. Aachen	[32]	general	local stratified	no
CORAL	U. Wisconsin	[53]	general	modularly stratified	modularly stratified
EKS-VI	ECRC	[28]	general	stratified	superset stratified
LogicBase	Simon Fraser U.		linear, some nonlinear	stratified	no
DECLARE	MAD Intelligent Systems	[35]	general	locally stratified	superset stratified
Hy+	U. Toronto	[42]	path queries	stratified	stratified
X4	U. Karlsruhe	[45]	general, but only binary preds	no	no
LDL LDL++	MCC	[67]	general	stratified restricted local	stratified restricted local
LOGRES	Polytechnic of Milan	[9]	linear	inflationary semantics	stratified
LOLA	Technical U. of Munich	[22]	general	stratified	computed predicates
Glue-Nail	Stanford U.	[17]	general	well-founded	glue only
Starburst	IBM Almaden	[46]	general	stratified	stratified
XSB	SUNY Stony Brook	[60]	general	well-founded	modularly stratified

Table 2.3: Summary of prototypes (part I)

Integrity Constraints. Some deductive systems allow logical rules that serve as integrity constraints.

Optimizations. Deductive systems need to provide some optimization of queries. Common techniques include magic sets or similar techniques for combining the benefits of both top-down and bottom-up processing, and seminaive evaluation for avoiding some redundant processing. A variety of other techniques are used by various systems, and summarized in this table.

Storage. Most systems allow External Database (EDB) relations to be stored on disk, but some also store Intensional Database (IDB) relations in secondary storage. Supporting disk-resident data efficiently is a significant task.

Interfaces. Most systems connect to one or more other languages or systems. Some of these connections are embeddings of calls to the deductive system in another language, while other connections allow other languages or systems to be invoked from the deductive system. Table 2.5 refers to this capability as extensibility; it is very useful for large applications.

Name	Updates	Constr	Optimizations	Storage	Interfaces
Aditi	no	no	magic sets, SN, join-order selection	EDB, IDB	Prolog
COL	no	no		main memory	ML
ConceptBase	yes	yes	magic sets, SN	EDB	C, Prolog
CORAL	yes	no	magic sets, SN, context factoring, projection pushing	EDB, IDB	C, C++, Extensible
EKS-VI	yes	yes	query-subquery, left/right linear,	EDB, IDB	Persistent Prolog
LogicBase	no	no	chain-based evaluation	EDB, IDB	C, C++, SQL
DECLARE	no	no	magic sets, SN, projection pushing	EDB	C, Lisp
Hy+	no	no		main memory	Prolog, CORAL, LDL, Smalltalk
X4	no	yes	top-down	EDB	Lisp
LDL, LDL++	yes	no	magic sets, SN left/right linear, projection pushing, bushy depth-first	EDB	C, C++, SQL
LOGRES	yes	yes	algebraic, SN	EDB, IDB	INFORMIX
LOLA	no	yes	magic sets, SN, projection pushing, join-order selection	EDB	TransBase (SQL)
Glue-Nail	glue only	no	magic sets, SN, right-linear, join-order selection	EDB	
Starburst	no	no	magic sets, SN variant	EDB, IDB	Extensible
XSB	no	no	memoing, top-down	EDB, IDB	C, Prolog

Table 2.4: Summary of prototypes (part II)

2.3 Chapter Summary

Throughout this chapter, Prolog and relational database systems were portrayed to comprehend possible coupling techniques between the two systems. Various concepts used in deductive database, together with the main alternatives used to couple a logic system with a relational database system were also introduced for a better understanding of the next chapters.

Chapter 3

Development Tools

This chapter describes the two main systems used in this thesis: the Yap Prolog system and the MySQL database management system. First, we briefly describe the main features of each system and then we focus on how to use their C language interface to build client programs. At the end, we describe Draxler's Prolog to SQL compiler as a means to optimize the translation of queries between Yap Prolog and MySQL.

3.1 The Yap Prolog System

Yap Prolog is a high-performance Prolog compiler that extends the Warren Abstract Machine (WAM) [75] with several optimizations for better performance. Yap follows the Edinburgh tradition, and is largely compatible with the ISO-Prolog standard, Quintus [62] and SICStus [10] Prolog. Yap has been developed since 1985. The original version was written in assembly, C and Prolog, and achieved high performance on m68k based machines. The assembly code was used to implement the WAM emulators. Later emulators supported the VAX, SPARC, and MIPS architectures. Work on the more recent version of Yap strives at several goals:

Portability: the whole system is now written in C. Yap compiles in popular 32 bit machines, such as Suns and Linux PCs, and in 64 bit machines, the Alphas running Unix and Linux.

Performance: the Yap emulator is comparable to or better than well-known Prolog

systems. The current version of Yap performs better than the original one, written in assembly language.

Robustness: the system has been tested with a large array of Prolog applications.

Extensibility: Yap was designed internally from the beginning to encapsulate manipulation of terms. These principles were used, for example, to implement a simple and powerful C-interface. The new version of Yap extends these principles to accommodate extensions to the unification algorithm, that we believe will be useful to implement extensions such as constraint programming.

Completeness: Yap has for a long time provided most built-in implementations expected from an Edinburgh Prolog. These include I/O functionality, database operations, and modules. Work on Yap aims now at being compatible with the Prolog standard.

Openness: new developments of Yap will be open to the user community.

Research: Yap has been a vehicle for research within and outside our group. Currently research is going on parallelisation and tabling, and support for Bayesian Networks. Yap 4.0 and early versions of Yap 4.1 were distributed under a license that enables free use in academic environments. From Yap4.1.15 onwards Yap is distributed under Perls's Artistic license. The developers follow an open development model: sources to the system are always made available from the home page, and contributions from users are always welcome.

3.2 C Language Interface to Yap

As many other Prolog systems, Yap provides an interface for writing predicates in other programming languages, such as C, as external modules. We will use a small example to briefly explain how it works. Assume that the user requires a predicate `my_random(N)` to unify `N` with a random number. To do so, first a `my_rand.c` module should be created. Figure 3.1 shows the code for it.

Next the module should be compiled to a shared object and then loaded under Yap by calling the `load_foreign_files()` routine. After that, each call to `my_random(N)` will unify `N` with a random number. Despite its small size, the example shows the key

```
#include "Yap/YapInterface.h"      // header file for the Yap interface to C

void init_predicates(void) {
  Yap_UserCPredicate("my_random", c_my_random, 1);
}

int c_my_random(void) {
  Yap_Term number = Yap_MkIntTerm(rand());
  return(Yap_Unify(Yap_ARG1,number));
}
```

Figure 3.1: My first Yap external module

aspects about the Yap interface. The include statement makes available the macros for interfacing with Yap. The `init_predicates()` procedure tells Yap the predicates being defined in the module. The function `c_my_random()` is the implementation of the desired predicate. Note that it has no arguments even though the predicate being defined has one. In fact the arguments of a Prolog predicate written in C are accessed through the macros `Yap_ARG1`, ..., `Yap_ARG16` or with `Yap_A(N)` where `N` is the argument number. In our example, the function uses just one local variable of type `Yap_Term`, the type used for holding Yap terms, where the integer returned by the standard Unix function `rand()` is stored as an integer term (the conversion is done by `Yap_MkIntTerm()`). Then it calls `Yap_Unify()`, to attempt the unification with `Yap_ARG1`, and returns an integer denoting success or failure.

3.2.1 Yap Terms

Terms, from the C point of view, can be classified as:

- *uninstantiated variables*;
- *instantiated variables*;
- *integers*;
- *floating-point numbers* (floats);
- *atoms* (symbolic constants);
- *pairs*;

- *compound terms.*

Integers, floats and atoms are respectively denoted by the following primitives `Yap_Int`, `Yap_Float` and `Yap_Atom`. For atoms, Yap includes primitives for associating atoms with their names: `Yap_AtomName()` returns a pointer to the string for the atom; and `Yap_LookupAtom()` looks up if an atom is in the standard hash table, and if not, inserts it. Table 3.1 lists the complete set of the available primitives to test, construct and destruct Yap terms.

Term	**Test**	**Construct**	**Destruct**
uninst var	Yap_IsVarTerm()	Yap_MkVarTerm()	(none)
inst var	Yap_NonVarTerm()		
integer	Yap_IsIntTerm()	Yap_MkIntTerm()	Yap_IntOfTerm()
float	Yap_IsFloatTerm()	Yap_MkFloatTerm()	Yap_FloatOfTerm()
atom	Yap_IsAtomTerm()	Yap_MkAtomTerm() Yap_LookupAtom()	Yap_AtomOfTerm() Yap_AtomName()
pair	Yap_IsPairTerm()	Yap_MkNewPairTerm() Yap_MkPairTerm()	Yap_HeadOfTerm() Yap_TailOfTerm()
compound term	Yap_IsApplTerm()	Yap_MkNewApplTerm() Yap_MkApplTerm()	Yap_ArgOfTerm() Yap_FunctorOfTerm()
		Yap_MkFunctor()	Yap_NameOfFunctor() Yap_ArityOfFunctor()

Table 3.1: Primitives for manipulating Yap terms

A pair is a term which consists of a tuple of two terms, designated as the *head* and the *tail* of the term. Pairs are most often used to build lists. One can construct a new pair from two terms, `Yap_MkPairTerm()`, or just build a pair whose head and tail are new unbound variables, `Yap_MkNewPairTerm()`. By using the primitives `Yap_HeadOfTerm()` and `Yap_TailOfTerm()`, it is possible to fetch the head and the tail of a pair.

A compound term consists of a *functor* and a sequence of terms with length equal to the *arity* of the functor. A functor, denoted in C by `Yap_Functor`, consists of an atom (functor name) and an integer (functor arity). As for pairs, we can construct compound terms from a functor and an array of terms, `Yap_MkApplTerm()`, or just build a compound term whose arguments are unbound variables, `Yap_MkNewApplTerm()`. Yap also

includes primitives to construct functors, `Yap_MkFunctor()`, to obtain the name and arity of a functor, the `Yap_NameOfFunctor()` and `Yap_ArityOfFunctor()` primitives, and to fetch the functor and terms in a compound term, the `Yap_FunctorOfTerm()` and `Yap_ArgOfTerm()` primitives.

To unify Prolog terms using the C API, Yap provides the following single primitive `Yap_Unify(Yap_Term a, Yap_Term b)`. This primitive attempts to unify the terms `a` and `b` returning `TRUE` if the unification succeeds or `FALSE` otherwise.

We next show an example that illustrates how these primitives can be used to construct and unify Prolog terms. Consider, for example, that we want to construct two compound terms, `p(VAR1,1)` and `p(a,VAR2)`, and perform its unification, that is, unify the variable `VAR1` with the atom `a` and the variable `VAR2` with the integer `1`. A possible implementation is shown next in Fig. 3.2.

```
YAP_Term arg[2], p1, p2;
YAP_Functor f;

f = YAP_MkFunctor(YAP_LookupAtom("p"), 2);          // construct functor p/2

arg[0] = Yap_MkVarTerm();
arg[1] = Yap_MkIntTerm(1);
p1 = YAP_MkApplTerm(f, 2, args);       // construct compound term p(VAR1, 1)

arg[0] = YAP_MkAtomTerm(YAP_LookupAtom("a"));
arg[1] = Yap_MkVarTerm();
p2 = YAP_MkApplTerm(f, 2, args);       // construct compound term p(a, VAR2)

YAP_Unify(t1, t2);                                          // unify both terms
```

Figure 3.2: Constructing and unifying compound terms in a Yap external module

3.2.2 Writing Predicates in C

Building interesting modules cannot be accomplished without two extra functionalities. One is to call the Prolog interpreter from C. To do so, first we must construct a Prolog goal `G`, and then it is sufficient to perform `YapCallProlog(G)`. The result will be `FALSE`, if the goal failed, or `TRUE` otherwise. When this is the case, the variables in `G` will store the values they have been unified with. The other interesting functionality is how we can define predicates. Yap distinguishes two kinds of predicates:

Deterministic predicates: which either fail or succeed but are not backtrackable;

Backtrackable predicates: which can succeed more than once.

Deterministic predicates are implemented as C functions with no arguments which should return zero if the predicate fails and a non-zero value otherwise. They are declared with a call to `Yap_UserCPredicate(char *name, int *f(), int arity)`, where the first argument is the name of the predicate, the second the name of the C function implementing the predicate, and the third is the arity of the predicate.

For backtrackable predicates we need two C functions: one to be executed when the predicate is first called, and other to be executed on backtracking to provide (possibly) other solutions. Backtrackable predicates are similarly declared, but using instead `Yap_UserBackCPredicate(char *name,int *f_init(),int *f_cont(),int arity, int sizeof)`, where `name` is the name of the predicate, `f_init` and `f_cont` are the C functions used to start and continue the execution of the predicate, `arity` is the predicate arity, and `sizeof` is the size of the data to be preserved in the stack (its use is detailed next).

When returning the last solution, we should use `Yap_cut_fail()` to denote failure, and `Yap_cut_succeed()` to denote success. The reason for using `Yap_cut_fail()` and `Yap_cut_succeed()` instead of just returning a zero or non-zero value, is that otherwise, when backtracking, our function would be indefinitely called.

Consider, for example, a predicate `lessthan(N,M)` that returns in `M` by backtracking all the positive integers less than `N`. `N` should be instantiated with an integer and `M` should be an uninstantiated argument. The predicate should succeed and provide by backtracking all the positive integers less than `N` for the `N-1` first calls and fail for the `N`th call. Figure 3.3 shows the code that implements this predicate.

The `c_lt_init()` function starts by testing if the arguments are of the desired type. When this is not the case, it calls `YAP_cut_fail()` and fails by returning `FALSE`. Remember that calling `Yap_cut_fail` is necessary because otherwise function `c_lt_cont()` would be indefinitely called when backtracking. On the other hand, if the correct arguments are given, the function converts the first argument to an integer C data-type (the conversion is done by `YAP_IntOfTerm()`) and then it tests if it is a positive value. If not, it also calls `YAP_cut_fail()` and fails. Otherwise, it calls `YAP_Unify()`

```
void init_predicates(void) {
  YAP_UserBackCPredicate("lessthan", c_lt_init, c_lt_cont, 2, sizeof(int));
}

int c_lt_init(void) {   // to be executed when the predicate is first called
  int limit, *number;
  YAP_Term n = YAP_ARG1;
  YAP_Term m = YAP_ARG2;

  if (YAP_IsIntTerm(n) && YAP_IsVarTerm(m)) {
    limit = YAP_IntOfTerm(n);
    if (limit > 0) {
      YAP_PRESERVE_DATA(number, int);
      *number = 1;
      return (YAP_Unify(m, YAP_MkIntTerm(*number)));
    }
  }
  YAP_cut_fail();
  return FALSE;
}

int c_lt_cont(void) {                       // to be executed on backtracking
  int limit, *number;
  YAP_Term n = YAP_ARG1;
  YAP_Term m = YAP_ARG2;

  limit = YAP_IntOfTerm(n);
  YAP_PRESERVED_DATA(number, int);
  *number++;
  if (*number < limit) {
    return (YAP_Unify(m, YAP_MkIntTerm(*number)));
  YAP_cut_fail();
  return FALSE;
}
```

Figure 3.3: The `lessthan/2` backtrackable predicate

to attempt the unification of 1 (the conversion is done by `YAP_MkIntTerm()`) with the second argument.

Note that, to obtain by backtracking the next integer, we need to preserve the last returned integer in a data structure associated with the current predicate call. This is done by calling `YAP_PRESERVE_DATA` to associate and allocate the memory space that will hold the information to be preserved across backtracking, and by calling `YAP_PRESERVED_DATA` to get access to it later. The first argument to these macros is the pointer to the corresponding memory space and the second is its type. The `c_lt_cont()` function uses the `YAP_PRESERVED_DATA` macro to obtain the last returned integer. It then updates the last returned integer and like the first function, it fails,

if reached the limit, or unifies the second argument with the new integer. For a more exhaustive description on how to interface C with Yap please refer to [13].

3.3 The MySQL Database Management System

MySQL was originally developed by Michael *Monty* Widenious in 1979 as a database tool for the Swedish company TcX. In 1994, TcX began looking for a SQL server to develop web applications. Since the existing ones were all very slow for TcX's large tables, TcX decided to develop its own server. The programming interface of this server was explicitly designed to be similar to the one used by mSQL, so many tools available to mSQL could be easily transferred to MySQL. In 1995, David Axmark tried to release MySQL on the Internet. David worked on the documentation and got MySQL to build with GNU. MySQL 3.11.1 was unleashed to the world in 1996 in the form of binary distributions for Linux and Solaris. Today, MySQL is used on many more platforms and is available in both binary and source forms.

MySQL is not an open source project because a license is necessary under certain conditions. Nevertheless, MySQL benefits from widespread popularity in the open source community, because the licensing terms are not very restrictive. MySQL is generally free unless it is used to make money by selling it or selling services that require it. Since its performance rivals any existing database system, MySQL runs on personal computers, commercial operating systems and enterprise servers. We next enumerate some of the important characteristics of the MySQL database software:

Connectivity: MySQL is fully networked and databases can be accessed from anywhere on the Internet, thus data can be shared with anyone anywhere using TCP/IP sockets on any platform.

Security: MySQL has a password system that is very flexible, secure, and allows host-based verification. Since all password traffic is encrypted when you connect to a server access control is certified.

Portability: MySQL runs on many varieties of UNIX, as well as on other non-Unix systems such as Windows and OS/2, from personal PCs to high-end servers.

Capability: Many clients can connect simultaneously to the server and use multiple databases. Several interfaces can access MySQL interactively to enter queries and view results such as: command-line clients, Web browsers or X Window System clients. In addition, a variety of programming interfaces are available for languages such as C, Peal, Java, PHP, and Python. Thus, it's possible to choose between prepacked client software and writing custom applications.

Speed and ease to use: MySQL developers claim that MySQL is the fasted database available. In addition, MySQL is a high-performance but relatively simple database system and is less complex to set up and administer than larger systems.

3.4 C Language Interface to MySQL

MySQL provides a client library written in C for writing client programs that access MySQL databases. This library defines an application programming interface that includes the following facilities:

- Connection management procedures;
- Establish and terminate sessions with a server;
- Procedures to construct, send and process the results of queries;
- Error handling procedures.

3.4.1 Writing Client Programs in C

Usually, the main purpose of a client program that uses the MySQL C API is to establish a connection to a database server in order to process a set of queries. Figure 3.4 shows the general code skeleton for a MySQL client program.

Initially, a connection handler (represented by the `MYSQL` data type) is allocated and initialized by calling the `mysql_init()` procedure. A connection is then established by calling the `mysql_real_connect()` procedure which includes, among others, arguments to define the name of the host to connect to, the database to use, and the name and password of the user trying to connect. Next, communication is done with the server

```
#include <mysql.h>                          // header file for the MySQL C API

int main() {
  MYSQL *conn;                                          // connection handler
  MYSQL_RES *res_set;                                          // result set
  MYSQL_ROW row;                                          // row contents
  char *query;                                          // SQL query string

  conn = mysql_init(...);      // obtain and initialize a connection handler
  mysql_real_connect(conn, ...);       // establish a connection to a server
  while(...) {
    query = ...;                                        // construct the query
    mysql_query(conn, query);                // issue the query for execution
    res_set = mysql_store_result(conn);           // generate the result set
    while ((row = mysql_fetch_row(res_set)) != NULL) {        // fetch a row
      ...                                   // do something with row contents
    }
    mysql_free_result(res_set);                      // deallocate result set
  }
  mysql_close(conn);                                // terminate the connection
}
```

Figure 3.4: Code skeleton for a MySQL client program

(possibly many times) in order to process a single or several queries. At last, the connection is terminated. Processing a query involves the following steps:

- Construct the query;
- Send the query to the server for execution;
- Handle the result set.

The result set (represented by the `MYSQL_RES` data type) includes the data values for the rows and also meta-data about the rows, such as the column names and types, the data values lengths, the number of rows and columns, etc.

The `mysql_query()` and `mysql_real_query()` are both used to send queries to the server. Since a query is considered a counted string and may contain anything, including binary data or null bytes, the `mysql_real_query()` is more generally used to deal with these all-purpose strings, because it is less restrictive than `mysql_query()`. Thus, queries that are passed to `mysql_query()` should be null-terminated strings, which means they cannot have `NULL` bytes in the text of the query. The advantage of

using `mysql_query()` is usually linked to the possibility of using standard C library string functions such as `strcopy()` and `sprintf()`.

A query may fail for many reasons. Some common causes are:

- The query contains a syntax error;
- The query is semantically illegal (for example, it refers to a non-existent column of a table);
- The user attempting to connect does not have sufficient privileges to access the information.

Queries can be classified into two wide-ranging categories: those that return a result and those that do not. Queries possessing statements such as `INSERT`, `DELETE`, and `UPDATE` are in the *no result returned* category, since they do not return any rows. For these queries, the only information returned is a simple count of the rows affected by the query goal. On the other hand, queries retaining statements such as `SELECT` and `SHOW` fall into the *result returned* category. After all, the whole purpose of issuing these statements is to retrieve information. The set of rows retrieved by theses queries is called a *result set*. Result sets are represented by the `MYSQL_RES` data type. This data type is a structure able to contain data values for the rows and meta-data about the values, such as, the column names and data value lengths. Note that, obtaining an empty result set, that is, one that returns zero rows, is distinct from obtaining *no result*.

3.4.2 Handling Queries that Return No Result

When a query that returns no result succeeds, it is possible to find out how many rows were affected by it by calling the `mysql_affected_rows()` procedure. For queries having statements such as `INSERT`, `REPLACE`, `DELETE`, or `UPDATE`, `mysql_affected_rows()` returns the number of rows inserted, replaced, deleted, or modified, respectively. The following example illustrates how to handle a query of this category.

```
if (mysql_query(conn, "INSERT INTO my_table SET name='JOHN'") != 0)
  printf("The query failed");
else
  printf("The query succeed: %lu rows affected", mysql_affected_rows(conn));
```

3.4.3 Handling Queries that Return a Result Set

Queries that return data do so in the form of a result set. It is important to realize that in MySQL, `SELECT` it is not the only query statement that returns rows, `SHOW`, `DESCRIBE` and `EXPLAIN` do so as well. After issuing a query, additional row-handling processing is necessary. Handling a result set also involves three steps:

- Generate the result set;
- Fetch each row of the result set to do something with it;
- Deallocate the result set.

A result set is generated by the `mysql_store_result()` or `mysql_use_result()` procedures. In the code skeleton of Fig. 3.4, the `mysql_store_result()` procedure was used to generate the result set. An alternative is to use the `mysql_use_result()` procedure. They are similar in that both take a connection handler and return a result set, but their implementations are quite different. The `mysql_store_result()` fetches the rows from the server and stores them in the client. Subsequent calls to `mysql_fetch_row()` simply return a row from the data structure that already holds the result set. On the other hand, `mysql_use_result()` does not fetch any rows itself. It simply initiates a row-by-row communication that must be completed by calling `mysql_fetch_row()` for each row. `mysql_store_result()` has higher memory and processing requirements because the entire result set is maintained in the client. `mysql_use_result()` only requires space to a single row at a time, and this can be faster because no complex data structures need to be setting up or handled. On the other hand, `mysql_use_result()` places a great burden on the server, which must hold rows of the result until the client fetches them all.

The `mysql_fetch_row()` must be called for each row of the result set. It returns `NULL` when there are no more rows left in the result set. Note that a row (represented by the `MYSQL_ROW` data type) is implemented as a pointer to an array of null terminated strings representing the values for each column in the row. Thus, when treating a value as, for instance, a numeric type, we need to convert the string beforehand. Moreover, accessing each value is simply a matter of accessing `row[i]`, with `i` ranging from 0 to the number of columns in the row minus one. The number of columns in a row can be obtained by calling the `mysql_num_fields()` procedure.

The `mysql_free_result()` is used to deallocate the memory used by the result set when it is no longer needed. The application will leak memory if this procedure is neglected. Of course, this situation is particularly troubling for long running applications, since the system will be slowly taken over by processing queries that consume increasing amounts of system resources. For a complete description on these topics and how to take fully advantage of the MySQL C API please refer to [19].

3.5 Prolog to SQL Translation

The interface between Prolog programs and database management systems is normally done via the SQL language. A particular Prolog predicate is assigned to a given relation in a database and its facts are made available through the tuples returned by a SQL query. This Prolog to SQL translation has been well described in the literature [31]. An important implementation of a generic Prolog to SQL compiler is the work done by Draxler [18]. It includes the translation of conjunctions, disjunctions and negation of goals, and also of higher-order constructs, such as grouping and sorting. Another important aspect of this work is the notion of *database set predicates*, which allows embedding the set-oriented evaluation of database systems into the standard tuple-oriented evaluation of Prolog, using Prolog itself to navigate this set structure. Draxler's Prolog to SQL compiler is entirely written in Prolog, and is efficient. It can thus be easily integrated in the pre-processing phase of Prolog compilers. In what follows we discuss the Draxler's compiler in more detail.

3.5.1 Database Schema Information

The implementation of database set predicates requires that a database access request be translated to the equivalent SQL query, and that the result of the database evaluation be retrieved and placed in a Prolog list data structure. Translating the database access request and the result relation retrieved from the database system can be implemented in standard Prolog. This translation is based on the schema information of the database to be accessed, and the translation procedure for the database access request. The database schema information is application dependent. For each database accessed schema information must be available. The translation

procedure is independent of the application program, but dependent of the database access language in the logic language and the target database query language.

Schema information is about relations and attributes of external relational databases. This schema information must be accessible to the translation program, so it can be able to map the predicates in the database goal to their appropriate database relations. The basic problem to overcome in the translation from Prolog to SQL is addressing different arguments and attributes. Prolog arguments are identified through their position in terms, whereas in SQL attributes are identified through their names and relations. Thus, the mapping of Prolog terms to SQL relations is a mapping of argument positions to qualified attribute names. In Draxler's compiler, the database schema information, is represented through Prolog facts. To illustrate this, consider an example from the world of flight connections and airplanes.

```
FLIGHT := FLIGHT_NO X DEPARTURE X DESTINATION X PLANE
PLANE := TYPE X SEATS
```

FLIGHT and PLANE are two relation tables as defined above. The attributes FLIGHT_NO, DEPARTURE, DESTINATION, PLANE/TYPE and SEATS represent respectively, the set of flight numbers, the set of possible departure hours, the set of airports, the set of airplane types, and the set of possible seats in a plane (natural numbers from 0 to 1000). In Prolog, this database schema information is stored as follows:

```
%% relation(PredicateName,RelationName,Arity)
relation(flight,'FLIGHT',4).
relation(plane,'PLANE',2).

%% attribute(AttributeName,RelationName,Position)
attribute('FLIGHT_NO','FLIGHT',1).
attribute('DEPARTURE','FLIGHT',2).
attribute('DESTINATION','FLIGHT',3).
attribute('TYPE','FLIGHT',4).
attribute('TYPE','PLANE',1).
attribute('SEATS','PLANE',2).
```

Note that type information can be also included in the schema description as an additional argument of the attribute description:

```
attribute('FLIGHT_NO','FLIGHT',1,'CHAR(5)').
```

3.5.2 Translation Rules

The database access language is defined to be a restricted sub-language of Prolog equivalent in expressive power to relational calculus (no recursion is allowed). A database access request consists of a database goal and a projection term.

A projection term is a term. Each variable in the projection term must also occur in the database goal. Note that the projection term may be an arbitrarily complex term. The database goal expresses the query that will be evaluated by the database system. The operators union, intersection, difference, selection and join, for relational databases, must be expressed through the database goal.

A database goal is a list of positive or negative literals $L_1, \ldots, L_n$ $(n \geq 1)$, connected through the logical connectives "," and ";" such that:

- Each L_i is a database predicate, a comparison operation, or an arithmetic or aggregate function;
- At least one L_i is a positive database predicate;
- All input arguments of functions are bound;
- All arguments of comparison operations are bound.

The functor of a database predicate is mapped to the appropriate relation table name, and each of the predicate arguments is assigned a relation attribute. The comparison operations in the database goal must be expressible in SQL. This is true for the standard comparison operations $>$, $<$, $=$, etc. The allowed functors for aggregate functions are `sum`, `avg`, `min`, `max`, and `count`.

Due to the order of execution imposed by the Prolog control strategy, complex database goals must contain first the calls to the positive database predicates that return bindings for the variable arguments. SQL queries consist of at least a `SELECT` and a `FROM` part, with optional `WHERE`, `GROUP BY`, and `ORDER BY` parts. The keyword `DISTINCT`, which is used to eliminate duplicate entries in the result relation, is also optional. The translation of database access requests to SQL queries is done according to the following informal translation rules.

- A conjunction of database goals is translated to a single SQL query.
- Disjunctions of database goals are translated to several SQL queries connected through the `UNION` operator.
- Negated database goals are translated to negated existential sub-queries.
- Goal functors other than comparison operators and function symbols are translated to relation names and assigned a unique identifier (range variable) in the `FROM` part of the query.
- Comparison operations and functions are translated to the equivalent SQL comparisons and functions over relation attributes or constant values.
- Variables in the projection term are translated to attribute names in the `SELECT` part of the query. These attributes are qualified by range variables.
- Variables occurring only once in the database goal are not translated.
- Shared variables, i.e. variables occurring in at least two base calls in the database goal, are translated to join-conditions in the `WHERE` part.
- Constant values in the database goal translate to comparison operations of the appropriate qualified relation attribute and the constant value in the `WHERE` part.
- Constants in the projection term are not translated. Depending on the database set predicate additional rules may be used for translation of higher-order constructs. For example, `db_setof/3` returns a result relation for each binding of the free variables in the database goal, duplicates are eliminated from the result relation, and the result relation is sorted. This requires the use of a `GROUP BY` and an `ORDER BY` part in the query.
- Free variables, i.e. variables occurring only in the database goal, are translated to qualified attributes in the `GROUP BY` part.
- Free variables and the variables occurring in the projection term are translated to qualified attributes in the `ORDER BY` part. The order in which the free variables and the projection term variables occur determines the order according to which the result relation is sorted.

- The keyword `DISTINCT` is added to the `SELECT` part of the query.

With these rules, the translation of higher-order constructs to SQL is achieved. Aggregate functions may only appear in the `SELECT` part of an SQL query. In database set predicates this is expressed by a projection term which contains a variable to hold the result of the aggregate function, and a database goal with the aggregate function represented through a ternary term, the functor of which is one of `sum`, `avg`, `min`, `max`, or `count`.

3.5.3 Translation Process

At the top level, the compiler consists of a `translate/3` predicate, where the first argument defines the projection term of the database access request, the second argument defines the database goal which expresses the query, and the third argument is used to return the correspondent SQL select expression. Figure 3.5 shows the translation steps of the `translate/3` predicate.

```
translate(ProjectionTerm,DatabaseGoal,Code):-
  %% lexical analysis
  tokenize_selection(DatabaseGoal,TokenizedGoal),
  tokenize_projection(ProjectionTerm,TokenProjection),
  %% syntax analysis
  push_negation_inside(TokenizedGoal,NegatedGoals),
  disjunction(NegatedGoals,Disjunction),
  %% code generation
  code_generation(Disjunction,TokenProjection,Code),
  %% output
  printqueries(Code).
```

Figure 3.5: The translation steps of the `translate/3` predicate

The compiler is called with a database access request consisting of a projection term and a complex database goal. Both terms may contain variable arguments. The lexical analysis transforms these terms into ground terms to prevent accidental instantiation of variable arguments during later translation steps. The ground database goal is then transformed into a logically equivalent disjunction of conjunctions in which negation appears only immediately before a simple goal. These conjunctions and the ground representation of the projection term are then passed on to the code generator to generate an intermediate structure which consists of separate lists of relation table

names, qualified attributes and conditions for the SQL query. This intermediate structure yields a query term from which the final output is created.

For example, consider again the database from the world of flight connections and planes, and the following database request: *"retrieve the flight numbers, departures, destinations and planes, for all the flights"*. Using the `translate/3` predicate, this request can be written as:

```
?- translate(proj_term(Flight_No,Departure,Destination,Type),
             flight(Flight_No,Departure,Destination,Type),
             QueryString).
```

As a result we will obtain:

```
QueryString = SELECT A.FLIGHT_NO, A.DEPARTURE, A.DESTINATION, A.TYPE
              FROM FLIGHT A
```

Consider now a different request that uses both tables in the database: *"retrieve the flight numbers for the flights with large planes, i.e. planes with more than 150 seats"*. This request can be written as:

```
?- translate(proj_term(Flight_No),
             (flight(Flight_No,_,_,Type), plane(Type,Seats), Seats > 150),
             QueryString).
```

The result will be:

```
QueryString = SELECT A.FLIGHT_NO
              FROM FLIGHT A, PLANE B
              WHERE A.TYPE = B.TYPE AND B.SEATS > 150
```

3.6 Chapter Summary

In this chapter, we introduced the Yap Prolog and the MySQL systems, and described in detail their client libraries to the C language. These libraries are necessary to implement the communication interface between both systems. Another important feature for linking both systems is how translation from Prolog to SQL is done. To optimize the translation of queries between both systems we introduced the Prolog to

SQL compiler written by Draxler. The interaction among all these features is essential to the development of the alternative approaches for coupling logic programming with relational databases that we propose in the next chapter.

Chapter 4

Coupling Approaches

In this chapter, we present and discuss three alternative approaches for coupling logic programming with relational databases. We consider three main approaches: **(i)** asserting database tuples as Prolog facts; **(ii)** accessing database tuples by backtracking; and **(iii)** transferring unification to the database. We present a detailed step-by-step description of each approach and discuss their advantages an disadvantages.

4.1 Generic Architecture

We used Yap and MySQL as the base systems to implement our three alternative approaches for coupling logic programming with relational databases. To develop running examples of each approach, we took advantage of the existent client libraries to implement the interface level as discussed in the previous chapter. The three approaches share a common generic architecture that is illustrated in Fig. 4.1.

The `yap2mysql.c` is the main module. It defines the low-level communication predicates and uses the Yap and MySQL interfaces to the C language to implement them. The `sqlcompiler.pl` is Draxler's Prolog to SQL compiler. It will be indispensable to implement successfully the third approach. The `yap2mysql.pl` is the Prolog module which the user should interact with. It defines the high-level predicates to be used and abstracts the existence of the other modules. After consulting the `yap2mysql.pl` module, the user starts by calling the `db_open/5` predicate to define a connection to a database server. Then, it calls `db_import/3` to map database relations into Prolog

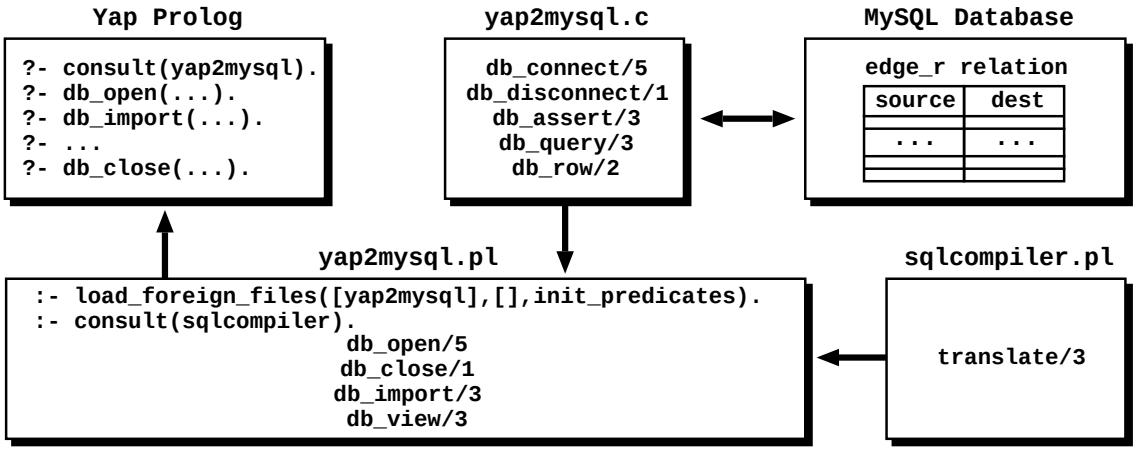

Figure 4.1: Generic architecture for the coupling approaches

predicates. We also allow, on the third approach, the definition of database views based on these predicates by the use of `db_view/3`. Next, it uses the mapped predicates to process query goals and, at last, it calls `db_close/1` to terminate the session with the database.

The Prolog definition of the `db_open/5` and `db_close/1` predicates is common to all the approaches.

```
db_open(Host,User,Passwd,Database,ConnName) :-
  db_connect(Host,User,Passwd,Database,ConnHandler),
  set_value(ConnName,ConnHandler).

db_close(ConnName) :-
  get_value(ConnName,ConnHandler),
  db_disconnect(ConnHandler).
```

The `db_open/5` and `db_close/1` predicates allow the user to define multiple connections. The predicates that make the low-level communication with the database server are `db_connect/5` and `db_disconnect/1`. Both predicates, `db_connect/5` and `db_disconnect/1`, are defined in the `yap2mysql.c` module as deterministic predicates in C. Figure 4.2 shows their implementation.

Predicate `db_connect/5` establishes a connection to the database. The arguments `Host`, `User`, `Passwd` and `Database` are necessary to validate the communication with the database management system, and must be all passed to the predicate. Predicate `db_disconnect/1` terminates a connection to the database. The only needed argument is a reference to the correspondent `ConnHandler` argument obtained from a previous

```
void init_predicates(void) {
  ...
  Yap_UserCPredicate("db_connect", c_db_connect, 5);
  Yap_UserCPredicate("db_disconnect", c_db_disconnect, 1);
}

int c_db_connect(void) {
  char *host = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG1));
  char *user = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG2));
  char *passwd = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG3));
  char *db = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG4));
  MYSQL *conn;

  // obtain and initialize a connection handler
  if ((conn = mysql_init(NULL)) == NULL)
    return FALSE;
  // establish a connection to a server
  if (mysql_real_connect(conn, host, user, passwd, db, 0, NULL, 0) == NULL)
    return FALSE;
  // unify the connection handler with the fifth argument
  if (!YAP_Unify(YAP_ARG5, YAP_MkIntTerm((int) conn)))
    return FALSE;
  return TRUE;
}

int c_db_disconnect(void) {
  // obtain the connection handler
  MYSQL *conn = (MYSQL *) YAP_IntOfTerm(YAP_ARG1);
  // terminate the connection
  mysql_close(conn);
  return TRUE;
}
```

Figure 4.2: The C implementation of `db_connect/5` and `db_disconnect/1`

call to the `db_connect/5` predicate.

The coupling approaches are implemented by changing the definition of `db_import/3` predicate. The implementation of the `db_import/3` predicate is thus defined by the coupling approach being used. We next describe and discuss our three approaches, and for that we will use a MySQL relation, `edge_r`, with two attributes, `source` and `dest`, where each tuple represents an edge of a directed graph. The Prolog predicate associated with this relation will be referred as `edge/2`. To map the database relation `edge_r` into the Prolog predicate `edge/2` we should call `db_import(edge_r,edge,my_conn)`, where `my_conn` is the same argument as the `ConnName` argument of a previous call to a `db_open/5` predicate.

4.2 Asserting Database Tuples as Prolog Facts

A first approach for mapping database relations into Prolog predicates is to assert the complete set of tuples of a relation as Prolog facts. To do so, a single connection to the database is needed to fetch the complete set of tuples. After that, the asserted facts are used as usual by the Prolog interpreter. To implement this approach, the `db_import/3` predicate is simply an alias to the `db_assert/3` predicate.

```
db_import(RelName,PredName,ConnName) :-
  get_value(ConnName,ConnHandler),
  db_assert(RelName,PredName,ConnHandler).
```

The Prolog predicate `db_assert/3` is implemented in C as a determinist predicate. Figure 4.3 shows its implementation. First, it constructs a '`SELECT * FROM <RelName>`' query in order to fetch and store the complete set of tuples in a result set. Then, for each row of the result set, it calls the Prolog interpreter to assert the row as a Prolog fact. To do so, it constructs Prolog terms of the form `assert(f_pred(t_args[0],...,t_args[arity-1]))`, where `f_pred` is the predicate name for the asserted facts and `t_args[]` are the data values for each row.

This approach minimizes the number of database communications, and can benefit from the Prolog indexing mechanism to optimize certain subgoal calls. On the other hand, memory requirements are higher, because it duplicates the entire set of tuples in the database as Prolog facts. Moreover, real time modifications to the database done by others are not visible to the Prolog system. Even Prolog modifications to the set of asserted tuples can be difficult to synchronize with the database.

4.3 Accessing Database Tuples by Backtracking

The next approach takes advantage of the Prolog backtracking mechanism to access the database tuples. Thus, when mapping a database relation into a Prolog predicate it uses the Yap interface functionality that allows defining backtrackable predicates, in such a way that every time the computation backtracks to such predicates, the tuples in the database are fetched one-at-a-time. To implement this approach, we changed the `db_import/3` definition. This predicate still receives the same arguments, but now

```
void init_predicates(void) {
  ...
  Yap_UserCPredicate("db_assert", c_db_assert, 3);
}

int c_db_assert(void) {
  char *rel = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG1));
  char *pred = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG2));
  MYSQL *conn = (MYSQL *) YAP_IntOfTerm(YAP_ARG3);
  YAP_Term t_args[arity], t_pred, t_assert;
  YAP_Functor f_pred, f_assert;
  int i, arity;
  MYSQL_RES *res_set;
  MYSQL_ROW row;
  char query[15 + strlen(rel)];

  sprintf(query, "SELECT * FROM %s", rel);            // construct the query
  if (mysql_query(conn, query) != 0)        // issue the query for execution
    return FALSE;
  if ((res_set = mysql_store_result(conn)) == NULL)   // generate result set
    return FALSE;
  arity = mysql_num_fields(result_set);
  f_pred = YAP_MkFunctor(YAP_LookupAtom(pred), arity);
  f_assert = YAP_MkFunctor(YAP_LookupAtom("assert"), 1);
  while ((row = mysql_fetch_row(res_set)) != NULL) {
    for (i = 0; i < arity; i++) {       // test each column data type to ...
      MYSQL_FIELD *field = mysql_fetch_field_direct(res_set, i);
      if (field->type == FIELD_TYPE_SHORT)
        t_args[i] = YAP_MkIntTerm(atoi(row[i]));
      else if (field->type == FIELD_TYPE_FLOAT)
        t_args[i] = YAP_MkFloatTerm(atof(row[i]));
      else if ...
    }                                  // ... construct the appropriate term
    t_pred = YAP_MkApplTerm(f_pred, arity, t_args);
    t_assert = YAP_MkApplTerm(f_assert, 1, &t_pred);
    YAP_CallProlog(t_assert);            // asserts the row as a Prolog fact
  }
  mysql_free_result(res_set);                        // deallocate result set
  return TRUE;
}
```

Figure 4.3: The C implementation of `db_assert/3`

it dynamically constructs and asserts a clause for the predicate being mapped. In this approach, the `db_import/3` predicate is defined as follows.

```
db_import(RelName,PredName,ConnName):-
  get_value(ConnName,ConnHandler),

  % generate query "describe <RelName>"
  name('describe',L1),
  name(RelName,L2),
  append(L1,L2,L),
```

```
  name(QueryDescribe,L),

  % send query to the database and obtain result set
  db_query(ConnHandler,QueryDescribe,ResultSetDescribe),

  % retrieve the arity of the result set
  db_num_fields(ResultSetDescribe,Arity),

  % construct the literals of the clause to assert
  make_pred_head(PredName,Arity,ArgList,PredTerm),
  make_db_query(ConnHandler,RelName,ResultSet,DbQueryTerm),
  make_db_row(ResultSet,ArgList,DbRowTerm),

  % assert the clause for the predicate being mapped
  assert(':-'(PredTerm,(DbQueryTerm,DbRowTerm))).
```

To construct the clause for the predicate being mapped, the `db_import/3` predicate uses the `db_query/3` and `db_row/2` predicates. For example, if we call the goal `db_import(edge_r,edge,my_conn)`, the following clause will be asserted.

```
edge(A,B) :-
  db_query(conn_handler_id,'SELECT * FROM edge_r',ResultSet),
  db_row(ResultSet,[A,B]).
```

This clause allows to obtain, by backtracking, all the tuples for the query '`SELECT * FROM edge_r`'. For example, if later, we call `edge(A,B)`, the tuples in the database are fetched one-at-a-time:

```
?- edge(A,B).

A = 1,
B = 2 ? ;

A = 2,
B = 5 ? ;

A = 7,
B = 5 ? ;

no
```

The `db_query/3` predicate, used in the definition of `edge/2`, generates the result set for the query given as the second argument ('`SELECT * FROM edge_r`' in the `edge/2` example). The `db_query/3` predicate is implemented in the `yap2mysql.c` module as a deterministic predicate in C (see Fig. 4.4 below).

```
void init_predicates(void) {
  ...
  YAP_UserCPredicate("db_query", c_db_query, 3);
}

int c_db_query (void) {
  MYSQL *conn = (MYSQL *) YAP_IntOfTerm(YAP_ARG1);
  char *query = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG2));
  MYSQL_RES *res_set

  if (mysql_query(conn, query) != 0)        // issue the query for execution
    return FALSE;
  if ((res_set = mysql_store_result(conn)) == NULL)   // generate result set
    return FALSE;
  return YAP_Unify(YAP_ARG3, YAP_MkIntTerm((int)res_set));
}
```

Figure 4.4: The C implementation of `db_query/3`

Figure 4.5 shows the implementation of the `db_row/2` predicate. It is implemented as a backtrackable predicate in C. For backtrackable predicates, we can define a function for the first time the predicate is called, and another for calls via backtracking. In this case, the same function is used for both calls.

The `c_db_row()` function starts by fetching a tuple from the result set, through `mysql_fetch_row()`, and then it checks if the last tuple has already been reached. If not, it calls `YAP_Unify()` to attempt the unification of values in each attribute of the tuple (`row[i]`) with the respective predicate arguments (`[A,B]` in the example). If unification fails it returns `FALSE`, otherwise it returns `TRUE`. On the other hand, if the last tuple has been already reached, it deallocates the result set, calls `YAP_cut_fail()` and returns `FALSE`.

With this approach, all the data is concentrated in a single repository. This simplifies its manipulation and allows users to see real time modifications done by others. Furthermore, this minimizes memory requirements. However, if `mysql_store_result()` is used to generate the result set, the entire set of tuples will be duplicated on the client side. On the other hand, for non generic calls (calls with not all arguments unbound) some tuples may be unnecessarily fetched from the database. For example, if `edge(A,1)` is called, this turns `B` ground when passed to the `p_db_row()` function. Thus, only the tuples which satisfy this condition will unify. However, all tuples will be fetched from the database and Prolog unification will select the matching tuples

```
void init_predicates(void) {
  ...
  YAP_UserBackCPredicate("db_row", c_db_row, c_db_row, 2, 0);
}

int c_db_row(void) {
  MYSQL_RES *result_set = (MYSQL_RES *) YAP_IntOfTerm(YAP_ARG1);
  int i, arity = mysql_num_fields(result_set);
  MYSQL_ROW row;
  YAP_Term t_field, t_head, t_list = YAP_ARG2;

  if ((row = mysql_fetch_row(result_set)) != NULL) {
    for (i = 0; i < arity; i++) {
      MYSQL_FIELD *field = mysql_fetch_field_direct(res_set, i);
      if (field->type == FIELD_TYPE_SHORT)
        t_field = YAP_MkIntTerm(atoi(row[i]));
      else if (field->type == FIELD_TYPE_FLOAT)
        t_field = YAP_MkFloatTerm(atof(row[i]));
      else if ...
      head = YAP_HeadOfTerm(list);
      list = YAP_TailOfTerm(list);
      if (!YAP_Unify(head, t_field))              // unify the next argument
        return FALSE;
    }
    return TRUE;
  }
  mysql_free_result(result_set);                    // deallocate result set
  YAP_cut_fail();
  return FALSE;
}
```

Figure 4.5: The C implementation of `db_row/2`

upon retrieval of each tuple from the result set.

4.4 Transferring Unification to the Database

The last approach uses the `translate/3` predicate from Draxler's compiler to transfer the Prolog unification process to the MySQL engine. Instead of using Prolog unification to select the matching tuples for a non generic call, specific SQL queries are dynamically constructed using `translate/3` to match the call. By doing this, the tuples that do not succeed are discarded before performing unification. This prevents unnecessary tuples from being fetched from the database, thus minimizing memory requirements and optimizing computation. To implement this last approach the definition of `db_import/3` was extended to include the `translate/3` predicate. If

the previous example is considered, the following clauses will now be asserted.

```
edge(A,B) :-
  rename_term_vars(edge(A,B),NewTerm),
  translate(NewTerm,NewTerm,QueryString),
  queries_atom(QueryString,Query),
  db_query(conn_handler_id,Query,ResultSet),
  db_row(ResultSet,[A,B]).

relation(edge,edge_r,2).
attribute(source,edge_r,1,integer).
attribute(dest,edge_r,2,integer).
```

Note that if we use the variables in the head of the predicate being built (variables `A` and `B` in the example above) to construct the projection term for the `translate/3` predicate then these variables will became bound. To avoid this, we need an auxiliary term, identical to the initial one but with different variables. We thus implemented an auxiliary predicate, `rename_term_vars/2`, that builds, in the second argument, a term similar to the one in the first argument, but with the variables renamed. For example, if we call `rename_term_vars(edge(A,5),NewTerm)`, then `NewTerm` will be instantiated with `edge(_A,5)`.

Another predicate used to construct the clause being asserted is `queries_atom/2`. This is an auxiliary predicate defined in Draxler's `sqlcompiler.pl` that converts a query string, obtained from the `translate/3` predicate, in a Prolog atom.

Consider now that we call `edge(A,1)`. The `translate/3` predicate uses the `relation/3` and `attribute/4` facts to construct a specific query to match the call: '`SELECT source FROM edge_r WHERE dest=1;`'. Thus, when the `db_query/3` sends this query to the database, it will only retrieve the desired tuples.

Assume now that a new predicate, `direct_cycle/2`, is defined to call twice the `edge/2` predicate:

```
direct_cycle(A,B) :- edge(A,B), edge(B,A).
```

If we execute the generic call `cycle(A,B)`, `translate/3` will generate a '`SELECT * FROM edge_r`' query for the `edge(A,B)` goal, that will access all tuples sequentially. For the second goal, `edge(B,A)`, it will get the bindings of the first goal and generate a query of the form '`SELECT 800, 531 FROM edge_r WHERE source=800 AND`

dest=531'. These queries return 1 or 0 tuples, and are efficiently executed thanks to the MySQL index associated to the primary key of relation edge_r.

However, this approach has a substantial overhead of generating, running and storing a SQL query for each tuple of the first goal. To avoid this we can use the translate/3 predicate to transfer the joining process to the MySQL engine. To do so, we create views using the a db_view/3 predicate. The db_view/3 predicate is defined as follows:

```
db_view(ViewQuery,ViewTerm,ConnName):-
  get_value(ConnName,ConnHandler),

  % rename variables in terms
  rename_term_vars(ViewQuery,ClauseHead),
  rename_term_vars(ViewTerm,NewViewTerm),

  % construct the body literals of the clause to assert
  make_db_view_pred_body(ConnHandler,ClauseHead,NewViewTerm,ClauseBody),

  % assert the clause for the view being mapped
  assert(':-'(ClauseHead,ClauseBody)).
```

For example, if we call the following Prolog goal: db_view((edge(A,B), edge(B,A)), direct_cycle(A,B),my_conn), then the following clause will be asserted:

```
direct_cycle(A,B) :-
  rename_term_vars(direct_cycle(A,B),NewViewTerm),
  rename_term_vars((edge(A,B),edge(B,A)),NewViewQuery),
  translate(NewViewTerm,NewViewQuery,QueryString),
  queries_atom(QueryString,Query),
  db_query(conn_handler_id,Query,ResultSet),
  !,
  db_row(ResultSet,[A,B]).
```

Later when direct_cycle(A,B) is called, only a single SQL query will be generated: 'SELECT A.source,A.dest FROM edge_r A,edge_r B WHERE B.source=A.dest AND B.dest=A.source'.

These two approaches of executing conjunctions of database predicates are usually referred in the literature as *relation-level* (one query for each predicate as in the first direct_cycle/2 definition) and *view-level* (a unique query with all predicates as in the asserted direct_cycle/2 clause). A step forward will be to automatically detect, when consulting a Prolog file, the clauses that contain conjunctions of database predicates and use view level transformations, as in the example above, to generate more efficient code.

4.5 Manipulating the Database from Prolog

Manipulation of the MySQL database from Prolog is made possible throw specific defined predicates for determined actions. These predicates allow for a quick and easy way to establish communication to the MySQL database, by generating specific SQL statements adapted to particular tasks, such as describing a database table, selecting particular rows, inserting rows, etc.

The `db_describe/2` is a useful predicate that prints information about existing relations. This predicate is described next. The first argument defines the name of the relation, while the second argument defines the connection to be considered.

```
db_describe(RelName,ConnName):-
  get_value(ConnName,ConnHandler),

  % generate query 'describe <RelName>'
  name('describe ',L1),
  name(RelName,L2),
  append(L1,L2,L),

  % send query to the database and obtain result set
  db_query(ConnHandler,QueryDescribe,ResultSetDescribe),

  % retrieve the arity of the result set
  db_num_fields(ResultSetDescribe,Arity),

  % print head information
  write('-----------------------------------------------------------'),nl,
  write('|  Name  |    Type     | Null |  Key  |  Default  |  Extra  |'),nl,
  write('-----------------------------------------------------------'),nl,

  % print attribute info
  print_attribute_info(0,Arity,ResultSetDescribe),
  write('-----------------------------------------------------------'),nl.
```

Note that part of this implementation is similar to the `db_import/3` predicate. Here, the important predicate is `print_attribute_info/3`. This predicate uses the `db_row/2` predicate to obtain by backtracking the rows that describe the attributes of the relation being considered.

We next show the result of applying the `db_describe/2` predicate to the `edge_r` relation.

```
?- db_describe(edge_r,my_conn).
```

```
-------------------------------------------------------------
|  Name  |    Type     | Null |  Key  |  Default  |  Extra  |
-------------------------------------------------------------
| source | smallint(6) |      | PRI   | 0         |         |
| dest   | smallint(6) |      | PRI   | 0         |         |
-------------------------------------------------------------
```

Another example of obtaining information from the database is the `db_sql_select/3` predicate. Assuming that relation `edge_r` is mapped to predicate `edge/2`, then the `db_sql_select/3` predicate makes it possible to connect to the database using explicit SELECT queries. Predicate `db_sql_select/3` is defined as:

```
db_sql_select(Query,ArgList,ConnName):-
  get_value(ConnName,ConnHandler),
  db_query(ConnHandler,Query,ResultSet),
  db_row(ResultSet,ArgList).
```

For example, if we call `db_sql_select('select * from edge_r where source=1', [A,B],my_conn)`, we will obtain:

```
?- db_sql_select('select * from edge_r where source=1',[A,B],my_conn).

A = 1,
B = 2 ? ;

no
```

Other predicates allow to generate SQL statements to perform specific tuple-at-a-time actions. For example, the `db_insert/3` predicate dynamically constructs and asserts a Prolog clause to perform insertions in a specific database relation. This predicate is described next. The first argument is the database relation where insertions should be done, while the second argument is the Prolog predicate being mapped. The first argument can be declared with some of its arguments bound. The third argument defines, as usual, the connection to be considered.

```
db_insert(InsertQuery,InsertTerm,ConnName):-
  get_value(ConnName,ConnHandler),

  % rename variables in terms
  rename_term_vars(InsertQuery,ClauseHead),
  rename_term_vars(InsertTerm,NewInsertTerm),

  % generate query "INSERT INTO <RelName> VALUES (...)"
```

```
  ClauseHead=..ListClauseHead,
  arg(1,ListClauseHead,PredName),
  relation(PredName,RelName,_),
  name('INSERT INTO ',L1),
  name(RelName,L2),
  append(L1,L2,L3),
  name(' VALUES ',L4),
  append(L3,L4,InsertList),

  % construct the body literals of the clause to assert
  make_db_pred_body(ConnHandler,InsertList,ClauseHead,NewInsertTerm,
                                                        ClauseBody),

  % assert the clause for the insert query being mapped
  assert(':-'(ClauseHead,ClauseBody)).
```

For example, the call to `db_insert(edge_r(A,7),insert_edge(A),my_conn)` asserts the following predicate in Prolog:

```
insert_edge(A) :-
  % construct term with values '(A,7)'
  make_values(edge(A,7),ValuesTerm),

  % append values to the 'INSERT INTO edge_r VALUES 'query
  append([73,78,83,69,82,84,32,73,78,84,79,32,114,97,109,111,115,32,86,65,
          76,85,69,83,32],ValuesTerm,QueryString),

  % construct query and send it to the database for execution
  name(Query,QueryString),
  !,
  db_query(conn_handler_id,Query,_),
```

If then we call, for example, `insert_edge(5)`, the `db_query/3` in the `insert_edge/1` clause will be called with the SQL statement '`INSERT INTO edge_r VALUES (5,7)`', which will insert the pair of values (5,7) in the `edge_r` relation.

4.6 Handling Null Values

Usually, the tuples in a relation may contain `NULL` values in some fields. Thus, when we fetch tuples with `NULL` values, we need to efficiently represent them in Prolog. Intuitively, `NULL` values can be represented in Prolog by atoms. However, in a relational database, the `NULL` value in a tuple is interpreted as a non-existing value for that field. In Prolog, if a `NULL` value is represented as a unique atom, all `NULL` values are

alike. To avoid unification between NULL values, each NULL value has to be differently represented.

We next show how the c_db_row() function was extended to deal with NULL values. We used the same representation as proposed and implemented in the XSB Prolog system [60], where different NULL values are represented by terms of the form null(1), null(2), ..., null(N).

```
int c_db_row(void) {
  int null_id = 0;
  YAP_Term null_atom[1];
  ...
  if ((row = mysql_fetch_row(result_set)) != NULL) {
    for (i = 0; i < arity; i++) {
      if (row[i] == NULL) {                        // dealing with NULL values
        null_atom[0] = YAP_MkIntTerm(null_id++);
        t_field = YAP_MkApplTerm(
                    YAP_MkFunctor(YAP_LookupAtom("null"), 1), 1, null_atom);
      }
      ...
    }
    ...
  }
  ...
}
```

To insert/delete rows with NULL values, or use NULL values in queries, we use 'NULL' in the SQL statement passed to the database management system. For example, insert_edge('NULL') generates the query 'INSERT INTO edge_r VALUES (NULL,7)'.

4.7 Handling Deallocated Result Sets

The last two coupling approaches use backtracking to fetch tuples from the result sets. During user interaction, various connections can be established to the relational database management system. Each connection may have several result sets associated to it. As a result, if the result sets are not deallocated after use, memory requirement might be drastically consumed. To avoid this situation, result sets must be deallocated after use.

Result sets are ready for deallocation either when all the tuples are retrieved from the result set, or when a cut occurs before all the tuples from the result set were

retrieved. The first case is trivially handled because when the last tuple is retrieved, the result can be deallocated. This is what is done in the `c_db_row()` function (see Fig. 4.5 for details). The second case is more problematic, because when a cut occurs, we lose access to the result set. We thus need an auxiliary mechanism that allows us to deallocate pruned result sets.

The deallocation of pruned result sets can be solved using different approaches. One approach is to implement a predicate that forces the deallocation of pending result sets. This was the approach followed in this thesis. To implement this approach we defined two new predicates, `deallocate_result_sets/1` and `deallocate_all_result_sets/0`. The `deallocate_result_sets/1` predicate deallocates all the result sets for a particular connection given as argument. The `deallocate_all_result_sets/0` predicate deallocates all the result sets for all open connections.

To support the implementation of these predicates, we used a two-level list data structure. An outter-list stores the references to the open connections. Each open connection points then to an inner-list that stores the references to the open result sets associated with the connection. These lists were implemented in C as follows:

```
struct connection_list{
  MYSQL *conn;
  result_set_list *first;
  struct connection_list *next;
};

struct result_set_list{
  MYSQL_RES *res_set;
  struct result_set_list *next;
};
```

As an optimization, when we invoke the `db_close/1` predicate to close a connection, all open result sets associated with the connection are also automatically deallocated, so that their references do not get lost.

A more efficient approach is to protect database predicates from cut operations [63]. This approach can be implemented in two distinct ways:

The explicit implementation: where the user is responsible for the protection of the database predicate from potential cut operations.

The implicit implementation: where the system transparently executes a cut procedure when a cut operation occurs.

In the first approach, the user is responsible for releasing beforehand the result sets for the predicates that can be pruned. For example, if we have a predicate like:

```
db_goal(A):- db_pred(A), !, other_pred(A).
```

When the cut is performed, the result set for `db_pred(A)` can became left pending. To avoid this, the user must protect the `db_pred(A)` predicate from the cut by adding mark predicates as follows:

```
db_goal(A):- db_mark, db_pred(A), db_cut, !, other_pred(A).
```

The `db_mark/0` predicate simply delimits the range of database predicates that the `db_cut/0` predicate must deallocate. In this approach, the user has to include explicit annotations to control the range of predicates to protect.

The second approach is based on an extension of the `YAP_UserBackCPredicate` declaration. An extra argument is used to declare the function that should be called when a cut operation prunes over the Prolog predicate being declared. This extra function can thus be used to protect result sets from became left pending, by deallocating them when it is called. To take advantage of this approach, the user only needs to declare and implement the extra function. From the user's point of view, dealing with standard predicates or relationally defined predicates is then equivalent. For a more detailed description of these approaches please refer to [63].

4.8 Chapter Summary

In this chapter, we introduced and explained the details of implementation for the three alternative approaches that we propose for coupling logic programming with relational databases. Alongside, we discussed how to access, use and update database information from Prolog; how to handle `NULL` values; and how to handle deallocated result sets.

Chapter 5

Performance Evaluation

The three distinct approaches of coupling Yap and MySQL described in the previous chapter are evaluated to estimate their performance. The execution time of certain goals by each approach is compared. View-level and relational-level accesses, along with optional indexing, are used to improve performance. Different indexing schemes are used both in the database system and in the Prolog system. In order to evaluate the performance of the three distinct approaches, we used Yap 4.4.4 and MySQL server 4.1.11 installed on the same machine, an AMD Athlon 1400 with 512 Mbytes of RAM. Two different benchmarks, `edge_r` and `query`, are used to illustrate the impact of view-level transformations and indexing over database predicates.

5.1 The `edge_r` Benchmark

The `edge_r` relation was created using the following SQL declaration:

```
CREATE TABLE edge_r (
   source  SMALLINT NOT NULL,
   dest    SMALLINT NOT NULL,
   PRIMARY KEY (source,dest));
```

Two queries over the `edge_r` relation were used. The first query was to find all the solutions for the `edge(A,B)` goal, which correspond to all the tuples of relation `edge_r`. The second query was to find all the solutions of the `edge(A,B),edge(B,A)` goal, which correspond to all the direct cycles. Execution time was measured using the `walltime`

parameter of the statistics built-in predicate, in order to correctly measure the time spent in the Yap process and in the MySQL process. In the tables that follow, timing results are always presented in seconds.

Table 5.1 presents the results for the different coupling approaches, with relation `edge_r` having 1,000 vertices and populated with 5,000, 10,000 and 50,000 random tuples.

Coupling Approach/Query	**Tuples**		
	5,000	**10,000**	**50,000**
Asserting Approach			
assert time	0.05	0.30	2.06
`edge(A,B)`	< 0.01	< 0.01	0.02
`edge(A,B),edge(B,A)`	7.17	30.10	753.80
Backtracking Approach			
`edge(A,B)` (*store_result*)	0.02	0.04	0.19
`edge(A,B)` (*use_result*)	0.02	0.04	0.19
`edge(A,B),edge(B,A)` (*store_result*)	91.23	359.40	9,410.7
`edge(A,B),edge(B,A)` (*use_result*)	n.a.	n.a.	n.a.
Backtracking + SQL Unify Approach			
`edge(A,B)` (*store_result*)	0.02	0.04	0.19
`edge(A,B),edge(B,A)` (*relation-level*)	0.98	2.20	19.70
`edge(A,B),edge(B,A)` (*view-level*)	0.02	0.04	0.28

Table 5.1: Execution times of the different approaches

During the asserting approach the two queries and the assert time of the involved tuples mentioned above were measured. This assert time is relevant because the other approaches of coupling do not have this overhead time. Despite involving multiple context switching between Prolog and C, the assert time is fast and can be used with large relations. Using the method described, asserting 50,000 tuples takes about 2 seconds. For comparison, if the relation was dumped into a file in the form of Prolog facts and consulted, Yap would take about 0.6 seconds, which is around 3 times faster. The `edge(A,B)` query involves sequentially accessing all the facts that have been asserted. This is done almost instantly, taking only 0.02 seconds for 50,000 facts. This approach shows larger difficulties while using the query `edge(A,B),edge(B,A)`. Even for 5,000 facts Yap already takes more than 7 seconds

and the growth is exponential, taking several minutes for 50,000 facts. This is due to the fact that Yap does not index dynamic predicates, by default, such as the asserted edge facts. For each `edge(A,B)`, Prolog execution mechanism has to access *all* the facts to see if they unify with `edge(B,A)`, because the absence of indexing cannot use the first goal variable bindings to limit the search space. Finally, this asserting approach reduces the overhead of communication with the MySQL server to the initial assert, and the solution of the queries has no communication with the MySQL server.

To evaluated the second approach `mysql_store_result()` and `mysql_use_result()` procedures were both used. No relevant differences were detected, mainly because the Yap and MySQL server were running on the same machine. The `mysql_use_result()` could not be used with the `edge(A,B),edge(B,A)` query because this version of MySQL does not allow multiple results sets on the server side (this should be possible with the latest version of MySQL, 5.0). Query `edge(A,B)` takes 0.19 seconds to return the 50,000 solutions. The overhead of communicating with the MySQL result set structure tuple by tuple causes a slowdown of around 10 times as compared to the previous asserting strategy. This 10 times factor is also reflected on the execution time of `edge(A,B),edge(B,A)`. For both edge goals, a '`SELECT * FROM edge_r`' query is generated and the join is computed by Yap using the two MySQL result sets. We should note that, on this approach, there are no indices on Yap that can be used to speed-up the query, as the `edge_r` tuples only exist in MySQL structures. Also, the difficulties explained for the asserting approach remain, as the indices existing on the MySQL server for the `edge_r` relation are of no use since the queries are '`SELECT * FROM edge_r`'.

The last approach of coupling, which tries to transfer unification to the SQL engine, gave exactly the same results for query `edge(A,B)`, as the query generated by `translate/3` is exactly the same of the previous approach ('`SELECT * FROM edge_r`'). Regarding query `edge(A,B),edge(B,A)` there are very significant differences. For this query we consider a *relation-level* access where `translate/3` is used for each goal, and a *view-level* access where `translate/3` is used to generate a SQL query which computes the join of the two goals.

For the relation-level access the speed-up obtained over the backtracking approach is of around 100 times for 5,000 tuples and, more important, allows the increase in execution time to become linear in the number of tuples. Note that the execution times

of this approach are not quicker because there is a large overhead of communication with MySQL, involving running one query and storing the result on the client for each tuple of the first goal. For view-level access `translate/3` generates a single query and Yap just sequentially accesses the returned result set. For $50,000$ tuples the execution time is of 0.28 seconds. This represents a speed-up of more than $2,500$ times over the asserting approach and of more than $30,000$ times over the backtracking approach.

Index performance is fundamental to interpret the results obtained. Note that the asserting approach relies on the logic system indexing capabilities, while the other approaches rely on the database system indexing capabilities. The asserting approach can be improved if indexing can be performed over the dynamic predicate asserted. Yap can index on the first argument of dynamic predicates if we declare the update semantics of dynamic predicates to be *logical* instead of the default *immediate* (`dynamic_predicate(edge/2,logical)`). Dynamic predicates with logical update semantics achieve similar performance when compared with static compiled predicates.

Relational database management systems have extended indexing capabilities as compared to Prolog systems. Every relational database system allows the declaration of several types of indices on different attributes. To evaluate the impact of changing the indexing approach on MySQL, the primary key index of relation `edge_r` was dropped: '`ALTER TABLE edge_r DROP PRIMARY KEY`'. The performance using a secondary index just on the first attribute of `edge_r` was also evaluated: '`ALTER TABLE edge_r ADD INDEX ind_source (source)`' (this is identical to the traditional indexing approach of Prolog systems). Table 5.2 compares the performance of these different indexing schemes. It presents the execution times for $50,000$, $100,000$ and $500,000$ tuples (for $500,000$ tuples we used $5,000$ vertices) using asserting and backtracking with SQL unification in view-level for query `edge(A,B),edge(B,A)`.

By observing the table, the dramatic impact of indexing when compared with no indexing is clear. An interesting comparison is the time taken with the asserting approach by Yap without indexing (753.80 and 5270.27 seconds for $50,000$ and $100,000$ tuples), and the time taken by MySQL also without indexing (487.35 and 1997.36 seconds for $50,000$ and $100,000$ tuples). Yap is about 1.5 to 2.5 times slower than MySQL dealing with no indexed data. Another interesting comparison is the time that Yap and MySQL using an equivalent index on the same argument take. They show almost the same performance, with a small overhead for Yap in this particular

Coupling Approach/Indexing Scheme	Tuples		
	50,000	**100,000**	**500,000**
Asserting Approach			
no index	753.80	5270.27	> 2 hours
index on first argument (source)	0.59	2.40	12.88
Backtracking + SQL Unify Approach			
no index	487.35	1997.36	> 2 hours
secondary index on (source)	0.54	1.93	10.25
primary key index on (source,dest)	0.28	0.67	3.81

Table 5.2: Index performance for query `edge(A,B),edge(B,A)`

query. As expected, best results are obtained for MySQL when using a primary key index on both attributes (0.28, 0.67 and 3.81 seconds for 50,000, 100,000 and 500,000 tuples).

5.2 The *query* Benchmark

The queries on the `edge/2` predicate presented on the previous section show very significant differences between relation-level accesses and view-level accesses. In this section we further study the impact of performing view-level transformations on database predicates using now the *query* benchmark program [75].

The *query* benchmark program, written by D. H. Warren, is a Prolog program which tries to find countries of approximately equal population density, based on a database listing populations and areas for several countries. Typically, the population and area knowledge is represented as Prolog facts, `pop/2` and `area/2`. In the context of a deductive database system such knowledge can instead be represented by relational tuples, that can be stored in two relations as shown in Fig. 5.1.

We import the tuples in these relations using the `db_import/3`, associating the `pop` relation with predicate `pop/2` and the `area` relation with predicate `area/2`:

```
:- db_import(pop,pop,my_conn).
:- db_import(area,area,my_conn).
```

Besides the `pop/2` and `area/2` predicates, the *query* benchmark program has the

```
pop relation
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| country    | varchar(20) |      | PRI |         |       |
| population | smallint(6) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+

area relation
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| country | varchar(20) |      | PRI |         |       |
| area    | smallint(6) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

Figure 5.1: Relations `pop` and `area`

following two predicates, `query/1` and `density/2`:

```
query([C1,D1,C2,D2]) :-
    density(C1,D1),
    density(C2,D2),
    D1 > D2,
    T1 is 100*D1,
    T2 is 101*D2,
    T1 < T2.

density(C,D) :-
    pop(C,P),
    area(C,A),
    D is (P*100)//A.
```

In the original *query* program, population is given in hundreds of thousand and area is given in thousands of square miles. The `density/2` predicate calculates the number of inhabitants per square mile. The `query/1` predicate selects pairs of countries in the database where the density varies less than 5%. Facts for 25 different countries were given. We slightly modified this program, increasing the number of countries to 232, using square kilometers instead of square miles and choosing countries with a variation of 1% instead of 5%. Figure 5.2 shows all the solutions for the query goal `query(L)`.

In Table 5.3 we compare the execution time of this query using our different coupling approaches. As previously, times are given in seconds.

For the asserting approach, we measured the assert time for the tuples in relations `pop` and `area`. As these amount to 464 facts, this assert time is very small, taking less

```
[Albania,125,Thailand,124]
[Armenia,113,Slovakia,112]
[Belgium,340,Japan,337]
[Burundi,222,Trinidad and Tobago,220]
[China,134,Moldova,133]
[Denmark,123,Indonesia,122]
[El Salvador,304,Sri Lanka,303]
[France,109,Uganda,108]
[Kuwait,123,Indonesia,122]
[Nepal,188,Pakistan,187]
[Pakistan,187,Korea (North),186]
[Poland,123,Indonesia,122]
[Portugal,109,Uganda,108]
[Thailand,124,Denmark,123]
[Thailand,124,Kuwait,123]
[Thailand,124,Poland,123]
[Uganda,108,Hungary,107]
```

Figure 5.2: All solutions for the query goal `query(L)`

than 0.01 seconds. As mentioned previously, dynamic facts are not indexed by default in Yap. Thus, finding all solutions for query `query(L)` takes 1.51 seconds without indexing. Altering the definition of the dynamic predicates `pop/2` and `area/2` with:

```
:- dynamic_predicate(pop/2,logical).
:- dynamic_predicate(area/2,logical).
```

makes indexing available on the first (`country`) argument, and largely speeds-up execution, taking now 0.09 seconds.

Our second approach accesses tuples from the database relations, tuple at a time, based on totally generic queries. As these queries are generic there is no gain from using indices. The overhead of communicating with MySQL result set for each tuple makes the execution time for finding all solutions for query `query(L)` increase to 170.03 seconds, which represents a slow-down factor of more than 112 times over the asserting approach also without indexing.

Our third approach uses dynamic SQL queries generation based on the instantiation of the goal's arguments. Indexing is useful for these queries and the declaration of relations `pop` and `area` included the creation of a primary index on attribute `country`. Using relation-level access, separate queries are generated for goals `pop(C,P)` and `area(C,A)`. In this scenario, execution time of the query program is 50.66. Though this represents an important speed-up over the previous 170.03 value, due to the use of

Coupling Approach/Query	Execution Time
Asserting Approach	
assert time	< 0.01
`query(L)` *(no indexing)*	1.51
`query(L)` *(indexing on first argument)*	0.09
Backtracking Approach	
`query(L)`	170.03
Backtracking + SQL Unify Approach	
`query(L)` *(relation-level)*	50.66
`query(L)` *(view-level 1)*	2.16
`query(L)` *(view-level 2)*	1.89
`query(L)` *(view-level 3)*	0.19
`query(L)` *(view-level 3) (indexing)*	0.11

Table 5.3: Execution times of `query(L)` on the different approaches

the index on the relation `country` and the instantiation of variable `C` for the `area(C,A)` goal, there is still a very large gap compared to the asserting approach.

Improving performance requires the use of view-level access, as discussed previously. This `query` program offers a number of opportunities for view-level transformations. The most obvious is joining together queries for `pop(C,P)` and `area(C,A)` goals, by declaring a view as follows:

```
:- db_view((pop(C,P),area(C,A)),pop_area(C,P,A),my_conn).
```

We call this view-level transformation *view-level 1* in Table 5.3. Execution time is now reduced to 2.16 seconds, using this simple transformation. This factor of more than 23 times over the relation-level approach clearly shows the relevance of view-level transformations.

We can still improve database access by further using view-level transformations. We can create a view for all the `density/2` predicate were the arithmetic calculation of density is performed by MySQL:

```
:- db_view((pop(C,P),area(C,A), D is P*100//A),density(C,D),my_conn).
```

We call this view-level transformation *view-level 2* and execution time is now reduced to 1.89 seconds.

A final view-level transformation can be done for the `query/1` predicate, joining together both `density/2` goals, along with the arithmetic operations and conditions:

```
:- db_view((pop(C,P),area(C,A), D is P*100//A),density(C,D),my_conn).
:- db_view((density(C1,D1),density(C2,D2),D1>D2,T1 is 100*D1,
   T2 is 101*D2,T1<T2),(query([C1,D1,C2,D2]),my_conn).
```

We call this view-level transformation *view-level 3* and execution time is now reduced to 0.19 seconds! The SQL query generated for this view is the following:

```
SELECT A.country ,A.population*100 div B.area as D1, C.country,
C.population*100 div D.area as D2 FROM pop A , area B, pop C, area D
WHERE B.country = A.country and D.country=C.country and
A.population*100 div B.area *20<C.population*100 div D.area *21
having D1>D2
```

Finally, we can use the indexing capabilities of MySQL and create secondary indices on attributes `population` and `area` of relations `pop` and `area`, respectively. This further reduces execution time to 0.11 seconds, which is very similar to the time taken by the asserting approach using indexing.

This *query* program shows clearly the importance of performing view-level transformation to improve database access. An automatic generator of these views, along with the creation of the relevant indices in MySQL is fundamental for incorporation with a coupling interface between a logic system and a relational database system, and we plan to develop it as future work.

5.3 Chapter Summary

The impact of using alternative approaches for coupling Yap Prolog with MySQL was studied and evaluated. Through experimentation, the possibility to couple logic systems with relational databases using approaches based on tuple-at-a-time communication schemes was observed. The results show however that, in order to be efficient, view-level transformations when accessing the database needs to be explored. Results also show that indexing is fundamental to achieve scalability. Indexing is important on the database server for view level access and on the Prolog system when tuples are asserted as facts.

For Yap, further evaluation should experiment with the current development version of this system, Yap 4.5, where indexing has been improved and can build indices using more than just the first argument.

Chapter 6

Conclusions

This final chapter summarizes the work developed in this thesis. First, we enumerate the main contributions of the thesis, next we suggest some relevant topics for further work, and then we conclude with a final remark.

6.1 Main Contributions

A major guideline for our work was to join the efficiency and safety of databases in dealing with large amounts of data with the higher expressive power and inference abilities of logic systems. In this regard, we have studied, implemented and evaluated the impact of using three distinct approaches for coupling the Yap Prolog system with the MySQL relational database system. The main contributions of this work are:

Coupling approaches: we have considered three main approaches for coupling the Yap Prolog system with the MySQL relational database system: asserting database tuples as Prolog facts; accessing database tuples by backtracking; and transferring unification to the database. The most relevant features of each approach are then briefly described.

Asserting database tuples as Prolog facts: this approach maps database relations into Prolog predicates by asserting the complete set of tuples of a database relation as Prolog facts. This minimizes the number of communications with the database because only a single connection is

required. On the other hand, the entire set of tuples is duplicated on the Prolog side, and real time modification done by others to the database are not visible to the Prolog system.

Accessing database tuples by backtracking: this approach uses the Prolog backtracking mechanism to access the database tuples one-at-a-time. This minimizes memory requirements because the tuples are only concentrated in a single repository, and allows users to see real time modifications done by others. However, for calls with not all arguments unbound, non-matching tuples are still fetched from the database, and it is then the Prolog unification mechanism that selects the matching tuples.

Transferring unification to the database: specific SQL queries are dynamically constructed to transfer the Prolog unification process to the MySQL engine. This prevents unnecessary tuples from being fetched from the database, thus minimizing memory requirements and optimizing computation. Moreover, conjunctions of database predicates can be further optimized if we use view level transformations to define single queries to compute the conjunctions.

Performance evaluation: we have carried out a detailed study to evaluate the performance of our proposals. All approaches were tested against different number of tuples in sets we believe are representative of existing applications. The most relevant conclusion of our results is that view-level transformations and indexing are fundamental to achieve scalability.

- Through experimentation, we observed that it is possible to couple logic systems with relational databases using approaches based on tuple-at-a-time communication schemes. However, in order to be efficient, we need to explore view-level transformations when accessing the database. Our results clearly showed the importance of performing view-level transformations to improve database access.

- Our results also showed that indexing is fundamental to achieve scalability. Indexing is important on the database server for view-level access and on the Prolog system when tuples are asserted as facts.

6.2 Further Work

The coupling of the Yap Prolog system with the MySQL relational database system was implemented on three different levels of communication, demonstrating that together, the two systems, can result in an efficient, powerful and insightful tool. However, in order to overcome some limitations of the combined system, more work still remains to be done:

- The current implementation needs to be tested with a wider range of applications. Many opportunities for refining the system exist, and more will certainly be found with an intensive experimentation of the system. Further evaluation should also experiment with the current development version of Yap, version 4.5, where indexing has been improved and can build indices using more than just the first argument.

- An important drawback of our system is the support to the transparent use of the cut operation over database predicates. This operation is extremely used in Prolog programs, both for efficiency and semantic preservation. However, its use after a database defined predicate can cause a lack of cursors and, more important, a lack of memory due to a number of very large non-deallocated data structures. To overcome these limitation, recently, Soares *et al.* [63] proposed an extension to the Prolog engine that allows to define procedures to be executed when a predicate is pruned by a cut, which for database predicates can thus be used to deallocate the associated result sets.

- We see that the importance of performing view-level transformations is fundamental to improve database access. An interesting feature of the system will be to extend it to incorporate a mechanism to dynamically detect conjunctions of database predicate and automatically generate the appropriate view-level transformations.

- Another interesting area for further research is to take advantage of the advanced features of the OPTYap system [56], such as tabling and or-parallelism, namely in the evaluation of recursive and concurrent queries.

6.3 Final Remark

Through this research we aimed at demonstrating the benefits of implementing coupled systems which combine the simplicity, portability and performance of both the Yap Prolog programming language with the MySQL database management system. The results obtained by making use of view-level access and indexing, reinforce the viability of such strategies. We hope that the work developed in this thesis will serve as an inspiration to others and be a resource for further improvements and research in this area.

Much work remains to be accomplished, along with the insight of loosely coupled systems, where the benefits of each system are preserved and assured in future developments and improved editions of each separate system.

References

[1] Serge Abiteboul and Stéphane Grumbach. A rule-based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 1991.

[2] Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. Efficient bottom-UP computation of queries on stratified databases. *Journal of Logic Programming*, 11(3–4):295–344, 1991.

[3] Isaac Balbin and Kotagiri Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3):259–262, 1987.

[4] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–16, 1986.

[5] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1987.

[6] M. Brodie and M. Jarke. On Integrating Logic Programming and Databases. In *Expert Database Workshop*, pages 191–207. Benjamin Cummings, 1984.

[7] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proceedings of the International Conference on Extending Database Technology (EDBT '88)*, volume 303 of *LNCS*, pages 488–505. Springer, 1988.

[8] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), 1997.

[9] Filippo Cacace, Stefano Ceri, Stefano Crespi-Reghizzi, Letizia Tanca, and Roberto Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *SIGMOD Conference*, pages 225–236, 1990.

[10] M. Carlsson. *Sicstus Prolog User's Manual*, February 1988.

[11] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[12] A. Colmerauer, H. Kahoui, and M. van Caneghem. Un systéme de communication homme-machine en français. Technical report, Gruppe Intelligence Artificielle, Université Aix-Marseille II, 1973.

[13] L. Damas, R. Reis, , R.Azevedo, and V. Costa. YAP user's manual, 2003.

[14] S. K. Das and J. Dicker. Coupling oracle with eclipse.

[15] Subrata Kumar Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.

[16] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 1995.

[17] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. Design and implementation of the glue-nail database system. In *SIGMOD Conference*, pages 147–156, 1993.

[18] Christoph Draxler. *Accessing Relational and Higher Databases through Database Set Predicates in Logic Programming Languages*. PhD thesis, Zurich University, Department of Computer Science, 1991.

[19] Paul DuBois. *MySQL*. New Riders Publishing, Carmel, IN, USA, 1999.

[20] M. Ferreira, R. Rocha, and S. Silva. Comparing Alternative Approaches for Coupling Logic Programming with Relational Databases. In *Proceedings of the Colloquium on Implementation of Constraint and Logic Programming Systems, CICLOPS'2004*, pages 71–82, 2004.

[21] M. Freeston. The BANG file: A new kind of grid file. In *Proceedings of the ACM SIGMOD Annual Conference*, pages 260–269. ACM Press, 1987.

[22] Burkhard Freitag, Heribert Schutz, and Gunther Specht. Lola - a logic language for deductive databases and its implementation. In *In Proceedings 2nd International Symposium on Database Systems for Advanced Applications*, pages 216 – 225, 1991.

[23] H. Gallaire and J. Minker, editors. *Logic and Databases.* Plenum, 1978.

[24] A. Van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[25] Allen Van Gelder, John S. Schlipf, and Kenneth A. Ross. The well-founded semantics for general logic programs, 1991.

[26] P. M. D. Gray and R. J. Lucas. *Prolog and Databases: Implementations and New Directions.* John Wiley and Sons Publishers, 1988.

[27] C. C. Green and B. Raphael. The use of theorem-proving techniques in question-answering systems. *Proceedings 23rd National Conference ACM*, 1968.

[28] Jiawei Han, Ling Liu, and Zhaohui Xie. Logicbase: A deductive database system prototype. In *CIKM*, pages 226–233, 1994.

[29] Lawrence J. Henschen and Shamim A. Naqvi. On compiling queries in recursive first-order databases. *Journal of the ACM*, 31(1):47–85, 1984.

[30] T. Irving. *A generalized interface between PROLOG and relational databases.* In PROLOG and Databases: Implementations and New Directions, Chichister, West Sussex: Ellis Horwood Limited, 1988.

[31] Matthias Jarke, Jim Clifford, and Yannis Vassiliou. Optimizing Prolog front-end to a relational query system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 14(2):296–306, 1984.

[32] Manfred Jeusfeld and Martin Staudt. Query Optimization in Deductive Object Bases. Technical Report AIB-26-1991, RWTH Aachen, 1993.

[33] David B. Kemp, Kotagiri Ramamohanarao, Isaac Balbin, and Krishnamurthy Meenakshi. Propagating constraints in recusive deduction databases. In *NACLP*, pages 981–998, 1989.

[34] David B. Kemp, Kotagiri Ramamohanarao, and Zoltan Somogyi. Right-, left- and multi-linear rule transformations that maintain context information. In *VLDB*, pages 380–391, 1990.

[35] Werner Kießling, Helmut Schmidt, Werner Strauß, and Gerhard Dünzinger. DECLARE and SDS: Early efforts to commercialize deductive database technology. *The VLDB Journal*, 3(2):211–243, 1993.

[36] R. Kowalski. Predicate Logic as a Programming Language. In *Proc. IFIP Conference*, pages 556–574. North-Holland, 1974.

[37] J. W. Lloyd. *Foundations of Logic Programming, Second Extended Edition*. Springer Verlag, 1993.

[38] Donald W. Loveland. *Automated Theorem Proving: a Logical Basis*. North-Holland, 1 edition, 1978.

[39] R. J. Lucas and keylink Computers Ltd. Prodata interface manual, 1997.

[40] Michael J. Maher and Raghu Ramakrishnan. Déjà vu in fixpoints of logic programs. In *NACLP*, pages 963–980, 1989.

[41] F. Maier, D. Nute, W. Potter, J. Wang, M. J. Twery, H. M. Rauscher, P. Knopp, S. Thomasma, M. Dass, and H. Uchiyama. PROLOG/RDBMS Integration in the NED Intelligent Information System. In *Confederated International Conferences DOA, CoopIS and ODBASE*, volume 2519 of *LNCS*, page 528. Springer-Verlag, 2002.

[42] Alberto O. Mendelzon and Mariano P. Consens. Hy+: A hygraph-based query and visualization system. Technical report, 1993.

[43] J. Minker. Perspectives in deductive databases (Abstract only). In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 135–136. ACM Press, 1987.

[44] J. Minker and J. M. Nicolas. On recursive axioms in deductive databases. *Information Systems*, 8(1):1–13, 1982.

[45] Guido Moerkotte and Peter C. Lockemann. Reactive consistency control in deductive databases. *ACM Trans. Database Syst.*, 16(4):670–702, 1991.

[46] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *SIGMOD Conference*, pages 247–258, 1990.

[47] Ben Napheys and Don Herkimer. A look at loosely-coupled prolog database systems. In *Expert Database Conference*, pages 257–271, 1988.

[48] J. F. Naughton. One-sided recursions. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 340–348. ACM Press, 1987.

[49] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Argument reduction by factoring. *Theoretical Computer Science*, 146(1–2):269–310, 1995.

[50] Jeffrey F. Naughton and Raghu Ramakrishnan. How to forget the past without repeating it. In *16th International Conference on Very Large Data Bases*, pages 278–289. Morgan Kaufmann Publishers, 1990.

[51] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *VLDB*, pages 359–371, 1990.

[52] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *JICSLP*, pages 273–287, 1992.

[53] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Coral - control, relations and logic. In *VLDB*, pages 238–250, 1992.

[54] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *J. Log. Program.*, 23(2):125–149, 1995.

[55] J. A. Robinson. A machine-oriented logic based on resolution principle. *Journal of the ACM*, 12(1):23–49, 1965.

[56] R. Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs*. PhD thesis, Department of Computer Science, University of Porto, 2001.

[57] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, volume 51(1) of *Journal of Computer and Systems Sciences*, pages 161–171. ACM Press, 1990.

[58] D. Sacca and C. Zaniolo. Magic counting methods. In *Proceedings of the ACM SIGMOD Annual Conference*, pages 49–59. ACM Press, 1987.

[59] Domenico Sacca and Carlo Zaniolo. The generalized counting method for recursive logic queries. *Theoretical Computer Science*, 62(1-2):187–220, 1988.

[60] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

[61] E. Sciore and D. S. Warren. Towards an integrated database-prolog system. In L. Kerschberg, editor, *Expert Database Systems*, page 293. Benjamin/Cummings, 1986.

[62] SICS. *Quintus Prolog Release 3.4*. Swedish Institute of Computer Science, 1999.

[63] T. Soares, R. Rocha, and M. Ferreira. Pruning Extensional Predicates in Deductive Databases. In *Proceedings of the Colloquium on Implementation of Constraint and LOgic Programming Systems, CICLOPS'2005*, 2005.

[64] Divesh Srivastava and Raghu Ramakrishnan. Pushing constraint selections. *Journal of Logic Programming*, 16(3-4):361–414, 1993.

[65] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *VLDB*, pages 501–511, 1991.

[66] S. Sudarshan and Raghu Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In *ILPS*, pages 557–574. The MIT Press, 1993.

[67] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data Language. In *International Conference on Very Large Data Bases*, pages 33–41. Morgan Kaufmann, 1986.

[68] Shalom Tsur and Carlo Zaniolo. LDL: A logic-based data language. In Yahiko Kambayashi, Wesley Chu, Georges Gardarin, and Setsuo Ohsuga, editors, *Twelfth international conference on very large data bases, proceedings (VLDB '86)*, pages 33–41. Morgan Kaufmann Publishers, 1986.

[69] Jeffrey D. Ullman. *Principles Of Database and Knowledge-Base Systems. Vol. 1. Rockville*. Rockville, MA: Computer Science Press, Inc., 1989.

[70] Mary-Claire van Leunen. *A Handbook for Scholars*. Knopf, 1979.

[71] M. vanEmden and R. Kowalski. The Semantics of Predicate Logics as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

[72] Raf Venken and Anne Mulkers. The interaction between bim-prolog and relational databases. In *Prolog and Databases*, pages 95–107. 1988.

[73] L. Vieille. A database-complete proof procedure based on SLD-resolution. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 74–103. MIT Press, 1987.

[74] Laurent Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, pages 253–267, 1986.

[75] D. H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[76] C. Zaniolo. Prolog: A database query language for all seasons. In L. Kerschberg, editor, *Expert Database Systems*, page 219. Benjamin/Cummings, 1986.