Pedro Miguel Pereira Mota da Costa

Relational Storage Mechanisms for Tabled Logic Programs

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2007

Pedro Miguel Pereira Mota da Costa

Relational Storage Mechanisms for Tabled Logic Programs

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

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2007

Dedicated to my beloved grandmother Aurora

Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisors, Ricardo Rocha and Michel Ferreira, for helping me to carry out the present work. Their constant support, encouragement and guidance were fundamental during the entire process of research, implementation, testing and presentation of results. I also thank them for the revisions, comments and suggestions regarding the improvement of this thesis.

I am very grateful to Prof. Altamiro da Costa Pereira for his kind contribution, allowing me the time to proceed with my work. To Ernesto Palhares and Telmo Fonseca, I thank their friendship and support. To all the staff from *Serviço de Bioestatística e Informática Médica*, I thank their fellowship through out these past few years.

To my beloved mother and father, Ana and Alberto, my brother and sister, José and Ana Maria, I would like to express my deepest gratitude and love. You made all of this possible from the very beginning. To my wife, Catarina, I thank for making a better person out of me with all of her affection, patience, encouragement and love. I love you very much.

At last, I would like to thank all of those relatives, friends and teachers who, throughout my life, have helped me to become what I am. Thank you all.

July 2007

Abstract

Logic programming languages, such as Prolog, provide a high-level, declarative approach to programming. Derived from the Horn Clause Logic, this paradigm is based on a simple theorem prover that given a set of assumptions and rules, searches for alternative ways to satisfy queries.

Although a powerful, flexible and well performing tool, a major effort has been made in the past years to increase Prolog's declarativeness and expressiveness. The potential of logic programming has been limited since Prolog's standard evaluation strategy – SLD resolution – is prone to infinite loops and redundant subcomputations. Of all the several proposals that have come forth to overcome this situation, one in particular, known as *tabling* or *memoing*, proved to be particularly effective, albeit when are used for applications that deal with huge answer sets, the risk of memory exhaustion is very real. In general, in order to recover some space, programmers have no choice but to arbitrarily select some of the tables for deletion.

With this research, we intend to demonstrate that an alternative approach is possible. Rather than deleting predicate tables, we propose the storage of tabled data in an external relational database system, from where answer subsets may be swiftly recovered whenever subsequent table calls occur, hence avoiding re-computation. To validate our approach, we propose DBTab, an extension of the YapTab tabling system providing engine support for data transactions between the YAP logical engine and the MySQL relational database management system.

The attained results show that DBTab may become an interesting approach when the cost of recalculating *tabling tries* exceeds the amount of time required to fetch the entire answer-set from the database. The results reinforced our belief that tabling can contribute to expand the range of applications for Logic Programming.

Resumo

As linguagens de programação em lógica, nomeadamente o Prolog, constituem uma abordagem declarativa e de alto nível à programação. Este paradigma, baseado na lógica de cláusulas de Horn, consiste num demonstrador de teoremas que, munido de um conjunto de assunções e regras lógicas, procura caminho alternativos para resolver as questões que lhe são propostas.

Apesar do poder, flexibilidade e bom desempenho dos motores de inferência actuais, os últimos anos têm sido férteis em propostas destinadas a melhorar o seu poder declarativo e expressivo. O real potencial da linguagem tem sido limitado pela susceptibilidade da resolução SLD a ciclos infinitos e subcomputações redundantes. Das propostas de resolução avançadas, a mais eficaz - denominada tabulação - é predisposta à exaustão de memória quando utilizada em aplicações com grandes conjuntos de respostas. A solução mais comum para este problema é fornecer aos programadores um conjunto de directivas que permitam eliminar arbitrariamente algumas das tabelas.

Com este trabalho, pretendemos demonstrar a existência de alternativas viáveis. A nossa proposta consiste em armazenar as tabelas a descartar para uma base de dados relacional, de onde podem ser fácil e rapidamente recolhidas sempre que uma chamada subsequente ocorre, evitando assim repetir toda a subcomputação. A proposta é validada por um extensão ao componente de tabulação sequencial YapTab, denominada DBTab, que adiciona ao motor de inferência YAP suporte para transacções relacionais com o sistema de gestão de base de dados MySQL.

Os resultados obtidos demonstram que a extensão DBTab se torna interessante quando o custo da recomputação excede o custo de recolher as respostas pré-calculadas das tabelas relacionais. Estes resultados reforçam a nossa convicção de que a tabulação pode alargar o âmbito de aplicabilidade da programação em lógica.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Logic programming is known to be a powerful, high-level, declarative approach to programming. This paradigm is based on the idea that programming languages should provide a precise, yet simple, formalism to express one's beliefs, assumptions and goals. Ultimately, computer programs should be a set of instructions that humans could easily understand and produce, rather than a set of highly-dependent machine-oriented operations.

Arguably, Prolog is the most popular and powerful logic programming language. Throughout the language's history, its potential has been demonstrated in research areas such as Artificial Intelligence, Natural Language Processing, Knowledge Based Systems, Machine Learning, Database Management, or Expert Systems. Prolog's popularity greatly increased since the introduction, in 1983, of the *Warren Abstract Machine* (*WAM*) [War83], a sequential execution model that proved to be highly efficient, increasing Prolog systems' performance and setting it close to that of equivalent C programs [Roy90].

Prolog programs are written as sets of Horn clauses, a subset of First-Order Logic that has an intuitive interpretation as positive facts and rules. Questions, or *queries*, are answered by executing a resolution procedure against those facts and rules. The operational semantics of Prolog is given by SLD resolution [Llo87], a refutation strategy particularly effective for stack based machines, albeit its limitations impose practical semantic concerns to programmers throughout software development. For instance, it is in fact quite possible that logically correct programs result in infinite execution

loops.

Aiming at the improvement of Prolog's expressive power, several proposals have been put forth to overcome SLD limitations. A specially popular one, denominated *tabling* [Mic68] (or *tabulation* or *memoing*), consists of storing intermediate answers for subgoals, allowing their reutilization whenever a repeated subgoal appears during the resolution process. Tabling based models are known to reduce the search space, avoid infinite looping and have better termination properties than standard SLD based models; in fact, it has been proven that termination can be guaranteed for all programs with the *bounded term-size property* [CW96].

Chen's work on SLG resolution [CW96], as implemented in the XSB logic programming system [SWS$^+$], proved the viability of tabling technology in research areas like Natural Language Processing, Knowledge Based Systems and Data Cleaning, Model Checking, and Program Analysis. SLG resolution also includes several extensions to Prolog, namely support for negation [AB94], thus introducing novel applications in the areas of Non-Monotonic Reasoning and Deductive Databases.

Tabling works for both deterministic and non-deterministic applications, but is frequently used to reduce the second's search space. One of model's features is the definition of scheduling strategies that decide which subgoal resolutions to execute and the best instant in time to perform those evaluations [FSW96]. However, being embedded in a standard logical programming environment, tabling models must successfully cope with pruning operations. For this reason, the choice of the correct scheduling strategies among all the possible existing ones becomes of the greatest importance, since the wrong choice may lead to situations in which several *incomplete tables* are to be pruned and, consequently, their computation results in an undesirable waste of time and resources [RSC05].

A common concern in tabling systems is the possibility of memory exhaustion whenever query resolution results in huge or numerous tables. In such cases, the only way to proceed execution is to remove some of the computed tables from memory. The common approach, implemented by most tabling systems, is to provide a set of primitive instructions that allow programmers to arbitrarily select tables for deletion.

Both the described situations result in loss of useful information. Either for incomplete or complete tables, every subsequent call to the respective subgoals will result in a

full re-evaluation, with all of its associated costs. These costs are significant if the calls to subgoals at stake are part of massive computation goal solvers or depend on input/output operations.

1.1 Thesis Purpose

It is our strong belief that all of the above mentioned problems have a similar solution, that of storing subgoal tables into secondary memory, namely relational databases. Such a mechanism could not only preserve information and provide fast access to it, but also introduce the possibility to expand tabling systems' memory to the limit in a safe, effective and scalable way while, at the same time, keeping all the features provided by tabling systems.

This thesis addresses the design, implementation and evaluation of DBTab [CRF06], an extension to the YapTab sequential tabling engine [RSS97]. DBTab couples the YAP engine [SDRA] with the famous MySQL relational database management system [WA02], as suggested by Ferreira *et al.* [FRS04]. DBTab enhances YapTab's support tabled evaluation for definite programs by tabling space into MySQL relational schemes, featuring multi-user concurrency, transactional safety and crash recovery capabilities.

DBTab's design is largely based on the work of Rocha *et al.* [Roc06, Roc07], on efficient support for incomplete and complete tables in the YapTab tabling system, the work of Ferreira *et al.* [FRS04], on coupling logic programming with relational databases, and the work of Florescu *et al.* [FK99] on storing and querying XML data using relational database management systems.

The developed work intends to understand the implications of combining tabling with relational databases and thereby develop an efficient execution framework to obtain good performance results. Accordingly, the thesis presents both novel and well-known data structures, algorithms and implementation techniques that efficiently solve some difficult problems arising from the integration of both paradigms. Our major contributions include the table storage and recovery algorithms and the enhancement of YapTab's *least recently used* applicability.

In order to substantiate our claims we have studied the performance of our implementa-

tion, DBTab. A couple of benchmark predicates was run against a set of graph related tabled programs and their performance was duely noted, compared and interpreted. The gathered results show that DBTab indeed becomes an interesting approach when the cost of recalculating a table trie largely exceeds the amount of time required to fetch that same answer-set from the mapping relational tuple residing in the database, typically situations in which heavy side-effected routines are used during program execution. In our study we gathered detailed statistics on the execution of each benchmark program to help us understand and explain some of the obtained execution results.

The final goal for this work is to substantiate our belief that tabling and relational databases can work together, enhancing the YapTab engine performance and increasing the range of applications for Logic Programming.

1.2 Thesis Outline

The thesis is structured in seven major chapters. A brief description of each chapter follows.

Chapter 1: Introduction. This chapter.

Chapter 2: Logic Programming, Tabling and Persistence Models. Provides a brief introduction to the concepts of logic programming and tabling, focusing on Prolog, SLG resolution, and abstract machines for standard Prolog and tabling, with particular focus on the WAM.

Chapter 3: YapTab: The Sequential Tabling Engine. Describes the fundamental aspects of the SLG-WAM abstract machine, discusses the motivation and major contributions of the YapTab design and presents in some detail YapTab's implementation: its main data areas, data structures and the algorithms introduced to extend the Yap Prolog system to support sequential tabling.

Chapter 4: Relational data models for tabling. Discusses the benefits of the association of tabling and external memory storage mechanisms such as relational databases. Introduces two possible relational schemes, as well as the required transactional algorithms, their advantages and disadvantages.

Chapter 5: DBTab: Extending the YapTab Design. Introduces the implementation of DBTab, starting with a brief contextualization of the developed work. Afterwards, the chosen relational database management system and its major assets are discussed. Finally, the transformations produced in YapTab to enable the implementation of relational tabling are introduced, covering topics such as the developed API, the changes to YapTab's table space structures and operating algorithms. An alternative way to inquire for trie answers without fully reloading the trie into memory is also discussed.

Chapter 6: Performance Analysis. Presents an analysis of DBTab's performance over a set of benchmark programs. Discusses the overheads introduced by the relational storage extension, compares DBTab's different implementations performance and draws some preliminary conclusions on the obtained results.

Chapter 7: Concluding Remarks. Discusses the research results, summarizes the contributions and suggests directions for further work.

Chapters 4 and 5 include pseudo-code for some important procedures. In order to allow an easier understanding of the algorithms being presented in such procedures, the code corresponding to potential optimization is not included, unless its inclusion is essential for the description.

Chapter 2

Logic Programming, Tabling and Persistence Models

This chapter introduces the fundamental aspects of the research areas covered by this thesis. Concepts such as logic programming and tabling are discussed, focusing on Prolog, SLD resolution and SLG resolution, studying the practical aspects of their use in the implementation of abstract machines such as the WAM. The final section discusses persistence models for logic programming and practical approaches between logical systems and relational databases.

2.1 Logic Programming

Logic is the tool of trade for computer science, providing a concise language to express knowledge, assumptions and goals. It lays the foundations for deductive reasoning, establishing the consistency of premises given the truth or falsity of other statements [SS94]. The use of logic directly as a programming language, called *logic programming* [Llo87], improves the interface between humans and computers, allowing the programmer to focus on what the problem is, rather than on the details of how to translate it into a sequence of computer instructions. This line of reasoning is motivated by the belief that programs should reflect one's knowledge about a problem, organizing it in a program as a set of beliefs over real world objects (*axioms*) and relations between these objects (*rules*). Problems can thus be formalized as theorems

(*goals*) to be proved, with computation becoming a deduction of logical consequences for posed queries.

Logic programming languages fall into a major class denominated *declarative languages*, distinguishable from all other classes of languages for their strong mathematical basis. Members of this class drift away from the mainstream of developing languages in that they are derived from an abstract model - the Horn Clause Logic, a subset of First Order Logic - with no direct relation or dependency to the underlying machine operation set. Technically, the logical programming paradigm is based on a simple theorem prover that, given a theory (or *program*) and a *query*, uses the theory to search for alternative ways to satisfy the query.

Kalrsson [Kar92] states that logic programming is often mentioned to include the following major features:

- Variables are *logical* variables, which remain *untyped* until instantiated.
- Variables can be instantiated *only once*.
- Variables are instantiated via *unification*, a pattern matching operation finding the most general common instance of two data objects.
- At unification failure the execution *backtracks* and tries to find another way to satisfy the original query.

According to Carlsson [Car90], logic programming languages, such as Prolog, present the following advantages:

Simple declarative semantics. A logic program is simply a collection of predicate logic clauses.

Simple procedural semantics. A logic program can be read as a collection of recursive procedures. In Prolog, for instance, clauses are tried in the order they are written and goals within a clause are executed from left to right.

High expressive power. Logic programs can be seen as executable specifications that despite their simple procedural semantics allow for designing complex and efficient algorithms.

Inherent non-determinism. Since in general several clauses can match a goal, problems involving search are easily programmed in these kind of languages.

These advantages enhance the compactness and elegance of code, making it easier to understand, program and transform, and bringing programming closer to the mechanics of human reasoning.

2.1.1 Logic Programs

The basic constructs of logic programming are inherited from logic itself. There is a single data structure - the logical term - and three basic statements - facts, rules and goals.

Terms in a program represent world objects. In their simplest form, they can appear as *atoms*, representing symbolic constants, or as *variables*, standing for unspecified terms. They can also be presented as *compound terms* of the form

$$f(u_1,\ ...,\ u_m)$$

where f is the *functor* and the $u_1,\ ...,\ u_m$ are themselves terms. The functor is characterized by its name, which is an atom, and its arity, or number of arguments. *Predicates* are used to establish relationships between terms and consist on a compound term of the form

$$p\ (t_1,\ ...,\ t_n)$$

where p is the predicate name, and the $t_1,\ ...,\ t_n$ are the terms used as arguments.

A program consists basically of a collection of Horn Clauses, logical clauses with, at most, one positive literal allowed in the body. The simplest kind of Horn Clause is called a *fact*, representing an assertion of truth about a specific object. Using Prolog's notation, it is simply written as

$$A.$$

A more complex type, the *rule*, can be used to establish new relationships between known facts and/or previously established rules. It represents a logical clause of the form

$$\neg\ B_1\ \vee\ ...\ \vee\ \neg\ B_n\ \vee\ A.$$

that can be rewritten equivalently as

$$A \leftarrow B_1 \wedge ... \wedge B_n.$$

which is the most common form of representation in logical programming. In fact, these statements are written in Prolog as

$$A :- B_1, ..., B_n.$$

where A is the *head* of the rule and the $B_1, ..., B_n$ are the *body* subgoals.

It is possible that a clause has no positive literal at all. Such a clause must be written as

$$\neg B_1 \vee ... \vee \neg B_n.$$

It can be proved that this statement is logically equivalent to

$$False \leftarrow B_1 \wedge ... \wedge B_n.$$

which is by definition a Horn Clause, in this case denominated as a *goal*. Prolog's goals are written as

$$:- B_1, ..., B_n.$$

where $B_1, ..., B_n$ are the *body* subgoals.

Goals or *queries* are means of retrieving information from a logic program. The execution of a query Q against a logic program P leads to consecutive *assignments* of terms to the variables in Q until a *substitution* θ satisfied by P is found. A substitution is a (possibly empty) finite set of pairs $X_i = t_i$, where X_i is a variable and t_i is a term, with X_i not occurring in t_i and $X_i \neq X_j$, for any i and j. *Answers* (or *solutions*) for Q are retrieved by reporting for each variable X_i in Q the corresponding assignment $\theta(X_i)$. The assignment of a term t_i to a variable X_i is called a *binding* and the variable is said to be *bound*. Variables can be bound to other distinct variables or non-variable terms.

Execution of a query Q with respect to a program P proceeds by reducing the initial conjunction of subgoals of Q to subsequent conjunctions of subgoals according to a refutation procedure. The first use of this approach in practical computing is a sequel to the *resolution principle* of Robinson's [Rob65], introduced by Kowalski [Kow74] in 1974 under the name of the *Selective Linear Definite resolution* (*SLD resolution*). Given the logical program and a goal to prove, the SLD resolution proceeds as follows:

1. Starting from the current conjunction of subgoals

$$:- G_1,\ ...,\ G_n.$$

a predefined $select_{literal}$ rule arbitrarily selects a subgoal (or literal) G_i. The order of reductions is irrelevant for answering the query, regarding that all subgoals are reduced;

2. The program is then searched for a clause whose head goal unifies with G_i. If there are such clauses then, according to a predefined $select_{clause}$ rule, one is selected;

3. If the selected clause, unifying with G_i, is of the form

$$A\ :- B_1,\ ...,\ B_m.$$

a substitution θ to the variables of A and G_i has been determined such that $A\theta\ =\ G_i\theta$. Execution proceeds by replacing G_i with the body subgoals of the selected clause, followed by the appliance of the substitution θ to the variables of the resulting conjunction of subgoals, thus leading to:

$$:- (G_1,\ ...,\ G_{i-1},\ B_1,\ ...,\ B_m,\ G_{i+1},\ ...,\ G_n)\theta.$$

Notice that if G_i is a fact, it is simply removed from the conjunction of subgoals, thus resulting in:

$$:- (G_1,\ ...,\ G_{i-1},\ G_{i+1},\ ...,\ G_n)\theta.$$

4. A *successful SLD derivation* is found whenever the conjunction of subgoals is reduced to the $True$ subgoal after a finite sequence of reductions. When there are no clauses unifying with a selected subgoal, then a *failed SLD derivation* is found, which can be solved, in Prolog, by applying a *backtracking* mechanism. Backtracking exploits alternative execution paths by **(i)** undoing all the bindings made since the preceding selected subgoal G_i, and by **(ii)** reducing G_i with the next available clause unifying with it. The computation stops either when all alternatives have been exploited or when an answer is found.

Notice that SLD resolution does not stand for a particular algorithm but for an entire class of algorithms, which particularly differ in their proof tree traversal methods and

halting conditions. In [Kow79], Kowalski stated that logic programming is about expressing problems as logic and using a refutation procedure to obtain answers from the logic. Therefore, in a computer implementation, the $select_{literal}$ and $select_{clause}$ rules must be specified. Different specifications lead to different algorithms, whose execution will result in different search spaces. Although the semantics of proofs is unaltered, in some cases, a successful derivation may not be attained [AvE82]. The Prolog implementation specifies that, given a query to solve, the $select_{literal}$ rule must choose the leftmost subgoal, while the $select_{clause}$ rule must follow the order in which the clauses appear in the program.

2.1.2 The Prolog Language

In 1965, Robinson published a revolutionary paper [Rob65] describing the unification algorithm and resolution principle. Based on Robinson's work, Kowalski [Kow74] acknowledged the procedural semantics of Horn clauses in the early 1970's. He provided some theoretical background showing that a logical axiom

$$A \leftarrow B_1 , \ \ldots , \ B_n$$

can have both a declarative reading - *A is true if* B_i *and . . . and* B_n *are true* - and a procedural reading - to solve A, solve B_1 and B_2 and . . . and B_n. Almost simultaneously, Colmerauer and his group developed a theorem prover that embodied Kowalski procedural interpretation [CKPR73]. It was called *Prolog*, as an abbreviation for *PROgramation en LOGic*, and intended to implement a man-machine communication system in natural language.

In the late years of that decade, David H. D. Warren and his colleagues made Prolog a viable language by developing the first efficient implementation of a compiler, named Prolog-10 [War77]. The compiler itself was almost entirely written in Prolog, proving the power of logic programming to solve standard programming tasks. Its major contribution, though, was in term of syntax formalism, setting a standard that would be followed in all subsequent implementations. In 1983, Warren laid another important landmark with his *Warren Abstract Machine* [War83] (WAM), an abstract machine for the execution of Prolog code. The model, consisting of a memory architecture and an instruction set, proved to be the most efficient way yet discovered to implement Prolog

compilers, being adopted by most *state-of-the-art* systems for logic programming languages.

In 1982, Japan's Ministry of International Trade and Industry started its *Fifth Generation Computer Systems* project. It was intended to create an *epoch-making* computer with supercomputer-like performance leveraged by massive parallelism, and usable artificial intelligence capabilities. This increased the interest in sequential and parallel models, and many of the proposed models become real implementations. The advances made in the compilation technology of sequential implementations of Prolog proved to be highly efficient, which has enabled Prolog compilers to execute programs nearly as fast as conventional programming languages such as C [Roy90].

Since then, Prolog has reached maturity, with the *The Edinburgh Prolog* family becoming a standard for all implementations. Still, there were some obstacles to overcome. First Order Logic, by itself, is not suited to cope with the needs of every day programming. Useful languages must include arithmetic and I/O operations, database management routines, among others. To cope with these demands, some extensions were introduced:

System predicates. Class of predicates for system-related procedures, such as arithmetic operation, ground terms comparison and basic structure inspection;

Meta-logical predicates. Introduced to give the programmer a tight control on the program execution. These predicates are often used to inquire the state of the computation and perform comparison and inspection of non-ground terms. The inspection operation plays a significant role: the introduction of system predicates causes the need to test the type of terms at runtime. As this feature is not taken into account in formal logic, these predicates have no logical meaning.

Extra-logical predicates. This kind of predicates fall outside the logic programming model, achieving some useful side-effects in the course of being satisfied as a logical goal. Three main types are defined: predicates that handle input/output operations; predicates interfacing the underlying operating system; and predicates used to manipulate the Prolog internal database, handling the assertion and removal of clauses from the program being executed.

Cut predicate. This predicate adds an explicit form of control over the backtracking

mechanism, aiming to make programs shorter and more efficient. The use of cut predicates generally makes programs more difficult to understand and less logical in nature. Two forms of the predicate are possible: the *green* and *red* cuts [SS86]. The first form of cut notifies the interpreter to stop looking for alternative answers; in a sense, it discards correct solutions that are not needed. Omitting green cuts from a program should still give the same answers, eventually taking a little longer to do so. The second and most dangerous form alters program execution by effectively pruning the answer tree, thus changing the logical meaning of the program; depending on the way they are used, wrong answers may even be introduced.

Other predicates. These include *extra control* predicates to perform simple control operations and *set* predicates that give the complete set of answers for a query.

All of these predicates manifest an *order-sensitive* behaviour, producing potentially different outcomes when different $select_{literal}$ or $select_{clause}$ rules are specified. Furthermore, some of these predicates, such as the cut, require a deep knowledge of the execution model of Prolog to be successfully and safely used. Readers not familiarized with Prolog should refer to the standard textbooks [CM94, Llo87, SS94], in search for more details on these subjects.

2.1.3 The Warren Abstract Machine

As mentioned before, the Warren Abstract Machine became the most popular way to implement efficient Prolog compilers. This subsection presents a rather simplified view of the WAM implementation. Readers are encouraged to refer to more complete works on the subject, such as the Aït-Kaci's tutorial [AK91], or Boizumault's book [Boi88] on Prolog's implementation.

The virtual machine architecture presents a memory layout divided into five distinct execution stacks and a large set of specialized registers. A robust, yet somewhat large, low-level instruction set is supplied to perform memory and register manipulation, control, indexing, choice and unification instructions. The state of a computation can be obtained observing the contents of the WAM data areas, data structures and registers at a particular instant in time. Figure 2.1 provides a closer look at the WAM's

organization.

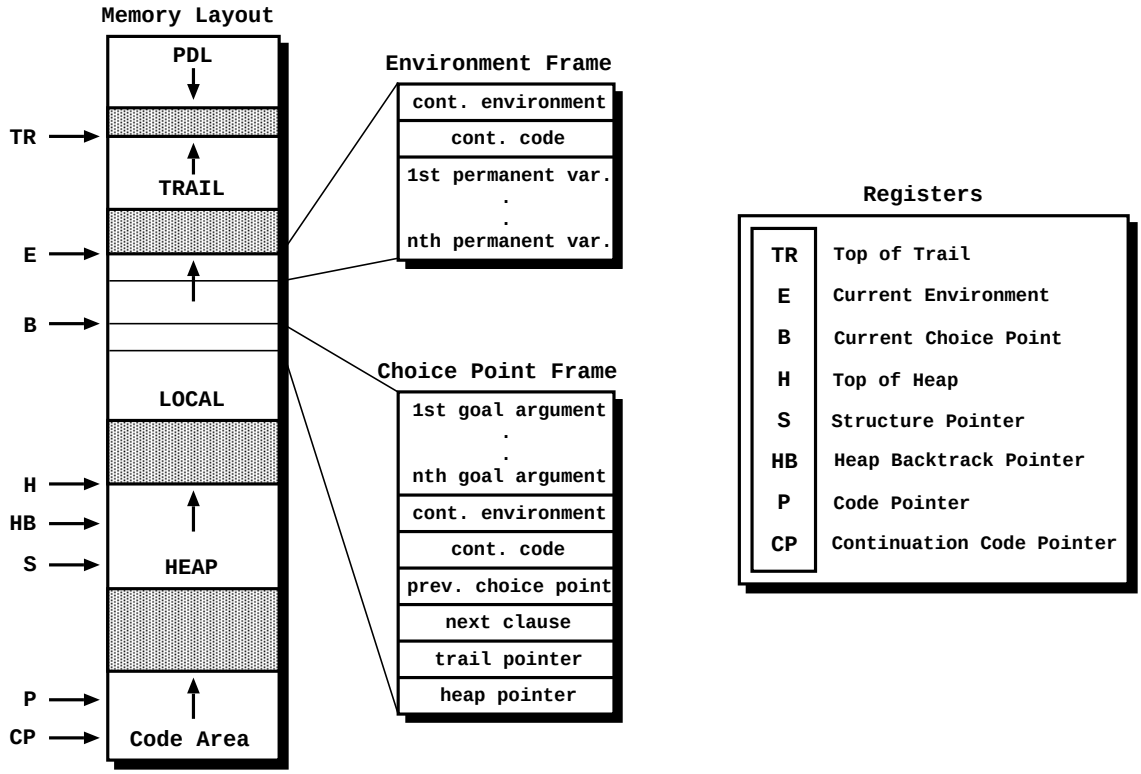

Figure 2.1: WAM memory layout, frames and registers

A brief explanation of its stacks and registers might be in order to better understand the virtual machine functioning:

PDL: A push down list used by the unification process.

Trail: An array of addresses used to store addresses of (Local or Heap) variables due to be reset upon backtracking. The `TR` register points at the top of this stack.

Local: Used to store the *environment* and *choice point* frames. This approach is not universal. In fact, some WAM implementations use separate execution stacks to store environments and choice points. In the original specification, the `E` register points at the current active environment, while the `B` register points at the current active choice point, which is always the last one pushed into the stack.

Heap: An array of data cells used to store variables and compound terms that cannot be stored in the Local stack. The `H` register points at the top of this stack, the `S`

register is used during unification of compound terms and the `HB` register is used to determine if the current binding is a conditional one.

Code Area: Used to keep the compiled instructions of the loaded programs. The `P` register points at the currently executing instruction and the `CP` register points at the next instruction to execute after successful termination of the currently invoked call.

The two mentioned data-structures, *environments* and *choice points*, play significant roles during program evaluation. Environments track the flow of control; every time a clause containing several body subgoals is picked for execution, an environment is pushed onto the stack, being popped off before the last body subgoal gets executed. Environment frames store a sequence of cells, one for each *permanent variable*[1] belonging to the body of the invoked clause. Two other important pieces of information are also kept: the stack address of the previous environment and the code address of the next instruction to execute upon successful return of the currently invoked one. Choice points, on the other hand, store open alternatives. Whenever there is more than one candidate clause for the currently executing goal, one of these structures is pushed onto the stack. The contents of the choice point are used either to restore the state of the computation back to when the clause was entered or, alternatively, to supply a reference to the next clause to try upon failure of current executed one. When all alternatives are exhausted, the structure is popped off the stack.

The five major groups of instructions in the WAM instruction set are briefly explained next:

Memory and register instructions Divided into *put*, *get* and *set* instructions, they are used on structures and variables to push or pop their addresses to or from the Heap and copy those addresses into specialized registers. These three major classes are further subdivided in specialized versions to differently treat the first and following occurrences of a variable in a clause, as well as constants, lists, and other compound terms;

Choice point instructions Divided into *try*, *retry* and *trust* instructions, they are used to manage the *allocation*/*deallocation* of choice points and to recover a

[1] A permanent variable is a variable which occurs in more than one body subgoal [AK91].

computation's state using the data stored within these structures;

Control instructions Destined to manage the *allocation/deallocation* of environments and the *call/execute/proceed* sequence of subgoals;

Unification instructions Implemented as specialized versions of the unification algorithm according to the position and type of the arguments. There are proper unification instructions to perform bound and unbound variable unification, global and local values unification and constant unification;

Indexing instructions Used to accelerate the choice of which clauses to unify with a given subgoal call. The call's first argument determines the destination address within the compiled code that can directly index the unifying clauses;

Cut instructions Used to discard alternative choice points up to the one associated with the current call and eliminate unconditional bindings from the trail up to that point.

2.2 Tabling for Logic Programs

Prolog's efficient SLD resolution strategy faces serious limitations when confronted with redundant subcomputations or infinite loops. Numerous attempts have been made to cope with these problems, establishing alternative resolution methods. One in particular, named *tabling*, *tabulation* or *memoing* [Mic68], consists of a resolution strategy similar to SLD that remembers subcomputations in order to reuse them on subsequent requests. Tabling has proven its value in several fields of research, such as Natural Language Processing, Knowledge Based Systems and Data Cleaning, Model Checking and Program Analyses. Several tabling methods have been proposed, the most famous being OLDT [TS86], SLD-AL [Vie89], Extension Tables [Die87], Backchain Iteration [Wal93].

Tabling has out-grown its initial purpose and has been used to other endings than ensuring termination for Horn clause programs. Its capacity to maintain global elements of a computation in *memory*, such as information about whether one subgoal depends on another and whether the dependency is through negation, proved its usefulness

for normal logic programs evaluation under the *Well-Founded Semantics* (WFS), a natural and robust declarative meaning to all logic programs with negation.

In practice, the WFS framework depends on the implementation of an effective and efficient evaluation procedure. Among the several implementation proposals, one in particular, called *Linear resolution with Selection function for General logic programs* [CW96] (*SLG resolution* for short), has become remarkably notorious. This tabling based resolution method has polynomial time data complexity and is sound and search space complete for all non-floundering queries under the well-founded semantics. SLG resolution can thus reduce the search space for logic programs and in fact it has been proven that it can avoid looping, thus terminating for all programs with the bounded-term-size property [CW96]. SLG resolution has broadened the application scope for Prolog, extending its usability in fields such as Non-Monotonic Reasoning and Deductive Databases.

2.2.1 SLG Resolution for Definite Programs

Logic programs that do not include negation are called *definite programs*. When applied to this class, SLG resolution is equivalent to SLD resolution with tabling support. Sagonas and Swift present a simplified overview of SLG resolution for definite programs in [SS98], from which this subsection borrows some definitions. A more formal and further detailed explanation is performed by Chen and Warren in [CW96].

In a nutshell, a *SLG system* is a forest of SLG trees along with an associated *table*. SLG trees are similar to those used by SLD resolution, but their root nodes stand for calls to subgoals of tabled predicates. Non-root nodes either have the form

$$:- \ fail.$$

or

$$Answer\_Template \ :- \ Goal\_List.$$

where $Answer\_Template$ is a positive literal and $Goal\_List$ is a possibly empty sequence of subgoals. The *table* is a set of ordered triples of the form

$$\langle Subgoal, \ Answer\_Set, \ State \rangle$$

where the first element is a subgoal, the second a set of positive literals, and the third one of the *complete* or *incomplete* constant values.

The nodes of an SLG tree are described by its *status*. They are divided into four main classes:

Generator nodes Root nodes are labeled with a *generator* status, as they correspond to the first calls to tabled subgoals. They use program clause resolution to produce answers;

Consumer nodes Non-root nodes standing for variant calls to the tabled subgoal. These consume their answers from the table space;

Interior nodes Non-root nodes standing for non-tabled subgoal calls. These nodes are evaluated by standard SLD evaluation;

Answer nodes Non-root nodes that have an empty *Goal_List*.

Given a tabled program $\mathcal{P}$, an *SLG evaluation* θ for a subgoal G of a tabled predicate is a sequence of systems

$$\mathcal{S}_0,\ \mathcal{S}_1,\ ...,\ \mathcal{S}_n$$

such that:

1. $\mathcal{S}_0$ is the forest consisting of a single SLG tree rooted by G and the table

$$\langle G,\ \emptyset,\ incomplete\rangle$$

2. for each finite ordinal k, transition to $\mathcal{S}_{k+1}$ is obtained from $\mathcal{S}_k$ by an application of one of the following tabling operations:

   New Tabled Subgoal Call. Given a node $\mathcal{N}$ with selected subgoal S, where S is not present in the table of $\mathcal{S}_k$, create a new SLG tree with root S and insert into the table the entry

$$\langle S,\ \emptyset,\ incomplete\rangle.$$

   Program Clause Resolution. Given a node $\mathcal{N}$ in $\mathcal{S}_k$, that is either a generator node or an interior node of the form

$$Answer\_Template\ :-\ S,\ Goals.$$

and the program clause

$$Head \ :- \ Body$$

whose $Head$ term has not been used for resolution at node $\mathcal{N}$, and unifies with S through substitution θ, produce a child for $\mathcal{N}$ of the form

$$(S \ :- \ Body)\theta.$$

if $\mathcal{N}$ is a generator node, and

$$(Answer\_Template \ :- \ Body, \ Goals)\theta.$$

if $\mathcal{N}$ is an interior node.

New Answer. Given a node

$$A \ :-$$

in a tree rooted by a subgoal S, such that A is not an answer in the table entry for S in $\mathcal{S}_k$, add A to the subgoal's answers set in the table.

Answer Resolution. Given a consumer node $\mathcal{N}$ of the form

$$Answer\_Template \ :- \ S, \ Goals.$$

and an answer A for S in $\mathcal{S}_k$, such that A has not been used for resolution against $\mathcal{N}$, produce a child of $\mathcal{N}$

$$(Answer\_Template \ :- \ Goals)\theta.$$

where θ is the substitution unifying S and A.

Completion. Given a set of subgoals $\mathcal{C}$ for which all its possible substitutions have been performed, remove all trees whose root is a subgoal in $\mathcal{C}$, and change the state of all table entries for subgoals in $\mathcal{C}$ from *incomplete* to *complete*. Subgoals in $\mathcal{C}$ are then said to be *completely evaluated*. If mutually dependencies are present, forming a *strongly connected component* (or *SCC*), start by the oldest subgoal in the SCC [SS98], denominated the *leader* subgoal.

If no operation is applicable to $\mathcal{S}_n$, $\mathcal{S}_n$ is called a $final\ system$ of θ.

2.2.2 Examples of Tabled Evaluation

Standard SLD evaluation is ineffective when asked to deal with positive loops. Figure 2.2 presents a small example of such a case. The upper box represents a Prolog system, which has loaded a small program defining the arcs of a small directed graph (`arc/2` predicate) and a relation of connectivity between two nodes of the graph (`path/2` predicate). The system is asked to solve the query goal `?- path(a,Z)`. The lower box depicts the infinite SLD tree when the leftmost branch of the corresponding search tree is explored.

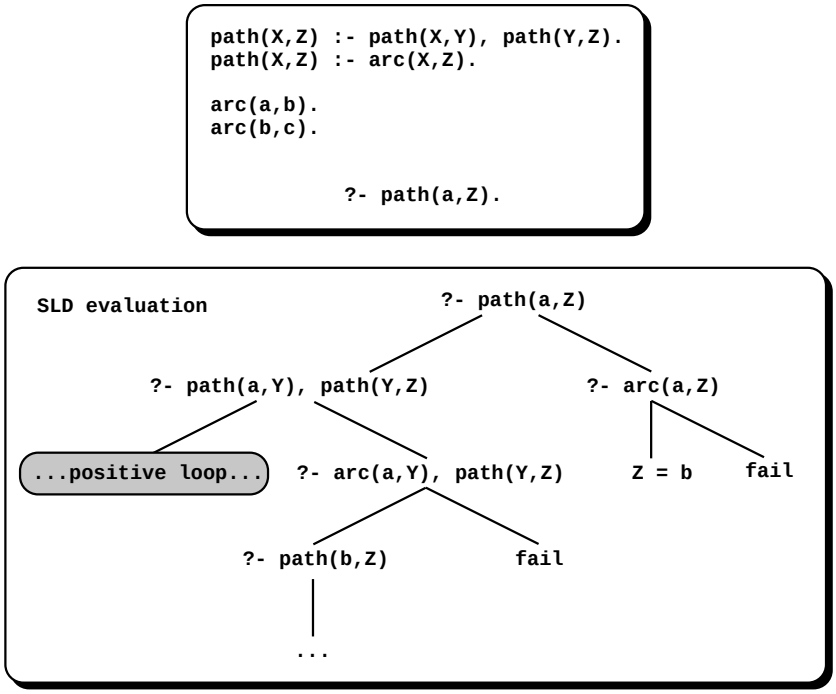

Figure 2.2: An infinite SLD evaluation

If a tabling strategy is applied to solve the same query goal over the same program, termination is ensured since the search tree is finite. Figure 2.3 depicts the evaluation sequence for the given query goal. At the top-left box, the program code now conveys the `:- table path/2` directive, an indication that tabling must be applied to solve predicate `path/2` subgoals calls. At the top-right box, the figure illustrates the appearance of the table space at the end of the evaluation. The bottom block shows the resulting forest of trees for the three tabled subgoal calls. The numbering of nodes denotes the evaluation sequence.

Evaluation starts by creating a new tree rooted by `path(a,Z)` and by inserting a new

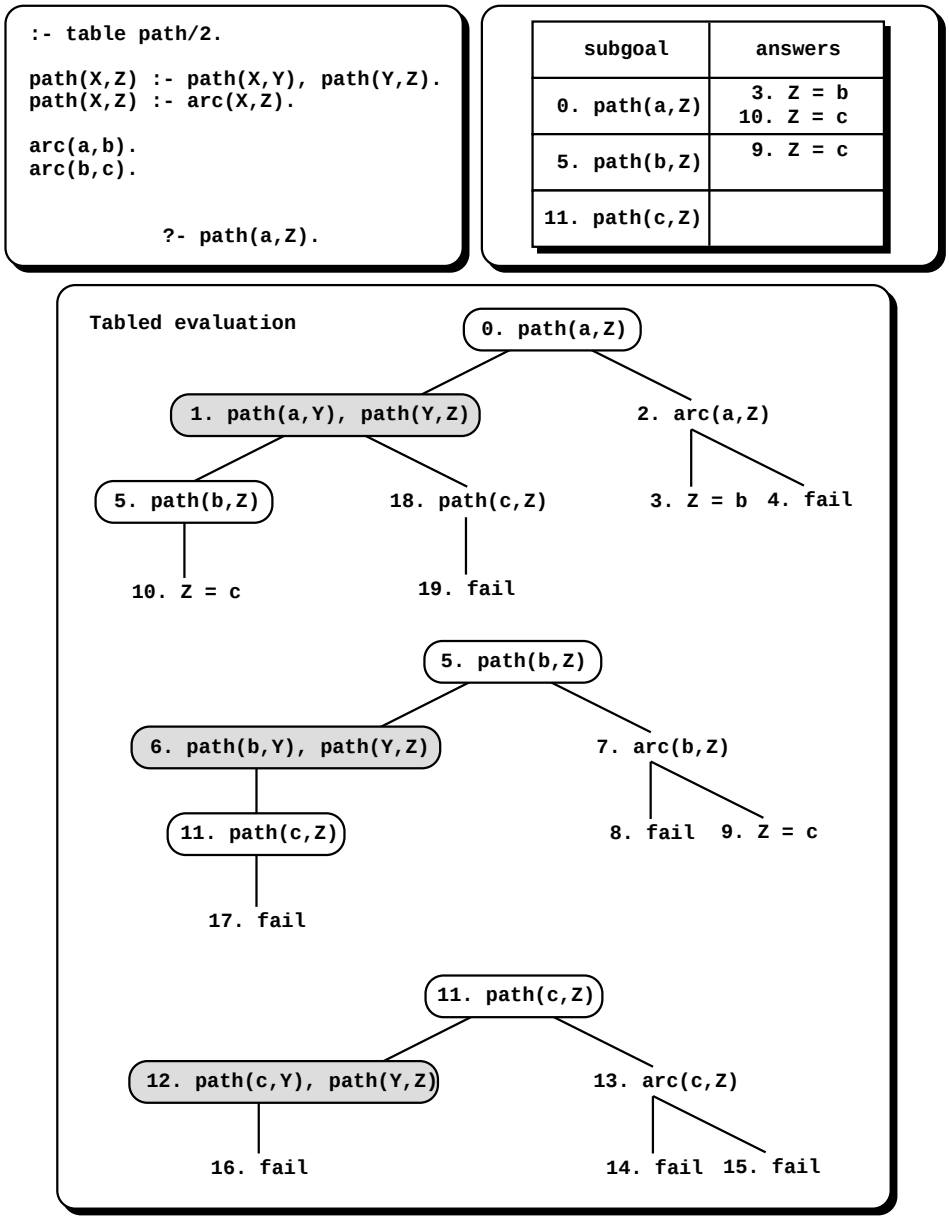

Figure 2.3: A finite tabled evaluation

entry in the table space for the subgoal. In node 1, the first clause for `path/2` is selected for evaluation, thus leading on a call to `path(a,Y)`. Since this is a variant of the initially encountered subgoal, no tree creation is required. Instead, previously stored answers are searched in the table space and, in their absence, evaluation of the node is suspended[2]. Since `path/2` is composed of two clauses, node 2 is then evaluated, resulting on the first solution to `path(a,Z)` (node 3). After exhausting

[2]Suspended nodes are depicted by the gray oval boxes surrounding the call.

all local alternatives, computation resumes at node 1, with the newly found answer allowing the call to subgoal `path(b,Z)`. As it is the first call to the subgoal, a new execution tree and a new entry in the table space are required. Evaluation proceeds as previously stated, giving rise to one answer for `path(b,Z)` (node 9). This is also a new solution to `path(a,Z)`, hence this answer is also stored in the table space (node 10). The call to subgoal `path(c,Z)` (node 11) fails to produce new answers, so execution backtracks to the top, `path(a,Z)` becomes *completely evaluated* and the top query succeeds with two different possible answers, as visible in the table. Termination is ensured by avoiding the recomputation of three found subgoals `path(a,Z)`,`path(b,Z)` and `path(c,Z)` in nodes 1, 5 and 13 respectively.

Tabling may also reduce the complexity of a program, requiring fewer steps to execute it. The top area of Fig. 2.4 presents a Prolog system that has loaded the well-known Fibonacci sequence generator program and is asked to solve the `?- fib(5,Z)` query goal. The bottom area illustrates two execution tables, one for standard SLD evaluation and the other for the evaluated counterpart. At the left side, with SLD evaluation, computing *fib(n)* for some integer n will search a tree whose size is exponential in n. On the other hand, when predicate `fib/2` is declared as tabled, each different subgoal call is only computed once, with repeated calls being recovered among the answers previously stored in the table space. Therefore, the search tree size becomes linear in n, thus reducing the required execution steps to cover the whole search space.

2.3 Persistence Models for Logic Programming

Most often in logic programming the execution of rules becomes an highly complex and/or resource consuming task. In such cases, it is desirable that state may be preserved at a global level, for instance within a module, and despite of backtracking occurrence. Usually, this is done through the declaration of *dynamic predicates*, i.e, predicates whose definition can be modified at runtime. Standard logic systems, like Prolog, usually provide an internal rule database, where dynamic predicates can be stored to, updated or removed from through the use of specialized built-in primitives.

In SLG based systems, the programmer can achieve the desired effect in a more transparent manner by declaring a *tabled* predicate. State is thus kept in the table space until the end of program execution, or until the programmer chooses to dynamically

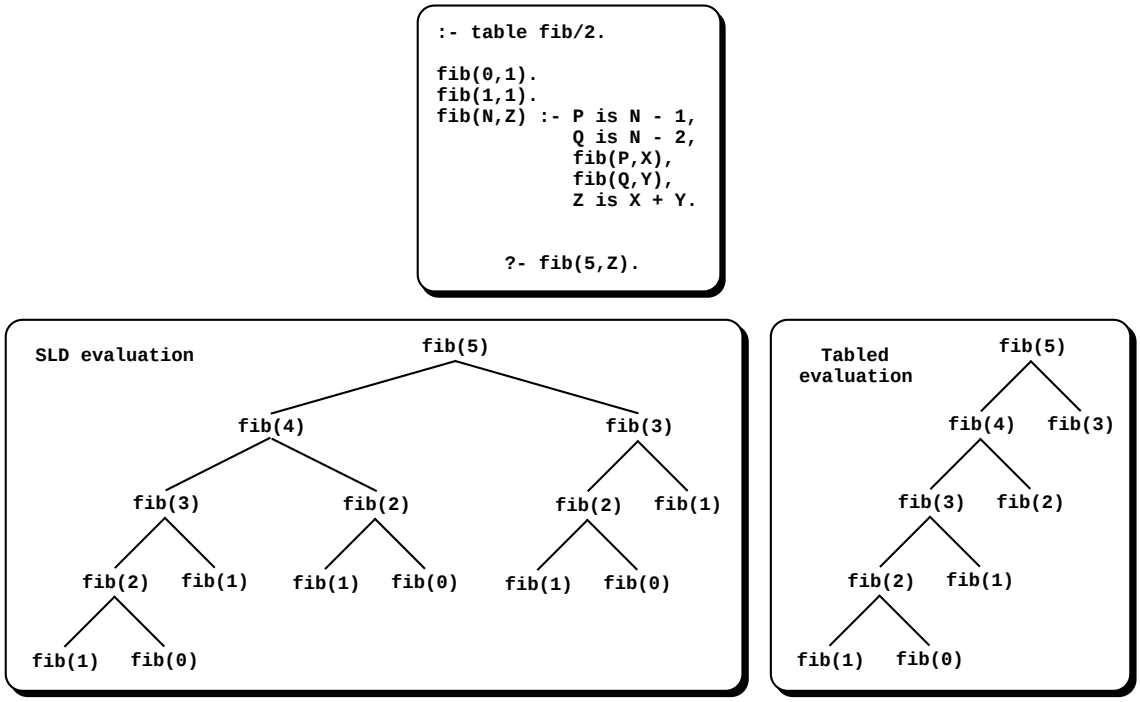

Figure 2.4: Fibonacci complexity for SLD and tabled evaluation

remove it by invoking tabling primitives used to abolish tables.

Although different in nature, both approaches suffer from common problems. First of all, since dynamic (or tabled) predicates are kept in memory, undesirable limitations to execution may rise due to the heavy dependency on the underlying operative system's memory space. Secondly, the lifetime of internal state is bound to that of the process that is running the logic system. Whenever it stops, the achieved results are lost. This means, of course, that posterior executions must repeat the entire process to reach the same state before resuming computation.

Several proposals have been developed to cope with these problems. Correas *et al.* define *persistent predicates* [CGC+04] as a special kind of dynamic data predicates that reside in some persistent external medium, such as sets of files or relational databases. The authors consider the main advantages of external storage to be:

- State of persistent predicates is preserved across executions of the logic program, providing that no other process changes it;
- The semantic reading of persistent predicates is kept, in that they may appear anywhere in the body of clauses just like regular dynamic predicates;

- Specialized built-ins may be defined or extended to insert, update and retract persistent predicates. Updating external stored data is considered to be atomic and transactional, since the external storage is definitely modified at the end of this operation;

- The concept abstracts the way in which persistency is implemented, enabling the swapping of the underlying storage technology without major transformations to the logic program, with the exceptiion of a single directive at the beginning of the program's code to redirect data manipulation tools to the desired media.

2.3.1 Prolog and Databases

Logic systems, such as Prolog, are designed to efficiently solve goals against memory-resident programs using a *top-down*, *tuple-oriented* approach. As mentioned before, some limitations arise from this model, since it is prone to infinite loops and redundant computation. Furthermore, the manipulation of small data blocks, such as tuples, is known to cause inefficient disk access, as disk-related operations commonly present block-searching overheads. When applications manipulate large amounts of data, most of it is bound to reside in secondary storage, hence high efficiency regarding disk access and storage capabilities is most desirable. These characteristics are often recognized to be the trademark of Relational Database Management Systems (RDBMS). If one can find a way to couple these two distinct worlds together, some leverage can be gained for logic programs, both functionally and semantically.

Semantically, *Relational Algebra* may be seen as a non-recursive subset of Horn Clause Logic [Lie82, vEBvEB86, IvE92]. One can express a logical predicate as a relation, a logical *ground* fact as a tuple belonging to that relation and each of the predicate's arguments as attributes for the relation. Logical goals can be expressed by means of relational queries, using the well known $\times$ (*join*), $\cup$ (*union*), $-$ (*difference*), π (*projection*), σ (*selection*) and ρ (*attribute renaming*) operators to obtain the set of all possible answers, in a similar way to second-order programming techniques. In relational databases management systems, evaluation is performed in a *bottom-up*, *set-oriented* fashion, thus ensuring *completeness* and *termination* of the evaluating methods.

Even though *Relational Algebra* is properly included in Horn Clause Logic, the opposite

is not true; a recursive Horn Clause rule may not be expressed as a single or even as a conjunction of relational operators. To illustrate this handicap, recall the example program introduced back in Fig. 2.3. A list of fact predicates `arc/2` is declared to state that a few directed connections between two arbitrary nodes of the graph are known. In a relational database, one can represent such a predicate by defining the relation

$$Arc(StartNode, EndNode)$$

and inserting tuples into it to establish the known arcs. To verify the presence of a particular arc in the relation, for instance the arc connecting the node labeled as a and b, the logical query

$$?- arc(a, b).$$

can be emulated using the relational operation

$$\sigma_{StartNode="a",EndNode="b"} Arc$$

which returns a single tuple if the relation holds or fails otherwise. The

$$?- path(a, c).$$

goal, on the other hand, may not be correctly evaluated, since no single relational expression, using one or several combined operators, can be devised to generally express the first clause

$$path(X, Z) :- path(X, Y), path(Y, Z).$$

All a relational system can do is evaluate the second clause as

$$path(X, Z) :- arc(X, Z).$$

or, equivalently,

$$\pi_{FirstNode,LastNode}Arc$$

and verify that no arc exists between the denominated nodes, thus failing.

In spite of these advantages, some adjustments must be made in order to produce practical implementations. First of all, Prolog's evaluation model must retain its goal-oriented approach, but it must also become more *set-oriented*. Ideally, program clauses must be fit for *concurrent* evaluation. Programs may have to change the execution order for some clauses, grouping predicate known to occur consecutively in a single transaction, thus minimizing communication overheads. Some precautions must also be taken with concern to data *consistency* and *visibility*. The use of relational databases introduces the notion of concurrency, enabling the use of the same relational space by several executing Prolog systems. Therefore, is it imperative that the final state of the common persistent predicates reflects the changes made by each single process, while maintaining these changes available for all of the others.

2.3.2 Deductive Databases

Deductive databases are database management systems whose query language and storage structure are (usually) designed around a logical model of data [RU93].

Research in this field began back in the first stages of logic programming. Green and Raphael [GR68] recognized, in 1968, the connection between theorem proving and deduction in databases. They have presented several question-answering systems that used a version of Robinson's resolution principle to systematically perform deduction in a database context. In 1976, Van Emden and Kowalski provided a strong basis for the semantics of logic programs in [vEK76], where they proved that the minimal *fixpoint* for a set of Horn clauses coincides with its minimal Herbrand model. Several important publications were compiled and published by Gallaire and Minkers, including Reiter's paper on *closed world database* assumption [Rei78] and Clark's paper on *negation-as-failure* [Cla78]. In 1979, Aho and Ullman published a paper [AU79] on relational algebra with a fixpoint operator, setting the principles for bottom-up query evaluation. Kellog and Travis addressed the distinction between *extensional* and *intensional* knowledge [KT79] in 1979. In 1982, Minkers and Nicolas [MN83]

described *bounded recursive* queries as recursive queries that may be transformed into non-recursive equivalents.

Deductive databases establish the bridge between logic programming and database management systems. They can be thought of as a natural evolution for relational databases, since both systems share a declarative approach to data querying and updating, allowing users to focus on what information to manipulate, rather than on how to process that manipulation. However, to overcome the previously explained limitation of expressiveness for relational languages (such as SQL), deductive databases use a new non-procedural query language, called *Datalog*. This new language, based on Relational Algebra, is extended to support recursion and defines two different types of statements:

Data (facts) are represented by predicates with *ground* arguments consisting of *constant atomic* terms. Each of these predicates is represented *extensionally* by the storing in the database a relation of all of the tuples known to be true for that predicate. Each relation must have an unique name (functor) and a fixed number of arguments (arity);

Rules (program) are represented in a Prolog-style notation as

$$p(X_1,\ \ldots,\ X_n)\ :-\ q_1(X_{11},\ \ldots,\ X_{1k}),\ \ldots,\ q_m(X_{m1},\ \ldots,\ X_{ml}).$$

Both the rule head p and each of the subgoals q_i must be atomic formulas consisting of a predicate applied to either atom or variable terms. Each q_i may be a fact predicate or the head of another rule. Each of the X_n arguments in the head p must appear at least once in one of the q_i formulas.

Logical rules can be defined to express complex views of database relations and even integrity constraints. The major advantage of rules over the common implementation of these database tools lies in the fact that rules can be defined in a recursive fashion, thus increasing the expressiveness of the query language. Rules may also depend upon each other. In [RU93], Ramakrishnan states that a predicate p *depends upon* a (not necessarily different) predicate q if some rule with p in the head has a subgoal whose predicate is either q or (recursively) depends on q. If p depends on q and q depends on p, they are said to be *mutually recursive*. A program is said to be *linear recursive*

if each rule contains at most one subgoal whose predicate is mutually recursive with the head predicate.

Despite of its apparent similitude to logic programs, Datalog is unique due two important characteristics:

1. the ability to manage persistent data in an efficient way;
2. guaranteed termination for query evaluation.

Efficient disk-access is a central concern on deductive databases. The existing prototypes may be categorized in one of two main classes:

Integrated Systems are Prolog systems that support persistent data by themselves. The most famous systems belonging to this class are LDL [TZ86], Aditi [VRK+93] and Glue-NAIL [PDR91];

Coupled Systems provide mechanisms that enable the translation of logical queries into SQL directives, using either built-in primitives or external *Prolog-to-SQL* compilers[3]. Systems like CORAL [RSS92], XSB [SSW94] and YAP [FR04] implement coupled approaches.

The major advantage of coupled systems is their ability to keep the deductive engine and the relational database management system separate. However, there is a significant price to pay, since the communication between these two modules most often constitutes a *tuple-a-time* bottleneck. Major efficiency improvements may be achieved through the development of a rigorous schema for data clustering and buffering, respectively, on disk and in main memory. The use of set-oriented database operators is of the most importance when aiming at a minimization of the number of memory pages sent to/retrieved from secondary storage.

Guaranteed complete evaluation is a consequence of bottom-up evaluation. Starting from the extensional facts in the database, the bodies of rules are repeatedly evaluated against both extensional and intensional facts, to infer new facts from the heads. This approach, denominated *naïve* evaluation, produces all possible answers for queries,

[3] A famous *Prolog-to-SQL* external compiler was produced by Draxler [Dra91].

even some undesired ones. Despite of this possible waste, bottom-up evaluation avoids infinite loops and repeated recomputation of subgoals, while enabling the use of *set-oriented* relational operators.

To avoid time and memory waste induced by pure bottom-up evaluation, deductive databases often use *magic-sets* [BMSU86, BR91], a rule rewriting technique that binds predicates arguments to constants to generate filters that avoid the generation of facts unrelated to the desired subgoals. Hence, magic-sets combine the benefits of top-down goal-orientation ability while using the loop-free and redundancy check bottom-up features. The magic-sets technique was developed to handle recursive queries, but is also fit to handle non-recursive queries. A full description of this topic is presented in [RU93].

Magic-sets are often used in conjunction with redundant subcomputation prevention techniques. The XSB engine uses *tabling* as discussed in section 2.2. Other techniques can be used, though. One in particular, called *semi-naïve evaluation* [BR87], is regarded as highly effective. In each round of a bottom-up evaluation, the substitution factors for a rule's subgoals are reorganizing so that at least one of the subgoals is replaced by a fact discovered in the last round. Semi-naïve evaluation is widely spread among the most important deductive systems, namely Aditi, CORAL, LDL and Glue-NAIL.

Deductive databases provide other extensions to Horn Clause Logic, such as negation in rules bodies, set-grouping and aggregation. However, since these topics fall out of this thesis purpose, they are not presented here. Introductory text and useful references on these topics are provided in [RU93].

2.4 Chapter Summary

This chapter provided a small introduction on logic programming, tabling and persistence models. It started with brief descriptions of logic programs, the Prolog language and its implementation based on the *Warren Abstract Machine* (WAM). It proceeded by briefly reviewing the underlying features of SLG resolution and illustrated the advantages of tabling in a logic programming framework by example. The final section presented persistence models for logic programming, discussed practical approaches

between logical systems and relational databases.

Chapter 3

YapTab: The Sequential Tabling Engine

YapTab [RSS00] is a sequential tabling engine embedded in the YAP Prolog [San99] system. YapTab is based on the SLG-WAM engine [Sag96, SSW96, SS98] as first implemented in the XSB Prolog [SWS+] system. YapTab is also the base tabling engine for the OPTYap [Roc01] system that combines or-parallelism with tabling.

First, the fundamental aspects of YapTab are briefly discussed, comparing their implementation with the ones in the SLG-WAM abstract machine, and then we detail the YapTab implementation. This includes presenting the main data areas, data structures and algorithms to extend the Yap Prolog system to support tabling. Due to its importance for this thesis development, the *table space* is presented in a section of its own. The final section introduces the special set of mechanisms used by YapTab to efficiently handle incomplete and complete tabling. The *least recently used* algorithm is presented and its applications are discussed as a motivation for the present work.

3.1 Extending YAP to Support Tabling

YapTab is a WAM based tabling computational model that can be integrated with an or-parallel component. The main goal of this project is to achieve parallel high performance while developing a sequential tabling implementation that runs as fast as the current available sequential systems, if not faster. The original design of YapTab

evaluates definite programs only.

Inspired by the original SLG-WAM [Sag96] implementation, YapTab inherits most of its characteristics: it extends the WAM with a new data area, the *table space*; a new set of registers, the *freeze registers*; a new area on the standard trail, *the forward trail*; and support for the four main tabling operations: *tabled subgoal call*, *new answer*, *answer resolution* and *completion*.

A special mechanism to *suspend* and *resume* the computation of subgoals is implemented, and several *scheduling strategies* are provided to support the decision on which operation to perform upon suspension. In particular, YapTab implements two tabling scheduling strategies, batched and local [FSW96]. The observations of Freire and Warren [FW97] about the usefulness of resorting to multiple strategies within the same evaluation are taken under consideration. A novel approach, supporting a dynamic alternation between batched and local scheduling, enables the run-time change of strategy to resolve the subsequent calls to a tabled predicate [RSC05].

3.1.1 Differences with the SLG-WAM

The SLG-WAM abstract machine was first implemented in the XSB system engine. The data structures, data areas, instructions set and algorithms used by this abstract machine to evaluate definite programs are described thoroughly in [SW94], while extensions to handle normal logic programs according to the well-founded semantics are discussed in [Sag96, SSW96, SS98].

The major differences between SLG-WAM and YapTab designs are related to potential sources of overheads in a parallel environment, namely the data structures and algorithms used to control the process of leader detection and scheduling of unconsumed answers.

The SLG-WAM design performs leader detection at the level of the data structures corresponding to the first calls to tabled subgoals. For that reason, whenever a new generator node is allocated, a new *completion frame* is assigned to it [SS98]. This frame, kept in the *completion stack*, maintains a unique *depth-first number* (DFN) and a reference to the oldest subgoal upon which the newly called subgoal may depend on. The DFN is updated whenever variant calls or completion checks are made. When

no dependencies are found, the subgoal is considered to be the leader and completion may be performed. The completion frames are popped off the completion stack when completion operation succeeds.

YapTab performs leader detection at the level of data structures corresponding to variant calls to tabled subgoals. Whenever a new consumer node is allocated, a new *dependency frame* [Roc01] is assigned to it. Each frame, kept in the *dependency space*, holds a reference to the leader node of the SCC to which the consumer node belongs to. All dependency frames are linked together to form a dependency list of consumer nodes. The leader node is computed traversing this list between the new consumer and its generator. When completion is performed, all dependency frames younger than the one assigned to the leader node are removed from the dependency space.

Dependency frames introduce some advantages to YapTab. Firstly, they may be used to efficiently detect completion points and jump between consumer nodes with unconsumed answers. Secondly, a separate completion stack becomes unnecessary. Thirdly, they can hold a significant number of extra fields that otherwise would need to be placed in the tabled choice points. At last, dependency frames allow the integration of YapTab with the parallel computational model of OPTYap. To benefit from these advantages, all the algorithms related with suspension, resumption and completion were redesigned.

3.1.2 Basic SLG Evaluation Structures and Operations

Section 2.2.1 provided a formal definition of the search space for the SLG resolution. In synthesis, the SLG-trees are built using different types of nodes, allocated accordingly to the nature of the called subgoal:

Interior nodes correspond to non-tabled predicates. These nodes are evaluated by standard SLD-evaluation;

Generator nodes correspond to first calls to tabled predicates. Program clause resolution produces their answers;

Consumer nodes correspond to variant calls to tabled predicates. Their answers are taken from the table space.

All nodes, may they be interiors, generators or consumers, are implemented as WAM choice points at engine level. The first type requires no special modification, since it handles normal non-tabled subgoal calls. The last two types, however, require the introduction of extra fields. Figure 3.1 illustrates these structures. Both generator and consumer nodes include a pointer to a dependency frame. The generator nodes include an extra pointer to a fundamental building block of predicate tables, the subgoal frame, an important data structure that is later explained, in subsection 3.2.2, in the context of table organization.

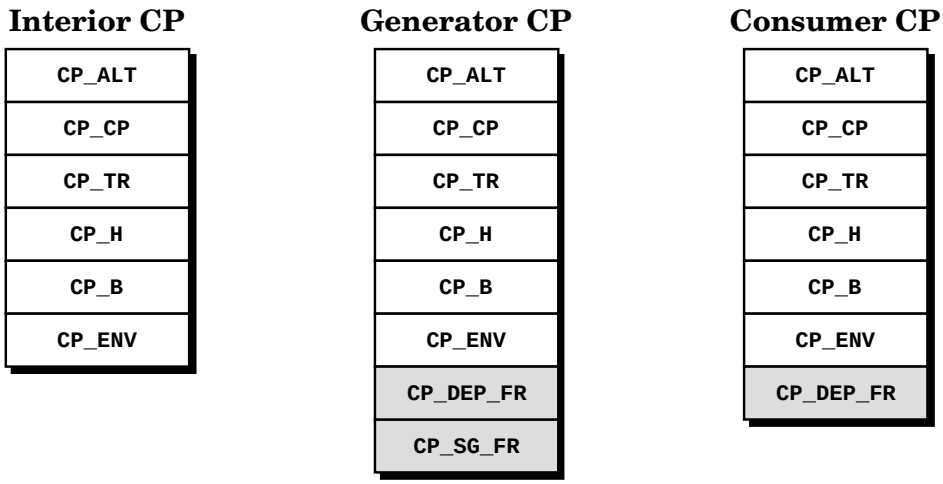

Figure 3.1: Structure of interior, generator, and consumer choice points

For definite programs, four main types of operations may be performed to operate over the search and table spaces:

Tabled Subgoal Call checks for the presence of a subgoal in the table. For the first call ever, it allocates a new generator node and inserts the subgoal into the table. For subsequent calls, it allocates a consumer node and starts consuming the available answers;

New Answer returns a new answer to a generator node. Each newly generated answer is looked-up in the table and, if not present, it is inserted into that data structure. Otherwise, the operation fails, thus avoiding unnecessary computations and, in some cases, looping;

Answer Resolution returns answers for a consumer node. Whenever such a node is reached, possible newly found answers are looked-up in the table and, if any

are available, the next one is consumed. Otherwise, the operation *suspends* the current computation and schedules a possible resolution to continue the execution. Answers are consumed in the same order they are inserted into the table;

Completion determines whether a tabled subgoal is *completely evaluated*, i.e., if the full set of resolutions for the subgoal is already found. In that case, the subgoal's table may be closed and the related stack positions reclaimed. Otherwise, a possible resolution must be scheduled on order for execution to proceed. This operation occurs when a generator node is reached by backtracking and all of its clauses have been already tried.

Although a crucial operation for tabling, completion may become a difficult task when several subgoals are mutually dependent, i.e., when subgoals depend upon each other for evaluation purposes and therefore must be completed together. In such cases, the subgoals form what is denominated a *strongly connected component* (SCC), where the oldest is called the *leader* of the SCC.

3.1.3 The Suspension/Resumption Mechanism

Performing tabled evaluation consists of a sequence of sub-computations that suspend and resume. The tabling engine must then be able to preserve the execution environments of the suspended sub-computations, so that they can be later recovered whenever evaluation is resumed.

The introduction of a new kind of registers, the *freeze registers*, enables YapTab to *suspend* the computation environments of incomplete tabled subgoals whenever its consumer nodes exhaust all available answers. A set of these registers is adjusted to conserve the tops of each one of the WAM stacks (local, heap and trail), preventing the elimination of all data belonging to suspended branches up to the time of their completion. A significant improvement can be achieved if freeze registers are updated whenever a new consumer choice point is allocated, since this guarantees that **(i)** the stacks are already protected by the time of the consumer's suspension and **(ii)** the adjustment of the freeze registers happens only once.

Upon completion, the frozen space is released by adjusting the freeze registers to the

top stack values stored in the youngest consumer choice point that is younger than the current completion choice point. The reference to the consumer node is kept in the top dependency frame, pointed by the `TOP_DF` register.

In some situations, the current top stack registers may point to positions older than the ones pointed by the freeze registers. To guarantee that the frozen segments are secured, the engine must guarantee that the next stored data is always placed at the younger position of both registers. By default, this is accomplished by **(i)** comparing the two registers and determining the youngest - this is the case for the local stack; **(ii)** ensuring that the top stack register is always the youngest of the two - this is the case for the heap and trail stacks.

An extension of the standard trail, the *forward trail*, is used to restore the binding values of the suspended branch before the execution is resumed. Each trail entry consists of two fields: the first one stores the address of the trailed variable; the second stores the value that is to be later restored into the variable. The chain between the frozen segments is maintained through an extension to the YAP trail, enabling it to store extra information beyond the standard variable trailing, in a similar way to that used to handle dynamic predicates and multi-assignment variables.

3.1.4 Scheduling Strategies

After suspension, one of several *scheduling strategies* may be applied to decide which operation to perform next. In a sequential system, this choice has an important influence both in the architecture and the performance of the abstract machine [FSW96, FSW97]. YapTab implements *Batched Scheduling* and *Local Scheduling* [FSW96] strategies.

Batched scheduling organizes clause execution in a WAM-like manner. It does so by favoring, in decreasing order of importance, forward execution, backtracking, consuming answers and completion. Newly found answers are stored in the tabled space and execution proceeds. New dependencies with older subgoals may contribute to enlarge the SCC, in which case the completion point must be delayed to an older generator node. Upon backtracking, the nature of the next action to be performed is determined accordingly to the following rules: **(i)** if a generator or interior node with available alternatives is reached, the next program clause is taken; **(ii)** if a consumer node is

reached, the next unconsumed answer is taken from the table space; **(iii)** whenever no available alternatives are found, verify if the reached node is the leader of the SCC. In this case, a check for completion must be performed. Otherwise, and if no unconsumed answers are available, simply backtrack to the previous node on the current branch.

Local scheduling tries to evaluate subgoals in the most possible independent way, processing one SCC at a time. Newly found answers are stored in the tabled space and execution *fails*, leading to the complete exploration of the SCC before returning any computed answers. Thus, the delivery of answers to the environment of the current SCC is delayed until all program clauses involving the corresponding subgoals are resolved. Due to the early completion of subgoals, less dependencies between subgoals are expected.

None of these two strategies may be considered better than the other. Batched scheduling provides immediate propagation of variable bindings, but is prone to the appearance to complex SCC dependencies. Conversely, local scheduling avoids dependencies most of the times but, lacks variable binding propagation. This last fact leads to significant overhead when returning the answers from the table, which in turn makes local scheduling slower, both in SLG-WAM [FSW96] and YapTab [RSS00].

3.1.5 Completion

Completion is required to recover space, and, in non-definite programs, to support negation. Before applying this operation on the leader of an SCC, the scheduler must ensure that all answers have been returned to all consumer subgoals in the SCC. Two essential algorithms, the *fixpoint check procedure* and *incremental completion*, are used for this purpose. The detailed description of these algorithms' implementation for the original SLG-WAM may be found in [Sag96, SS98]. For YapTab, these algorithms are implemented as follow.

The *fix-point check procedure*, an iterative process defined at engine level, is executed whenever execution backtracks to a generator choice point. It begins by verifying if the reached choice point is the leader of the SCC. In that case, the dependency frames associated with the younger consumer nodes in the SCC are traversed bottom-up to look for unconsumed answers. Every frame matching this condition causes the resumption of execution to the corresponding consumer node. When no more uncon-

sumed answers are found, a *fixpoint* is reached and completion may be performed. This includes marking all subgoals in the SCC as completed and the deallocation of the younger dependency frames. When the reached choice point is not the leader, the taken action depends on the selected scheduling strategy. In local mode, the procedure consumes all available answers for the choice point. In batched mode, it simply backtracks to the previous node.

The *incremental completion* mechanism, first appearing in [CSW95], reclaims the stack space occupied by subgoal sets when their evaluation is considered to be complete, rather than waiting for all subgoals to complete. By cleaning up the stack in this way, the mechanism increases the efficiency of the tabling engine in terms of memory space and consequently it improves its effectiveness on large programs.

It is possible to further improve the performance of incremental completion by introducing a new design feature, the *completed table optimization*. This technique avoids the allocation of a consumer node for repeated calls to a completely evaluated subgoal. Instead, whenever such calls occur, a modified interior node is allocated to consume the set of found answers executing compiled code directly from the subgoal's table data structures [RRS+95, RRS+99].

3.1.6 Instruction Set for Tabling

The new set of instructions used to support the tabling operations is based on the WAM instruction set.

The `table_try_me` instruction extends the WAM's `try_me` instruction to support the tabled subgoal call operation. It is called in three different situations - the subgoal at hand corresponds to **(i)** the first call to the subgoal; **(ii)** a variant call to the incomplete subgoal; **(iii)** a variant call to the completed subgoal. In the first case, the called subgoal is inserted into the table space, a new generator choice point and a new dependency frame are pushed, respectively, onto the local stack and onto the dependency space, and execution proceeds by executing the next compiled instruction. In second case, a consumer choice point is allocated and control proceeds to the `answer_resolution` instruction, which starts to consume the already available answers. The third and last case leads to the implementation of the completed table optimization and consequent execution of compiled code directly from the subgoal's

table data structures.

The `table_retry_me` and `table_trust_me` instructons differ from their WAM's `retry_me` and `retry_me` relatives because they always restore a generator choice point, rather than an interior (WAM-style) choice point. These two tabling instructions terminate differently. While `table_retry_me` continues to the next compiled instruction, `table_trust_me` proceeds to the `completion` instruction.

As in the WAM case, a small optimization may be introduced when tabled predicates are defined by a single clause. In such cases, clauses are compiled using the `table_try_me_single` instruction. Similarly to the `table_trust_me` instruction, this instruction is followed by the `completion` instruction.

The `new_answer` instruction implements the new answer operation. The instruction is responsible for the introduction of new answers into the tabled space. Since this is the final instruction produced by the compiler, by the time it is, all variables occurinng in the body of the clause have already been instantiated. As a consequence, the binding substitution that identifies the answer for the subgoal can be obtained by de-referencing the arguments. If the answer is not yet found in the table space, it is inserted by allocating the required data structures. Otherwise, the operation fails. As previously seen, the final result depends on the adopted scheduling strategy: for batched mode, the instruction succeeds and execution proceeds to the next instruction; for local mode, the instruction fails.

Another new instruction, `answer_resolution`, is responsible for guaranteeing that all answers are given to each variant subgoal once and only once. The answer resolution operation gets executed through backtracking or through direct failure to a consumer node in the fix-point check procedure. The instruction starts by looking for unconsumed answers. If any are found, the next one is recovered from the table space and execution proceeds. Otherwise, it schedules a backtracking node.

The `completion` instruction implements the fix-point check procedure. It takes place when execution backtracks to a generator leader node that has exhausted all of its clauses.

Figure 3.2 shows the resulting compiled code for the two clauses of the tabled predicate `path/2`, introduced in the example program appearing in Fig. 2.3. Since this predicate is defined by several clauses, a `table_try_me` instruction is placed at the beginning

of the code, taking as argument the label pointing at the start of the second clause. This, in turn, is the last clause for `path/2`, so its code begins with a `table_trust_me` instruction. Both clauses are coded in the usual WAM fashion for the head and body subgoals, but a `new_answer` instruction closes each block.

```
path/2_1:
    table_try_me path/2_2   // path
    get_variable Y1, A2     //     (X,Z) :-
    put_variable Y2, A2     //               arc(X,Y
    call         arc/2      //                      ),
    put_value    Y2, A1     //                         path(Y,
    put_value    Y1, A2     //                               Z
    call         path/2     //                                )
    new_answer              //                                 .

path/2_2:
    table_trust_me          // path(X,Z) :-
    call         arc/2      //               arc(X,Z)
    new_answer              //                       .
```

Figure 3.2: Compiled code for a tabled predicate

3.2 The Table Space

YapTab includes a proper space for tables. The *table space* can be accessed in a number of ways: **(i)** to lookup a subgoal in the table and if not found insert it; **(ii)** to verify whether a newly found answer is already in the table and, if not, insert it; and **(iii)** to load answers to variant subgoals. The performance of these operations largely depends on the implementation of the table itself; being called upon very often, fast look up and insertion capabilities are mandatory. Millions of different calls can be made, hence compactness is also required. The YapTab table space is implemented using *tries*, as suggested by Ramakrishnan *et al.* [RRS+95, RRS+99]. The authors proposed the tabling trie [RRS+95] as a variant of the discrimination net [BCR93] formalism introduced by Bachmair *et al.*. Given a set $\mathcal{T}$ of terms, tries can be used to produce a partition accordingly to the terms' structure, thus enabling the definition of efficient lookup and insertion routines. A tabling trie is a tree-structured automaton in which the root represents the start node and the leaves correspond to the terms in $\mathcal{T}$. Each internal state specifies a symbol to be inspected in the input term when reaching that state. The outgoing transactions specify the function symbols expected at that position. A transaction is taken if the symbol in the input term at that position

matches the symbol on the transition. On reaching a leaf state the input term is said to *match* the term associated with the leaf. The root-to-leaf path corresponds to a pre-order traversal of a term in $\mathcal{T}$.

3.2.1 Tabling Trie Properties

Ramakrishnan *et al.* enumerates the main advantages of using tries to implement tabling in [RRS+99], from which we borrow some important topics. Please refer to that paper for further explanation on the subject.

One of the most important properties of tries is their *compactness*. Root-to-leaf paths branch off each other at the first distinguishing symbol, assuring that common prefixes for represented terms are stored only once, while guaranteeing a *complete discrimination* for all terms. Clearly, the number of terms with common prefixes greatly influences the compactness of a tabling tree.

Let us use Fig. 3.3 to demonstrate this property. Initially, the trie contains only the root node. Then, as the `t(X,a)` term is inserted, three nodes are generated: one for the functor `t/2`, one for the variable `X` and one for the constant `a` (Fig. 3.3(a)). Next, the `u(X,b,Y)` term is inserted. Since the inserted terms do not share common prefixes, four new nodes are inserted into the trie (Fig. 3.3(b)). At last, the `t(Y,1)` term is inserted. Now, this term shares two nodes with the first inserted term, namely, the functor and variable nodes. Hence, only the last sub-term, the constant 1, requires a new node(Fig. 3.3(c)).

Notice that variables are stored following the formalism proposed by Bachmair *et al.*. Each variable is represented as a distinct constant [BCR93]. Formally, this corresponds to a bijection `numbervar()`, defined from the set of variables in a term to the sequence of constants `VAR0`,...,`VARN`, such that `numbervar(X)` $<$ `numbervar(Y)` if `X` is encountered before `Y` in the left-to-right traversal of a term. This standardisation is performed while a term is being inserted in a trie, thus allowing the detection of terms that are the same up to variable renaming.

A second important property is the *efficiency* of the access algorithms. For an insert or check operation, a single traversal of the input term is required. Let us see why this is so.

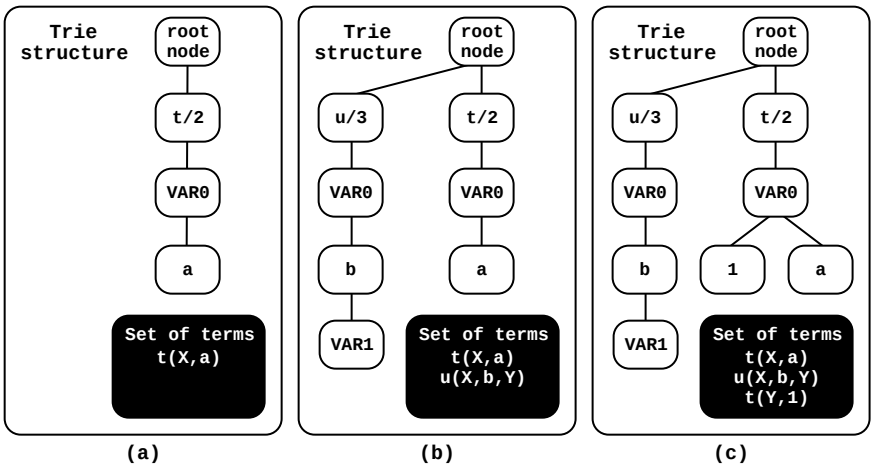

Figure 3.3: Using tries to represent terms

When checking for the presence of a term in a trie, a transition is taken every time the next element of the input term matches one of the outgoing function symbols of the current node. If a leaf node is reached, the checking succeeds, otherwise the checking operation fails at the state where no outgoing transition may be taken.

Inserting a new term requires a check for its presence first, but the mode in which the trie is structured enables the collapsing of the two steps. Whenever the new input term shares a common prefix with a previously stored one, the checking routine fails at the point of differentiation. A new node representing the current input term is then inserted into the trie and an outgoing transaction to it is added to the node where the point of failure was reached. The remaining input terms are stored in the same manner, and when the last input term is reached, its corresponding node becomes a leaf. In the worst-case scenario, the search fails at the root node and the insertion requires the allocation of as many nodes as needed to represent the entire path. Conversely, inserting repeated terms into the table requires no node allocation at all; in fact, this resumes to a check traversal.

In all the presented cases, the internal nodes of the traversed paths are visited only once. Furthermore, the two operations may be merged without changing their complexity. Usually, the merger is referred to as the *insert/check* function.

3.2.2 Table Organization

In YapTab, tables are implemented using two levels of tries, one for subgoal calls, other for computed answers. This separation, called *substitution factoring*, promotes the data structure compactness and improves the efficiency of the access methods, therefore introducing minimal impact on the engine performance. In fact, Ramakrishnan states in [RRS+95, RRS+99] that for any given subgoal $\mathcal{G}$ and an answer $\mathcal{A}$ for $\mathcal{G}$, the use of substitution factoring guarantees the execution of the *insert/check* and answer backtracking routines in time linear to the proportional size of the answer substitution of $\mathcal{A}$. Figure 3.4 illustrates the concept.

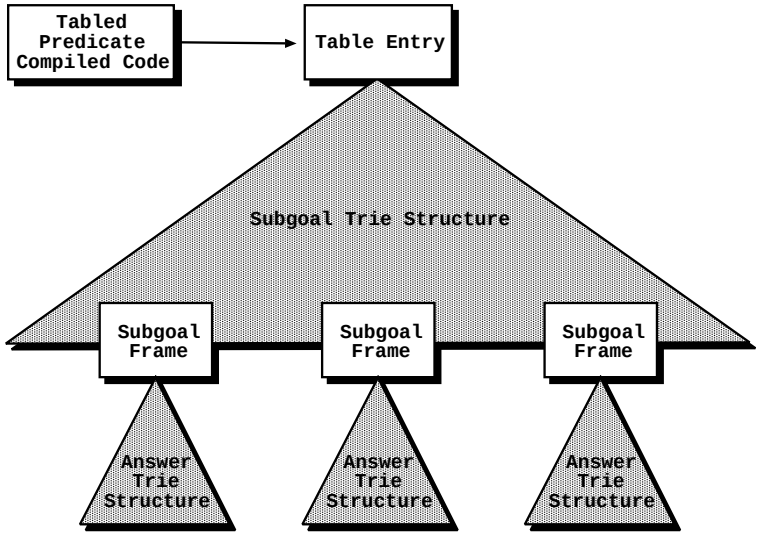

Figure 3.4: Using tries to organize the table space

YapTab's table space is thus organized in the following way. Each tabled predicate has a *table entry* data structure assigned and inserted into the table space. This structure acts as the entry point for the *subgoal trie*. Each unique path in this trie represents a different subgoal call, with the argument terms being stored within the *subgoal trie nodes*. The path ends when a *subgoal frame* data structure is reached. This structure, in turn, acts as an entry point to the *answer trie* and holds additional information about the subgoal. Each unique path in this trie represents a different answer, with the substituting factors for the free variables in the called subgoal being kept inside the *answer trie nodes*.

Figure 3.5 exhibits the details of the two main data structures used to control the flow of a tabled computation: the *trie node* and the *subgoal frame*.

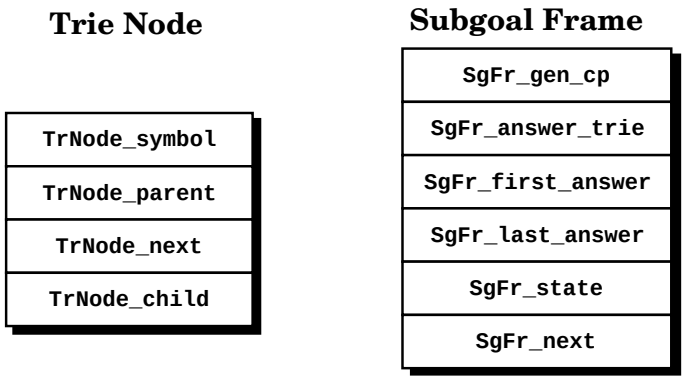

Figure 3.5: Structure of trie nodes and subgoal frames

The *subgoal frame* structure is divided into six fields: `SgFr_gen_cp` is a back pointer to the corresponding generator choice point; `SgFr_answer_trie` points at the top answer trie node; `SgFr_first_answer` points at the leaf answer trie node of the first available answer; `SgFr_last_answer` points at the leaf answer trie node of the last available answer; `SgFr_state` is a flag that indicates the current state of subgoal; `SgFr_next` points at the next subgoal frame.

The subgoal state changes throughout evaluation. Its state is said to be *ready* when no generator choice point is assigned to the subgoal frame; *evaluating* when an assigned generator choice point is being evaluated; and *complete* when the generator is fully evaluated and consequently removed from the execution stacks.

The answer trie leaf nodes are chained as a linked list. Leafs are linked in order of insertion so that recovery may happen the same way; this is done using their `TrNode_child` field. The subgoal frame `SgFr_first_answer` and `SgFr_last_answer` fields are set to point, respectively, at the first and last answers of this list; in particular, the second pointer is updated at each new answer insertion. This chain guarantees that, starting from the answer pointed by `SgFr_first_answer` and following the leaf node links, `SgFr_last_answer` is reached once and only once. Consumer nodes can then use the established chain to guarantee that no answer is skipped or consumed twice. This may be done with the help of a pointer to the next unconsumed answer, kept inside the associated dependency frame. A variant subgoal call can retrieve the already computed answers following the chain in a single direction, updating the next unconsumed answer pointer as it goes along. Each answer may be loaded traversing the answer trie in a bottom-up order until the subgoal frame is reached.

When performing completion, YapTab uses the `SgFr_next` pointer to reach the subgoal frame for the youngest generator older than the current choice point. The `TOP_SG_FR` register may be used to access the subgoal frames chain, since it points at the youngest subgoal frame.

The *trie node* consists of four fields: `TrNode_symbol` stores the YAP term assigned to the node; `TrNode_parent` stores the memory address of the node's parent; `TrNode_child` stores the memory address of the node's first child; `TrNode_next` stores the memory address of the immediate sibling node.

The presence of all these pointers in the trie node structure enables a powerful trie navigation method. Given a particular node, the complete list of outgoing transitions can be determined by a simple iteration: make the child node the current node and consider its term field the first possible transaction symbol; while the current node's sibling pointer is a valid address, make it the current node and consider its term field the next alternative transaction symbol.

The `TrNode_symbol` field is destined to hold the subgoal's sub-terms. As previously observed, in subsection 2.1.1, logical terms may assume a large variety of forms, such as variables, numeric values of several precision ranges, character strings, or compound terms like pairs and applications. However different in size and nature, YAP is required to handle all of these types uniformly.

Internally, YAP terms are 32 or 64-bit structures divided into two main sets, *mask* and *non-mask* bits, used respectively to distinguish among the several types of terms and to store the actual primitive value for the term. Obviously, the mask bits cannot be used for data storage purposes, which means that the available space for the non-mask bits may be smaller than the one required for the usual underlying representation of primitive values. The rule of thumb is then to place the primitive value into the term's non-mask slot if it fits, otherwise keep the primitive value somewhere else in memory and store that memory address into the non-mask slot of the term. This is the case, for instance, of atom terms, whose primitive value is kept in the symbols table.

However, the indirection schema is not very useful in the case of long numeric primitives[1]. In fact, YAP splits these primitive values and places the pieces in consecutive

[1]Floating-point numbers, for instance, usually require twice the physical space used by integer numbers.

stack cells. YapTab's *insert/check* function mimics this behaviour in order to simplify the transference of data between the tries and the stack. Two possible situations are contemplated: if the numeric term is relatively small, a trie node is allocated to hold it, otherwise, the term's primitive value is split into pieces small enough to fit into a generic YAP term and a sufficient number of trie nodes is allocated to store the pieces; additional special markers are placed into the trie branch to delimit the data area.

Figure 3.6 illustrates how the presented data structures are used to build and navigate a tabling trie. At the bottom, the figure displays the state of the predicate table `f/2` after the execution of the tabled instructions presented in the top box.

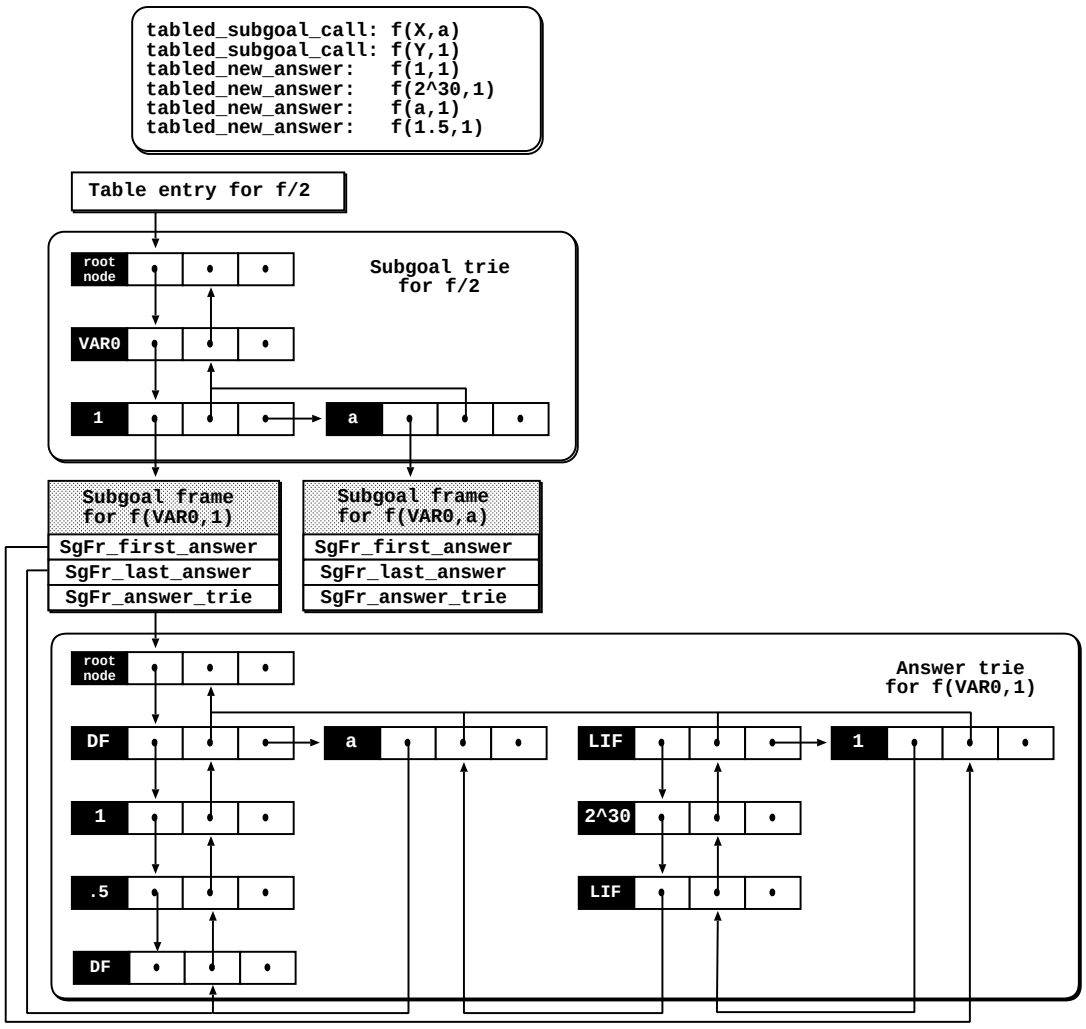

Figure 3.6: Table structures for f/2

Initially, when the predicate is compiled, the *table entry* is inserted in the table space and a new *subgoal trie* is created containing only the root node. When subgoal `f(X,a)`

is called, two internal nodes are inserted: one for the variable `X`, and a second last for the constant `a`. The subgoal frame is inserted as a leaf, waiting for the answers to appear. Then, subgoal `f(Y,1)` is inserted. It shares one common node with `f(X,a)`, but the second argument is different so another subgoal frame needs to be created. Next, answers for `f(Y,1)` are stored in the answer trie as their values are computed. Notice how the different primitives values are stored in the answer trie, as previously discussed. The first and third terms, respectively an integer and an atom, easily fit in a single trie node. On the other hand, the second and fourth terms, respectively a 32-bit integer and a 64-bit double, exceed the available space, hence they are split and stored in multiple nodes. Recall that these terms must each be surrounded by two additional special markers to delimit their true value; the marker nodes are tagged with the *long int functor* (LIF) and the *double functor*(DF).

3.3 Efficient Support for Tabling

The increasing interest and research on tabling execution models has resulted in exciting proposals for practical improvements. Sagonas and Stuckey proposed the *just enough tabling* (*JET*) mechanism [SS04], introducing the possibility to arbitrarily suspend and resume a tabled evaluation without the need for full recomputation. Saha and Ramakrishnan proposed in [SR05] an incremental evaluation algorithm to freshen the tables in case of addition or deletion of facts and/or rules that avoids the recomputation of the full set of answers. Rocha *et al.* [RSC05] proposed the ability to support a mixed-strategy evaluation, using both batched and local scheduling, that may change dynamically in run-time. This last proposal has been implemented in the YapTab engine, hence this is a relevant topic to be further explored in this thesis context.

The choice of a scheduling strategy has a direct influence in the efficiency of a tabled evaluation. In fact, Rocha suggests that the best performance can be achieved when batched or local scheduling strategies are chosen accordingly to the nature of the subgoals to be evaluated [RSC05]. In YapTab, batched scheduling is the default strategy; program clauses are scheduled in a depth-first manner, favoring *forward execution* and automatically propagating new answers as they are found. Local scheduling, on the other hand, tries to *force* completion before returning any computed answers. This

eliminates complex dependencies within the SCC, albeit it may result in an inglorious effort: the table may be scarcely used or a small initial subset of answers may be sufficient for a subgoal evaluation to succeed.

For some applications, the previous discussion is of the highest relevance. The decision process becomes crucial when pruning operations over the tabled predicates are introduced. This potentiates the appearance of *incomplete tables* when the full set of answers is not yet determined by the time the pruning takes place. By removing the computational state of the pruned subgoals from the execution stacks, the already found answers become unreliable to subsequent variant calls, since information is no longer protected and may have been overwritten meanwhile. The use of local scheduling may sound appealing, since it avoids incomplete tabling. However, as explain before, this might not be adequate.

Memory exhaustion may occur when applications produce large table spaces, possibly with many large tables resulting from sub-computations. In such situations, the only way to continue execution is to arbitrarily remove some tables, hopefully the ones least used. Most tabling systems allow the programmer to arbitrarily remove tables from memory through calls to specialized built-in control predicates.

The remainder of this section introduces YapTab's approach for the presented problems. Incomplete tabling is addressed first, to introduce important concepts. Memory recovery is addressed next.

3.3.1 Incomplete Tabling

Most systems handle incomplete tables in the simplest way possible: they are thrown away and recomputed every time. YapTab implements by default a different approach, keeping the incomplete tables for pruned subgoals. A later variant call starts by consuming the incomplete table, with recomputation occurring if and only if the previously computed answers have already been consumed. If the subgoal is pruned again, the process repeats itself and, eventually, a complete evaluation is reached. The purpose of this procedure is to avoid recomputation when the stored answers are sufficient to evaluate a variant call.

To support incomplete tables, the subgoal frame structure must be altered. A new

state, *incomplete*, is made available to signal subgoals whose corresponding generator choice points were pruned from the stacks. To guarantee that no answer is ignored or consumed twice when evaluating repeated calls to pruned subgoals, a new `SgFr_try_answer` field is added to hold a reference to the currently loaded answer.

The `tabled_subgoal_call` must also be transformed to handle the new state. When this instruction is called and the subgoal is marked as incomplete, a new generator node is allocated. However, instead of using the program clauses to evaluate the subgoal call, the new generator loads the first available answer from the incomplete table and places a reference to it in the `SgFr_try_answer` field. When the generator is reached again by backtracking, a variant of the answer resolution operation is called. This variant checks if any unconsumed answers remain in the incomplete table, and if so, loads the next one and updates the `SgFr_try_answer` field. When all answers have been loaded, the computation of the table is restarted from the beginning.

An important optimization may take place for subgoals with no answers. In such cases, the subgoal frame is marked as ready instead of incomplete. When variant calls appear, execution resumes as it was a first call rather than passing through the described mechanism.

The use of generator points to implement the calls to incomplete tables enables the reutilization of all data structures and algorithms used by the tabling engine without major changes safe from the pointed out cases. At the engine level, when these generators are placed into the execution stacks, they are regarded as first calls to the subgoal since the previous representation has been pruned.

The described mechanism is similar to the one used in the *just enough tabling*. The JET proposal uses auxiliary memory to store a copy of the segments pruned from the execution stacks; upon latter calls, these copies are restored to resume evaluation. Compared to JET, the YapTab approach requires no extra data areas nor does it introduce any overhead to the pruning process.

3.3.2 Memory Recovery

Most tabling engines provide a set of built-in primitives to remove tabled subgoals from the tabled space, and YapTab is no exception. However, YapTab provides a

more robust approach with the *least recently used* algorithm, a memory management strategy that dynamically recovers the space allocated to the least recently tables when memory is exhausted. Programmers may thus delegate the choice of which tables to delete on the memory management system.

Table elimination must be performed guaranteeing that no errors are introduced into the evaluation and that no relevant information is lost in the process. To avoid tampering with the SLG-evaluation search space, the algorithm must determine if the candidate subgoal table is represented in the execution stacks. Such presence may be in the form of **(i)** a generator choice point allocated to handle either a first call to the tabled subgoal or a repeated call after pruning; **(ii)** any consumer choice points allocated to handle variant calls to the tabled subgoal; **(iii)** any interior nodes allocated to handle a completed table optimisation. In what regards to the search space, a subgoal then is said to be *active* if it is represented in the executions stacks. Otherwise, it is said to be *inactive*. Inactive subgoals are thus solely represented in the table space [Roc07].

In the table space, the state of a subgoal may be *ready*, *evaluating*, *complete* or *incomplete*. Subgoals in the ready and incomplete states are inactive, while subgoals in the evaluating state are active. Subgoals in the complete state may be either active or inactive. A new state, *complete-active*, is introduced to signal active completed subgoal calls. The former *complete* state now indicates that the subgoal is inactive. This change enables the distinction of the two cases without changing the subgoal frame implementation.

When the system runs out of memory, the *least recently used* algorithm is called to recover some of the space allocated to inactive subgoals. Figure 3.7 illustrates the process. Subgoal frames corresponding to inactive subgoals are chained in a double linked list using two subgoal frame fields, `SgFr_next` and `SgFr_previous`. Two new global registers, `Inact_most` and `Inact_recover`, point respectively to the most and least recently inactive subgoal frames. The algorithm starts at the last register and navigates through the linked subgoal frames, using the `SgFr_next` field, until a page of memory can be recovered. Only the tables that store at least one answer are considered for space recovery (completed nodes with a *yes/no* answer are ignored), and for these only the answer trie space is recovered. Rocha argues that for a large number of applications, these structures consume more that 99% of the table space [Roc07].

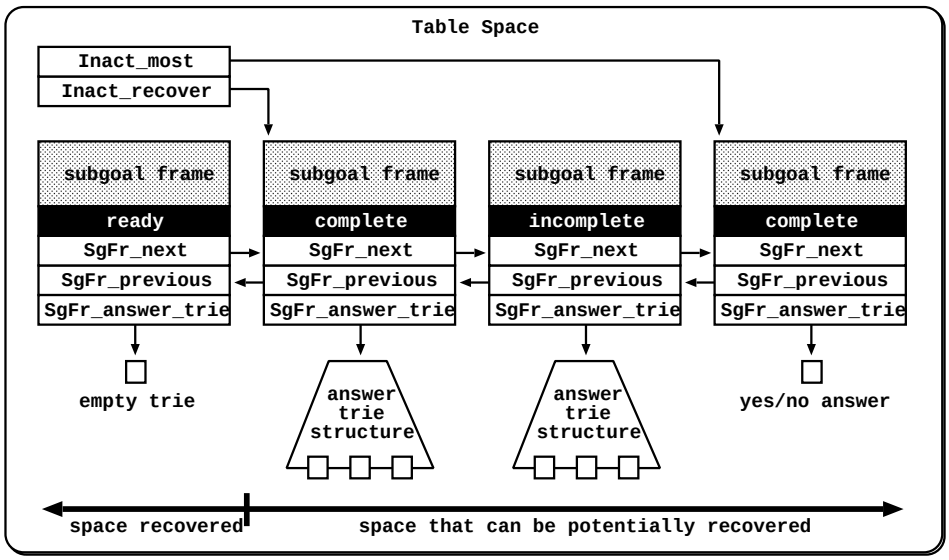

Figure 3.7: Recovering inactive tries space

During execution, the inactive list is updated whenever subgoals become inactive by executing completion, being pruned or failing from internal nodes that have just finished to execute compiled code from the answer table. In the latter case, the space allocated to the subgoal table can only be recovered when all interior nodes in such conditions are made inactive. YapTab uses the trail to detect this situation. On the other hand, every inactive subgoal that is activated by a repeated call must be removed from the inactive list. The `tabled_subgoal_call` is transformed to implement this state transition.

Despite the improvement introduced to memory management by the *least used algorithm*, situations may arise when the space required by the set of active subgoals exceeds the available memory space and the algorithm is incapable of recovering any space from the set of inactive subgoals. In such cases, Rocha suggests the resource to an external storage media, such as a relational database, to store the already found data before removing it from memory. When later required, tables can be loaded from this external storage avoiding possible recomputation overhead. The same situation may occur for inactive subgoals and, in particular, the memory management algorithm may be used to decide what tables to store externally, rather than eliminating them. Rocha's suggestion motivates the work developed in this thesis ambit. The remaining chapters present the proposed relational models, the implementation of DBTab and the discussion of gathered results.

3.4 Chapter Summary

This chapter introduced the YapTab engine, an extension to the YAP Prolog system to support sequential tabling. It started by briefly presenting the YapTab tabling engine and comparing it to the SLG_WAM as first implemented in the XSB system. The focus was then moved to its key aspects, namely, a novel data structure, the dependency frame, and the new completion detection algorithm based on it. It was explained how YapTab innovates by considering that the control of leader detection and scheduling of unconsumed answers should be done at the level of the data structures corresponding to variant calls to tabled subgoals.

The table space, the organization of tables and their internal components were presented next in a separate section, due to their importance to the remainder of this work. At last, some optimizations to the YapTab original implementation were described. These optimizations focused on the efficient handling of incomplete and complete tables. The *least recently used* algorithm was presented and its possible applications were discussed, in particular those which became the motivation for the present work.

Chapter 4

Relational Data Models for Tabling

This chapter begins by introducing the reasons why tabling may benefit from external memory storage mechanisms, such as relational databases. Two possible relational schemes are discussed, along with the required transactional algorithms, their advantages and disadvantages. Primitive value handling for YAP terms is also discussed, in both contexts. Finally, the last section discusses meta-data as a mean for databases schema partitioning.

4.1 Motivation

Although a powerful tool, tabling still has flaws. For some classes of problems, the table space becomes so big that it exhausts the main memory. This state of memory exhaustion disables the engine's ability to solve the proposed queries. Tabling engines generally address this problem by arbitrarily removing tables from the table space to recover some memory, thus allowing execution to proceed. However, this is not an adequate approach since the information kept in those tables may be required almost immediately after removal.

Arguably, Rocha's *least used algorithm* for memory recovery provides a safer and stronger approach [Roc07]. In YapTab, subgoal calls are classified as *active* or *inactive* according to their presence in the execution stacks. A linked list of the inactive subgoal frames is kept, ordered by time of inactivation, so that the algorithm may select the ancient unused answer tries with at least one answer for removal. If the algorithm

fails to find any candidate subgoals, there is no choice other than arbitrary removal. Rocha suggests that, for such situations, an external data media like a relational database could be used to dump the older tables before recovering their assigned memory pages [Roc07]. Later calls to the removed tables could be resolved by loading their externally stored representations into memory tries, thus enabling the established machinery to continue as if the tables were never removed.

Our proposal pursues this idea and considers the storage of all tabling tries chosen by the algorithm for removal. We consider that, for a certain class of problems, tabling tries should be constructed once and dumped to/recovered from auxiliary memory whenever required. This is expected to improve execution performance for the set of subgoals whose transaction and trie reconstruction times exceeds the complete reevaluation of previously found sets of answers.

4.2 Tries as Relations

Mapping tree-based data structures, such as tries, to relational database tables and vice-versa is a long known discussion topic in several research fields, from database systems [CA97, Gen03] and data retrieval languages [AU79, CH80] to a more recent area of XML storage and analysis [FK99, STZ+99, SKWW01, TVB+02, BFRS02, AYDF04].

Finding a solution to this problem constitutes an interesting challenge due to the difference between the two formalisms. The *tree-to-table* transformation faces its main difficulty in the fact that the hierarchical nature of tries is not well handled in the relational world [STZ+99, Gen03]. Conversely, the *table-to-tree* transformation is hard because ther is no intrinsic ordering for the tuples belonging to a relation [Cod70], thus the relation is required to include sufficient information within its attributes to assure that the correct hierarchy among nodes is preserved when the tree is built.

Florescu *et al.* developed some interesting and useful solutions [FK99] for a similar problem[1]. The authors suggested three methods to store and query tree-based

[1]The authors suggest that XML documents can be mapped to ordered direct graphs that largely resemble YapTab tabling tries in that: **(i)** both graphs start at single root node; **(ii)** each element is represented as a node in the graph; **(iii)** *parent-to-child* relationships are the edges in the graph;**(iv)**

structures using a relational database, two of which (denominated *Edge* and *Universal table*) resemble the ones proposed in the context of this thesis.

Regardless of the used relational representation, storing a tree structure into a relational schema requires a traversal of the entire set of branches. In what concerns to YapTab tabling tries, the information stored in the nodes enables two possible directions for branch traversal: *bottom-up* (from leaf to root) or *top-down* (from root to leaf). The best way to choose is to inquire the state of the associated subgoal in YapTab:

- whenever a subgoal is active and incomplete, new nodes may be inserted *anytime* and *anywhere* in an answer trie due to the asynchronous nature of the answers insertion and consumption operations. In this scenario, top-down control algorithms require complex implementations hence answer tries are usually traversed in a bottom-up manner, efficiently guaranteeing that all the nodes are visited; in particular, the ones in the vicinity of the root (included) more than once.
- when a subgoal becomes inactive, or completed, it is removed from the execution stacks as a consequence, hence the corresponding trie becomes immutable, enabling top-down algorithms. In particular, this is the mode in which the trie is navigated after the completed table optimization takes place. It may be used to efficiently guarantee that all the nodes are visited only once.

Constructing a tree from a relational table requires a correct ordering of the set of tuples. In the case of YapTab, this ordering is even more important, since it is expected that the trie branches are reconstructed and reordered as originally stored. From the set of tuples belonging to the relation, the reconstruction of the trie may be done either by creating the nodes and placing them one by one into the trie structure, or by reconstructing the stored terms and passing them to the *insert/check* function, letting it handle the creation and placing of nodes.

4.2.1 Tries as Sets of Subgoals

For any given subgoal, the corresponding table entries are used to collect its already known answers. The enumeration of all these answers may be regarded as a set of

the outgoing transactions of a node to another are maintained as edges in the graph.

ground predicate facts that are known to hold in the subgoal execution context. Since predicates are known to have a functor and a fixed set of subterms, they can be mapped to a fixed size structure like a database relation. This idea is not entirely new; in fact, as observed back in subsection 2.3.2, one may consider storing the set of answers *extensionally* as Datalog facts. From this point on, we shall denominate this model as the *Datalog model*.

The main idea is straightforward: each tabled predicate p of arity n (p/n for short) is mapped into a relation $\mathcal{P}n$ with **n** attributes

$$\mathcal{P}n(\underline{\arg_1} : term, \ \ldots, \ \underline{\arg_n} : term)$$

and each answer for p/n is mapped to a tuple belonging to $\mathcal{P}n$ in which the subterms of the predicate provide the values for the relation attributes. The uniqueness of each answer in the subgoal table is enforced by the association of all the $\texttt{arg}_k$ attributes to form the relation's *primary key*.

This proposal is somewhat *loose* in what comes to answer ordering itself. There is no guarantee that the relation tuples are recovered in the same order as they where created, hence the resulting predicate table may have its answers ordered differently than the original one. The solution lies in imposing an unique sequential identifier to the tuples as they are inserted. The $\mathcal{P}n$ relation has its set of attributes enlarged to

(I) $$\mathcal{P}n(\ \underline{\text{order}} : answer\_id, \arg_1 : term, \ \ldots, \arg_n : term)$$

where `order` becomes the *primary key* of the relation. For the intended purpose, the *answer_id* may be simply mapped to a relational INTEGER domain. The mapping between logical terms and relational domains is a big problem in the context of this relation and will be later addressed. For the time being, assume that all terms, regardless of their types, may be expressed in an universal representation.

In this mapping schema, storing the table that holds the information regarding the subgoal into the database relation becomes equivalent to that of enumerating the entire set of answers for a given subgoal. In the particular case of YapTab, this means that such an algorithm must browse through the respective answer trie while undoing the substitution factoring in the process. The pseudo-code for the storing algorithm is

shown in Fig. 4.1.

```
store(subgoal frame sg_fr) {
    answer id answer_id

    {TEMPLATE} = fetch_subgoal_terms(sg_fr) // #{TEMPLATE} = ARITY(sg_fr)

    answer = FIRST_ANSWER(sg_fr)
    while answer <> nil {
        answer_id = create_answer_id()
        {TERMS} = bind_answer_terms({TEMPLATE},answer)

        INSERT INTO Pn (ORDER,ARG1, ..., ARGn)
        VALUES (answer_id, TERM1, ..., TERMn)

        answer = CHILD(answer) // Next answer in insertion order
    }
}
```

Figure 4.1: Storing an answer trie flatly

Initially, the function begins to create a new array of terms, called the *template*, whose size matches the arity of the subgoal. To initialize this array, the subgoal trie branch is traversed from leaf to root, checking the value of the *term* field for each reached node. Whenever a bound term is found, a copy of its value is placed in the respective position of the template array; otherwise, the position is left blank, indicating the finding of a variable term. The attention is then turned to the answer trie, cycling through its branches in order to parse all known answers[2]. Again, each branch is traversed leaf to root and the term values stored inside the nodes *term* field are used to instantiate the remaining blank positions in the template array, thus producing the tuple that is to be stored into the relation.

The presented storing algorithm guarantees that

- a single bottom-up traversal of the subgoal branch is required to construct the template, a single bottom-up traversal of each answer branch is required to instantiate the template;
- navigating the answer trie branches in insertion order ensures that each branch is traversed only once;

[2]Recall that the subgoal frame includes two special pointers to the first and last known answers and that insertion order is preserved through the child pointer of the leaf nodes, as mentioned back in subsection 3.2.2

- a single insertion operation is performed for each tuple, conveying the entire set of its subterms;
- subterm order within the answer is maintained intrinsically by the relation's `order` attribute;
- the tuples are correctly ordered, thus preserving the correct ordering of branches when the trie is reconstructed.

An efficient branch-by-branch answer trie construction may be achieved simply by retrieving the necessary tuples from the relation in correct order and, while browsing through them, passing their attribute values to the *insert/check* function, letting the function handle the placement of the input terms within the answer trie structure. The algorithm, depicted in Fig. 4.2, has a straight-forward underlying rationale: minimize the effort, both in data transaction and trie reconstruction stages.

```
load(subgoal frame sg_fr) {

    <VARIABLES> = enum_variable_terms(sg_fr)
    <CONDITION> = enum_ground_terms(sg_fr)

    {T} = SELECT <VARIABLES>
          FROM Pn
          WHERE <CONDITION>
          ORDER BY Pn.ORDER

    foreach tuple in {T} {
        insert/check(CHILD(sg_fr), tuple.ARG1, ..., tuple.ARGn)
    }
}
```

Figure 4.2: Loading an answer trie flatly

Whenever a subgoal answer trie is to be reloaded from the database, a full retrieval of the mapping relation may not be appropriate. On one hand, the relation holds all the answers for a given predicate, so it is necessary to impose some sort of restriction over the relation so that the resulting tuple set is limited to those belonging to the called subgoal. On the other hand, the values of the attributes related with the ground terms within the subgoal (if any) are already known; in fact, they may be found in the subgoal trie branch. Therefore, one needs to fetch only those attributes related with unbound variables.

Following this line of thinking, the function starts by parsing the subgoal trie branch

to produce two SQL subexpressions[3]. The first, denoted as <VARIABLES>, enumerates all the $\mathcal{P}n.\text{arg}_i$, ..., $\mathcal{P}n.\text{arg}_j$ attributes of the relation associated with variable terms. The second, denoted as <CONDITION>, enumerates all of the restrictive conditions over ground terms. In practical terms, it consists of a list of $\mathcal{P}n.\text{arg}_k$ = CONSTANT subexpressions, meaning that the relation's kth attribute must hold a *constant* value. If some variables appear more than once in the subgoal call, the restrictive condition holds a list of $\mathcal{P}n.\text{arg}_m = \mathcal{P}n.\text{arg}_n$ subexpressions meaning that the mentioned attributes refer to the same variable terms.

The resulting expressions are incorporated in the query expression used to consult the database. If neither ground terms nor repeated variable terms appear in subgoal call, all of the attributes in the relation will be present in the variable list, hence the restriction makes no sense and may be omitted. Once the query is applied to the relation, the resulting tuple set is consumed sequentially, passing the attribute values in each tuple to the *insert/check* function that will, in turn, reconstruct the respective trie branch. Notice that the tuples are retrieved in insertion order.

This trie loading algorithm guarantees that

- the entire set of tuples is returned;
- a single tuple selection operation is performed for all answers, thus introducing minimal transactional overheads to YapTab performance;
- each trie branch is reconstructed in constant and minimal time[4];
- the correct ordering of answers has been preserved.

Figure 4.3 presents a practical use of the proposed relation schema and algorithms. The top right box contains a first set of instructions that is used to create and populate the presented trie. Again, assume that at some point in execution, the trie becomes inactive and that the `recover_space()` function is called after that point, as a consequence of a main-memory exhaustion situation. The storing algorithm is called to dump the inactive trie to the database. The main cycle of the storing algorithm

[3]The pseudo-code shows this step divided in two for clarity purposes, in fact this is done by a single instruction.

[4]Please refer to subsection 3.2.2

uses the subgoal frame information to visit the known answers in trie insertion order and initialize the SQL instruction arguments (box **a**). At some posterior point in time, a variant subgoal call occurs and the loading algorithm is called to reconstruct the trie from the tuples in the relation. The issued SELECT instruction (box **b**) retrieves only the first subterm since the second one refers to the same variable. Notice that the resulting R tuple set is retrieved ordered.

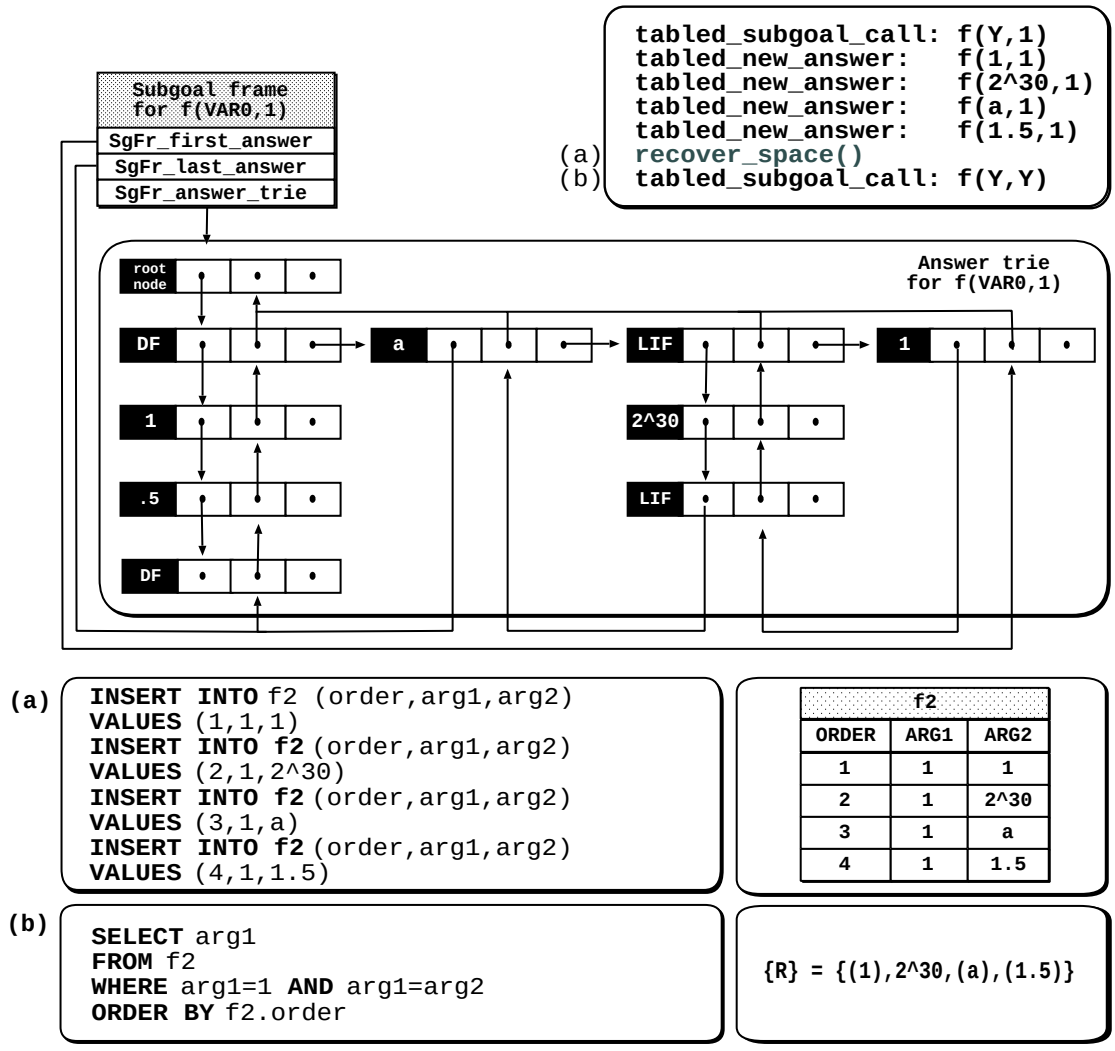

Figure 4.3: A subgoal branch relational schema

The proposed relational data model presents the following advantages:

1. a single table is required for each predicate. Different subgoal calls may easily be stored in the same table without loss of generality, since each individual call answer set may be distinguished by its constant terms;

2. it allows the use of iterative algorithms both to store and retrieve the answer tries. For storage purposes, the number of required data transactions amounts to the number of answers; for retrieval purposes, a single data transaction is required;

3. the loading algorithm uses the highly effective *insert/check* function to reconstruct the original answer trie, thus reducing the complexity of the algorithm.

The following major disadvantages may be pointed out:

1. each node of the answer trie is visited more than once, both at storage and retrieval. In particular, the nodes near the vicinity of the root are visited quite often;

2. it is not minimal in terms of information, i.e., each relation holds extensionally the complete set of answer, thus undoing the benefits of the substitution factoring for the corresponding subgoal call;

3. it is unable to represent the entire set of terms that can be stored in a answer trie because, since it is based on *Datalog*, it restricted to atomic terms. Moreover, the same tuple structure must accommodate all possible subterms of a subgoal's answer. Hence, the devised mapping schema must be tailored to support the storage of the different YAP primitive types in a common relational domain. Two distinct approaches, discussed in the next section, may be used to solve this problem;

4. an index is needed for each $\texttt{arg}_i$ attribute, in order to speed-up tuple search;

5. for incomplete tries, a full traversal phase is required whenever a storing operation is requested since it is not possible to determine which answers have already been stored and which ones where not. While no tuple regeneration is required and new answers may be added without disrupting the previously existing ones, a reordering of the tuples is required if the answers are to be kept in insertion order;

6. the storage algorithm's simplicity introduces a potential bottleneck due to the tuple-by-tuple data transaction.

4.2.2 Tries as Sets of Nodes

The previous model placed its focus on the answer themselves, directly mapping each predicate answer set into a database relation. Yet, storing the complete answer set extensionally is a space (and time) consuming task. A more fruitful approach may be followed if one steps back a little and looks at the problem from a different perspective.

Tabling tries are organized according to the principle of *substitution factoring*, as previously observed in section 3.2.2. That same line of reasoning may be applied when mapping a predicate table into a relational representation, completely separating the upper and lower parts of table. Moreover, in the particular context of the *least used algorithm*, one can focus on the transactions involving the answer tries, since the subgoal trie is bound to remain in memory.

The major challenge of this approach lies in the choice of a representation model. In practice, hierarchical structures such as trees (and tries) may be divided in two categories: those with a fixed number of levels and those without. When a fixed number of levels is known to exist, the tree can be represented by making each level a single column of table. Unfortunately, that results in heavy data redundancy and does not reflect the true essence of the hierarchic structure. The best way to represent a hierarchy with a fixed number of levels is to turn each level into a single table and, for each pair of levels, establishing a one-to-many relationship between the lower and the higher levels [FK99, Gen03]. Many trees, however, present a non fixed number of levels. In particular, YapTab tries fall under this category, as observed back in section 3.2. To mimic the very structure of tries, a possible approach is to establish the relation as a recursive set of tuples representing the tree node [FK99, Gen03]. This requires a different type of table, one whose attributes include enough information to allow the correct hierarchical ordering of its records. From this point on, we shall denominate this approach as the *hierarchical model*.

In YapTab, the trie hierarchy is maintained by the pointers that link the nodes. The trie is a collection of nodes, representing the parsing states for input terms, that are linked by pointers, representing the transitions between states [RRS+95]. Each tree node consists of four fields, named *symbol*, *parent*, *next*, and *child*, that holds, respectively, the expected input symbol when the node is reached, the address of the parent node, the address of the immediate sibling of the node and the address of the

last inserted child. If a direct mapping is performed, the *Trie* relation can then be defined as

$$Trie(\underline{\text{parent}} : node,\ \underline{\text{child}} : node,\ \underline{\text{next}} : node,\ \text{symbol} : term)$$

exhibiting an attribute for each field in the trie node structure. The unique positioning of each node in the trie is determined by the conjunction of the values for its pointer fields, so the conjunction of corresponding attributes becomes the primary key of the relation.

Choosing a correct relational domain to map the *term* and *node* types requires some specialized knowledge of the YAP engine. It has been observed, back in section 3.2.2, that internal representation of terms in YAP may differ in their precision accordingly to their primitive values. For this reason, *insert/check* splits larger subterms and assigns each piece a different node in the trie branch. Since each node holds a generic YAP term and larger terms are divided in multiple nodes, relational INTEGER domains may be defined to store the *term* fields without loss of generality.

Mapping nodes, however, is a totally different problem that can be solved in many possible ways [MYK95]. The key attributes for the tuple may be easily instantiated if one directly assigns the value held by the pointer fields (memory addresses) of the node. However, those values become senseless after the removal of the trie from memory. Another possibility is to apply the principles of the relational model as established by Codd [Cod70] and assign a *unique identifier field* to each tuple. This increases the attribute set to

$$Trie(\underline{\text{this}} : node\_id,\ \text{parent} : node\_id,\ \text{child} : node\_id,\ \text{next} : node\_id,\ \text{symbol} : term)$$

where `this` is a *unique* node identifier, acting as the *primary key*, and `parent`, `child` and `next` are identifiers of the same type, acting as *foreign keys* over the relation. Several techniques may be applied to obtain the required unique identifier [AYDF04] but, for reasons later explained, a single sequential INTEGER counter is enough for this purpose.

At a first glance, this would be sufficient. One could insert and retrieve the different tuples into and from the relation using a conveniently adapted version of *insert/check*

function. However, when the practical aspects are considered, one comes to the conclusion that establishing a perfect parallelism between the node structure and the relation tuple is not productive. Let us see why this is so.

First of all, the mapping of all the pointer fields to attributes of the relation is troublesome. The introduction of foreign keys would enforce the consistency of data, guaranteeing that no answers were to be lost due either to a missing or non-existing tuple identifier. However, that would require that those attributes should be correctly instantiated whenever a new tuple was to be inserted, which in turn would require the previous insertion of three other tuples, representing the parent, the child and the closest sibling nodes. Unfortunately, that would be impossible because **(i)** no simple traversal algorithm could be used to insert tuples in these conditions and **(ii)** since the integrity constraints would be defined over the very including relation, the restrictions would propagate to all attempts of insertion, turning the complete process impossible.

Secondly, the implementation of an adapted *insert/check* function is not practical, because navigating the relation instance to insert a single tuple would require extra several transactions; in particular, a SELECT for each valid output transaction (the checking part of the algorithm). This renders the `child` and `next` attributes useless, since no practical gains are obtained from their presence. In fact, their presence would require some extra relational operations (at least one UPDATE instruction to the `child` attribute of the preceding tuple) and a slightly more sophisticated storing algorithm. As a consequence, one can then reduce the relation's attribute set to

$$Trie(\underline{\text{this}} : node\_id,\ \text{parent} : node\_id,\ \text{symbol} : term)$$

In order to optimize forward searches over this relation, distinct indexes are defined over the `this` and `parent` attributes.

Last, but not least, the relation must be named in such a way that the answer trie for which it stands for may be immediately identified. Since several subgoal calls may originate from the same predicate, the functor and arity concatenation is not a viable solution here, unless one would append a distinctive context identifier to it. Assuming that such an identifier is easily computable, the previous relation can then be renamed as

(II) $\mathcal{P}n\_Sf(\underline{\text{this}} : node\_id,\ \text{parent} : node\_id,\ \text{symbol} : term)$

where Sf stands for a specific subgoal context suffix.

The number of required transactions can be minimized if the answer tries are traversed in a top-down manner (from root to leaf). The tuple storing algorithm, whose pseudo-code is presented in Fig. 4.4, stored the trie in insertion order, proceeding as follows: for any given trie node, label it as the current node. Look for its chain of siblings and, if such a chain exists, follow it until the last (oldest) node is reached and mark it as the current node. Until the initial node becomes the current one again, repeat three instructions: **(i)** insert a new tuple into the relation representing the current node; **(ii)** if an outgoing transaction for the current node exists, label it as the current node and start again; **(iii)** fall back into the previous sibling and label it as the current one. When both node and siblings are covered, return to the parent node and resume.

Whenever in step **(ii)**, the algorithm must take extra precautions. Since, as previously mentioned in section 3.2.2, leaf nodes are connected by their `TrNode_child` fields, a simple test to check for the presence of outgoing transactions could lead to an erroneous interpretation of the trie. Hence, in order to assure the correct storage of the trie nodes, the algorithm must perform a search over the *leaf* set (`L`) for the current node, following any outgoing transaction if and only if that search fails.

```
store(trie node trie_node, node id parent_id) {
     node id sibling_id
     trie node sibling_node

     {S} = SIBLING_NODES(trie_node) // trie_node included
     {L} = get_trie_leaf_nodes()    // trie leafs

     foreach sibling_node in {S} {
         sibling_id = create_node_id()
         child_node = CHILD_NODE(sibling_node)

         INSERT INTO Pn_Sf (this, parent, symbol)
         VALUES (sibling_id, parent_id, SYMBOL(sibling_node))

         if child_node and child_node not in {L} {
             store(child_node, sibling_id)
         }
     }
}
```

Figure 4.4: Storing an answer trie hierarchically

The choice of such a traversal mode during the storing phase guarantees that

- all nodes are visited only once;
- a single tuple insertion operation is performed[5];
- whenever a node is reached the tuple representing its parent is already present in relation;
- the tuples are correctly ordered, thus preserving the correct hierarchy among nodes when the trie is reconstructed.

The opposite operation consists on the generation of an ordered list of the complete set of tuples belonging to the $\mathcal{P}n\_Sf$ relation. The most natural way to produce this list is to perform an *index-nested-loop* as proposed by Blasgen *et al.* [BE77] and succinctly described in [SC90, DNB93]. Although no standard relational algebra expression can be used to correctly describe the attained tuple set due to the recursive nature of this algorithm, a quite acceptable approximation can be defined as

$$\mathcal{P}n\_Sf^{\star} \;=\; \bigcup_{i=0}^{n} \sigma_{\mathcal{P}n\_Sf'.this,\mathcal{P}n\_Sf'.symbol}\,(\mathcal{P}n\_Sf' \;\otimes\; (\sigma_{\mathcal{P}n\_Sf.this=i}\mathcal{P}n\_Sf))$$

where $\mathcal{P}n\_Sf'$ is just another view of the $\mathcal{P}n\_Sf$ relation. The `parent` attribute is omitted in the resulting $\mathcal{P}n\_Sf^{\star}$ relation since its information is required only to establish the relationship among the tuples of $\mathcal{P}n\_Sf$.

The pseudo-code used to compute such a tuple set is shown in Fig. 4.5, Notice that, in essence, it is rather similar to that defined for storage differing only in swapping of the roles assigned to the involved data structures: the relation instance becomes the source of data and the trie becomes the target. The algorithm proceeds as follows: for any given tuple identifier, select the complete set of descendant tuples. For each returned tuple, repeat three instructions: **(i)** create a new trie node and initialize its `parent` and `symbol` fields with the recovered information; **(ii)** recall the loading

[5]Recall that a *bottom-up* traversal of the trie results in frequent visits to the nodes in the vicinity of the root and, consequently, to repeated and undesirable attempts to insert the corresponding tuple into the relation. If a key policy has been defined, repeated database transaction errors will occur. Otherwise, the resulting transactions will introduce redundant information into the relation. Either way, undesirable overheads are added to the process.

procedure for the current tuple identifier and initialize the node's `child` field with the returned child node; **(iii)** initialize the node's `next` field with the immediately older sibling, if any is created; when tuples are covered, return the address of the current node and resume.

```
trie_node load(trie node trie_node, node id parent_id) {
    trie node new_node
    trie node next_node

    {T} = SELECT this, symbol
          FROM Pn_Sf
          WHERE parent = parent_id
          ORDER BY this

    next_node = nil
    foreach tuple in {T} {
        new_node = new trie node()

        PARENT_NODE(new_node) = trie_node
        CHILD_NODE(new_node) = load(new_node, tuple.this)
        NEXT_NODE(new_node) = next_node
        SYMBOL(new_node) = tuple.symbol

        next_node = new_node
    }
    return new_node
}
```

Figure 4.5: Loading an answer trie hierarchically

The recursive nature of the proposed retrieval procedure over the entire set of the tuples guarantees that

- the entire set of tuples is returned;
- all nodes are visited only once (when they are created);
- a single tuple selection operation is performed for each node (the list of outgoing transactions);
- whenever a node is created, its parent has already been placed in the trie (since it is reconstructed one level at a time);
- the correct hierarchy among nodes is preserved.

Figure 4.6 presents a practical use of the proposed relation schema and algorithms. The top right box contains a first set of instructions that are used to create and

populate the presented trie. Let us assume that at some point in execution, the trie becomes inactive and that the `recover_space()` function is called after that, as a consequence of a main-memory exhaustion situation. The storing algorithm is called to dump the inactive trie to the database: its main cycle visits the nodes producing the SQL instructions used to map them into the relation tuples (box **a**). The visiting order is shown by the numbers in italic placed above the nodes. Later on, when a variant subgoal call of the tabled predicate occurs, the loading algorithm is called to reconstruct the trie from the tuples in the relation. The issued SELECT instructions (box **b**) retrieve the R_i tuple sets, that are in turn parsed to instantiate the nodes. Notice that the node creation order is the same as that followed in storage phase.

Two small optimizations may be observed. First, the root node is not stored, because not only no relevant information is kept in that node but also because it is never removed from memory during the memory recovery operation. Second, only one of the special delimiters surrounding long atomic terms *32-bit* integers or floating-point numbers is required to identify the specific primitive type[6].

The proposed relational data model presents the following advantages:

1. it is minimal in terms of information, storing no more data than the one available in the answer trie. In other words, each relation holds the substitution factoring of the corresponding subgoal call;

2. it is able to represent the entire set of terms that can be stored in an answer trie, including list and application terms. This is guaranteed by the fact that the value of each node has its `symbol` field copied into the corresponding attribute of each record;

3. each node of the trie is visited only once, both at storage and retrieval;

4. the retrieval and reconstruction of the answer trie is self-contained and direct, i.e, the loading algorithm inserts the nodes in the correct position without having to reconstruct the original terms and pass them to the *insert/check* function.

Unfortunately, it also presents some major disadvantages:

[6]Please refer to section 4.3 for an insight on this topic.

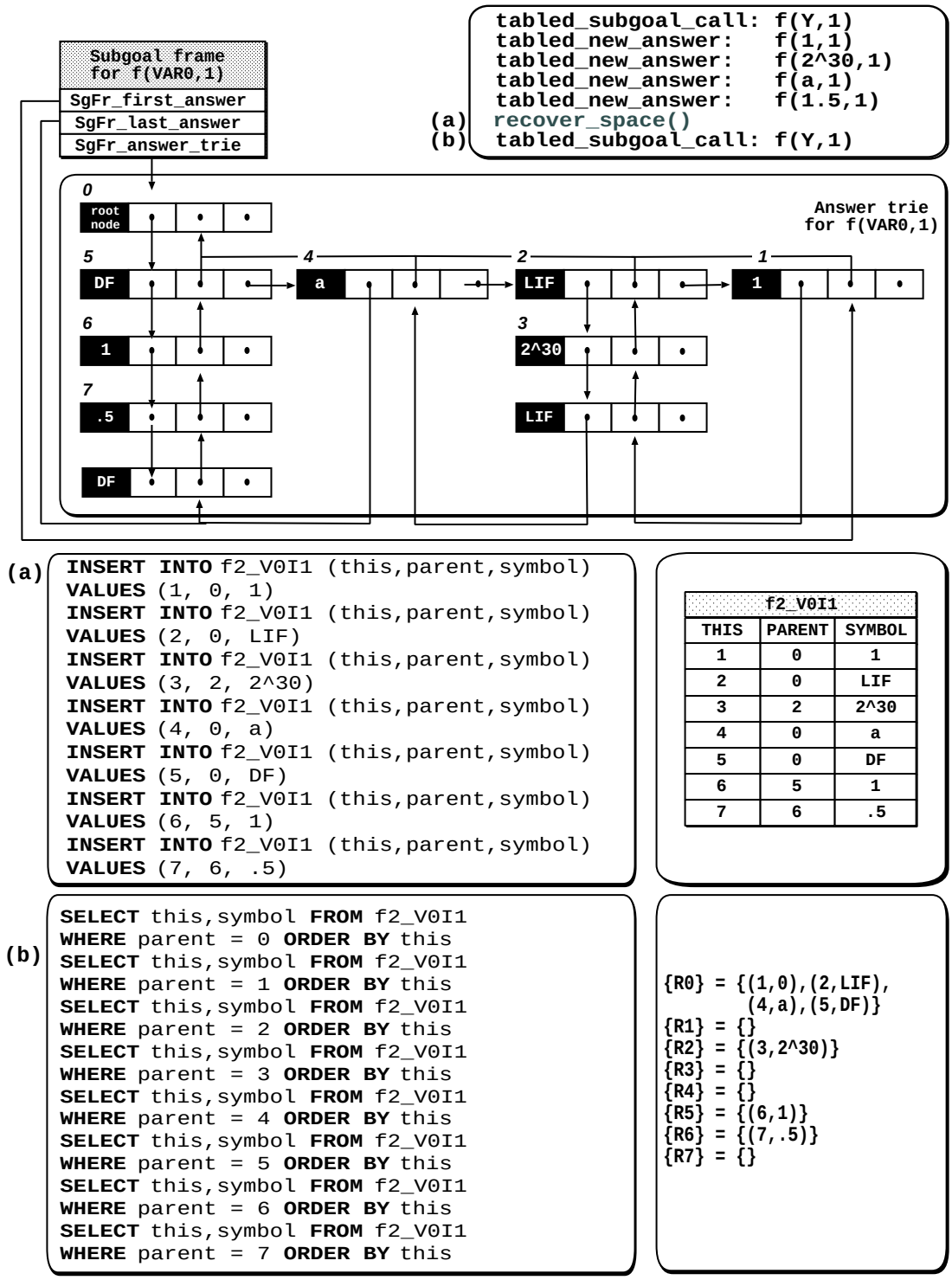

Figure 4.6: A node hierarchy relational schema

1. it requires highly recursive algorithms to both store and retrieve the answer tries, involving several data transactions. The network round trips required to

store and retrieve the entire trie structure have a considerable impact on the performance of YapTab;

2. for incomplete tries, a full traversal and tuple generation phase is required whenever a storing operation is requested. Since the correct ordering of tuples relies on a snapshot of the trie, the implementation of an incremental storing procedure becomes impossible because the addition of new nodes to the trie will result in a completely different snapshot, and consequently in a new ordering of tuples. This is why no complex system of key assignment works. In fact, if such an addiction occurs, the entire set of tuples must be flushed and the storing operation must be fully repeated.

4.3 Storing Term Primitive Values

As observed back in section 3.2, YapTab tries to accommodate YAP term values differently according to their type and size. Most often, the *insert/check* function splits primitives values into pieces and distributes the tokens by several trie nodes. Obviously, DBTab must provide some sort of mechanism that assures the correct reorganization of tries during retrieval operations.

The two possible mapping schemes have distinct capabilities in what comes to primitive value representation. The *hierarchical* model reflects the actual trie structure, hence it is able to store all possible primitive values. The *Datalog* model, on the other hand, is limited to constant atomic terms by definition. Relation (I), presented back in subsection 4.2.1, contemplates the storage of terms in general. However, the decision on which relational domain should be used for the pretended term representation is a hard one to make. It is possible that an attribute $\texttt{arg}_i$ of the relation, mapping a subterm of the predicate, is required to hold values of distinct types, such as strings and numbers, or different precision ranges, such as integer and floating-point numbers. Two possible solutions for this problem are considered as suggested by Florescu *et al.* in [FK99].

4.3.1 Inline Values

A first and simple solution is to extend the generic relation labeled as (I), adding a new attribute for each of the conceivable data types per predicate subterm; Florescu *et al.* refer to this approach as *inlining*.

Some primitive types, such as integer terms, perfectly fit into the space allocated for each arg_i attribute, hence no additional attribute is required for them. Other primitive types, such as floating-point numbers, large integer numbers[7] and atomic strings clearly exceed the available space. This means that, in practice, each of those types requires a special extra attribute. A new generic relation

$$\mathcal{P}n(\ldots,\ \mathrm{arg}_i : \mathit{integer},\ \mathrm{atom}_i : \mathit{string},\ \mathrm{lint}_i : \mathit{integer},\ \mathrm{fltp}_i : \mathit{double},\ \ldots)$$

may then be defined for each tabled predicate, where each $\langle \mathrm{arg}_i, \mathrm{atom}_i, \mathrm{lint}_i, \mathrm{fltp}_i \rangle$ attribute subset is considered a single subterm. Since no term can belong to two distinct primitive types, at most one of the additional attributes will contain a value other than NULL. In order to tell which of the placeholder holds the correct value, a special *flag* is placed in the $\texttt{arg}_i$ attribute whenever the ith subterm value is stored in one of the auxiliary attributes.

However, the additional attribute destined for character strings is not strictly required. This is due to a particular YAP implementation feature, aimed at the improvement of performance. Whenever a new atom appears in the context of a logic program, YAP stores the primitive string value in its *symbols table*, placing a reference to that position inside the newly created term. From that point on, YAP uses that reference for all required operations; the primitive string value is seldom used by input/output instructions or specific atom manipulation instruction.

Since the symbols' table is maintained during execution, the storage of the internal representation of atoms into the $\texttt{arg}_i$ attributes of a mapping relation residing in the database is more than enough to keep things running smoothly. Hence, the previous relation is reduced to

[7] Large integers are all those integer values that require more space than the one available in the term non-mask bit set.

(III) $\mathcal{P}n(\ldots,\ \mathrm{arg}_i : \mathit{integer},\ \mathrm{lint}_i : \mathit{integer},\ \mathrm{fltp}_i : \mathit{double},\ \ldots)$

The storing and loading algorithms require few changes to work with this extended relation. In fact, the only required adaptation is a simple test to determine the type of the term to store or retrieve: in the first case, to decide which additional attribute to initialize; in the second to correctly reconstruct the term before passing it as argument to the *insert/check* function in the correct position.

The major advantage of this schema is its simplicity. Each relation concentrates all the primitive values for the predicate's subterms and a low number of simple data transactions is required to manipulate the stored information[8].

Obviously, this schema produces highly sparse tables, where a large number of NULL values occur. An *index-per-attribute* policy may not be implemented because **(i)** the $\texttt{arg}_i$ attribute alone contains few information on the term's actual primitive value; and **(ii)** it is possible that the extra attributes contain only NULL values [FK99]. In particular, this last possibility makes the definition of indices impossible under MySQL [WA02].

4.3.2 Separate Value Tables

Another possible way to solve this problem is to introduce specialized lookup tables of the generic form

(IV) $\mathit{PrimitiveType}(\mathrm{id} : \mathit{term\_id},\ \mathrm{value} : \mathit{primitive})$

where $\mathit{PrimitiveType}$ is one of the possible relation names $\{\mathit{PrimitiveLongint}, \mathit{PrimitiveFloat}\}$, `id` is a term identifier, typically an INTEGER value, and `value` is the primitive INTEGER or FLOAT value of the 32-bit integer or float term.

The generic relation labeled as (I) is transformed so that all $\texttt{arg}_i$ attributes are retyped to the *term_id* type. Since the contents for such attributes may originate from the `id` attribute of the two different auxiliary tables, no foreign key policy can

[8] Including an INSERT operation for each answer submission and a single SELECT operation for a complete data set retrieval.

be implemented. Additionally, some sort of distinguishing mechanism is required to identify the auxiliary relation from which the term's primitive value is to be retrieved. Florescu [FK99] suggests the addition of a new *flag* attribute for each of the transformed $\texttt{arg}_i$ attributes; in the YAP engine context, however, this extra attribute is dispensable. Since YAP terms already convey built-in tags that enable the engine to identify the type of the term, one may then take advantage of this characteristic and mask the sequential term identifiers with these flags.

The storing algorithm must be slightly altered to be used with this approach. Whenever a term is of one of the specially parted types, two transactions must occur: firstly, its primitive value must be stored into the respective primitive table and, secondly, the generated identifier must then be placed in the correct $\texttt{arg}_i$ attribute of the answer tuple that is to be stored in the main table. The `create_primitive_id()` function generates sequential identifiers based on a seed value whose origin will be later explained. The code snippet in Fig. 4.7 illustrates the idea.

```
store(subgoal frame sg_fr) {
    ...
    {TERMS} = bind_answer_terms({TEMPLATE},answer)
    foreach TERM in {TERMS} {
        if TERMi is primitive  {
            prim_id = create_primitive_id()

            INSERT INTO PrimitiveType (id,value)
            VALUES (prim_id, PRIMITIVE_VALUE(TERMi))

            TERMi = prim_id
        }
    }
    INSERT INTO Pn (order,ARG1, ..., ARGn)
    VALUES (answer_id, TERM1, ..., TERMn)
    ...
}
```

Figure 4.7: Storing primitive term values

To assure that the loading algorithm works independently of the primitive values representation schema, it must be transformed to accept a tuple set similar to the one proposed for the *inlining variant*. In other words, all $\texttt{arg}_i$ attributes must be followed by the additional primitive container attributes. The expected tuple set

$$\mathcal{P}_n^* = \bigcup_{i=0}^{n} \pi_{\mathrm{arg}_i,\iota.value,\phi.value}((\mathcal{P}_n \bigotimes_{\mathrm{arg}_i=\iota.id} \iota) \bigotimes_{\mathrm{arg}_i=\phi.id} \phi)$$

results from the application of a series of *left outer joins* (denoted as $\otimes$) on the main table, one for each $\texttt{arg}_i$ attribute admissible primitive type (ι and ϕ respectively denote large integer numbers and floating-point numbers).

In practical terms, this data set is obtained executing a SQL query like the one exhibited in Fig. 4.8. The <VARIABLES> subexpression now contains additional $\mathcal{P}n.\text{lint}_i$, $\mathcal{P}n.\text{fltp}_i$ items for each $\mathcal{P}n.\text{arg}_i$ item. The <CONDITION> subexpression may also hold references to these additional attributes if some kind of testing is needed over the respective constant values.

```
SELECT <VARIABLES>
FROM Pn
    LEFT JOIN PrimitiveInteger AS I ON Pn.ARG1=I.id
    LEFT JOIN PrimitiveDouble  AS D ON Pn.ARG1=D.id
    ...
    LEFT JOIN PrimitiveInteger AS I ON Pn.ARGn=I.id
    LEFT JOIN PrimitiveDouble  AS D ON Pn.ARGn=D.id
WHERE <CONDITION>
ORDER BY Pn.ORDER
```

Figure 4.8: Flat approach resulting tuple set

Keeping primitive values in separate tables presents two major advantages. First, it provides a straightforward conceptualization by grouping primitive values according to their respective types. Second, it promotes table compactness by deferring the large sized primitive values to the auxiliary tables and keeping the smaller identifiers in the main table. Moreover, compactness is enhanced by the absence of NULL value occurrences in the main table. The major drawbacks are the need for multiple insert operations and the number of *left outer joins*. These steps are bound to cause execution overheads, even in a fast database system.

4.4 Meta-data

So far, the relational schema has been presented as a persistent place holder for data. Although persistency is the primary concern of the developed schema, surely it is not the only one. Most modern relational database management systems allow multiple users to operate over the same database, the same table or even the same record. Because of this, concurrency control is a major concern in such systems.

Although not directly related to DBTab, that issue must also be considered during

this relational schema design. Despite the possibility for each user to define a different database for each of its applications, it is also possible that a single predefined schema should be used for several reasons. For instance, institutional policies may force all system users affiliated with an institution, department or group to share predefined database schemes. Another significant example is the simultaneous execution of several instances of the same logical program that relies on relational tabling.

Regardless of the causes, the developed system is expected to perform correctly when database schemes are shared among different running instances of YapTab, allowing the storage of non disjunctive set of answers for the equally named predicate at the same instant in time. This peaceful coexistence can only be achieved by the conjugation of the relational templates proposed throughout this chapter with some kind of special mechanism that allows each YapTab instance to uniquely identify all of its stored subgoals and respectively computed answers.

This line of reasoning leads to the introduction of the concept of *session*. The underlying rationale is quite simple: if each Yaptab instance maps its table space into a particular partition of the database schema, the dangers of data loss, misplacement or corruption are no more. All it takes is a way to assure the complete isolation of each session context.

To help in this purpose, a special set of relations is introduced. Rather than being directly involved in the maintenance of run-time data, this new set is concerned with the maintenance of status information, i.e., number of currently open sessions, number and ownership of tabled predicates, etc. From this point on, this set is referred to as the *system control table set*, or *control tables* for short, in opposition to the data related relation set, to be known henceforth as *session context table set*, or *session tables* for short.

The first and most basic of DBTab's control relation is

$$Sessions(\underline{\text{sid}} : integer)$$

whose single attribute[9] is a placeholder for the numerical identifiers of the active

[9]This is a simplified implementation. This relation could store other attributes, used for several ends. For instance, it would be interesting to establish a session timeout policy based on the sessions' creation date.

sessions. The second control relation is defined as

$$Sequences(\underline{\text{sid}} : integer,\ \underline{\text{name}} : varchar(255),\ \text{value} : integer)$$

and it is used to hold the seed values for special sequences, such as the session identifiers or the primitive identifiers mentioned back in section 4.3.2. The `sid` attribute is a *foreign key* to the homonym attribute in the *Sessions* relation. The other two attributes have quite obvious meanings: `name` holds a character string that identifies the name of each sequence, while `value` holds the specific seed value. The primary key of this relation comprises the `sid` and `name` attributes.

The third and final control relation has different prototypes each of the mapping approaches. For the Datalog model, the *Predicates* relation is defined as

$$Predicates(\underline{\text{sid}} : integer,\ \underline{\text{functor}} : varchar(255),\ \underline{\text{arity}} : integer)$$

where `sid` attribute is a *foreign key* to the homonym attribute in the *Sessions* relation, `functor` is a character string placeholder for the tabled predicates functors and `arity` is an integer placeholder the predicates arities. The primary key of the relation is comprised of all three attributes. In the hierarchical model, the relation is extended to

$$Predicates(\underline{\text{sid}} : integer, \underline{\text{functor}} : varchar(255), \underline{\text{arity}} : integer, \underline{\text{subgoal}} : varchar(255))$$

where all the previously described attributes maintain their meaning, and `subgoal` is a character string destined to hold the special subgoal identifying suffix. All four attributes are used as primary key to the relation.

Supported by these relations, the database schema partition now becomes a quite easy task to fulfil. A first possible solution would be to extend the relations templates, adding an extra attribute that would hold a session identifier. Despite of its simplicity, this approach is not pursued for two main reasons. First of all, that would somewhat taint the original mapping concept, introducing an attribute into mapping relations that was not present or required in the original relational data model. Second, and perhaps more importantly, the concentration of answers originating from different sessions on the same relation is inadvisable. Despite of InnoDB tables' capacity

to grow arbitrarily, MySQL developers recommend the splitting of large tables into several smaller ones, in order to increase engine performance [WA02]. A second and more reasonable solution is to embed the session identifier into the names of context dependent relations. The names of those tables could, for instance, carry a special prefix $ssSi$, where Si stands for the session identifier value.

An example may clarify the idea. Figure 4.9 presents two table space samples, one for each mapping approach. Notice **(i)** how the control tables are differently defined and populated, **(ii)** how the different subgoals of the $f/2$ predicate are registered into the *Predicates* control table and **(iii)** how the different mapping relations for those subgoals are named in the two distinct approaches.

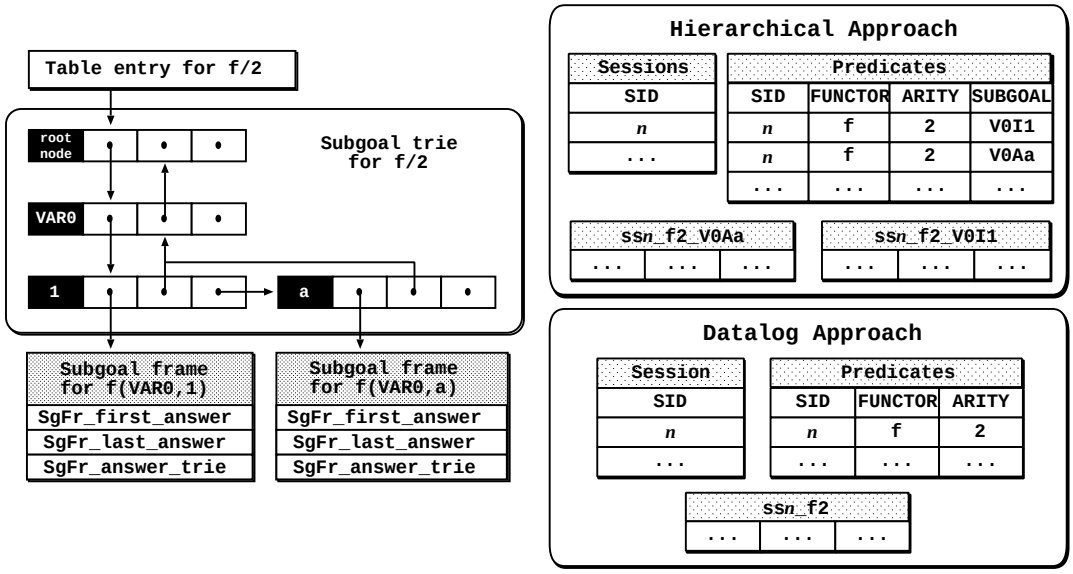

Figure 4.9: Different mapping approaches

4.5 Chapter Summary

This chapter introduced the reasons why tabling may benefit from external memory storage mechanisms, such as relational databases. Two possible relational schemes were discussed: one following an approach closely related to *Datalog*, the other aiming at a direct mapping of answer tries. The distinct loading and storing algorithms were presented along with their advantages and disadvantages. Primitive value handling for YAP terms was then discussed, in both contexts. Finally, the last section discussed

meta-data as a mean for databases schema partitioning. Control and sessions relation concepts were introduced and differentiated and naming conventions for the schemes relations were presented.

Chapter 5

DBTab: the YapTab Relational Extension

In this chapter, the implementation of DBTab is stressed. A brief contextualization of the developed work is provided, focusing on its connections with previous work on the subject. The relational database management system chosen to support DBTab is introduced and its major assets are discussed next. The transformations produced in YapTab to enable the implementation of relational tabling are discussed afterwards, covering topics like the developed API and the changes to Yaptab's table space structures and managing algorithms. It concludes with the presentation of an alternative way to lookup subgoals' answers without fully reloading tries into memory.

5.1 A Coupled System

The work developed during this thesis draws its inspiration and guidance from two different sources: on one hand, the work of Rocha on efficient support for incomplete and complete tables in the YapTab tabling system [Roc07], and on the other hand, the work of Ferreira *et al.* on coupling logic programming with relational databases [FRS04, FR04, FR05, TSR05]. In [FRS04], these authors propose several alternative ways to couple YAP with the famous MySQL database management system, using both systems' built-in C-language interfaces and Draxler's *Prolog-to-SQL* compiler [Dra92]. The presented results are very auspicious, certifying the

effectiveness, efficiency and robustness of such coupling approaches.

This work drifts off from those original proposals by excluding Draxler's compiler from the middle tier interface. MYDDAS [TSR05, SFRF06] uses the compiler as a generic tool due to its ability to translate Prolog queries in their SQL counterparts. The translation language is equivalent to relational calculus in expressive power, enabling the transformation of conjunctions, disjunctions and negation of goal and high-order constructs such as sorting and grouping. Obviously, the use of such a tool induces significant execution overheads, mostly originating from the query parsing phase. While standard database application users are willing to endure this as an acceptable price to pay for greater expressive power, tabling system users expect their applications to execute with the smallest possible impact on execution performance. The external compiler then becomes more of a problem than a solution: as observed back in chapter 4, DBTab requires a much simpler mechanism to send (relatively) simple SQL queries to the database and retrieve the resulting tuple sets.

5.1.1 Some Advantages of Using MySQL

The MySQL *C API for Prepared Statements* [WA02] constitutes a more suitable tool to implement the connection between the RDBMS and the logical engine. Prepared execution is particularly efficient in situations where the same SQL statements are executed more than once, mainly because:

1. each SQL statement is parsed only once, whenever it is first sent to the database. If the parsing is successful, specialised data structures are placed in memory at server-side, enabling the repeated execution of the statement. Whenever the statement includes variable input parameters, specially sized and typed buffers are allocated and kept in memory for later use;

2. since the statement invocation requires only the transmission of the respective input parameters, rather than the complete SQL statement, network traffic is substantially reduced;

3. a binary protocol is used to exchange data between the client and the server. This not only reduces the size of exchanged data blocks but also dismisses the otherwise required *string-to-primitive value* conversion.

As of version 5.0, MySQL supports *stored procedures* and *stored functions*. In short, a stored procedure (or function) is a set of SQL commands that reside at server-side, most often in user-defined libraries. This feature enhances program modularity, execution performance and overall security. For instance, client applications may invoke stored procedures that execute a batch of SQL statements rather than issuing the individual commands one by one, thus reducing network traffic. Moreover, stored procedures provide a consistent and self-contained execution environment, avoiding users and applications direct access to tables while assuring the logging of each operation at the same time.

DBTab relies on the InnoDB MySQL storage engine, a storage engine designed for maximum performance when processing large volumes of data. Developers claim that InnoBD's CPU efficiency is most likely unpaired by any other disk-based relational engine. [WA02]. The InnoBD engine features:

- commit, rollback and crash-recovery capabilities;
- row level write locking, while preserving non-locking read capabilities;
- no limitation in terms of table growth, even in file systems where such limits exist;
- proper table space for tables and indexes;
- support for FOREIGN KEY constraints with CASCADE abilities.

5.1.2 The Database Layer

Some of the tasks performed by DBTab may be performed entirely at database level. Features like session maintenance, predicate tables definition and control tables manipulation may be hidden from the YapTab system layer, somewhat simplifying the application programming interface (API). For this purpose, each storage schema is provided with six stored routines that constitute a first control layer:

`session_register(sid)` searches for the supplied argument within the *Sessions* table. In case of success, it returns immediately. Otherwise, it creates a new

session by inserting a new record into the referred table. Additionally, and depending on the selected storage schema, it may create the appropriate session auxiliary tables. The *separate values* variant of the Datalog model must define the *PrimitiveFloat* and *PrimitiveInteger* relations. In all implementations, the routine performs internal session variables setting, including the current session identifier @SID, which is embedded in the prefix all of the session tables. The procedure is displayed in Fig. 5.1;

`predicate_register(functor,arity)` searches for an existing tuple in the *Predicates* relation whose attribute values correspond to the passed argument values. In case a matching record is found, the routine terminates immediately. Otherwise, it inserts a new tuple where the first attribute (`sid`) value is obtained from the session identifier internal variable, the second and third attributes are initialized with the supplied `functor` and `arity` homonym arguments. Afterwards, it creates a new relational table to hold the registered subgoal answers. In a sense, this operation mimics YapTab's table entry creation for new subgoals. The details are shown in Fig. 5.2;

`start_transaction()` is used to prepare MySQL for a data transaction. In the *separate value* variant, the routine returns the next starting sequential identifier for the application-masked terms (floating-point and 32-bit integers) stored in *Sequences*;

`end_transaction(success)` is used to finish a data transaction. If the supplied `success` argument carries the TRUE logical value, the transaction is *committed*, otherwise, a *rollback* operation occurs, restoring the database status to that previous to the call of `start_transaction`. In the *separate value* variant, it also updates the value for the application-masked terms sequential identifier.

`predicate_unregister(functor,arity)` issues a drop instruction for the predicate mapping relation whose name conveys the passed arguments. The statement includes a "IF NOT EXISTS" clause to prevent MySQL to report an error if no such table exists. Afterwards, the procedure attempts to delete a tuple belonging to the *Predicates* relation whose `functor` and `arity` attributes match the homonym procedure's arguments. Please refer to Fig. 5.2 for details;

`session_unregister(sid)` is used to finish the session identified by the supplied `sid`

```
CREATE PROCEDURE session_register(INOUT sid INTEGER)
BEGIN
    -- Search the identified session
    SET @SID = 0;
    SELECT s.SID INTO @SID FROM Sessions AS s
    WHERE s.SID = sid;

    IF (@SID=0) THEN
        -- Create new session
        UPDATE Sequences SET VALUE = LAST_INSERT_ID(VALUE+1)
        WHERE NAME LIKE 'SEQ_SESSIONS';

        SET @SID = LAST_INSERT_ID();
        INSERT INTO Sessions (SID) VALUES (@SID);

        -- In the "separate value" variant,
        -- Create auxiliary tables
        SELECT CONCAT("CREATE TABLE "ss",@SID,"_Longints (",
            "ID INTEGER NOT NULL, ",
            "VALUE INTEGER NOT NULL, ",
            "PRIMARY KEY (ID)) ENGINE=InnoDB;")
        INTO @EXPR;
        PREPARE stmt FROM @EXPR; EXECUTE stmt;

        SELECT CONCAT("CREATE TABLE "ss",@SID,"_Floats (",
            "ID INTEGER NOT NULL, ",
            "VALUE DOUBLE NOT NULL, ",
            "PRIMARY KEY (ID)) ENGINE=InnoDB;")
        INTO @EXPR;
        PREPARE stmt FROM @EXPR; EXECUTE stmt;

        DEALLOCATE PREPARE stmt;
    END IF;
END //
```

Figure 5.1: Session opening stored procedure

argument and perform clean-up tasks, i.e., the procedure drops all predicate tables associated with the session identified by the `sid` argument, including the existing auxiliary lookup tables. The complete procedure is presented in Fig. 5.4.

The presented `predicate_register` and `predicate_unregister` stored procedures originate from the Datalog model implementation. In the *hierarchical* model, these procedures present two small differences. Firstly, in order to cope with the slightly different declaration of the *Predicates* relation, both procedures present a third argument, `subgoal`. The value carried by this argument is supplied as an unique identifier of the subgoal frame for which the registered relation stands for. Secondly, the table creation code in the `predicate_register` is a lot simpler, since the total number of fields for the new relation is always constant. The table creation cycles

```
CREATE PROCEDURE predicate_register(functor VARCHAR(255),arity INTEGER)
BEGIN
    DECLARE n INTEGER DEFAULT 1;

    -- Search predicate's table
    SET @TABLE_EXISTS = FALSE;

    SELECT TRUE INTO @TABLE_EXISTS FROM Predicates AS p
    WHERE p.SID=@SID AND p.FUNCTOR=functor AND p.ARITY=arity;

    IF NOT @TABLE_EXISTS THEN
        -- Create predicate's table
        INSERT INTO Predicates (SID,FUNCTOR,ARITY)
        VALUES (@SID,functor,arity);

        SELECT CONCAT("CREATE TABLE ss",@SID,"_",functor,arity," (")
        INTO @EXPR;
        REPEAT
            SELECT CONCAT(@EXPR, "ARG",n," INTEGER NOT NULL, ") INTO @EXPR;
            SET n = n+1;
        UNTIL n = arity END REPEAT;

        SELECT CONCAT(@EXPR,"PRIMARY KEY(ORDER)) ENGINE=InnoDB;") INTO @EXPR;
        PREPARE stmt FROM @EXPR;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
    END IF;
END //
```

Figure 5.2: Predicate registering procedure

```
CREATE PROCEDURE predicate_unregister(functor VARCHAR(255),arity INTEGER)
BEGIN
        -- Drop table
        SELECT CONCAT("DROP TABLE IF EXISTS ss",
                      @SID,"_",functor,arity,";")
        INTO @EXPR;
        PREPARE stmt FROM @EXPR; EXECUTE stmt;
        DEALLOCATE PREPARE stmt;

        DELETE p FROM dbtab_predicates AS p
        WHERE p.SID = @SID
          AND p.FUNCTOR LIKE functor
          AND p.ARITY=arity;
END //
```

Figure 5.3: Predicate unregistering procedure

```
CREATE PROCEDURE session_unregister(sid INTEGER)
BEGIN
    DECLARE functor VARCHAR(255);
    DECLARE arity, done INTEGER DEFAULT 0;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;

    DECLARE cur1 CURSOR FOR
        SELECT p.FUNCTOR,p.ARITY FROM Predicates AS p
        WHERE p.SID=@SID;

    -- Drop session's tabled predicates
    OPEN cur1;
    REPEAT
        FETCH cur1 INTO functor,arity;
        CALL predicate_unregister(functor,arity);
    UNTIL done END REPEAT;
    CLOSE cur1;

    -- If in "separate value" variant,
    -- Drop session's auxiliary tables
    SELECT CONCAT("DROP TABLE ss",@SID,"_Longints;") INTO @EXPR;
    PREPARE stmt FROM @EXPR; EXECUTE stmt;

    SELECT CONCAT("DROP TABLE ss",@SID,"_Floats;") INTO @EXPR;
    PREPARE stmt FROM @EXPR; EXECUTE stmt;

    DEALLOCATE PREPARE stmt;

    -- Remove session id from sessions table
    DELETE s FROM dbtab_sessions AS s
    WHERE s.SID = @SID;
    SET @SID = 0;
END //
```

Figure 5.4: Session closing stored procedure

```
CREATE PROCEDURE predicate_register(
        functor VARCHAR(255),arity INTEGER,subgoal VARCHAR(255))
BEGIN
    (...)
    IF NOT @TABLE_EXISTS THEN
        -- Create predicate's table
        (...)
        SELECT CONCAT("CREATE TABLE ss",@SID,"_",
                      functor,arity,"_",subgoal," (",
                      "THIS   INTEGER NOT NULL,",
                      "PARENT INTEGER NOT NULL,",
                      "TOKEN VARCHAR(255) NOT NULL,",
                      "PRIMARY KEY (THIS,PARENT))ENGINE = InnoDB;")
        (...)
    END IF;
END //
```

Figure 5.5: Variant predicate registering procedure

appearing in the body of the procedure are thus replaced by a single expression, as illustrated by Fig 5.5.

5.2 Extending the YapTab Design

In order to supply YapTab with a relational representation of its table space some new features must be introduced. First of all, the correct implementation of the mapping strategies presented back in chapter 4 depends on some small alterations to the table space data structures. Additionally, database communication skills must be introduced. Arguably, the best way to successfully fulfil this task is to keep changes restricted to small areas of the primitive code. The first and obvious candidates are the table space structures, namely the *subgoal frame* structure due to its central role in the *least used algorithm*. Other structures, such as the *table entry* and the *loader choice point*, may also require some small modifications for reasons that will become obvious as this section unfolds.

The remainder of this section presents some solutions for the enumerated issues. It begins by addressing the modifications to the original design of Yaptab. Next, prepared statements are introduced as an easy and swift way to access the database tables. It follows by presenting the developed middleware API and the way it works. At last, an alternative way to inquire for trie answers without fully reloading the trie into memory is presented.

5.2.1 Modified Data Structures

The implementation of the *Datalog* model revealed the first obstacle to overcome. As previously mentioned, the tabling trie is organized in such a manner that, starting from a given table entry, one can reach all of its subgoal frames simply by descending through the subgoal trie branches. However, given a particular *subgoal frame*, there is no way to reach the respective *table entry*, because the first structure has no knowledge of its ancestors in the subgoal trie branch. Since this is an essential step in the storing procedure of the mentioned model, it then becomes necessary to augment the *subgoal frame* with a pointer to its parent node.

Another important decision regarded the database communication channels and its efficiency. Given the importance of keeping YapTab's performance pristine, or at least, reduce impact to the minimum, the deployment of a fast database access mechanism is highly desirable. For that reason, DBTab's communication with the RDBMS is mostly done through the MySQL *C API for prepared statements*. This library defines specialized functions that, on successful parsing and/or execution of the submitted SQL statements, return specialized data structures reflecting the state of the server buffers. Obviously, these must be stored somewhere for future use. The different implementation traits of the two mapping models were determinant in the process choosing the possible locations:

In the Datalog model a predicate table space is kept in a single table. Since the mapping relation attributes cannot be null, all of the arguments for an insertion statement must be initialized before its execution. Hence, if all arguments are assumed to be variable, one can formulate a generic statement that enables the insertion of any subgoal of the predicate. It then seems obvious that the best way to store the prepared statement handler is the *table entry* structure. On the contrary, a particular subgoal instance is clearly context dependent, since some of its subterms may be constant values. For that reason, each *subgoal frame* is a serious candidate to hold a specific prepared statement handler. The selection statement can describe specific filtering conditions involving the subgoal arguments that are bound to non-variable atomic terms, placing all found free variables in the list of fields to retrieve;

In the hierarchical model each subgoal is associated with a specific relation whose

name conveys an uniquely identifying suffix, as mentioned back in section 4.2.2, hence prepared statements cannot be generalized. For that reason, each *subgoal frame* must hold a pair of prepared statement handlers: one to insert the answer trie nodes configuration for the mapping table and one to perform the opposite operation.

These specifications still face some efficiency problems. As mentioned before in sections 2.3 and 4.2, coupled systems face potential bottlenecks when data transactions between the logical and database engines are performed *tuple-by-tuple*. If a single SELECT statement is enough to prevent such a problem during data retrieval, a single INSERT statement is obviously a potential source for trouble. The ideal solution would be to place the entire answer set in a buffer and send it to the database as a single cluster. Fortunately, MySQL provides the exact tool for that. A special trait of the INSERT statement allows users to store several tuples into a table in a single call. The number of inserted rows is arbitrary and is limited only by the size of a specialized buffer residing at server side. For further details, the reader should consult the MySQL reference manual [WA02]. Let us see how clustering, allied with the used of prepared statements, can achieve a good storage performance. MySQL development team states that, in average, an INSERT call executes in time $\mathcal{T}$, consisting of the following fraction:

$$\mathcal{T} = \underbrace{3t}_{connecting} + \underbrace{2t}_{\substack{sending\\query}} + \underbrace{2t}_{parsing} + \underbrace{\mathrm{K}t}_{\substack{inserting\\record}} + \underbrace{\mathrm{I}t}_{\substack{insering\\indexes}} + \underbrace{t}_{closing}$$

where K is the size of each record and I is the number of indexes [WA02]. Assume that a set of 100 tuples is to be inserted into a relation residing at the database, without the resource to prepared statements. The cost of such a storing transactions is

$$\begin{aligned} 100\mathcal{T} &= 100\ (\ 3t\ +\ 2t\ +\ 2t\ +\ \mathrm{K}t\ +\ \mathrm{I}t\ +\ t\ ) \\ &= 300t\ +\ 200t\ +\ 200t\ +\ 100\mathrm{K}t\ +\ 100\mathrm{I}t\ +\ 100t \\ &= \underbrace{400t}_{connection} + \underbrace{400t}_{\substack{transmission\\and\ parsing}} + \underbrace{100\mathrm{K}t\ +\ 100\mathrm{I}t}_{insertion} \end{aligned}$$

The introduction of prepared statements enables a time gain of $400t$, since the SQL statement is sent to the database and parsed only once. Let us now assume that

a cluster of 10 answers is established and that the connecting, sending, parsing and closing subtotals remain the same. In this scenario, the cost of insertion remains the same, but the number of connections is now reduced by a factor of ten. Hence, the total cost of sending the entire tuple-set is now of

$$\begin{aligned}\mathcal{T}^{\star} &= 10\ (\ 3t\ +\ 10\mathrm{K}t\ +\ 10\mathrm{I}t\ +\ t\ ) \\ &= 30t\ +\ 100\mathrm{K}t\ +\ 100\mathrm{I}t\ +\ 10t \\ &= \underbrace{40t}_{connection}\ +\ \underbrace{100\mathrm{K}t\ +\ 100\mathrm{I}t}_{insertion}\end{aligned}$$

The adherence to the formerly devised strategy presents a new problem. Sharing a single INSERT statement by all subgoals of a particular tabled predicate means that one must compromise with a particular cluster size; one cannot send all answer tuples at once, because their number varies according to the size of the handled subgoal's answer set. On the other hand, when the number of answers is smaller than the established cluster size, the statement cannot be used since the unbound parameters will introduce erroneous tuples into the predicate's mapping relation. It is then necessary to define a second prepared statement to perform this specialized insertion. With these two statements, the answer set may now be split into pieces and sent over to the database in a faster way. Whenever the number of answers is insufficient to fill the cluster's buffer, the originally defined prepared statement can be used to send those answers one by one to the database. Figure 5.6 summarizes the modifications introduced by DBTab in the table space structures.

5.2.2 Prepared Statements Wrappers

Despite its power, MySQL prepared statement structures are minimal in terms of state representation, leaving out some other useful information about the server state that must be obtained through other mechanisms. For instance, meta-data regarding the result of the statement's execution, such as the number of fields in the statement's resulting dataset, their names, types, sizes and offsets, is kept in an additional MYSQL_RES result set structure. If the reader is not familiarized with MySQL C programming API, please refer to the MySQL Reference Manual [WA02] for further explanations.

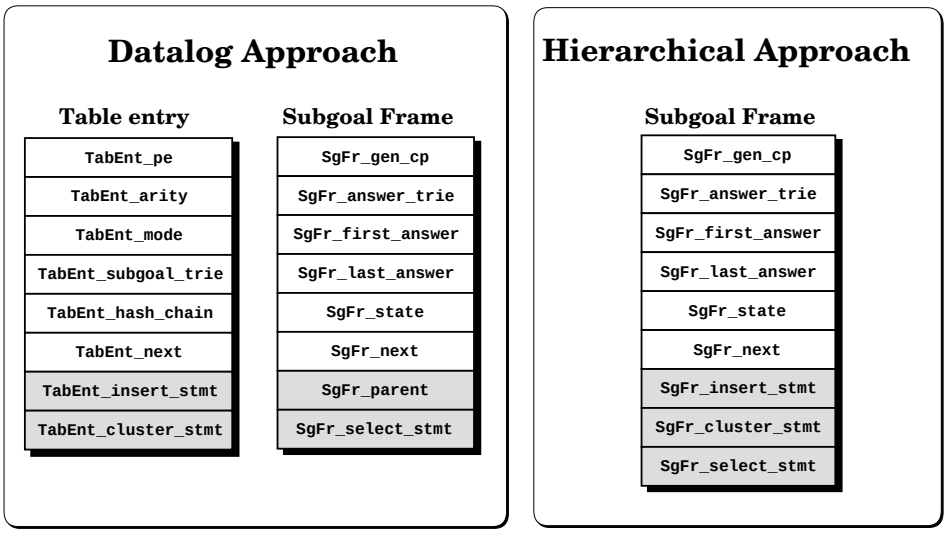

Figure 5.6: Modified YapTab structures

MySQL prepared statements also need client-side defined buffers to collect arguments and/or store tuple attributes retrieved from the tuple sets resulting from the statement's execution. However, these buffers are physically kept outside those structures; in fact, prepared statements simply hold pointers to the buffers memory addresses. For simplicity reasons, all that is related with statement execution should be kept together. To cope with this, a new wrapper structure, denominated `DBTabPreparedStatement`, is introduced in the DBTab middleware layer. Figure 5.7 presents its member fields.

Upon successful parsing of the SQL statements issued by YapTab, the handle returned by MySQL is used to initialize the `statement` field. The `affected_rows` field holds the expected number of rows affected by the execution of the statement. The sub-structure `params` is used to hold information regarding any used input arguments: the count, buffer addresses, respective sizes and null flags. If a tuple set is to result from statement's execution, the `fields` sub-structure is initialized with the same type of information, this time regarding the tuple set attributes. Additional information, such as meta-data, the total number of rows and the actually selected one, is stored in the `records` sub-structure. Despite the similarity of the `affected_rows` and `num_rows` structure fields, the information kept in these fields is in fact different as it results from the call of two distinct MySQL *C API functions*. According to the Reference Manual [WA02], `mysql_affected_rows()` is to be used with non-returning statements such as INSERT, UPDATE and DELETE. Oppositely, `mysql_row_count()` is to be used with result-set returning statements, such as SELECT.

```
typedef struct dbtab_prepared_statement {
  MYSQL_STMT       *statement;
  my_ulonglong     affected_rows;
  struct {
    int        count;
    MYSQL_BIND *bind;
    my_ulong   *length;
    my_bool    *is_null;
  } params;
  struct {
    int        count;
    MYSQL_BIND *bind;
    my_ulong   *length;
    my_bool    *is_null;
  } fields;
  struct {
    MYSQL_RES        *metadata;
    MYSQL_FIELD      *metadata_row;
    my_ulonglong     num_rows;
    my_ulonglong     actual_row;
  } records;
  GenericBuffer    stmt_buffer;
} *DBTabPreparedStatement;
```

Figure 5.7: The prepared statement wrapper

The `stmt_buffer` field is typed as a `GenericBuffer` pointer. This data structure, displayed in Fig. 5.8, is introduced to enable the generic use of the `DBTabPreparedStatement` structure. In the Datalog model, the `stmt_buffer` is initialized with an array of `GenericBuffer` cells whose size is determined as previously explained. Each array cell guarantees enough buffering space for all of the four supported primitive types, thus assuring that any possible primitive value[1] may be sent to/retrieved from the database, either as an argument or as a field value. In the *hierarchical* model, the size of the mapping relation tuple is fixed and universally known. Hence, the `GenericBuffer` structure may be molded to it. For this reason, the `stmt_buffer` is set to point to a single `GenericBuffer` cell.

A practical aspect enables a small optimization that reduces the memory requirements for each `DBTabPreparedStatement`. Since the statement `parameters` and `fields` buffers are sequentially used in each execution context, the required memory space may be shared. The maximum between arguments and fields count is used to set the length of the `stmt_buffer`, `bind`, `length` and `is_null` arrays.

[1]Recall that **(i)** only atomic terms are handled and **(ii)** these vary in size, but atoms and standard integer terms are equally sized.

```
typedef struct generic_buffer {
#if DATALOG_APPROACH
  int    term;
  int    lint;
  double real;
#else /* HIERARCHICAL_APPROACH */
  int    this;
  int    parent;
  int    symbol;
#endif /* HIERARCHICAL_APPROACH */
} *GenericBuffer;
```

Figure 5.8: The multiple term buffer

As previously mentioned in subsection 5.2.1, some table space structures were extended with prepared statements handlers. It now becomes clear that those handlers are none other than `DBTabPreparedStatements` structures. In the *Datalog* model both *table entry* and *subgoal frame* data structures are extended with wrappers, while in the *hierarchical* model only the *subgoal frame* is transformed. Figures 5.9 and 5.10 display the SQL statement templates used in the initialization of the prepared statement wrappers for the *separate tables* variant of the *Datalog* model and for the *hierarchical* model. For the time being, the size of the cluster is omitted for simplicity purposes.

INSERT INTO ss$\mathcal{S}$_$\mathcal{P}n$ (arg*1*,...,arg$n$) values (?,...,?)
INSERT INTO ss$\mathcal{S}$_$\mathcal{P}n$ (arg*1*,...,arg$n$) values (?,...,?), ... ,(?,...,?)

SELECT ⟨*variables*⟩ FROM ss$\mathcal{S}$_$\mathcal{P}n$ AS pn
LEFT JOIN ⟨*join table*⟩ ON ⟨*join condition*⟩
(...)
WHERE ⟨*constants*⟩ ORDER BY pn.order

Figure 5.9: The Datalog model prepared statements

INSERT INTO ss$\mathcal{S}$_$\mathcal{P}n$_Sf (this,parent,symbol) values (?,?,?)
INSERT INTO ss$\mathcal{S}$_$\mathcal{P}n$_Sf (this,parent,symbol) values (?,?,?), ... ,(?,?,?)

SELECT this, symbol FROM ss$\mathcal{S}$_$\mathcal{P}n$_Sf WHERE parent=? ORDER BY this

Figure 5.10: The hierarchical model prepared statements

Additionally, DBTab may place prepared statement wrappers at a global level. The

separate value variant of the *Datalog* model uses these structures to hold the statements responsible for the manipulation of the auxiliary primitive relations belonging to the session context. The statements, known as the *session statements*, are presented in Fig. 5.11. A pair of statements is used for each auxiliary table, one for clustered tuple insertion and one for single tuple insertions. The rationale is the same as before: sending the primitive values one-by-one to the database may lead to a bottleneck.

INSERT INTO ss$\mathcal{S}$_Floats (term,value) VALUES (?,?)
INSERT INTO ss$\mathcal{S}$_Floats (term,value) VALUES (?,?), ... ,(?,?)

INSERT INTO ss$\mathcal{S}$_Longints (term,value) VALUES (?,?)
INSERT INTO ss$\mathcal{S}$_Longints (term,value) VALUES (?,?), ... ,(?,?)

Figure 5.11: Session prepared statements

5.2.3 The DBTab API

The developed API functions constitute a second layer of control. These routines, embedded in the YapTab architecture, establish the interface between top level commands, tabling instructions and database stored procedures. They are responsible for both the traversal and storage of tabling tries during the memory recovery stage and the reloading and reconstruction of the same tries during posterior variant calls. The API functions are briefly presented in this subsection. The full disclosure will be revealed in the remainder of this chapter, as the storage and loading mechanisms are fully explained.

The API comprises the following routines:

`dbtab_init_session(MYSQL *handle, int sid)` starts by validating the database connection handle passed in the first argument. In case of success, the handle is kept for further use in all other functions. The second argument is passed to the `session_register` stored procedure, which is called to effectively start a new session. On successful execution, the stored procedure returns a session identifier that is placed in a global variable. At last, the function initializes the session prepared statements;

`dbtab_init_table(...)` has two distinct prototypes, one for each of the possible mapping models. In the *Datalog* model it receives a *table entry* structure as argument. In the *hierarchical* model it receives a *subgoal frame* structure. In both cases, the *table entry* is accessed (directly or after branch traversal) to obtain the functor and arity of the tabled predicate. In the second case, during the traversal, the term entries of the nodes are concatenated and the resulting string becomes the context suffix. These two (or three) values become the arguments for the `predicate_register()` stored procedure, that in fact creates the new mapping relational table. They are also embedded in the body of the INSERT statement that initializes the specific prepared statement wrappers inside the structural argument;

`dbtab_init_view(subgoal frame sg_fr)` has two distinct prototypes, one for each of the possible mapping models. In the *Datalog* model, it traverses the *subgoal trie* branch from which the frame hangs. Two data vectors, sized accordingly to the arity of the predicate, are used to split the node entries. Every bound term has its value copied its respective position in the first array, while free variable terms have their internal index[2] stored in the second array. By the time the root node is reached, all of the statement's arguments consisting of free variables will have NULL place-holders in the first array[3]. The two arrays are then used to customize the SELECT prepared statement: the first reference to every variable term ends in the list of fields to be retrieved, while every bound term and subsequent references to variable terms end up in the conditional expression, used to refine the search. In the *hierarchical* model, the functor, arity and context suffix are retrieved from global buffers[4] and used to build the SELECT expression. Since the mapping table size is well-defined, no extra operations or extra buffers are required to create the query. Both implementations use the created SQL statements to initialize the prepared statement wrapper inside the received *subgoal frame* argument;

`dbtab_export_trie(subgoal frame sg_fr)` starts by recovering the INSERT prepared statement associated with *subgoal frame* argument. In the Datalog model this

[2]YapTab maintains an indexed array of variable terms found in each subgoal call.

[3]For practical reasons, INSERT prepared statement `GenericBuffer` is used as first array. Recall that in the *inline* variant each field triplet is regarded as a single predicate argument, so the extra required statement arguments are not considered for variable term identification purposes.

[4]These buffers were previously initialized in `dbtab_init_table()`.

is done via the respective table entry, in the *hierarchical* model directly from the argument. In both models, it cycles through the answer trie, traversing each branch, substituting the statement's NULL parameters with the node entries and executing the prepared statement to create the respective tuples in the mapping relational table. If in the *separated values* variant, it additionally executes calls to the auxiliary session prepared statement that handle the insertion of the supported primitive type values. The details of this function are presented further ahead in Fig. 5.15;

`dbtab_import_trie(subgoal frame sg_fr)` starts the data retrieval transaction, executing the SELECT prepared statement associated with the subgoal frame passed as argument. It then navigates the resulting tuple set to reconstruct the answer trie entries[5]. The details of this function are presented further ahead in Fig. 5.18;

`dbtab_free_table(...)` has two distinct prototypes. Just like before, the *table entry* structure is consulted to obtain the functor, arity and possible context suffix that are to be passed as arguments to the `predicate_unregister()` stored procedure, thus dropping the mapping table. It also frees the prepared statement wrappers inside the received data structure;

`dbtab_free_view(subgoal frame sg_fr)` frees the SELECT prepared statement associated with the *subgoal frame* structure passed as argument;

`dbtab_kill_session(void)` frees the customized session prepared statements and kills the currently opened session by calling the `session_unregister()` stored procedure;

A practical example may help to clarify the scenario. Figure 5.12 presents the state of the *table entry* and *subgoal frame* structures after the `dbtab_init_table()` and `dbtab_init_view()` calls on both mapping models. Notice how **(i)** the prepared statement wrappers are distributed in the table space and **(ii)** the context suffix reflects the node entries in the *hierarchical* model.

[5]Later in this chapter, an alternative approach in which this last step does not take place will be introduced.

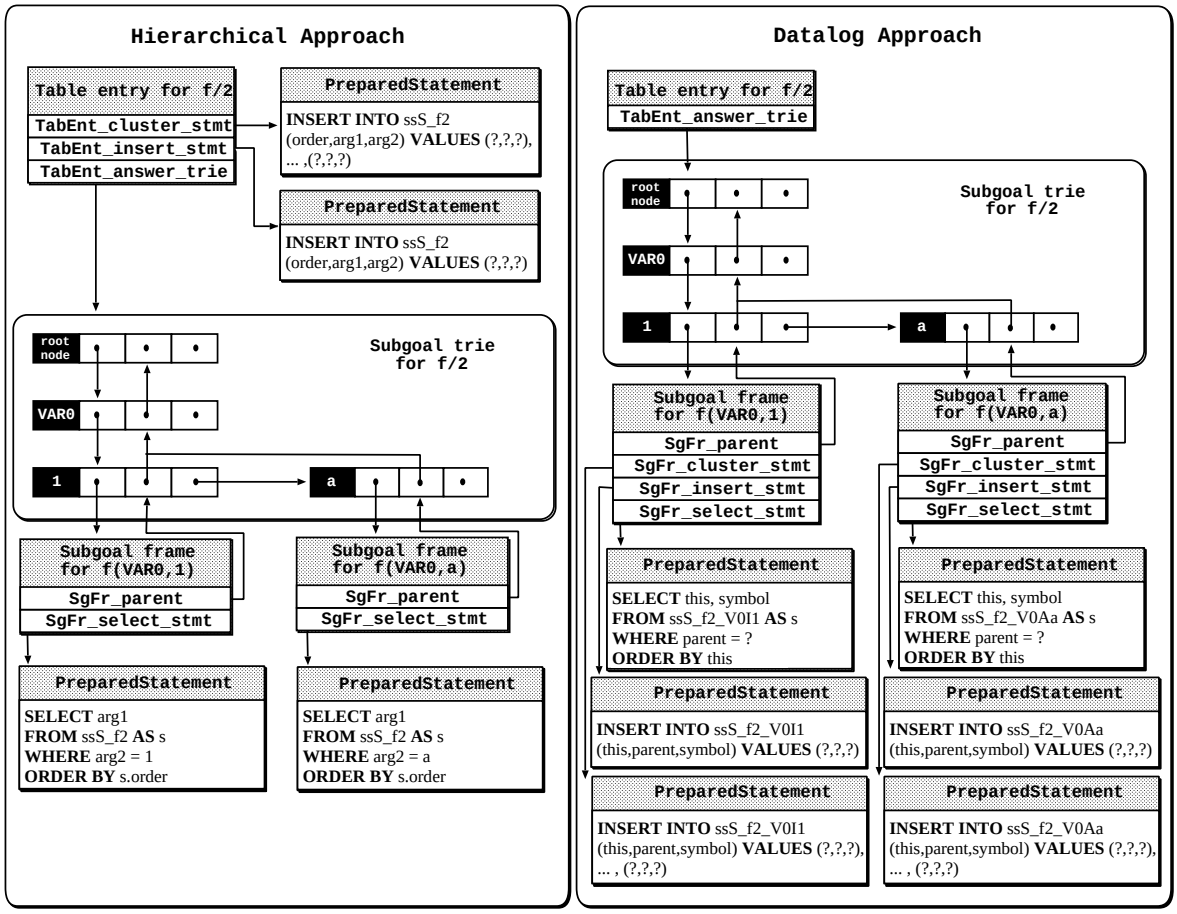

Figure 5.12: Initialized table space structures

5.2.4 The Top-Level Predicates

The top-level built-in predicates constitute the third and final layer of control. Two new predicates are added and three pre-existing ones are slightly changed to act as front-ends to the developed API functions.

To start a session, `tabling_init_session/2` must be called. It takes two arguments, the first being a database connection handler[6] and the second being a *session identifier*. The identifier can either be a free variable or an integer term. The arguments are then repassed to `dbtab_init_session()`. This will either start a new session, binding the new identifier to the variable argument, or restart the previously existing session identified by the given integer.

The `tabling_kill_session/0` terminates the currently open session by calling

[6]This argument is obtained using YAP's MYDDAS package

`dbtab_kill_session()`. It needs no arguments because the session's identifier is kept since the session starts.

YapTab's directive '`:- table p/n.`' sets predicate p of arity n up for tabling. For the *Datalog* model, an expanded version of `table/1` passes the newly created *table entry* structure to `dbtab_init_table()` to set up the process of relational mapping generation.

The `abolish_table/1` built-in predicate is expanded to call the `dbtab_free_table()` function before releasing the table entry. Every subgoal frame found under this entry becomes the entry parameter of `dbtab_free_view()`. Both in YapTab and DBTab, the `abolish_all_tables/0` predicate can be used to dispose of all table entries: the action takes place as if `abolish_table/1` was called for every tabled predicate.

5.2.5 Exporting Answers

Recall section 3.3 and the *least used algorithm*. Whenever memory becomes scarce, the algorithm starts to look for inactive tables to delete. This search takes place inside a function called `recover_space()`, whose pseudo-code is illustrated in Fig. 5.13. For every table in such condition, the `dbtab_export_trie()` API function is called to initiate a new data transaction. Notice how the state of the *subgoal frame* structure is changed to signal the table dumping into the relational database.

In order to illustrate the most important aspects of the answer set storage process, the focus will be set on the *Datalog* model implementation. Since the underlying rationale has already been presented in chapter 4, we consider that dissecting the *hierarchical* model implementation with the same level of detail would be tedious for the reader and would not bring any new contributions to the presentation.

Prior to the analysis of the `dbtab_export_trie()` itself, it is important to introduce a special feature that has a major influence in the process. Figure 5.14 will be used to illustrate the explanation. As previously mentioned, two prepared statements are used to perform the storage of answer sets into the relational models; one statement is capable of sending a cluster of tuples, the other one is used to send tuples one by one. However, the storing algorithm described back in subsection 4.2.1 was designed to traverse the answers tries handling one answer a time. Obviously, some sort of

```
recover_space(STR_PAGES) {
    sg_fr_ptr sg_fr = inact_recover;
    do {
       if(SgFr_first_answer(sg_fr) &&
          SgFr_first_answer(sg_fr) != SgFr_answer_trie(sg_fr)) {
          if(SgFr_state(sg_fr) == complete) {
             dbtab_export_trie(sg_fr);
             free_answer_trie_branch(SgFr_answer_trie(sg_fr));
             SgFr_state(sg_fr) = stored_complete;
          } else {
             SgFr_state(sg_fr) = ready;
          }
          SgFr_first_answer(sg_fr) = NULL;
          SgFr_last_answer(sg_fr)  = NULL;
       }
       (...)
       if(sg_fr) {
          sg_fr = SgFr_next(sg_fr);
       }
    } while(free_pages(GLOBAL_PAGES_void) == free_pages(STR_PAGES));
    inact_revover = sg_fr;
}
```

Figure 5.13: Pseudo-code for `recover_space()`

adaptation is required in order to allow the use of the clustered insertion statement. The easiest way to meet this end without disrupting the established work-flow is to force the two statements to share their internal buffers.

The clustered statement buffers are divided into several frames accordingly to the arity of the tabled predicate. When cycling through the answer set trie, the algorithm assigns a buffer frame to each trie branch using simple pointer arithmetic; the frame's size is used to calculate the offset of the desired frame, which is added to the initial address of the clustered buffer. When all frames are occupied, the clustered statement may be is executed and the process may start all over again, until the entire answer trie is processed.

The pseudo-code of the `dbtab_export_trie()` function in shown in Fig. 5.15. The storing process begins with a call to the `dbtab_start_transaction()`, thus signaling the start of a new data transaction. Next, the subgoal trie branch contents are placed inside the template array. Control proceeds cycling through the answer trie branches, copying the template contents to the branch's assigned buffer frame and binding the YAP terms stored within each branch nodes to the buffer's NULL parameters.

The next call occurs only in the *separate value* variant and regards the supported primi-

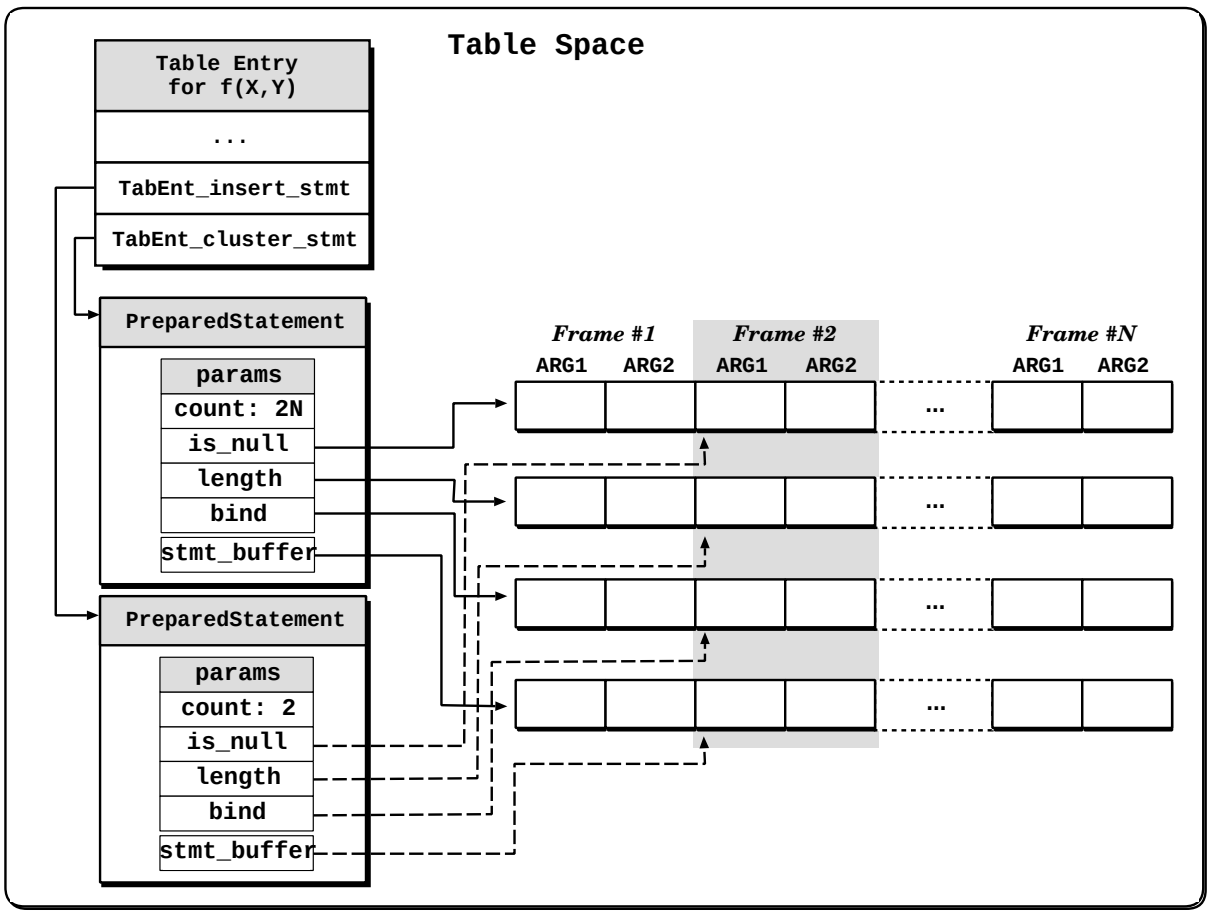

Figure 5.14: Prepared Statements sharing internal buffers

tive values storage. The details, obscured by the single call to `store_primitive_terms()`, include the initialization of the *session* statements' buffers in a similar way to that previously explained. The input parameters of the currently active frame buffer are bound to the next sequential value for application-masked terms[7] and to the term

[7]The first value of this sequence, obtained every time the `dbtab_start_transaction()` stored

primitive value itself. The identifier's value is bound to the subgoal's statement respective argument, thus creating a kind of a *foreign key*. Notice that repeated occurrences of a primitive value in the tabling tries result in multiple tuples in the respective auxiliary table, each one of them presenting a different sequential key. The alternative would be to introduce an additional statement that would search for a previously stored occurrence of the primitive and return its key if possible. However, in order to obtain some probable gain in terms of space, one would undoubtedly introduce an undesirable performance overhead.

As previously mentioned, the clustered insertion statement is executed only when the clustered buffer is full. This means that when the answer trie traversal cycle is finished, some initialized buffers may still have not been sent to the database. The situation is handled by the final cycle, which executes the single insertion prepared statement, thus adding the remaining tuples to the mapping relation.

Finally, the state of the transaction is evaluated. A COMMIT occurs if and only if all INSERT statements are executed correctly. In this case, the subgoal trie is removed from memory space. A ROLLBACK operation is performed whenever errors occur during transaction.

Figure 5.16 illustrates the final result of the described process for all proposed storage schemes. The shaded answer trie is removed from memory at the end of the transaction. Notice how

- in the *hierarchical* model, the small optimizations discussed back in section 4.2.2 are implemented saving the space of three tuples from the relation;

- in the *separate value* variant, the ARG*1* field of the second and third tuples hold the keys for the auxiliary tables records.

- in the *inline value* variant, the ARG*1* field of the second and third tuples refer to the valid primitive value;

procedure is called, is kept in main memory.

```
dbtab_export_trie(SubgoalFrame sg_fr) {
    GenericBuffer template, terms;

    frame_no     = 0;
    insert_stmt  = TabEnt_insert_stmt(SgFr_tab_ent(sg_fr));
    cluster_stmt = TabEnt_cluster_stmt(SgFr_tab_ent(sg_fr));

    dbtab_start_transaction();

    bind_subgoal_terms(answer, template);
    answer   = SgFr_first_answer(sg_fr);

    while (answer != NULL) {
        /* shift frame and prepare record */
        terms = PS_BUFF_FRAME(cluster_stmt,frame_no);
        copy(template,terms);

        bind_answer_terms(answer, terms);
#if SEPARATE_VALUE
        store_primitive_terms(terms);
#endif
        if(frm_no == CLUSTER_FRAME_COUNT) {
            commit &= exec_prep_stmt(insert_stmt,terms);
            frame_no = 0;
        } else {
            frame_no = ++;
        }
        answer = TrNode_child(answer);
    }
    for(i=0; i<frame_no; i++) {
        terms = PS_BUFF_FRAME(cluster_stmt,i);
#if SEPARATE_VALUE
        /* Send last buffered primitive terms */
        store_primitive_terms(terms);
#endif
        commit &= exec_prep_stmt(insert_stmt,terms);
    }

    dbtab_finish_transaction(commit);
}
```

Figure 5.15: Pseudo-code for `dbtab_export_trie()`

5.2.6 Importing Answers

After answer trie dumping, the first call to one of YapTab's `table_try`, `table_try_me` or `table_try_single` instructions finds the state of the *subgoal frame* set to *stored*. This immediately triggers a call to `dbtab_import_trie()`, the routine responsible for the execution of the specific SELECT statement that is used to fetch all answers for the subgoal. Figures 5.17 and 5.18 show respectively the adaptation of those instructions to DBTab and the pseudo-code for the data import routine `dbtab_import_trie()`.

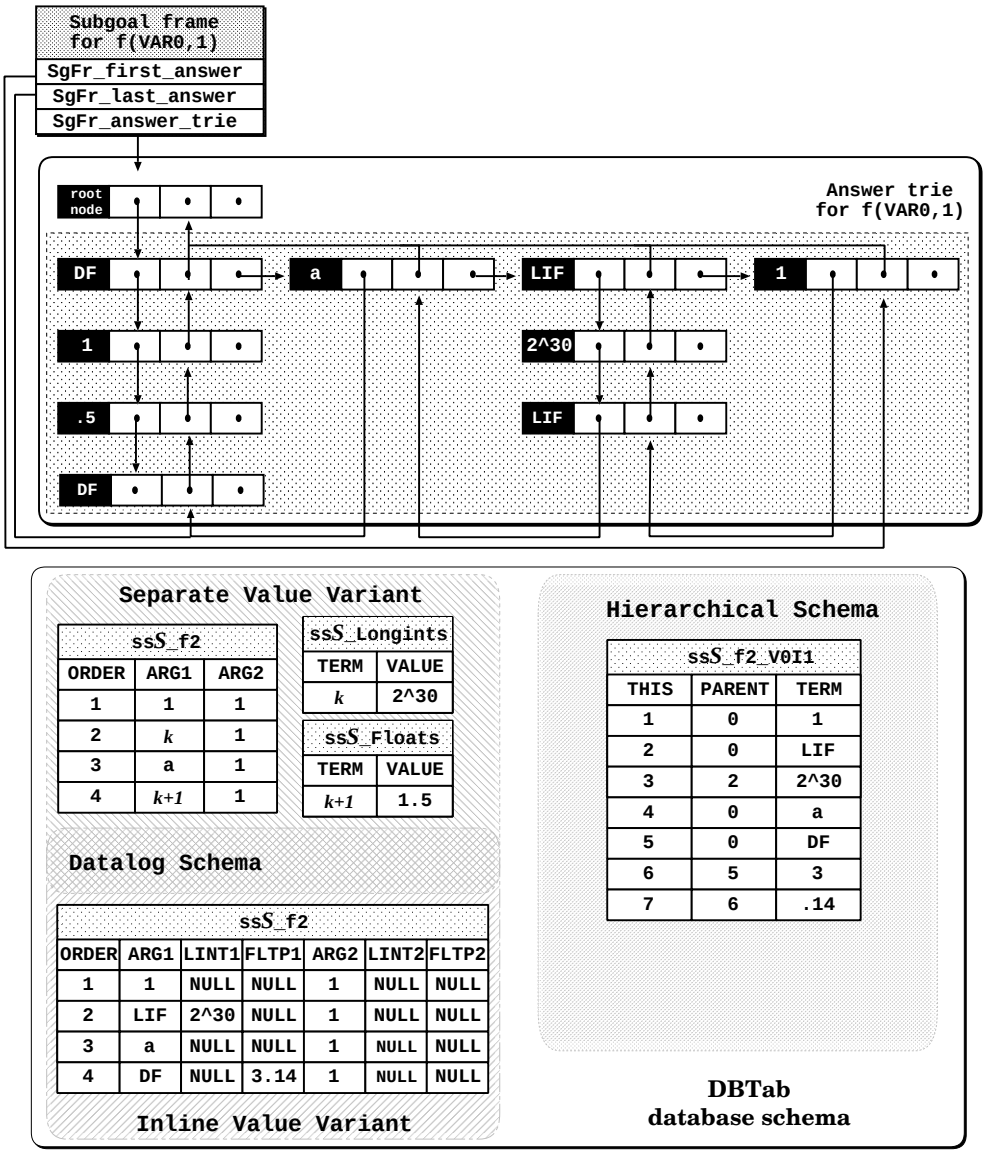

Figure 5.16: Exporting $f(X, 1)$: three relational mappings

The body of routine is implemented differently in the two proposed mapping models. Once again, the focus will be set on the *Datalog* model implementation, since the other implementation underlying rationale has already been presented in chapter 4.

Back in chapter 4 we have discussed how the different loading algorithms pose specific queries to the database generate and use the different tuple sets to reload the answer tries. These differences are again illustrated in Fig. 5.19.

The *hierarchical* model fetches the entire tuple set from the database, using all attributes to reconstruct answer trie nodes and establish their order properly. No

```
PBOp(table_try, ld)
    sg_fr_ptr sg_fr;
    (...)
    } else if (SgFr_state(sg_fr) == stored_complete) {
        dbtab_import_trie(sg_fr);
    }
    (...)
ENDPBOp();

PBOp(table_try_me, ld)
    sg_fr_ptr sg_fr;
    (...)
    } else if (SgFr_state(sg_fr) == stored_complete) {
        dbtab_import_trie(sg_fr);
    }
    (...)
ENDPBOp();

PBOp(table_try_single, ld)
    sg_fr_ptr sg_fr;
    (...)
    } else if (SgFr_state(sg_fr) == stored_complete) {
        dbtab_import_trie(sg_fr);
    }
    (...)
ENDPBOp();
```

Figure 5.17: Modified YapTab instructions

particular attention must be payed to any of the tuple's attributes, safe from the case in which 32-bit integer or floating-point terms are to be inserted into the trie: in that case, an additional delimiter term must be inserted after the last valid tuple in order to comply with YapTab's answer branch construction protocol.

The Datalog model may not retrieve the entire tuple set. The refinements placed in the search condition, within the WHERE clause, shorten the retrieved fields list, thus reducing the amount of data returned by the server. The returned $\texttt{arg}_k$ attributes may be immediately followed by additional columns in case any values from the auxiliary primitive tables are to be fetched. To determine the type of the retrieved term, the focus is set on the $\texttt{arg}_k$ attributes, where no NULL values can be found. Additional columns are regarded as possible value-holders for answer terms only when these *main* fields convey long atomic masked sequential terms. In such a case, the first additional non-NULL attribute placed to the right of $\texttt{arg}_k$ supplies for the value that is used to create the specific YAP term.

After the SELECT prepared statement's execution, the resulting tuple set is available

```
dbtab_import_trie(SubgoalFrame sg_fr) {
    select_stmt = SgFr_select_stmt(sg_fr);
    dbtab_start_transaction();
    exec_prep_stmt(select_stmt);

    /* switch on the number of rows */
    if (PS_NROW(select_stmt) == 0) {   // no answers
      SgFr_first_answer(sg_fr) = NULL;
      SgFr_last_answer(sg_fr) = NULL;
      SgFr_answers(sg_fr) = NULL;
      return;
    }

    /* handle the tuple set */
#if REBUILD_TREE
    do {                                              // multiple answers
      offset = prep_stmt_fetch(select_stmt,NULL);
      answer = bind_answer_record(select_stmt);   // bind the term array

      ans_node = answer_search(sg_fr,answer);     // insert/check
      TAG_AS_ANSWER_LEAF_NODE(ans_node);

      if(SgFr_first_answer(sg_fr) == NULL) {
        SgFr_first_answer(sg_fr) = ans_node;
      } else {
        TrNode_child(SgFr_last_answer(sg_fr)) = ans_node;
      }
      SgFr_last_answer(sg_fr) = ans_node;
    } while(offset)
    prep_stmt_free_resultset(select_stmt);        // free the tuple set
    dbtab_finish_transaction(TRUE);
#else /* BROWSE_TUPLESET */
    SgFr_first_answer(sg_fr) = PS_TOP_RECORD(select_stmt);
    SgFr_last_answer(sg_fr)  = PS_BOTTOM_RECORD(select_stmt);
#endif /* BROWSE_TUPLESET */
    dbtab_finish_transaction(TRUE);
}
```

Figure 5.18: Pseudo-code for `dbtab_import_trie()`

for usage. Two possible strategies may be used to supply these answers back to the YapTab engine.

A first possible strategy is to use the retrieved tuple set to rebuild the *answer trie* and discard it afterwards. This corresponds to the first code block in `dbtab_import_trie()` function as shown in Fig. 5.18. The records are traversed sequentially in a *top-to-bottom* fashion and the retrieved attribute values (answers) are used to create the substitution factoring of the respective subgoal call, exactly as when the tabling `new_answer` operation occurs. By the end of the cycle, the entire answer trie resides in the table space and the record-set can then be released from memory. This

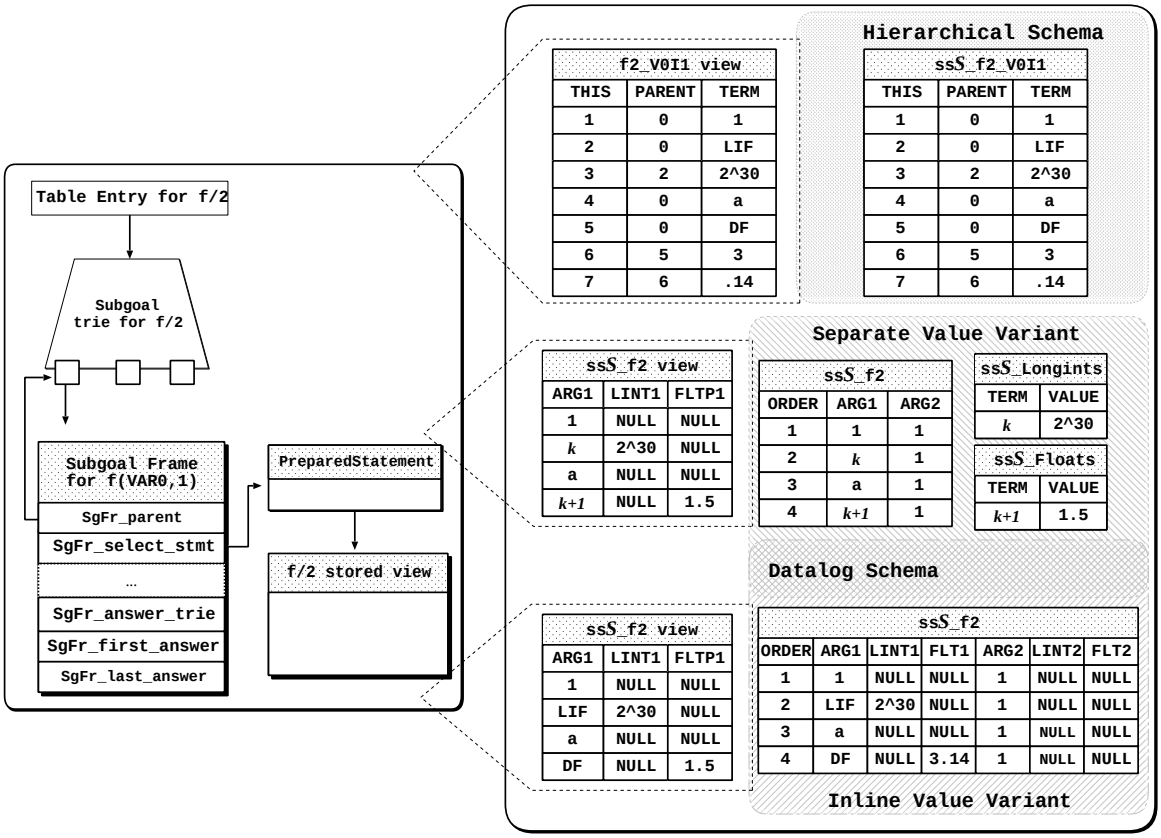

Figure 5.19: Importing $f(X, 1)$: three relational mappings

strategy requires no alteration to the YapTab's implemented API, except for the call to `dbtab_import_trie()` in specific points of the `table_try`, `table_retry` and `table_try_single` instructions.

Sometimes, it may not be advisable to reconstruct the complete *answer trie* to find out if the answer set is useful or not in the present execution context. For instance, a small initial subset of the answers may be enough to decide if a subgoal is useful to help solving a particular goal. In order to prevent undesirable time waste, all subsequent subgoal calls could fetch their answers directly from the retrieved tuple set simply by browsing through its contents. The second code block in Fig. 5.18 illustrates how the ancillary YapTap constructs can be used to implement the idea.

This second strategy enables a memory gain, since **(i)** the relational representation of trie nodes dispenses the three pointers and focus on the symbol storage, hence the size of the memory block required to hold the answer trie can be reduced by a factor of four; and **(ii)** longer atomic terms representation narrows its memory requirements at

least down to one eighth of the usually occupied memory because these type usually require three or four trie nodes to be stored in memory.

Despite its possible benefits, the browsing strategy presents two significant drawbacks. First of all, it may only be used with completed subgoal calls, whose respective answer tries never change. For incomplete answers, this will simply not work, due to the inability to add new answers to the tuple set. Second, it is only advantageous when used in conjunction with the *Datalog* model. DBTab navigates this compacted structure using tuple-length offsets to swiftly access the next answer and its subterms. The *hierarchical* model tuples, conveying mostly trie dependent information, present more sparsely distributed term values, possibly distributed for more than one tuple (in case the term is a floating-point one). This makes the task of searching for term values more difficult, while increasing the complexity of the algorithm.

Figure 5.20 shows how this strategy is used in a particular runtime example. The left side box presents the state of the *subgoal frame* after answer collection for $f(Y, 1)$. The internal pointers are set to the first and last rows of the record-set. Another YapTab internal structure, the *loader choice point* is extended with a field destined to hold the offset from the top of the record-set to the last consumed answer. A loader choice point is a WAM choice point augmented with the offset for the last consumed record and a pointer to the subgoal frame data structure.

This strategy requires an extra function, `dbtab_load_next_answer()`, to browse the tuple set and bind the answer terms in the subgoal in hand. The process is quite simple, as illustrated in Fig. 5.21. When loading answers, the first record's offset, retrieved from the *loader choice point* `cp_last_answer` field , is added to the record-set address kept in *subgoal frame* `SgFr_first_answer`. This offset is used to fetch a tuple whose field values are then used to bind the free variable terms in the `subs_ptr` array. The new offset is placed inside the choice point that is then sent back to the *local stack*. When backtracking occurs, the choice point is reloaded and the last recorded offset is used to proceed to the next answer. When an invalid offset past the end of the tuple set is reached, the *loader choice point* is discarded thus signaling the positioning at the last answer. The ongoing evaluation is then terminated and the tuple set is discarded.

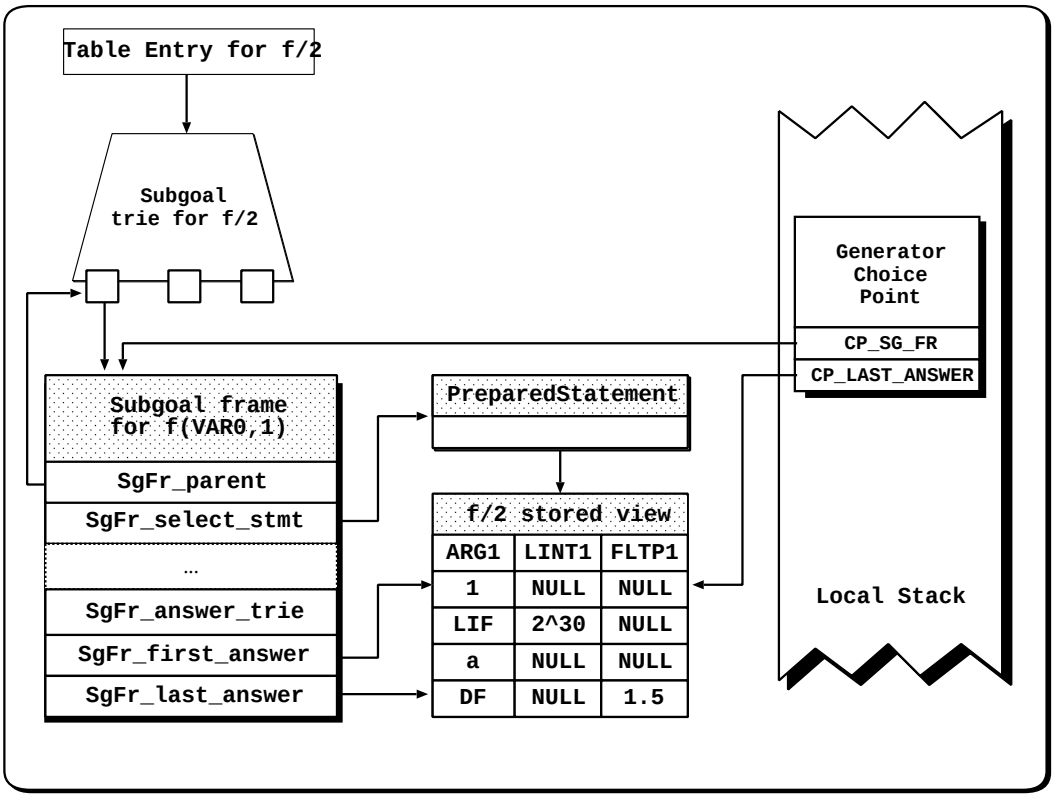

Figure 5.20: Importing $f(Y, 1)$: browsing the tuple set directly

5.3 Chapter Summary

In this chapter, the implementation of DBTab was stressed. First, we have briefly contextualized the developed work, establishing the connections to Rocha's work on efficient support for incomplete and complete tables in the YapTab tabling system [Roc07] and on the work of Ferreira *et al.* on coupling logic programming with relational databases [FRS04]. Next, the MySQL relational database management system chosen to support DBTab was introduced and its major assets were discussed. The transformations produced in YapTab to enable the implementation of relational tabling was discussed afterwards. Topics like the developed API and the changes to Yaptab's *table entry*, *subgoal frame* and *loader choice point* structures, the `recover_space()` procedure and the `table_try` instruction family of tabling instructions were covered. At the end, we have presented an alternative way to inquire for subgoal answers without reconstructing the answer trie.

```
void dbtab_load_next_answer(subgoal frame sg_fr,
                loader choice point l_cp,CELL subs_ptr) {
  int i, n, subs_arity;
  Term t;
  PreparedStatement select_stmt;
  MYSQL_ROW_OFFSET offset;

  if((subs_arity = *subs_ptr) == 0)
    return;

  select_stmt = SgFr_select_stmt(sg_fr);
  offset = CP_last_answer(l_cp) + SgFr_first_answer(sg_fr);
  offset = prep_stmt_fetch(select_stmt,offset);

  for(i=0, n=subs_arity; n>=1; n--) {
    CELL subs_var = subs_ptr + n;
    t = BIND_BUFF_PNTR(select_stmt,i);
    i++;
    if (IsAtomOrIntTerm(t)) {
      Bind(subs_var, t);
    } else if(IsApplTerm(t)) {
      Functor f = (Functor)t;
      if(f == LongIntFunctor) {
        int k;
        k = BIND_BUFF_PNTR(select_stmt,i+LINT_FSET);
        t = MkLongIntTerm(k);
        Bind(subs_var, t);
      } else if(f == DoubleFunctor) {
        double d;
        d = BIND_BUFF_PNTR(select_stmt,i+FLTP_FSET);
        t = MkFloatTerm(d);
        Bind(subs_var, t);
      }
    } else {
      Yap_Error(INTERNAL_ERROR, TermNil, "unknown type tag (dbtab_load_next_answer)");
    }
    // Locate next ARGi    // Locate next ARGi
    for(;(i<PS_FLDS_COUN(select_stmt)) && !VW_FIELD_IS_ARG(i);i++);
  }
}
```

Figure 5.21: The `dbtab_load_next_answer()` function

Chapter 6

Performance Analysis

In this chapter, an analysis on DBTab's performance over a set of benchmark programs is performed. The first part of the chapter presents an overall evaluation of the overheads introduced by the relational storage extension, comparing DBTab's different approaches performance to that of YapTab for a standard evaluation. The second part discusses and draws some preliminary conclusions on the obtained results.

6.1 Performance on Tabled Programs

To place performance results in perspective, a first batch of tests was performed both in YapTab and DBTab environments. The detailed analysis of DBTab's performance allowed us to assess the efficiency of the relational tabling implementation based on measured overheads introduced by the data transactions between the logical and the database engines. The experimental environment was *Humpty Dumpty*, a white-line computer featuring a Pentium®4 XEON 2.6GHz, 2048 Mbytes of main memory and running a Linux 2.6.18-2869.fc6PAE kernel. DBTab is implemented over the YAP-5.1.1 engine and uses MySQL 5.0.2 for relational database management system.

In order to obtain a valid and credible comparison between each of the proposed relational models, they should be tested in similar circumstances. Recall that the *Datalog* model is restricted to atomic constants, more specifically to three simple types of atomic constants. These are atomic strings, floating-point numbers and integer numbers, even though these last are divided into two sub-classes: standard (those

which fit the non-mask part of the term) and longints (those whose size matches the complete term). Hence, for fairness sake, only the enumerated primitive types were used for evaluation purposes.

A graph connectivity problem, presented in Fig. 6.1, was used to measure both YapTab and DBTab performances in terms of answer generation. The program's main goal is to determine all existing paths starting from a particular node of the graph. The `go/1` predicate is the top query goal. It determines the benchmark predicates performance by calculating the difference between the main program's uptime before and after the benchmark execution.

Benchmark predicates are defined by two clauses: a first one that executes a call to the tabled goal followed by the automatic failure mechanism and a second one that ensures the successful completion of the top query goal. Two tabled predicates were used to determine all existing paths in the loaded graphs. Predicate `path/2`, henceforth called *benchmark #1*, helped us to establish minimum performance standards, since its answer set is relatively easy to obtain and handle through the use of standard tabling. However, every day problems often require more sophisticated algorithms, usually with sub-components that perform heavy calculations or some sort of input/output operation. With this in mind, predicate `write_path/2`, henceforth called *benchmark #2*, was introduced. It basically results from the addition of a call to an output predicate, `writeln/1`, at the very end of both clauses of `path/2`. Our aim was to determine if DBTab could improve the overall performance of the main program when such heavy operations were needed during answer set computing.

For a relatively accurate measure of execution times, the test program was executed twenty-five times for each problem instance and the average of measured times, in milliseconds, was found. Graph topologies and sizes changed through the test phase, as an attempt to provide a larger insight on overall performance. Answer sets were sent to the database in clusters of 25 tuples. Each of the tested graphs had its nodes labelled by a different value of a single primitive type. Mind that both engines were running in the same machine, which might have affected performance.

Results of the previously described test batches were summarized in two kinds of tables. The first kind summarizes the performance of YapTab regarding the execution of the two available benchmark predicates. The reported topics include the number vertexes belonging to the graph, the number of possible paths in the graph (answers),

```
% Connection handle creation stuff ...
:- consult('graph.pl').
:- tabling_init_session(Conn,Sid).

% Utilities
writeln([])   :-nl.
writeln([X|L]):-write(X),writeln(L).

% path(A,Z) succeeds if there is a path between A and Z
:- table path/2.
path(A,Z):- path(A,Y), edge(Y,Z).
path(A,Z):- edge(A,Z).

% write_path(A,Z) succeeds if there is a path between A and Z to be written
:- table write_path/2.
write_path(A,Z):- write_path(A,Y), edge(Y,Z), writeln(['(',A,',',Z,')']).
write_path(A,Z):- edge(X,Y), writeln(['(',A,',',Z,')']).

benchmark(1):- path(A,Z), fail.
benchmark(1).

benchmark(2):- write_path(A,Z), fail.
benchmark(2).

go(N) :- statistics(walltime, [Start,_]),
         benchmark(N),
         statistics(walltime, [End,_]),
         Time is End-Start,
         writeln(['WallTime is ',Time]).
```

Figure 6.1: The test program

the primitive types used to label the vertexes, the number of trie nodes used to represent the computed answers in the table space, the total amount of memory in Kbytes required to store the answers tries, the amount of time spent in solving the benchmark predicates and the amount of time required to produce a second answer to the same queries, after table completion. A second type of tables established a comparison between all the proposed database storage models. Reported topics include the primitive type used to label the vertexes, the number of vertexes belonging to the tested graphs and the primitive type used in their labeling. This size is compared to the original memory space requirement for the predicated tables. Next comes the amount of time spent in data transactions between the logical and the database managing engines, in both directions. Each of these time measurements is compared with the time spent in solving both the benchmarks. A final block of columns is used to exhibit the size in Kbytes of the retrieved tuple sets and the execution times for the tuple set browsing strategy, comparing its performance in terms of time to the

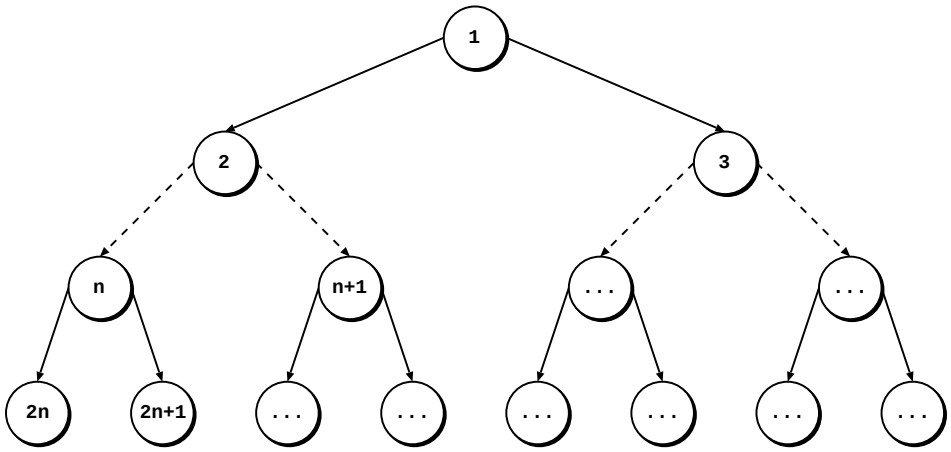

Figure 6.2: Binary tree

standard tabling tries. In both sorts of tables, execution times strings have the format *Minutes:Seconds.Milliseconds*.

At the end of each case study, a chart is used to visually compare the presented performance figures. The X axis measures the number of nodes, the Y axis the time in Milliseconds. Green squares and diamonds reflect YapTab's times for benchmarks #1 and #2. Circles and triangles reflect the three alternative implementations storage and retrieval times; Blue corresponds to the *hierarchical* model, Yellow and Red corresponds to the *separate values* and *inline* variants of the *Datalog* model. The lines in between the discrete values illustrate performance trends. Trend equations are made available to enable a better comparison. At the left side of the charts, the equations refer to benchmark #1 and data retrieval operations performance; at the right side, the equations refer to benchmark #2 and data storage operations performance.

6.1.1 Case Study #1: Binary Tree

The first chosen topology is the *complete binary tree*. This is a binary tree in which **(i)** every node has zero or two children, **(ii)** all leaves are at the same depth and **(iii)** each vertex labeled k has its direct children labeled $2k$ and $2k + 1$. A finite complete binary tree $\mathcal{T}_n$ with n vertexes has a depth of $log_2(n)$ and a total count of $n - 1$ edges. Figure 6.2 illustrates the produced structures.

Table 6.1 summarizes YapTab's execution times for the benchmark programs when run

against this type of graphs. It is clear that label types have an important influence on the execution times. In average, answer sets generation using long integer and floating-point labels is 1,4 and 1,7 times slower than when integer labels are used. As expected, the input/output operation has a major impact on performance. Comparing the time spent in solving benchmarks #1 and #2, it becomes clear that the second predicate is much slower than the first. In average, the first predicate spends 44 times the time required by the second to terminate.

Vertexes	Answers	Type	Nodes	Memory	Bm #1	Bm #2	Traverse
256	1538	Integer	1666	33	00:00.001	00:00.025	00:00.001
		Longint	3459	68	00:00.001	00:00.054	00:00.001
		Double	5124	100	00:00.002	00:00.071	00:00.002
1024	8194	Integer	8706	170	00:00.005	00:00.217	00:00.005
		Longint	17923	350	00:00.007	00:00.357	00:00.007
		Double	26628	520	00:00.011	00:00.445	00:00.011
4096	40962	Integer	43010	840	00:00.026	00:01.210	00:00.026
		Longint	88067	1720	00:00.046	00:01.845	00:00.046
		Double	131076	2560	00:00.067	00:02.220	00:00.067
16384	196610	Integer	204802	4000	00:00.129	00:06.274	00:00.129
		Longint	417795	8160	00:00.188	00:10.452	00:00.188
		Double	622596	12160	00:00.201	00:11.856	00:00.201

Table 6.1: YapTab's times for benchmarks #1 and #2 with the Binary Tree

Table 6.2 statistics show that DBTab's impact on the performance of YapTab reflects mainly during tuple storage. For the *inline* variant of the *Datalog* model, the median overhead is of 36,1 times the amount of time required to execute benchmark #1, within an interval roughly ranging from 17,9 to 69 times. For the *separate value* variant, the median overhead grows to 57,3 times the execution time of benchmark #1, within an interval ranging from 34,1 to 127 times. For the *hierarchical* model, the median overhead is of 35,8 times the computation time, within an interval ranging from 18,5 to 62 times. When input/output is performed, the discussed ratios drop abruptly. For the *inline* variant, the median overhead for storage is 0,7 times the amount of time required to execute benchmark #2, within an interval ranging from 0,5 to 2,7 times. For the *separate value* variant, the median overhead grows to 1,2 times, within an interval ranging from 0,7 to 2,7 times. For the *hierarchical* model, the median overhead is of 0,4 times the computation time of benchmark #2, within an interval ranging from 0,4 to 2,3 times.

On the contrary, retrieve operations are quite inexpensive compared with the answer set computing time. For the *inline* variant of the *Datalog* model, the median induced overhead is of 8,3 times the amount of time required by benchmark #1 to terminate, within an interval of 5,0 to 16,7 times. For the *separate value* variant, the median overhead grows to 10,3 times, within an interval ranging from 8,0 to 17,5 times. For the *hierarchical* model, the median overhead is of 5,8 times the computation time of benchmark #2, within an interval ranging from 3,6 to 13,7 times. The discussed values suffer a significant decrease for benchmark #2. In fact, the ratios reveal an actual speedup rather than an overhead. For both the variants of the *Datalog* model, the median retrieval time is of 0,2 times the amount of time required to execute benchmark #2, within an interval ranging from 0,1 to 0,3 times, although the *inline variant* is slightly faster than its *separate value* counterpart. The *hierarchical* model median time requirement is of 0,1 times the computation time of benchmark #2, within an interval ranging from (nearly) 0,0 to 0,2 times.

The last two columns help to understand the performance of *answer set browsing*. For the *Datalog* model, the retrieved tuple set size equals in average 0,4 of the correspondent answer trie size, while for the *hierarchical* model the ratio grows up to 0,5. *Answer set browsing* times increase along with the graph size. For the *inline* variant, it takes in average 3,6 times more the time required to traverse the respective answer trie, while for the *separate value* variant, it takes in average 3,3 times.

The chart in Fig. 6.3 shows that all implementations have rather distinct behaviours in both data transaction types. For the storage and transaction, the *inline* variant of *Datalog* model is the fastest implementation, followed by the *hierarchical* model and in last, the *separate value* variant of *Datalog* model. Conversely, in the retrieval phase, the *hierarchical* model is the fastest implementation, followed by the *inline* variant and in last, the *separate value* variant of the *Datalog* model, whose performance no doubt decays due to the involved LEFT JOIN operations. One important fact emerges from this chart: all of DBTab's measured retrieval times are not only significantly lower than the execution time of benchmark #2, but also their growth rate is very much slower.

Type	Answers	Strategy	Write	Read	Dataset	Browse
Integer	1538	Inline	00:00.067 (67/2,7)	00:00.008 (8/0,3)	14 (0,4)	00:00.001 (1,0)
		Separate	00:00.068 (68/2,7)	00:00.008 (8/0,3)	14 (0,4)	00:00.001 (1,0)
		Hierarchy	00:00.058 (58/2,3)	00:00.005 (5/0,2)	21 (0,6)	N.A.
	8194	Inline	00:00.244 (48,8/1,1)	00:00.044 (8,8/0,2)	72 (0,4)	00:00.005 (2,5)
		Separate	00:00.208 (41,6/1,0)	00:00.043 (8,6/0,2)	72 (0,4)	00:00.004 (2,0)
		Hierarchy	00:00.192 (38,4/0,9)	00:00.035 (17,5/0,2)	111 (0,6)	N.A.
	40962	Inline	00:01.093 (42/0,9)	00:00.237 (9,1/0,2)	360 (0,4)	00:00.009 (3,0)
		Separate	00:00.887 (34,1/0,7)	00:00.226 (8,7/0,2)	360 (0,4)	00:00.009 (2,7)
		Hierarchy	00:00.835 (32,1/0,7)	00:00.143 (47,7/0,1)	546 (0,6)	N.A.
	196610	Inline	00:04.796 (37,2/0,8)	00:01.175 (9,1/0,2)	1728 (0,4)	00:00.057 (4,1)
		Separate	00:04.566 (35,4/0,7)	00:01.171 (9,1/0,2)	1728 (0,4)	00:00.057 (3,9)
		Hierarchy	00:04.272 (33,1/0,7)	00:00.891 (63,6/0,1)	2600 (0,6)	N.A.
Longint	1538	Inline	00:00.069 (69/1,3)	00:00.010 (10/0,2)	23 (0,3)	00:00.001 (1,0)
		Separate	00:00.127 (127/2,4)	00:00.016 (16/0,3)	23 (0,3)	00:00.001 (1,0)
		Hierarchy	00:00.062 (62/1,1)	00:00.006 (6/0,1)	26 (0,4)	N.A.
	8194	Inline	00:00.246 (35,1/0,7)	00:00.053 (7,6/0,1)	136 (0,4)	00:00.005 (2,5)
		Separate	00:00.481 (68,7/1,3)	00:00.086 (12,3/0,2)	136 (0,4)	00:00.004 (2,0)
		Hierarchy	00:00.206 (29,4/0,6)	00:00.036 (18/0,1)	117 (0,3)	N.A.
	40962	Inline	00:01.121 (24,4/0,6)	00:00.272 (5,9/0,1)	680 (0,4)	00:00.028 (7,0)
		Separate	00:02.376 (51,7/1,3)	00:00.444 (9,7/0,2)	680 (0,4)	00:00.026 (2,7)
		Hierarchy	00:00.850 (18,5/0,5)	00:00.164 (41/0,1)	572 (0,3)	N.A.
	196610	Inline	00:05.017 (26,7/0,5)	00:01.626 (8,6/0,2)	3264 (0,4)	00:00.117 (6,5)
		Separate	00:11.400 (60,6/1,1)	00:02.815 (15/0,3)	3264 (0,4)	00:00.112 (6,2)
		Hierarchy	00:04.454 (23,7/0,4)	00:00.990 (55/0,1)	2626 (0,3)	N.A.
Double	1538	Inline	00:00.076 (38/1,1)	00:00.013 (6,5/0,2)	38 (0,4)	00:00.001 (1,0
		Separate	00:00.128 (64/1,8)	00:00.022 (11/0,3)	38 (0,4)	00:00.001 (1,0)
		Hierarchy	00:00.096 (48/1,4)	00:00.015 (15/0,2)	44 (0,4)	N.A.
	8194	Inline	00:00.278 (25,3/0,6)	00:00.073 (6,6/0,2)	200 (0,4)	00:00.007 (2,3)
		Separate	00:00.540 (54/1,2)	00:00.110 (11/0,2)	200 (0,4)	00:00.006 (2,0)
		Hierarchy	00:00.416 (41,6/0,9)	00:00.078 (26/0,2)	228 (0,4)	N.A.
	40962	Inline	00:01.195 (17,8/0,5)	00:00.337 (5/0,2)	1000 (0,4)	00:00.047 (7,8)
		Separate	00:02.601 (38,8/1,2)	00:00.533 (8/0,2)	1000 (0,4)	00:00.044 (7,3)
		Hierarchy	00:01.608 (24/0,7)	00:00.330 (55/0,1)	1118 (0,4)	N.A.
	196610	Inline	00:06.347 (31,6/0,5)	00:03.346 (16,6/0,3)	4800 (0,4)	00:00.121 (4,5)
		Separate	00:12.521 (62,3/1,1)	00:03.512 (17,5/0,3)	4800 (0,4)	00:00.116 (4,3)
		Hierarchy	00:09.237 (46/0,8)	00:02.752 (101,9/0,2)	5304 (0,4)	N.A.

Table 6.2: Transactional overheads for the Binary Tree

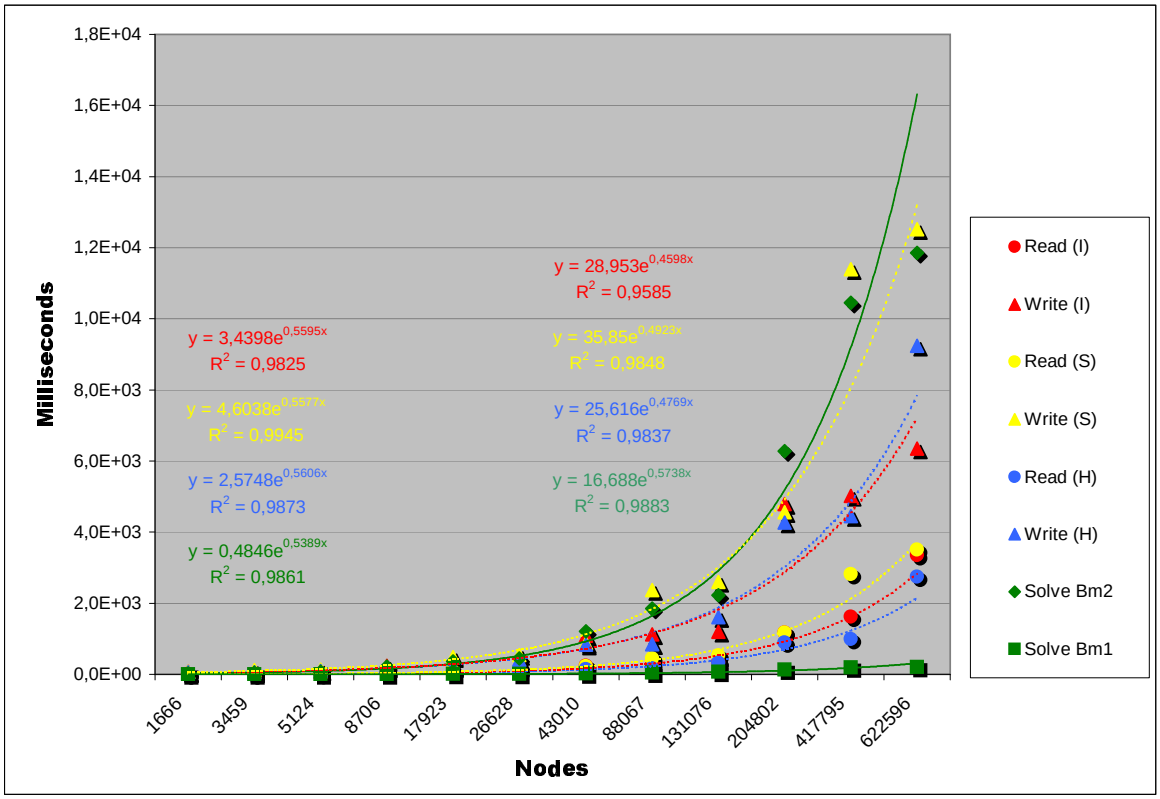

Figure 6.3: Binary Tree - comparing different alternatives performance

6.1.2 Case Study #2: Directed Grid

The second tested topology is the *directed grid graph*. The grid corresponds to the square lattice and it is isomorphic to the graph having a vertex corresponding to every pair of integers (a, b), and an edge connecting (a, b) to $(a + 1, b)$ and $(a, b + 1)$. The finite grid graph $\mathcal{G}_{n,n}$, presented in Fig. 6.4, is obtained by restricting the ordered pairs to the range $0 = a \leq m,\ 0 = b \leq n$.

The execution times for the benchmark programs are summarized in Table 6.3. As in the previous example, the number of nodes and the label types have significant impact on the execution times. In average, answer set generation using long integer and floating-point labels is 1,6 and 2,0 times slower than when integer labels are used. Again, input/output operations introduce a significant increase in performance. In average, benchmark #2 executes 87 times slower than the benchmark #1.

The execution times for the benchmark predicates are presented in Table 6.4. Once again, the most expensive operation is the storage of the answer sets. For the *inline*

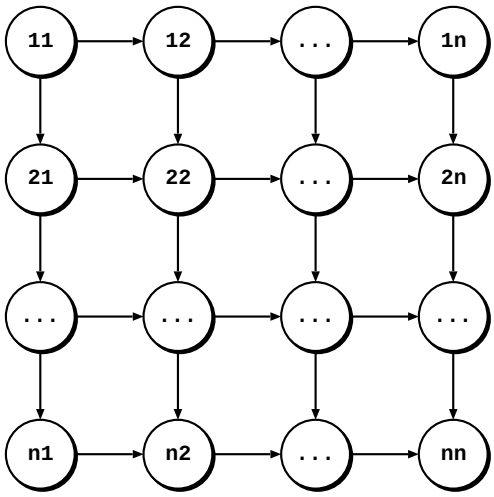

Figure 6.4: Directed Grid

Vertexes	Answers	Type	Nodes	Memory	Bm #1	Bm #2	Traverse
		Integer	6084	119	00:00.003	00:00.260	00:00.003
144	5940	Longint	12311	241	00:00.004	00:00.438	00:00.004
		Double	18394	359	00:00.006	00:00.541	00:00.006
		Integer	18496	361	00:00.009	00:00.817	00:00.009
256	18240	Longint	37247	728	00:00.017	00:01.431	00:00.017
		Double	55742	1089	00:00.019	00:01.697	00:00.019
		Integer	90000	1758	00:00.055	00:04.432	00:00.055
576	89424	Longint	180575	3527	00:00.086	00:07.379	00:00.086
		Double	270574	5285	00:00.104	00:08.569	00:00.104
		Integer	278784	7398	00:00.169	00:13.969	00:00.169
1024	277760	Longint	558591	10910	00:00.275	00:23.379	00:00.275
		Double	834490	16299	00:00.335	00:26.954	00:00.335

Table 6.3: YapTab's times for benchmarks #1 and #2 with the Directed Grid

variant of the *Datalog* model, the median induced overhead is 26,9 times the amount of time required for benchmark #1 to execute, within an interval ranging from 22,5 to 50,3 times. For the *separate value* variant, the median overhead grows to 48,6 times the execution time of benchmark #1, within an interval ranging from 27,5 to 80,5 times. For the *hierarchical* model, the median overhead is of 39,6 times the computation time, within an interval ranging from 21,6 to 64,0 times. The discussed ratios drop drastically when calculated for the execution times of benchmark #2. For the *inline* variant, the median storage overhead is 0,3 times the amount of time required to execute benchmark #2, within an interval ranging from 0,2 to 0,6 times.

For the *separate value* variant, the median overhead grows to 0,6 times, within an interval ranging from 0,3 to 0,7 times. For the *hierarchical* model, the median induced overhead is of 0,5 times the computation time of benchmark #2, within an interval ranging from 0,2 to 0,7 times.

Retrieval operation costs remains relatively low. For the *inline* variant of the *Datalog* model, the median retrieval time takes 9,9 times the amount of time required by benchmark #1 to terminate, within an interval of 7,2 to 11,3 times. For the *separate value* variant, the average overhead grows to 13,5 times, within an interval ranging from 10,0 to 17,5 times. For the *hierarchical* model, the average overhead is of 7,7 times the computation time of benchmark #1, within an interval ranging from 3,9 to 14,5 times. The discussed values drop when benchmark #2 is executed, so drastically that a speedup is observed. For the *inline* variant of the *Datalog* model, the minimum, median and maximum retrieval times are 0,1 times the amount of time required to execute benchmark #2. The performance of the other variant is similar, although in the worst-case scenario the median retrieval time may rise to 0,2. For the *hierarchical* model, the median retrieval time is 0,1 times the computation time of benchmark #2, within an interval ranging from (nearly) 0,0 to 0,2 times.

The performance of *answer set browsing* shows no significant change when compared with the previous example. The retrieved tuple set size equals in average 0,4 of the correspondent answer trie size for the *Datalog* model and up to 0,5 for the *hierarchical* model. Browsing times also increase along with the graph size. For the *inline* variant, it takes in average 4,6 times the time required to traverse the respective answer trie. For the *separate value*, the average is 4,3 times.

The chart in Fig. 6.5 reveals a similar scenario from the previous example in terms of storage. For the storage transaction, the *inline* variant of *Datalog* model is the fastest implementation, followed by the *hierarchical* model and in last, the *separate value* variant of *Datalog* model. Conversely, in the retrieval phase, the *hierarchical* model is the fastest implementation, followed very closely by the *inline* variant of the *Datalog* model and in last, the *separate value* variant of the *Datalog* model, whose performance no doubt decays due to the involved LEFT JOIN operations. Notice that, once again, all of DBTab's measured retrieval times are significantly lower than the execution time of benchmark #2, as well as the correspondent growth lines slopes.

Type	Answers	Strategy	Write		Read		Dataset	Browse
Integer	5940	Inline	00:00.151	(50,3/0,6)	00:00.034	(11,3/0,1)	52 (0,4)	00:00.002 (2,0)
		Separate	00:00.125	(41,7/0,5)	00:00.033	(11/0,1)	52 (0,4)	00:00.001 (1,0)
		Hierarchy	00:00.192	(64/0,7)	00:00.023	(23/0,1)	77 (0,6)	N.A.
	18240	Inline	00:00.415	(46,1/0,5)	00:00.097	(10,8/0,1)	160 (0,4)	00:00.004 (4,0)
		Separate	00:00.350	(38,9/0,4)	00:00.097	(10,8/0,1)	160 (0,4)	00:00.004 (4,0)
		Hierarchy	00:00.564	(62,7/0,7)	00:00.070	(70/0,1)	235 (0,6)	N.A.
	89424	Inline	00:02.040	(37,1/0,5)	00:00.562	(10,2/0,1)	786 (0,4)	00:00.027 (4,5)
		Separate	00:01.521	(27,7/0,3)	00:00.552	(10/0,1)	786 (0,4)	00:00.018 (3,0)
		Hierarchy	00:01.841	(33,5/0,4)	00:00.337	(56,2/0,1)	1143 (0,6)	N.A.
	277760	Inline	00:06.452	(38,2/0,5)	00:01.887	(11,2/0,1)	3223 (0,4)	00:00.061 (3,6)
		Separate	00:06.094	(36,1/0,4)	00:02.207	(13,1/0,2)	3223 (0,4)	00:00.064 (3,8)
		Hierarchy	00:06.363	(37,7/0,5)	00:01.362	(80,1/0,1)	3539 (0,5)	N.A.
Longint	5940	Inline	00:00.153	(38,3/0,3)	00:00.041	(10,3/0,1)	99 (0,4)	00:00.004 (4,0)
		Separate	00:01.654	(413,5/3,8)	00:00.070	(17,5/0,2)	99 (0,4)	00:00.003 (3,0)
		Hierarchy	00:00.240	(60/0,5)	00:00.026	(26/0,1)	79 (0,3)	N.A.
	18240	Inline	00:00.425	(25/0,3)	00:00.122	(7,2/0,1)	303 (0,4)	00:00.011 (5,5)
		Separate	00:04.039	(237,6/2,8)	00:00.195	(11,5/0,1)	303 (0,4)	00:00.011 (5,5)
		Hierarchy	00:00.642	(37,8/0,4)	00:00.074	(37/0,1)	238 (0,3)	N.A.
	89424	Inline	00:02.104	(24,5/0,3)	00:00.823	(9,6/0,1)	1485 (0,4)	00:00.056 (7,0)
		Separate	00:25.090	(291,7/3,4)	00:01.339	(15,6/0,2)	1485 (0,4)	00:00.057 (7,1)
		Hierarchy	00:01.861	(21,6/0,3)	00:00.338	(42,3/≈0)	1150 (0,3)	N.A.
	277760	Inline	00:06.666	(24,2/0,3)	00:02.258	(8,2/0,1)	4611 (0,4)	00:00.162 (5,4)
		Separate	01:13.874	(268,6/3,2)	00:04.282	(15,6/0,2)	4611 (0,4)	00:00.167 (5,6)
		Hierarchy	00:06.438	(23,4/0,3)	00:01.405	(46,8/0,1)	3552 (0,3)	N.A.
Double	5940	Inline	00:00.173	(28,8/0,3)	00:00.057	(9,5/0,1)	145 (0,4)	00:00.004 (4,0)
		Separate	00:01.582	(270,2/3,0)	00:00.083	(13,8/0,2)	145 (0,4)	00:00.003 (3,0)
		Hierarchy	00:00.373	(62,2/0,7)	00:00.057	(57/0,1)	156 (0,4)	N.A.
	18240	Inline	00:00.464	(24,4/0,3)	00:00.164	(8,6/0,1)	445 (0,4)	00:00.011 (5,5)
		Separate	00:04.692	(257,3/2,9)	00:00.245	(12,9/0,1)	445 (0,4)	00:00.011 (5,5)
		Hierarchy	00:00.849	(44,7/0,5)	00:00.162	(81/0,1)	473 (0,4)	N.A.
	89424	Inline	00:02.367	(22,8/0,3)	00:00.871	(8,4/0,1)	2183 (0,4)	00:00.057 (5,2)
		Separate	00:26.501	(251,8/3,1)	00:01.456	(14/0,2)	2183 (0,4)	00:00.059 (5,4)
		Hierarchy	00:04.317	(41,5/0,5)	00:00.843	(76,6/0,1)	2292 (0,4)	N.A.
	277760	Inline	00:07.536	(22,5/0,3)	00:03.778	(11,3/0,1)	6781 (0,4)	00:00.171 (4,5)
		Separate	01:15.270	(225/2,8)	00:05.184	(15,5/0,2)	6781 (0,4)	00:00.167 (4,4)
		Hierarchy	00:12.112	(36,2/0,4)	00:04.867	(128,1/0,2)	7091 (0,4)	N.A.

Table 6.4: Transactional overheads for the Directed Grid

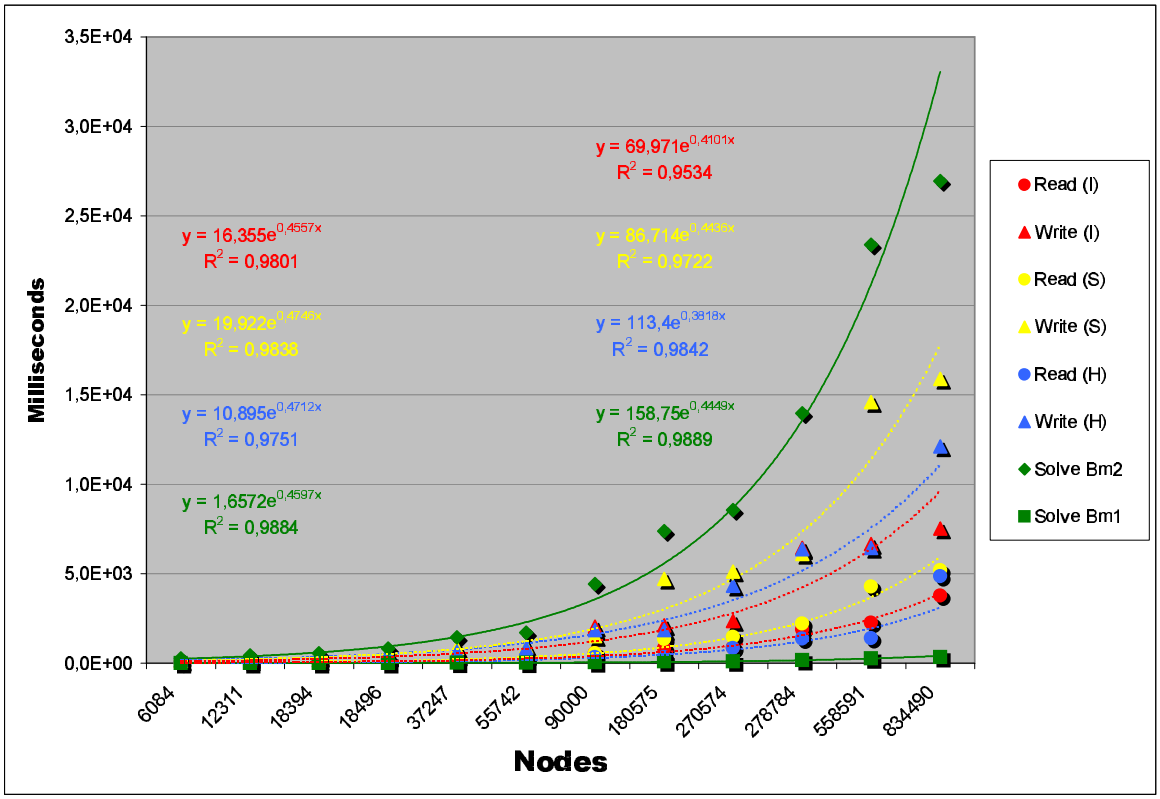

Figure 6.5: Directed Grid - comparing different alternatives performance

6.1.3 Case Study #3: Cyclic Graph

None of the graphs presented so far contained cyclic paths. One could wonder what would happen if such paths were possible and what would be the impact on performance. In order to answer this question, both graphs were transformed to include cyclic paths.

The third graph topology is the *cyclic graph*. The new graph consists of an adulterated binary tree, introducing cyclic paths in the structure basically by making the leaves disappear. In this new structure, leaves l_k and l_{k+1} sharing the same father p are connected by two new edges, (l_k,l_{k+1}) and (l_{k+1},l_k). Additionally, l_k is connected to node n_i, where $i = log_2(n)/2$, and l_{k+1} is connected to node n_j, with $j = log_2(n)/4$. Figure 6.6 illustrates the new graph.

Table 6.5 summarizes YapTab's execution times for the benchmark programs. As expected, the number of nodes and the label types influences the execution times. In average, answer set generation using long integer and floating-point labels is 1,3 and

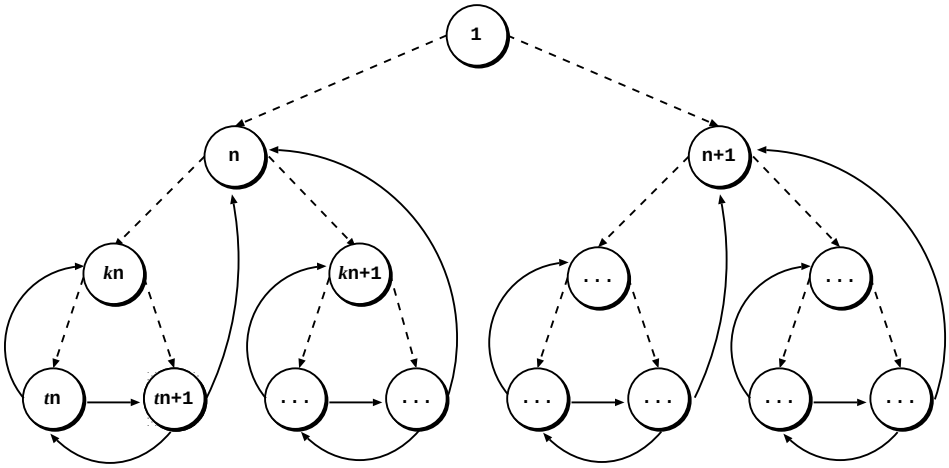

Figure 6.6: Cyclic Graph

1,7 times slower than when integer labels are used. The introduction of input/output operations has a great impact on performance, slowing benchmark #2 execution in average 105,6 times the time spent by benchmark #1.

The increase of possible paths between two nodes has proved to have some impact on the performance of benchmark predicates. The comparison of the acyclic and cyclic cases for the same number of nodes reveals that the introduction of edges connecting the leafs in pairs and each leaf to an ancestor increases memory requirement growth in average by a factor of 3. Execution time increases in average factors of 2,7 and 5,8 for benchmarks #1 and #2 respectively.

Vertexes	Answers	Type	Nodes	Memory	Bm #1	Bm #2	Traverse
		Integer	2298	45	00:00.001	00:00.089	00:00.001
256	2170	Longint	4723	92	00:00.002	00:00.171	00:00.002
		Double	7020	137	00:00.003	00:00.229	00:00.003
		Integer	17906	350	00:00.010	00:00.807	00:00.010
1024	17394	Longint	36323	710	00:00.014	00:01.447	00:00.014
		Double	54228	1059	00:00.018	00:01.880	00:00.018
		Integer	139234	2720	00:00.078	00:07.404	00:00.078
4096	137186	Longint	280515	5479	00:00.104	00:12.445	00:00.104
		Double	419748	8198	00:00.150	00:15.569	00:00.150
		Integer	1089474	21279	00:00.596	00:58.655	00:00.596
16384	1081282	Longint	2187139	42718	00:00.897	01:30.044	00:00.897
		Double	3276612	63996	00:01.284	01:52.490	00:00.284

Table 6.5: YapTab's times for benchmarks #1 and #2 with the Cyclic Graph

The execution times presented in Table 6.6 reveal a similar pattern to the one found in the acyclic example, as most of DBTab's execution time is spent during the storage phase. For the *inline* variant of the *Datalog* model, the median induced overhead is 36,4 times the amount of time required to execute benchmark #1, within an interval ranging from 23,9 to 85,0 times. For the *separate value* variant, the median overhead grows to 58,1 times the execution time of benchmark #1, within an interval ranging from 31,4 to 144 times. For the *hierarchical* model, the median overhead is 42,8 times the computation time, within an interval ranging from 28,2 to 75,0 times. These ratios drop again drastically when calculated for the execution times of benchmark #2. For the *inline* variant, the median induce overhead is 0,3 times the amount of time required to execute benchmark #2, within an interval ranging from 0,2 to 0,9 times. For the *separate value* variant, the median overhead grows to 0,6 times, within an interval ranging from 0,4 to 0,8 times. For the *hierarchical* model, the median overhead is 0,4 times the computation time of benchmark #2, within an interval ranging from 0,3 to 0,8 times.

As expected, retrieval cost remains relatively low when compared with computation time. For the *inline* variant of the *Datalog* model, the median induced overhead is 9,2 times the amount of time required by benchmark #2 to terminate, within an interval of 8,5 to 13 times. For the *separate value* variant, the median overhead grows to 16,2 times, within an interval ranging from 9,1 to 34,1 times. For the *hierarchical* model, the median overhead is 7,4 times the computation time of benchmark #2, within an interval ranging from 4,9 to 17,4 times. When benchmark #2 is executed, the ratios drop as previously observed, and overheads become speedups. For the *inline* variant of the *Datalog* model, the minimum, median and maximum retrieval time are 0,1 times the amount of time required to execute benchmark #2. The performance of the *separate value* variant is very similar, except for the maximum retrieval time which is 0,3 times. For the *hierarchical* model, minimum and median times are very close to 0,1 times the amount of time required to execute benchmark #2, and the maximum retrieval time which is 0,2 times.

The performance of *answer set browsing* also displays a pattern similar to that of the previous examples. Despite the absolute increase of the tuple set sizes, the retrieved tuple set size equals in average 0,4 of the correspondent answer trie size for the *Datalog* model and up to 0,5 for the *hierarchical* model. Again, it is possible to observe that

answer set browsing times grow with the graph size. For the *inline* variant, it takes in average 3,8 times more the time required to traverse the respective answer trie. For the *separate value*, it takes only 3,6 times.

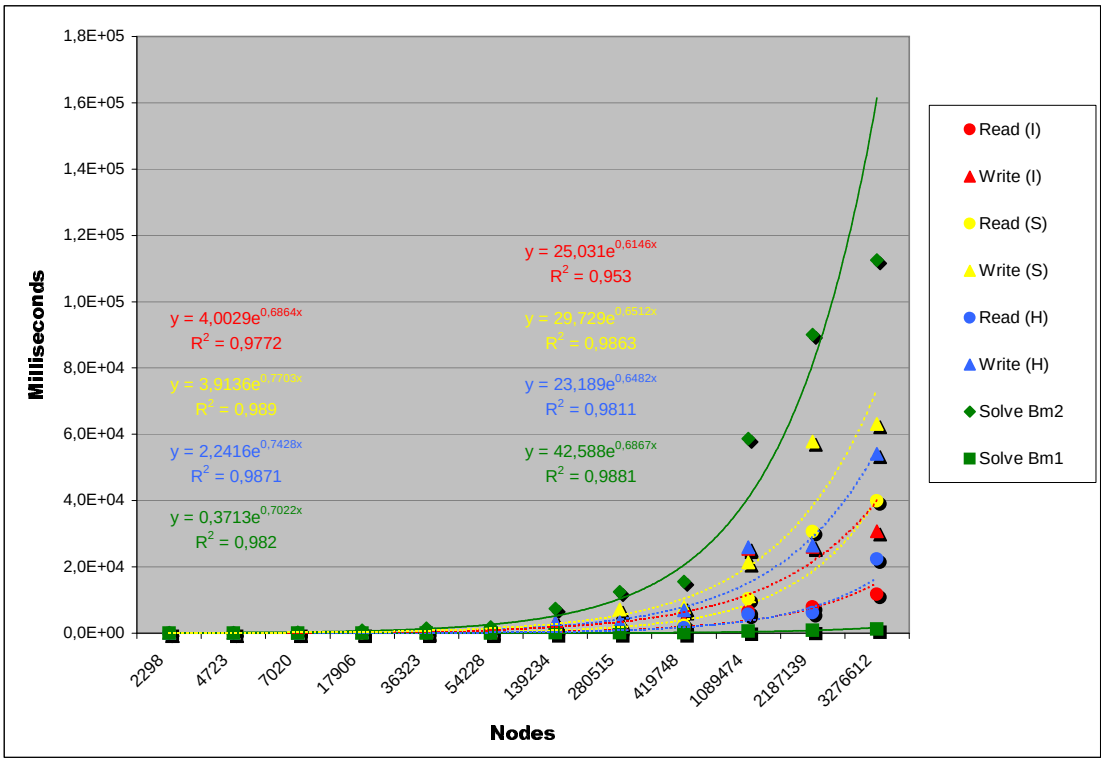

Figure 6.7: Cyclic Graph - comparing different alternatives performance

Figure 6.7 displays the performance chart for the three implementations, in a similar pattern to the one found in previous cases. For the retrieval transaction, the *inline* variant of the *Datalog* model is the fastest implementation, followed by the *hierarchical* model and at last by the *separate value* variant. Retrieval operations maintain a certain regularity of performance. Again, the fastest implementation is the *hierarchical* model, followed by the *inline* variant and, at last by the *separate value* variant. However, one can observe that **(i)** the difference between the two first implementations is very small and **(ii)** the third implementation's central tendency line indicates that, for larger graphs, performance decreases very fast - notice the line's slope increase near the end. Notice that once again, all of DBTab's measured times are still significantly lower than the execution time of benchmark #2, as well as the growth line slopes.

Type	Answers	Strategy	Write	Read	Dataset	Browse
Integer	2170	Inline	00:00.077 (77/0,9)	00:00.011 (11/0,1)	19 (0,4)	00:00.001 (1,0)
		Separate	00:00.072 (72/2,3)	00:00.010 (10/0,1)	19 (0,4)	00:00.001 (1,0)
		Hierarchy	00:00.073 (73/0,8)	00:00.006 (6/0,1)	29 (0,6)	N.A.
	17394	Inline	00:00.427 (42,7/0,5)	00:00.093 (9,3/0,1)	153 (0,4)	00:00.005 (2,5)
		Separate	00:00.341 (34,1/2,1)	00:00.091 (9,1/0,1)	153 (0,4)	00:00.004 (2,0)
		Hierarchy	00:00.369 (36,9/0,5)	00:00.066 (33/0,1)	227 (0,6)	N.A.
	137186	Inline	00:03.231 (41,4/0,4)	00:00.861 (11/0,1)	1206 (0,4)	00:00.034 (3,8)
		Separate	00:02.810 (36/1,7)	00:00.800 (10,3/0,1)	1206 (0,4)	00:00.031 (3,4)
		Hierarchy	00:03.195 (41/0,4)	00:00.494 (54,9/0,1)	1768 (0,6)	N.A.
	1081282	Inline	00:25.398 (42,6/0,4)	00:06.549 (11/0,1)	9503 (0,4)	00:00.267 (3,0)
		Separate	00:21.332 (35,8/1,9)	00:10.381 (17,4/0,2)	9503 (0,4)	00:00.229 (2,6)
		Hierarchy	00:25.923 (43,5/0,4)	00:05.733 (65,1/0,1)	13831 (0,6)	N.A.
Longint	2170	Inline	00:00.085 (85/0,5)	00:00.013 (13/0,1)	36 (0,4)	00:00.001 (1,0)
		Separate	00:00.144 (144/0,8)	00:00.024 (24/0,1)	36 (0,4)	00:00.001 (1,0)
		Hierarchy	00:00.075 (75/0,4)	00:00.008 (8/0)	31 (0,3)	N.A.
	17394	Inline	00:00.436 (31,1/0,3)	00:00.119 (8,5/0,1)	289 (0,4)	00:00.011 (5,5)
		Separate	00:00.887 (63,4/0,6)	00:00.194 (13,9/0,1)	289 (0,4)	00:00.010 (5,0)
		Hierarchy	00:00.395 (28,2/0,3)	00:00.070 (35/0)	234 (0,3)	N.A.
	137186	Inline	00:03.258 (31,3/0,3)	00:00.922 (8,9/0,1)	2278 (0,4)	00:00.085 (7,1)
		Separate	00:07.133 (68,6/0,6)	00:01.968 (18,9/0,2)	2278 (0,4)	00:00.084 (7,0)
		Hierarchy	00:03.355 (32,3/0,3)	00:00.509 (42,4/0)	1794 (0,3)	N.A.
	1081282	Inline	00:26.016 (29/0,3)	00:07.883 (8,8/0,1)	17951 (0,4)	00:00.638 (5,7)
		Separate	00:57.856 (64,5/0,6)	00:30.662 (34,2/0,3)	17951 (0,4)	00:00.626 (5,6)
		Hierarchy	00:26.451 (29,5/0,3)	00:06.174 (55,1/0,1)	13935 (0,3)	N.A.
Double	2170	Inline	00:00.088 (44/0,4)	00:00.019 (9,5/0,1)	53 (0,4)	00:00.002 (2,0)
		Separate	00:00.151 (75,5/0,7)	00:00.033 (16,5/0,1)	53 (0,4)	00:00.001 (1,0)
		Hierarchy	00:00.123 (61,5/0,5)	00:00.021 (21/0,1)	60 (0,4)	N.A.
	17394	Inline	00:00.473 (26,3/0,3)	00:00.159 (8,8/0,1)	425 (0,4)	00:00.011 (5,5)
		Separate	00:00.931 (51,7/0,5)	00:00.241 (13,4/0,1)	425 (0,4)	00:00.010 (5,0)
		Hierarchy	00:00.812 (45,1/0,4)	00:00.162 (81/0,1)	461 (0,4)	N.A.
	137186	Inline	00:03.684 (24,6/0,2)	00:01.319 (8,8/0,1)	3349 (0,4)	00:00.088 (4,6)
		Separate	00:07.926 (52,8/0,5)	00:02.389 (15,9/0,2)	3349 (0,4)	00:00.087 (4,6)
		Hierarchy	00:06.946 (46,3/0,4)	00:01.550 (81,6/0,1)	3561 (0,4)	N.A.
	1081282	Inline	00:30.707 (23,9/0,3)	00:11.685 (9,1/0,1)	26398 (0,4)	00:00.721 (4,4)
		Separate	01:03.031 (49,1/0,6)	00:39.860 (31/0,4)	26398 (0,4)	00:00.654 (4,0)
		Hierarchy	00:54.104 (42,1/0,5)	00:22.399 (136,6/0,2)	27766 (0,4)	N.A.

Table 6.6: Transactional overheads for the Cyclic Graph

6.1.4 Case Study #4: Bidirectional Grid

The last and final topology is the bidirectional grid graph. All edges in the graph become bidirectional, as illustrated in Fig. 6.8. More accurately, the graph generator is modified so that each edge connecting two distinct vertexes is complemented with a new edge establishing a connection in the opposite direction. This enables the appearance of cyclic paths within the graph, thus largely increasing the costs of the path finding operation.

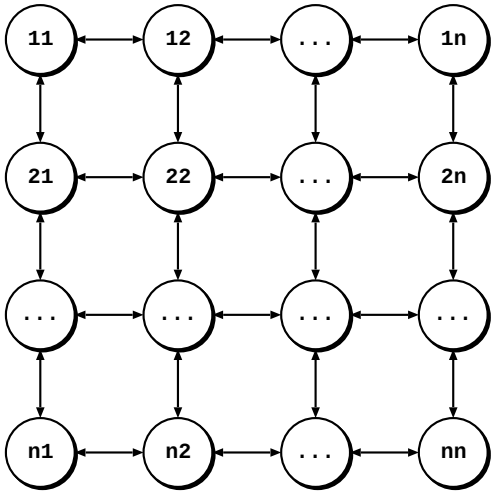

Figure 6.8: Bidirectional Grid

Table 6.7 summarizes YapTab's execution times for the benchmark programs. Needless to say, the number of nodes and the label types influences the execution times. In average, answer set generation using long integer and floating-point labels is 1,6 and 2,3 times slower than when integer labels are used. The introduction of input/output operations again has a great impact on performance, slowing benchmark #2 execution in average 112 times relatively to that spent by benchmark #1.

This is the case study in which the increase of possible paths between two nodes has the most significant impact on the performance of benchmark predicates. In average, for the same number of nodes, the existence of an edge in the opposite direction increased memory space requirements by a factor of 3,6. Execution time increases in average factors of 6,1 and 7,8 for benchmark #1 and #2 respectively.

As with the acyclic example, the execution times presented in Table 6.8 reveal a familiar pattern. Let us again start to observe the storage operation. For the *inline*

Vertexes	Answers	Type	Nodes	Memory	Bm #1	Bm #2	Traverse
		Integer	20881	408	00:00.015	00:01.850	00:00.015
144	20736	Longint	41906	819	00:00.028	00:03.345	00:00.028
		Double	62786	1226	00:00.042	00:04.150	00:00.042
		Integer	65793	1285	00:00.057	00:06.521	00:00.057
256	65536	Longint	131842	2575	00:00.088	00:10.414	00:00.088
		Double	197634	3860	00:00.116	00:13.467	00:00.116
		Integer	332353	6491	00:00.338	00:34.514	00:00.338
576	331776	Longint	665282	12994	00:00.480	00:57.107	00:00.480
		Double	997634	19485	00:00.658	01:09.790	00:00.658
		Integer	1049601	28313	00:00.954	01:52.920	00:00.954
1024	1048576	Longint	2100226	41020	00:01.600	02:52.936	00:00.600
		Double	3149826	61520	00:02.235	03:41.624	00:00.235

Table 6.7: YapTab's times for benchmarks #1 and #2 with the Bidirectional Grid

variant of the *Datalog* model, the median induced overhead is 16,5 times the amount of time required to execute benchmark #1, within an interval ranging from 13,0 to 30,0 times. For the *separate value* variant, the median overhead grows to 28,4 times the execution time of benchmark #1, within an interval ranging from 18,7 to 37,6 times. For the *hierarchical* model, the median overhead is 21,1 times the computation time, within an interval ranging from 14,1 to 29,1 times. As usual, the mentioned ratios drop immensely when calculated for the execution times of benchmark #2. For the *inline* variant, the median induced overhead is 0,1 times the amount of time required to execute benchmark #2, within an interval ranging from 0,1 to 0,2 times. For the *separate value* variant, the median overhead grows to 0,3 times, within an interval ranging from 0,2 to 0,4 times. For the *hierarchical* model, the median overhead is 0,2 times the computation time of benchmark 2, within an interval ranging from 0,1 to 0,2 times.

Retrieval cost are also low when compared with computation time. For the *inline* variant of the *Datalog* model, the average induced overhead is 11 times the amount of time required by benchmark #2 to terminate, within an interval of 5 to 27 times. For the *separate value* variant, the average overhead grows to 13 times, within an interval ranging from 7 to 32 times. The *hierarchical* model is faster, taking round half the time of any of the *Datalog* variant. The average overhead is 5 times the computation time of benchmark #2, within an interval ranging from 3 to 9 times. When benchmark #2 is executed, the ratios drop as previously observed, with overheads becoming speedups.

For both variants of the *Datalog* model, the median retrieval time is 0,1 times the amount of time required to execute benchmark #2, within an interval ranging from (nearly) 0,0 to 0,3, although the *inline* variant is faster than the *separate value*. For the *hierarchical* model, the minimum, median and maximum times may be never exceed 0,1 times the amount of time required to execute benchmark #2.

In both models, the retrieved tuple set size equals in average 0,4 of the correspondent answer trie size. *Answer set browsing* times also grow along with the graph size. For the *inline* variant, it takes in average 4,8 times more the time required to traverse the respective answer trie. For the *separate value*, it takes in average 4,7.

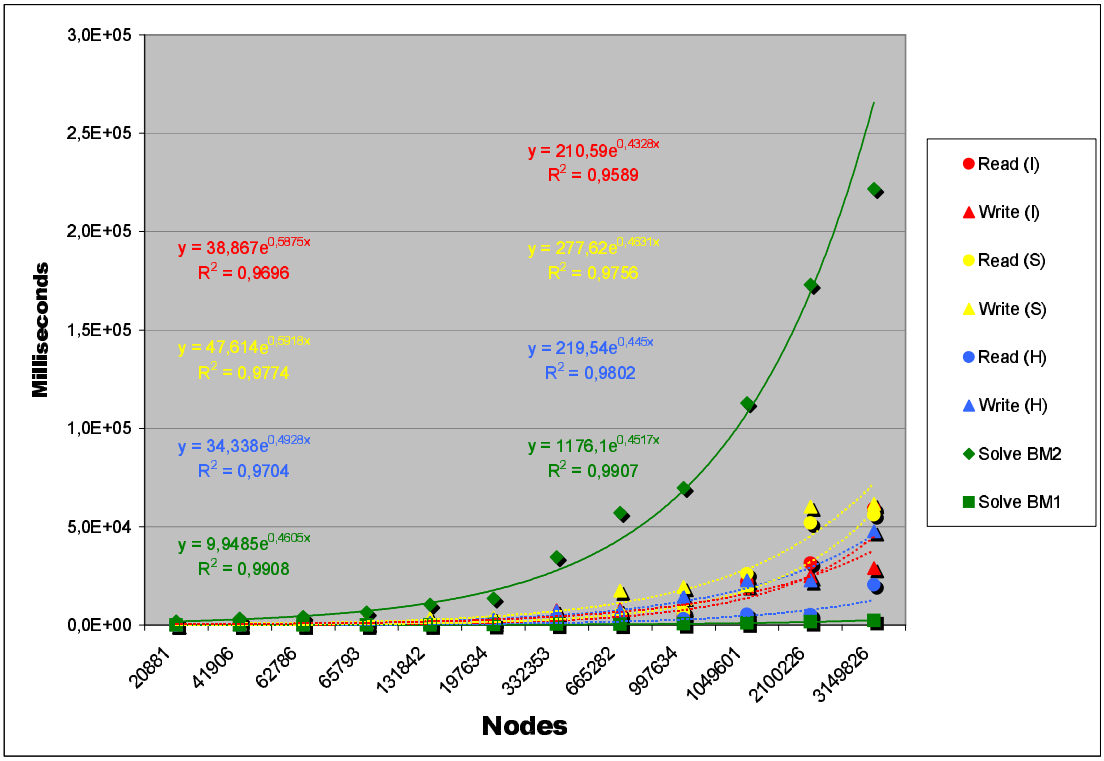

Figure 6.9: Bidirectional Grid - comparing different alternatives performance

The performance chart in Fig. 6.5 reveals a partially new scenario. Up to a certain point, it confirms the performance pattern found in the all previous examples. Regarding the storage operation, the fastest implementation is once again the *inline* variant of the *Datalog* model, followed by the *hierarchical* model and the *separate value* variant. Retrieval operations also maintained their regularity in performance. The fastest implementation is still the *hierarchical* model, followed by the *inline* variant and, at last, by the *separate value* variant. However, for the last quarter onwards,

Type	Answers	Strategy	Write	Read	Dataset	Browse
Integer	20736	Inline	00:00.450 (30/0,2)	00:00.116 (7,7/0,1)	182 (0,4)	00:00.005 (5,0)
		Separate	00:00.389 (25,9/0,2)	00:00.110 (7,3/0,1)	182 (0,4)	00:00.005 (5,0)
		Hierarchy	00:00.437 (29,1/0,2)	00:00.076 (76/≈0)	265 (0,6)	N.A.
	65536	Inline	00:01.363 (23,9/0,2)	00:00.445 (7,8/0,1)	576 (0,4)	00:00.014 (3,8)
		Separate	00:01.255 (22/0,2)	00:00.385 (6,8/0,1)	576 (0,4)	00:00.015 (3,5)
		Hierarchy	00:01.211 (21,2/0,2)	00:00.253 (63,3/≈0)	835 (0,6)	N.A.
	331776	Inline	00:07.532 (22,3/0,2)	00:02.788 (8,2/0,1)	2916 (0,4)	00:00.080 (3,3)
		Separate	00:06.339 (18,8/0,2)	00:02.196 (6,5/0,1)	2916 (0,4)	00:00.075 (3,1)
		Hierarchy	00:07.047 (20,8/0,2)	00:01.486 (61,9/≈0)	4219 (0,6)	N.A.
	1048576	Inline	00:23.402 (24,5/0,2)	00:21.907 (23/0,2)	10779 (0,4)	00:00.287 (4,6)
		Separate	00:20.184 (21,2/0,2)	00:26.022 (27,3/0,2)	10779 (0,4)	00:00.326 (4,0)
		Hierarchy	00:22.744 (23,8/0,2)	00:05.381 (75,8/≈0)	13325 (0,5)	N.A.
Longint	20736	Inline	00:00.461 (16,5/0,1)	00:00.143 (5,1/≈0)	344 (0,4)	00:00.013 (4,3)
		Separate	00:01.009 (36/0,3)	00:00.233 (8,3/0,1)	344 (0,4)	00:00.012 (4,0)
		Hierarchy	00:00.494 (17,6/0,1)	00:00.085 (28,3/≈0)	267 (0,3)	N.A.
	65536	Inline	00:01.542 (17,5/0,1)	00:00.574 (6,5/0,1)	1088 (0,4)	00:00.042 (7,0)
		Separate	00:03.266 (37,1/0,3)	00:00.832 (9,5/0,1)	1088 (0,4)	00:00.041 (6,8)
		Hierarchy	00:01.275 (14,5/0,1)	00:00.238 (39,7/≈0)	839 (0,3)	N.A.
	331776	Inline	00:07.961 (16,6/0,1)	00:03.446 (7,2/0,1)	5508 (0,4)	00:00.198 (6,8)
		Separate	00:17.571 (36,6/0,3)	00:05.589 (11,6/0,1)	5508 (0,4)	00:00.195 (5,4)
		Hierarchy	00:07.109 (14,8/0,1)	00:01.263 (35,1/≈0)	4227 (0,3)	N.A.
	1048576	Inline	00:24.687 (15,4/0,1)	00:31.411 (19,6/0,2)	17408 (0,4)	00:00.640 (6,4)
		Separate	01:00.147 (37,6/0,3)	00:51.788 (32,4/0,3)	17408 (0,4)	00:00.628 (6,3)
		Hierarchy	00:22.623 (14,1/0,1)	00:05.230 (52,3/≈0)	13338 (0,3)	N.A.
Double	20736	Inline	00:00.546 (13/0,1)	00:00.194 (4,6/≈0)	506 (0,4)	00:00.013 (4,3)
		Separate	00:01.450 (25,1/0,3)	00:00.290 (6,9/0,1)	506 (0,4)	00:00.013 (4,3)
		Hierarchy	00:00.878 (20,9/0,2)	00:00.181 (60,3/≈0)	532 (0,4)	N.A.
	65536	Inline	00:01.641 (14,1/0,1)	00:00.719 (6,2/0,1)	1600 (0,4)	00:00.045 (5,0)
		Separate	00:03.363 (32/0,3)	00:01.054 (9,1/0,1)	1600 (0,4)	00:00.043 (4,8)
		Hierarchy	00:02.858 (24,6/0,2)	00:00.556 (61,8/≈0)	1674 (0,4)	N.A.
	331776	Inline	00:08.854 (13,5/0,1)	00:04.089 (6,2/0,1)	8100 (0,4)	00:00.206 (4,2)
		Separate	00:19.532 (29,5/0,3)	00:06.538 (9,9/0,1)	8100 (0,4)	00:00.201 (4,3)
		Hierarchy	00:14.435 (21,9/0,2)	00:03.150 (65,6/≈0)	8446 (0,4)	N.A.
	1048576	Inline	00:29.035 (13/0,1)	00:59.163 (26,5/0,3)	25600 (0,4)	00:00.890 (6,5)
		Separate	01:01.402 (27,6/0,3)	00:55.914 (25/0,3)	25600 (0,4)	00:00.680 (5,0)
		Hierarchy	00:47.659 (21,3/0,2)	00:20.420 (149,1/0,1)	26663 (0,4)	N.A.

Table 6.8: Transactional overheads for the Bidirectional Graph

the lines show a tendency not yet observed. Basically, they show that the retrieval operation times for both variants of the *Datalog* model tend to outrun their storage times - in fact, this is actually a fact for the *inline* variant. All retrieval times are now significantly lower than the execution time of benchmark #2 and this discrepancy is evident from the start.

6.2 Discussion

In all performed tests, the data transaction performances revealed a similar pattern. Storage revealed to be an expensive operation. In all cases, this operation's cost largely exceeded the cost of recomputing the same answer set; the exception to this rule was observed in the last example, when the answer set got too big. Concerning this operation, the *inline* variant of the *Datalog* model was always the fastest, the *hierarchical* model the second fastest and the *separate value* was always the slowest. However, when the computation involved side-effected operations, this scenario has radically changed and the cost of storing the answer set became quite acceptable.

Things were somewhat different with the retrieval operation. Of all implementations, the *hierarchical* model was always the fastest and the *separate value* variant the slowest. This last implementation significant performance decay was no doubt induced by the use of LEFT JOIN clauses in the retrieval SELECT statement (as seen in Fig. 4.8). In average, this variant of the *Datalog* model took at least twice the time to retrieve the tuple sets than its *inline* counterpart. When the answer set evaluation involved no costy operations, reloading answers from the database was obviously slower. On the other hand, when the side-effected operations were introduced, reloading became a quite attractive possibility.

In what regards to term types, integer terms where obviously the simplest and fastest to handle. The other two primitive types (long integers and floating-point numbers), requiring special handling, induced significant overheads to both storage and retrieval operations. Atom terms, not considered in the tests, are expected to behave as standard integers if one sticks to their YAP's internal representation. If a external symbols table policy is implemented, the expected storage behaviour should resemble that of the *longints* or *doubles*, since the established mechanism is similar. At the limit, storage performance would be slower than that of *doubles* when the atoms' string sizes

exceed 64 bits.

Clustering answers for insertion revealed to be a wise policy, as the practical example presented in Table 6.9 illustrates. The table exhibits two columns comparing the storage execution times, without and with clustering, for the largest answer set generated for case study #4. The obtained speedup is displayed within parentheses.

Answers	Type	Strategy	Non-clustered	Clustered
1048576	Integer	Inline	01:43.271	00:23.402 (4,4)
		Separate	02:22.568	00:20.184 (7,1)
		Hierarchy	01:32.711	00:22.774 (4,1)
	Longint	Inline	01:49.654	00:24.687 (4,4)
		Separate	06:42.155	01:00.147 (6,7)
		Hierarchy	01:33.262	00:22.623 (4,1)
	Double	Inline	01:59.155	00:29.035 (4,1)
		Separate	07:05.271	01:01.402 (6,9)
		Hierarchy	03:14.141	00:47.659 (4,1)

Table 6.9: Clustered and non-clustered storage for the Bidirectional Graph

As previously mentioned, a 25 tuple cluster was used to obtain such execution times. It should be noticed that a bigger cluster size would produce better times. For instance, experimental results shown that the storage time of the same 1048576 answers, with floating-point numbers labels, using the *separate value approach* and a cluster size of 250 was 00:51.227, i.e., increasing the buffer to 250 would result in a speedup of 0,1(66) in the largest measured execution time. However, the choice of a cluster size was based on the size of the character buffer required to create the necessary SQL statement. In short, for a predicate of arity a, a cluster of k tuples would require a character buffer s whose size $|s|$ could be determined by

$$|s| = k\,(\underbrace{2}_{parentheses} + \underbrace{a-1}_{commas} + \underbrace{a}_{question\ marks}) + \underbrace{k-1}_{commas}$$

For a predicate of arity 5, a cluster size of 250 would produce a query with at least 3499 characters, i.e, approximately 3,5 Kbytes, which is dangerously close to the 4kbytes, the size of each DBTab's buffer cell. Obviously, one could increase this cell size to 8Kytes for instance, but since this buffer is composed of four equally sized cells, that would result in a total memory requirement 32Kbytes of memory space sitting inside

YAP' memory space with no other use than the construction of SQL statements. Arguably, this is too much of a waste, specially if one considers that MySQL's net buffer size is 16Kbytes long by default[1].

For small tuple sets, tuple set browsing might be enough to locate a small initial subset of answers and decide wether that subset is the adequate one or if it should be ignored; in either case, this allows saving both the time and memory resources required for the complete trie reconstruction. However, as tuple sets size increase, this traversal approach performance decays, reaching up to three times the amount of time required to reconstruct the table. In this last case, the only advantage of tuple set browsing is the introduced saving in terms of memory requirements, as discussed back in section 5.2.6. Figures 6.10, 6.11, 6.12 and 6.13 compare the size of the memory blocks required to hold the same answers sets in both formats. Alternatively, a partial retrieval policy could be implemented taking advantage of a special MySQL feature that enables the tuple set partitioning through a special LIMIT sub-clause in the SELECT clause. This partial retrieval could save a lot of time if the answer set was found to be inappropriate just by checking its first answers. In the worst-case scenario, the entire data set would be retrieved slower due to the overheads induced connection, parsing, execution, transmission and closing phases of the query handling.

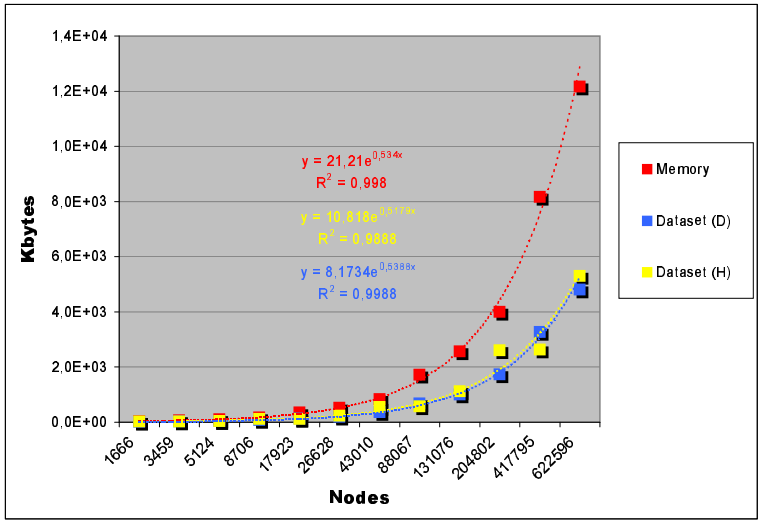

Figure 6.10: Binary Tree - table space and tuples sets memory requirements

The obtained results corroborate those of Florescu [FK99]. In that paper, the authors

[1]For further details, please refer to MySQL Reference Manual [WA02]

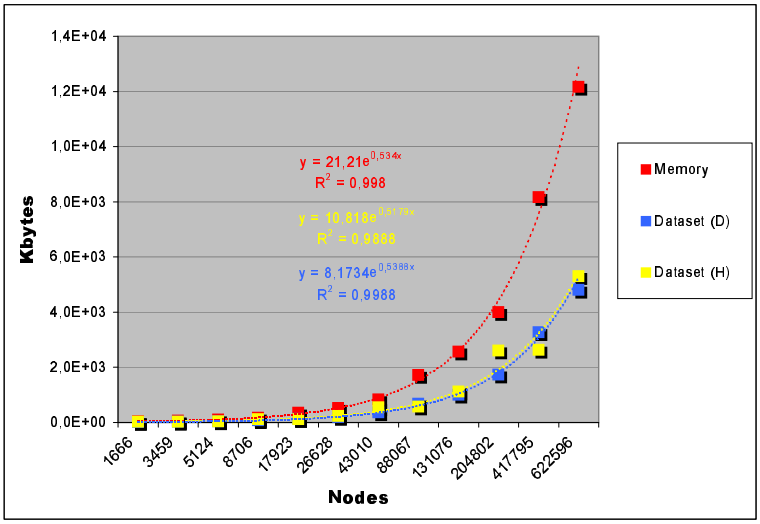

Figure 6.11: Directed Grid - table space and tuples sets memory requirements

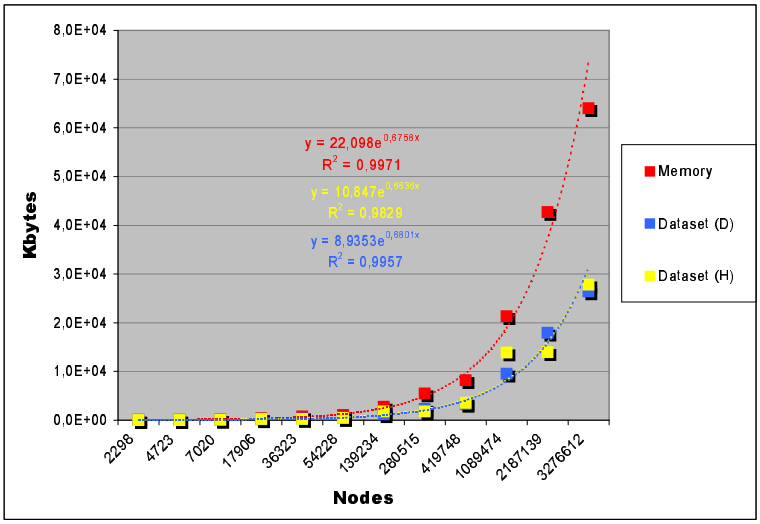

Figure 6.12: Cyclic graph - table space and tuples sets memory requirements

claim that the *hierarchical* model is always better than the *Datalog* model[2], except when *inlining* is applied. The obtained execution times are inline to those provided by Michel *et al.* in [FRS04]. In that paper, the authors claim the storage of 50000 tuples in relation `path/2` in 2 seconds, approximately the time required to store the same amount of tuples using the *separate value* variant of the *Datalog* model using integer labels.

Our final consideration is that our main hypothesis was valid in all performed tests.

[2]In Florescus'paper, these implementations are respectively denominated *Edge* and *Universal*

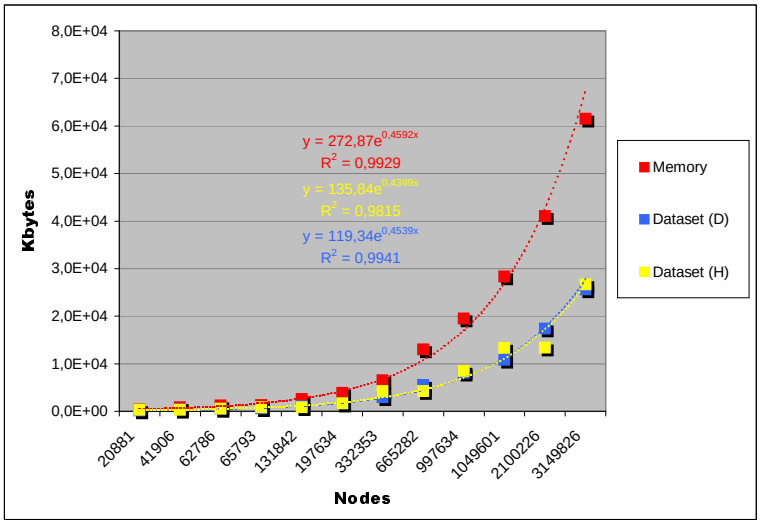

Figure 6.13: Bidirectional Grid - table space and tuples sets memory requirements

As always in science, it is possible that other testings may undo this result. It is our strong conviction that further testing should be performed to provide a deeper insight of the problem and to strengthen the obtained results.

6.3 Chapter Summary

In this chapter we have presented a detailed analysis of DBTab's performance. We have started by presenting an overall view of DBTab's performance for execution of tabled programs, measuring the sequential tabling behavior of DBTab and comparing it with Yaptab. At the end we have discussed the obtained results and we have drawn some conclusions from them. The initial results shown that DBTab may became an interesting approach when the cost of recalculating a table trie largely exceeded the amount of time required to fetch the entire answer tuple set from the mapping relational tuple residing in the database. These situations have risen when heavy side-effected routines were used during program execution.

Chapter 7

Concluding Remarks

In this final chapter, we begin by summarizing the main contributions of this thesis to logic programming and we suggest several directions for further travel. At the end, a final remark completes the chapter and the thesis.

7.1 Conclusion

In this thesis, we have introduced the design, implementation and evaluation of the DBTab system, a relational storage mechanisms for tabled logic programs. To the best of our knowledge, this is the first approach to the mapping of tabling tries to database relations in the context of Prolog engines.

DBTab was designed to be used as an alternative solution to the problem of recovering space when the tabling engine runs out of memory. The common approach, implemented in most tabling systems, is to provide a set of tabling primitives that programmers may use to dynamically delete some of the tables. By storing tables externally, rather than deleting them, DBTab provides YapTab with a powerful tool to avoid standard tabled re-computation when subsequent calls to dropped tables occur.

A major concern during development was to make the best use of the excellent technology already deployed in YAP's sequential Prolog engine [San99, SDRA], as its basic framework, and in YapTab [Roc01] sequential tabling engine, as the basis

for its tabling components. DBTab has been implemented from scratch and it was developed to be fully integrated with YapTab's tabling component. Our proposed extension provides efficient engine support for large data transactions between the logical engine and the database management system.

Another important aspect of DBTab is the memory gain introduced by the tuple set browsing technique. By keeping the tuple sets resulting from database consultations as blocks of binary data, DBTab introduces significative savings regarding memory space requirements when the retrieved answers convey respectively large integer and floating-point terms.

In terms of relational models, three different database schemes have been adopted to represent YapTab's table space externally in a relational environment. Relevant implementation aspects, such as the problem of how to generally represent atomic terms of different natures and sizes, have been disclosed.

A detailed study took place to assess the performance of DBTab. During evaluation, the system was examined against a selected set of benchmark programs that we believe are representative of possible applications. Our preliminaries results show that DBTab may become an interesting approach when the cost of recalculating tabling tries largely exceeds the amount of time required to fetch the entire answer tuple-set from the database. The results reinforced our belief that tabling can contribute to expand the range of applications for Logic Programming.

7.2 Further Work

As further work we plan to investigate the impact of applying DBTab to a more representative set of programs. We also plan to introduce some other enhancements to improve the quality of the developed model.

The expansion of the actual DBTab *Datalog* based models to cover all term representation possibilities presented by YapTab is the first goal to achieve in a near future. Again, two distinct approaches may be followed. A first solution is to store pairs, lists and general non-tabled application terms as *binary large objects* (BLOB) attributes of auxiliary tables, in a similar way to that applied for large atomic terms. This, of course, requires the implementation of specialized API procedures that enable the *term-to-*

BLOB and *BLOB-to-term* translation. Alternatively, the hierarchical approach may be adopted, forming a *mixed* solution. Complex term may then be represented as record trees stored within auxiliary tables. As expected, the root record TERM field shall hold an unique sequential identifier, used as a "foreign" key in the predicate's relational table.

During execution, YapTab processes may have to stop due to several reasons: hosts may crash or have to be turned off, the users may want to interrupt process evaluation, etc. If such a situation arises, table space residing in memory is lost, leading to re-calculation of both the subgoal tries and its associated completed answer tries in later program executions. A possible solution to this problem is to search for some sort of meta-information of terms before starting the process of tabling. This meta-information could be stored in additional tables, similar to the data tables. Such information could include, for each tabled subgoal, constant atomic terms values, variable terms' internal indexes and total count. If such meta-data tuples could be found, its information could be used to not only to rebuild the corresponding branch in the subgoal trie but also to reconstruct the prepared statements required to both store new found answers and retrieve previously computed ones.

The current implementation needs to be tested more intensively with a wider range of applications. Many opportunities for refining the system exist, and more will almost certainly be uncovered with profound experimentation of the system. The system still has some limitations that may reduce its use elsewhere and its contribution in the support of realistic applications.

7.3 Final Remark

The research we have developed in this thesis context was built on the vigorous research effort produced by the preceding researchers. Their ideas enlightened us, guiding our steps during the entire design and development processes. With our work, we hope to have somewhat contributed to logic programming dissemination. Hopefully, we might have also started a path that others may follow.

References

[AB94] K. Apt and R. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19 & 20:9–72, 1994.

[AK91] H. Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction.* MIT Press, 1991.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. The universality of data retrieval languages. In *POPL*, pages 110–120, 1979.

[AvE82] Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.

[AYDF04] Sihem Amer-Yahia, Fang Du, and Juliana Freire. A comprehensive solution to the xml-to-relational mapping problem. In *WIDM '04: Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 31–38, New York, NY, USA, 2004. ACM Press.

[BCR93] L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative Commutative Discrimination Nets. In *International Joint Conference on Theory and Practice of Software Development*, number 668 in LNCS, pages 61–74. Springer-Verlag, 1993.

[BE77] Mike W. Blasgen and Kapali P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):362–377, 1977.

[BFRS02] Philip Bohannon, Juliana Freire, Prasan Roy, and Jme Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM Press.

[Boi88] M. Boizumault. *Prolog: l'implantation. Etudes et recherches en informatique.* Masson, Paris, France, 1988.

[BR87] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Log. Program.*, 4(3):259–262, 1987.

[BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3 & 4):255–299, 1991.

[CA97] Michael J. Corey and Michael Abbey. *Oracle8: A Beginner's Guide.* ORACLE Press, 1997.

[Car90] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, The Royal Institute of Technology, 1990.

[CGC+04] J. Correas, J.M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations). In *Practical Aspects of Declarative Languages: 6th International Symposium*, number 3057 in LNCS, pages 104–119. Springer-Verlag, 2004.

[CH80] Ashok K. Chandra and David Harel. Structure and complexity of relational queries. In *IEEE Symposium on Foundations of Computer Science*, pages 333–347, 1980.

[CKPR73] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un syste de communication homme–machine en francais. Technical report cri 72-18, Groupe Intelligence Artificielle, UniversitAix-Marseille II, 1973.

[Cla78] K. Clark. Negation as Failure. In *Logic and DataBases*, pages 293–322. Plenum Press, 1978.

[CM94] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, fourth edition, 1994.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[CRF06] P. Costa, R. Rocha, and M. Ferreira. DBTAB: a Relational Storage Model for the YapTab Tabling System. In E. Pontelli and Hai-Feng Guo, editors, *Proceedings of the Colloquium on Implementation of Constraint and LOgic Programming Systems, CICLOPS'2006*, pages 95–109, Seattle, Washington, USA, August 2006.

[CSW95] W. Chen, T. Swift, and D. S. Warren. Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[CW96] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[Die87] S. Dietrich. *Extension Tables for Recursive Query Evaluation*. PhD thesis, Department of Computer Science, State University of New York, 1987.

[DNB93] David J. DeWitt, Jeffrey F. Naughton, and Joseph Burger. Nested loops revisited. In *PDIS*, pages 230–242, 1993.

[Dra91] C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates*. PhD thesis, Zurich University, 1991.

[Dra92] C. Draxler. Accessing Relational and NF^2 Databases Through Database Set Predicates. In *UK Annual Conference on Logic Programming*, Workshops in Computing, pages 156–173. Springer Verlag, 1992.

[FK99] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[FR04] M. Ferreira and R. Rocha. The MyYapDB Deductive Database System. In *European Conference on Logics in Artificial Intelligence*, number 3229 in LNAI, pages 710–713. Springer-Verlag, 2004.

[FR05] M. Ferreira and R. Rocha. Coupling OPTYap with a Database System. In *IADIS International Conference Applied Computing*, volume II, pages 107–114. IADIS Press, 2005.

[FRS04] M. Ferreira, R. Rocha, and S. Silva. Comparing Alternative Approaches for Coupling Logic Programming with Relational Databases. In *Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 71–82, 2004.

[FSW96] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in LNCS, pages 243–258. Springer-Verlag, 1996.

[FSW97] J. Freire, T. Swift, and D. S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In *International Conference on Logic Programming*, pages 198–212, 1997.

[FW97] J. Freire and D. S. Warren. Combining Scheduling Strategies in Tabled Evaluation. In *Workshop on Parallelism and Implementation Technology for Logic Programming*, 1997.

[Gen03] J. Gennick. Querying Hierarchies: Top-of-the-Line Support. Technical report, ORACLE Technology Network, 2003. http://www.oracle.com/technology/oramag/webcolumns/2003/techarticles/gennick_connectedby.html.

[GR68] C. Cordell Green and Bertram Raphael. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 1968 23rd ACM national conference*, pages 169–181, New York, NY, USA, 1968. ACM Press.

[IvE92] H. Ibrahim and M. H. van Emden. Towards applicative relational programming, March 1992.

[Kar92] R. Karlsson. *A High Performance OR-parallel Prolog System.* PhD thesis, The Royal Institute of Technology, 1992.

[Kow74] R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing*, pages 569–574. North-Holland, 1974.

[Kow79] R. Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series. North-Holland, 1979.

[KT79] Charles Kellogg and Larry Travis. Reasoning with data in a deductively augmented data management system. In *Advances in Data Base Theory*, pages 261–295, 1979.

[Lie82] Y. Edmund Lien. On the equivalence of database models. *J. ACM*, 29(2):333–362, 1982.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[Mic68] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[MN83] Jack Minker and Jean-Marie Nicolas. On recursive axioms in deductive databases. *Inf. Syst.*, 8(1):1–13, 1983.

[MYK95] Weiyi Meng, Clement Yu, and Won Kim. A theory of translation from relational queries to hierarchical queries. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):228–245, 1995.

[PDR91] Geoffrey Phipps, Marcia A. Derr, and Kenneth A. Ross. Glue-nail: a deductive database system. In *ACM SIGMOD Conference on Management of Data*, pages 308–317, 1991.

[Rei78] R. Reiter. On closed-word data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.

[Rob65] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Roc01] R. Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs*. PhD thesis, Department of Computer Science, University of Porto, 2001.

[Roc06] R. Rocha. Handling Incomplete and Complete Tables in Tabled Logic Programs. In *International Conference on Logic Programming*, number

4079 in LNCS, pages 427–428, Seattle, Washington, USA, 2006. Springer-Verlag.

[Roc07] R. Rocha. On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In M. Hanus, editor, *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages, PADL'2007*, number 4354 in LNCS, pages 155–169, Nice, France, January 2007. Springer-Verlag.

[Roy90] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

[RRS+95] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *International Conference on Logic Programming*, pages 687–711. The MIT Press, 1995.

[RRS+99] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

[RSC05] R. Rocha, F. Silva, and V. Santos Costa. Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In *International Conference on Logic Programming*, number 3668 in LNCS, pages 250–264. Springer-Verlag, 2005.

[RSS92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: A Deductive Database Programming Language. In *International Conference on Very Large Data Bases*. Morgan Kaufmann, 1992.

[RSS97] R. Rocha, F. Silva, and V. Santos Costa. On Applying Or-Parallelism to Tabled Evaluations. In *International Workshop on Tabling in Logic Programming*, pages 33–45, 1997.

[RSS00] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[RU93] R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[Sag96] K. Sagonas. *The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs*. PhD thesis, Department of Computer Science, State University of New York, 1996.

[San99] V. Santos Costa. Optimising Bytecode Emulation for Prolog. In *Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 261–267. Springer-Verlag, 1999.

[SC90] Eugene J. Shekita and Michael J. Carey. A performance evaluation of pointer-based joins. *SIGMOD Rec.*, 19(2):300–311, 1990.

[SDRA] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*. Available from `http://www.ncc.up.pt/~vsc/Yap`.

[SFRF06] T. Soares, M. Ferreira, R. Rocha, and N. A. Fonseca. On Applying Deductive Databases to Inductive Logic Programming: a Performance Study. In E. Pontelli and Hai-Feng Guo, editors, *Proceedings of the Colloquium on Implementation of Constraint and LOgic Programming Systems, CICLOPS'2006*, pages 80–94, Seattle, Washington, USA, August 2006.

[SKWW01] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of XML documents. *Lecture Notes in Computer Science*, 1997:137+, 2001.

[SR05] D. Saha and C. R. Ramakrishnan. Incremental Evaluation of Tabled Logic Programs. In *International Conference on Logic Programming*, number 3668 in LNCS, pages 235–249. Springer-Verlag, 2005.

[SS86] Leon Sterling and Ehud Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[SS98] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[SS04] K. Sagonas and P. Stuckey. Just Enough Tabling. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 78–89. ACM, 2004.

[SSW94] 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.

[SSW96] K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Computing the Well-Founded Semantics. In *Joint International Conference and Symposium on Logic Programming*, pages 274–288. The MIT Press, 1996.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.

[SW94] T. Swift and D. S. Warren. An abstract machine for SLG resolution: Definite Programs. In *International Logic Programming Symposium*, pages 633–652. The MIT Press, 1994.

[SWS+] K. Sagonas, D. S. Warren, T. Swift, P. Rao, S. Dawson, J. Freire, E. Johnson, B. Cui, M. Kifer, B. Demoen, and L. F. Castro. *XSB Programmers' Manual*. Available from `http://xsb.sourceforge.net`.

[TS86] H. Tamaki and T. Sato. OLDT Resolution with Tabulation. In *International Conference on Logic Programming*, number 225 in LNCS, pages 84–98. Springer-Verlag, 1986.

[TSR05] M. Ferreira T. Soares and R. Rocha. The MYDDAS Programmer's Manual. Technical Report DCC-2005-10, Department of Computer Science, University of Porto, 2005.

[TVB+02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215, New York, NY, USA, 2002. ACM Press.

[TZ86] 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.

[vEBvEB86] Ghica van Emde Boas and Peter van Emde Boas. Storing and evaluating horn-clause rules in a relational database. *IBM J. Res. Dev.*, 30(1):80–92, 1986.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[Vie89] L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.

[VRK+93] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask, and J. Harland. The Aditi Deductive Database System. Technical Report 93/10, School of Information Technology and Electrical Engineering, Univ. of Melbourne, 1993.

[WA02] M. Widenius and D. Axmark. *MySQL Reference Manual: Documentation from the Source*. O'Reilly Community Press, 2002.

[Wal93] A. Walker. Backchain Iteration: Towards a practical inference method that is simple enough to be proved terminating, sound, and complete. *Journal of Automated Reasoning*, 11(1):1–23, 1993.

[War77] D. H. D. Warren. *Applied Logic – Its Use and Implementation as a Programming Tool*. PhD thesis, Edinburgh University, 1977.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.