Universidade da Coruña

Departamento de Computación

Chase of datalog programs and its application to solve the functional dependencies implication problem

A Coruña, Decembro de 2001

Doutorando
José Ramón Paramá Gabía

Directores
Nieves R. Brisaboa e Héctor J. Hernández

Ph.D. Thesis directed by
Tese doutoral dirixida por

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultade de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
`brisaboa@udc.es`

Héctor J. Hernández
Computer Science Department
Texas Tech University
Lubbock, TX 79409, USA
`hector@cs.ttu.edu`

“Es, pues, de saber, que este sobredicho hidalgo, los ratos que estaba ocioso -que eran los más del año-, se daba a leer libros de caballerías con tanta afición y gusto, que olvidó casi de todo punto el ejercicio de la caza, y aun la administración de la hacienda; y llegó a tanto su curiosidad y desatino en esto, que vendió muchas hanegas de tierra de sembradura para comprar libros de caballerías en que leer, y así, llevó a su casa todos cuantos pudo haber dellos... Con estas razones perdía el pobre caballero el juicio, y desvelbase por entenderlas y desentrañarles el sentido que no se lo sacara ni las entendiera el mesmo Aristóteles, si resucitara para sólo ello... En resolución, él se enfrascó tanto en su lectura, que se le pasaban las noches leyendo de claro en claro, y los días de turbio en turbio; y así, del poco dormir y del mucho leer se le secó el cerebro, de manera que vino a perder el juicio”.

El ingenioso hidalgo Don Quijote de la Mancha, cap. 1.

Gracias Raquel.

“Nós creemos que a auga doce dos ríos pode facer doce a auga salgada do mar; que a morte, enchéndose de vidas, será vida; que a “nada”, enchéndose de ilusións, será “todo”...”

Alfonso R. Castelao

Gracias mon.

Acknowledgments

Thanks to Dra. Nieves Rodríguez Brisaboa for giving me the opportunity of studying the PhD and providing me everything that I needed (including encouragement). Also, I would like to thank her for being a friend and at the same time my advisor.

Thanks to Dr. Héctor Hernández for hosting me kindly in his universities and homes and for all those nice "cancer breaks" and other moments from which I learnt a lot.

Thanks, especially for those "coffee breaks" and laughs, to all (actual and former) members of the database laboratory at the Universidade da Coruña, Dr. Miguel Penabad, Manuel, Eva, María, Dr. Mon López Rodríguez (with "doctor" in front of your name for first time, I guess), Charo, Ángeles, Mariajo, Fran, Miguel Luaces, Fari, José Ramón and Toni.

Let me express my special acknowledgements to Dr. Miguel Penabad for his comments, suggestions and, of course, corrections about my Thesis. I know that it was a hard job.

Thanks to all my family for their love and support. I want to make a special acknowledgement to my brother Juan and my grandmother Lola for encouraging me to study when I was only a child.

Thank you Raquel for all your love, support and for standing most of my bad moments during this Thesis. I am sure that you will finish your Thesis in the future, you deserve it.

Thanks to all my good friends, especially (in no particular order) to Miguel, Ramiro, Mon (thank you for those ICQ nice chats), Pili (thank you also for our ICQ chats when I was alone), Dani, Manuel, Eva, Ton and Silvia. Thank you for hearing me and giving me strength to continue in the bad moments.

I also would like to thank to Dra. Sagrario López Poza and all of the institutions[1] who have made possible the database laboratory of the Universidade da Courña.

[1]This work is partially supported by Grant # TEL99-0335-C04-02 of Comisión Interministerial de Ciencia y Tecnología (CICYT).

Agradecementos

Quero agradecer á miña directora de tese, a Doutora Nieves Rodríguez Brisaboa, que me dera a oportunidade de estudiar o doutorado e me proporcionara todo o necesario (incluíndo ánimo) para acadar dito obxectivo. Tamén quero facer expreso o meu agradecemento por ser unha amiga e, ao mesmo tempo, cando era necesario, a miña directora.

O meu agradecemento tamén ao Doutor Héctor Hernández por terme acollido dun xeito tan amable tanto no traballo como no seu fogar. Desexo darlle as gracias polos "cancer breaks" e tódolos bos intres que pasamos xuntos e dos que aprendín moito.

Gracias, especialmente polos "coffe breaks" e as risas que botamos xuntos, a tódolos membros (actuais e pasados) do laboratorio de bases de datos da Universidade da Coruña, Dr. Miguel Penabad, Manuel, Eva, María, Dr. Mon López Rodríguez (con "doutor" diante do teu nome por primeira vez, creo), Charo, Ángeles, Mariajo, Miguel Luaces, Fari, José Ramón e Toni.

Quero expresar o meu especial agradecemento ao Dr. Miguel R. Penabad polos seus comentarios, suxerencias e por suposto correccións sobre a miña Tese, sei que foi un traballo difícil.

Gracias a toda a miña familia polo seu apoio, amor e ánimo. Quero facer especial mención do meu irmán Juan e a miña avoa Lola que me animaron a estudar cando aínda era un neno.

Gracias a Raquel por todo o seu amor, apoio e por aguantarme nos malos momentos durante esta Tese, ela foi entón, o meu principal apoio. Estou seguro que rematarás a túa Tese nun futuro máis ou menos lonxano, o mereces.

Gracias a tódolos meus bos amigos, especialmente (sen ningún orden en particular) a Miguel, Ramiro, Mon (gracias polos bos ratos que pasei falando contigo por ICQ), Pili (gracias tamén por aquelas paroladas que botamos cando estaba so), Dani, Manuel, Eva, Ton e Silvia. Gracias por aguantar as miñas queixas e por darme forzas para seguir nos malos momentos.

Tamén quero facer constar o meu agradecemento á Doutora Sagrario López Poza e a tódalas institucións galegas e españolas[2] que fixeron posible a existencia do laboratorio de bases de datos da Universidade da Coruña.

[2]Esta Tese foi realizada en parte con medios subvencionados polo proxecto TEL99-0335-C04-02 da Comisión Interministerial de Ciencia y Tecnología (CICYT).

A Raquel e Juan

Contents

Chapter 1

Introduction

At the end of the 1970's, the relational model [Cod80] was quickly becoming the database model of choice among researchers of the database field. At about the same time, some researchers started to report on the limitations of query languages over relational databases. One example is a paper by Aho and Ullman [AU79] describing the inability of those query languages to express the transitive closure problem. Due to this limited expressive power of relational query languages, a strong movement towards investigating new models started to develop.

In order to solve the problems found in the relational query languages, several researchers focused in logic. Note that the relational model has a strong logical basis. In fact, in this model, information is considered implicitly as an interpretation of a first order theory [NG77]. This connection was not explicitly exploited in the design of the relational query languages. However, some researchers saw the power and advantages that a more explicit relationship of logic and databases would bring:

> "Mathematical logic provides a conceptual framework for many different areas of science. It has been recognized recently that logic is also significant for databases." [GM77]

This is the foreword of the book "Logic and Databases" edited by Gallaire and Minker. This text is a collection of papers that were presented at a workshop held in Tolouse, France, on November 1977. This foreword continued: "...It will be seen that logic can be used as a programming language, as a query language, to perform deductive searches, to maintain the integrity of databases, to provide a formalism for handling negative information, to generalize concepts in knowledge representation, and to represent and manipulate data structures. Thus, logic provides a powerful tool for databases that is accomplished by no other approach developed to date [GM77]." That workshop, whose main subject was the interaction of logic and databases, is considered by most researchers as the birth of "deductive databases." Since then, the term *deductive databases* has been used to denote database systems whose data manipulation language is expressed using logic clauses (i.e., rules).

Two decades after the birth of deductive databases, there is some consensus that this field has reached maturity. A large body of research exists [Ull88, Ull89, AH88, Min88a, Min88b] and several prototypes have been developed in order to demonstrate the feasibility of this database model [Ull89, RBSS90]. Most of the prototypes use datalog as query language. Informally, datalog can be seen as Prolog without function symbols.

It is not difficult to argue that datalog, as a query language, is more convenient than the languages based on the relational model. To start with, the extent of predicates in a datalog program can be naturally interpreted as relations. Therefore, datalog can be seen as a powerful extension to the relational model.

The expressive power in datalog is based mainly in its ability to define relations in terms of themselves; that is, it can express recursive queries (something that could not be done in the relational query languages until SQL99).

There has been much research in datalog since deductive databases were first mentioned as a viable database framework. However, several problems remain to be solved. While the field is mature, there is need for research in several areas like integrity constraint satisfaction and query optimization.

Query optimization is a general term for referring to techniques used to speed up queries in database management systems. In datalog, optimizing queries is very important since its ability to express recursive queries may result in slow response time. Thus, query optimization has been a very active research area in datalog.

The recent appearance of the new SQL standard SQL99 [MS02, UW97] reaffirms the necessity of research in this area, given that SQL99 includes queries with linear recursion (that is the type of recursion that we study in this work). Previous standards of SQL did not include recursion, thus now it is necessary to increase the research in query optimization to provide the suitable algorithms that will be included in the query optimizers of the database management systems in order to speed up the execution of recursive queries.

Note that as we pointed out, datalog can be seen as an extension of the relational model, thus even though we use in this dissertation datalog syntax, our results are applicable to DBMS with SQL99 syntax straightforward.

In order to optimize recursive datalog programs, the first approach was to try to see if it is possible to remove the recursion [Var88]; this is equivalent to testing whether there is a non-recursive datalog program that is equivalent to the recursive one. If this is the case, the recursive program is said to be *bounded*[a] [Nau86, NS87]. In general, the problem of testing whether a datalog program is bounded is known to be undecidable even for linear programs with one intensional predicate [GMSV87].

If the program is not known to be bounded, an attractive alternative approach is to see if we can somehow transform the program to make the recursion "smaller" and "cheaper" to evaluate. One possibility to do that is the *semantic query optimization* that uses integrity constraints associated with databases in order to improve the efficiency of the query evaluation [CGM88].

In this work, we use a semantic query optimization approach to optimize datalog programs when the input databases satisfy an important class of integrity constraints, the functional dependencies (fds).

Two of our main contributions in this dissertation are two algorithms developed in order to speed up the execution of datalog programs: *the chase of datalog programs* ($Chase_F(P)$) and *the cyclic chase of datalog programs ($CChase_F(P)$).* The chase of datalog programs is introduced in Chapter 5 and the cyclic chase of datalog programs is showed in Chapter 6. Both algorithms have as their input a linear recursive datalog program P and a set of fds F, and the output is a program P' equivalent to P when both (P and P') are evaluated over databases satisfying F. The algorithms do not optimize an execution of a datalog program over a certain database, rather they provide an alternative program that is obtained in compile time since the new program is independent of the database to which it will be applied. The new program is equivalent to the original one when it is applied to any database that satisfies the set of fds used to compute such a program.

The new programs are cheaper to evaluate than the original one. However, when they do not obtain a program that is cheaper to evaluate than the original one, that means that the output of the algorithms is the program provided as input. That is, they never make things worst.

[a] A datalog program P is *bounded* when there is a non-recursive datalog program equivalent to P. Intiutively, it is very easy to see that it is much more cheaper to evaluate a bounded datalog program than a unbounded one.

Both algorithms have their roots in the "chase", a term that appears for the first time in the lossless join test of Aho, Beeri, and Ullman [ABU79, Ull88]. Despite the original objective of the chase, we use its main idea (the equalization of symbols following functional dependencies) with a different target. Our algorithms obtain, from a datalog program P and a set of fds F, an equivalent program P' where its recursive rules may have less different variables and, sometimes, less atoms. That is, our algorithms obtain a program where the variables may be equated among them due to the effect of funcional dependencies. Moreover, due to those equalizations of variables, an unbounded datalog program P may become a bounded datalog program!

The reader may wonder why do we develop two algorithms that have the same objective. The reason is that depending on the datalog program P that we are trying to optimize and the set F of functional dependencies involved, it would be more suitable either the $Chase_F(P)$ or the $CChase_F(P)$. However, it is possible the combination of both algorithms. Given datalog program P and a set of functional dependencies F, it is mandatory to apply first the chase of datalog programs ($Chase_F(P)$) and then, over the result (say P'), it is possible to apply the cyclic chase of datalog programs ($CChase_F(P')$) to obtain a new program P'' that obtains benefits from both algorithms.

With the next examples, we give an intuitive idea of how our algorithms optimize datalog programs. At this point we do not show how the algorithms work, but only the advantages of the programs they output with respect to the original ones.

Example 1.0.1 Let $P = \{r_0, r_1\}$, where:
$r_0 = p(X,Y) :- e(X,Y)$
$r_1 = p(X,Y) :- a(X,Z), e(Z,Y), a(X,X), p(Z,Y)$

Note that P is an unbounded datalog program.

Let F be the set of fds $F = \{e : \{1\} \to \{2\}, a : \{1\} \to \{2\}\}$. The fd $e : \{1\} \to \{2\}$ indicates that the set of facts over the predicate e satisfies that the values of the first argument determine the values in the second position. The other functional dependency has a similar meaning. For example, the atoms $e(1,3)$ and $e(1,4)$ violate the fd $e : \{1\} \to \{2\}$.

As we shall show later, $Chase_F(P)$ obtains, in this case, a bounded program $P' = \{s_0, s_1\}$ with less variables and less atoms:

$s_0 = p(X,Y) :- e(X,Y)$
$s_1 = p(X,Y) : -a(X,X), e(X,Y), p(X,Y)$

Note that s_1 has only two different variables and three atoms whereas r_1 has three different variables and four atoms.

It is obvious that P' is less expensive to evaluate. If we observe P', the rule s_1 has the atom $e(X,Y)$ (the atom in the body of s_0) in its body and the head is equal to the head of s_0 thus it does not obtain any fact that it is not obtained by s_0. Then it is easy to see that P' is bounded!

So far, our algorithm does not remove s_1, however another algorithms have been developed in order to remove redundant atoms and rules [Sag87]. $\square$

Example 1.0.2 Let $P = \{r_0, r_1\}$ be:
r_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
r_1: $p(X,Y,Z,A,B,C) :- e(Y,X,Y,C,A,D), p(Z,X,Y,B,C,D)$

Let F be $\{e : \{6\} \rightarrow \{1\},\ e : \{6\} \rightarrow \{4\}\}$. The $Chase_F(P)$ does not introduce any optimization in this case, however the $CChase_F(P)$ produces an optimized datalog program $P' = \{s_0, s_1, s_2, s_3\}$:

s_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
s_1: $p(X,Y,Y,A,B,B) :- e(Y,X,Y,B,A,D), e(Y,X,Y,B,B,D)$
s_2: $p(X,X,Z,A,B,C) :- e(X,X,X,C,A,C), e(X,Z,X,C,B,D^1), e(X,Z,X,C,C,D^1)$
s_3: $p(X,X,X,A,B,C) :- e(X,X,X,C,A,D), p(X,X,X,B,C,D)$

Notice that in r_1, there are six different variables whereas in s_3 (the only recursive rule of P') there are only four different variables. The reader may note that the algorithm adds rules, but those rules are non-recursive rules, and all of them have some variables equated. $\square$

Other aspect of this Thesis (shown in Chapter 7) obtains results in the area of integrity constraint satisfaction. *Integrity constraints* are general laws that databases must satisfy [Ull88]. They arise naturally in practical applications and restrict the domain of the input databases to a subset of all the possible input databases. Functional dependencies [Ull88] are a type of integrity constraints that all valid databases are required to fulfill through time.

Research in satisfaction of functional dependencies is very important. Note that for example, violations of other types of constraints as *tuple generating dependency* are due to the lack of complete information about the world (i.e., the lack of certain facts asserted by the tuple-generating dependency), and therefore they can be removed by adding more information to make it complete, such as by treating all given tuple-generating dependencies as additional rules of the program. On the other hand, violations of constraints as the functional dependencies are due to the presence of certain "incorrect" information in the database, such as the same social security number assigned to two persons. There is no way to remove such violations without undoing updates or discarding some original information. In the latter case, it is no always clear what portion should be discarded. Therefore, more attention should be paid to enforcement of constraints of this kind.

One of the most well known problems related with functional dependencies satisfaction in datalog is the *preservation of functional dependencies.* Given a datalog program, the fact that an input database satisfies a given set of functional dependencies does not necessarily imply that the output database still satisfies these fds. Thus, it is interesting to know whether the output satisfies or not the given fds.

Other interesting problem is the *implication of functional dependencies* (also known as the *FD-FD implication problem*). This problem has to do with the discover of new functional dependencies. That is, given a datalog program P and a set of fds F satisfied by the input database, the output of this program with such database as input may produce a database satisfying a new set of functional dependencies. Then, the new set of fds satisfied by the output are *implied* by F (in P).

We use again the basic idea of the chase (i.e. the equalization of symbols following functional dependencies) to tackle the *FD-FD implication problem.*

In this work we provide two methods to check if a given set of fds will be satisfied by the output database without computing such database. With such objective, we introduce a syntactic condition that serves us to decide if a program, which is in a subclass of linear datalog programs, implies a fd (from a given set of fds). Besides, provided that the scheme of the database is in Boyce Codd Normal Form with respect to the functional dependencies, we offer a syntactic condition that serves us to identify programs that do not imply a specific fd.

The following two examples illustrate the FD-FD implication problem and show us its difficulty.

Example 1.0.3 Let $P = \{r_0, r_1\}$, where
$r_0 : p(X, X, Y) : -e(X, Y).$
$r_1 : p(X, Y, Y) : -e(Y, Y), p(X, X, Y).$

Let $F = \{e : \{1\} \rightarrow \{2\}\}$. We shall consider only databases that satisfy F. Let $f = p : \{1\} \rightarrow \{3\}$. We shall see that P implies f.

Let us suppose that d contains the following facts:

$e(1,2)$
$e(2,4)$
$e(4,4)$

Note that these facts trivially satisfy F. If we apply P to d, the output $(P(d))$ contains:

$e(1,2)$	$p(1,1,2)$
$e(2,4)$	$p(2,2,4)$
$e(4,4)$	$p(4,4,4)$
	$p(2,4,4)$

We can see that the atoms defined over p satisfy the fd f. In fact, we can see that if d satisfies F, $P(d)$ will always satisfy f. This is true given that positions 1 and 3 of the two p-atoms of r_1 have the same variables (and in the same order) and positions 1 and 3 of the p-atom of r_0 have the same variables (and in the same order) as positions 1 and 2 of the e-atom in the body of the rule. □

Example 1.0.4 Let $P = \{r_0, r_1\}$ the typical chain program, where:
$r_0 : p(X,Y) : -e(X,Y)$.
$r_1 : p(X,Y) : -e(X,Z), p(Z,Y)$.

Let $F = \{e : \{1\} \rightarrow \{2\}\}$ and $f = p : \{1\} \rightarrow \{2\}$. We shall see that P does not imply f. Let us assume that d contains the following facts:

$e(1,2)$
$e(2,3)$

If we apply P to d, $P(d)$ contains:

$e(1,2)$	$p(1,2)$
$e(2,3)$	$p(2,3)$
	$p(1,3)$

In this case, we can see that the atoms defined over p do not satisfy the fd f, because $p(1,2)$ and $p(1,3)$ violates $p : \{1\} \rightarrow \{2\}$. □

The outline of this dissertation is as follows. In Chapter 2, we give the basic definitions and some basic results related to our work. In Chapter 3, we define the chase of rules and trees and some results related to these definitions are given.

In Chapter 4, we introduce a concept very important in this Thesis, equalization chains. In this chapter can be found several results that we developed for this Thesis. Although, those results can be used in other works, the main contributions of this Thesis are in the following three chapters.

In Chapter 5, we introduce our first algorithm to optimize datalog programs, *the chase of datalog programs* ($Chase_F(P)$). In Chapter 6, we show the second algorithm that optimizes datalog programs, *the cyclic chase of datalog programs* ($CChase_F(P)$). In Chapter 7, we present our results related to the FD-FD implication problem.

Finally, in Chapter 8, we present our conclusions and directions for future work.

Chapter 2

Basic Definitions

Although the aim of this dissertation is to provide algorithms to be included in commercial DBMS (which use SQL99 syntax), we use datalog syntax that is easier to manipulate than the SQL syntax. Note that since the underlying mathematical model of data for datalog is essentially that of the relational model, therefore any datalog query can be implemented as sequence of steps in relational algebra [Ull88].

In fact, as we will see in this chapter, with the new standard SQL99, the translation from some recursive datalog programs to a SQL99 query can be done in just one step.

The outline of this chapter is as follows. In Section 2.1, we define the syntax of datalog. In Section 2.3, we give definitions related to databases. In Section 2.4, we define datalog programs and we provide several definitions related with them. In Section 2.5, we give definitions related to functional dependencies. Finally, in Section 2.6, we define what a tree is and we provide some results and definitions related to them.

2.1 Datalog

Datalog is one of the most popular query languages for deductive databases. Informally, datalog can be seen as a version of Prolog suitable for database systems. Indeed, datalog is basically equal to Prolog without function symbols.

The underlying mathematical model of data for datalog is essentially that of the relational model. A *predicate name* (*predicate symbol* or just simply *predicate*) in datalog defines a relation as in the relational model. However, as in the formal definition of relational algebra, these relations do not have attributes to name their columns. Rather they are relations in the set-of-lists sense, where components appear in a fixed order, and reference to a column is only by its position among the arguments of a given predicate symbol.

We shall assume that each predicate symbol is associated with a particular number of arguments that it takes. Therefore, for each predicate name p we assume the existence of a total function, denoted by α, from p to the set of strictly positive integers. If $\alpha(p) = n$, we say that the *arity of* p *is* n, that is, such predicate has n attributes.

A *term* is a constant or a variable. An *atomic formula* (or just simply *atom*), is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate name of arity n and each of the t_i's, $1 \leq i \leq n$, is a term. Sometimes, when we refer to an atom over the predicate name p, we call it $p - atom$.

Basically, a predicate is the name of a function that returns a Boolean value. If P is a relation with n attributes in a fixed order, we use p as the name of a predicate correspondent to this relation. Let $a_1, a_2, \ldots, a_n$ be constants. Then, the atom $p(a_1, a_2, \ldots, a_n)$ is `TRUE`, if $(a_1, a_2, \ldots, a_n)$ is a tuple of P; likewise, if $(a_1, a_2, \ldots, a_n)$ is not a tuple of P, then $p(a_1, a_2, \ldots, a_n)$ is `FALSE`.

A predicate may contain variables and constants as terms. If an atom contains variables in one or more of its terms, it will be a function that returns `TRUE` or `FALSE` depending on the values assigned to the variables when the predicate is evaluated over a database.

Example 2.1.1 If p is a predicate with arity 2, and the relation of p is:

A	B
1	2
3	4
3	3

Then, $p(X, Y)$ is a function that shows that, for all X and Y, whether the tuple X, Y is in the relation P. For example, $p(X, Y)$ returns `TRUE` if $X = 1$ and $Y = 2$ whereas if $X = 2$ and $Y = 2$, $p(X, Y)$ returns `FALSE`.

Note that $p(X, X)$ can only be `TRUE` when $X = 3$.

□

Another distinction between the relational model and the datalog model is that in datalog there are two types of relations that can be defined. A predicate whose relation is stored in the database is called an *extensional database* (EDB) relation, whereas one defined by logical rules is called an *intensional database* (IDB) relation. We assume that each predicate symbol either denotes an EDB relation or an IDB relation, but not both.

In relational model, all relations are EDB relations.

We use the following convention when we talk about predicates, variables and constants. We denote variables by strings of characters starting with a capital letter, and constants and predicate names by strings of characters starting with a lower case letter. Two constants (respectively, variables, predicate names) are *distinct* unless they are syntactically the same (they have the same name).

Given an atom q of the form $p(t_1, \ldots, t_k)$, we say that $q[n]$, $1 \leq n \leq k$, is the *n-th component* of q. Given a set of argument positions m and a predicate q, $q[m]$ denotes the tuple formed by the terms in the positions defined by m.

Example 2.1.2 Consider the atom $q_1 = e(X, Y, a)$. $q_1[1] = X$, $q_1[\{2, 3\}] = q_1[2, 3] = (Y, a)$. $\square$

A *literal* is either an atomic formula or a *negated* atomic formula. A negated atomic formula is a *negative literal* and an atomic formula is a *positive literal*. We denote negative literals by $\neg q$, where q is an atom. A *clause* is a formula of the form:

$$\forall X_1 \ldots \forall X_s (L_1 \vee \ldots \vee L_m)$$

where the L_i's are literals and the X_j's are the variables that appear in the L_i's.

A *Horn clause* is a clause with at most one positive literal. A horn clause is thus either:

1. A single positive literal, e.g., $p(X, Y)$, which we regard as a *fact*. When all the terms in the fact are constants, we say that it is a *ground fact*.

2. A positive literal and one or more negative literals, which is a *rule*.

3. One or more negative literals, with no positive literal, which is an integrity constraint, and which will not be considered in our discussion.

We shall use Prolog notation to represent Horn clauses.

Example 2.1.3 Consider the following Horn clause:

$$\forall X \forall Y \forall Z (\neg p(X, Z) \vee \neg p(Z, Y) \vee p(X, Y))$$

We represent it as:

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

□

The *head* of a (datalog) rule is the atom to the left of the symbol ":-". The *body* of a rule is the list (conjunction) of atoms to the right of ":-".

A rule is *safe* if each of the variables in the head of the rule appears in one of the atoms in the body of the rule. Any predicate that appears in the head of a rule is called an *IDB predicate*; all others are called *EDB predicates.*

An atom is an *EDB (IDB) atom* if its predicate name is an EDB (IDB) predicate.

Given a rule r, all variables that appear in the head of r are called *distinguished*; all other variables of r are called *non-distinguished.*

2.2 Substitutions

A substitution is a finite set of pairs of the form X_i/t_i where X_i is a variable and t_i is a term, which is either a variable or a constant.

The result of applying a substitution, say θ, to an atom A, denoted by $\theta(A)$, is the atom A with each occurrence of X replaced by t for every pair X/t in θ. For example, consider $\theta = \{X/a, Y/b\}$ and the atom $p(X, Y)$, then $\theta(p(X, Y))$ will be $p(a, b)$. A substitution θ can be applied to a set of atoms, to a rule or to a tree to get another set of atoms, rule or tree with each occurrence X replaced by t for every X/t in θ.

Given a rule r, $\theta(r)$, an *instantiation of* r, is the rule obtained from r by replacing each literal l_i in r by $\theta(l_i)$. An instantiation of r is *ground* if for all pairs X/t in θ, t is a constant.

Let σ and θ be two substitutions. In order to obtain the *composition* of σ and θ ($\sigma(\theta)$), first apply θ and over the result, apply σ.

2.3 Databases and Relations

A *datalog database schema* is a finite set of predicates. We assume that all predicates used in the rest of this dissertation are implicitly in $\mathcal{U}$, a fixed datalog database schema.

Given a predicate e, *the relation associated with* e is a finite set of ground facts about e. A *database* is a finite set of ground facts about predicate names in $\mathcal{U}$. An *extensional database (EDB)* is a database that does not contain facts about IDB predicates.

2.4 Datalog Programs

A *(datalog) program* is a finite set of rules. A program is *safe* if all the rules in the program are safe. We only consider safe datalog programs; we shall refer to them simply as programs. We denote the set of predicates in a program by $pred(P)$, the set of EDB predicates in the program by $EDB(P)$, and the set of IDB predicates in the program by $IDB(P)$.

For simplicity, we do not allow constants in the programs.

We say that a program P *defines* a predicate p if p is the only IDB predicate in $pred(P)$. A *single recursive rule program (sirup)* [CK86] is a program that consists of exactly one recursive rule and several non-recursive rules and the program defines a predicate p. A *1-sirup* is a program that consists of exactly one recursive rule and it does not have any non-recursive rule. A *2-sirup* is a sirup that contains only one non-recursive rule (and one recursive rule) and the non-recursive rule has only one atom in its body. A rule is *linear* if there is at most one IDB atom in its body. A *linear sirup (lsirup)* [Var88] is a sirup such that its rules are linear. A *1-lsirup (2-lsirup)* is a 1-sirup (2-sirup, respectively) such that its rules are linear. That is, a $2 - lsirup$ is a program defining a predicate p with one non-recursive rule, which has only one atom in its body, and one recursive rule, which has only one IDB atom in its body.

From now on, we denote with r_1 the recursive rule in a $2-lsirup$ whereas we use r_0 to denote the non-recursive rule.

In the next subsection we are going to give a brief and intuitively idea of the operations that can be done with a $2 - lsirup$. For an extended explanation see [Ull88].

2.4.1 From relational algebra to datalog

The relational algebra operators can be replicated by one or more datalog rules. In this section, we analyze each relational operator in order to illustrate that $2 - lsirups$ include all basic relational algebra operators.

Intersection

The intersection of two relations is expressed with a rule that contains two atoms in the body, one for each relation, and with the same variables in the correspondent positions.

Example 2.4.1 Let us use the relations:

$$r(name, address, gender, birthdate)$$

$$s(name, address, gender, birthdate)$$

The intersection of those relations in datalog is obtained by the following datalog rule:

$$i(N, A, G, B) : -r(N, A, G, B), s(N, A, G, B)$$

Here, i is an IDB predicate that obtains $r \bigcap s$. □

Union

The union of two relations is obtained with two rules. Both rules should have the same IDB predicate in the head.

Example 2.4.2 The union of the relations r and s of Example 2.4.1 is:

$$\begin{aligned} u(N, A, G, B) &: -r(N, A, G, B) \\ u(N, A, G, B) &: -s(N, A, G, B) \end{aligned}$$

□

Projection

In order to obtain the projection of a relation, the IDB atom in the head of the rule contains only the variables corresponding to the desired attributes.

Example 2.4.3 We want the name and the address of the persons in relation r of Example 2.4.1:

$$p(N, A) : -r(N, A, G, B)$$

□

Selection

The selection are covered by two strategies. One is the use of built-in predicates, and the other strategy is the repetition of variables in atoms. We do not consider built-in predicates in this work, but our results are applicable also to programs with built-in predicates.

Example 2.4.4 If we want to obtain the persons from relation r of Example 2.4.1 that are male:

$$m(N, A, G, B) : -r(N, A, G, B), G = \text{``}male\text{''}$$

□

Example 2.4.5 Let us suppose the following two relations of scheme

$n(name,\ addres,\ city,\ birthplace)$
$c(city, country)$

We want to obtain persons who live in the same city where they were born:

$$p(N) : -n(N, A, C, C)$$

Notice that in order to obtain tuples that have the same value in the attributes *city* and *birthplace*, we put, in the atom in the body of the rule, the same variable in the terms that correspond to such attributes. This would be equivalent to $p(N) : -n(N, A, C, B), C = B$.

□

Cartesian product

The cartesian product can be expressed in a datalog rule. It should have two atoms in the body, one for each relation in the product. Those atoms in the body have different variables, one for each attribute of the relations. The IDB predicate in the head must have all the variables in the atoms of the body.

Example 2.4.6 The cartesian product of the relations of Example 2.4.1 is:

$$p(A, B, C, D, W, X, Y, Z) : -r(A, B, C, D), s(W, X, Y, Z)$$

□

Join

The join is very similar to the cartesian product, we only have to put the same variable(s) in the positions of the predicates that define the join.

Example 2.4.7 Using the relations of Example 2.4.5 we want the information about persons including the country where they live.

$$p(N, A, G, B, C) : -n(N, A, G, B), c(G, C)$$

Observe that in order to obtain the country where persons live, we put in both atoms (in the body of the rule) the same variable (G) in the positions of the attribute *city*.

□

2.4.2 Recursion in SQL99

In this section, we are going to focus our attention in SQL99 [MS02, UW97]. Previous SQL standards did not include recursive queries, now SQL99 includes such facility and some commercial products start to include this type of queries. In SQL99, only linear recursion is obligatory. Note that linear recursion is the type of recursion that is considered by the results of this dissertation.

SQL99 has a proposition which is introduced with the key word `WITH`. This proposition allows to define IDB relations.

`WITH` R `AS` <definition of R><query that includes R >

Example 2.4.8 Let us define an EDB relation *flights* that has the information of flights of airlines:

$$flights(airline,\ from,\ to,\ departs,\ arrives)$$

The meaning of the attributes are clear. Let us suppose that we want to compute the pairs of cities such that it is possible to flight from one to another. The query can be expressed with a $2 - lsirup$:

$reaches(X, Y) : -flights(A, X, Y, D, R)$
$reaches(X, Y) : -flights(A, X, Z, D, R), reaches(Z, Y)$

In SQL99 the query could be issued as:

```
WITH
   RECURSIVE reaches(frm, to) AS
          (SELECT frm, to FROM flights)
          UNION
          (SELECT R1.FRM, R2.TO
          FROM flights AS R1, reaches AS R2
          WHERE R1.to=R2.frm)
SELECT * FROM reaches;
```

□

In other words, it is defined a temporal relation R that is used later in a query. More generally, it is possible to define several relations after the `WITH`, separating their definitions by commas. Any of these definitions may be recursive. Several defined relations may be defined from other definition, even from themselves. Any relation that is involved in a recursion must be preceded by the key word `RECURSIVE`. Thus, a `WITH` statement has the form:

1. The key word `WITH`.
2. One or more definitions. Definitions are separated by commas and each definition consists of
   (a) An optional keyword `RECURSIVE`, which is required if the relation being defined is recursive.
   (b) The name of the relation being defined.
   (c) The keyword `AS`.
   (d) The query that defines the relation.
3. A query, which may refer to any of the prior definitions and, it is the result of the `WITH` statement.

Definitions of relations in `WITH` statements can be only used inside of such proposition.

Hence, we have illustrated that $2 - lsirups$ (among other datalog programs) can be translated to SQL99 straightforward.

2.4.3 $\mathcal{L}$: a special class of $2 - lsirups$

We define a special class of $2 - lsirups$ denoted by $\mathcal{L}$ and formed by the $2 - lsirups$ where the predicate name of the atom in the body of the non-recursive rule does not appear in the recursive rule.

Example 2.4.9 Let $P = \{r_0, r_1\}$, where:
$r_0 = p(X,Y) :- f(X,Y)$
$r_1 = p(X,Y) :- a(X,Z), a(Z,Y), e(X,X), p(Z,Y)$

P is a program in class $\mathcal{L}$.

□

2.4.4 Semantics of datalog Program Evaluation

In this dissertation we use the *proof-theoretic* interpretation of rules. That is, from a set of rules (datalog program) the "proof-theoretic meaning" of such set of rules is the set of facts derivable from the given set of facts, or database facts, using the rules in the "forward" direction only, that is, by inferring left sides (consequents or conclusions) from right sides (antecedents or hypothesis).

However, *proof-theoretic*, *model-theoretic* and *fix-point* interpretations coincide for datalog programs [Ull88, AHV95].

In the literature about the field, the evaluation method shown below is known as a naive bottom-up evaluation.

Given a program P and a database d, the *non-recursive application* of P to d, denoted by $P^1(d)$, consists of the set of facts derivable from d by applying the rules in P as follows: A fact q is in $P^1(d)$ if one of the following two cases is true:

1. q is in d.
2. There is a substitution θ and a rule r in P of the form $p_0 : -p_1, \ldots, p_n$, such that $\theta(p_i) \in d,\ 1 \leq i \leq n$ and $q = \theta(p_0)$.

Let $P^0(d) = d$ and $P^{i+1}(d) = P^1(P^i(d))$. Then $P(d)$, the output of P with input d, is $\bigcup_{i \geq 0} P^i(d)$, that is, the set of all facts that can be derived from d by repeated non-recursive applications of P. Note that $P(d)$ always contains d. It is well known that the set $P(d)$ is finite (we are dealing with finite databases, and the application of P to d does not create new terms). This means that for a given program P there exists an integer $k \geq 0$, that depends on d, that $P^k(d) = P(d)$ [CH82].

A datalog program P is *bounded* [Nau86, NS87] if there is a constant c such that for any extensional database d over $EDB(P)$ the number of non-recursive applications of P is less than c. Clearly, if a program is bounded it is essentially non-recursive, although it may appear to be recursive syntactically.

Sometimes, we just refer to the output of a rule. Let r be a rule, and let d be a database, $r(d)$ denotes the output of r with input d. Observe that a rule can be considered a program, thus $r(d)$ is equal to $P(d)$, where $P = \{r\}$.

Example 2.4.10 Let $P = \{r_0, r_1\}$, where

$r_0 : p(X, X, Y) : -e(X, Y).$
$r_1 : p(X, Y, Y) : -e(Y, Y), p(X, X, Y).$

Let us suppose that d contains the following facts:

$e(1,2)$
$e(2,4)$
$e(4,4)$

$P^1(d)$ is:

$e(1,2)$	$p(1,1,2)$
$e(2,4)$	$p(2,2,4)$
$e(4,4)$	$p(4,4,4)$

Note that in the computation of $P^1(d)$ only r_0 was used. It is the only choice, given that in order to apply r_1 it is needed the presence of p-atoms in the input database. Hence, clearly r_1 cannot be applied in $P^1(d)$.

In the computation of $P^2(d)$, r_1 is applied since $P^1(d)$ contains facts about p, thus a new fact, $p(2,4,4)$, is obtained using r_1 with $P^1(d)$ as input database. Eventually, $P^2(d)$ is:

$e(1,2)$	$p(1,1,2)$
$e(2,4)$	$p(2,2,4)$
$e(4,4)$	$p(4,4,4)$
	$p(2,4,4)$

At this point, it is clear that it is not possible to obtain any new atom, thus $P^2(d)$ is $P(d)$. □

2.4.5 Equivalence of datalog programs

Let P_1 and P_2 be programs. P_2 *contains* P_1, written $P_1 \subseteq P_2$, iff $P_1(d) \subseteq P_2(d)$ for all EDBs d. P_1 and P_2 are *equivalent*, written $P_1 \equiv P_2$, iff $P_1 \subseteq P_2$ and $P_2 \subseteq P_1$.

Let $\boldsymbol{S}$ be a set of EDBs over a given datalog scheme $\mathcal{U}$. P_2 *contains* P_1 *over* $\boldsymbol{S}$, written $P_1 \subseteq_{\boldsymbol{S}} P_2$, iff $P_1(d) \subseteq P_2(d)$ for all EDBs $d \in \boldsymbol{S}$. P_1 and P_2 are *equivalent over* $\boldsymbol{S}$, written $P_1 \equiv_{\boldsymbol{S}} P_2$, iff $P_1 \subseteq_{\boldsymbol{S}} P_2$ and $P_2 \subseteq_{\boldsymbol{S}} P_1$.

2.4.6 Rule expansion

Let P be a *lsirup*. Let r and s be two rules of P, where at least r is recursive and s may be r again. We can expand (compose or unfold) r with s, denoted by $r \circ s$, if the head of s is defined over the same predicate name as the IDB atom (q_i) in the body of r and there is a substitution ϕ from the variables in the head of s to the terms in q_i.

Let ϕ' be a new substitution constructed as follows: ϕ' contains all the pairs of ϕ and, for any non-distinguished variable V in s that also appears in r, ϕ' contains the pair V/V', where V' is a variable that does not appear in anywhere else.

Let $b_1, \ldots, b_n$ be the atoms in the body of s. Then, $r \circ s$ is r, where q_i is substituted by $\phi'(b_1), \ldots, \phi'(b_n)$.

Example 2.4.11 Given the rules:
$r_0 : p(X, Y) : -e(X, Y, Z)$
$r_1 : p(X, Y) : -e(X, X, Z), p(Z, Y)$

The IDB atom in the body of r_1 is $p(Z, Y)$ and the head of the rule r_0 is $p(X, Y)$. Taking $\phi = \{X/Z\}$ we have that $p(Z, Y) = \phi(p(X, Y))$. Then, $r_1 \circ r_0$ is: $p(X, Y) : -e(X, X, Z), e(Z, Y, Z^1)$.

Observe that Z is a non-distinguished variable in r_0 that also appears in r_1. Thus, ϕ' sends it to a new and distinct variable Z^1. □

r^k denotes the composition (self-unfolding) of r with itself k times.

Example 2.4.12 Let $P = \{r_0, r_1\}$, where:

$r_0 = p(X, Y) :- e(X, Y)$
$r_1 = p(X, Y) :- e(X, Z), e(X, Y), p(Z, Y)$

The expansion of the rule r_1 with r_0 (denoted by $r_1 \circ r_0$) is:

$r_1 \circ r_0 = p(X, Y) :- e(X, Z), e(X, Y), e(Z, Y)$

In the same way, we can build:

$r_1 \circ r_1 \circ r_0 = r_1^2 \circ r_0 = p(X,Y) :- e(X,Z), e(X,Y), e(Z,Z'), e(Z,Y), e(Z',Y)$
$r_1^3 \circ r_0 = p(X,Y) :- e(X,Z), e(X,Y), e(Z,Z'), e(Z,Y), e(Z',Z''), e(Z',Y), e(Z'',Y)$ □

2.4.7 Projection Operator

We use the symbol, π, like the projection operator of the relational algebra. The output of $\pi_{i_1,\ldots,i_k}[p, P(d)]$ is the projection over the columns $i_1, \ldots, i_k$ of the p-facts of $P(d)$.

Example 2.4.13 Using the database $P(d)$ of Example 2.4.10, $\pi_{1,2}[p, P(d)]$ is:

$p(1,1)$
$p(2,2)$
$p(4,4)$
$p(2,4)$

□

2.5 Functional Dependencies

In this section we give a formal definition of fds and the notation we shall use to represent them.

Let p be a predicate of arity t. Let n be the set $\{x_1, \ldots, x_k\}$ and m be the set $\{y_1, \ldots, y_l\}$, where the x_i's and the y_j's are integers in the range from 1 to t. A *functional dependency (fd) over* p is a statement of the form $p : \{n\} \rightarrow \{m\}$, read as "$n$ functionally determines m over p."

Let $f = p : \{n\} \rightarrow \{m\}$. Then, we say that p is *the predicate of* f, n is the *left-hand side of* f, and that m is the *right-hand side of* f. A database d *satisfies* the fd $p : \{n\} \rightarrow \{m\}$ if for every two facts μ and ν about p in d such that $\mu[n] = \nu[n]$, it is also true that $\mu[m] = \nu[m]$. If d does not satisfy $p : \{n\} \rightarrow \{m\}$, then we say that it *violates* that dependency.

We denote with $a_k[L_j]$ the set of variables of the atom a_k that are placed in the positions defined by the left-hand side of a fd f_j. When by the context it is clear which fd is used, we only denote $a_k[L]$.

In the same way, we denote with $a_k[R_j]$ the set of variables of the atom a_k that are placed in the positions defined by the right-hand side of a fd f_j. When by the context it is clear which fd is used, we only denote $a_k[R]$.

Example 2.5.1
Let us suppose that the predicate *manages*(*Manager*, *Employee*) asserts that a manager, denoted by *Manager*, manages the employee denoted by *Employee*. The instance of the relation associated with *manages* shown

in Table 2.1 satisfies the fd *manages* : $\{2\} \rightarrow \{1\}$, but it does not satisfy *manages* : $\{1\} \rightarrow \{2\}$. □

MANAGES	
MANAGER	EMPLOYEE
ana	raquel
ana	pedro
ana	gustavo
pedro	jorge
pedro	ramiro
gustavo	juan
juan	pablo
juan	agustin

Table 2.1: The relation $MANAGES$

Let F be a set of fds. A database d satisfies F if d satisfies every fd in F. Let Q be a set of predicates. F is *defined over* Q if all fds in F are over some predicate in Q.

$SAT(F)$ represents the set of all databases over a given datalog schema $\mathcal{U}$ that satisfy F.

Let P be a program and let F and G be sets of fds over $pred(P)$. Consider a fd g defined over p, a predicate in $pred(P)$. Then we say that g is *implied by F (in P)* [AH88], denoted $F \models_P g$, if $P(d)$ satisfies g, for all d in $SAT(F)$ such that d is defined over $EDB(P)$. We say that F *implies G (in P)*, denoted by $F \models_P G$, if $F \models_P g$ for every fd g in G.

If we allow d to contain tuples about predicates in $IDB(P)$, in addition to contain tuples about predicates in $EDB(P)$, then we say that g is *uniformly implied by F (in P)* [AH88], and we denote it by $F \models_P^u g$, if for all d in $SAT(F)$, $P(d)$ satisfies g. We say that F *uniformly implies G (in P)*, denoted by $F \models_P^u G$, if $F \models_P^u g$ for every fd g in G.

Let $f = p : \{n\} \rightarrow \{m\}$. Then, f is *left-hand (side) minimal with respect to F* if for all proper subsets h of n, $F \not\models_P p : \{h\} \rightarrow \{m\}$. *$f$ is minimal with respect to F* if the following conditions are all true: f is left-hand minimal with respect to F; m is a singleton set; and $F \models_P f$.

P preserves F if $F \models_P F$. *P unifomly preserves F* if $F \models_P^u F$.

We shall consider the consequents of all fds as singleton sets. By Armstrong's axioms [Arm74], such assumption does not restrict our results.

2.6 Expansion Trees

An *expansion tree* is a description for the derivation of an intensional fact by the application of some rules to extensional facts and the set of intensional facts generated earlier. The leaves of a tree are EDB atoms. All non-leaf atoms are IDB literals. Every non-leaf atom represents an application of a rule, whose head is the non-leaf atom, and the children of such atom are the atoms in the body of the rule.

For the sake of simplicity, from now on, we shall refer to expansion trees simply as trees.

Let r be the rule $q :- q_1, q_2, \ldots, q_k$. Then, a tree T can be built from r as follows: the node at the root of T is q and q has k children, q_i, $1 \leq i \leq k$. We denote this tree as $tree(r)$.

Example 2.6.1 Let $P = \{r_0, r_1\}$, where:
$r_0 = p(X,Y) :- e(X,Y)$
$r_1 = p(X,Y) :- a(X,Z), e(Z,Y), a(X,X), p(Z,Y)$

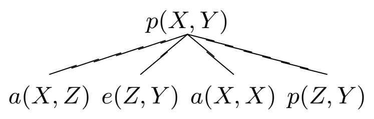


Figure 2.1: $tree(r_1)$

□

Let S and T be two trees. Then, *S and T are isomorphic*, if there are two substitutions θ and α such that $S = \theta(T)$ and $T = \alpha(S)$.

The variables appearing in the root of a tree T are called *the distinguished variables of T*. All other variables appearing in nodes of T that are different from the distinguished variables of T are called *the non-distinguished variables of T*.

The previous definition of a tree only considers expansion trees built from a rule. However, an expansion tree may have different levels coming from successive composition (or applications) of rules.

Let S and T be two trees. Assume that exactly one of the leaves of S is an IDB atom[a], denoted by p_s. The *expansion (composition)* of S with T,

[a]That is the case of the trees generated by $2 - lsirups$, since in the recursive rule of such programs, there is only one IDB predicate.

denoted by $S \circ T$ is defined if there is a substitution θ, from the variables in the head of T (h_t) to the terms in p_s, such that $\theta(h_t) = p_s$. Then, $S \circ T$ is obtained as follows: build a new tree, isomorphic to T, say T', such that T' and T have the same distinguished variables, but all the non-distinguished variables of T' are different from all of those in S. Then, substitute the atom p_s in the last level of S by the tree $\theta(T')$.

From now on, we use the expression $tree(r_j \circ r_i)$ to denote $tree(r_j) \circ tree(r_i)$.

A *level of a tree* is formed by all the atoms that have the same parent atom.

Let T be a tree. *The level of a node in* T is defined as follows: the root of T is at level 0, the level of a node n of T is one plus the level of its parent node. *Level* j of T is the set of atoms of T with level j.

2.6.1 TopMost and frontier of a tree

Two rules that can be extracted from a tree. Let T be a tree:

- *the frontier of* T (also known as *resultant*), denoted by $frontier(T)$, is the rule $h :- b$, where h is the root of T and b is the set of the leaves of T.
- *the topMost of* T, denoted bt $topMost(T)$, returns the rule $h :- b$, where h is the root of T and b is the set of the atoms that are the children of the root.

Example 2.6.2 Using the tree $T_2 = tree(r_1^2 \circ r_0)$ in Figure 2.2 we have:

$$frontier(T_2) : p(X,Y) : -a(X,Z), e(Z,Y), a(X,X), a(Z,Z'), e(Z',Y), a(Z,Z), e(Z',Y)$$

$$topMost(T_2) : p(X,Y) : -a(X,Z), e(Z,Y), a(X,X), p(Z,Y)$$

□

2.6.2 The set $trees(P)$

From a $2{-}lsirup$ P an infinite number of trees $T_0, T_1, T_2, \ldots$ can be obtained. Each T_i denotes $tree(r_1^i \circ r_0)$. We call $trees(P)$ the infinite ordered set of trees generated by the procedure *expandTree* (see Figure 2.3):

Example 2.6.3 Let $P = \{r_0, r_1\}$ be:

r_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
r_1: $p(X,Y,Z,A,B,C) :- e(Y,X,Y,C,A,D), p(Z,X,Y,B,C,D)$

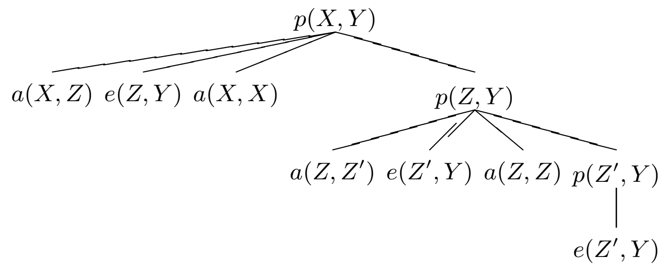


Figure 2.2: T_2

```
Procedure expandTree(P)
Input: A 2-lsirup $P = \{r_0, r_1\}$, where $r_1$ is the recursive rule.
Output: An infinite ordered set of trees.
Assumptions: $r_1 \circ r_1$ is defined.

Let $L$ be the empty ordered set of trees;
append $tree(r_0)$ to $L$;
Let $T = tree(r_1)$;
while TRUE
        append $T \circ tree(r_0)$ to $L$;
        Let $T = T \circ tree(r_1)$;
```

Figure 2.3: Generating an infinite ordered set of trees

T_2 is obtained as follows: $tree(r_1)$ is composed with itself and then, the resulting tree ($tree(r_1^2)$) is composed with $tree(r_0)$ obtaining, finally, $tree(r_1^2 \circ r_0)$.

$T_2 = tree(r_1^2 \circ r_0)$ (built from P) is shown in Figure 2.4.

□

The following conventions are used when we refer to trees:

- When we use the expression *a tree T_j is bigger or higher than a tree T_k*, it means that $j > k$.
- When we use the expression *a tree T_j is shorter or smaller than a tree T_k*, it means that $j < k$.
- When we use the expression *a tree T_j has more levels than a tree T_k*,

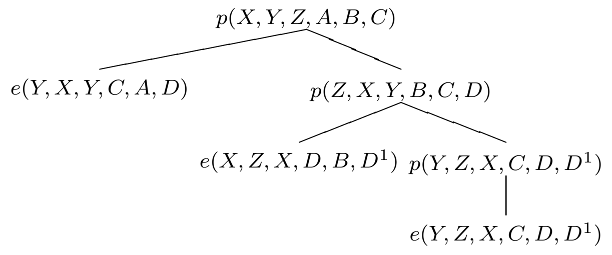


Figure 2.4: T_2

it means that $j > k$.

- When we use the expression *a tree T_j has less or fewer levels than a tree T_k*, it means that $j < k$.
- We say that two levels i and k (in T_j) are *separated by w levels* if $i = k + w$ or $k = i + w$, and $i \leq j$ and $k \leq j$.

Note that in all the trees in $trees(P)$ where P is a $2 - lsirup$, any level is isomorphic to any other level in the tree, except the last level and level 0. This is true given that all the levels excepting the last one are obtained applying the recursive rule. We can extend this affirmation as in the following lemma.

Lemma 2.6.1 Let P be a $2 - lsirup$. Let T_i be a tree in $trees(P)$ and let T_{sub} be the tree formed by the last j levels of T_i ($j \leq i$) and rooted by the IDB atom in level $j - 1$. Then, T_j and T_{sub} are isomorphic.

Proof It follows from the definition of trees and given that there is only one recursive rule and only one non-recursive rule. □

Example 2.6.4 Let $P = \{r_0, r_1\}$ be:

r_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
r_1: $p(X,Y,Z,A,B,C) :- e(Y,X,Y,C,A,D), p(Z,X,Y,B,C,D)$

In Figure 2.5(a), we can see the tree T_1 constructed with P. In Figure 2.5(b), we can see T_{sub}, the tree formed by the last two levels of T_2 (in Figure 2.4) and rooted with the IDB atom in level 1 (of T_2).

It is easy to see that these trees are isomorphic. Then, applying a substitution to one of them, they became equal. For example, let us consider the substitution:

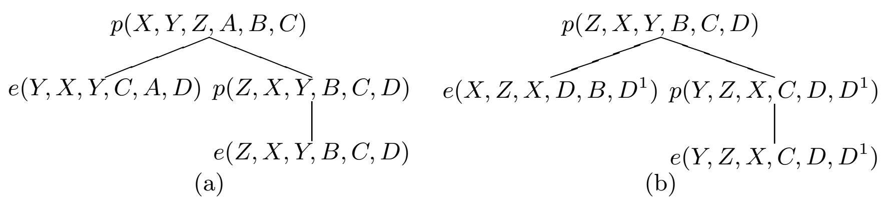


Figure 2.5: T_1 and T_{sub}

$$\theta = \{Z/X,\ X/Y,\ Y/Z,\ B/A,\ C/B,\ D/C,\ D^1/D\}$$

$\theta(T_{sub})$ is equal to T_1. □

2.6.3 Relative position and columns

An atom in a tree can be represented as $a_{i,j}$. In such case, $a_{i,j}$ represents an atom that is in level i and in the $j-th$ *relative position* inside of level i. The *relative position* of an atom q_k in level l of T_w is, intuitively, the position of such atom in level l, counting atoms from left to right and beginning at 1, except in the case of levels 0 and $w+1$, as it can be seen in the formal definition given below.

- *If* $0 < l \leq w$, the relative position of q_k in level l is the position of q_k in level l counting the atoms in level l from left to right and beginning at 1.
- *If l is 0 or* $w+1$, that is, if q_k is either the root or the atom in the last level of T_w, the relative position of q_k in level l is the number of atoms in the body of the recursive rule used to build the tree.

Example 2.6.5 In the tree of Figure 2.6, $p(Y,Z,X,C,D,D^1)$ is denoted by $a_{2,2}$, that is, $p(Y,Z,X,C,D,D^1)$ is in level 2 and in relative position 2. Note that the atoms in the root of the tree and in the last level are in relative position 2 given that there are two atoms in r_1. Thus, $p(X,Y,Z,A,B,C)$ is denoted by $a_{0,2}$ and $e(Y,Z,X,C,D,D^1)$ is denoted by $a_{3,2}$.

□

Similarly, for a variable in an atom of a tree, we define its *column* as its position, counting variables of its level, from left to right and beginning at 1. Again, variables in level 0 and in the last level of the tree represent the exceptions.

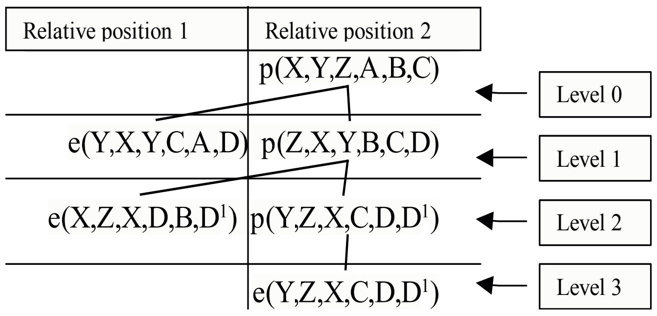


Figure 2.6: T_2

More precisely, we define the column of a variable in a tree as follows. Let X be a variable in th n-th position of $a_{i,j}$, an atom a tree T. The *column* of X (in the n-th position of $a_{i,j}$) is computed as follows: if the recursive rule is $r_1 = a_0(\overrightarrow{X_0}) : -a_1(\overrightarrow{X_1}), a_2(\overrightarrow{X_2}), \ldots, a_n(\overrightarrow{X_n})$, then the column of X is the arity of a_1, plus the arity of a_2 ... plus the arity of a_{j-1}, plus n.

Example 2.6.6 In the tree of Figure 2.7, $a_{2,2}[2] = Y$ is in level 2 and in column 4.

Observe that $a_{3,4}$ $(e(Z', Y))$ is an atom in the last level. Therefore, $a_{3,4}[1]$ (Z') is in the 7-th column, since the addition of the arities of the EDB atoms of r_1 is six and Z' is in the first position of $a_{3,4}$. In the same way, $a_{3,4}[2]$ (Y) is in the 8-th column, obtained adding six to the second position of Y in $a_{3,4}$.

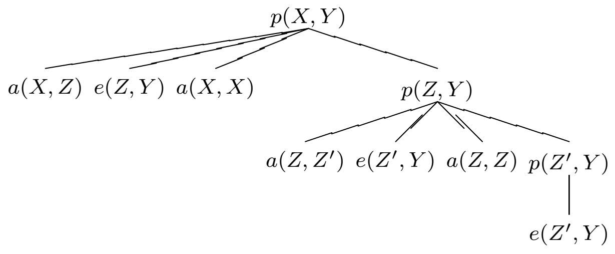


Figure 2.7: T_2

□

2.6.4 Expansion Graph G of a $2-lsirup$

In this section we introduce a graph that is used to capture how the variables appear and disappear through the levels of a tree.

Let P be a $2-lsirup$. Let p_h and p_b be the IDB atoms in the head and in the body of r_1, the recursive rule of P. The *Expansion Graph of a program* P is generated with this algorithm.

1. If the arity of the IDB predicate in P is k, add k nodes named $1,\dots,k$.
2. Add one arc from the node n to the node m, if a variable X is placed in the position n of p_h, and X is placed in the position m of p_b.
3. Add one arc from the node n without target node, if a variable X is placed in the position n of p_h, and it does not appear in p_b.
4. Add one arc without source node and target node m, if a variable X is placed in the position m of p_b and it does not appear in p_h.

Example 2.6.7 Let $P=\{r_0,r_1\}$ where r_1 contains the following IDB atoms:

$p(A,B,C,D,E,F,G,H,I,J,K,L,M):-\dots p(B,A,E,C,D,F,W,G,G,X,J,L,L)$

In Figure 2.8, we can see the expansion graph of P. □

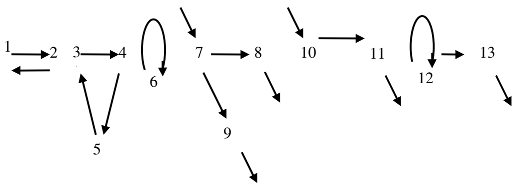


Figure 2.8: Expansion Graph of P

Claim 2.6.1 Let P be a $2-lsirup$. In the expansion graph G of P, a chunk *is a subgroup of G that is connected. That is, there is at least one node such that all the other nodes in the chunk can be reached from it.* □

Example 2.6.8 The graph in Figure 2.8 has 6 chunks.

Definition 2.6.1 The length of a chunk *is the maximum number of nodes (of the chunk) that can be visited using only once the arcs that form the chunk. Each node must be counted once, even when the chunk is a cycle.* □

Example 2.6.9 The lengths of the six chunks in the Figure 2.8 are respectively: 2, 3, 1, 2, 2, 2. □

2.6.5 Variable types in trees

Let P be a $2 - lsirup$. In a tree T, where T is in $trees(P)$, we define two types of variables:

- Variables in the head of r_1 that correspond (in the expansion graph) to nodes that belong to cyclic chunks are called *cyclic variables* (CV's). Since these variables correspond to cyclic chunks, then cyclic variables appear in all levels of any tree in $trees(P)$.

- Variables that are not cyclic variables are called *acyclic variables* (AC's). Acyclic variables correspond to acyclic chunks, and thus acyclic variables do not appear in all levels bigger than a certain level[b] (that depends on the program used to built the tree).

Example 2.6.10 Let $P = \{r_0, r_1\}$ be:

$$r_0 : p(A, B, C) : -e(X, Y, B, C, A)$$
$$r_1 : p(A, B, C) : -e(M, N, A, H, I), a(B, C, M, I), p(B, A, H)$$

In Figure 2.9, we show $T_3 = r_1^3 \circ r_0$ with its levels and columns.

Columns	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Level 0										p(A,	B,	C)		
Level 1	e(M,	N,	A,	H,	I)	a(B,	C,	M,	I)	p(B,	A,	H)		
Level 2	e(M^1,	N^1,	B,	H^1,	I^1)	a(A,	H,	M^1,	I^1)	p(A,	B,	H^1)		
Level 3	e(M^2,	N^2,	A,	H^2,	I^2)	a(B,	H^1,	M^2,	I^2)	p(B,	A,	H^2)		
Level 4										e(X,	Y,	A,	H^2	B)

Figure 2.9: T_3

If we inspect the expansion graph of P in Figure 2.10, we can see that the variables in positions 1 and 2 of the head of the tree (A and B) correspond to positions 1 and 2 of the expansion graph. These positions conform a

[b]Obviously if the tree has enough levels.

cyclic chunk, thus A and B are cyclic variables. The other variable in the head of the tree (C) corresponds to the position 3 in the expansion graph. Such position is in an acyclic chunk, thus C is an acyclic variable. The rest of variables are also acyclic variables.

Note that C does not appear in levels bigger than one, since it is an acyclic variable. On the other hand, since A and B are cyclic variables, they appear in all levels of the tree.

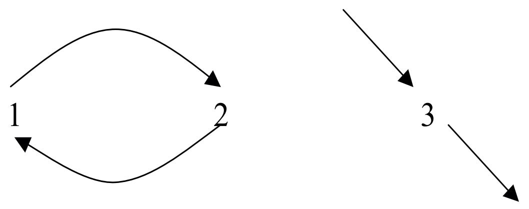


Figure 2.10: Expansion graph of P

□

2.6.6 The number $\mathcal{N}$

$\mathcal{N}$ is a number defined from the expansion graph of a $2-lsirup$. Basically, it establishes rules that variables (both cyclic and acyclic) in any tree built from a $2-lsirup$, from where $\mathcal{N}$ was computed, must satisfy.

Definition 2.6.2 Let G be the expansion graph of a $2-lsirup$ *P, then* $\mathcal{N}$ *is the least common multiple of the chunk lengths of the chunks in G.*

Example 2.6.11 The graph in Figure 2.8 has $\mathcal{N} = 6$ $(6 = \textit{least common multiplier of } 2,3,1,2,2,2)$.

In the next two subsections, we introduce some properties based on $\mathcal{N}$.

Properties of $\mathcal{N}$ and cyclic variables

Let P be a $2-lsirup$, let T_i be a tree in $trees(P)$. Levels of T_i separated by $\mathcal{N}$ levels have the cyclic variables in the same columns (excepting in the last level and level 0 of the tree). That is, for any cyclic variable in $a_{k,l}[n]$, where $a_{k,l}$ is an atom in T_i, by construction of $\mathcal{N}$, for all integer c such that $a_{k+c\mathcal{N},l}$ is an atom in T_i, it is true that $a_{k+c\mathcal{N},l}[n]$ (if $a_{k+c\mathcal{N},l}[n]$ exists in T_i) is the same variable as $a_{k,l}[n]$.

Lemma 2.6.2 Let P be a $2-lsirup$. Let T_i be a tree in $trees(P)$. If there is a cyclic variable V in $a_{k,l}[n]$, the variable in $a_{k+c\mathcal{N},l}[n]$ (if $a_{k+c\mathcal{N},l}[n]$ exists in T_i), where c is an integer, is V as well.

Proof If follows from the definition of CV's and the definition of $\mathcal{N}$. □

Example 2.6.12 Let $P = \{r_0, r_1\}$, where:

$$r_0 = p(X,Y) :- e(X,Y)$$
$$r_1 = p(X,Y) :- e(Z,X), e(X,Y), p(Y,X)$$

Using P, T_3 is:

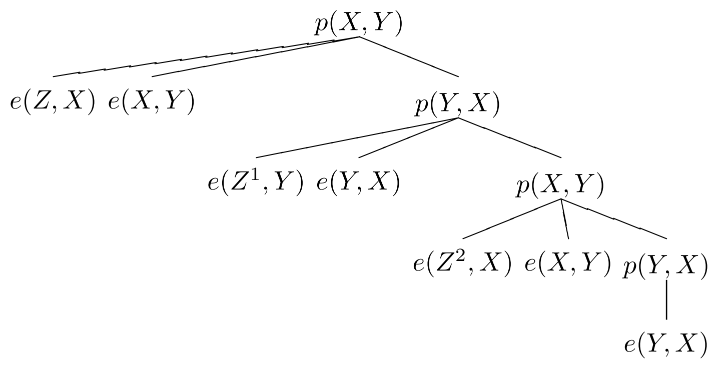


Figure 2.11: T_3

In this example, $\mathcal{N} = 2$.

In the tree of Figure 2.11, X and Y are the CV's. Thus, we can see that if $a_{1,1}[2] = X$, then $a_{1+1\mathcal{N},1}[2]$ is X as well. In the same way, $a_{1,2}[1,2] = a_{1+1\mathcal{N},2}[1,2]$.

□

We can extend the properties of the number $\mathcal{N}$ to two trees such that one has more levels than the other and the difference of levels between the two trees is a multiple of $\mathcal{N}$.

Lemma 2.6.3 Let P be a $2-lsirup$. Let T_i and T_j be two trees in $trees(P)$ such that $j = c\mathcal{N} + i$ and c is a positive integer. For all cyclic variable V in $a_{k,l}[n]$, where $a_{k,l}$ is an atom in T_i, then in T_j, $a_{k+c\mathcal{N},l}[n]$ is V as well.

Proof $a_{k,l}$ exists also in T_j given that $j > i$. Therefore, $a_{k,l}$ is the same atom in T_i and T_j. Thus, by Lemma 2.6.2, in T_j, for all cyclic variable V which is in the $n-th$ position of $a_{k,l}$ (i.e., $a_{k,l}[n] = V$) then $a_{k+c\mathcal{N},l}[n]$ is V as well. □

Example 2.6.13 Let $P = \{r_0, r_1\}$ be:

r_0: $p(X, Y, Z, A, B, C) :- e(X, Y, Z, A, B, C)$
r_1: $p(X, Y, Z, A, B, C) :- e(Y, X, Y, C, A, D), p(Z, X, Y, B, C, D)$

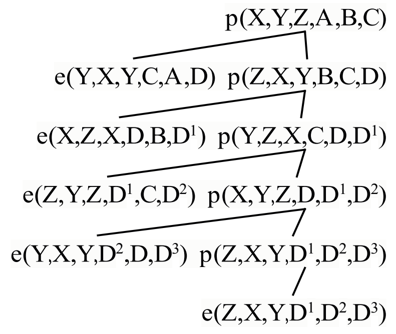

p(X,Y,Z,A,B,C)
e(Y,X,Y,C,A,D) p(Z,X,Y,B,C,D)
e(X,Z,X,D,B,D¹) p(Y,Z,X,C,D,D¹)
e(Z,Y,Z,D¹,C,D²) p(X,Y,Z,D,D¹,D²)
e(Y,X,Y,D²,D,D³) p(Z,X,Y,D¹,D²,D³)
e(Z,X,Y,D¹,D²,D³)

Figure 2.12: T_4

In Figure 2.12 is shown T_4, and in Figure 2.5(a) is shown T_1, both built with P. $\mathcal{N}$ for P is 3, thus, in this case, 4 is $1 + 1\mathcal{N}$.

Let us check the atom $a_{1,1}$ of T_1 ($e(Y, X, Y, C, A, D)$). In T_4, $a_{1,1}$ is exactly the same atom, thus any CV is in the same positions in both atoms.

Moreover, the atom $a_{4,1}$ of T_4 ($e(Y, X, Y, D^2, D, D^3)$) is $\mathcal{N}$ levels downwards from $a_{1,1}$, therefore all the CV's (X and Y) are in the same positions in $a_{1,1} = e(Y, X, Y, C, A, D)$ and $a_{1+1\mathcal{N},1} = e(Y, X, Y, D^2, D, D^3)$. □

Properties of $\mathcal{N}$ and acyclic variables

Until now, we have studied the relationship of $\mathcal{N}$ with cyclic variables, however $\mathcal{N}$ is not only associated with cyclic variables. $\mathcal{N}$ has also properties related with acyclic variables.

Lemma 2.6.4 Let P be a $2 - lsirup$, T_i be a tree in $trees(P)$ and $a_{l,m}[n]$ be an acyclic variable. Then, $a_{l,m}[n]$ cannot appear in levels smaller than $l - \mathcal{N}$ and bigger than $l + \mathcal{N}$.

Proof It follows from the definition of $\mathcal{N}$. □

In other words, two appearances of an acyclic variable in a tree cannot be separated by more than $\mathcal{N}$ levels. Therefore, if a variable X is in two atoms (say q_i and q_k) separated by more than $\mathcal{N}$ levels, then according to the definition of $\mathcal{N}$ (and the definition of chunk length), X is a cyclic variable, and then it must appear in the same columns each $\mathcal{N}$ levels.

Example 2.6.14 The tree of Figure 2.12 was built with the program P:

$$r_0\text{:}\ \ p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$$
$$r_1\text{:}\ \ p(X,Y,Z,A,B,C) :- e(Y,X,Y,C,A,D), p(Z,X,Y,B,C,D)$$

$\mathcal{N}$, for P, is 3. The CV's of the tree of the Figure 2.12 are X, Y and Z. Let us inspect any other variable, for example D. D appears in level 1, 2 and 3. However, in level 4 D does not appear. If D would appear in a level bigger than 3, it would be a cyclic variable. Thus it is clear that D is an acyclic variable. □

Chapter 3

Chase of rules and trees

The term “chase” appears for the first time in the lossless-join test of Aho, Beeri, and Ullman [ABU79, Ull88]. Right after that, the chase began to be used to solve problems different from its original motivation (i.e. determining if a certain decomposition is a lossless-join decomposition). For example, Maier, Mendelzon and Sagiv [MMS79] use the chase in order to discover (data) dependencies and Beeri and Vardi [BV84] extend the algorithm of Maier, Mendelzon and Sagiv to generalized dependencies.

Reviewing the relational model, the chase is used for different purposes (see [Mai83],[AHV95]): to optimize tableau queries (that can be generalized to conjunctive queries), to characterize equivalence of conjunctive queries with respect to a set of dependencies and to determine logical implication between sets of dependencies.

Moreover, its applications have even crossed the boundaries of the relational model, and there are applications of the chase in almost any major database model.

In datalog, the chase has been used in different areas: Lerat [Ler86] applies the chase to resolve null values in databases with incomplete information; Sagiv [Sag87] used the chase to test uniform containment of datalog programs; Torlone and Atzeni [TA91] use a variation of the chase with resolution to take into account fds that are defined on extensional database predicates, for consistency checks and for resolving ambiguities; Wang and Yuan use the chase to solve the uniform implication problem [WY92] whereas Hernández et al. use the chase to solve the implication problem [HPB97b, HP97, HPB97a]; Lakshmanan and Hernández [LH91] use the chase as part of a procedure to “factor-out” goals from a class of linear sirups under the presence of fds.

The chase as a tool to optimize queries in the framework of datalog is also used by several researchers [LH91, GT95, Tan97, BGTHP98a, BGTHP98b, PBPH00, BHPP01].

Recent data models have also adopted the chase to optimize queries. Papakonstantinou and Vassalos [PV99] use the chase as a rewriting technique for optimizing semistructured queries. Popa et. al. [Pop00, PDST00] use the chase to optimize queries in the framework of the object/relational model.

All these works reveal the chase as a very powerful multipurpose tool, but why is the chase useful for such a variety of problems? The reason should be found in the effect of the chase over databases, programs or rules. Basically, the chase may equate symbols or add tuples using the information provided by constraints. Such equalizations and new tuples may be used to resolve ambiguities, to remove null values, or in many cases to equate variables that may lead to remove atoms or other purposes.

The outline for this chapter is as follows. In Section 3.1, we give the definition of the chase of a rule and some related results. In Section 3.2, we give the definition of the chase of a tree and some related notation and results. In Section 3.3, we define a special type of chase of trees called partial chase. Finally, in Section 3.4 we introduce the cyclic topMost.

3.1 The chase of a datalog rule

The chase [Mai83, AHV95] is a general technique that is defined as a nondeterministic procedure based on the successive application of dependencies (or generalized dependencies) to a set of tuples (that can be generalized to atoms).

Although the chase has many applications, we apply the chase in order to optimize datalog rules. Originally, the chase takes advantage of functional dependencies and join dependencies, however, some authors have extended the chase to other types of constraints [Sag87, Pop00, PDST00].

In our work, we use the chase using functional dependencies that are semantic constraints representing equalities that must be held in databases [Ull88]. Basically, the idea behind the chase (as it is used in our work) is that when a rule r is evaluated over a database d that satisfies a set of fds F, the substitutions that map the variables in atoms of r to the constants in the facts of d can map different variables to the same constant, since in the database there is less variability (due to the fds) than in the rule. The chase, in order to optimize the rule, pushes these equalities into the variables of the rule.

Even though the chase was first applied to optimize conjunctive queries [AHV95], Gonzalez Tuchmann, to the best of our knowledge, coined the term *chase of a datalog rule* [GT95].

Though the chase of datalog rules it is not a new definition of our work, we offer in this section a deep analysis of such procedure, since it is not a standard procedure and it is very important for this dissertation.

Consider the following:

- Let F be a set of fds defined over $EDB(P)$, for some program P.
- Let r be a rule of P.
- Let $f = p : \{n\} \rightarrow \{m\}$ be a fd in F.
- Let q_1 and q_2 be two atoms in the body of r such that the predicate name of q_1 and q_2 is p, $q_1[n] = q_2[n]$ and $q_1[m] \neq q_2[m]$.

Note that $q_1[m]$ and $q_2[m]$ are variables since we are assuming that programs do not contain constants. An *application of the fd f to r* is the uniform replacement in r of $q_1[m]$ by $q_2[m]$ or vice versa.

Definition 3.1.1 [GT95] Let r be a rule and let F be a set of fds over the predicates in the body of r. The chase of r with respect to F*, denoted by $Chase_F(r)$, is the rule obtained by applying every fd in F to the atoms in the body of r until no more changes can be made.*

□

Example 3.1.1 Let r be the rule $p(X, Y) :- e(X, Z), e(X, Y), e(Z, Y)$. Let F be the set of fds $F = \{f = e : \{1\} \rightarrow \{2\}\}$.

The $Chase_F(r)$ is a new rule $r' = p(X, Y) :- e(X, Y), e(Y, Y)$ obtained after the equalization of Z with Y in the atoms of the original rule r. This equalization is due to the existence of the atoms, $e(X, Z)$ and $e(X, Y)$, which have the same variable in the position defined by the left-hand side of the fd $e : \{1\} \rightarrow \{2\}$. Thus, variables in the position defined by the right-hand side of the fd (Z and Y) are equated. Note that r and r' produce the same output when they are evaluated over databases in $SAT(F)$. □

From now on, we may use the terms chase of a datalog rule and chase of a rule indistinctly.

Because the chase of rules does not introduce new variables, it turns out that the chase procedure always terminates. Applying a fd to a rule r can be performed within time polynomial in the size of r [Mai83, AHV95], thus the time of its computation is tractable.

Note that the chase of a rule equates some of its variables, thus the chase defines a substitution where in each pair both members are variables in the rule. That is, there is a substitution ϕ such that $\phi(r) = Chase_F(r)$.

Let r be a rule and F be a set of fds. If we introduce an order in the variables of r (let say $X < Y < Z, \ldots$) and when the chase equates (for example) X and Z, then Z is replaced by X (the variable closer to the end is replaced by the variable closer to the begin). Then the $Chase_F(r)$ always produce the same rule independently of the order of the fd applications, that is, the chase of a rule is Church-Rosser [Mai83, AHV95].

The following lemma proves that $r' \equiv_{SAT(F)} r$, that is, r' is equivalent to r when both are applied over databases satisfying the set of fds F.

Lemma 3.1.1 Let r be a datalog rule, F a set of fds over $EDB(r)$ and r' the $Chase_F(r)$. Then, $r' \equiv_{SAT(F)} r$.

Proof Let d be a database in $SAT(F)$. We have to prove that for any atom q in $r(d)$, q must be in $r'(d)$ as well, and if $q' \in r'(d)$ then q' has to be in $r(d)$. First, we are going to prove that if q is in $r(d)$, then q must be in $r'(d)$.

In this proof, we are going to consider only one equalization of variables due to a fd application during the chase. The extension to several fd applications (and thus equalizations) is straightforward.

Let $f = p : \{n\} \rightarrow \{m\}$ be a fd in F and let $b_1, \ldots, b_k$ be the atoms in the body of r. Let b_i and b_j be two atoms in the body of r such that the predicate name of b_i and b_j is p, $b_i[n] = b_j[n]$ and, $b_i[m] \neq b_j[m]$.

If q is in $r(d)$ then there is a substitution θ such that $\theta(b_1), \ldots, \theta(b_n)$ are in d, and $\theta(h_r) = q$, where h_r is the head of r. Therefore, if $\theta(b_i) = g_1$ and $\theta(b_j) = g_2$, then g_1 and g_2 are in d. Note that $\theta(b_i[n]) = \theta(b_j[n])$, since $b_i[n] = b_j[n]$, thus $g_1[n] = g_2[n]$. Then, since d is in $SAT(F)$, $g_1[m] = g_2[m]$ (see Figure 3.1).

Let b'_i and b'_j be b_i and b_j after the chase of r (r'), that is, b'_i and b'_j are in the body of r'. In r', due to the chase, $b_i[m]$ is equated to $b_j[m]$ or vice versa, let say that $b_i[m]$ is equated to $b_j[m]$. Therefore, if the substitution defined by the chase is ϕ, that is, $r' = \phi(r)$, then ϕ contains the pair $b_i[m]/b_j[m]$.

Let θ' be the substitution $\theta' = \phi(\theta)$. Note that for each pair in θ that includes b_i ($b_i[m]/g_x[m]$), in θ' there is a pair exactly the same but with b_i replaced by b_j ($b_j[m]/g_x[m]$). It is easy to see that $\theta'(b'_i) = g_1$ and $\theta'(b'_j) = g_2$, since in θ, $b_i[m]$ and $b_j[m]$ are mapped to the same constant ($g_1[m] = g_2[m]$)

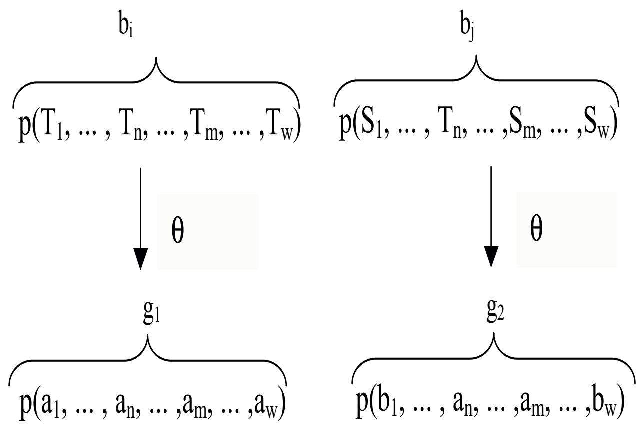

b_i
b_j
p(T_1, ... , T_n, ... ,T_m, ... ,T_w)
p(S_1, ... , T_n, ... ,S_m, ... ,S_w)
θ
θ
g_1
g_2
p(a_1, ... , a_n, ... ,a_m, ... ,a_w)
p(b_1, ... , a_n, ... ,a_m, ... ,b_w)

Figure 3.1: Regular evaluation

whereas in θ', there is only one variable ($b_j[m]$) that is mapped to the same constant. Hence, it is easy to see that if $\theta(h_r)$ produces q, then $\theta'(h'_r)$ (where h'_r is h_r after the chase) produces q and, $\theta(b'_1), \ldots, \theta(b'_n)$ (where $b'_1, \ldots, b'_n$ are $b_1, \ldots, b_n$ after the chase) are in d (see Figure 3.2).

The other side, that is, if q' is in $r'(d)$ then q' is in $r(d)$, is trivial since r' is r with some of its variables equated. □

Notice that even if not all the possible equalizations that the chase of a rule (r) may produce were applied to r, the resulting rule (say r^t) is equivalent to r when both are applied to databases satisfying the set of fds used to perform the equalizations. That is, let r^t be the rule r after $t \in \mathbb{N}$ fd applications using a set of fds F. That is, r^t may not include all the possible fd applications that the chase may produce. r^t is still equivalent to r when both are applied over databases satisfying the set of fds F used to produce the fd applications applied over r^t.

3.1.1 Benefits of the chase of rules

Although, it is easy to see that the equalizations of variables lead to cheaper evaluations, such benefits are discussed in this section.

Let us consider a datalog rule (without negation), an evaluation of such

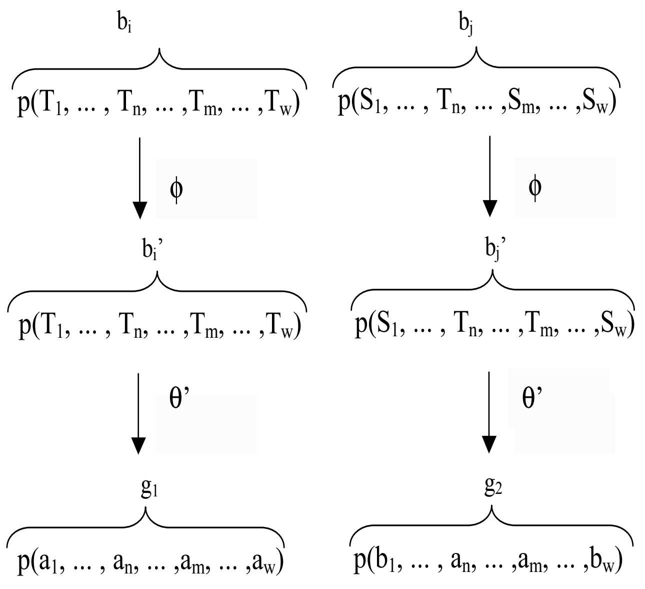

b_i
p(T1, ... , Tn, ... ,Tm, ... ,Tw)
φ
b_i'
p(T1, ... , Tn, ... ,Tm, ... ,Tw)
θ'
g1
p(a1, ... , an, ... ,am, ... ,aw)
b_j
p(S1, ... , Tn, ... ,Sm, ... ,Sw)
φ
b_j'
p(S1, ... , Tn, ... ,Tm, ... ,Sw)
θ'
g2
p(b1, ... , an, ... ,am, ... ,bw)

Figure 3.2: Evaluation through the chase

rule can be viewed as the evaluation of a relational algebra expression[a] [Ull88, AHV95]. In order to do that, we introduce a method that builds, from a given rule, a relational algebra expression that computes the same output database as the application of the body of a rule to an input database.

Let r be a rule. We shall assume that the body of r consists of subgoals $b_1, \ldots, b_n$ involving variables $X_1, \ldots, X_m$. For each $b_i = p_i(X_{i1}, \ldots, X_{ik_i})$, there is a relation R_i already computed, where the $X_i's$ are arguments.

For each b_i, let Q_i be the expression $\pi_{V_i}(\sigma_{W_i}(R_i))$, where π and σ are the projection and selection operators of relational algebra, respectively. Here, V_i is the set of different variables that appear among the terms of b_i. W_i is a conjunction of expressions of the form $\$k = \$l$. The expression $\$k = \$l$ is in W_i, if the positions k and l of b_i both contain the same variable.

The final expression is obtained by applying the natural join of all the $Q_i's$ previously defined.

Example 3.1.2 Let r be the rule:

[a]If the rule is recursive, then the relational algebra expression may have to be evaluated several times in order to obtain the same result as the original rule.

$r:\ c(X,Y):-p(X,Z),p(Y,W),s(Z,W)$

Suppose we have relations P and S computed for predicates p and s, respectively. We may imagine there is one copy of P with attributes X and Z and another with attributes Y and W. We suppose the attributes of S are Z and W. Then, the relation that corresponds to the body of r is:

$$C(X,Z,Y,W)=P(X,Z)\bowtie P(Y,W)\bowtie S(Z,W)$$

Obviously, over C, a projection is needed in order to obtain the desired final result. □

Example 3.1.2 is a very simple one. Now, let us consider in the next example a rule where there are atoms with some variables equated.

Example 3.1.3 Let r be the rule:

$r:\ c(X,Y):-r(X,Z,X),c(Z,Y)$

Suppose that we have already computed the relations R and C for subgoals r and c, respectively[b].

Observe that $r(X,Z,X)$ has the same variable in its first and third positions. Thus, this atom produces the following expression:

$$U(X,Z)=\pi_{1,2}(\sigma_{\$1=\$3}(R))$$

Note that the presence of the same variable in different positions generates a selection operation over R.

Thus, the final expression for the body of r is:

$$E(X,Y,Z)=U(X,Z)\bowtie C(Z,Y)$$

Obviously, over E, a projection is needed in order to obtain the desired final result. □

Therefore we can conclude that the equalization of variables during the chase has three main effects:

- It may introduce selections.

[b]Obviously, we consider the version of C resulting of previous non-recursive applications of recursive or non-recursive rules.

- It may introduce more arguments to perform the natural join. For example, we may have, before the chase, a join $e(X, Z) \bowtie p(Y, Z)$ and, after the chase, we may have to evaluate $e(Y, Z) \bowtie p(Y, Z)$. In the latter case, the join is performed using two arguments (Y and Z) whereas, in the previous case, the join is performed through only equalizations in the Z attribute.

- It may remove joins. For example, let us suppose that in the body of a rule there are two atoms $e(X, Y)$ and $e(Y, X)$ and that the chase of such a rule produces the equalization of the variables X and Y (assume that Y is replaced by X). Then one of the atoms can be removed (since both atoms became $e(X, X)$).

- It may transform a unbounded datalog program in a bounded dotalog program.

Now, let us remind some basic concepts about the computational costs of the relational algebra operators (see [AHV95]).

Selection can be realized in a straightforward manner by a scan of the argument relation and thus can be achieved in linear time. Access structures such as B-trees indexes or hash tables can be used to reduce the search time needed to find the selected tuples. Moreover, in the case of selections with single tuple output, this permits evaluation within essentially constant time (e.g., two or three page fetches). For larger outputs, the selection may take two or three page fetches per output tuple; this can be improved significantly if the input relation is clustered (i.e., stored so that all tuples with a given attribute value are on the same or contiguous disk pages).

However, the equi-join (or natural join $\bowtie$) is typically much more expensive because two relations are involved. A naive implementation of $\bowtie$ will take time on the order of the product $n_1 \times n_2$ of the sizes of the input relations I_1 and I_2. Obviously, several improvements have been developed. For example, the use of the *sort-merge* reduces the running time to the order of $max(n_1\ log\ n_1 + n_2\ log\ n_2$, size of output).

In addition to the discussion showed above, we have to keep in mind that an evaluation of a recursive rule may lead to the evaluation of the relational algebra expression several times in order to compute the whole answer. Thus, even the evaluation includes a intelligent strategy such as the incremental evaluation [Ull88], the recursion leads to computations much more costly than the case of the relational model.

Therefore, if the chase removes an atom in a rule, obviously it is a big

```
$J := 0$;
**foreach** $v$ in $I_2$
    **foreach**
        **If** $u$ and $v$ are joinable **then** $J := J \bigcup \{u \bowtie v\}$
```

Figure 3.3: Naive Join

improvement, since such removal leads to the removal of, at least, a join [c]. Although the chase would not remove atoms, if it equates variables, it still produces benefits in the running time because of the following two effects.

One of them is the equalization of variables in an atom. In this case, the equalization introduces a selection over in input relation. This effect reduces the number of tuples of one of the input relations of the join. This is a clear advantage since the reduction of tuples in the input of the join (as we have already seen, an operation much more expensive as the number of input tuples grows) leads to a reduction in the running time. It is clear that the benefits in the join pay the cost of the computation of the selection (an operation much less expensive).

On the other hand, the inclusion of more variables common to several atoms, may lead to the transformation of a cartesian product into a join, or at least, introduce more equality conditions in the join. A join with more arguments in common allows for an intelligent use of indexes that replaces, in the algorithm of Figure 3.3, the inner loop by indexed retrievals of tuples of T_2 that match the tuple of I_1 under consideration. Observe that the more attributes in common between the two relations, the more chances to use indexed retrievals. Therefore, using this approach, and assuming that a small number of tuples of I_2 match a given tuple of I_1, this approach computes the join in time proportional to the size of I_1, which is a great improvement.

3.1.2 Equivalence of a program and the chase of its rules

The definition of the chase of rules can be extended to programs as follows.

Let P be a program. Then *the chase of the rules of a program P with*

[c] In fact, it leads to the removal of a join operation for each application of the relational algebra expression to the database.

respect to F [GT95], denoted by $ChaseRules_F(P)$, is given by

$$ChaseRules_F(P) = \{r' \mid r' = Chase_F(r), r \in P\}$$

As the reader may expect, a program and the chase of its rules are equivalent over $SAT(F)$.

Corollary 3.1.1 Let P *be a program and let* F *be a set of fds defined over* $EDB(P)$*. Then* $P \equiv_{SAT(F)} ChaseRules_F(P)$*.*

Proof It follows from Lemma 3.1.1. $\square$

3.2 Chase of a tree

Let P be a $2-lsirup$, F a set of fds over $EDB(P)$ and T a tree in $trees(P)$. The *chase of* T *with respect to* F, denoted by $Chase_F(T)$ [GT95], is obtained by applying every fd in F to the atoms that are the leaves of T until no more changes can be made.

Example 3.2.1 Considering the tree T in Figure 3.4(a) and $F = \{e : \{1\} \rightarrow \{2\}\}$. T and $Chase_F(T)$ are:

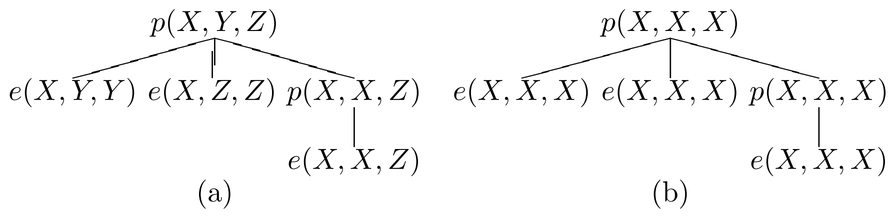


Figure 3.4: T and $Chase_F(T)$

Note that in T, the atoms $e(X, Y, Y), e(X, Z, Z)$ and $e(X, X, Z)$ have the same variable in the position defined by the left-hand side of the fd $e : \{1\} \rightarrow \{2\}$. Thus, variables Y, Z and X, which are placed (in those atoms) in the position defined by the right-hand side (of the same fd), are equated in $Chase_F(T)$.

Obviously from those trees the topMost and frontier can be computed:

$$topMost(T) = p(X, Y, Z) : -e(X, Y, Y), e(X, Z, Z), p(X, X, Z)$$

$$frontier(T) = p(X,Y,Z) : -e(X,Y,Y), e(X,Z,Z), e(X,X,Z)$$

$$topMost(Chase_F(T)) = p(X,X,X) : -e(X,X,X), e(X,X,X), p(X,X,X)$$

$$frontier(Chase_F(T)) = p(X,X,X) : -e(X,X,X), e(X,X,X), e(X,X,X)$$

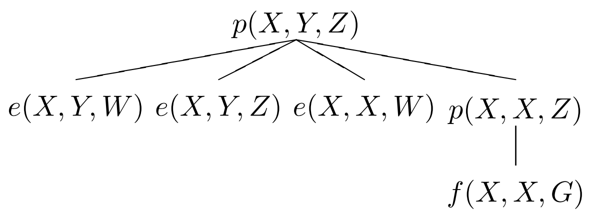
□

3.2.1 The chase is Church-Rosser

The *Church-Rosser* property of the chase [Mai83, AHV95] indicates that, given a tree T and a set of fds F, the chase of T with respect to F produces a unique end result.

However, the chase of a tree, as it is defined until now, would produce several isomorphic results. This occurs because the chase (until now) does not indicates which variable replaces the others, mainly because it does not matter at all.

Example 3.2.2 Let $F = \{e : \{1\} \rightarrow \{2\}, e : \{1\} \rightarrow \{3\}\}$, and let T be:



Two possible outputs of the $Chase_F(T)$ can be seen in Figure 3.5.

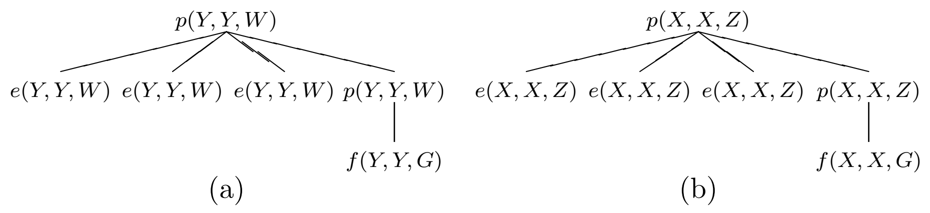


Figure 3.5: Two isomorphic chased trees

The trees in Figures 3.5(a) and 3.5(b) are isomorphic. Both trees are completely equivalent, however from now on, in order to use the word "equal" instead of "isomorphic", we are going to introduce a set of rules for equating variables during the chase of a tree.

□

Defining an order in the equalizations during the chase

We follow the two rules presented below in order to guarantee that the final result of the chase is independent of the order of the application of the fds and the order of the equalizations of variables.

When the chase of a tree equates two variables A and B, we use the following rules:

- If B is an AV and A is a CV, then all the occurrences of B are replaced by A.
- If both are either acyclic variables or cyclic variables:
    - If the first appearance of A in a level of the tree is in a level smaller than the level of the first appearance of B (i.e. A is in a level closer to the root), then all the occurrences of B are replaced by A.
    - If the first appearance of both variables is in the same level, then the variable in the biggest column in such a level is replaced by the variable in the smallest column.

Example 3.2.3 In Example 3.2.1, $Chase_F(T)$ produces two fd applications (that produce two equalizations):

- The equalization of Z and Y. Using the rules shown above, since Z is a CV and Y is an AV, Y is replaced by Z.
- The equalization of Z and X. In this case, since both are cyclic variables and their first appearance in T is in the same level (0), we replace the variable in the biggest column of level 0 (Z) by the variable in the smallest column (X).

Therefore, the resulting tree is in Figure 3.4(b). Observe that regardless of the fd application order, the tree in 3.4(b) is the final result.

$\square$

The Church-Rosser property of the chase

We now show that the order of the fd applications during the chase does not matter, that is, we guarantee that regardless of the fd application order during the chase, the resulting tree will be not only isomorphic but identical. That is, the chase is Church-Rosser [Mai83, AHV95].

Lemma 3.2.1 [Mai83, AHV95] Let T be a tree in $trees(P)$ where P is a $2-lsirup$, and let F be a set of fds over $EDB(P)$. During the $Chase_F(T)$, the order of application of the fds does not change the final result.

Proof It is obvious that all the variables that would be equated by the chase, finally are equated regardless of the fd application order. The application of a fd over two or more atoms leads to the equalization of several variables that before the fd application were distinct. Therefore, the application of a fd over several atoms does not exclude that other fds would be applied over other atoms.

In addition, the rules of renaming variables during the chase guarantees that independently of the order of the fd applications, at the end, all the variables that are equated among them are equated always to the same variable.

Hence, we can conclude that for any tree T_i in $trees(P)$ where P is a $2-lsirup$ and a set of fds F, there is only one $Chase_F(T_i)$. □

Example 3.2.4 Using the tree of Example 3.2.1, it is clear that the chase equates all the variables to X.

Let us consider different orders in the equalizations produced by the chase. One possibility is to equate first Y and Z.

Since Z is a CV and Y is an AV, then Y is replaced by Z. The next fd application equates Z and X, then Z is replaced by X, given that in both cyclic variables appears in level 0, and in such level, the variable in the smallest column is X.

Other possibility is to equate first X and Z, again Z would be replaced by X, and then Y would be replaced by X since Y is an AV.

Therefore, regardless of the fd application order, all variables are equated to X. □

3.3 Partial Chase of a tree

The *chase of datalog programs* (our first algorithm to optimize datalog programs) does not use the chase of trees, it uses a restricted version of the chase that we call *partial chase*.

The *partial chase of T with respect to* F, denoted by $ChaseP_F(T)$, is obtained by applying every fd in F to the leaves of T, except the atom(s) in the last level, until no more changes can be made.

Example 3.3.1 Using the tree T and the fds of the Example 3.2.1 we construct $ChaseP_F(T)$:

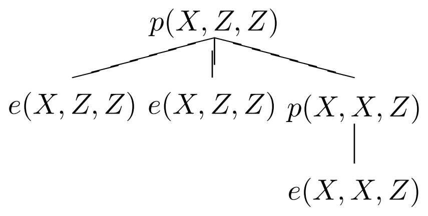


Figure 3.6: $ChaseP_F(T)$

$Chase_F(T)$ would produce also the equalization of X and Y, given that $e(X, X, Y)$ and $e(X, Y, Y)$ have the same variable in the position defined by the left-hand side of the fd $e : \{1\} \rightarrow \{2\}$, and X and Y are in the position defined by the right-hand side of such fd.

However, the partial chase does not consider the atom in the last level and then, the chase terminates with the tree shown above. □

3.3.1 Properties of the partial chase of trees built from $2 - lsirups$

In the partial chase, the atom in the last level is not involved in the fd applications during the chase. This exclusion has important effects in the chase, mainly because when we chase two trees built from the same program, the chase of the big one considers all the atoms considered in the chase of the small one plus other atoms.

This situation is not produced during the chase of trees (that is, considering the atom in the last level). In that case, the biggest tree does not have the atom in the last level of the small one, thus the chase of the big one may not contain all the equalization produced during the chase of the small one.

Lemma 3.3.1 Let P be a $2 - lsirup$, let F be a set of fds over $EDB(P)$. Let T_m and T_n be two trees in $trees(P)$ where $m > n$. Then, any level j of $ChaseP_F(T_m)$, where $j \leq n < m$, has less or equal different variables than level j of $ChaseP_F(T_n)$.

Proof Note that T_m has all the atoms of T_n that would be involved in the chase. The unique atom in T_n that T_m does not have, is the atom in the last level. However, the partial chase does not consider this atom.

Observe that level j of T_n and level j of T_m (before the chase) are equal. Moreover, any equalization in $ChaseP_F(T_n)$ is also produced in $ChaseP_F(T_m)$, therefore $ChaseP_F(T_m)$ may have more equalizations and thus, level j of $ChaseP_F(T_m)$ may have less different variables. □

Example 3.3.2 Let $P = \{r_0, r_1\}$ be:

$r_0 : p(X, Y, Z) : -e(X, Y, Z)$
$r_1 : p(X, Y, Z) : -e(X, Y, Y), e(X, Z, Z), p(X, X, Z)$

Let $F = \{e : \{1\} \to \{2\}\}$. Next, we can see T_1 and its partial chase.

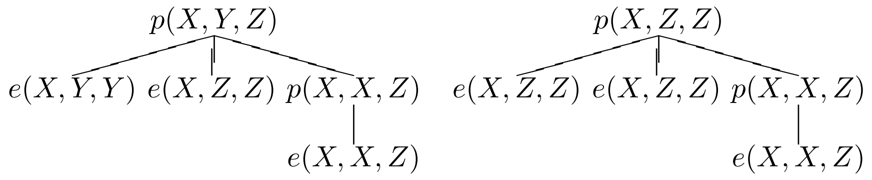


Figure 3.7: T_1 and $ChaseP_F(T_1)$

We can observe that the only equalization in level 1 is the equalization of Y and Z. Now, let us check T_2 (in Figure 3.8) and its partial chase (in Figure 3.9).

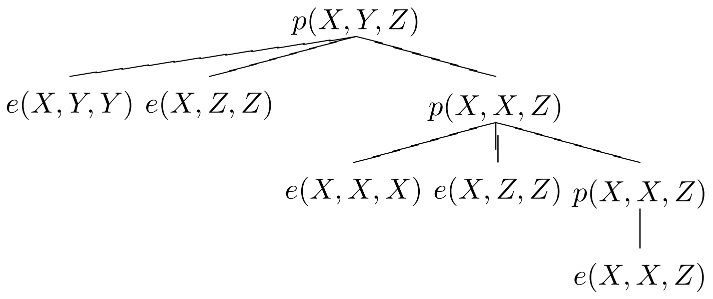


Figure 3.8: T_2

Level 1 of $ChaseP_F(T_2)$ includes the equalization of Y and Z as well. However, in such level, there is a new equalization, Y is equated to X due to the existence in level 2 of $e(X, X, X)$. Such atom and $e(X, Y, Y)$ (in level

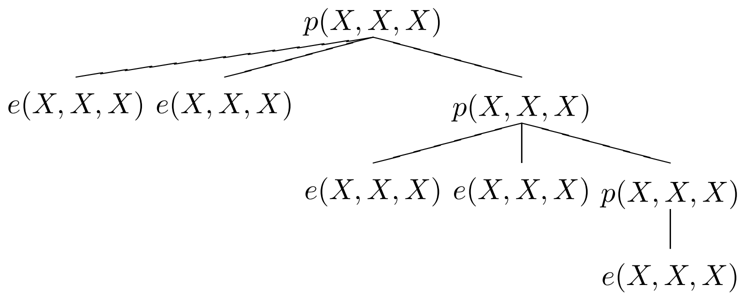


Figure 3.9: $ChaseP_F(T_2)$

1) have the same variable (X) in the position defined by the left-hand side of $e : \{1\} \rightarrow \{2\}$, therefore Y and X are equated.

Thus finally, Z, Y and X are equated. It is easy to see that the presence of a new level (level 2) generates a new equalization. However, the equalizations produced in $ChaseP_F(T_1)$ are also produced in $ChaseP_F(T_2)$, since T_2 contains the atoms $e(X, Y, Y)$ and $e(X, Z, Z)$[d].

□

The topMost of a tree is formed by levels 0 and 1, thus the previous lemma can be extended to the topMost.

Lemma 3.3.2 Let P be a $2 - lsirup$, let F be a set of fds over $EDB(P)$. Let T_m and T_n be two trees in $trees(P)$ where $m > n$. Then, $topMost(ChaseP_F(T_m))$ has equal or less different variables than $topMost(ChaseP_F(T_n))$.

Proof This lemma is a special particularization of Lemma 3.3.1. □

Example 3.3.3 Using the Example 3.3.2, if we check $ChaseP_F(T_1)$ and $ChaseP_F(T_2)$ we observe that the $topMost(ChaseP_F(T_2))$ includes all the equalizations found in $topMost(ChaseP_F(T_1))$ (the equalization of Y and Z) plus another equalization, the equalization of X and Z. □

Given a $2 - lsirup$ P, the set of variables in the topMost of any tree in $trees(P)$ is finite, since the topMosts of all the trees $T_i \in trees(P)$, with $i > 0$, are equal. Then, given a set of fds F over $EDB(P)$, by the previous lemma, it is obvious that there is a tree T_k such that all trees with more levels than T_k have the same topMost after their partial chase.

[d]These atoms are the atoms of T_1 that generate the equalization of Y and Z.

Lemma 3.3.3 Let P be a $2 - lsirup$, let F be a set of fds over $EDB(P)$. There is a tree T_k such that, for all $i \geq k$, $topMost(ChaseP_F(T_i))$ is equal to $topMost(ChaseP_F(T_k))$.

Proof Note that all the trees in $trees(P)$ with more than two levels have the same topMost, in fact, such topMost is r_1. Thus, the set of all variables in the topMost of all the trees in $trees(P)$ is finite. Therefore, there is a limit in the equalizations that can be produced in the topMost.

By Lemma 3.3.2, for all i, j such that $i > j > 0$, any equalization in $topMost(ChaseP_F(T_j))$ is also included in $topMost(ChaseP_F(T_i))$. Therefore, if T_k is a tree such that $ChaseP_F(T_k)$ produces in the topMost all the possible equalizations that may be produced in the topMost of the trees in $trees(P)$ with F (in the extreme case, all the variables of the topMost will be equated), then any tree with more levels will produce the same topMost after the chase. □

Example 3.3.4 Again, using the Example 3.3.2 it is easy to see that any tree bigger than T_2 will have the same topMost after its partial chase as T_2. Observe that $topMost(ChaseP_F(T_2)$ has only one variable, thus it is not possible to introduce new equalizations. □

The reasoning made for the previous lemma can be extended to any level of a tree.

Lemma 3.3.4 Let P be a $2-lsirup$, let F be a set of fds over $EDB(P)$. For any j in $\mathbb{N}$, there is a tree T_u, where $u > j$, such that level j of $ChaseP_F(T_u)$ is equal to level j of $ChaseP_F(T_M)$, for all $M > u$. That is, all the trees with more than u levels have, after the partial chase, equal level j.

Proof The number of variables in level j of all trees in $trees(P)$ is finite given that all levels j (if they exist) of any tree in $trees(P)$ are equal (if we except the case of T_{j-1}). This fact plus Lemma 3.3.1 implies that there is a tree T_u where all the possible equalizations among variables in level j are produced (in the extreme case, all the variables are equated). Therefore, for all trees T_M with $M > u$, level j of those trees are equal after the partial chase.

By the rules to rename variables during the chase introduced in Section 3.2.1, it is guaranteed that the equalizations in $ChaseP_F(T_u)$ and in $ChaseP_F(T_M)$ produce the same level j. □

Example 3.3.5 Let us consider the program and fds of Example 3.3.2. For level 1, it is obvious that T_2 is the tree such that level 1 of the partial chase of any tree bigger than T_2 will be equal to level 1 of $ChaseP_F(T_2)$, because all variables are equated.

□

3.3.2 Equivalence of the partial chase to the chase

In some special cases the partial chase of a tree may produce the same equalizations as the regular chase.

Lemma 3.3.5 Let P be a $2-lsirup$ and let Q be a program in class $\mathcal{L}$[e] *such that both programs have the same recursive rule. Let F and G be a two sets of fds defined over $EBD(P)$ and $EDB(Q)$ respectively, such that F and G have the same fds for all the fds defined over atoms that are in the recursive rule. Let T_j be a tree in $trees(P)$ and T'_j be a tree in $trees(Q)$ with $j+1$ levels, then the first j levels of $ChaseP_F(T_j)$ are equal to the first j levels of $Chase_G(T'_j)$.*

Proof Note the partial chase does no consider the atom in the last level, and for the programs in class $\mathcal{L}$ the (unique) predicate name of the atom in the non-recursive rule is not present in any atom of the recursive rule, therefore the atom in the last level would not be involved in a fd application. Thus in both cases, only the atoms in the leaves of all the levels except the last one are considered.

Given that we have considered that the recursive rules in both cases are the same rule, then clearly the first j levels of $ChaseP_F(T_j)$ are equal to the first j levels of $Chase_G(T'_j)$. □

Example 3.3.6 Let $P = \{r_0, r_1\}$, where:

$$r_0 = p(X,Y) :- a(X,Y)$$
$$r_1 = p(X,Y) :- a(Z,Z), a(X,Y), p(Z,Y)$$

Let $Q = \{r_0, r_1\}$, where:

$$r_0 = p(X,Y) :- e(X,Y)$$
$$r_1 = p(X,Y) :- a(Z,Z), a(X,Y), p(Z,Y)$$

Figure 3.10 shows T_2 and T'_2 obtained from P and Q respectively.

[e]Introduced in section 2.4.3.

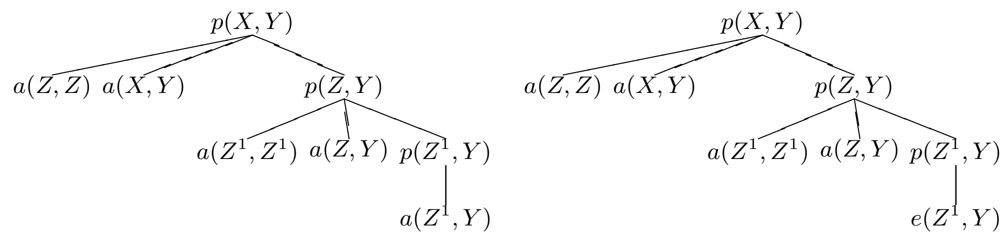


Figure 3.10: T_2 and T_2'

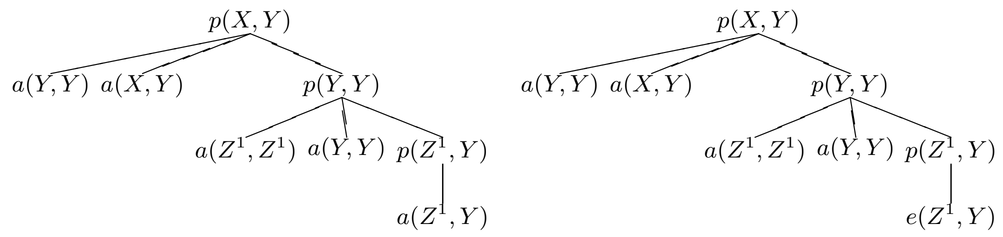


Figure 3.11: $ChaseP_F(T_2)$ and $Chase_G(T_2')$

Let F and G be $\{a : \{1\} \rightarrow \{2\}\}$. Figure 3.11 shows $ChaseP_F(T_2)$ and $Chase_G(T_2')$.

The first 3 levels of $ChaseP_F(T_2)$ and $Chase_G(T_2')$ are equal. However, note that if we chase T_2 w.r.t F, another equalization would be produced due to the existence of $a(Z^1, Y)$ and $a(Z^1, Z^1)$ that would equate Z^1 to Y. $\square$

3.4 Cyclic topMost

The cyclic topMost is a special type of topMost extracted from a chased tree.

Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. Let T_i be a tree in $trees(P)$, the cyclic topMost of T_i with respect to F ($CtopMost_F(T_i)$) is computed as follows:

- Let θ_i be the substitution defined by the $Chase_F(T_i)$.
- Let θ_i^c be θ_i where all the pairs X/Y are removed if X or Y (or both) are $AV's$.

- Then $CtopMost_F(T_i) = topMost(\theta_i^c(T_i))$.

That is, the cyclic topMost of a tree is the topMost of the chase of such a tree where the equalizations among CV's are maintained and the rest of the equalizations are removed[f]. Obviously, in this case, the $CtopMost$ introduces in the topMost less equalizations than the $topMost(Chase_F(T_i))$. However, the equalizations among CV's have special properties that will be helpful, as we will see in later chapters.

Example 3.4.1 Let $P = \{r_0, r_1\}$ be:

r_0: $p(X, Y, Z, A, B, C) :- e(X, Y, Z, A, B, C)$
r_1: $p(X, Y, Z, A, B, C) :- e(Y, X, Y, C, A, D), p(Z, X, Y, B, C, D)$

Let F be $\{\ e : \{6\} \rightarrow \{1\},\ e : \{6\} \rightarrow \{4\}\}$. In Figure 3.12 is shown T_2.

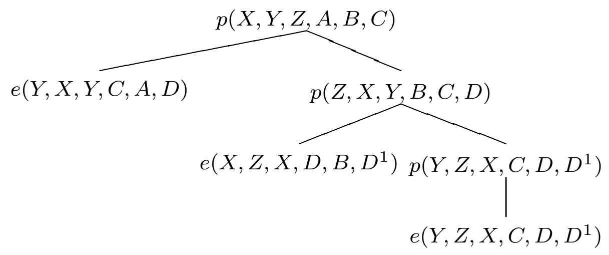


Figure 3.12: T_2

In Figure 3.13, $Chase_F(T_2)$ is shown.

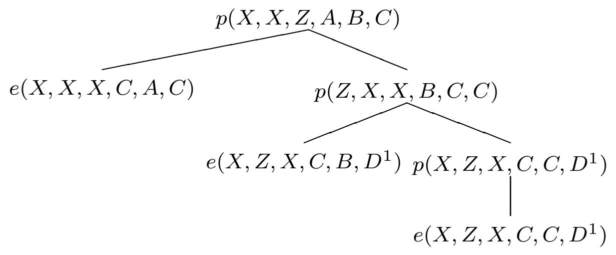


Figure 3.13: $Chase_F(T_2)$

The substitution defined by $Chase_F(T_2)$ (in Figure 3.13) is $\theta_2 = \{Y/X, D/C\}$. In order to compute the $CtopMost_F(T_2)$, we only consider

[f]The equalizations produced uses the rules showed in Section 3.2.1

the pairs of θ_2 where both variables are CV's, that is, $\theta_2^c = \{Y/X\}$. Hence, $CtopMost_F(T_2) = topMost(\theta_2^c(T_2))$ is :

$$p(X, X, Z, A, B, C) : -e(X, X, X, C, A, D), p(Z, X, X, B, C, D)$$

$\square$

Chapter 4

Equalization chains

Let us consider a fd application in a specific step of the chase of a tree, obviously such fd application equates two or more variables. These variables, which are placed in two or more atoms in the positions defined by the right-hand side of a fd, may be also placed in the L-th position (the position defined by the left-hand side of a fd) in two (or more) atoms with the same predicate name (say e). Then, assuming that there is a fd $e : \{L\} \rightarrow \{R\}$, then the chase in another step equates the variables in the R-th position of those atoms. This process can be repeated again and again producing an *equalization chain*.

The skeleton of these equalization chains is composed by atoms forming a structure that we call *atom chains*.

Although, equalization chains and atom chains are not final results, we think that they are valuable tools to study lsirups. Particularly, we use these structures in order to study the behavior of the chase when it is applied to expansion trees.

Therefore, due to the crucial importance in our results and the utility that they can have to solve different problems, we think that equalization chains and atom chains deserve a chapter to introduce them.

The outline of this chapter is as follows. In Section 4.1, we define what is a left-hand common set of atoms. In Section 4.2, we define the concept of atom chains. Finally, in Section 4.3, we give the definition of equalization chain.

4.1 Left-hand common set of atoms

The very first equalizations of the chase process are due to the presence of groups of atoms in the original tree (i.e., before the chase process begins) with the same predicate name and such that they have, in the positions defined by the left-hand side of a fd, the same variables (obviously, in the same order). We denote such group of atoms as *left hand common set of atoms.*

Definition 4.1.1 Let T be a tree in $trees(P)$ where P is a $2-lsirup$, and let F be a set of fds over $EBD(P)$. A left-hand common set of atoms *(LHCSA for short) is a set of, at least, two atoms of T, $\{a_1, a_2, \ldots, a_n\}$, such that they have the same predicate name e, there is a fd in F $e : \{L\} \rightarrow \{R\}$ and $a_1[L] = a_2[L] = \ldots = a_n[L]$.*

□

Example 4.1.1 Let $P = \{r_0, r_1\}$, where:

$$r_0 = p(X, Y) :- e(X, Y)$$
$$r_1 = p(X, Y) :- e(Z, X), e(X, Z), p(Z, Y)$$

Let F be $\{e : \{1\} \rightarrow \{2\}\}$. Using P, $T_2 = tree(r_1^2 \circ r_0)$ is shown in Figure 4.1.

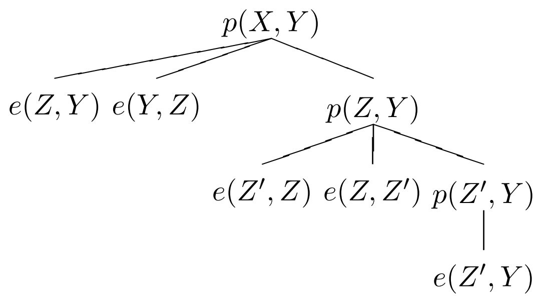


Figure 4.1: T_2

We can see in T_2 two $LHCSA$. One is formed by $e(Z, Y)$ and $e(Z, Z')$ and the other is formed by $e(Z', Z)$ and $e(Z', Y)$. □

4.2 Atom chains

Due to the substitution applied when the IDB atom in the body of the recursive rule is expanded (unfolded), different levels of a tree may have variables in common. Thus, in a tree T, a variable X may be in several atoms in the position defined by the right-hand side a fd, while X may be in another atoms in the positions defined by the left-hand side of another (or the same) fd. Such atoms form a structure that we call *atom chains*. Atom chains are a crucial concept to understand equalization chains.

Definition 4.2.1 Let P be a $2 - lsirup$, let F be a set of fds over $EDB(P)$. Let T_j a tree in $trees(P)$. An ***atom chain*** *is an ordered set of atoms:*

$$Ch = \{< a_1, a_2, a_3, ..., a_l >\}$$

where for all $1 \leq i < l$, $a_i[R_k] = a_{i+1}[L_t]$. That is, the variable in a_i in the position defined by the right-hand side of f_k is equal to the variable in a_{i+1} in the position defined by the left-hand side of f_t (that may be f_k again). □

Observe that it is not necessary that the fd used to define the right-hand side of a_i (f_k) should be equal to the fd used to define the left-hand side of a_{i+1} (f_t). Moreover, the predicate name of a_i may not be equal to the predicate name of a_{i+1}. It is possible to think in more complicated atom chains, for example, when the left-hand side of some fd has two variables, but all those complicated cases can be reduced to this simple atoms chains straightforward.

Example 4.2.1 Let F be $e : \{1\} \rightarrow \{5\}$ $a : \{1\} \rightarrow \{2\}$. Let $P = \{r_0, r_1\}$ be :

$r_0 :\ p(X, Y, Z, W, Q) : -e(X, Y, Z, W, Q)$
$r_1 :\ p(X, Y, Z, W, Q) : -a(V, Y), a(Z, W), e(Y, V, W, X, V), p(Y, V, W, Q, Z)$

If we inspect the tree in Figure 4.2, we can see the following five atom chains (notice that there are other atom chains):

- $Ch_1 = \{a_{5,3}, a_{5,1}, a_{4,1}, \ldots, a_{1,1}\}$
- $Ch_2 = \{a_{6,4}, a_{5,2}, a_{3,2}, a_{1,2}\}$
- $Ch_3 = \{a_{4,2}, a_{2,2}\}$

Each atom chain is illustrated in the Figure 4.2 with edges.

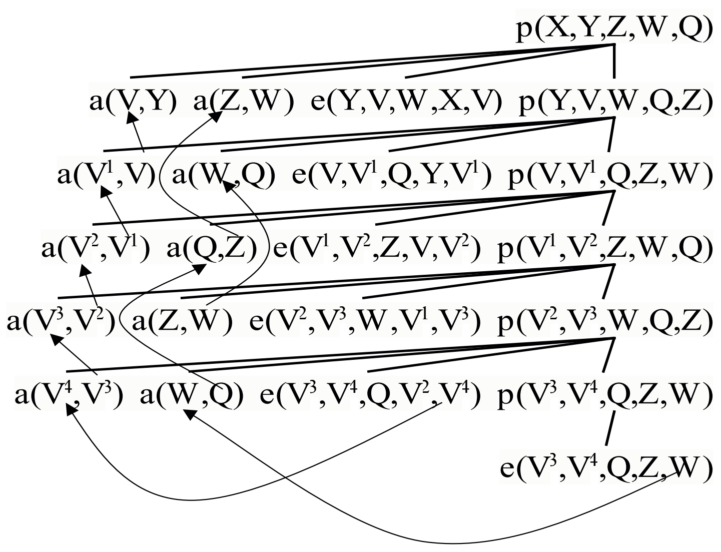

p(X,Y,Z,W,Q)
a(V,Y) a(Z,W) e(Y,V,W,X,V) p(Y,V,W,Q,Z)
a(V1,V) a(W,Q) e(V,V1,Q,Y,V1) p(V,V1,Q,Z,W)
a(V2,V1) a(Q,Z) e(V1,V2,Z,V,V2) p(V1,V2,Z,W,Q)
a(V3,V2) a(Z,W) e(V2,V3,W,V1,V3) p(V2,V3,W,Q,Z)
a(V4,V3) a(W,Q) e(V3,V4,Q,V2,V4) p(V3,V4,Q,Z,W)
e(V3,V4,Q,Z,W)

Figure 4.2: Atom chains

□

Let us suppose that an atom a_i belonging to an atom chain Ch has a cyclic variable in the position defined by the left-hand side of a fd that links it with the previous atom of the atom chain. Then, there may be another atom chains since, as we have already seen, CV's are in levels separated by $\mathcal{N}$ levels in the same columns. Let us illustrate that with the following example.

Example 4.2.2 Let $P = \{r_0, r_1\}$ be:

r_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
r_1: $p(X,Y,Z,A,B,C) :- e(B,C,E,Y,X,Z), e(Y,X,Y,C,A,D), p(Z,X,Y,B,C,D)$

Let F be $\{e : \{1\} \to \{3\}\}$. Then, in T_7 (in Figure 4.3), we can find (among others) the atom chain $Ch = \{a_{8,3}, a_{7,2}, \ldots\}$, that is:

$$Ch = \{e(Z,X,Y,D^4,D^5,D^6), e(Y,X,Y,D^5,D^3,D^6), \ldots\}$$

Notice that $e(Z,X,Y,D^4,D^5,D^6)$ has in its third position Y, that is, Y is in the position defined by the right-hand side of $e : \{1\} \to \{3\}$, and

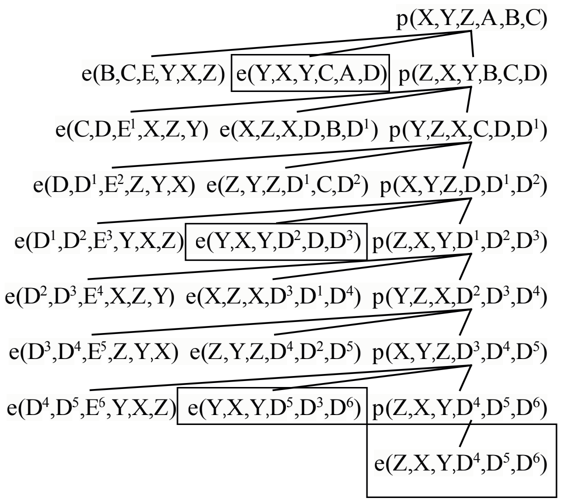

p(X,Y,Z,A,B,C)
e(B,C,E,Y,X,Z) e(Y,X,Y,C,A,D) p(Z,X,Y,B,C,D)
e(C,D,E¹,X,Z,Y) e(X,Z,X,D,B,D¹) p(Y,Z,X,C,D,D¹)
e(D,D¹,E²,Z,Y,X) e(Z,Y,Z,D¹,C,D²) p(X,Y,Z,D,D¹,D²)
e(D¹,D²,E³,Y,X,Z) e(Y,X,Y,D²,D,D³) p(Z,X,Y,D¹,D²,D³)
e(D²,D³,E⁴,X,Z,Y) e(X,Z,X,D³,D¹,D⁴) p(Y,Z,X,D²,D³,D⁴)
e(D³,D⁴,E⁵,Z,Y,X) e(Z,Y,Z,D⁴,D²,D⁵) p(X,Y,Z,D³,D⁴,D⁵)
e(D⁴,D⁵,E⁶,Y,X,Z) e(Y,X,Y,D⁵,D³,D⁶) p(Z,X,Y,D⁴,D⁵,D⁶)
e(Z,X,Y,D⁴,D⁵,D⁶)

Figure 4.3: T_7

$e(Y, X, Y, D^5, D^3, D^6)$ has in its first position Y again. Since Y is a cyclic variable then in T_7, we can find (among others) the following atom chains:

$$\begin{array}{l||l} Ch^1 = \{a_{8,3}, a_{4,2}, \ldots\} & Ch^1 = \{e(Z, X, Y, D^4, D^5, D^6), e(Y, X, Y, D^2, D, D^3), \ldots\} \\ Ch^2 = \{a_{8,3}, a_{1,2}, \ldots\} & Ch^2 = \{e(Z, X, Y, D^4, D^5, D^6), e(Y, X, Y, C, A, D), \ldots\} \end{array}$$

The first atom $a_{8,3}$ ($e(Z, X, Y, D^4, D^5, D^6)$) remains the same in the three atom chains. However in Ch^1 and Ch^2, even the other two atoms ($a_{4,2}$ and $a_{1,2}$) are the same relative position (2) as $a_{7,2}$ is in Ch, $a_{4,2}$ and $a_{1,2}$ are separated three (in the case of $a_{4,2}$) and six (in the case of $a_{1,2}$) levels with respect to the atom $a_{7,2}$ in Ch, given that $\mathcal{N}$ for P is 3.

□

All these situations and other circumstances are captured by equalization chains.

4.3 Equalization chains

The programs that we tackle have only one linerar recursive rule. Therefore, if we inspect a tree T_i built from a 2-lsirup (or *lsirup*), all levels bigger than level 0 and smaller than level $i + 1$ are isomorphic given that there is only one recursive rule. This a very important fact that reader has to keep in mind since we will exploit it intensively in this section.

Let us consider a fd application in a specific step of the chase of a tree, obviously such fd application equates two or more variables. These variables, which are placed in two or more atoms in the positions defined by the right-hand side of a fd, may be also placed in the L-th position (the position defined by the left-hand side of a fd) in two (or more) atoms with the same predicate name (say e). Then, assuming that there is a fd $e : \{L\} \rightarrow \{R\}$, then the chase, in another step, equates the variables in the R-th position of those atoms. This process can be repeated again and again producing an *equalization chain*[a].

We are going to characterize these equalization chains in order to show a very important property of the partial chase of trees built from a $2-lsirup$.

Equalization chains are extremely unpredictable since they depend on the $2 - lsirup$ used to build the tree, the number of levels of the tree and finally, the fds used to chase the tree. Any of these three factors is very important, but let us point out that the height of the chased tree is a new factor that has not been considered in other problems solved by the chase. The height of a tree is a consequence of the recursion.

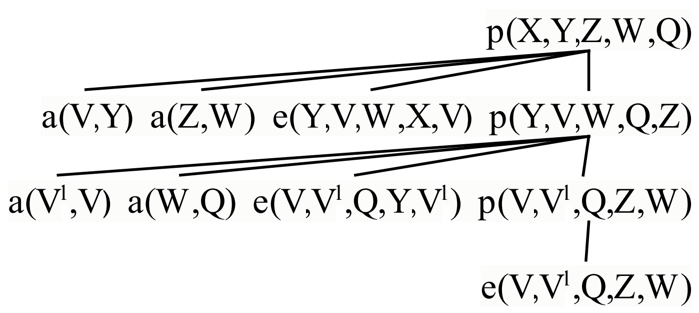
Hence, the chase, a tool that until now was shown as a very useful procedure to solve different problems, turns a very intricate process to analyze, and the most important, it is very difficult to predict its behavior.

With the discussion showed above we try to show the difficulty of the problem, and the necessity of tools to analyze what happens during the chase of the trees.

Definition 4.3.1 Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. An equalization chain *in the process of chasing a tree (that may be a partial chase as well) is an ordered set of groups of atoms (links) $l_1, l_2, \ldots, l_n$, where:*

- *l_1 is formed by a left-hand common set of atoms.*
- *The variable in the position defined by the left-hand side of a fd f_k of any atom in a group l_i $(i > 1)$ is equal to the variable in the position defined by the right-hand side of a fd f_j (that may be f_k again) of some atom of the group l_{i-1}.*
- *After the equalizations produced by the chase using the atoms in the group l_{i-1}, for any atom a_j in l_i defined over the predicate name e, there is in l_i, at least, another e-atom a_k that has the same variable as a_j in the position defined by the left-hand side of a fd.*

[a]Observe that an equalization chain always starts due to a LHCSA.



p(X,Y,Z,W,Q)
a(V,Y) a(Z,W) e(Y,V,W,X,V) p(Y,V,W,Q,Z)
a(V1,V) a(W,Q) e(V,V1,Q,Y,V1) p(V,V1,Q,Z,W)
e(V,V1,Q,Z,W)

Figure 4.4: T_2

Example 4.3.1 Let E be the equalization chain:

$$\overbrace{\{a(X,Y),a(X,Z)\}}^{l_1},\overbrace{\{b(Y,V),b(Z,W),b(Y,M),c(Z,U),c(Y,L)\}}^{l_2},\overbrace{\{a(V,H),a(W,F)\}}^{l_3}$$

Let us suppose that the set of fds used is $\{a:\{1\}\to\{2\},\ b:\{1\}\to\{2\},\ c:\{1\}\to\{2\}\}$.

The chase equates, in l_1, Y to Z. In the next step of the chase, that is l_2, equates two sets of variables; one equates V, W and M, the other equates U and L. Therefore, the variables in the position defined by the left-hand side of $a:\{1\}\to\{2\}$ in the atoms of the set l_3 are equated. Then finally, H is equated to F. □

Note that in each group of an equalization chain, each atom is a member of an atom chain. That is, the skeleton of the equalization chains are atom chains.

Example 4.3.2 In Figure 4.4 we show the tree T_2 built from the program of Example 4.2.1, using the set of fds $F=\{a:\{1\}\to\{2\},\ e:\{1\}\to\{5\}\}$ there is one equalization chain:

$$\{e(V,V^1,Q,Z,W),e(V,V^1,Q,Y,V^1)\},\{a(W,Q),a(V^1,V)\}$$

The first group is a LHCSA that equates W and V^1. These variables are also in the position defined by the left-hand side of $a:\{1\}\to\{2\}$ in the atoms of the second group. Thus continuing with the chase, V and Q are equated as well.

In the equalization chain, there are two atom chains involved:

$$\{e(V, V^1, Q, Z, W), a(W, Q)\}$$
$$\{e(V, V^1, Q, Y, V^1), a(V^1, V)\}$$

Note that each group of the equalization chain takes one atom from each atom chain.

$\square$

4.3.1 Partial chase and equalization chains

Let us suppose that the chase of a tree T produces the equalization of two variables placed in $a_{i,j}[n]$ and $a_{k,l}[n]$. Then, if we suppose that T is a tree big enough, then the chase of T also equates the variables placed in $a_{i-1,j}[n]$ and $a_{k-1,l}[n]$, and in $a_{i+1,j}[n]$ and $a_{k+1,l}[n]$. That is a very important property that will be very useful in the following chapter. Next, we formalize this affirmation and we prove it.

Lemma 4.3.1 Let P be a $2-lsirup$, let F be a set of fds over $EDB(P)$. Let T_M be a tree in $trees(P)$. If $ChaseP_F(T_M)$ equates the variables in $a_{i,j}[R]$ and $a_{m,n}[R]$ ($i < m$), then $ChaseP_F(T_M)$ equates also the variables in the $R-th$ position of:

- *$a_{i-x,j}$ and $a_{m-x,n}$, for all x, $0 < x \leq \boldsymbol{u}$, and*
- *$a_{i+y,j}$ and $a_{m+y,n}$, for all y, $0 < y \leq \boldsymbol{b}$.*

Where $\boldsymbol{b}$ ($b \leq M - m$) and $\mathbf{u}$ ($u < i$) are two numbers that depend on the equalization chain and the tree height.

Proof Let $q : \{L\} \rightarrow \{R\}$ be the fd applied over $a_{i,j}$ and $a_{m,n}$. Let us suppose that the equalization of $a_{i,j}[R]$ and $a_{m,n}[R]$ is due to an equalization chain $E = \{l_1, l_2, \ldots, l_y\}$. Let us precise some of the groups[b] that form E:

$$E = \{\overbrace{\{a_{u,v}, a_{f,g}\}}^{l_1}, \ldots, \overbrace{\{a_{h,w}, a_{c,k}\}}^{l_j}, \ldots, \overbrace{\{a_{z,x}, a_{r,s}\}}^{l_k}, \ldots, \overbrace{\{a_{i,j}, a_{m,n}\}}^{l_y}\}$$

Let us suppose that $a_{h,w}$ is the atom in E that is in the smallest level. Then, h is the shortest level where it is possible to find an atom in E.

[b]We assume that two atoms form each group. However, our results can be generalized to groups with a larger number of atoms

P is a $2-lsirup$, hence there is only one recursive rule in P. Therefore, for any $T_i \in trees(P)$ all levels bigger than level 0 and smaller than level $i+1$ are isomorphic. Thus there are $h-1$ equalization chains that are "parallel" to E but in levels upwards with respect to those where can be found atoms of E:

$$\{a_{u-(h-1),v}, a_{f-(h-1),g}\}, \ldots, \{a_{h-(h-1),w}, a_{c-(h-1),k}\}, \ldots, \{a_{i-(h-1),j}, a_{m-(h-1),n}\}$$
$$\{a_{u-(h-2),v}, a_{f-(h-2),g}\}, \ldots, \{a_{h-(h-2),w}, a_{c-(h-2),k}\}, \ldots, \{a_{i-(h-2),j}, a_{m-(h-2),n}\}$$
$$\vdots$$
$$\{a_{u-1,v}, a_{f-1,g}\}, \ldots, \{a_{z-1,x}, a_{r-1,s}\}, \ldots, \{a_{h-1,w}, a_{c-1,k}\}, \ldots, \{a_{i-1,j}, a_{m-1,n}\}$$

Thus in this case, $\mathbf{u}$ is $h-1$. Note that $a_{h-(h-1),w}$ ($a_{1,w}$) is an atom in the first level. An equalization chain started by $a_{u-h,v}$ and $a_{f-h,g}$ would be interrupted in the link corresponding to $a_{h-h,w}$ since such atom does not exist.

The same reasoning can be done thinking in the atom of E that is in the biggest level. $\square$

Let us suppose that the atom in E which is in the biggest level is $a_{r,s}$, then we can see in Figure 4.5 that $\mathbf{b}$, in this case, is $M-r$ since $a_{r+(M-r),s}$ is an atom in the the level before the last one, that is, the last level that contains atoms that can be involved in the partial chase. Thus, an equalization chain started by $a_{u+(M-r)+1,v}$ and $a_{f+(M-r)+1,g}$ would be interrupted in the link corresponding to $a_{r+(M-r)+1,s}$ ($a_{M+1,s}$) since such atom does not exist.

Example 4.3.3 Let $P = \{r_0, r_1\}$ be:

r_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
r_1: $p(X,Y,Z,A,B,C) :- e(B,C,E,Y,X,Z), e(Y,X,Y,C,A,D), p(Z,X,Y,B,C,D)$

Let F be $\{e : \{1\} \to \{6\}\}$. Then, in T_7 (in Figure 4.6) can be found (among others) the equalization chain $E_1 = \{\{a_{1,2}, a_{4,2}\}, \{a_{3,1}, a_{6,1}\}\}$:

$$E_1 = \{e(Y,X,Y,C,A,D), e(Y,X,Y,D^2,D,D^3)\}, \{e(D,D^1,E^2,Z,Y,X), e(D^3,D^4,E^5,Z,Y,X)\}$$

Note that there can not be a parallel equalization chain using atoms in the same relative positions as the atoms in E_1 but in smaller levels, since in this case, the atom in E_1 that is in the smallest level is in level 1 ($e(Y,X,Y,C,A,D)$). However, the atom in E_1 that is in the biggest level is in level 6, thus there is a parallel equalization chain $E_2 = \{\{a_{2,2}, a_{5,2}\}, \{a_{4,1}, a_{7,1}\}\}$:

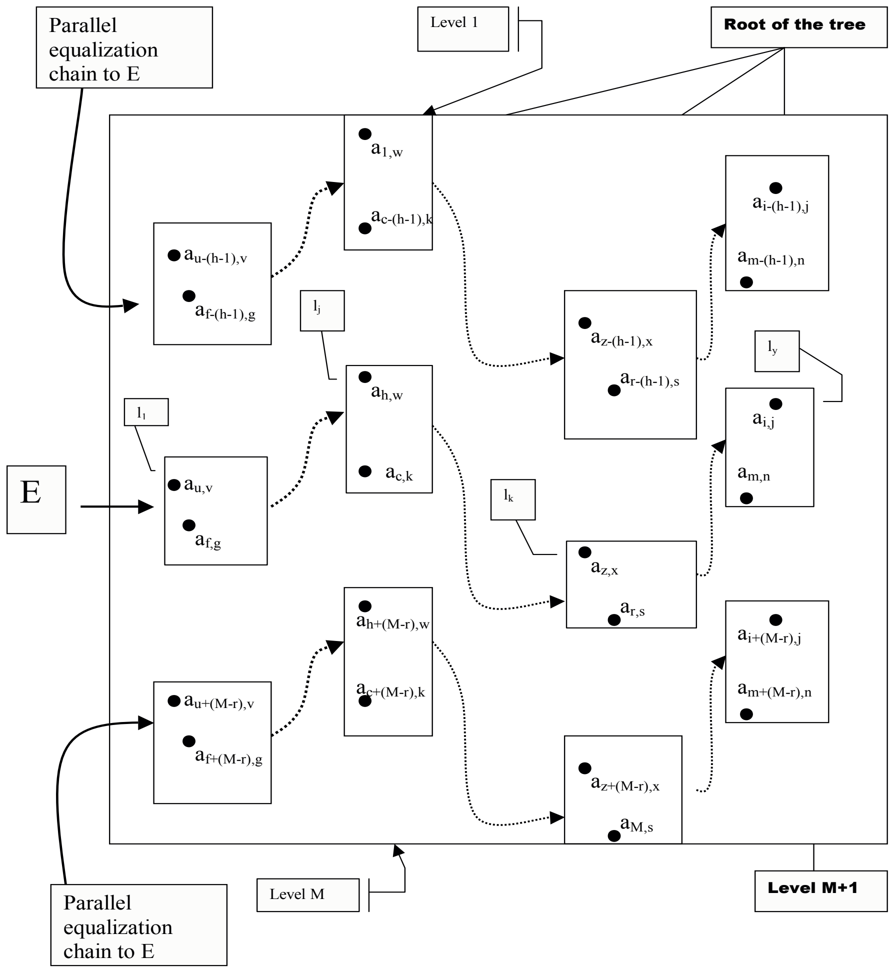


Figure 4.5: Parallel equalization chains in T_M

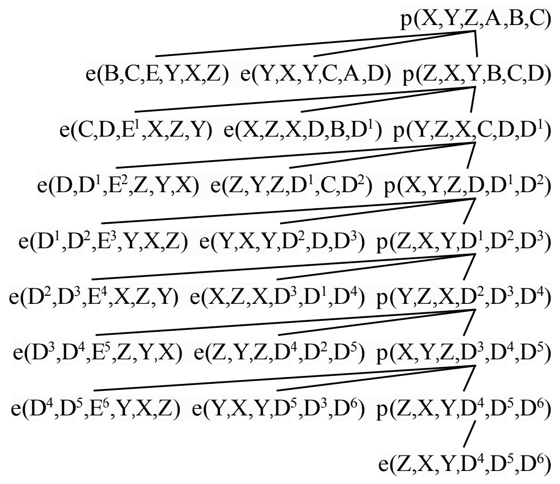

p(X,Y,Z,A,B,C)
e(B,C,E,Y,X,Z) e(Y,X,Y,C,A,D) p(Z,X,Y,B,C,D)
e(C,D,E1,X,Z,Y) e(X,Z,X,D,B,D1) p(Y,Z,X,C,D,D1)
e(D,D1,E2,Z,Y,X) e(Z,Y,Z,D1,C,D2) p(X,Y,Z,D,D1,D2)
e(D1,D2,E3,Y,X,Z) e(Y,X,Y,D2,D,D3) p(Z,X,Y,D1,D2,D3)
e(D2,D3,E4,X,Z,Y) e(X,Z,X,D3,D1,D4) p(Y,Z,X,D2,D3,D4)
e(D3,D4,E5,Z,Y,X) e(Z,Y,Z,D4,D2,D5) p(X,Y,Z,D3,D4,D5)
e(D4,D5,E6,Y,X,Z) e(Y,X,Y,D5,D3,D6) p(Z,X,Y,D4,D5,D6)
e(Z,X,Y,D4,D5,D6)

Figure 4.6: T_7

$E_2 = \{e(X,Z,X,D,B,D^1), e(X,Z,X,D^3,D^1,D^4)\}, \{e(D^1,D^2,E^3,Y,X,Z), e(D^4,D^5,E^6,Y,X,Z)\}$

Note that for each atom $a_{l,m}$ in l_i of E_1, we can find the atom $a_{l+1,m}$ in l_i of E_2.

Observe that there is not any other equalization chain parallel to E_1 and E_2 since E_2 has an atom in the level before the last one ($e(D^4,D^5,E^6,Y,X,Z)$).

Therefore, the left hand common set of atoms formed by $e(Z,Y,Z,D^1,C,D^2)$ and $e(Z,Y,Z,D^4,D^2,D^5)$, which equates D^2 and D^5, does not equate variables placed in the left-hand side of the fd $e : \{1\} \rightarrow \{6\}$, given that D^5 does not appear in the first position of any atom. □

As a consequence of the previous lemma it is easy to see that if during the $ChaseP_F(T_l)$, $a_{i,j}[n]$ is equated to $a_{k,m}[n]$, then in $ChaseP_F(T_{l+1})$, $a_{i,j}[n]$ is equated to $a_{k,m}[n]$, and $a_{i+1,j}[n]$ is equated to $a_{k+1,m}[n]$. This property can be proven by the same reasons as the previous lemma has been proved correct.

We present it to the reader to provide to the reader some background knowledge useful in the next chapter.

Chapter 5

Optimization using partial chase of trees

We have so far seen that the chase is used with several purposes. Among them, we can find the optimization of tableau queries [AHV95], optimization of queries in the framework of the object-relational model [PDST00] or even the optimization of structural queries [PV99]. All these different applications of the chase in different data models demonstrates that the chase is a very useful tool to optimize queries.

However, it is not easy to find literature about the use of the chase in order to optimize recursive queries (deductive model). The closest work may be the one from Sagiv [Sag87] that uses the chase to test uniform containment of datalog programs.

However, whereas Sagiv used tuple-generating dependencies, we use functional dependencies. The only works we know that deal with the optimization of recursive datalog programs using fds are those from Lakshmanan and Hernández [LH91], Gonzalez-Tuchmann [GT95], Tang [Tan97] and Brisaboa et al. [BGTHP98a, BGTHP98b, PBPH00, BHPP01]. This deficit of works should not be taken as an indication that the problem is not interesting. On the contrary, the ubiquitousness of fds in real databases makes this approach very attractive [AHV95]. The reason for the scarcity of results may lie in the fact that, from a research point of view, this problem is extremely intricate.

In this chapter, we introduce our first algorithm that builds a program P' equivalent to a given $2-lsirup$ P, when both are applied over databases satisfying a set of functional dependencies. Although, for simplicity, we present our results for $2-lsirups$, it is easy to see that these results can be extended to *lsirups*, as we will illustrate later.

This algorithm is based in the application of the partial chase using a set of fds F over a sequence of trees in $trees(P)$ until it is obtained a tree T_k such that all trees bigger than it have the same topMost (after their partial chase). That is, the topMost of any tree bigger than T_k (after its partial chase) is equal to $topMost(Chase_F(T_k))$. Once we get such topMost, we provide it to the algorithm of optimization that we call *chase of datalog programs* (shown in Figure 5.7) that obtains the program P' which is equivalent to P when both are applied to databases in $SAT(F)$.

In Chapter 3, we showed the chase of a rule. Such definition was extended to the chase of the rules of a program. Now, in this chapter we define our first algorithm that optimizes a datalog program, the chase of datalog programs.

The difference between the chase of the rules of a program and the chase of a datalog program is in the number of equalizations. A rule has a small amount of atoms in comparison with the atoms that can be found in all the trees that can be built from the program that contains such rule. Therefore, the chase of such rule has few chances to introduce equalizations in comparison with the chances of the chase of all the trees that can be built from the program. Note that in all those trees, there are much more atoms, in fact, there are infinite atoms (because $trees(P)$ has infinite trees).

We provide a method that translates the equalizations found in those trees to the program itself. This is a big advantage with respect to the chase of a rule since the chase of datalog programs pushes more equalizations. Therefore, the new program will be cheaper to evaluate.

Our procedure is very similar to the one introduced by Lakshmanan and Hernández [LH91]. Both of them transform a lsirup into some non-recursive rules plus a recursive rule where some variables are equated. The difference between our technique and Lakshmanan and Hernández's approach is that ours pushes more equalizations among variables than Lakshmanan and Hernández's does.

The outline of this chapter is as follows. First, in Section 5.1, we introduce a new notation for variables in trees. In Section 5.2, we define what is a *stabilized tree*. Next, in Section 5.3, we show how to find a tree T_k such that the topMost of the partial chase of any bigger tree will be equal to the topMost of the partial chase of T_k. Finally, in Section 5.4 we introduce the algorithm of the *chase of datalog programs*.

5.1 Renaming of variables

In this chapter, when we use trees, we will rename the acyclic variables with special names that will be helpful to reveal some properties of the trees that will be needed later. Any acyclic variable in a tree is renamed with the label $V_{l,c}$, where l is the first level where the variable appears in the tree, and c is the column of such appearance, always inspecting the tree from top to bottom and from left to right.

Example 5.1.1 Let $P = \{r_0, r_1\}$ be:

$r_0 : p(A, B, C) : -e(X, Y, B, C, A)$
$r_1 : p(A, B, C) : -e(M, N, A, H, I), a(B, C, M, I), p(B, A, H)$

Using P, T_3 (in the normal notation) is shown in Figure 5.1.

Columns	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Level 0										p(A,	B,	C)		
Level 1	e(M,	N,	A,	H,	I)	a(B,	C,	M,	I)	p(B,	A,	H)		
Level 2	e(M^1,	N^1,	B,	H^1,	I^1)	a(A,	H,	M^1,	I^1)	p(A,	B,	H^1)		
Level 3	e(M^2,	N^2,	A,	H^2,	I^2)	a(B,	H^1,	M^2,	I^2)	p(B,	A,	H^2)		
Level 4										e(X,	Y,	A,	H^2	B)

Figure 5.1: T_3 without renamed variables

From now on, we shall use the notation of the tree shown in Figure 5.2. In this example, there are two CV's, A and B. These variables maintain the name that they have in the program. The rest are $AV's$, thus they are renamed.

Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Level 0										p(A,	B,	$V_{0,12}$)		
Level 1	e($V_{1,1}$,	$V_{1,2}$,	A,	$V_{1,4}$,	$V_{1,5}$)	a(B,	$V_{0,12}$,	$V_{1,1}$,	$V_{1,5}$)	p(B,	A,	$V_{1,4}$)		
Level 2	e($V_{2,1}$,	$V_{2,2}$,	B,	$V_{2,4}$,	$V_{2,5}$)	a(A,	$V_{1,4}$,	$V_{2,1}$,	$V_{2,5}$)	p(A,	B,	$V_{2,4}$)		
Level 3	e($V_{3,1}$,	$V_{3,2}$,	A,	$V_{3,4}$,	$V_{3,5}$)	a(B,	$V_{2,4}$,	$V_{3,1}$,	$V_{3,5}$)	p(B,	A,	$V_{3,4}$)		
Level 4										e($V_{4,10}$	$V_{4,11}$	A	$V_{3,4}$	B)

Figure 5.2: A tree with renamed variables

Let us focus, for example, in the variable H^2 (in the tree of of Figure 5.1). Inspecting T_3 from top to bottom, and form left to right, H^2 appears for first time in level 3 and column 4, thus, this variable is denoted by $V_{3,4}$.

□

Observe that this notation does not introduce any change in the tree, trees in Figures 5.1 and 5.2 are completely equivalent.

5.2 Stabilized tree

Given a $2 - lsirup$ and a set of fds F, in order to compute the chase of P with respect to F ($Chase_F(P)$), we first need to find a tree T_k such that the $topMost(ChaseP_F(T_l))$, for all $l > k$, is equal to $topMost(ChaseP_F(T_k))$. In this chapter, we will use T_k to denote such tree.

In Lemma 3.3.3, the existence of T_k was shown. However, it is not easy to find T_k, it is a very complex process based mainly in the concept of *stabilized tree* introduced in this section.

Notice that as we have seen in Lemma 4.3.1, if we do not consider the atom in the last level during the chase of a tree, that is, when we use the partial chase, equalization chains may be broken in the first level, or in the level before the last level of the tree. Thus, in intermediate levels, there are more equalizations among their variables than in the first and last levels. That is, as it was proven in Lemma 4.3.1, for any equalization during the partial chase of a tree that equates $a_{i,j}[R]$ and $a_{m,n}[R]$, there are two numbers b and u such that the partial chase of such tree also equates:

- $a_{i-x,j}[R]$ and $a_{m-x,n}[R]$, for all x, $0 < x \leq u$, and
- $a_{i+y,j}[R]$ and $a_{m+y,n}[R]$, for all y, $0 < y \leq b$.

Therefore, depending on the equalization chain that produces the equalization of $a_{i,j}[R]$ and $a_{m,n}[R]$, more precisely, on the atoms of such an equalization chain that are in the biggest and lowest level, variables in the first or last levels may be equated or not.

By Lemma 3.3.4, we know that there is a tree in $trees(P)$ containing levels enough to produce all the possible equalizations among the variables of an intermediate level, and trees with more levels than such tree, by Lemma 4.3.1, after the partial chase, have more intermediate levels with all the possible equalizations among their variables. However, first and last levels may have less equalizations among their variables, due to the interruption of equalization chains.

We call those trees big enough to present all the possible equalizations (after the partial chase) in their intermediate levels, *stabilized trees.* When a tree is stabilized, all trees bigger than such tree are also stabilized.

Given a $2 - lsirup$ P and a set of fds F over $EDB(P)$, it is clear that there is a tree T_n such that, T_n is stabilized. It follows immediately from Lemmas 4.3.1 and 3.3.4.

Intuitively, a tree T_i is stabilized when it has enough levels to produce all the possible equalization chains, and then, bigger trees will have parallel equalization chains to those in T_i, as we saw in Lemma 4.3.1.

From the discussion shown above, we can easily see that all stabilized trees, after their partial chase, can be divided in three zones:

1. *The upper zone* which is formed by the first U levels of all stabilized trees, where U depends on P and F. The upper zones are identical in all stabilized trees (for a given P and F) and therefore, all stabilized trees have the same topMost (after the partial chase).

2. *The bottom zone* which is formed by the last B levels of all stabilized trees, where B depends on P and F. The bottom zones of all stabilized trees are isomorphic.

3. *The intermediate zone* of a tree is formed by the intermediate levels that do not belong to the upper zone and bottom zone. All levels in intermediate zones are isomorphic among them, given that all the possible equalizations among their variables have been performed by the partial chase[a].

Let us recall that our main target is the computation of the chase of a datalog program with respect to a set of fds. In order to achieve such goal, we need to find T_k[b] for a given $2 - lsirup$ P and a set of fds F.

Thus, we have to provide a method to find T_k. We use the concept of stabilized tree, since all stabilized trees have the same topMost after their partial chase. Hence, we conclude that T_k is the smallest tree in $trees(P)$ which has the same topMost (after its partial chase) as the topMost (also after the partial chase) of any stabilized tree.

[a]Observe that before the partial chase, all levels except level 0 and the last level are isomorphic.

[b]A tree such that the $topMost(ChaseP_F(T_l))$, for all $l > k$, is equal to $topMost(ChaseP_F(T_k))$.

Summarizing, we have reduced the problem of finding T_k to the search of a stabilized tree.

Therefore, we need to find a condition that allows us to check if a tree (in $trees(P)$) is stabilized. That is, we need to find a tree with enough levels to allow the chase to perform all the patterns of equalizations that any tree in $trees(P)$ (using a set of fds F) may present. In order to do that, we use the concept of *formation rule* that is introduced in the next subsection.

Example 5.2.1 Let $P = \{r_1, r_0\}$ be the program:

$r_0 : p(A, B, C, D, E, F, G, H, I) : -a(A, B, C, D, E, F, G, H, I)$
$r_1 : p(A, B, C, D, E, F, G, H, I) : -$
$e(A, W), e(F, V), e(D, I), e(I, U), e(V, F), p(C, A, B, N, D, E, M, G, H)$

$\mathcal{N}$ for P is 3. Using P, we show in Table 5.1 T_{10}. The first column shows the level of the tree. The following five columns represent the variables of the five e-atoms in each level (with the AV's already renamed). The last column represents the variables of the p-atom in each level.

L	e atoms					p atom
0						A B C $V_{0,14}$ $V_{0,15}$ $V_{0,16}$ $V_{0,17}$ $V_{0,18}$ $V_{0,19}$
1	A $V_{1,2}$	$V_{0,16}$ $V_{1,4}$	$\boxed{V_{0,14} V_{0,19}}$	$\boxed{V_{0,19} V_{1,8}}$	$V_{1,4}$ $V_{0,16}$	C A B $V_{1,14}$ $V_{0,14}$ $V_{0,15}$ $V_{1,17}$ $V_{0,17}$ $V_{0,18}$
2	C $V_{2,2}$	$V_{0,15}$ $V_{2,4}$	$V_{1,14}$ $V_{0,18}$	$V_{0,18}$ $V_{2,8}$	$V_{2,4}$ $V_{0,15}$	B C A $V_{2,14}$ $V_{1,14}$ $V_{0,14}$ $V_{2,17}$ $V_{1,17}$ $V_{0,17}$
3	B $V_{3,2}$	$\boxed{V_{0,14} V_{3,4}}$	$V_{2,14}$ $V_{0,17}$	$V_{0,17}$ $V_{3,8}$	$\boxed{V_{3,4} V_{0,14}}$	A B C $V_{3,14}$ $V_{2,14}$ $V_{1,14}$ $V_{3,17}$ $V_{2,17}$ $V_{1,17}$
4	A $V_{4,2}$	$V_{1,14}$ $V_{4,4}$	$V_{3,14}$ $V_{1,17}$	$V_{1,17}$ $V_{4,8}$	$V_{4,4}$ $V_{1,14}$	C A B $V_{4,14}$ $V_{3,14}$ $V_{2,14}$ $V_{4,17}$ $V_{3,17}$ $V_{2,17}$
5	C $V_{5,2}$	$V_{2,14}$ $V_{5,4}$	$V_{4,14}$ $V_{2,17}$	$V_{2,17}$ $V_{5,8}$	$V_{5,4}$ $V_{2,14}$	B C A $V_{5,14}$ $V_{4,14}$ $V_{3,14}$ $V_{5,17}$ $V_{4,17}$ $V_{3,17}$
6	B $V_{6,2}$	$V_{3,14}$ $V_{6,4}$	$V_{5,14}$ $V_{3,17}$	$V_{3,17}$ $V_{6,8}$	$V_{6,4}$ $V_{3,14}$	A B C $V_{6,14}$ $V_{5,14}$ $V_{4,14}$ $V_{6,17}$ $V_{5,17}$ $V_{4,17}$
7	A $V_{7,2}$	$V_{4,14}$ $V_{7,4}$	$V_{6,14}$ $V_{4,17}$	$V_{4,17}$ $V_{7,8}$	$V_{7,4}$ $V_{4,14}$	C A B $V_{7,14}$ $V_{6,14}$ $V_{5,14}$ $V_{7,17}$ $V_{6,17}$ $V_{5,17}$
8	C $V_{8,2}$	$V_{5,14}$ $V_{8,4}$	$V_{7,14}$ $V_{5,17}$	$V_{5,17}$ $V_{8,8}$	$V_{8,4}$ $V_{5,14}$	B C A $V_{8,14}$ $V_{7,14}$ $V_{6,14}$ $V_{8,17}$ $V_{7,17}$ $V_{6,17}$
9	B $V_{9,2}$	$V_{6,14}$ $V_{9,4}$	$V_{8,14}$ $V_{6,17}$	$V_{6,17}$ $V_{9,8}$	$V_{9,4}$ $V_{6,14}$	A B C $V_{9,14}$ $V_{8,14}$ $V_{7,14}$ $V_{9,17}$ $V_{8,17}$ $V_{7,17}$
10	A $V_{10,2}$	$V_{7,14}$ $V_{10,4}$	$V_{9,14}$ $V_{7,17}$	$V_{7,17}$ $V_{10,8}$	$V_{10,4}$ $V_{7,14}$	C A B $V_{10,14}$ $V_{9,14}$ $V_{8,14}$ $V_{10,17}$ $V_{9,17}$ $V_{8,17}$
11						C A B $V_{10,14}$ $V_{9,14}$ $V_{8,14}$ $V_{10,17}$ $V_{9,17}$ $V_{8,17}$

Table 5.1: T_{10}

Let $F = \{e : \{1\} \rightarrow \{2\}\}$. Then, $ChaseP_F(T_{10})$ is shown in Table 5.2.

It is easy to see that in T_{10} there are enough levels to produce parallel equalization chains. For example, if we check the equalization chain E, shown below (and marked in Table 5.1), we notice that there are parallel equalization chains (E^1, $E^2, ...$) in successive levels.

Level	e atoms					p atom
0						A B C $V_{0,14}$ $V_{0,15}$ $V_{0,16}$ $V_{0,17}$ $V_{0,18}$ $V_{0,19}$
1	A $V_{1,2}$	$V_{0,16}$ $V_{1,4}$	$V_{0,14}$ $V_{0,19}$	$V_{0,19}$ $V_{0,14}$	$V_{1,4}$ $V_{0,16}$	C A B $V_{1,14}$ $V_{0,14}$ $V_{0,15}$ $V_{1,17}$ $V_{0,17}$ $V_{0,18}$
2	C $V_{2,2}$	$V_{0,15}$ $V_{2,4}$	$V_{1,14}$ $V_{0,18}$	$V_{0,18}$ $V_{1,14}$	$V_{2,4}$ $V_{0,15}$	B C A $V_{2,14}$ $V_{1,14}$ $V_{0,14}$ $V_{2,17}$ $V_{1,17}$ $V_{0,17}$
3	B $V_{3,2}$	$V_{0,14}$ $V_{0,19}$	$V_{2,14}$ $V_{0,17}$	$V_{0,17}$ $V_{2,14}$	$V_{0,19}$ $V_{0,14}$	A B C $V_{3,14}$ $V_{2,14}$ $V_{1,14}$ $V_{3,17}$ $V_{2,17}$ $V_{1,17}$
4	A $V_{1,2}$	$V_{1,14}$ $V_{0,18}$	$V_{3,14}$ $V_{1,17}$	$V_{1,17}$ $V_{3,14}$	$V_{0,18}$ $V_{1,14}$	C A B $V_{4,14}$ $V_{3,14}$ $V_{2,14}$ $V_{4,17}$ $V_{3,17}$ $V_{2,17}$
5	C $V_{2,2}$	$V_{2,14}$ $V_{0,17}$	$V_{4,14}$ $V_{2,17}$	$V_{2,17}$ $V_{4,14}$	$V_{0,17}$ $V_{2,14}$	B C A $V_{5,14}$ $V_{4,14}$ $V_{3,14}$ $V_{5,17}$ $V_{4,17}$ $V_{3,17}$
6	B $V_{3,2}$	$V_{3,14}$ $V_{1,17}$	$V_{5,14}$ $V_{3,17}$	$V_{3,17}$ $V_{5,14}$	$V_{1,17}$ $V_{3,14}$	A B C $V_{6,14}$ $V_{5,14}$ $V_{4,14}$ $V_{6,17}$ $V_{5,17}$ $V_{4,17}$
7	A $V_{1,2}$	$V_{4,14}$ $V_{2,17}$	$V_{6,14}$ $V_{4,17}$	$V_{4,17}$ $V_{6,14}$	$V_{2,17}$ $V_{4,14}$	C A B $V_{7,14}$ $V_{6,14}$ $V_{5,14}$ $V_{7,17}$ $V_{6,17}$ $V_{5,17}$
8	C $V_{2,2}$	$V_{5,14}$ $V_{3,17}$	$V_{7,14}$ $V_{5,17}$	$V_{5,17}$ $V_{7,14}$	$V_{3,17}$ $V_{5,14}$	B C A $V_{8,14}$ $V_{7,14}$ $V_{6,14}$ $V_{8,17}$ $V_{7,17}$ $V_{6,17}$
9	B $V_{3,2}$	$V_{6,14}$ $V_{4,17}$	$V_{8,14}$ $V_{6,17}$	$V_{6,17}$ $V_{9,8}$	$V_{4,17}$ $V_{6,14}$	A B C $V_{9,14}$ $V_{8,14}$ $V_{7,14}$ $V_{9,17}$ $V_{8,17}$ $V_{7,17}$
10	A $V_{1,2}$	$V_{7,14}$ $V_{5,17}$	$V_{9,14}$ $V_{7,17}$	$V_{7,17}$ $V_{10,8}$	$V_{5,17}$ $V_{7,14}$	C A B $V_{10,14}$ $V_{9,14}$ $V_{8,14}$ $V_{10,17}$ $V_{9,17}$ $V_{8,17}$
11						C A B $V_{10,14}$ $V_{9,14}$ $V_{8,14}$ $V_{10,17}$ $V_{9,17}$ $V_{8,17}$

Table 5.2: $ChaseP_F(T_{10})$

$$E = \{a_{1,3}, a_{3,2}\}, \{a_{1,4}, a_{3,5}\}$$
$$E^1 = \{a_{2,3}, a_{4,2}\}, \{a_{2,4}, a_{4,5}\}$$
$$E^2 = \{a_{3,3}, a_{5,2}\}, \{a_{3,4}, a_{5,5}\}$$
$$\vdots$$

Therefore, in this example it is easy to see that the tree in Table 5.2 is stabilized. However, we need a procedure to be able to find which is the first (smallest) stabilized tree (after the partial chase) for any $2 - lsirup$ P and any set of fds over $EDB(P)$. □

5.2.1 Formation rules

Formation rules (frs) are a special type of rules that represent levels of trees. They are not normal datalog rules, they are rules similar to the levels of a chased tree where some variables have been renamed in order to show the equalizations that the chase produces. Each level of a stabilized tree corresponds (is adjusted) to a specific formation rule. In formation rules, variables from a level of a tree are represented in an unambiguous way, and they show where a variable in a specific position of the tree is coming from.

The aim of the definition of formation rules is the search of a finite set of formation rules that describes precisely any stabilized tree for a given $2 - lsirup$ and a set of fds. In other words, let P be a $2 - lsirup$ and let F be a set of fds over $EDB(P)$. There is a finite set of formation rules $FR_F(P)$ that can be used to build any stabilized tree (after its partial chase) since each level of any stabilized tree (for P and F) is "adjusted" to a specific formation rule in $FR_F(P)$. Such finite set of rules depends on P and F. Next, let us show what we mean when we say that a level of a partially chased tree is adjusted to a formation rule.

Definition 5.2.1 Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. Let T_i be a tree in $trees(P)$. Level $j > 0$ of T_i is adjusted to the formation rule fr if and only if:

1. *The body of fr corresponds to level j of $ChaseP_F(T_i)$.*
2. *The head of fr corresponds the IDB atom at level $j-1$ of $ChaseP_F(T_i)$.*
3. *Any CV has the same name and positions in level j (and the IDB atom in level $j-1$) of $ChaseP_F(T_i)$ and in fr.*
4. *For any acyclic variable $V_{n,m}$ in level j or in the IDB atom of level $j-1$:*
   (a) *If the variable $V_{n,m}$ appears in the same column in other levels before or after level j (or level $j-1$ if it is the case) of $ChaseP_F(T_i)$, then $V_{n,m}$ remains unchanged in the formation rule.*
   (b) *In other case, the variable $V_{n,m}$ is represented in fr by the relative variable $R_{j-n,m}$ or $R_{j-1-n,m}$, if $V_{n,m}$ is the IDB atom at level $j-1$.*

□

$j - n$ shows the difference in levels between the first level n where $V_{n,m}$ appears and the level j corresponding to the formation rule.

Therefore, variables in a formation rule give us precisely the position of the first appearance in the tree (from top to bottom and from left to right) of each variable in the level to which the formation rule is adjusted.

Example 5.2.2 Level 3 of the tree T_{10} in Example 5.2.1 is adjusted to the formation rule:

$$
\begin{aligned}
fr:\ & p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})\text{:-}\\
& e(B,V_{3,2}),e(R_{3,14},R_{3,19}),e(R_{1,14},R_{3,17}),e(R_{3,17},R_{1,14}),e(R_{3,19},R_{3,14}),\\
& p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})
\end{aligned}
$$

Let us check for example the left-most atom in the body of fr. It is equal to the correspondent atom in level 3 of $ChaseP_F(T_{10})$. Note that its first variable (B) is a CV and then, such variable remains unchanged in fr. The second one, ($V_{3,2}$), appears in other levels of $ChaseP_F(T_{10})$ in the second column as well, thus this variable remains unchanged.

Now, let us check the atom in the second relative position of level 3 of $ChaseP_F(T_{10})$, $e(V_{0,14},V_{0,19})$. The correspondent atom in fr is

$e(R_{3,14}, R_{3,19})$. In this case, original variables $V_{0,14}$ and $V_{0,19}$ are renamed since they are AV's, and they do not appear in their respective columns in levels bigger or smaller than level 3. Therefore, $V_{0,14}$ is renamed by the relative variable $R_{3-0,14}$ and $V_{0,19}$ is renamed by $R_{3-0,19}$.

Let us reason about the meaning of the variables in a formation rule. Let us go back again to $e(B, V_{3,2})$, and focus in $V_{3,2}$. With the information provided by the sub-indexes we know that $V_{3,2}$ appears for first time (in $ChaseP_F(T_{10})$) in level 3, and column 2.

Now, we turn our attention to a relative variable, for example, $R_{1,14}$ (in atom $e(R_{1,14}, R_{3,17})$). Since level 3 is adjusted to fr, $R_{1,14}$ represents the variable $V_{2,14}$ ($R_{3-2,14}$). That is, the first appearance (in $ChaseP_F(T_{10})$) of the variable in level 3 that corresponds to $R_{1,14}$ was in level 2, and column 14.

□

5.2.2 The finite set of formation rules $FR_F(P)$

Given a $2-lsirup$ P and a set of fds F, in order to find a stabilized tree (for P and F), we need to find the finite set of formation rules $FR_F(P)$ that describes any stabilized tree (for P and F). In this section we analyze $FR_F(P)$. This set of formation rules for any $2-lsirup$ P and a set of fds F over $EDB(P)$ is formed by 3 different ordered subsets where the last one has $\mathcal{N}$ ordered subsets :

$$FR_F(P) = \{\{< frU_1, \ldots, frU_U >\}, \{< frI_1, \ldots, frI_{\mathcal{N}} >\}, \\ \{\{< frB_{1,1}, \ldots, frB_{1,B} >\}, \ldots, \{\{< frB_{\mathcal{N},1}, \ldots, frB_{\mathcal{N},B} >\}\}$$

Next, we describe the composition of these 3 subsets of formation rules for a stabilized tree T_M (see Figure 5.3):

1. A set of U formation rules, one for each level in the first U levels of any stabilized tree. U is fixed for a program P and a set of fds F.

   $$< frU_1,\ frU_2, \ldots,\ frU_U >$$

   It is possible that 2 or more formation rules in this set would be equal. Let l be a level of a stabilized tree such that $l \leq U$, then l is adjusted to the formation rule frU_l.

2. An ordered set of $\mathcal{N}$ formation rules for the intermediate levels:

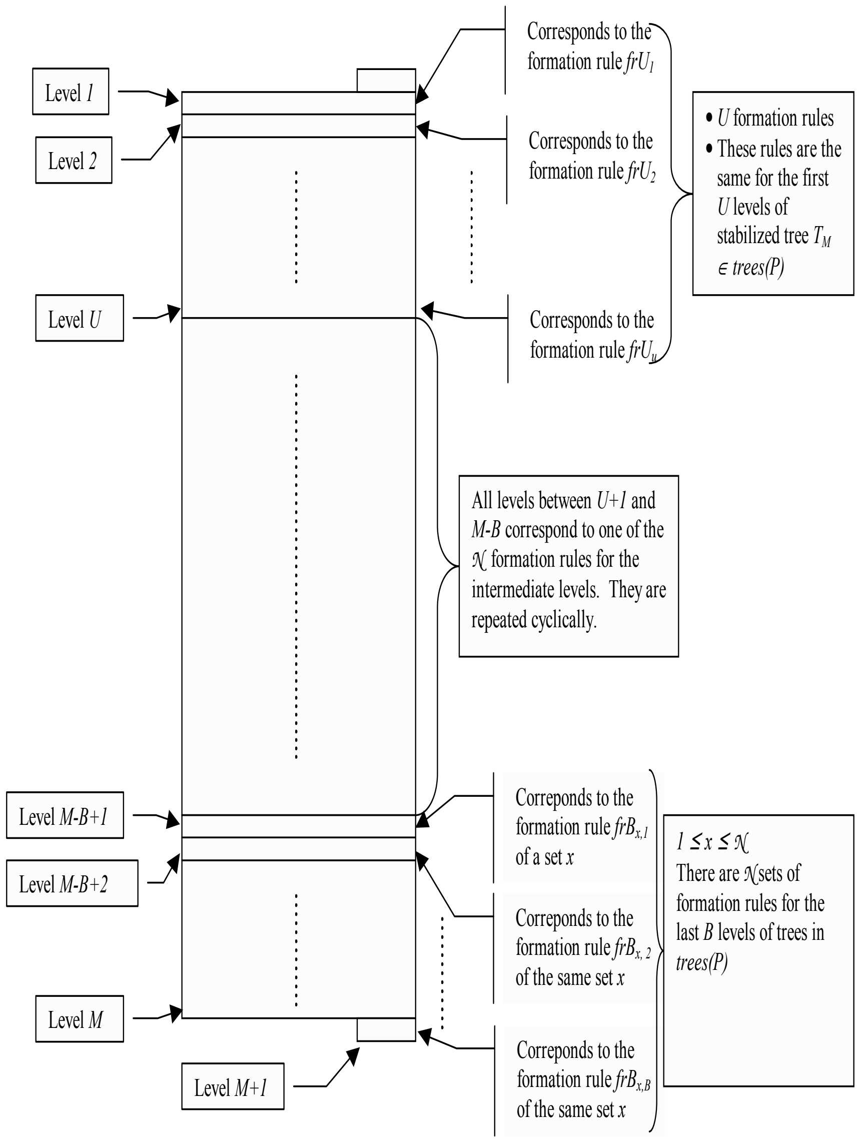

Level *1*
Level *2*
Level *U*
Level *M-B+1*
Level *M-B+2*
Level *M*
Level *M+1*
Corresponds to the formation rule frU_1
Corresponds to the formation rule frU_2
Corresponds to the formation rule frU_u
• *U* formation rules
• These rules are the same for the first *U* levels of stabilized tree T_M ∊ *trees(P)*
All levels between *U+1* and *M-B* correspond to one of the 𝒩 formation rules for the intermediate levels. They are repeated cyclically.
Correponds to the formation rule $frB_{x,1}$ of a set *x*
Correponds to the formation rule $frB_{x,2}$ of the same set *x*
Correponds to the formation rule $frB_{x,B}$ of the same set *x*
$1 \le x \le \mathcal{N}$
There are 𝒩 sets of formation rules for the last *B* levels of trees in *trees(P)*

Figure 5.3: Structure of a stabilized tree T_M

$$< frI_1, frI_1, \ldots, frI_{\mathcal{N}} >$$

Levels in the intermediate zone are cyclically adjusted to these formation rules. That is, let level l be a level in the intermediate zone ($U < l \leq M - B$), then if l adjusts to frI_j, then level $l+1$ adjusts either to frI_{j+1} or, if frI_{j+1} does not exist, to frI_1.

3. For the last B levels of stabilized trees, there are $\mathcal{N}$ ordered sets of formation rules.

$$\begin{aligned}
\{B_1 &= \{< frB_{1,1}, frB_{1,2}, \ldots, frB_{1,B}\} > \\
B_2 &= \{< frB_{2,1}, frB_{2,2}, \ldots, frB_{2,B} >\} \\
&\vdots \\
B_{\mathcal{N}} &= \{< frB_{\mathcal{N},1}, frB_{\mathcal{N},2}, \ldots, frB_{\mathcal{N},B}\} >\}
\end{aligned}$$

Each set has B formation rules, where B depends on P and F.

Consecutive levels in this last zone of the tree adjust to consecutive formation rules and consecutive trees adjust cyclically to consecutive sets of formation rules. That is, if the last B levels of a stabilized tree T_i adjust to the set B_j, then the last B levels of T_{i+1} adjust either to B_{j+1} or, if B_{j+1} does not exist, to B_1.

Each set is completely isomorphic to the others, that is, for all $1 \leq x \leq B$, $frB_{1,x}, frB_{2,x}, \ldots, frB_{\mathcal{N},x}$ are isomorphic. The difference among the sets is in the CV's, each set corresponds to one combination of CV's in the positions where CV's are in the formation rules.

Next, we formalize the previous discussion, defining when a tree after its partial chase is adjusted to a set of formation rules.

Definition 5.2.2 Let T_M be a tree built from a $2-lsirup$ P, and let F be a set of fds over $EDB(P)$. We say that T_M is adjusted to a set of frs $FR_F(P)$, if:

- *Any level l, $l \leq U$, is adjusted to frU_l.*
- *For any intermediate level l, $U < l \leq M-B$, if l adjusts to frI_j, then level $l+1$ adjusts either to frI_{j+1} or, if frI_{j+1} does not exist, to frI_1.*

- *For any level l, such that $l > M - B$, that is, it belongs to the bottom zone, if l adjusts to the formation rule $frB_{j,k}$, then $l+1$ adjusts either to $frB_{j,k+1}$, or if $frB_{j,k+1}$ does not exist, to $frB_{j,1}$.*
- *If the last B levels of T_M adjusts to the set B_j, then the last B levels of T_{M+1} adjusts either to B_{j+1} or, if B_{j+1} does not exist, to B_1.*

□

Example 5.2.3 The tree T_{10} of Example 5.2.1 is adjusted to the set of formation rules:

frU_1 : $p(A,B,C,R_{0,14},R_{0,15},R_{0,16},R_{0,17},R_{0,18},R_{0,19}) :-$
$e(A,V_{1,2}), e(R_{1,16},R_{0,4}), e(R_{1,14},R_{1,19}), e(R_{1,19},R_{1,14}), e(R_{0,4},R_{1,16}),$
$p(C,A,B,R_{0,14},R_{1,14},R_{1,15},R_{0,17},R_{1,17},R_{1,18})$

frU_2 : $p(C,A,B,R_{0,14},R_{1,14},R_{1,15},R_{0,17},R_{1,17},R_{1,18})$:-
$e(C,V_{2,2}), e(R_{2,15},R_{0,4}), e(R_{1,14},R_{2,19}), e(R_{2,18},R_{1,14}), e(R_{0,4},R_{2,15}),$
$p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

frU_3 : $p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(B,V_{3,2}), e(R_{3,14},R_{3,19}), e(R_{1,14},R_{3,17}), e(R_{3,17},R_{1,14}), e(R_{3,19},R_{3,14}),$
$p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

frU_4 : $p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(A,V_{1,2}), e(R_{3,14},R_{4,18}), e(R_{1,14},R_{3,17}), e(R_{3,17},R_{1,14}), e(R_{4,18},R_{3,14}),$
$p(C,A,B,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

frI_1 : $p(C,A,B,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(C,V_{2,2}), e(R_{3,14},R_{5,17}), e(R_{1,14},R_{3,17}), e(R_{3,17},R_{1,14}), e(R_{5,17},R_{3,14}),$
$p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

frI_2 : $p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(B,V_{3,2}), e(R_{3,14},R_{5,17}), e(R_{1,14},R_{3,17}), e(R_{3,17},R_{1,14}), e(R_{5,17},R_{3,14}),$
$p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

frI_3 : $p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(A,V_{1,2}), e(R_{3,14},R_{5,17}), e(R_{1,14},R_{3,17}), e(R_{3,17},R_{1,14}), e(R_{5,17},R_{3,14}),$
$p(C,A,B,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

$frB_{1,1}$: $p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(B,V_{3,2}), e(R_{3,14},R_{5,17}), e(R_{1,14},R_{3,17}), e(R_{3,17},R_{0,8}), e(R_{5,17},R_{3,14}),$
$p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

$frB_{1,2}$: $p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(A,V_{1,2}), e(R_{3,14},R_{5,17}), e(R_{1,14},R_{3,17}), e(R_{3,17},R_{0,8}), e(R_{5,17},R_{3,14}),$
$p(C,A,B,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

$frB_{1,3}$: $p(C,A,B,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$a(C,A,B,R_{1,14},R_{2,14},R_{3,14},R_{1,17},R_{2,17},R_{3,17})$

Observe that the first four levels of the tree $ChaseP_F(T_{10})$ follow the formation rules $frU_1, frU_2, frU_3, frU_4$, respectively. Such formation rules are the same for the four first levels of any stabilized tree in $trees(P)$, where P and the fds used to chase the trees are those of Example 5.2.1.

Notice that intermediate levels start in level five. The formation rules for the intermediate levels are three (since $\mathcal{N}$ is 3) and we call them frI_1, frI_2, frI_3. Note that levels 5 and 8 adjust to the formation rule frI_1. In trees bigger than T_{10}, intermediate formation rules will correspond cyclically to the intermediate levels (that is, for example, frI_1 will describe levels 5, 8, 11, 14, etc., frI_3 will describe levels 7, 10, 13, 16, etc.).

In any stabilized tree in $trees(P)$ (being P and F of Example 5.2.1), the bottom zone will have three levels.

The last 3 levels of T_{10} adjusts to formation rules $frB_{1,1}$, $frB_{1,2}$ and $frB_{1,3}$. Thus, T_{11} adjusts to $frB_{2,1}$, $frB_{2,2}$ and $frB_{2,3}$.

Formation rules $frB_{2,1}$, $frB_{2,2}$ and $frB_{2,3}$ are:

$frB_{2,1}$: $p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(A,V_{3,2}),e(R_{3,14},R_{5,17}),e(R_{1,14},R_{3,17}),e(R_{3,17},R_{0,8}),e(R_{5,17},R_{3,14}),$
$p(C,A,B,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

$frB_{2,2}$: $p(C,A,B,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(C,V_{1,2}),e(R_{3,14},R_{5,17}),e(R_{1,14},R_{3,17}),e(R_{3,17},R_{0,8}),e(R_{5,17},R_{3,14}),$
$p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

$frB_{2,3}$: $p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$a(B,C,A,R_{1,14},R_{2,14},R_{3,14},R_{1,17},R_{2,17},R_{3,17})$

Finally, the last B levels of T_{12} are adjusted to the remaining set of B formation rules, $frB_{3,1}$, $frB_{3,2}$ and $frB_{3,3}$.

$frB_{3,1}$: $p(C,A,B,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(A,V_{3,2}),e(R_{3,14},R_{5,17}),e(R_{1,14},R_{3,17}),e(R_{3,17},R_{0,8}),e(R_{5,17},R_{3,14}),$
$p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

$frB_{3,2}$: $p(B,C,A,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$e(B,V_{1,2}),e(R_{3,14},R_{5,17}),e(R_{1,14},R_{3,17}),e(R_{3,17},R_{0,8}),e(R_{5,17},R_{3,14}),$
$p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$

$frB_{3,3}$: $p(A,B,C,R_{0,14},R_{1,14},R_{2,14},R_{0,17},R_{1,17},R_{2,17})$:-
$a(A,B,C,R_{1,14},R_{2,14},R_{3,14},R_{1,17},R_{2,17},R_{3,17})$

Notice that the only difference, for example, among $frB_{1,1}$, $frB_{2,1}$ and $frB_{3,1}$ is in the CV's. Since $\mathcal{N}$ is 3, there are three formation rules to describe the first level of the bottom zone of any tree. In each set, CV's are placed in different positions. With $frB_{1,1}$, $frB_{2,1}$ and $frB_{3,1}$, it is covered all the possibilities of combinations of CV's in the formation rule correspondent to the first level of the bottom zone.

□

5.3 A procedure to find T_k

Let us recall that we are searching for T_k, given a $2-lsirup$ P and a set of fds F.

We know, by definition of stabilized trees, that if T_n is a stabilized tree, then $topMost(ChaseP_F(T_l))$, $l > n$, is equal to $topMost(ChaseP_F(T_n))$. Notice that T_k does not necessarily have to be a stabilized tree. If there is a tree T_h such that it is not stabilized but its topMost (after its partial chase) is equal to the topMost of the stabilized trees (after their partial chase), then it is sure, by Lemma 3.3.2, that there is not any tree bigger than T_h with a different topMost (after its partial chase). Hence, T_k is the smallest tree such that the topMost of its partial chase is equal to $topMost(ChaseP_F(T_n))$ (where T_n is a stabilized tree).

The procedure to find T_k consists in computing the trees $T_0, T_1, T_2, \ldots$ and chasing them using F, until we find $2\mathcal{N}$ trees from T_n to T_N, $N = n+2\mathcal{N}$, and $n \geq 2\mathcal{N}$, such that all the trees $T_n, \ldots, T_N$ adjust to the same set of formation rules. Then, we can be sure that all those trees are stabilized, and therefore all of them have the same topMost after their partial chase. Therefore, T_k is the smallest tree with such topMost after its partial chase.

Next, we are going to prove that after $2\mathcal{N}$ consecutive trees adjusted to the same set of FR, any bigger tree will be adjusted to the same set of formation rules.

Theorem 5.3.1 Let P be a $2-lsirup$, let F be a set of fds over $EDB(P)$. If $T_n, \ldots, T_{n+2\mathcal{N}}$ ($n \geq 2\mathcal{N}$) are adjusted to the same set of formation rules FR. Then, any tree bigger than $T_{n+2\mathcal{N}}$ is also adjusted to the same set FR (and therefore we can say that such set is $FR_F(P)$).

Proof Let us suppose that there is a tree T_M bigger than $T_{n+2\mathcal{N}}$ such that after its partial chase (with respect to F), T_M does not adjust to FR. Also, let us suppose that T_M is the smallest tree after $T_{n+2\mathcal{N}}$ which is not adjusted to FR.

By Lemma 3.3.1, any equalization produced during the partial chase of any tree smaller than T_M is also produced in the $ChaseP_F(T_M)$. Then, since $ChaseP_F(T_M)$ is not adjusted to FR, $ChaseP_F(T_M)$ should include, at least, one equalization among variables of T_M that produces, at least, a level that is not adjusted to the frs in FR. That is, the reason because T_M does not adjust to FR cannot be the lack of an equalization that is produced in smaller trees.

Let us suppose that such equalization is produced by an equalization chain E:

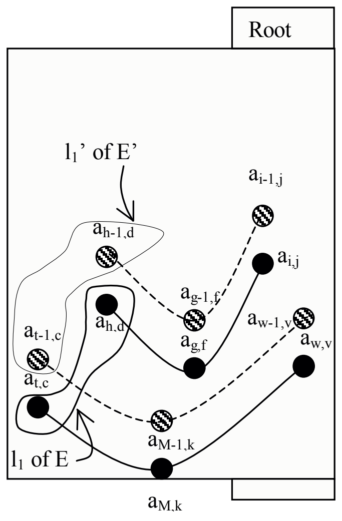


Figure 5.4: Representation of two parallel equalization chains in a tree

$$E = l_1, l_2, \ldots, l_q$$

Note that in E, there should be at least one atom in level M, otherwise such equalization would be done during $ChaseP_F(T_{M-1})$ (which has all its levels adjusted to FR).

We have two cases, when E does not include any atom in the first level, and when it does:

Case 1 If there is one or more atoms of E in level M, and the rest of atoms are not in the level one.

Let E' be the equalization chain constructed as follows. For any atom $a_{x,y}$ in l_i (of E), then l'_i (of E') contains the atom $a_{x-1,y}$ (we know that it exists by Lemma 4.3.1). That is, there is an equalization chain E' that is parallel to E and where for any atom in E, E' has an atom in the same relative position as E but in the previous level.

$$E' = \{l'_1, l'_2, \ldots, l'_q\}$$

E' exists, since E does not include any atom in the first level, but in fact, there are as many parallel equalization chains, as levels of difference are between the atom of E that is in the smallest level and the first level.

E' equates variables that are in the same columns as the variables equated by E but in the previous level.

E' is produced also in $ChaseP_F(T_{M-1})$, and all levels of T_{M-1} adjusts to the formation rules in FR. Thus, it is easy to see that if the equalizations of E' allow T_{M-1} to be adjusted to the formation rules in FR, then the equalizations of E allow T_M to be adjusted to formation rules in FR.

Case 2 If in E there are atoms in level M and in level one.

Let a_p and a_u be the first two atoms of E that are in levels 1 and M, respectively. Let $a_{f,c}$ and $a_{h,i}$ be the atoms (in E) such that $a_{f,c}[R]$ is equal to $a_u[L]$ and $a_{h,i}[R]$ is equal to $a_p[L]$. That is, the variables in the right-hand side (defined by a fd in F) of $a_{f,c}$ and $a_{h,i}$ are the same as the variables in the left-hand side (defined by a fd in F) of a_p and a_u, respectively[c].

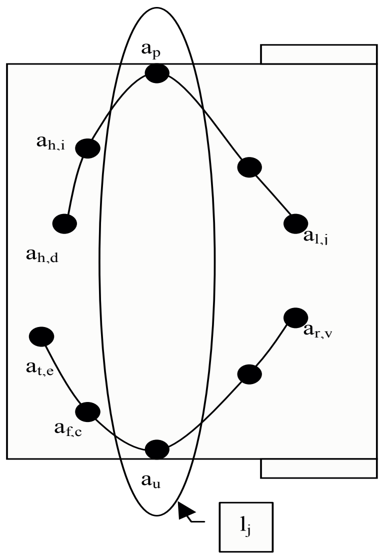
We are going to consider two cases, when there are CV's in E, and when there are no CV's in E.

- *If there are not CV's in E.*

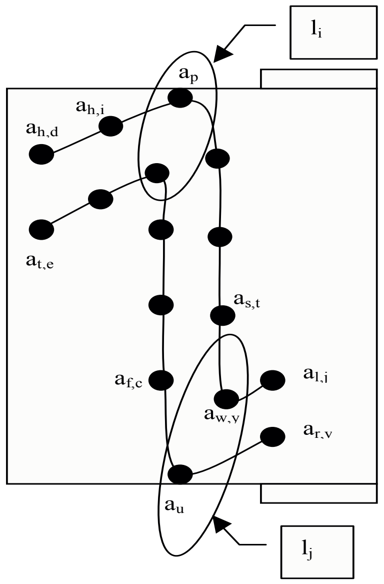
  Again, we have two cases, when a_p and a_u are in the same group of E, and when they are in different groups (as showed in Figure 5.5 cases (a) and (b), respectively):

  - *If a_p and a_u are in the same group l_j of E (Figure 5.5(a))*
    Note that if E is an equalization chain that produces an equalization of variables that generates levels that are not adjusted to FR, then it is not possible that in $ChaseP_F(T_{M-1})$, $a_{u-1}[R]$ (where a_{u-1} is an atom in the same relative position as a_u, but in the previous level) were equated to $a_p[R]$. Otherwise, the equalization of $a_p[R]$ to $a_u[R]$, or any other equalization in E, would generate levels adjusted to FR. This is true, since all levels of T_{M-1} are adjusted to FR and by Lemma 2.6.1.

[c]For the shake of simplicity we do not consider the case that a_u, a_p or both are in l_1. Such cases can be inferred from the cases showed in this proof straightforward.



(a) a_u and a_p in the same group



(b) a_p in a previous group to a_u.

Figure 5.5:

Let us suppose that a_p and a_u are in the group l_j of the equalization chain E, as we can see in Figure 5.5(a). Therefore, $a_{h,i}$ and $a_{f,c}$ are in l_{j-1}.
Since there are not CV's in E, by Lemma 2.6.4, all the variables in E cannot appear in levels separated by more than $\mathcal{N}$ levels. Therefore, $a_{h,i}$ should be in a level smaller or equal to $\mathcal{N}$, and $a_{f,c}$ should be in a level bigger or equal to $M - \mathcal{N}$, given that $a_{f,c}[R]$ is equal to $a_u[L]$ and $a_{h,i}[R]$ is equal to $a_p[L]$.
Thus, in $ChaseP_F(T_{M-(\mathcal{N}+1)})$, the equalization of $a_{f,c}[R]$ and $a_{h,i}[R]$ is not produced, given that $a_{f,c}[R]$ (in $T_{M-(\mathcal{N}+1)}$) does not exist.
However, in $ChaseP_F(T_{M-1})$, the equalization of $a_{h,i}[R]$ and $a_{f,c}[R]$ is produced given that in T_{M-1}, $a_{h,i}$ and $a_{f,c}$ exist (note that we have assumed that a_u is the first atom of E in level M). Therefore, T_{M-1} and $T_{M-(\mathcal{N}+1)}$ do not adjust to the same set of formation rules.
Then, we reached a contradiction since we have supposed

that T_M is the first tree after $T_{n+2\mathcal{N}}$ that is not adjusted to FR. Hence, it is not possible that T_M would not adjust to FR.

– *If a_p and a_u are in different groups of E (Figure 5.5(b))*
Without lost of generality let us assume that a_p is in a group previous to the group where a_u is placed. That is, $a_p \in l_i$ and $a_u \in l_j$ where $i < j$.
Let E_j be the equalization chain $l_1, \ldots, l_{j-1}$. In T_{M-1}, E_j exists since we have supposed that l_j is the first group of E that includes an atom in level M. By hypothesis, E_j does not produce equalizations that generate levels which are not adjusted to FR.
Let us suppose that the variables that are equated in l_{j-1} are the variables in the position defined by the right-hand side (of a certain fd) of $a_{f,c}$ and $a_{s,t}$ [d]. Let us suppose that (in T_M) $a_{f,c}[R]$ is equal to $a_u[L]$ and $a_{s,t}[R]$ is equal to $a_{w,y}[L]$. That is, a_u and $a_{w,y}$ are atoms in l_j.
The equalization produced by $a_{f,c}$ and $a_{s,t}$ is produced in $ChaseP_F(T_{M-1})$, and such equalization is included by hypotheses in the formation rules in FR.
Then, in $Chase_F(T_{M-1})$, $a_{f-1,c}[R]$ is equated to $a_{s-1,t}[R]$ given that in $ChaseP_F(T_{M-2})$, (that is adjusted to FR) $a_{f-1,c}[R]$ is equated to $a_{s-1,t}[R]$, otherwise, the formation rules of level f of T_{M-1} and level $f-1$ of T_{M-2} would be different, and then levels of T_{M-1} would not adjust to FR.
If the equalization of $a_{f,c}[R]$ and $a_{s,t}[R]$ equates (in $ChaseP_F(T_M)$) $a_u[L]$ and $a_{w,y}[L]$, then the equalization of $a_{f-1,c}[R]$ and $a_{s-1,t}[R]$ (in $ChaseP_F(T_{M-1})$) equates $a_{u-1}[L]$[e] and $a_{w-1,y}[L]$ (notice that $a_{w-1,y}$ exists in T_M and in T_{M-1}), otherwise we are again in the already proven case of being a_u and a_p ($a_{w,y}$) in the same group).
Thus, the equalization of $a_u[R]$ and $a_{w,y}[R]$ generates levels adjusted to FR sice the equalization of $a_{u-1}[R]$ and $a_{w-1,y}[R]$ in $ChaseP_F(T_{M-1})$ generates levels adjusted to FR. Then any other equalization in the groups of E after l_j, by Lemma 2.6.1, also adjusts to FR.

[d]We assume that are two atoms in l_{j-1}, but there can be others

[e]An atom in the same relative position as a_u but in the previous level, i.e. in level $M-1$.

- *The equalization chains have CV's*

  Previous cases were based in the fact that two atoms that contain an acyclic variable cannot be separated by more than $\mathcal{N}$ levels.

  If there are CV's in E but the variables in $a_{f,c}[R]$ and $a_{h,i}[R]$ are not cyclic variables, the proofs of the previous cases are still valid.

  We are going to focus our attention in $a_{f,c}$, the atom in l_{l-1} that has a variable in common with a_u. However, the case of $a_{h,i}$ can be proven in the same way.

  Let us suppose that the group l_j is the group in E that includes a_u and let us suppose that $a_{w,y}$ is an atom in l_j ($a_{w,y}$ can be a_p). Let us suppose that $a_{f,c}$ and $a_{s,t}$ ($a_{s,t}$ can be $a_{h,i}$) are atoms in l_{j-1} such that $a_{f,c}[R]$ is equal to $a_u[L]$ and $a_{s,t}[R]$ is equal to $a_{w,y}[L]$.

  Since $a_{f,c}[R]$ is a cyclic variable, then $a_{s,t}[R]$, in $ChaseP_F(T_M)$, by Lemma 2.6.2, is equated to the variables in the $R-th$ position of several atoms. Atoms that are in the same relative position as $a_{f,c}$ but separated from $a_{f,c}$ a number of levels that is multiple of $\mathcal{N}$. Therefore such equalizations (in T_M) will produce the equalization of the variables in the R-th position (defined by the corresponding fd) of $a_{w,y}$, a_u and all the atoms that are in the same relative position as a_u but separated from a_u a number of levels that is multiple of $\mathcal{N}$.

  $ChaseP_F(T_{M-1})$ equates $a_{f,c}[R]$ and $a_{s,t}[R]$, given that we supposed that a_u is the first atom in E that is in level M. Then, the chase of T_{M-1} generates the equalization of the right-hand sides of $a_{w,y}$ and all the atoms that are in the same relative position as a_u but separated from a_u a number of levels multiple of $\mathcal{N}$ (but not a_u, since it does not exist in T_{M-1}). Therefore the equalization of $a_u[R]$ and $a_{w,y}[R]$ in T_M does not generates levels which are not adjusted to FR.

Note that we have seen that the individual equalization chains do not break the formation rules FR, then it is easy to see that the mix of the equalization chains does not break the formation rules FR either. □

Now, we are ready to provide a procedure in order to find T_k (see Figure 5.6).

```
**Procedure** Find $T_k$ (P, F)
**Input:** A $2-lsirup$ $P$ and a set of fds over $EDB(P)$
**Output:** $T_k$ and $topMost(ChaseP_F(T_k))$
    Let $i = 2\mathcal{N}$
    Let continue=true
    While continue
        Let continue=false
        Let j=i
        While $j \leq i + \mathcal{N}$
            Extract the formation rules of $T_j$ and $T_{j+\mathcal{N}}$
            If the formation rules of the first $U$ levels of $T_j$
              are equal to the formation rule of the first $U$ levels of $T_{j+\mathcal{N}}$ and
              the formation rules of the last $B$ levels of $T_j$
              are equal to the formation rules of the last $B$ levels of $T_{j+\mathcal{N}}$ and
              the formation rules of the remaining intermediate levels of $T_j$
              are equal to the formation rules of the remaining intermediate
              levels of $T_{j+\mathcal{N}}$
            then
              Let j=j+1
            else
              Let continue=true
              Let i=i+1
              exitWhile
            endIf
        endwhile
    endWhile
    Output the smallest tree $T_k$ such that $topMost(ChaseP_F(T_k))$ is equal
           to $topMost(ChaseP_F(T_i))$ and $topMost(ChaseP_F(T_k))$
```

Figure 5.6: A procedure to find T_k

5.4 The algorithm to compute P'

In order to compute the chase of a datalog program, first we have to find T_k using the procedure shown in Figure 5.6.

Once we found T_k we are ready to obtain the *chase* of a $2-lsirup$ P with respect to a set of fds F using the algorithm shown in Figure 5.7. This algorithm is very similar to the algorithm introduced by Lakshmanan and Hernández [LH91].

However, we will see later that our procedure pushes more equalizations than their algorithm since we use a more powerful procedure to perform the chase.

Basically, the algorithm outputs the frontier of the (partially) chased trees smaller than T_k and the $topMost(ChaseP_F(T_k))$.

```
Chase of a datalog program
Input A $2-lsirup$ $P$, a set of fds $F$ over $EDB(P)$, $T_k$ (for $P$ and $F$) and
      $topMost(ChaseP_F(T_k))$
Ouput A set of rules that conforms a datalog program $P'$, which is $Chase_F(P)$

  For any tree $T_i$ with $i < k$
      Output the rule $frontier(ChaseP_F(T_i))$
  EndFor
  Output the rule $topMost(ChaseP_F(T_k))$
```

Figure 5.7: Algorithm of the chase of a datalog program

The improvement with respect to Lakshmanan and Hernández's algorithm comes from the terminating condition. Their algorithm terminates when it finds two consecutive trees (one tree with one level more than the other) with the same topMost after the partial chase. However, it is clear that after two consecutive trees with the same topMost after the partial chase, there would be bigger trees that may introduce more equalizations in the topMost after the partial chase of those trees. Our algorithm terminates in a tree T_k such that it is sure that there is not any tree bigger than T_k introducing more equalizations in the topMost after its partial chase. Hence, our algorithm introduces more equalities in the recursive rule of the new program, and therefore it will be cheaper to evaluate than the one in [LH91].

Note that in our algorithm, it is used the partial chase of trees, that is, in the fd applications it is not considered the atom in the last level. If we would include this atom in the computation of the chase of the tree, we could not find T_k, that is, the Lemma 3.3.1 would not be true. However there is a tree (say T_r) in $trees(P)$ such that all the trees bigger than T_r have a topMost after the *chase of the tree*[f] in a finite set of possible topMosts.

Gonzalez-Tuchmann [GT95] tackled that problem. He defined a "chase" of a datalog program considering the atom in the last level, unfortunately he did not provide an algorithm to compute such optimized version of a datalog program.

Our procedure may obtain more equalizations that Lakshmanan and Hernández's algorithm, but may be less than Gonzalez-Tuchmann. However, we provide a method to compute our optimized program (Gonzalez-

[f]That is, considering the atom in the last level.

Tuchmann did not provide a method to compute his chase), and we prove that the program P' in the output of our algorithm is equivalent to the original one P when both are applied over databases satisfying a set of fds.

Example 5.4.1 Let $P = \{r_0, r_1\}$ be:
$r_1 :\ p(X, Y, Z, W)\ :-\ a(Y, Z), b(X, W), p(X, X, Y, Z)$
$r_0 :\ p(X, Y, Z, W)\ :-\ e(X, Y, Z, W)$

and let F be $a : \{1\} \rightarrow \{2\}$.

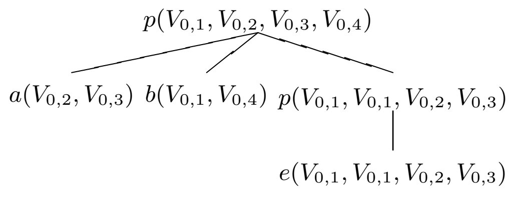

$p(V_{0,1}, V_{0,2}, V_{0,3}, V_{0,4})$
$a(V_{0,2}, V_{0,3})$
$b(V_{0,1}, V_{0,4})$
$p(V_{0,1}, V_{0,1}, V_{0,2}, V_{0,3})$
$e(V_{0,1}, V_{0,1}, V_{0,2}, V_{0,3})$

Figure 5.8: T_1

The partial chase with respect to F of T_1 (in Figure 5.8) does not make any equalization, thus $ChaseP_F(T_1)$ is equal to T_1. Now, let us check T_2 :

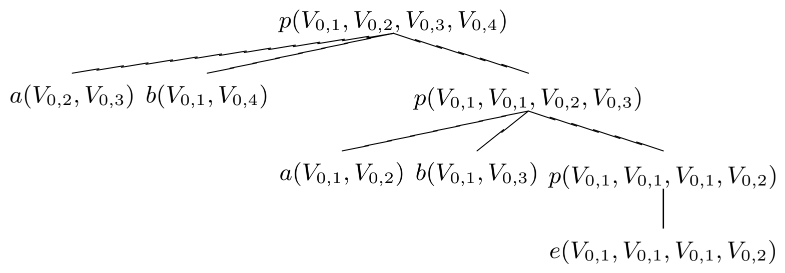

$p(V_{0,1}, V_{0,2}, V_{0,3}, V_{0,4})$
$a(V_{0,2}, V_{0,3})$
$b(V_{0,1}, V_{0,4})$
$p(V_{0,1}, V_{0,1}, V_{0,2}, V_{0,3})$
$a(V_{0,1}, V_{0,2})$
$b(V_{0,1}, V_{0,3})$
$p(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,2})$
$e(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,2})$

Figure 5.9: T_2

$ChaseP_F(T_2)$ does not make any equalization among the variables in T_2 either, since the atoms defined over a have different variables in the left-hand side of $a : \{1\} \rightarrow \{2\}$. Therefore, $topMost(ChaseP_F(T_2))$ and $topMost(ChaseP_F(T_1))$ are equal. In this point Lakshmanan and Hernández's algorithm terminates, given that two consecutive trees have the same topMost after their partial chase. Thus, their algorithm, in this case, outputs the input program without any equalization, and thus it does not optimize the input program.

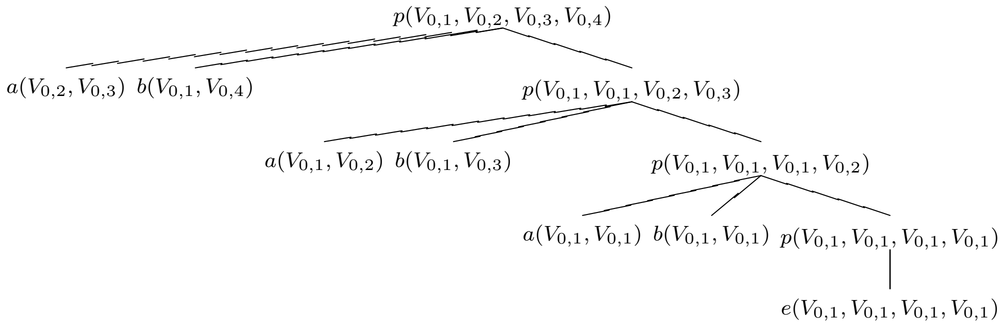

$p(V_{0,1}, V_{0,2}, V_{0,3}, V_{0,4})$
$a(V_{0,2}, V_{0,3})$
$b(V_{0,1}, V_{0,4})$
$p(V_{0,1}, V_{0,1}, V_{0,2}, V_{0,3})$
$a(V_{0,1}, V_{0,2})$
$b(V_{0,1}, V_{0,3})$
$p(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,2})$
$a(V_{0,1}, V_{0,1})$
$b(V_{0,1}, V_{0,1})$
$p(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,1})$
$e(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,1})$

Figure 5.10: T_3

However, let us take a look to T_3 (in Figure 5.10). We find two atoms $a(V_{0,1}, V_{0,2})$ and $a(V_{0,1}, V_{0,1})$ with the same variable in the left-hand side of $a : \{1\} \rightarrow \{2\}$, and thus, $V_{0,2}$ is equated to $V_{0,1}$. Then, the $topMost(ChaseP_F(T_3))$ changes with respect to the $topMost(ChaseP_F(T_2))$, decreasing the number of different variables.

In fact, it can be checked that T_3 is the first stabilized tree, thus any bigger tree would have the same topMost after the partial chase.

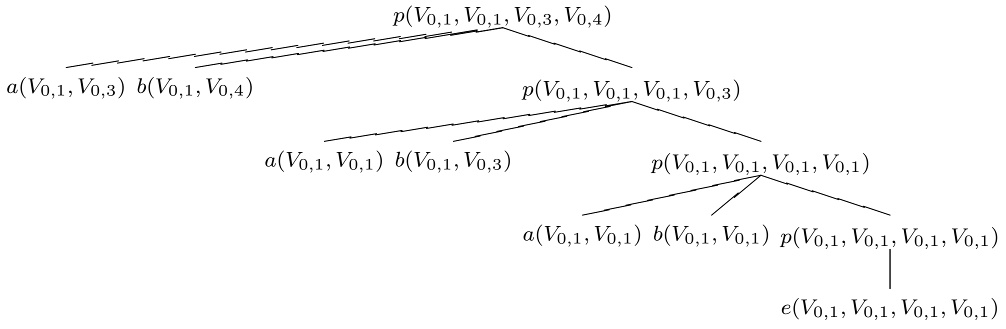

$p(V_{0,1}, V_{0,1}, V_{0,3}, V_{0,4})$
$a(V_{0,1}, V_{0,3})$
$b(V_{0,1}, V_{0,4})$
$p(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,3})$
$a(V_{0,1}, V_{0,1})$
$b(V_{0,1}, V_{0,3})$
$p(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,1})$
$a(V_{0,1}, V_{0,1})$
$b(V_{0,1}, V_{0,1})$
$p(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,1})$
$e(V_{0,1}, V_{0,1}, V_{0,1}, V_{0,1})$

Figure 5.11: $ChaseP_F(T_3)$

Then, our algorithm produces a program formed by the frontier of $ChaseP_F(T_0), ChaseP_F(T_1), ChaseP_F(T_2)$, and $topMost(ChaseP_F(T_3)$:

$s_0 :\ p(X, Y, Z, W)\ :-\ e(X, Y, Z, W)$
$s_1 :\ p(X, Y, Z, W)\ :-\ a(Y, Z), b(X, W), e(X, X, Y, Z)$
$s_2 :\ p(X, Y, Z, W)\ :-\ a(Y, Z), b(X, W), a(X, Y), b(X, Z), e(X, X, X, Y)$

$s_3: \ p(X, X, Z, W) \ :- \ a(X, Z), b(X, W), p(X, X, X, Z)$

Notice that the recursive rule has less different variables than the original one. However, Lakshmanan and Hernández's algorithm would not introduce any change in the original program.

□

5.4.1 A note on complexity

Although we are not specially concerned about the complexity of our algorithms since the computation of the chase can be done in compile time without taking into account the databases to which the new program will be applied, we are going to illustrate that this algorithm has a very low computational pay load.

In order to compute the chase of a datalog program P with respect to a set of fds F, first we have to compute $\mathcal{N}$. A naive implementation of an algorithm that computes the length of the chunks can take exponential time in the arity of the IDB atom. Although better algorithms can be developed to compute such lengths, it is not very important, since the arity of the IDB atom would be typically very small in comparison with the extent of the relations to which the program will be applied.

Once we have compute the chunks length, the computation of $\mathcal{N}$ takes linear time, using the Euclidean algorithm (there are also improvements of the Euclidean algorithm).

Next we have to find T_k, due to the nature of linear sirups that only have one recursive rule, it is easy to see that the maximum difference in number of levels between two atoms in a atom chain is $\mathcal{N} \times a$, where a is the number of EDB atoms in the recursive rule of the program.

Therefore, once the procedure to find a stabilized tree reaches a tree with $\mathcal{N} \times a$ levels, after $2\mathcal{N}$ more trees, it is easy to see that the procedure terminates.

Thus, the number of cycles performed by the procedure that finds a stabilized tree grows in polynomial time on the size of r_1. Notice that $\mathcal{N}$ also depends on the size of r_1 (in fact, $\mathcal{N}$ is the least common multiplier of the chunk lengths of the expansion graph).

Thus, since the chase of a tree (using only fds) can be computed in polynomial time on the size of the tree [AHV95], and the biggest tree that can be computed is $T_{2\mathcal{N}+\mathcal{N}\times a}$, then the chase of a datalof program can be computed in a tractable time.

5.4.2 Equivalence of P' and P

Next, we show that the output of the *Chase of datalog programs* is equivalent to the original program when both are applied to databases satisfying the set of fds used to obtain the chase.

Theorem 5.4.1 Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. The $Chase_F(P)$ (P') is equivalent to P when both are evaluated over databases in $SAT(F)$.

Proof It follows from Lemmas 5.4.1 and 5.4.2 given below. □

Lemma 5.4.1 Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. Then $P' \subseteq_{SAT(F)} P$.

Proof

Let NR be the set of non-recursive rules in P', those coming from the $frontier(ChaseP_F(T_i))$ and $i < k$.

Let s be a rule in NR, by definition, $s = frontier(ChaseP_F(T_i))$ for some tree T_i in $trees(P)$. Then, by Lemma 3.1.1 $\{s\} \subseteq_{SAT(F)} P$.

Let t be recursive rule in P', the one which is the $topMost(ChaseP_F(T_k))$. Therefore, $t = \phi(topMost(T_k))$, where ϕ is the substitution defined by $ChaseP_F(T_k)$. Given that $topMost(T_k) = r_1$, then $t = \phi(r_1)$, therefore by Lemma 3.1.1 $\{t\} \subseteq_{SAT(F)} \{r_1\}$.

Hence, we have shown that for all the rules r in P', $\{r\} \subseteq_{SAT(F)} P$. □

Lemma 5.4.2 Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. Then $P \subseteq_{SAT(F)} P'$.

Proof

We are going to prove that any fact produced by a tree T in $trees(P)$ when T is applied to a database d in $SAT(F)$ is also produced by P', when P' is applied to d.

Let d be a database in $SAT(F)$, and assume that q is in $T(d)$, we are going to prove that q is in $P'(d)$. We prove by induction on the index (number of levels) of the tree T in $trees(P)$ that if q is in $T(d)$ then q in $P'(d)$.

Basis i=0, q is in $T_0(d)$ ($T_0 = tree(r_0)$). Then q is in $P'(d)$, since, r_0 is also in P'[g].

[g]Note that $topMost(ChaseP_F(T_0))$ cannot be isomorphic to any other topMost of a tree in $trees(P)$, then, it produces a non recursive rule in P' that is equal to the rule r_0 of P.

Induction hypothesis (IH): Let $q \in T_i(d)$, $1 \leq i < j$ and $q \in P'(d)$.
Induction step: i=j.

q is in $T_j(d)$. Assume q is not in any $T_m(d)$, $0 \leq m < j$, otherwise the proof follows by the IH. Thus, there is a substitution θ such that q is $\theta(p_j)$, where p_j is the root of T_j and where $\theta(t_l) \in d$ for all the leaves t_l of T_j. Therefore, q is also in $\{frontier(T_j)\}(d)$.

We have two cases:

Case 1: $frontier(ChaseP_F(T_j))$ is one of the non-recursive rules of P'. Then by Lemma 3.1.1 q is in $P'(d)$.

Case 2: $frontier(ChaseP_F(T_j))$ is not one of the non-recursive rules of P'.

$q \in T_j(d)$ thus, by Lemma 3.1.1 $q \in \{ChaseP_F(T_j)\}(d)$ (assuming that $d \in SAT(F)$). Let γ be the substitution defined by the $ChaseP_F(T_j)$.

Let T_{sub} be the subtree of T_j that is rooted in the node at the first level of T_j that is the recursive atom at that level. T_{sub} has one level less than T_j, therefore T_{sub} is isomorphic to T_{j-1}.

Let q_{sub} be an atom in $T_{sub}(d)$, since T_{sub} is isomorphic to T_{j-1}, then q_{sub} is in $T_{j-1}(d)$. Hence, by IH $q_{sub} \in P'(d)$.

It is easy to see that $q \in \{topMost(ChaseP_F(T_j))\}(d \bigcup q_{sub})$, that is, $q \in \{\gamma(r_1)\}(d \bigcup q_{sub})$.

By construction of P', in P' there is a rule $s_t = \gamma(r_1) = topMost(ChaseP_F(T_j))$. We have already shown that q_{sub} is a fact in $P'(d)$. Therefore, since $s_t(d \cup q_{sub})$ obtains q thus we have proven that if q is in $T_j(d)$ then q is also in $P'(d)$. $\square$

5.4.3 Generalization to *lsirups*

Since the partial chase only takes into account the EDB atoms in levels before the last one, that is, it only considers the atoms produced by the expansions produced with the recursive rule, then all the results referred to the partial chase of $2 - lsirups$ are applicable also to *lsirups*.

We have to be more careful in the computation of the alternative program $Chase_F(P)$ (for a given *lsirup* P and a set of fds F over $EDB(P)$). If we consider *lsirups* there can be several non-recursive rules. Thus, in that case, for each non-recursive rule in P we would have to apply the algorithm in Figure 5.7 having as input the recursive rule and such non-recursive rule. The set of rules obtained after the application of the algorithm to all the non-recursive rules is the optimized program.

Therefore, it is easy to see that with just taking into account the non-recursive rules, the chase of datalog programs can be easily extended.

Chapter 6

Optimization using cyclic topMost

In the previous chapter we showed an algorithm to optimize a $2 - lsirup$ based on the use of the partial chase of trees. This type of chase of trees does not consider the atom in the last level of the tree (the level resulting after the application of the non-recursive rule). However, there are programs that produce trees where the partial chase does not make any equalization whereas the chase of trees does. Such situation is due to the presence of the same variable (or variables) in the position(s) defined by the left-hand side of a fd in the atom in the last level and in other atom. That is, the atom in the last level is in a LHCSA. Then, such LHCSA produces an equalization of variables, and then, such equalization may start equalization chains that produce equalizations through all the tree.

We have pointed out the worst case, that is, the partial chase of trees does not introduce equalizations whereas the chase of trees does. In any case, it is clear that the chase of trees may introduce more equalizations than the partial chase of trees, even if the partial chase of trees introduces some equalizations. This claim is true since, as we saw, the atom in the last level may produce equalizations in conjunction with other atoms of the tree.

Therefore, it seems that with the partial chase of trees we are loosing equalizations that can be obtained. However, we have pointed out that using the chase of trees, if we compute the chase of trees of $T_0, T_1, T_2, \ldots$, there is not a tree such that all trees bigger than it have the same topMost after their chase. Thus the algorithm showed in the previous chapter is not valid using the chase of trees.

Gonzalez-Tuchmman [GT95] tackled that problem. He defined a “chase” of a datalog program using the chase of trees, but unfortunately he did not provide an algorithm to compute such optimized version of a datalog program. Thus, our challenge was the definition of a method to "chase" programs using the chase of trees.

In this chapter, we present our second algorithm to optimize linear sirups *the cyclic chase of datalog programs*. It does not use the partial chase (like the algorithm showed in the previous chapter), it uses the chase of trees (a "complete" chase of trees) and the *cyclic topMost*. Although, the *chase of datalog programs* (showed in previous chapter) and the cyclic chase have the same background, there are big differences. For example, in the cyclic chase, equalizations in recursive rules are limited to $CV's$. However, we will see later that the cyclic chase has other advantages with respect to the chase of datalog programs.

Although we present the results of this chapter for $2-lsirups$, our results can be extended to $lsirups$.

On the other hand, the output of the chase of datalog programs can be provided as input of this algorithm. Thus, the two algorithms (the chase of datalog programs and the cyclic chase of datalog programs) can be combined.

The outline of this chapter is as follows. In Section 6.1 we introduce the algorithm of the cyclic chase, we proof its correctness and we discuss its complexity.

6.1 An algorithm to optimize a $2 - lsirup$

In Figure 6.1, the algorithm to compute the cyclic chase of a datalog program P it is shown. Given a $2 - lsirup$ P and a set F of fds over $EDB(P)$, the $CChase_F(P)$ obtains a program P' equivalent to P when both are applied to databases in $SAT(F)$. P' is cheaper to evaluate than the original one due to the equalizations of variables (as it was show in Section 3.1.1).

The algorithm of the cyclic chase begins computing $T_0, T_1, T_2, \dots$ until it finds $\mathcal{N}$ consecutive trees $T_n, \dots, T_{n+\mathcal{N}}$ such that for any tree T_i, where $n < i \leq n + \mathcal{N}$, $CtopMost_F(T_i)$ is equal to $CtopMost_F(T_{i-\mathcal{N}})$.

Then, the algorithm outputs the different rules found in:

$$CtopMost_F(T_n), \dots, CtopMost_F(T_{n+\mathcal{N}})$$

and the frontier of the chase of the trees in $T_0, \dots, T_{n-\mathcal{N}-1}$.

Notice that in this case the output may have more than one recursive rule, since, as we have pointed out, with the chase of trees, there is not a tree such that all trees bigger than it have the same topMost after the chase.

Also, notice that the recursive rules only have equalizations among CV's.

```
CChase
Input: P: a $2 - lsirup$ and F a set of functional dependencies over EDB(P)
Output: $CChase_F(P)$ that is the optimized program $P'$

Let $n = \mathcal{N}$ for $P$
Let $i = 2n$
Let continue=true
While continue
        Let continue=false
        For $j = i - n$ to $i$
            If $CtopMost_F(T_j)$ is not equal to $CtopMost_F(T_{j-n})$
              Let continue=true
              breakFor
            endIf
        endFor
        If continue
           Output $frontier(Chase_F(T_{i-2n}))$
           Let i=i+1
        endIf
endWhile
For $j = i - n$ to $i$
     If $CtopMost_F(T_j)$ is not isomorphic to any previously output rule
        Output $CtopMost_F(T_j)$
     endIf
endFor
```

Figure 6.1: Cyclic Chase

Example 6.1.1 Let $P = \{r_0, r_1\}$ be:

r_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
r_1: $p(X,Y,Z,A,B,C) :- e(Y,X,Y,C,A,D), p(Z,X,Y,B,C,D)$

Let F be $\{\ e : \{6\} \rightarrow \{1\},\ e : \{6\} \rightarrow \{4\}\}$. $\mathcal{N}$ for P is 3.

Using P, T_0 is shown in Figure 6.2. The $Chase_F(T_0)$ does not introduce any equalization.

T_1 is shown in Figure 6.3. The partial chase of T_1 with respect to F does not produce any equalization. However, the $Chase_F(T_1)$ equates two set of variables. Using the fd $e : \{6\} \rightarrow \{1\}$, we can observe that $e(Y,X,Y,C,A,D)$ and $e(Z,X,Y,B,C,D)$ have the same variable in the position defined by the left-hand side of the fd. Thus, the chase equates Y and Z.

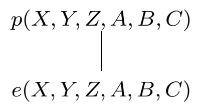


Figure 6.2: T_0

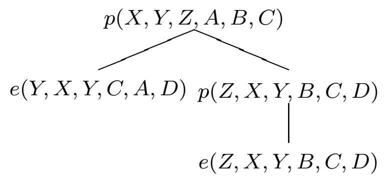


Figure 6.3: T_1

Using the fd $e : \{6\} \rightarrow \{4\}$, again, we can observe that $e(Y,X,Y,C,A,D)$ and $e(Z,X,Y,B,C,D)$ have the same variable in the position defined by the left-hand side of such fd. Thus, $Chase_F(T_1)$ equates C and B, resulting the tree in Figure 6.4.

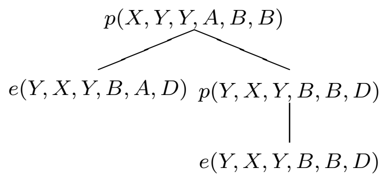


Figure 6.4: $Chase_F(T_1)$

Now, let us check T_2 (in Figure 6.5). The partial chase would not introduce equalizations in this case either. In fact, the partial chase, with the program and fds of this example, does not introduce equalizations in any tree of $trees(P)$. Nevertheless, $Chase_F(T_2)$ introduces equalizations, as shown in Figure 6.6. Therefore, in this example the cyclic chase is the best choice to optimize the program.

In this example, any tree bigger than T_3 has the same cyclic topMost (with respect to F) as T_3. Then, the algorithm terminates in T_8, when it founds $\mathcal{N}$ trees (T_6, T_7 and T_8) with the same cyclic topMost as it correspondent tree with $\mathcal{N}$ less levels (T_3, T_4 and T_5, respectively). Therefore, the output program is formed by the frontier of the chase of trees smaller than T_3, (that is, $frontier(Chase_F(T_0))$, $frontier(Chase_F(T_1))$,

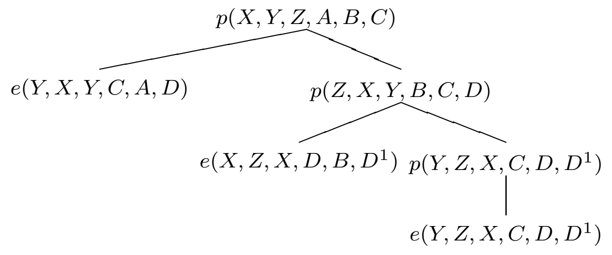


Figure 6.5: T_2

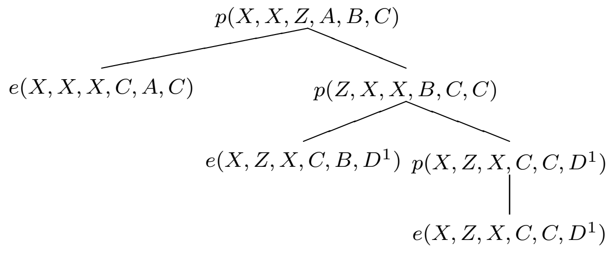


Figure 6.6: $Chase_F(T_2)$

and $frontier(Chase_F(T_2)))$ and $CtopMost_F(T_3)$, the only recursive rule (since in $CtopMost_F(T_3),\ldots,CtopMost_F(T_8)$, this is the only different rule).

$CChase_F(P)$:
s_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
s_1: $p(X,Y,Y,A,B,B) :- e(Y,X,Y,B,A,D), e(Y,X,Y,B,B,D)$
s_2: $p(X,X,Z,A,B,C) :- e(X,X,X,C,A,C), e(X,Z,X,C,B,D^1), e(X,Z,X,C,C,D^1)$
s_3: $p(X,X,X,A,B,C) :- e(X,X,X,C,A,D), p(X,X,X,B,C,D)$

The correspondence is the obvious, s_0 is $frontier(Chase_F(T_0))$, s_1 is $frontier(Chase_F(T_1))$, s_2 is $frontier(Chase_F(T_2))$ and finally, s_3 is $CtopMost_F(T_3)$.

Although, in this case only one recursive rule is output by the cyclic chase, it is not always the case. Sometimes, several recursive rules can be obtained due to the presence of different cyclic topMosts in the last $\mathcal{N}$ chased trees when the algorithm terminates.

Let us consider another example using the same program P but with a new set of functional dependencies $Q = \{\ e : \{6\} \rightarrow \{1\}\}$.

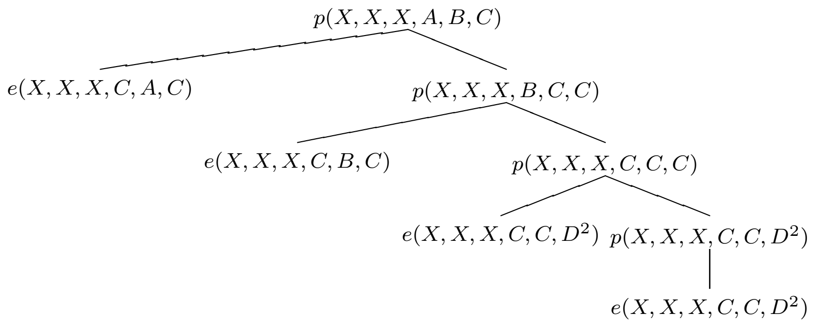


Figure 6.7: $Chase_F(T_3)$

T_0 and $Chase_Q(T_0)$ are the same as the tree in Figure 6.2. However, the chase of T_1 (in Figure 6.3) with respect to F is different from $Chase_Q(T_1)$, as can be seen in Figure 6.8.

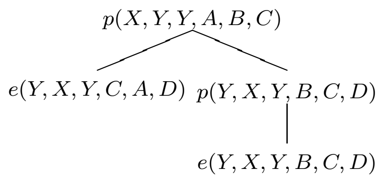


Figure 6.8: $Chase_Q(T_1)$

As it can be observed, the equalization of C and B produced in the $Chase_F(T_1)$ is not produced in $Chase_Q(T_1)$.

$Chase_Q(T_2)$ is showed in Figure 6.9. In this case, the equalization of D and C (produced in $Chase_F(T_2)$) is not produced.

Now, let us check $Chase_Q(T_3)$ (in Figure 6.10). In comparison with $Chase_F(T_3)$ only one of the equalizations found in such a tree is present in $Chase_Q(T_3)$, the equalization of X and Z.

$Chase_Q(T_4)$ only equates Z and Y, the same equalization as in the case of $Chase_Q(T_1)$. Since Z and Y are CV's, then the $CtopMost_Q(T_1)$ is equal to the $CtopMost_Q(T_4)$.

If we continue chasing trees, we find that $CtopMost_Q(T_2)$ is equal to the $CtopMost_Q(T_5)$ and that $CtopMost_Q(T_3)$ is equal to the $CtopMost_Q(T_6)$. In this point the algorithm finds that the last $\mathcal{N}$ trees (T_4, T_5 and T_6) have the same cyclic topMost as T_1, T_2 and T_3, respectively. Then, the algorithm terminates outputting the $frontier(Chase_Q(T_0))$,

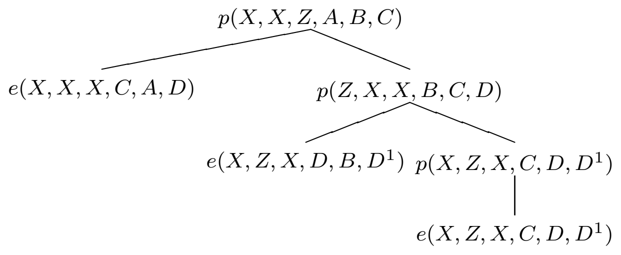


Figure 6.9: $Chase_Q(T_2)$

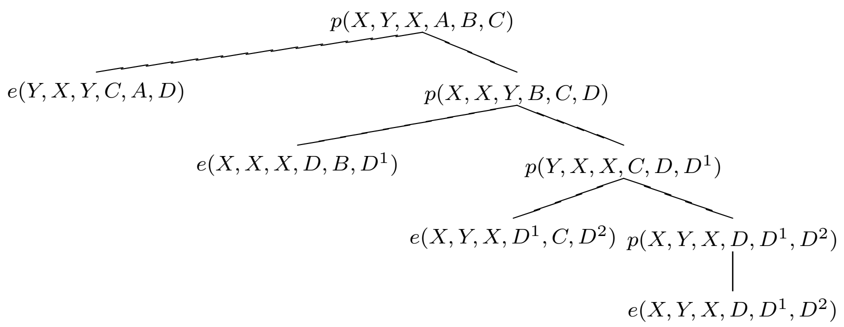


Figure 6.10: $Chase_Q(T_3)$

$CtopMost_Q(T_1)$, $CtopMost_Q(T_2)$ and $CtopMost_Q(T_3)$ (since these three rules are different).

Therefore, the $CChase_Q(P)$ is:

s_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
s_1: $p(X,Y,Y,A,B,C) :- e(Y,X,Y,C,A,D), p(Y,X,Y,B,C,D)$
s_2: $p(X,X,Z,A,B,C) :- e(X,X,X,C,A,D), p(Z,X,X,B,C,D)$
s_3: $p(X,Y,X,A,B,C) :- e(Y,X,Y,C,A,D), p(X,X,Y,B,C,D)$

□

6.1.1 Termination of the cyclic chase

In order to prove that the algorithm in Figure 6.1 terminates, we have to prove that if θ_i^c is the substitution defined by $CtopMost_F(T_i)$ and θ_w^c is the substitution defined by $CtopMost_F(T_w)$, where $w = i + I\mathcal{N}$ and I is an integer such that $I > 0$, then $\theta_i^c \subseteq \theta_w^c$.

Once we have proven that $\theta_i^c \subseteq \theta_w^c$, it is easy to see that the algorithm terminates. Observe that the topMost of any tree in $trees(P)$, except T_0, is r_1, thus since $\theta_i^c \subseteq \theta_w^c$ and the set of variables in r_1 is finite, then the algorithm terminates, in the extreme case, when all the variables in the topMost are equated.

In fact, if there are n different variables in r_1, observe that in the worst case, the maximum number of chased trees inspected by the algorithm will be the initial $2\mathcal{N}$ trees, plus $\mathcal{N} \times n$. Such case considers that each execution of the first for loop sets the variable `continue` to `TRUE` due to only a new equalization. Since, in order to terminate, the algorithm has to find that the last $\mathcal{N}$ chased trees have an equal cyclic topMost to its correspondent chased tree with $\mathcal{N}$ less levels, then by Lemma 6.1.1 (proven below), the maximum number of iterations of the while loop is $\mathcal{N} \times n$.

Lemma 6.1.1 Let P be a $2-lsirup$ and F a set of fds over $EDB(P)$. Let T_i and T_w be a pair of trees in $trees(P)$ where T_w has $I\mathcal{N}$ more levels than T_i (I is an integer bigger than 0) and where θ_i^c and θ_w^c are the substitutions defined by $CtopMost_F(T_i)$ and $CtopMost_F(T_w)$, respectively. Then, it is true that $\theta_i^c \subseteq \theta_w^c$.

Proof

The lemma implies that for any tree T_i in $trees(P)$ the equalizations produced by its cyclic topMost will be also produced by the chase of any tree T_w, for all w such that $w = i + I\mathcal{N}$ (for any integer I bigger than 0).

Let T_{sub} be the subtree of T_w formed by the last i levels and rooted by the IDB atom of level $i-1$.

Since $w = i + I\mathcal{N}$ and by Lemma 2.6.1 we can state that T_i is isomorphic to T_{sub}. In addition, by Lemma 2.6.3, any cyclic variable in $a_{k,l}[y]$ in T_i is also placed in $a_{k,l}[y]$ in T_{sub}. Thus, since they are isomorphic and they have the CV's in the same positions, the equalizations among CV's in $Chase_F(T_i)$ and $Chase_F(T_{sub})$ are the same.

Let us call $\emptyset$ the substitution defined by the equalizations in $Chase_F(T_i)$ (and $Chase_F(T_{sub})$) where only CV's are involved. Hence, since T_{sub} is a subtree of T_w, we have proven that any equalization among CV's in $topMost(Chase_F(T_i))$ is also produced in $topMost(Chase_F(T_w))$.

$\square$

Example 6.1.2 Let $P = \{r_0, r_1\}$ be:

r_0: $p(X,Y,Z,A,B,C) :- e(X,Y,Z,A,B,C)$
r_1: $p(X,Y,Z,A,B,C) :- e(Y,X,Y,C,A,D), p(Z,X,Y,B,C,D)$

$\mathcal{N}$ for this program is 3. Let T_{sub} be (shown in Figure 6.12(b)) the subtree of T_4 (shown in Figure 6.11) formed by the last two levels of T_4 and rooted by the IDB atom in level 3. That is, T_{sub} is isomorphic to T_1 (shown in Figure 6.12(a)).

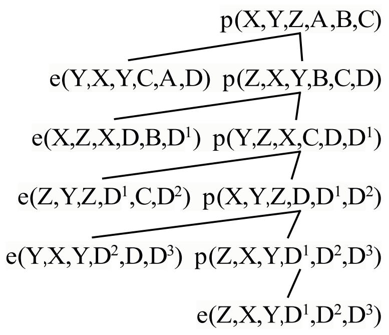
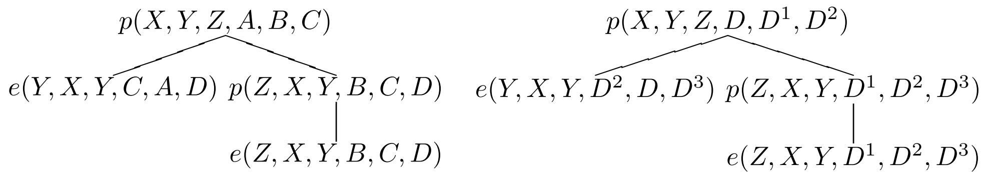

p(X,Y,Z,A,B,C)
e(Y,X,Y,C,A,D) p(Z,X,Y,B,C,D)
e(X,Z,X,D,B,D¹) p(Y,Z,X,C,D,D¹)
e(Z,Y,Z,D¹,C,D²) p(X,Y,Z,D,D¹,D²)
e(Y,X,Y,D²,D,D³) p(Z,X,Y,D¹,D²,D³)
e(Z,X,Y,D¹,D²,D³)

Figure 6.11: T_4


p(X, Y, Z, A, B, C)
e(Y, X, Y, C, A, D) p(Z, X, Y, B, C, D)
e(Z, X, Y, B, C, D)
p(X, Y, Z, D, D¹, D²)
e(Y, X, Y, D², D, D³) p(Z, X, Y, D¹, D², D³)
e(Z, X, Y, D¹, D², D³)

(a) T_1 (b) T_{sub}

Figure 6.12:

Observe that T_1 and T_{sub} are isomorphic. Moreover, they have the CV's placed in the same positions since T_4 has $\mathcal{N}$ levels more than T_1.

Thus, it is easy to see that any equalization (due to the chase) among variables in T_1 has its equivalent equalization in the chase of T_{sub}. In addition, if this equalization is among CV's, then such equalization is also produced (with the same variables) by the chase of T_{sub}.

For example, let $F = \{e : \{3\} \rightarrow \{4\}\ e : \{4\} \rightarrow \{1\}\}$. The $Chase_F(T_1)$ equates, first, C and B, since $e(Y, X, Y, C, A, D)$ and $e(Z, X, Y, B, C, D)$ have the same variable (Y) in the left-hand side of $e : \{3\} \rightarrow \{4\}$. Then, variables in the fourth position of such atoms became equal, and hence, the fd $e : \{4\} \rightarrow \{1\}$ is applied. Finally, after this fd application, Y and Z are equated.

$Chase_F(T_{sub})$ equates, first, D^2 and D^1, and then Y and Z are equated. Since T_{sub} is a subtree of T_4, these equalizations are produced (among others) in $Chase_F(T_4)$, as well.

Therefore, it is easy to see that the equalization of Y and Z in the $topMost(Chase_F(T_1))$ is also produced (among others) in the $topMost(Chase_F(T_4))$.

$\square$

A note on complexity

Although we are not specially concerned about the complexity of our algorithms since the computation of the chase can be done in compile time without taking into account the databases to which the new program will be applied, we are going to illustrate that this algorithm has a very low computational pay load.

In order to compute the cyclic chase of a $2 - lsirup\ P$ with respect to a set of fds F, first we have to compute $\mathcal{N}$. A naive implementation of an algorithm that computes the length of the chunks can take exponential time in the arity of the IDB atom. Although better algorithms can be developed to compute such lengths, it is not very important, since the arity of the IDB atom would be typically very small in comparison with the extent of the relations to which the program will be applied.

Once we have computed the chunks length, the computation of $\mathcal{N}$ takes linear time, using the Euclidean algorithm (there are also improvements of the Euclidean algorithm).

Now, let us focus in the algorithm of Figure 6.1. We have already shown that our algorithm terminates after, at most, $2\mathcal{N} + \mathcal{N} \times n$ cycles (being n the number of different variables in r_1). Thus, the number of cycles performed by the algorithm grows in polynomial time on the size of r_1. Notice that $\mathcal{N}$ also depends on the size of r_1 (in fact, $\mathcal{N}$ is the least common multiplier of the chunk lengths of the expansion graph).

Thus, since the chase of a tree (using only fds) can be computed in polynomial time on the size of the tree [AHV95], and the biggest tree that can be computed is $T_{2\mathcal{N}+\mathcal{N}\times n}$, then the cyclic chase can be computed in a tractable time.

6.1.2 Equivalence of the cyclic chase

Now, we are ready to prove that the output of our algorithm P' is equivalent to the original program P when both are applied over databases satisfying a set of fds F.

Theorem 6.1.1 Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. Then, $P \equiv_{SAT(F)} P'$.

Proof It follows from Lemma 6.1.2 and Lemma 6.1.3 given below. □

Lemma 6.1.2 Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. Then, $P' \subseteq_{SAT(F)} P$.

Proof Let NR be the set of non-recursive rules in P', and let R be the set of recursive rules in P'.

Let s be a rule in NR, by the algorithm in Figure 6.1, $s = frontier(Chase_F(T_i))$ for some tree T_i in $trees(P)$. Then, by Lemma 3.1.1 $\{s\} \subseteq_{SAT(F)} r_1^i \circ r_0$, then $\{s\} \subseteq_{SAT(F)} P$.

Let r be a rule in R. Therefore, $r = \theta_j^c(topMost(T_j))$, where θ_j^c is the substitution defined by $CtopMost_F(T_j)$ and T_j is a tree in $trees(P)$. Given that $topMost(T_j) = r_1$, then $r = \theta_j^c(r_1)$, therefore $r \subseteq r_1$.

Hence, we have shown that for any rule r_i in P', $\{r_i\} \subseteq_{SAT(F)} P$. □

Lemma 6.1.3 Let P be a $2-lsirup$ and let F be a set of fds over $EDB(P)$. Then $P \subseteq_{SAT(F)} P'$.

Proof We are going to prove that any fact q produced by P when P is applied to a database d in $SAT(F)$ is also produced by P', when P' is applied to d.

Let d be a database in $SAT(F)$, and assume that q is in $T(d)$, we are going to prove that q is in $P'(d)$.

We prove by induction on the index (the number of levels) of the tree T in $trees(P)$ that if q is in $T(d)$ then q is in $P'(d)$.

Basis i=0, q is in $T_0(d)$. Then q is in $P'(d)$, since, r_0 is also in P' given that $topMost(Chase_F(T_0))$ is r_0.

Induction hypothesis (IH): Let $q \in T_i(d)$, $1 \leq i < k$ and $q \in P'(d)$.

Induction step: i=j.

q is in $T_j(d)$. Assume q is not in any $T_m(d)$, $0 \leq m < j$, otherwise the proof follows by the IH. Thus, there is a substitution θ such that q is $\theta(p_j)$, where p_j is the root of T_j and where $\theta(t_l) \in d$ for all the leaves t_l of T_j. Therefore, q is also in $\{frontier(T_j)\}(d)$.

We have two cases:

Case 1: $frontier(Chase_F(T_j))$ is one of the non-recursive rules of P'. Then by Lemma 3.1.1 q is in $P'(d)$.

Case 2: $frontier(Chase_F(T_j))$ is not one of the non-recursive rules of P'.

$q \in T_j(d)$ thus, by Lemma 3.1.1 $q \in \{Chase_F(T_j)\}(d)$ (assuming that $d \in SAT(F)$). Let γ be the substitution defined by the $Chase_F(T_j)$.

Let T_{sub} be the subtree of T_j that is rooted in the node at the first level of T_j that is, the recursive atom at that level. T_{sub} has one level less than T_j, therefore T_{sub} is isomorphic to T_{j-1}.

Let q_{sub} be an atom in $T_{sub}(d)$, since T_{sub} is isomorphic to T_{j-1}, then q_{sub} is in $T_{j-1}(d)$. Hence, by IH $q_{sub} \in P'(d)$.

It is easy to see that $q \in \{topMost(Chase_F(T_j))\}(d \bigcup q_{sub})$, that is, $q \in \{\gamma(r_1)\}(d \bigcup q_{sub})$.

By construction of P', in P' there is a rule $s_t = \theta_t^c(r_1)$, where θ_t^c is the substitution defined by the cyclic topMost of some tree.

By lemma 6.1.1 and the definition of the cyclic topMost $\theta_t^c \subseteq \gamma$, therefore if $q \in \{\gamma(r_1)\}(d \bigcup q_{sub})$ then $q \in \{\theta_t^c(r_1)\}(d \bigcup q_{sub})$.

We have already shown that q_{sub} is a fact in $P'(d)$. Therefore, since $s_t(d \cup q_{sub})$ obtains q thus we have proven that if q is in $T_j(d)$ then q is also in $P'(d)$. $\square$

6.1.3 Generalization to *lsirups*

In the case of the cyclic chase, it is used the chase of trees, that takes into account the atom(s) in the last level (the one resulting after the application of a non-recursive rule).

Thus, if the input program P is a *lsirup*, for each non-recursive rule in P, the algorithm in Figure 6.1 should be applied having as input the recursive rule in P and one of the non-recursive rules. At the end (when all non-recursive rules were considered) the whole set of obtained rules is the $CChase_F(P)$.

Therefore, for each non-recursive rule, several non-recursive rules plus several recursive rules may be obtained.

Chapter 7

The FD-FD implication problem

In the framework of the relational model, the chase was used to know when a set of functional (or generalized) dependencies D implies another set of functional (or generalized) dependencies d [MMS79, BV84].

In datalog, the implication problem has also been treated by several researchers [AH88, WY92, GT95, HPB97b, HP97, HPB97a].

Formally, the FD-FD implication problem in datalog is the following. Given a datalog program P, a set of functional dependencies F on the extensional predicates of P, and a set of functional dependencies G on both the extensional and intensional predicates of P, is it true that for all databases d defined on the EDB predicates of P, d satisfies F implies that the output produced by P with d as its input satisfies G? It has been proven by Abiteboul and Hull [AH88] that this problem is undecidable for general datalog programs.

Since then, several researchers started to delimit set of datalog programs where this problem can be decidable.

Gonzalez-Tuchmann [GT95] uses the chase to provide a syntactic condition to solve the implication problem for a class of datalog programs.

Wang and Yuan [WY92] use the chase as a basic component of an algorithm for testing if a set of integrity constraints (including functional dependencies) IC_1 uniformly implies a set of integrity constraints IC_2 in a datalog program, provided that IC_1 is preserved by the program.

The problem tackled by Wang and Yuan is slightly different from the problem treated in this chapter. They test *uniform* implication, whereas we deal with implication (without uniformity).

In this chapter, we introduce a syntactic condition that allows us to decide if a program implies a functional dependency when we use programs in a subclass of linear datalog programs. Besides, provided that the scheme of the database is in Boyce Codd Normal Form (BCNF) with respect to the functional dependencies, we offer a syntactic condition that allows us to identify programs that do not imply a fd.

The outline of this chapter is as follows. In Section 7.1 a subclass of linear datalog programs to which our results apply is defined. In Section 7.3 a test to identify programs that imply a fd is introduced. Finally, in Section 7.4 a test to identify programs that do not imply a fd is presented.

7.1 The class $\mathcal{P}$ of linear datalog programs

The results of this chapter apply to the class $\mathcal{P}$ of linear programs of the following form.

The class is composed of linear datalog programs P definig only one IDB predicate. Furthermore, exactly one of the rules is non-recursive (say r_0), such rule is required to have exactly one atom in its body. Eventually, r_0 must have the following form: $p(\vec{X}) : -e(\vec{X})$, where $\vec{X}$ is a list of distinct variables.

The programs considered by Gonzalez-Tuchmann [GT95] are more restricted than programs in class $\mathcal{P}$. Therefore, his results cannot be applied in our case.

In the programs tackled by Gonzalez-Tuchmann, the predicate name of the atom in the body of the non-recursive rule cannot appear in the recursive rules.

For simplicity, we assume that the terms in the predicates are variables. This assumption do not restrict our results.

7.2 Pivoting

In this section we present a new concept which is very important in this chapter.

Let r be a rule of a program P in class $\mathcal{P}$. We say that a set of argument positions is *pivoting* in r if the positions defined by those argument positions have the same variables and in the same order in the head of such a rule and in either an EDB atom in the body of the rule with the same predicate name as the atom in the body of the non-recursive rule of P or in the IDB atom in the body of r.

Basically, the idea is that if a set of argument positions are pivoting in a rule r, then when r is applied to a database the constants in the positions of the atoms obtained by r will be in the same order (and positions) as they were in ground atoms or previously derived IDB atoms.

That is, the underlying idea is that if the ground atoms satisfy a fd, and the applied rules are pivoting in the positions defined by such a fd, then the derived atoms will satisfy such a fd as well.

Definition 7.2.1 Let r be a rule of a program P in class $\mathcal{P}$ or its chase. Let us denote the atom in the head of r is p_h, and let the predicate name of p_h be p. If the rule is recursive, let p_b be the atom in the body of r whose predicate name is p. Let e_j be an atom in the body of r whose predicate name is e (the EDB predicate name of the atom in the body of the non-recursive rule of P). Then, the argument position i is pivoting in r *if $p_h[i] = e_j[i]$, or $p_h[i] = p_b[i]$. A set N of argument positions is* pivoting in r *if $p_h[N] = e_j[N]$, or $p_h[N] = p_b[N]$.*

□

Note that pivoting is a simply syntactic condition that can be tested straightforward. This concept is the key of our tests of implication.

Example 7.2.1 Let $P = \{r_0, r_1, r_2\}$ where:

$r_0 : p(X, Y, W, Z) : -e(X, Y, W, Z)$
$r_1 : p(X, Y, W, W) : -a(V, W), e(Y, W, W, V), e(V, Y, W, X), p(X, Y, V, W)$
$r_2 : p(X, Y, W, Q) : -e(X, W, Q, X), p(Y, X, V, V)$

We can see that the positions $N = \{1, 4\}$ are pivoting in r_1 since:

$$p_h[1, 4] = p_b[1, 4]$$

However, in r_2 this set of positions N are not pivoting given that $p_h[1, 4] \neq e[1, 4]$ and $p_h[1, 4] \neq p_b[1, 4]$. In r_2, we can only find pivoting position 1 since $p_h[1] = e[1]$. □

7.3 A test to identify programs that imply a fd

In this section we present a test to identify programs that imply a fd f.

The test is based in two conditions (that depends on a fd) such that if they are satisfied by a program in class $\mathcal{P}$, we can assure that this program implies such fd.

Lemma 7.3.1 Let P be a program in class $\mathcal{P}$. Let F be a set of fds defined over $EDB(P)$ and let $f = p$: $\{i_1, \ldots, i_k\} \rightarrow \{i_{k+1}\}$, where p is the IDB predicate defined by P. Let e the EDB predicate in the non-recursive rule of P. If the two conditions showed below are true then $F \models_P f$.

- *Condition 1* The fd $e : \{i_1, \ldots, i_k\} \rightarrow \{i_{k+1}\}$ is in F^+,
- *Condition 2 The set of argument positions $\{i_1, \ldots, i_k, i_{k+1}\}$ are pivoting in $Chase_F(r_i)$ for all recursive rule r_i in P.*

Proof Let $P' = \{r_0, Chase_F(r_i)\}$, for all recursive rule r_i in P. Let d a EDB defined over *EDB(P)*. Let us suppose that d satisfies F. Assume that the Conditions 1 and 2 are satisfied. By Condition 1, $P'(d)$ satisfies $e : \{i_1, \ldots, i_k\} \rightarrow \{i_{k+1}\}$. By Condition 2, $\pi_{i_1,\ldots,i_k,i_{k+1}}[p, P'(d)] = \pi_{i_1,\ldots,i_k,i_{k+1}}[e, P'(d)]$[a]. Then, $P'(d)$ satisfies $p : \{i_1, \ldots, i_k\} \rightarrow \{i_{k+1}\}$. Finally, $P(d)$ satisfies $p : \{i_1, \ldots, i_k\} \rightarrow \{i_{k+1}\}$ since by Corollary 3.1.1 $P(d) \equiv_{SAT(F)} P'(d)$. □

Example 7.3.1 Let $P = \{r_0, r_1\}$, where

$r_0 : p(X, Y, W, Z) : -e(X, Y, W, Z).$
$r_1 : p(X, Y, W, Z) : -\ a(V, W), a(V, Z), e(Y, W, Z, V), e(V, Y, W, X), p(X, Y, V, W).$

Let $F = \{e : \{1\} \rightarrow \{4\}, e : \{1\} \rightarrow \{3\}, e : \{2, 3\} \rightarrow \{4\}, e : \{2, 3\} \rightarrow \{1\}, a : \{1\} \rightarrow \{2\}\}$ and let us consider only databases that satisfy F.

Let $f = p : \{1\} \rightarrow \{4\}$. Note that P is in $\mathcal{P}$.

We can see that the positions 1 and 4 are not pivoting in r_1. However, if we chase r_1 we obtain the rule:

$$Chase_F(r_1) = p(X, Y, W, W) : -a(V, W), e(Y, W, W, V), e(V, Y, W, X), p(X, Y, V, W)$$

Then in $Chase_F(r_1)$ positions 1 and 4 are pivoting and then, the conditions of Lemma 7.3.1 are maintained and therefore $F \models_P f$. □

7.4 A test to identify programs that do not imply a fd

In this section we present a test that allow us to identify programs that do not imply a fd.

[a]Notation introduced in section 2.4.7.

If a program P (in class $\mathcal{P}$) implies a fd f then we show two conditions that must be held, otherwise the program P does not imply the fd f.

Lemma 7.4.1 Let P be a program in class $\mathcal{P}$. Let F be a set of fds over EDB(P) in BCNF. Let $f = p : \{i_1, \ldots, i_k\} \rightarrow \{i_{k+1}\}$, where p is the IDB predicate defined by P. Let e be the EDB predicate name in the non-recursive rule of P. Assume e : $\{i_1, \ldots, i_k\} \rightarrow \{i_{k+1}\}$ is left-hand minimal with respect to F. If $F \models_P f$ then we have that the following two conditions must be held:

- *Condition 1* The minimal fd $e : \{i_1, \ldots, i_k\} \rightarrow \{i_{k+1}\}$ is in F^+,
- *Condition 2 The set of argument positions $\{i_1, \ldots, i_k, i_{k+1}\}$ are pivoting in $Chase_F(r_i)$ for any composition of rules of P, $r_i = r_1 \circ r_2 \circ \ldots \circ r_n \circ r_0$, where $r_1, r_2 \ldots r_n$ (where there can be repeated rules) are recursive rules and r_0 is the non-recursive one.*

$\square$

Proof Note that Condition 1 is trivially necessary. Thus, in what follows we assume that is true.

Then, assuming that Condition 1 is true and Condition 2 is false, we give a procedure for constructing, for all programs P in $\mathcal{P}$ and for all F defined on predicates in $EDB(P)$, a counterexample database, denoted by d_{count}. d_{count} satisfies the following two conditions:

1. d_{count} is in $SAT(F)$.
2. $P(d_{count})$ is not in $SAT(\{f\})$.

Let P be a program in class $\mathcal{P}$ and let F be a set of fds over $EDB(P)$ in Boyce Codd Normal Form.

Let r_{count} be the rule that does not satisfy Condition 2. We build r_{count} as follows: $r_{count} = Chase_F(r_i)$, where $r_i = r_1 \circ r_2 \circ \ldots \circ r_n \circ r_0$, and $r_1, r_2 \ldots r_n$ are recursive rules in P (where some of them may be repeated) and r_0 is the non-recursive one. Let r_{count} be of the form:

$$r_{count} : p(X_1, \ldots, X_n) : -L, E$$

where L denotes the conjunction of EDB atoms in the body of r_{count} that are not defined over the predicate name e (the predicate name of the atom in the body of the non-recursive rule). E represents the set of atoms in the body of r_{count} defined over the predicate name e.

From now on, in order to build the counterexample database d_{count}, we consider all the variables in r_{count} (and therefore in L and E) as constants. Let d be the database formed by $L \cup E \cup e_h$. Note that L and E are set of ground facts because we are considering their variables as constants and e_h is an atom constructed as follows: $e_h = e(A_1, \dots, A_n)$, where:

- **for all** $j, 1 \leq j \leq n$, if $j \in \{i_1, \dots, i_k\}$, then $A_j = X_j$;
- **else** A_j is a new, distinct constant

Notice that e_h has the same constants as the head of r_{count} on the argument positions in the antecedent of f, and distinct constants anywhere else.

Let $d_{count} = Chase_F(d)$. We claim that d_{count} is a counter example of $F \models_P f$. In order to prove that, first we have to prove that the chase of d can be done without produce any change in L and E and, $e_h[\{i_1, \dots, i_k\}] = e'_h[\{i_1, \dots, i_k\}]$, where e'_h is e_h after the chase.

The set $E \cup L$ satisfies F because they come from the body of r_{count} that is the chase of r_i. Then, since all the EDB atoms (in d) different from e_h (defined over the predicate name e) are in $SAT(F)$, the first application of a fd over d during the chase must be a fd $e : X \rightarrow \{a\}$ denoted, for example, by g. Moreover, g must equate variables in the atoms $e_i \in E$ and e_h. We have to prove that $X = \{i_1, \dots, i_k\}$.

Suppose that $X \subset \{i_1, \dots, i_k\}$. Since the scheme of the database is in BCNF, X is a key. If we compute $(X)^+_F = \mathcal{U}$, then we have that $e : \{i_1, \dots, i_k\} \rightarrow \{i_{k+1}\}$ is not minimal, contradiction since we supposed $X \subset \{i_1, \dots, i_k\}$.

Now assume $\{i_1, \dots, i_k\} \subset X$, in this case, we cannot apply g since e_h in the positions different from $\{i_1, \dots, i_k\}$ has new variables.

If we apply g over e_h, since $\{i_1, \dots, i_k\}$ is a key, the variable equalizations can be performed such that after the application of a set of fds, $e_i = e_h$, given that all the terms in e_h that are not in $\{i_1, \dots, i_k\}$ are new constants that are not in any other part of d.

Then we can remove one of the facts and therefore, there is no possibility of apply any other fd. Hence, the chase ends.

Now we are ready to define the facts that violate f.

p_1 is the head of r_{count} and p_2 is constructed as follows. Let p_2 be the p-fact that we can prove by applying r_0 to $\{e'_h\}$. This implies that $p_2 = p(A'_1, \dots, A'_n)$[b].

[b]Where $A'_1, \dots, A'_n$ are the variables in e'_h.

Note that

- d_{count} is in $SAT(F)$ and d_{count} contains facts about predicates in $EDB(P)$ only.
- p_2 is in $P(d_{count})$ (by the definition of p_2).
- p_1 is in $P(d_{count})$. This is true since, as we have already seen $d_{count} = E \cup L \cup \{e'_h\}$. Thus, p_1 can be obtained by applying r_{count} to $\{E, L\}$[c].
- p_1 and p_2 violate *f*. This fact is proven below.

In order to do this, we prove that $p_1[\{i_1, \dots, i_k\}]$ is the same as $p_2[\{i_1, \dots, i_k\}]$ and then that $p_1[i_{k+1}]$ is different from $p_2[i_{k+1}]$.

By construction, $p_2[\{i_1, \dots, i_k\}] = e'_h[\{i_1, \dots, i_k\}]$. We saw that $e_h[\{i_1, \dots, i_k\}] = e'_h[\{i_1, \dots, i_k\}]$. Therefore, $p_2[\{i_1, \dots, i_k\}] = e_h[\{i_1, \dots, i_k\}]$. However, by construction of e_h and p_1, $p_1[\{i_1, \dots, i_k\}] = e_h[\{i_1, \dots, i_k\}]$. Therefore, $p_1[\{i_1, \dots, i_k\}] = p_2[\{i_1, \dots, i_k\}]$.

By construction, $e'_h[i_{k+1}]$ is the same as $p_2[i_{k+1}]$, then since, by construction, $p_1[i_{k+1}] \neq e_h[i_{k+1}]$, if $e_h[i_{k+1}]$ does not change during the $Chase_F(d)$ the proof ends, else the proof continues.

If $e_h[i_{k+1}]$ changes during chase, then $e'_h = e_i$, where $e_i \in E$. Thus, we have by construction:

$$e_i[\{i_1, \dots, i_k\}] = e'_h[\{i_1, \dots, i_k\}] = p_1[\{i_1, \dots, i_k\}]$$

However, since in r_{count} the argument positions $\{i_1, \dots, i_k, i_{k+1}\}$ are not pivoting, we have that $e_i[i_{k+1}] \neq p_1[i_{k+1}]$, hence, $e'_h[i_{k+1}] \neq p_1[i_{k+1}]$. □

Example 7.4.1 Let $P = \{r_0, r_1, r_2\}$ where:

$r_0 : p(X, Y, W, Z) : -e(X, Y, W, Z)$
$r_1 : p(X, Y, W, W) : -a(V, W), e(Y, W, W, V), e(V, Y, W, X), p(X, Y, V, W)$
$r_2 : p(X, Y, W, Q) : -e(X, W, Q, X), p(Y, X, V, V)$

Let F be $\{e : \{1\} \rightarrow \{4\}\}$. Let f be $p : \{1\} \rightarrow \{4\}$. We want to know if $F \models_P f$.

Let us check, for example, $r_1 \circ r_2 \circ r_0$:

$$p(X, Y, W, W) : -a(V, W), e(Y, W, W, V), e(V, Y, W, X), e(X, V, W, X), e(Y, X, V', V')$$

[c]Remember that E and L do not change during the chase.

$Chase_F(r_1 \circ r_2 \circ r_0)$ is:

$$p(X,Y,W,W) : -a(V,W), e(Y,W,W,V), e(V,Y,W,X), e(X,V,W,X), e(Y,X,V,V)$$

Since positions 1 and 4 are not pivoting in $Chase_F(r_1 \circ r_2 \circ r_0)$, hence we are sure that $F \models_P f$ is not satisfied by P. $\square$

Chapter 8

Conclusions

In the last years, research efforts in databases have put more strength in new data models like object oriented databases, semi-structured databases, and others. In certain circles, researchers think that the deductive model is out of date.

Leaving apart the discussion of deductive databases being a current issue or not, it is clear that many of the problems that arise in the deductive model are still useful to solve problems in other areas like logic programming or data warehousing.

However, the main reason to refute the argument of those researchers that think that the deductive model is out of date is the recent appearance of the new SQL standard, SQL99 [MS02, UW97]. SQL99 includes queries with linear recursion, thus it is mandatory to develop optimization techniques to be included in the DBMS in order to obtain better results in the running time of recursive queries.

In this dissertation, we have presented results in such area. More precisely, our work fits in the the field of semantic query optimization.

Two different algorithms to optimize recursive datalog programs were presented. Both algorithms seek the same target, that is, from a linear datalog program and a set of fds, our algorithms obtain an equivalent program when both are applied to databases satisfying the input set of fds.

The reader may wonder why do we provide two algorithms with the same target. The reason is that depending on the input program and the input set of fds, one of the algorithms could produce better results than the other. However, the output of the chase of datalog programs can be introduced as input of the cyclic chase of datalog programs. Therefore, given linear sirup, both algorithms can be combined in order to obtain the benefits from both algorithms.

The differences between the two algorithms is that the *chase of datalog programs* uses the partial chase of tress that does not consider the atoms resulting from the application of the non-recursive rule of the input program. However, the *cyclic chase of datalog programs* uses the chase of trees that considers the atoms in the last level of the tree. Nevertheless, the chase of datalog programs considers all the variables whereas the cyclic chase is more limited in such aspect.

As a future work, we will try to develop an algorithm that would take into account all the variables and the atoms resulting from the application of the non-recursive rule(s) of the input program. It is easy to see that such algorithm would obtain better results, however our believe is that such algorithm would be very expensive in computation if we compare it with our algorithms, thus the two algorithms presented in this work would be, even we found such algorithm, valuable. In addition, it would be also interesting the extension of the chase of datalog programs and the cyclic chase to larger classes of datalog programs.

Also, as a future work, it would very desirable the construction of a prototype of query optimizer in order to acquire experimental results of the benefits obtained using our algorithms.

Apart from our most palpable results, that is, two algorithms to optimize linear recursive datalog programs, we believe that we have made advances in a very interesting area where not many works can be found.

The scarcity of research in the area of semantic query optimization of recursive datalog programs that are evaluated over databases satisfying a set of fds should not be seen like an indicative of the lack of interest of this area. The mainly used tool (the chase) has demonstrated that is valuable in many different data models, so why not in the deductive model?

We believe that the reason should be found in the fact that from a research point of view, the use of the chase in recursive queries is very intricate.

In that way, we believe that some tools developed in this thesis may be useful to the study of recursive datalog programs. Atom chains, equalization chains and formation rules, can hopefully be used to go further in the research in this area.

We also provide results in another area, the implication of functional dependencies. This problem is completely solved in the relational model. Moreover, in any computer school it is taught most of the theory about this issue in the relational model.

However, as many other problems that are easy to tackle in the relational model, they become very difficult in the deductive model. In fact, Abiteboul and Hull [AH88] showed that the implication problem for datalog programs in general is undecidable. Therefore, the research efforts are now focused on the definition of classes of datalog programs where the problem can be studied and solved.

In this field, we provide a syntactic condition that, given a program belonging to a special class of linear datalog programs and a set of fds, determines if such program implies a fd. We also provide a syntactic condition that given a program in such a special class of linear datalog programs and a set of fds, determines if such program does not imply a fd. In both cases we can determine if the program implies or not a fd without computing the output database, with just checking a very simple syntactic condition. Thus, both methods are very valuable.

As a future work, we would try to find an necessary and sufficient condition to determine whether a *lsirup* applied to databases satisfying a set of fds implies or not a fd.

Bibliography

[ABU79] Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. The theory of joins in relational databases. *ACM TODS*, 4(3):297–314, 1979.

[AH88] Serge Abiteboul and Richard Hull. Data functions, datalog and negation. In *Proc. Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principle of Database Systems*, pages 143–153, 1988.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[Arm74] W. W. Armstong. Dependency structures of data base relationships. In *Proc. 1974 IFIP Congress*, pages 580–583, 1974.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Sixth Symposium on Principles of Programming Languages*, pages 110–117, 1979.

[BGTHP98a] Nieves R. Brisaboa, Agustin Gonzalez-Tuchmann, Héctor J. Hernández, and José R. Paramá. Chasing programs in datalog. In *Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming DDLP98*, pages 13–23. GMD- Forschungzentrum Informationstechnik GmbH 1998 (GMD Report 22), 1998.

[BGTHP98b] Nieves R. Brisaboa, Agustin Gonzalez-Tuchmann, Héctor J. Hernández, and José R. Paramá. The chase of datalog programs. In *Advances in Databases: proceedings of 16th British National Conference on Databases, BNCOD16*, pages 165–166. Lecture Notes in Computer Science Vol. 1405, 1998.

[BHPP01] Nieves R. Brisaboa, Héctor J. Hernández, José R. Paramá, and Miguel R. Penabad. An algorithm to optimize 2-lsirups under functional dependencies. In *Proceedings of 10th Portuguese Conference on Artificial Intelligence, EPIA '01*, 2001. To be published.

[BV84] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31:718–741, 1984.

[CGM88] Upen S. Chakravarthy, John Grant, and Jack Minker. Foundations of semantic query optimization for deductive databases. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kauffmann Publishers, 1988.

[CH82] A. K. Chandra and David Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, (25):99–128, 1982.

[CK86] Satvros S. Cosmadakis and Paris C. Kanellakis. Parallel evaluation of recursive rule queries. In *Proc. Fifth ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 280–293, 1986.

[Cod80] E.F. Codd. A relational model for large shared data banks. *C. ACM*, 13(6):377–387, 1980.

[GM77] H. Gallaire and Jack Minker, editors. *Logic and Databases.* Plenum, 1977.

[GMSV87] Haim Gaifman, Harry G. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. In *Proc. 2nd IEEE Symp. on Logic in Computer Science*, pages 106–115, 1987.

[GT95] Agustin Gonzalez-Tuchmann. *The chase of datalog programs.* PhD thesis, New Mexico State University, Department of Computer Science, Las Cruces, NM 88003-0001, 1995.

[HP97] Héctor J. Hernández and José R. Paramá. Implicación de dependencias funcionales en datalog sobre esquemas en forma normal boyce-codd. In *Actas del Congreso Internacional en Ciencias Computacionales (CIICC 97)*, pages 57–68, Durango (México), September 1997.

[HPB97a] Héctor J. Hernández, José R. Paramá, and Nieves R. Brisaboa. Condición sintáctica para la implicación de dependencias funcionales en 2-lsirups aplicados a edbs en forma normal boyce-codd. In *Memorias del Primer Encuentro de Computación ENC 97*, pages 27–34, Queretaro (México), September 1997.

[HPB97b] Héctor J. Hernández, José R. Paramá, and Nieves R. Brisaboa. Implicación de dependencias funcionales en datalog sobre esquemas en forma normal boyce-codd. In *Actas de las Segundas Jornadas de Investigación y Docencia en Bases de Datos*, pages 71–79, Madrid (Spain), July 1997.

[Ler86] Nadine Lerat. Query processing in incomplete logical databases. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, volume 243, pages 260–277. Springer-Verlag, Roma, Italy, 1986. Proceedings of the International Conference on Database Theory.

[LH91] Laks V. S. Lakshmanan and Héctor J. Hernández. Structural query optimization - a uniform framework for semantic query optimization in deductive databases. In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principle of Database Systems*, pages 102–114, 1991.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[Min88a] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffmann Publishers, 1988.

[Min88b] Jack Minker. Perspectives in deductive databases. *Journal of Logic Programming*, (5):33–60, 1988.

[MMS79] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM TODS*, 4(4):455–469, 1979.

[MS02] Jim Melton and Alan R. Simon. *SQL:1999 Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[Nau86] J.F. Naughton. Data independent recursion in deductive databases. In *Proc. Fifth ACM SIGACT-SIGMOD*

Symposium on Principle of Database Systems, pages 267–279, 1986.

[NG77] J.M. Nicolas and H. Gallaire. Data base: Theory vs interpretation. In Gallaire and Minker, editors, *Logic and Databases*, pages 33–54. Plenum, 1977.

[NS87] J.F. Naughton and Yehoshua Sagiv. A decidable class of bounded recursions. In *Proc. Sixth ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 227–236, 1987.

[PBPH00] José R. Paramá, Nieves R. Brisaboa, Miguel R. Penabad, and Héctor J. Hernández. Un procedimiento de optmización semántica de programas datalog. In *Proceedings of the V Jornadas de ingeniería de software y Bases de Datos*, pages 295–306, 2000.

[PDST00] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far. In *SIGMOD*, pages 273–284, 2000.

[Pop00] L. Popa. *Object/Relational Query Optimization with Chase and Backchase*. PhD thesis, University of Pennsylvania, 2000.

[PV99] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 455–466, 1999.

[RBSS90] Ragu Ramakrishnan, Per Bothner, Divesh Srivastava, and S. Sudarshan. Coral: A databases programming language. Technical Report TR-CS-90-14, Kansas State University, Department of Computing and Information Sciences, 1990.

[Sag87] Yehoshua Sagiv. Optimizing datalog programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 17, pages 659–698. Morgan Kauffmann Publishers, 1987.

[TA91] Riccardo Torlone and Paolo Atzeni. Updating deductive databases with functional dependencies. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Lecture Notes in*

Computer Science, volume 566, pages 278–291. Springer-Verlag, Munich, Germany, 1991. Second International Conference, Deductive and Object-Oriented Databases.

[Tan97] D. Tang. *Linearization-Based Query Optimization in Datalog.* PhD thesis, New Mexico State University, Las Cruces, New Mexico, 1997.

[Ull88] Jeffrey D. Ullman. *Principles of Database And Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database And Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[UW97] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems.* Prentice-Hall, 1997.

[Var88] Moshe Y. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *Proc. Seventh ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 341–351, 1988.

[WY92] Ke Wang and Li Yan Yuan. Preservation of integrity constraints in definite datalog programs. *Information Processing Letters*, 44(4), 1992.

Symbol Index