Implementation of a Database System with Boolean Algebra Constraints

by

András Salamon

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Peter Z. Revesz

Lincoln, Nebraska

May, 1998

Implementation of a Database System with Boolean Algebra Constraints

András Salamon, M.S.

University of Nebraska, 1998

Advisor: Peter Z. Revesz

This thesis describes an implementation of a constraint database system with constraints over a Boolean Algebra of sets. The system allows within the input database as well as the queries equality, subset-equality and monotone inequality constraints between Boolean Algebra terms built up using the operators of union, intersection and complement. Hence the new system extends the earlier DISCO system, which only allowed equality and subset-equality constraints between Boolean algebra variables and constants.

The new system allows Datalog with Boolean Algebra constraints as the query language. The implementation includes an extension of Naive and Semi-Naive evaluation methods for Datalog programs and algebraic optimization techniques for relational algebra formulas.

The thesis also includes three example applications of the new system in the area of family tree genealogy, genome map assembly, and two-player game analysis. In each of these three cases the optimization provides a significant improvement in the running time of the queries.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor Dr. Peter Revesz for his guidance and encouragement on this project. I also thank Dr. Jean-Camille Birget and Dr. Spyros Magliveras for serving on my advisory committee and for their time.

I would like to dedicate my thesis for the memories of my grandfathers Károly Németh and Sándor Salamon.

Contents

Introduction

Although among the database systems the relational database systems are the widespreadest systems at this moment, constraint databases is a very perspective approach to change them.

Constraint databases can contain quantifier-free first-order formulas. With the help of these formulas constraint databases are able to express more than traditional relational databases. For instance one tuple can contain infinite number of traditional tuples.

Constraint database systems can be categorized according to the type of the constraint. Some of the well-known constraint types are for instance: linear constraints, polynomial constraints, integer gap constraints.

In this system the constraints are Boolean Constraint, hence the name of the system is *Datalog with Boolean Constraint*. This system extends the possibilities of a previous system (Datalog with Integer Set COnstraint = DISCO) which was implemented in the department. DISCO system allows only subset-equality constraints between Boolean Algebra variables and constants, this system allows subset-equality, equality, and monotone inequality constraints between Boolean Algebra terms. A description of the DISCO system can be found in [2].

First there is a theoretical overview (Chapter 1) based on [6], then a chapter about the current implementation (Chapter 2), what kind of Boolean Algebra is implemented, the main structure of the program.

Because one really important part of the program is related to Relational Algebra, Chapter 3 describes the Relational Algebra formulas, how to store, convert and optimize these formulas.

The name of the implemented program is `GreenCoat`, Chapter 4 gives information about the user interface of the program. The predecessor of this program was implemented in the previous semester by Song Liu and I.

Chapter 5 describes some examples and try to demonstrate the possibilities of the system.

The system also supports the use of some multiset operators. Chapter 6 contains more information about the multisets.

Chapter 1

Datalog with Boolean Algebra Constraints

In this chapter I give an overview of the *'Datalog with Boolean Algebra Constraint'*. This was introduced by Kanellakis et al. [4] and extended by Peter Z. Revesz [6]. First I present the basic definitions necessary to understand the concept. Later I define the syntax methods.

1.1 Boolean Algebra

The following definition is taken from [6], more information can be found about Boolean Algebras in [1].

A **Boolean algebra** is a sextuple (δ, $\wedge$, $\vee$, $'$, 0, 1), where δ is the domain set, $\wedge$ and $\vee$ are binary operators ($\wedge : \delta \times \delta \rightarrow \delta$, $\vee : \delta \times \delta \rightarrow \delta$), $'$ is a unary operator ($' : \delta \rightarrow \delta$), 0 and 1 are two special elements of the domain ($0 \in \delta, 1 \in \delta$). They are also called *zero* and *identity* elements. Every Boolean algebra satisfies the following axioms: ($\forall x, y \in \delta$:)

$$\begin{aligned} x \vee y &= y \vee x & x \wedge y &= y \wedge x \\ x \vee (y \wedge z) &= (x \vee y) \wedge (x \vee z) & x \wedge (y \vee z) &= (x \wedge y) \vee (x \wedge z) \\ x \vee x' &= 1 & x \wedge x' &= 0 \\ x \vee 0 &= x & x \wedge 1 &= x \end{aligned}$$

$$0 \neq 1$$

Boolean term: All the elements of δ (including 0 and 1) are Boolean terms. All the elements of V (set of variables), and all the elements of C, where C is the set of constants (except 0 and 1), are Boolean terms. If t_1 and t_2 are both Boolean terms, than $t_1 \vee t_2$, $t_1 \wedge t_2$, t_1' are also Boolean terms.

Precedence constraint: If a constraint has the following form: $x \wedge y' = 0$, (where $x, y \in \delta \cup V \cup C$), then we call this constraint precedence constraint and denote with $x \leq y$.

Monotone Boolean function: A g Boolean function is monotone if $\forall x_i \leq y_i \; (1 \leq i \leq n): \; g(x_1, \ldots, x_n) \leq g(y_1, \ldots, y_n)$.

Monotone inequality constraint: $g(x_1, \ldots, x_n) \neq 0$ is a monotone inequality constraint if g is a monotone Boolean function.

The following is a well-known fact for Boolean terms:

Proposition 1.1 *Every t Boolean term can be converted to* **disjunctive normal form** *(DNF):*

$$t(z_1, z_2, \ldots, z_n) = \bigvee_{\underline{a} \in \{0,1\}^n} (t(a_1, a_2, \ldots, a_n) \wedge z_1^{a_1} \wedge z_2^{a_2} \wedge \ldots \wedge z_n^{a_n})$$

where z^0 denotes z', and z^0 denotes z.

1.2 Syntax of Datalog Queries with Boolean Constraints

The following basic definitions can also be found in [6]. Every Datalog program contains a set of facts (constraint tuples) and a set of rules. The facts can be seen as special rules as well. The general form of the facts is:

$$R(x_1,\ldots,x_k) : -f(x_1,\ldots,x_k) = 0, g_1(x_1,\ldots,x_k) \neq 0,\ldots,g_l(x_1,\ldots,x_k) \neq 0.$$

where f and $g_i (1 \leq i \leq l)$ are Boolean terms.

The general form of the rules is:

$$\begin{aligned} R(x_1,\ldots,x_k) \ \ :- \ \ & R_1(x_{1,1},\ldots,x_{1,k_1}),\ldots,R_n(x_{n,1},\ldots,x_{n,k_n}), f(\underline{x}) = 0, \\ & g_1(\underline{x}) \neq 0,\ldots,g_l(\underline{x}) \neq 0. \end{aligned}$$

where $R, R_1, \ldots, R_k$ are relation symbols (not necessary distinct symbols), $x's \in \delta \cup V \cup C$, $\underline{x}$ is the set of variables in the rule, and f and $g_i(1 \leq i \leq l)$ are Boolean terms.

It is not a real restriction that one side of the constraint is always 0. $(f = g) \equiv (((f \wedge g') \vee (f' \wedge g)) = 0)$ hence we can convert all the constraints to this form. Without loss of generality we can also assume that we have only one equality constraint, because $(f_1 = 0,\ldots,f_n = 0) \equiv ((f_1 \vee \ldots \vee f_n) = 0)$, therefore we can connect several equality constraints to create one constraint.

1.3 Quantifier elimination

A quantifier elimination method is an equivalency between an existentially quantified formula and a quantifier-free formula.

Quantifier elimination is used for variables on the right-hand side of a rule which do not occur as variables in the left-hand side.

There are three elimination methods described in [6]. The correctness proofs of the elimination methods also can be found in the article.

1.3.1 Elimination method for equality constraints

The first elimination method [6, Lemma 2.2] (which originates with George Boole) can be used for equality constraints:

$$\exists x(f(x, y_1, \ldots, y_k) = 0) \quad \equiv \quad f(0, y_1, \ldots, y_k) \wedge f(1, y_1, \ldots, y_k) = 0$$

1.3.2 Elimination method for precedence and monotone inequality constraints

The other [6, Lemma 2.4] can be used for precedence ($x \leq y$) and monotone inequality constraints:

$$\begin{aligned}\exists x(\quad & z_1 \leq x, \ldots, z_m \leq x,\\ & x \leq y_1, \ldots, x \leq y_k,\\ & w_1 \leq u_1, \ldots, w_s \leq u_s,\\ & g_1(x, v_1, \ldots, v_{1,n_1}) \neq 0,\\ & \vdots\\ & g_l(x, v_1, \ldots, v_{1,n_l}) \neq 0,)\end{aligned}$$

is equivalent to:

$$\begin{aligned}& z_1 \leq y1, \ldots, z_1 \leq y_k, z_2 \leq y_2, \ldots, z_m \leq y_k\\ & w_1 \leq u_1, \ldots, w_s \leq u_s,\\ & g_1((y_1 \wedge y_2 \wedge \ldots \wedge y_k), v_1, \ldots, v_{1,n_1}) \neq 0,\\ & \vdots\\ & g_l((y_1 \wedge y_2 \wedge \ldots \wedge y_k), v_1, \ldots, v_{1,n_l}) \neq 0,)\end{aligned}$$

where z_i, y_i, w_i, u_i, v_i's are variables or constants. Although they are not necessarily distinct symbols, they are different from x.

1.4 Naive and Semi-naive evaluation methods

If our input Datalog program does not contain recursive rules, then the evaluation of the program is simple, because it is enough to evaluate every rule only once using a standard algorithm for the ordering of the rules. This algorithm is described for example in [8]. If the input program contains recursive rules, then the rules have to be evaluated more than once.

1.4.1 Naive method

Example 1.1 The example is taken from [5]. We have an input database which contains parents-children pairs, and our goal is to find the ancestors of a specific person. The Datalog program is the following: (There is a longer description of this example in section 5.1)

```
children(P, C) :- P={"parent1", "parent2"}, C={"person", "brother"}.
children(P, C) :- P={"gparent1", "gparent2"}, C={"parent1", "uncle"}.
children(P, C) :- P={"gparent3", "gparent4"}, C={"parent2", "aunt"}.
children(P, C) :- P={"ggparent1", "ggparent2"}, C={"ggparent3"}.


AAncestor(P) :- children(P,C), {"person"} <= C.
AAncestor(P) :- children(P,C), AAncestor(P2), C /\ P2 != @.
```

After evaluating all of the rules once, we get the parents (`parent1`, `parent2`) of the specific person. If we evaluate the rules again, we get the parents and grandparents (`gparent1`, `gparent2`, `gparent3`, `gparent4`), of the person. After the third evaluation we get the great-grandparents (`ggparent1`, `ggparent2`) too. After the fourth evaluation the method does not give new ancestors, hence we can stop.

The previously used algorithm is called *Naive method*. The pseudo-code of the method is the following: (taken from [8])

```
for := 1 to m do
        $P_i := \emptyset$
repeat
        for i:= 1 to m do
                $Q_i := P_i$;
        for i:= 1 to m do
                $P_i :=$ EVAL(i,$Q_1$, ..., $Q_m$);
until $P_i = Q_i$ for all i ( $1 \leq i \leq m$);
```

Where P_i is the tuples of the ith relation. Q_i is the tuples of the ith relation in the previous step. At the beginning we erase all tuples. Than repeat the steps of the algorithm until the results of the last two steps are identical. During one step we store the tuples first ($Q_i := P_i$), than calculate the new tuples using the tuples calculated in the previous steps. The calculation is done by the `EVAL` function (1.4.3). In the function, i denotes the index of the current relation, $Q_1, \ldots, Q_m$ denote the tuples of the relations, which can be used by `EVAL`.

1.4.2 Semi-Naive method

Tha main disadvantage of the Naive method is that it recalculates the same tuples in every iteration. In the previous example during the first step the algorithm calculates the parents; during the second step the parents, and the grandparents; during the third and fourth step the parent, grandparents, and great-grandparents. Therefore the algorithm calculated the parents four times. If the number of the steps are greater – and in a real application it is several times greater – then this disadvantage is also greater. The main idea of the Semi-Naive method is to omit these recalculations.

If during the calculation we use only old tuples (tuples which were calculated before the previous step), then we only recalculate some older tuples. Therefore if we want to calculate new tuples we should use at least one new tuple (tuple which were calculated during the previous step).

Naturally the first step is an exception, because there are no new tuples before the first step, hence the first steps of the Semi-naive and Naive methods are identical.

The pseudo-code of the semi-naive evaluation (also taken from [8])

```
for := 1 to m do
    $\Delta P_i$ := EVAL($p_i, \emptyset, \ldots, \emptyset$)
    $P_i$ := $\Delta P_i$
repeat
    for i:= 1 to m do
        $\Delta Q_i$ := $\Delta P_i$;
    for i:= 1 to m do begin
        $\Delta P_i$ := EVAL_INCR(i, $P_1$, ..., $P_m$, $\Delta Q_1$, ..., $\Delta Q_m$);
        $\Delta P_i$ := $\Delta P_i$ - $P_i$;
    end;
    for i:=1 to m do
        $P_i$ := $P_i \cup \Delta P_i$
```

until $\Delta P_i = \emptyset$ for all i ($1 \leq i \leq m$);

Where P_i denotes the tuples of the ith relation, ΔP_i the new tuples in the current, ΔQ_i the new tuples in the previous step of the relations. At the first step we use the `EVAL` function to calculate the tuples. Than we repeat the steps of the algorithm until there is no new tuples in the last step. In a step first we store the new tuples ($\Delta Q_i = \Delta P_i$), then calculate the new tuples using `EVAL_INCR` function (1.4.4). After this we check whether the new tuples are really new tuples ($\Delta P_i = \Delta P_i - P_i$). At the end of the step we should update the value of P_i ($P_i = P_i + \Delta P_i$). The `EVAL_INCR` function has more parameters than the `EVAL` function, because the `EVAL_INCR` function needs not only all the tuples, but the new tuples as well.

1.4.3 `EVAL` function

This function calculates new tuples from the previously known tuples. Every relation is converted to relational algebra formulas (3.1), and these formulas are optimized (3.2). By using these formulas, the `EVAL` function can easily calculate the new tuples. Every formula is a tree, and the leaves of the formulas are the relations. If we substitute the relations with the tuples of the relations and execute the relational algebra operators in the nodes, then the root of the tree will contain the new tuples.

1.4.4 `EVAL_INCR` function

This function is similar to the `EVAL` function. The difference is that `EVAL_INCR` should use at least one new tuple during the calculation. To achieve this, we clone all the rules as many times as relations occur in the right hand side of the rule. In the ith clone we put a Δ before the ith relation. In Example 1.1 the clones of the rules of relation AAncestor:

```
AAncestor(P) :- Δchildren(P,C), "person" <= C.
AAncestor(P) :- Δchildren(P,C), AAncestor(P2), C /\ P2 != @.
AAncestor(P) :- children(P,C), ΔAAncestor(P2), C /\ P2 != @.
```

Instead of the original rules we convert and optimize these rules to relational algebra formulas. With this we have the possibility to calculate `EVAL_INCR`

There is an other possibility to simplify these rules. In Example 1.1 `children` relation has only facts, hence $\Delta children$ is always empty. Therefore all the rules which contains $\Delta children$ can be eliminated. If we eliminate these rules we only have one rule left:

```
AAncestor(P) :- children(P,C), ΔAAncestor(P2), C /\ P2 != @.
```

More generally if all the rules of relation R contain only facts, then we can eliminate every rule which contains ΔR in it.

Although not implemented in the system, sometimes we can eliminate other rules too.

Example 1.2 Assume that we have `mother` and `father` relations in our input database, and our goal is to find the ancestors of a particular person. First we can define a `parent` relation, and after that the solution is the same as in Example 1.1. Here is a part of the program:

```
mother(M,C) :- M == {"mother"}, C == {"child1", "child2"}.
father(F,C) :- F == {"father"}, C == {"child1", "child2"}.
parent(P,C) :- mother(P,C).
parent(P,C) :- father(P,C).
⋮
```

Although the `parent` relation has two rules, and neither of them are facts, we calculate the tuples of the `parent` relation in the beginning of the evaluation, and after this step no new tuples will be added to this relation. Hence there is only one step in which Δ`parent` is not empty. Therefore it would be possible for some of the relations to calculate the number of steps after Δ will be always empty and eliminate the appropriate rules then.

Chapter 2

Implementation

2.1 The implemented Boolean Algebra

Chapter 1 gave a small overview of the basics of Datalog with Boolean Constraints. All the definitions, lemmas are working with all the possible Boolean Algebras. Although during the implementation most of the system are working with all the possible Boolean Algebras, only one Boolean Algebra is implemented.

2.1.1 Sets

In the first Boolean Algebra, δ contains the sets of integers and strings. Because of storage restrictions, δ contains only finite sets, or the sets which complement is finite. It is not a strict restriction because the length of input files are finite, hence the user can define only these sets, and all the operators are closed. The Boolean Algebra operators are defined in the following way: $\wedge \equiv \cap$, $\vee \equiv \cup$, $' \equiv$ complement set. We should define 0 and 1 elements also: $0 = \emptyset = \{\}, 1 = \{\}' =$ complete set $=$ set of all integers and strings.

2.2 Implemented quantifier elimination methods

In [6] P. Revesz describes three elimination methods. One of them works only with atomless Boolean Algebras. Because the implemented Boolean Algebra of Section 2.1.1 is not an atomless Boolean Algebra, that method is not implemented. The other two methods (described earlier in Sections 1.3.1, 1.3.2) are implemented in this system. The elimination method described in Section 1.3.2 can be used only if all the inequality constraints are monotone constraints. The system does not check the monotonity, but it assumes that all the inequality constraints are monotone inequality constraints.

2.3 Hardware and Software

The system is implemented in Java language. Originally the system was implemented under IRIX 6.2, using JDK 1.0.2 (Hardware: 4 CPU SGI R10000), although it was tested also under WinNT (Hardware: Pentium 200, Pentium 133, Pentium II 267) using Microsoft Visual J++ 1.1. Because one of the main properties of Java language is portability, the system should work on most well-known systems even without recompilation.

The parser was implemented using Java Compiler Compiler (JavaCC), Version 0.7pre3.

2.4 Java program

2.4.1 Packages

The Java language supports using packages (collection of similar classes). The pack-

Name	Function
storage	Storage of Datalog programs
relalg	Storage of Relational Algebra Operators
relalg.optimize	RA optimization methods
elimination	Elimination methods
evaluation	Naive & Semi-Naive evaluation
parser	The parser
util	Miscellaneous classes

Table 2.1: Packages of the Java program

ages of the system and their function can be seen in Table 2.1.

`relalg` package

`relalg` and its subpackage `relalg.optimize` contain classes related to relational algebra formulas. Chapter 3 contains more information about relational algebra formulas. The optimization methods (Section 3.2) are implemented in `relalg.optimize` package.

`elimination` package

Quantifier elimination methods (Section 1.3) are implemented in this package. Because two methods are implemented, and the system does not know in advance which one can be used, a new quantifier elimination method is implemented. This method is only a container of other quantifier elimination methods (right now two methods), and tries to execute the first method, and if it is not possible, than the following one until one of the methods was successful, or none of them was successful.

`evaluation` package

This package implements the generic code of the Naive (Section 1.4.1) and Semi-Naive (Section 1.4.2) evaluation methods. The `eval`, `eval_incr` functions are also defined in this package.

`parser` package

The function of this package is to parse the input files and the user commands. Chapter 4 contains more information about the user commands and the input file format. The java source files in this package is created by JavaCC from a grammar description file (`.jj`).

`storage` package

This package stores the Datalog programs. The hierarchy among the classes can be seen in Figure 2.1. This is not a superclass-subclass hierarchy, every class shown on the picture contains one or more instances of the classes shown below the class. At the top of this hierarchy there is the `Database` class, which contains our database. A database is a set of `Relations`. Every relation have one or more `Rules`. Every rule have a head, which represented by a RelationTitle, and a body. A body can contains other relation names (`RelationTitles`), and `Constraints`. A Constraint can be an equality or an inequality constraint. Every constraint have the form: 'Boolean Term' $= 0$ or 'Boolean Term' $\neq 0$. A Boolean Term is represented by a `Term`. Because every boolean term can be transformed to Disjunctive Normal Form (DNF), every term is stored as an array of basic Conjunctions (`Conjunction`). A basic conjunction is a conjunction of literals, which can be stored as an array of `Literals`. A literal can be

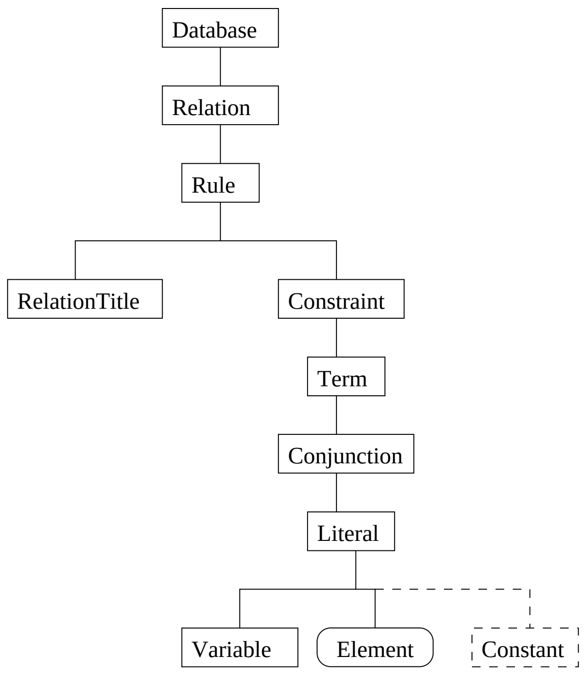

Figure 2.1: The hierarchy among the classes of `storage` package

a variable (`Variable`), an element of δ (`Element`) and a constant (`Constant`). The constants are not implemented in the current version of the system, but it is worth to mention the possibility to integrate constants to the system.

`Element` is an abstract class, it can contain the elements of all possible Boolean Algebras. It defined the necessary method which has to be implemented to represent a concrete Boolean Algebra. `ElementSet` is a subclass of `Element` it can store the element of all the possible set-typed Boolean Algebras. The only non-abstract subclass of `ElementSet` is `ElementFSet`, which implements $\delta = sets$ (Section 2.1.1). Figure 2.2 shows a superclass-subclass hierarchy among these classes. Angled rectangle indicates that the class is not abstract, while oval-shaped rectangle indicates that the class is abstract.

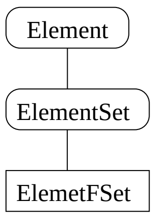

Figure 2.2: The subclasses of `Element`

Chapter 3

Relational Algebra Formulas

3.1 Relational Algebra Formulas

Because the relational algebra formulas are described in several textbooks, for example [8], in this part I describe only the differences between the general relational algebra and relational algebra formulas used in this program.

In this system there are four relational algebra operators: join ($\bowtie$), project (Π), union ($\cup$), select (σ). Because cross-product ($\times$) can be seen as a special join, in this system join represents both of them. The other main difference, that join and union originally are binary operators, hence the number of operands are always two. In this system the number of operand are greater or equal than two. For instance if we want to represent the join of four relations, originally we need three join operators, the new system needs only one.

The system stores relational algebra formulas as a tree, it makes easy to change the formula, and to represent an operations if it has more than two operands. It is also very useful when we want to visualize a formula.

The input from the user contains Datalog rules, hence it is necessary to convert these rules to relational algebra formulas. A conversion method is described in Ullman's book [8, chapter 3]. However, the method works with pure Datalog rules, hence that method needs to be extended.

3.1.1 Converting a Rule

A general *Datalog with Boolean Algebra* rule has the following form:

$$R(X_1, \ldots, X_k) : - \, Q_1(Y_{1,1}, \ldots, Y_{1,l_1}), \ldots, Q_m(Y_{m,1}, \ldots, Y_{m,l_m}), \sigma_1, \ldots, \sigma_n$$

Where $m \geq 0$ is the number of relations on the right-hand side, $n \geq 0$ is the number of selections on the right-hand side. Although either m or n can be zero, they cannot be zero at the same time. $(n + m > 0)$.

If $m > 1$ then first we need to join the relations on the right hand side. After that we can issue the selection one after the other. (It would be possible to combine the selections into one selection, but the optimization method works better if we do not combine them.)

If $\{Y_{1,1}, \ldots, Y_{1,l_1}, \ldots, Y_{m,1}, \ldots, Y_{m,l_m}\} \subseteq \{X_1, \ldots, X_k\}$ then the right-hand side contains only variables which can be found in the left hand side, therefore it is not necessary to use projection. Otherwise we need a projection $(\Pi_{X_1,\ldots,X_k})$ as well.

3.1.2 Converting a Relation

First the algorithm converts all the rules of the relation. If the number of the rules is greater than one then the algorithm connects the formulas with union.

Example 3.1 If relation R has the following two Datalog rules:

```
R(x,y) :- C(x,y).
R(x,y) :- A(x,z), B(z,y), D(y), z != {1,2,3}.
```

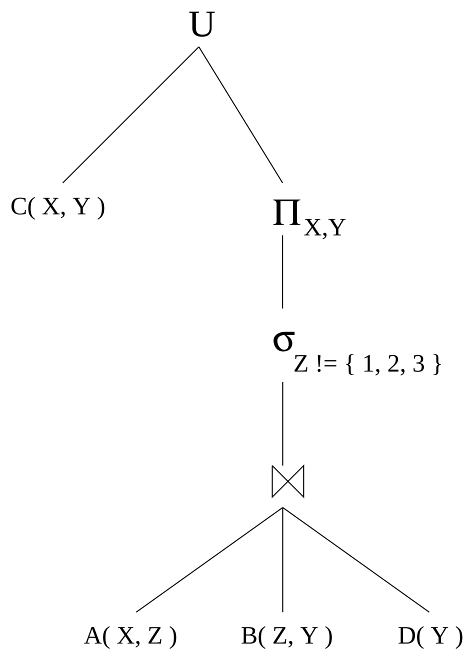

Figure 3.1: The formula of Relation R

then the algorithm first converts the first rule, and we get the formula: $C(x, y)$. Next the second rule is converted yielding: $\pi_{x,y}(\sigma_{z!=\{1,2,3\}}(A(x, z) \bowtie B(z, y) \bowtie D(y)))$, and finally the two formulas are joined together with a union operator. $C(x, y) \cup \pi_{x,y}(\sigma_{z!=\{1,2,3\}}(A(x, z) \bowtie B(z, y) \bowtie D(y)))$ (see Figure 3.1).

3.2 Optimization of Relational Algebra Formulas

Although after converting Datalog rules to relational algebra formulas we are able to use the formulas, it is better to first optimize the formula. Using optimization methods we calculate a new formula from our original formula. The new formula should be equivalent with the original one (if we evaluate it, the result should be the same), and it should be evaluated faster. No algorithm can improve all formulas.

Usually the optimization algorithms improve the majority of the formulas, and leave unaltered or even worsen some formulas.

There are many optimization methods. The algebraic manipulation method used in our system is described in the next subsection.

3.3 Algebraic Manipulation

This method is also described in [8]. In this method we will use some equations between formulas. These equations are also called laws. First we give a list of these laws, and later an algorithm which can change the original formula using these laws. After the changes the new formula can be usually evaluated faster than the original

We have to optimize only a subset of the possible formulas, because our formulas are originally Datalog programs.

3.3.1 Laws

We use the following laws from [8]

1. *Commutative law for join:*

$$R_1 \bowtie R_2 \equiv R_2 \bowtie R_1$$

2. *Associative law for joins :*

$$(R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3)$$

3. *Cascade of projections:*

$$\pi_{A_1,A_2,\ldots,A_k}(\pi_{B_1,B_2,\ldots,B_l}(R)) \equiv \pi_{A_1,A_2,\ldots,A_k}(R)$$

if $\{A_1, A_2, \ldots, A_k\} \subseteq \{B_1, B_2, \ldots, B_l\}$

4. *Cascade of selections:*

$$\sigma_{F_1}(\sigma_{F_2}(R)) \equiv \sigma_{F_1 \wedge F_2}(R) \equiv \sigma_{F_2}(\sigma_{F_1}(R))$$

5. *Commuting selections and projections:*

If the set of attributes in condition F is the subset of $\{A_1, A_2, \ldots, A_k\}$:

$$\pi_{A_1,A_2,\ldots,A_k}(\sigma_F(R)) \equiv \sigma_F(\pi_{A_1,A_2,\ldots,A_k}(R))$$

If the set of attributes in F is $\{A_{i_1}, A_{i_2}, \ldots, A_{i_m}\} \cup \{B_1, B_2, \ldots, B_l\}$:

$$\pi_{A_1,A_2,\ldots,A_k}(\sigma_F(R)) \equiv \pi_{A_1,A_2,\ldots,A_k}(\sigma_F(\pi_{A_1,A_2,\ldots,A_k,B_1,B_2,\ldots,B_l}(R)))$$

6. *Communing selection with Join:*

If all the attributes of F are the attributes of R_1:

$$\sigma_F(R_1 \bowtie R_2) \equiv \sigma_F(R_1) \bowtie R_2$$

If $F = F_1 \wedge F_2$, and the attributes of F_1 are only in R_1, and the attributes in F_2 are only in R_2, then:

$$\sigma_F(R_1 \bowtie R_2) \equiv \sigma_{F_1}(R_1) \bowtie \sigma_{F_2}(R_2)$$

If $F = F_1 \wedge F_2$, and the attributes of F_1 are only in R_1, but the attributes in F_2 are in both R_1 and R_2:

$$\sigma_F(R_1 \bowtie R_2) \equiv \sigma_{F_2}(\sigma_{F_1}(R_1) \bowtie R_2)$$

7. *Commuting a projection with a join:*

   $\{A_1, A_2, \ldots, A_k\} = \{B_1, B_2, \ldots, B_l\} \cup \{C_1, \ldots, C_m\}$, where B_is are attributes of R_1, and C_is are attributes of R_2:

$$\pi_{A_1,A_2,\ldots,A_k}(R_1 \bowtie R_2) \equiv \pi_{B_1,B_2,\ldots,B_l}(R_1) \bowtie \pi_{C_1,C_2,\ldots,C_m}(R_2)$$

3.3.2 Principles

These are three main principles of algebraic query optimization:

1. Perform selections as early as possible

2. Perform projections as early as possible

3. Combine sequences of unary operations

3.3.3 The Algorithm

The steps of the algorithm

1. For each selection use rule (4) – (6) to move the selection down.

2. Move projections down using rules (3), (7), If possible, delete projections.

3. Use rule (4) to combine cascades of selection into one selection.

Move selections down

In this step our goal is to move selections as down as possible. Originally we have a set of selections (S_1, S_2, ... ,S_k), and a set of relations (R_1,R_2, ... ,R_l). In the original execution order, we connect the relations with a join operator (if the number of the operations is greater than zero), than calculate the selections one after the other.

During the optimization, we first check which relations and selections have common variables. Let V_i be the set of relations which have common variables with S_i. More formally:

$$V_i = \{R_n \mid (variables\ in\ R_n) \cap (variables\ in\ S_i) \neq \emptyset\}$$

A selection (S_i) can be executed, if the join of all the relations mentioned in V_i is already calculated. The join can contain other relations also.

If V_i is empty or contains all the relations, then the selection is executed only after we join all the relations. Therefore we should find the place of the other selections.

If $\exists i \forall j : V_i \subseteq V_j$, then S_i will be executed before all the other selections. If such an index (i) does not exist, then the program chooses any index, which has a small size V_i. Next we modify the V_j ($j \neq i$) sets.

$$V_j := \begin{cases} V_j \setminus V_i \ \cup \ S_i & if\ V_i \cap V_j \neq \emptyset \\ V_j & if\ V_i \cap V_j = \emptyset \end{cases}$$

As we can see, V_j contains not only relations but selections as well.

The previously described method is one step of the optimization. This step should be repeated until all the selections are chosen. If there are some relations which are not used during the optimization (no selection contains any variables of the relation), then a final join should connect these relations and the selections.

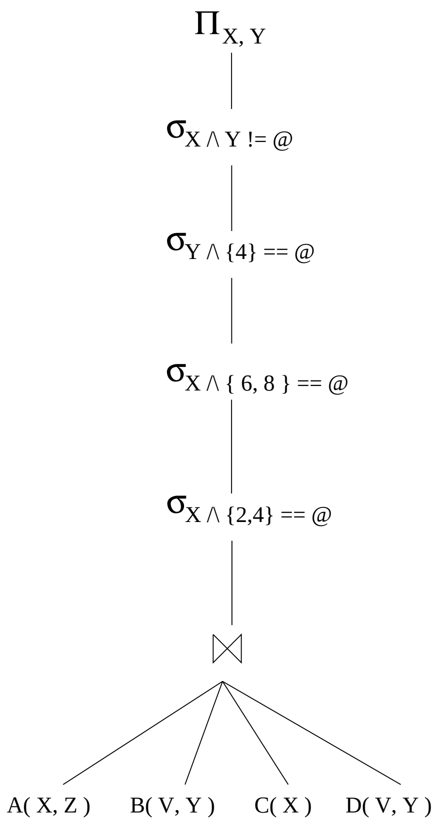

Figure 3.2: The formula before the optimization

Example 3.2 Assume, we have the following Datalog program:

$$\begin{aligned} R(X,Y) :- \quad & A(X,Z), B(V,Y), C(X), D(V,Y), X \cap \{2,4\} == @, \\ & X \cap \{6,8\} == @, Y \cap \{4\} == \emptyset, X \cap Y! = \emptyset. \end{aligned}$$

the corresponding relational algebra formula (see Figure 3.2):

$$\pi_{x,y}(\sigma_{x\cap y\neq\emptyset}(\sigma_{y\cap\{4\}=\emptyset}(\sigma_{x\cap\{6,8\}=\emptyset}(\sigma_{x\cap\{2,4\}=\emptyset}(A(x,z) \bowtie B(v,y) \bowtie C(x) \bowtie D(v,y))))))$$

$\sigma_{y\cap\{4\}=\emptyset}$ and $\sigma_{x\cap\{6,8\}=\emptyset}$ and $\sigma_{x\cap\{2,4\}=\emptyset}$ each contain only one variable: y, x, and x respectively. The variables in $\sigma_{x\cap y\neq\emptyset}$ are x and y. $\sigma_{x\cap y\neq\emptyset}$ has common variables with relations A,B,C, and D, therefore $V_1 = \{A, B, C, D\}$. Similarly, $V_2 = \{B, D\}$, $V_3 = \{A, C\}$, $V_4 = \{A, C\}$.

There exists no V_i, which is the subset of all the other V_j's hence the algorithm chooses one selection, which has the smaller V_i. In this example the algorithm can choose V_2, V_3, and V_4, assume that it chooses V_2. As we issue S_2, we get the following formula: $\sigma_{y\cap\{4\}=\emptyset}(B(v, y) \bowtie D(v, y))$.

The new values of V_1, V_3, and V_4 are: $V_1 = \{A, B, C, D\}\backslash\{B, D\} \cup \{S_2\} = \{A, C, S_2\}$, V_3 and V_4 are unchanged because V_3 and V_4 has no common variables with S_2. ($V_3 = \{A, C\}$, $V_4 = \{A, C\}$)
Now $V_3 \subseteq V_1$, $V_3 \subseteq V_4$ hence we can issue S_3, and get $\sigma_{x\cap\{6,8\}=\emptyset}(C(x) \bowtie A(x, z))$. The new values of V_1 and V_4 are: $V_1 = \{A, C, S_2\}\backslash\{A, C\} \cup \{S_3\} = \{S_2, S_3\}$, $V_4 = \{A, C\}\backslash\{A, C\} \cup \{S_3\} = \{S_3\}$.
Now $V_4 \subseteq V_1$, so the algorithm can issue S_4, and we get $\sigma_{x\cap\{2,4\}=\emptyset}(\sigma_{x\cap\{6,8\}=\emptyset}(C(x) \bowtie A(x, z)))$
Finally we issue V_1, and get $\Pi(x, y)(\sigma_{x\cap y\neq\emptyset}(\sigma_{x\cap\{2,4\}=\emptyset}(\sigma_{x\cap\{6,8\}=\emptyset}(C(x) \bowtie A(x, z)) \bowtie \sigma_{y\cap\{4\}=\emptyset}(B(v, y) \bowtie D(v, y)))))$ (Figure 3.3)

Moving Projections Down

In this step our goal is to move projections as down as possible. Originally we have a projection, below that maybe some selections and finally a join (Figure 3.4) If it is possible then we evaluate the projection before the join. Usually it is not possible, but we can eliminate at least some of the variables before the join.

Denote PV the set of variables in the projection. Denote SV the set of variables in the selections. Denote $V[i]$ the variables of the ith branch of the join. All the

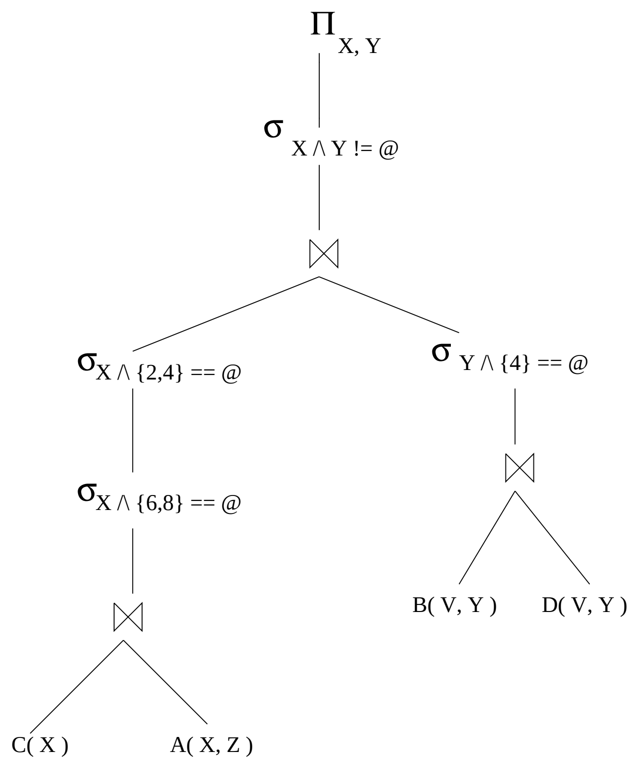

Figure 3.3: The formula after the first step of optimization

variable in PV or SV cannot be eliminated before the selections.

If a variable occurs only in one branch, and the variable is not in PV or in SV, then this variable easily can be eliminated before the join. For instance, if our original formula is $\Pi(x)(A(x) \bowtie B(x,y))$ as shown in Figure 3.5, then y occurs only in the second branch, therefore we can eliminate y before the join. The optimized formula is: $A(X) \bowtie \Pi(x)(B(x,y))$ as shown in Figure 3.6.

If there exist no variable which occurs only in one brach, then the algorithm chooses one variables which occurs in the least branches. If all the variables are occur in all the branches then the algorithm cannot eliminate any variables. If more than

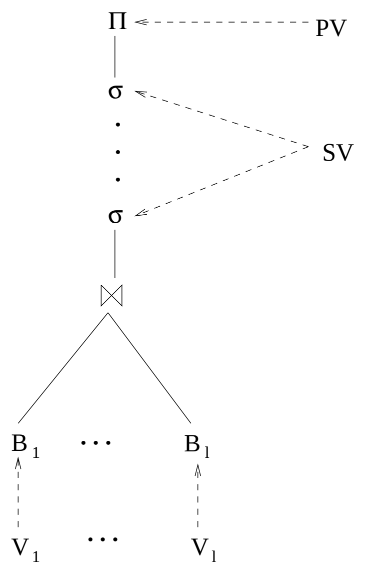

Figure 3.4: The variables of the formula

one variables occur in the same branches then the algorithm eliminates these variables at the same time.

After this step the branches of the join has changed, therefore the algorithm should recalculate the values of $V[i]'s$. This recalculation is similar to the recalculation during the first step (Moving selections down) of the algorithm. There algorithm is running until we cannot find any eliminable variables.

Because one brach of the join can contain other join operators, after the algorithm move a projection below the join, we should check whether it is possible to move the projection even more below the other join. To achieve this the algorithm calls itself in a recursive way. The new instance of the algorithm works only on a subtree (subformula) of the original tree (formula).

Example 3.3 After the first step of the optimization we got the formula shown in

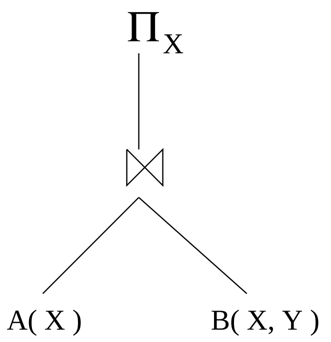

Figure 3.5: Optimization input

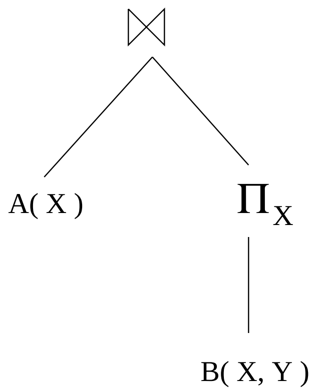

Figure 3.6: Optimization output

Figure 3.3. The variables of the join are X, Y ($PV = \{X, Y\}$), the variables in the selection are also X and Y ($SV = \{X, Y\}$). $PV \cup SV = \{X, Y\}$, therefore we cannot eliminate X and Y. The upper join has two branches. The variables in the first branch are: X, Z, therefore $V[1] = \{X, Z\}$. Similarly $V[2] = \{V, Y\}$. Z and V are local variables because Z occurs only in the first, and V occurs only in the second branch. Therefore we can eliminate Z from the first branch before the join, and V from the second one. The variables in the first brach are: $\{X, Z\}$. If we eliminate Z, we have only X, hence we should issue a $\Pi(X)$ below the join. In the same way we should issue $\Pi(Y)$ below the second branch. Our new formula is shown in Figure 3.7.

Finally the algorithm tries to move the projections even below the other joins. In the second branch, Y cannot be eliminated, because Y is a variable in the projection (and in the selection also). V cannot be eliminated because it occurs in all the branches. In the first branch X is ineliminable, because X is a variable in the projection. Z is a local variable of the second branch, hence it can be eliminated before the join. Therefore the algorithm can move the projection below the join, and

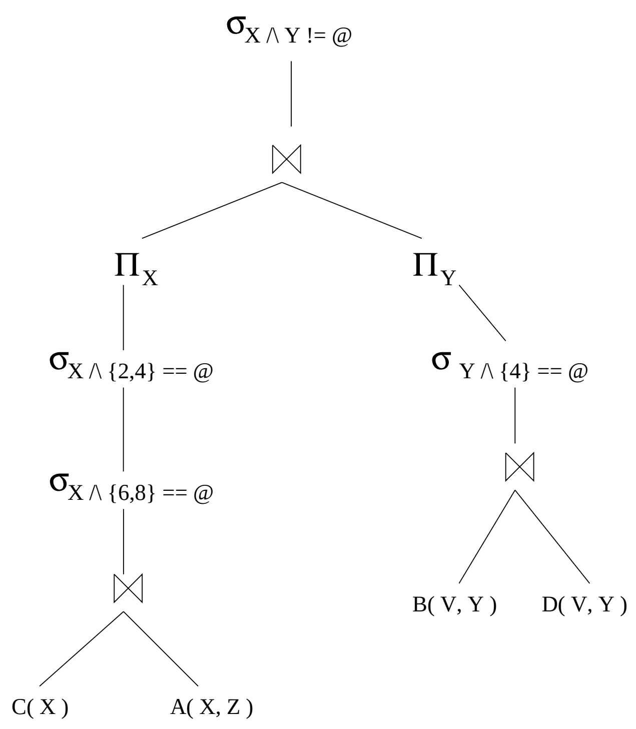

Figure 3.7: The formula during the second step of optimization

we get our new formula, which is shown in Figure 3.8.

Connecting Selections

This is the final and the easiest step of the optimization. In this step, the algorithm combine cascades of selection into one selection. The algorithm simply checks every edge in the tree, and if both vertices of this edge are selections, then connects the two vertices and erases the edge.

Example 3.4 In our example (Figure 3.8) there is only on pair of selections which can be connected. ($\sigma_{x \cap \{2,4\} = \emptyset}$ and $\sigma_{x \cap \{6,8\} = \emptyset}$. After this step, we get our final formula, which can be seen in Figure 3.9.

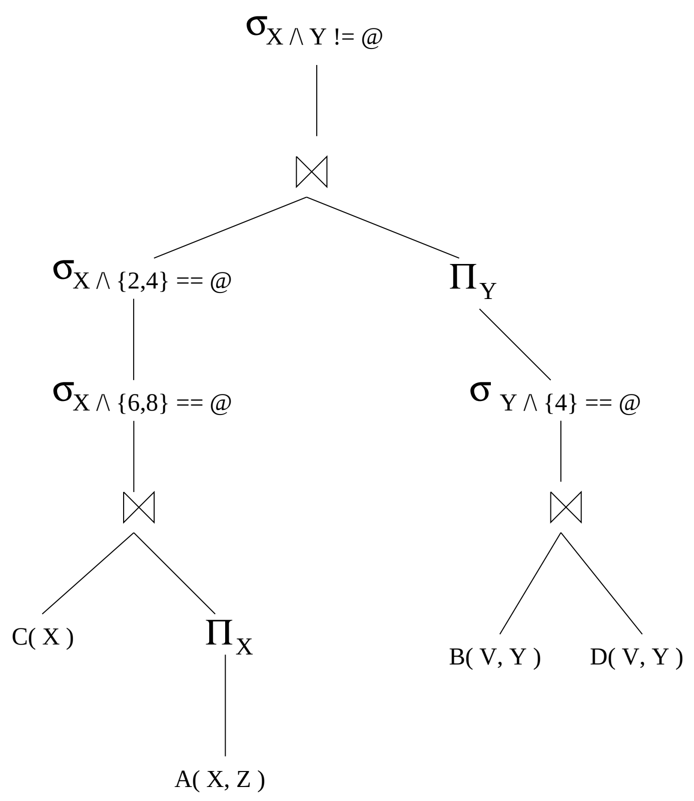

Figure 3.8: The formula after the second step of optimization

3.4 Calculating multiple joins

When we have to join more than two relations, then the simplest way to join them is to choose one tuple from each relation, create a new tuple and write the result to the output relation. For instance we have four relations A, B, C, D, and we want to calculate

$$A(X,Y) \bowtie B(Y,Z) \bowtie C(Z,V) \bowtie (V,W)$$

Assume that there are 100 tuples in the relations. In this case we have to create $100^4 = 10^8$ tuples.

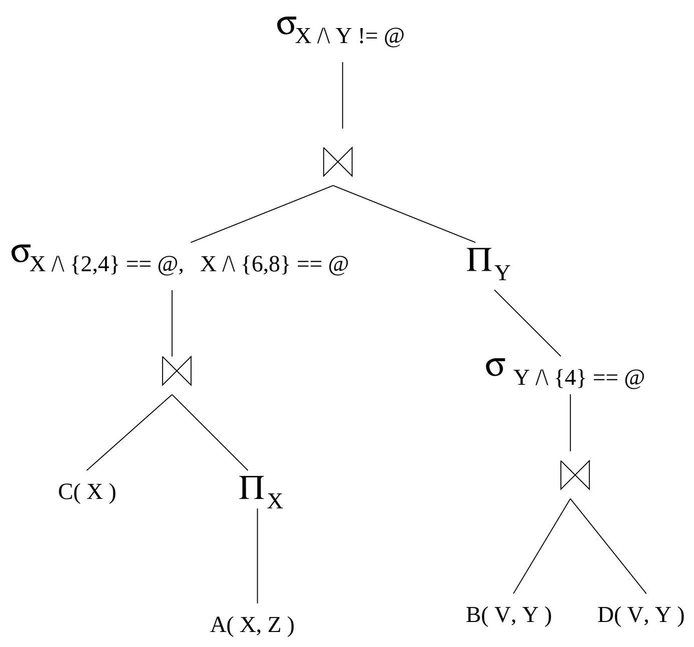

Figure 3.9: The formula after the optimization

A better solution if we join two relations, than join the third to the result of the previous join, and finally join the fourth to the last result. For instance if we calculate $A(X,Y) \bowtie B(Y,Z)$, than the size of the result usually is less than $100^2 = 10^4$. Let us assume that the results always contain 100 tuples. In this case we have to calculate $3 * 100^2 = 3 * 10^4$ tuples, which is less than 1 percent of the original calculation.

Of course we do not know the size of the result relation before we create the result. If the two relation have no common variables, than the join is a cross-product, so the size of the result relation is the multiplication of the size of the original relations. In other cases we only know that the size of the result is not greater than the multiplication of the size of the original relations.

If the algorithm is able to estimate the size of the resulting relation, then it is possible to join the relations in a good order, therefore the algorithm is able to decrease the cost of a multiple join. The good order can be determined using dynamic programming. Because the algebraic manipulation considerably decreases the occurrence of multiple joins, and makes it more difficult to estimate the size of the resulting relation, this system does not change the execution order of the joins, but simply executes the joins from left to right.

Chapter 4

User's Manual

The user interface of the program is a character based interface. (One student in the department is working on a Graphical User Interface)

After staring the program the system waits for a user command. After the user enters the command, the system waits for the next command. This process is finished if the user exists from the system.

With the help of some commands the user can change the values of the switches, with the other command the user can ask the program to give information or execute a process. First I describe the switches and later the other commands.

The user is also able to give a new rule or fact to the system. The general form of the rules and facts are described in Section 1.2. There are some differences:

- The user can enter more than one equality constraint.
- The user can enter constraints in which neither side of the constraint is zero.
- The user can use not only $\cup$, $\cap$, $'$ but $\subseteq$ and $\supseteq$ as well.
- Because keyboard does not contain symbols like $\cup$, $\cap$, or the other operators,

Mathematical	In the system
$\cup$	`\/`, and
$\cap$	`/\`, or
R'	not(R), R'
$\subseteq$	`<=`, `=<`, `[=`
$\supseteq$	`=>`, `>=`, `=]`
$=$	`=`, `==`
$\neq$	`!=`, `<>`
$\emptyset$	@, ZERO

Table 4.1: Operators

the user should use other symbols instead of. Table 4.1 shows the symbols which can be used by the system. Usually more than one symbols can be used, they are separated by commas.

4.1 Switches

Most of the switches has two possible values, they are either turned on or off. All the possible values should be typed with small letters, in the examples the capital letters show the default value of the switch.

- `time on|OFF.`

  If the switch is turned on, then program after each evaluation displays the time used during the evaluation. It displays not only the total time, but some part-time (for instance time used by different relation operators) as well.

  All the time values are in second, and they are real second, not CPU second.

- `tempfile on|OFF.`

If the switch is turned on, after each step of the Naive (Section 1.4.1) or Semi-Naive (Section 1.4.2) evaluation, the program prints out the derived tuples into a temporary file (`'temp.bld'`). This file is not being overwritten by the following steps, hence the user can analyse the results after each step. Although the file contain the results of more steps, each result has the same format as the input file, therefore it is possible to use different parts of this file as an input file, and continue an interrupted execution.

- `trace on|OFF.`

  If the switch is turned on then the program display the inner representation of a rule after a new rule added to the system.

- `optimize ON|off.`

  If the switch is turned on, then the program uses relational algebra optimization (Section 3.2), if not then the program uses the original formula. The value of the switch should be changed before loading the input file to make effect.

- `method old|naive|SEMINAIVE.`

  With this switch the user can choose between the implemented evaluation methods. The Naive (Section 1.4.1), Semi-Naive (Section 1.4.2) methods work with also non-recursive and recursive queries, but the 'old' method works only with non-recursive queries.

4.2 Commands

- `exit|bye.`

The command exists the program.

- `load|consult 'filename'.`

  The command load the file with the given filename, and reads the queries from the file. The format of the input file is described in Section 4.3.

- `formula "R" [onto 'filename'].`

  The command displays the relational algebra formula of "R" which is used by the Naive evaluation (Section 1.4.1). If the user specify a file name, then the formula will be printed out to the file, otherwise it will be printed out to the screen.

- `formuladelta "R" [onto 'filename'].`

  The command displays the relational algebra formula of "R" which is used by the Semi-Naive evaluation (Section 1.4.2). If the user specify a file name, then the formula will be printed out to the file, otherwise it will be printed out to the screen.

- `display ["R"] [onto 'filename'].`

  If the user do not specify a relation name then the program prints out the names and the arities of all relations.

  If the user specify a relation name then the program prints out the rules of the relation.

  Similarly to the `formula` and `formuladelta` commands the user can name a file, otherwise the result of the command will be printed out to the screen.

- `displaydf [onto 'filename'].`

  The command prints out all the derived facts in the database.

  Similarly to the previous commands the user can name a file, otherwise the result of the command will be printed to the screen.

- `memory.`

  The command displays the total and the free memory used by the system.

- `clear.`

  The command erases all the relations, rules from the database.

- $\mathrm{R}(E_1, \ldots, E_n)$?

  With this command the user can ask the derived fact of a relation. E_i can be either a variable or an element of δ. One variable can occur more than once. At the first time using this command the system calculates the derived facts using of of the evaluation method. Later the system uses the derived facts stored in the memory, hence the answer will be faster.

  Note: This is the only command which ends with a `'?'` *instead of a* `'.'`.

4.3 Input file format

The input file starts with a line contains `'begin'` and ends with a line `'end'`. Between these lines there are the rule definitions.

The input file may contain empty lines, one-line comments (after `//` as in C++ or Java), multi-line comments (between `/*` and `*/` as in C, C++ or Java).

If a line would be too long it can be splitted into more lines, each line but the last should end with a `'\'` character. In the examples of this thesis we do not use the `'\'` character.

Chapter 5

Examples

5.1 Ancestors example

5.1.1 The problem

The ancestors example appeared in [5]. In the input database we store informations about parents and their children. Our goal is to calculate all the ancestors of one particular person.

5.1.2 The input database

Pure Datalog

In pure Datalog one of the easiest way to use the *children* relation. One tuple can contains one parent and one child. For instance if `husband` and `wife` have three children: `child1, child2, child3`, then our input database is the following:

```
children( husband, child1 ).
children( husband, child2 ).
children( husband, child3 ).
children( wife, child1 ).
children( wife, child2 ).
children( wife, child3 ).
```

New system

In the new system, we need only one tuple to represent the previous example:

```
children( { husband, wife }, {child1, child2, child3} ).
```

Comparison

The previous example shows that in the new system, we need fewer tuples to store the same data. It also can be seen, that the tuples in the new system are more complex then the tuples in pure Datalog. In the example we needed only one tuple instead of six. More generally, if a couple has k children, then the pure Datalog needs $2k$ tuples, in contrast to the new system, which needs only one.

5.1.3 The Datalog program

Pure Datalog

```
AAncestor(P) :- children(P, {"person"}).
AAncestor(P) :- children(P, C), AAncestor(C).
```

New system

```
AAncestor(P) :- children(P,C), {"person"} <= C.
AAncestor(P) :- children(P,C), AAncestor(P2), C /\ P2 != @.
```

Comparison

The program contains two rules in both systems. The rules are very similar, although the new system has slightly more complex rules.

5.1.4 The output database

Pure Datalog

In the pure Datalog every tuple in the output relation represents one of the ancestors.

New system

In the new system every tuple in the output relation represents two ancestors.

Comparison

The new system contains half the number of tuples as pure Datalog.

5.1.5 Execution complexity

In this comparison I assume that both systems are using the Semi-Naive or the naive evaluation.

Pure Datalog

At the first step, the system finds the parents of `person` using the first rule. The system should check all the `children` tuples, and find those in which `person` is the child.

Later we need to use the second rule, hence we need to evaluate a join.

New system

At the first step, the systems finds the parents of `person` using the first rule. The system should check all the `children` tuples, and find those in which `person` is one of the children.

Later we need to use the second rule, hence we need to evaluate a join, and a selection.

Comparison

The first step is almost identical. Because the number of tuples is less in the new system, in that case the program should check fewer tuples. On the other hand, the tuples are more complex in the new system, hence to check one tuple is more time-consuming. If we analyze one step, then we can find real advantages.

If our goal is to find the ancestors of child3, then in the original system we should check all the six tuples. Check means here to evaluate `child1` == `child3`, `child2` == `child3`, `child3` == `child3` . We should evaluate each of them twice, because each children occur in two tuples.

In the new system, we should check only one tuple. Check here means that we should calculate the intersection of {`child3`} and { `child1, child2, child3` }. It means, that we should evaluate `child1` == `child3`, `child2` == `child3`, `child3` == `child3`. In this case we need to evaluate these only once. Finally we should check whether the intersection is empty or not.

In the later steps, both systems evaluate a join, the second one also evaluates a selection. If in the pure datalog system we denote the number of tuples in relations `children` and `AAncestor` C_p and A_p respectively, then the number of tuples in the

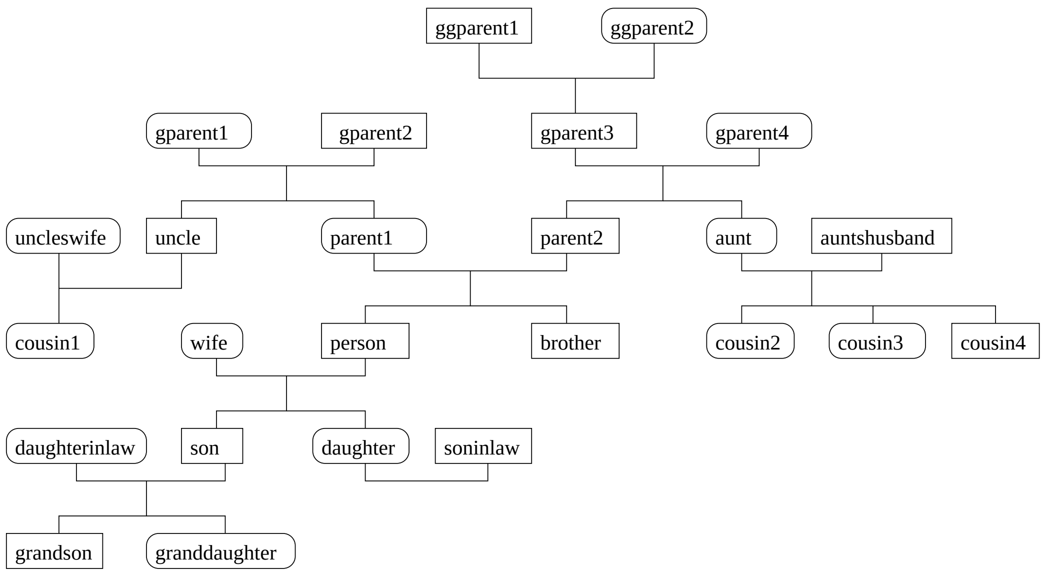

Figure 5.1: A family tree

new system are $C_n = \frac{C_p}{2}$ and $A_n = \frac{A_p}{2k}$, where `k` is the average number of children. Therefore the number of basic operation during join is $A_p\,C_p$ in the old system and $A_n\,C_n = \frac{A_p\,C_p}{4k}$ in the new system.

If we are using Semi-Naive evaluation (1.4.2) than the algorithm uses only the new tuples, hence the number of basic operation is $\Delta A_p\,C_p$ in the old, and $\Delta A_n\,C_n = \frac{\Delta A_p\,C_p}{4k}$ in the new system.

Similarly to the first step, one basic step is more complex in the second system, but there is still an advantage of using the new system. Usually the cost of the selection is much more less than the cost of the join, hence it is not a problem that in the new system we need a selection as well.

5.1.6 Run-time results

Figure 5.1 shows a family tree, which was used for test purposes. The squared rectangles contain the names of the men, oval shaped rectangles contain the names of the women. In the program there is no difference between the two sex, it only helps to understand the family tree.

Table 5.1 shows the running times. During the evaluation, the program calculated not only the ancestors of `person`, but the ancestors of everybody. Because the optimization of relational algebra formulas does not change the formulas in this example, the optimization has no effect on the running-time (the small differences are only because of the inaccuracy of time-measurement). As can be seen in the table the Semi-Naive evaluation is approximately eight times faster then the Naive evaluation, hence the Semi-Naive evaluation is a great improvement.

5.2 Genome Map

5.2.1 The problem

The following genome map problem is described in [7].

The deoxyribonucleic acid (DNA) is a sequence of nucleotides. There are four nucleotides: adenine (A), cytosine (C), guanine (G), and thymine (T).

Random substrings of a given DNA are called clones. Clones may overlap each other. It is possible to cut a DNA string into clones with so called restriction enzymes. After cutting we loose all information about the order of the clones. Each clone can be analysed further. By various enzymes the clones can be digested, and we can measure the fragments after the digestion. To eliminate the errors of measurement we can round the length of fragments.

As an input we have a set of clones $(c_1, \ldots, c_n)$ of a DNA string, and the lengths of the fragments of each clone.

The goal is to find the original order of the clones. This problem is NP-complete. However there exist different heuristics which make it possible to solve the problem.

In this example we have an order of clones, and the task is to decide whether it is a possible order of clones or not.

5.2.2 Solution

The idea for the algorithm is described in [7].

Further restrictions

To apply this solution we need some further restrictions in the input database:

- No clone contains any other clone.
- No clone contains two different fragments with the same length. Although it is possible that two different clones have different fragments with the same length.
- There exists k such that each fragment is contained in at most k clones.
- If $(c_1, \ldots, c_n)$ is the correct order of the clones then $\forall i$ $(1 \leq i < n) : c_i$ overlaps c_{i+1}

Automaton

Because every fragment is contained at most k clones, it is enough to analyze $k + 1$ clones at the same time. We call $k + 1$ adjacent clones a *window*. At the beginning the window contains the first $k+1$ clones, but during the solution this window will shift right. We denote the clones in the window $A_1, \ldots, A_{k+1}$.

During the solution we will change the values of $A_1, \ldots A_{k+1}$. The values will change if we shift the window, or if we pick fragments from the clones. We pick fragments from several clones $(A_1, \ldots, A_l \ (1 \leq l \leq k))$ at the same time. If a clone must contain the fragments that we pick next, then we call the clone *active*. If A_j active then $\forall i \ (i < j) \ A_i$ also active, hence one number is enough to store the set of active clones.

We create a a non-deterministic automaton to solve the problem. The automaton contains $k+2$ states, where S_0 is the initial state, H is the halt stage, and S_i is the stage which represents when i clone active.

Now we need to define the transition of the automaton.

If $A_i \subseteq A_{i+1}$ then we cannot pick a fragment from the first i clones which is not in A_{i+1}, therefore A_{i+1} can be declared an active clone too.

If $A_1 = \emptyset$ then we can shift the window right.

If there are fragments which are in the active clones and not in the first non-active clone, then we can pick these fragments.

Figure 5.2 shows an automaton when $k=5$. This automaton is taken from [7]. Because this system supports more Boolean operators than the DISCO system which was used in [7], the description of the edges of this automaton is simpler than in [7].

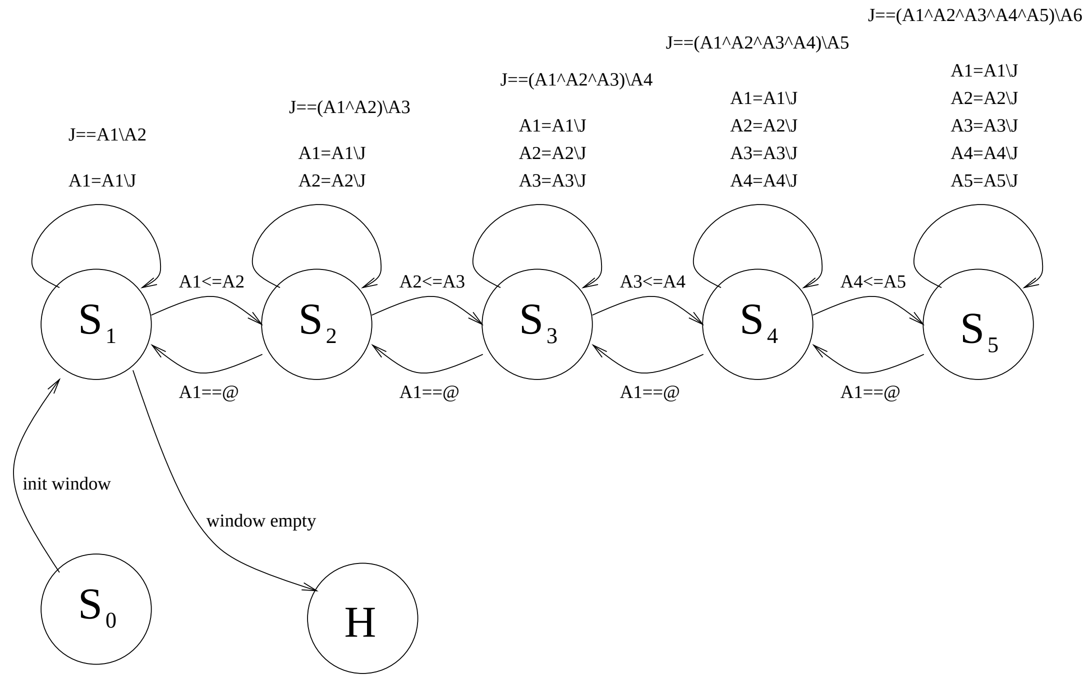

Figure 5.2: The non-deterministic automation for k=5

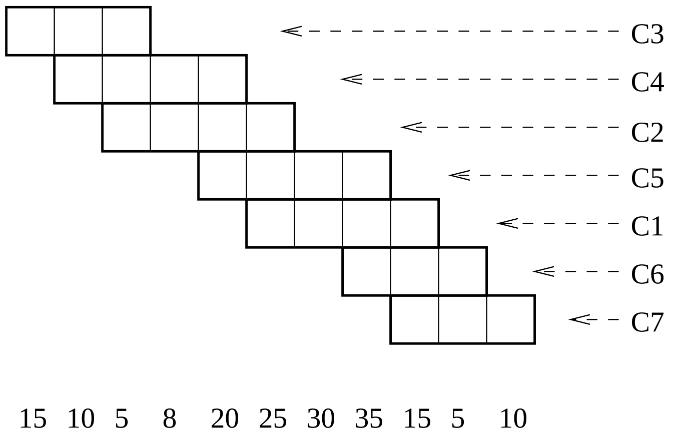

Figure 5.3: The clones of the smaller example

5.2.3 Concrete examples

Example with input size n=11, k=3

This is a small example, when $k = 3$. Figure 5.3 shows the clones with the fragments and also shows the correct order of the clones.

To represent this example we need the following part of the input file:

```
clone(N,X) :- N == {1}, X == {25,30,35,15}.
clone(N,X) :- N == {2}, X == {5,8,20,25}.
clone(N,X) :- N == {3}, X == {15,10,5}.
clone(N,X) :- N == {4}, X == {10,5,8,20}.
clone(N,X) :- N == {5}, X == {20,25,30,35}.
clone(N,X) :- N == {6}, X == {35,15,5}.
clone(N,X) :- N == {7}, X == {15,5,10}.
clone(N,X) :- N =={99}, X == @.

firstClone(N) :- N == {3}.
nextClone(N1,N2) :- N1 == {3}, N2 == {4}.
nextClone(N1,N2) :- N1 == {4}, N2 == {2}.
nextClone(N1,N2) :- N1 == {2}, N2 == {5}.
nextClone(N1,N2) :- N1 == {5}, N2 == {1}.
nextClone(N1,N2) :- N1 == {1}, N2 == {6}.
nextClone(N1,N2) :- N1 == {6}, N2 == {7}.
nextClone(N1,N2) :- N1 == {7}, N2 == {99}.
nextClone(N1,N2) :- N1 =={99}, N2 == {99}.
```

To implement the automation we need the second part of the input file:

```
pick(J, A, B) :- A = B \/ J, B /\ J ==@.

S1(L, A1, A2, A3, A4) :- L == @, firstClone(s1), clone(s1, A1),
                                  nextClone(s1,s2), clone(s2, A2),
                                  nextClone(s2,s3), clone(s3, A3),
                                  nextClone(s3,s4), clone(s4, A4).
```

Problem	Naive		SemiNaive	
	Without optimization	With optimization	Without optimization	With optimization
Genome map ($n = 11, k = 3$)	1810	1146	103	98
Genome map ($n = 16, k = 5$)	—	—	1008	932
Ancestor	59.4	59.6	7.6	7.5
Unavoidable sets	30.1	253.9	19.2	68.9
Multiset	1197	1181	61	61

Table 5.1: Test results (Pentium II 267 MHz)

```
// i -> i+1
S2(L, A1, A2, A3, A4) :- L == @, S1(J, A1, A2, A3, A4), A1 <= A2.
S3(L, A1, A2, A3, A4) :- L == @, S2(J, A1, A2, A3, A4), A2 <= A3.

// i+1 -> i
S1(L, A2, A3, A4, A5) :- L == @, S2(J, A1, A2, A3, A4),
              A1 == @, clone(c1, A4), nextClone(c1, c2), clone(c2, A5).
S2(L, A2, A3, A4, A5) :- L == @, S3(J, A1, A2, A3, A4),
              A1 == @, clone(c1, A4), nextClone(c1, c2), clone(c2, A5).

//i -> i
S1(J, B1, A2, A3, A4) :- S1(JJ,A1,A2,A3,A4), J<=A1, J/\A2=@,
              pick(J,A1,B1).
S2(J, B1, B2, A3, A4) :- S2(JJ,A1,A2,A3,A4), J<=A1\/A2, J/\A3=@,
              pick(J,A1,B1), pick(J,A2,B2).
S3(J, B1, B2, B3, A4) :- S3(JJ,A1,A2,A3,A4), J<=A1\/A2\/A3,
              J/\A4=@, pick(J,A1,B1), pick(J,A2,B2), pick(J,A3,B3).

GOOD(X) :- S1(J, @, @, @, @).
```

I measured the evaluation time of this example in four different situations. Table 5.1 shows the results. All the numbers are real seconds, not CPU seconds, the test

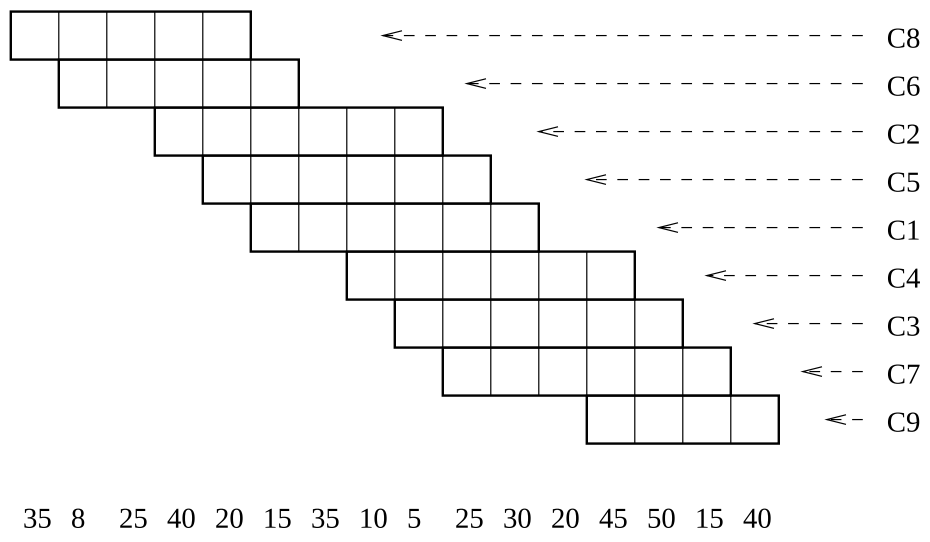

Figure 5.4: The clones of the bigger example

was running under WinNT (Pentium II 267 MHz). As can be seen the Semi-Naive evaluation is a great improvement, we need only the 5.69 percent (optimization off) or 8.55 percent (optimization on) of the time as the time of the Naive evaluation. The optimization has a remarkable effect with Naive evaluation (we save 36 percent of the time), and a a slight effect if optimization is on (4.8 percent). The reason of the small effect is that the original rules are rather optimized, there is no much possibility to optimize the rules more. However it has to be mentioned that the time of the optimization (less than 1 second) is much more smaller than this small effect, therefore the optimization is useful.

Example with input size n=16, k=5

The original example described in [7] was also tested in this system. The clones and the fragments are also shown in Figure 5.4. Table 5.1 shows the test result of this example too.

5.3 Unavoidable Sets

5.3.1 Problem

The problem originates from two-player games, such as chess for instance. We can assign different labels to different positions. In chess we can use labels like: white wins, black wins, draw. It is possible that a position has no labels assigned. If we define more labels, then it is also possible that more then one labels are assigned to a position. For instance if we define labels like: white has a queen, white has a rook, white has a bishop, then if white has two rooks and a queen, and no bishops, then the first two labels are assigned to the position.

Assume that white wants to reach a position which has a specific label, and black wants to avoid it. We can build a tree which contain the possible positions, the current position is the root, and there is a directed edge between two positions if one player can move from one position to the other. If the players turn in alternate, then this graph is a bipartite graph (one position contains the name of the player who will turn next). We assume that the graph is an acyclic graph. In chess this is really acyclic, because if the same position occurs thrice, then the game is draw.

Our goal is to calculate those labels, which are unavoidable by black, if white wants to reach the label.

5.3.2 Solution

We can assign labels to the leaves. To calculate the labels for the other nodes, we do the following. If black has to move, then we calculate the intersection of the labels

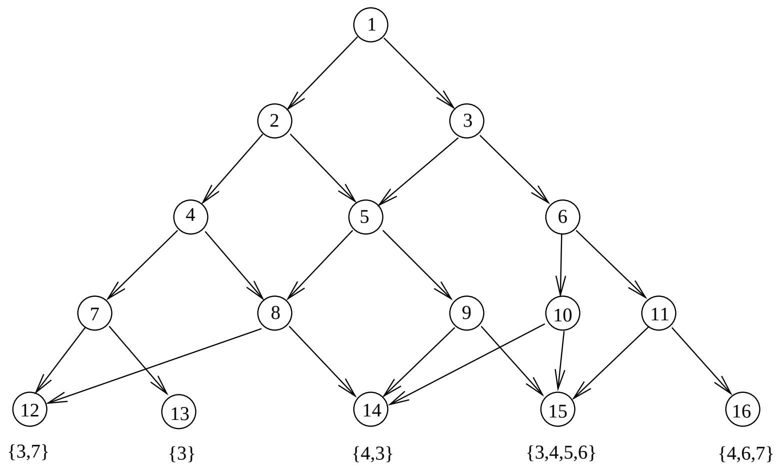

Figure 5.5: The acyclic graph

assigned to the children of the node, because black wants to avoid the label. If white has to move, then we calculate the union of the labels assigned to the children of the node, because white wants to reach the label. If we assign labels one after the each other, then after finite steps we reach root, because the graph is acyclic.

Mathematically the problem is the following: We have a directed acyclic bipartite graph. Let A and B the two disjunct sets of vertices. The graph has a special vertex for which the in-degree equals zero. We call this vertex root. Let us suppose that root is in A. Sets are assigned to the leaves. If the sets of all the children of a vertex are already defined, then we can assign a set to the vertex. If the vertex is in A, then we assign the union of the sets of the children, if the vertex is in B then the intersection of the sets of the children. The goal is to find the set which is assigned to root.

5.3.3 Example

In this example the graph and the labels assigned to the leaves are shown in Figure 5.5.

To store the structure of the graph, we need the first part of the input file:

```
left(P,C)  :- P == {1}, C == {2}.
right(P,C) :- P == {1}, C == {3}.
left(P,C)  :- P == {2}, C == {4}.
right(P,C) :- P == {2}, C == {5}.
left(P,C)  :- P == {3}, C == {5}.
right(P,C) :- P == {3}, C == {6}.
left(P,C)  :- P == {4}, C == {7}.
right(P,C) :- P == {4}, C == {8}.
left(P,C)  :- P == {5}, C == {8}.
right(P,C) :- P == {5}, C == {9}.
left(P,C)  :- P == {6}, C == {10}.
right(P,C) :- P == {6}, C == {11}.
left(P,C)  :- P == {7}, C == {12}.
right(P,C) :- P == {7}, C == {13}.
left(P,C)  :- P == {8}, C == {12}.
right(P,C) :- P == {8}, C == {14}.
left(P,C)  :- P == {9}, C == {14}.
right(P,C) :- P == {9}, C == {15}.
left(P,C)  :- P == {10}, C == {14}.
right(P,C) :- P == {10}, C == {15}.
left(P,C)  :- P == {11}, C == {15}.
right(P,C) :- P == {11}, C == {16}.
```

To store the sets of the leaves we need the following part:

```
white(X, S) :- X == {12}, S == {3,7}.
white(X, S) :- X == {13}, S == {3}.
white(X, S) :- X == {14}, S == {3,4}.
white(X, S) :- X == {15}, S == {3,4,5,6}.
white(X, S) :- X == {16}, S == {4,6,7}.
```

The part which calculates the new sets contains only three rules:

```
black(X, S) :- left(X, L),right(X, R),white(L, W1),white(R, W2), S=W1/\W2.
white(X, S) :- left(X, L),right(X, R),black(L, B1),black(R, B2), S=B1\/B2.
un(S) :- white(X, S), X == {1}.
```

If we give this input file to the system we get the result, that the set of unavoidable labels is $\{3, 4\}$. Table 5.1 shows the used time during evaluation. This example also shows the advantage of the Semi-Naive evaluation. Because the number of iterations are relatively small in this example, the effect is not too big. This example also shows that the optimization method may worsen a formula. This is very rare, the problem is that in this example the right-hand side of `black` and `white` rules contains four relations and a selection, and in most cases it is useful to evaluate selections before join, in this special case join before selection would have been better.

Chapter 6

Multisets

6.1 Introduction

A natural extension of the program is using multisets instead of sets. Multisets are similar data structures to sets, the difference is that a set can contain an element at most once, while a multiset can contain several copies of an element. Unfortunately, multisets do not form a Boolean Algebra. However, multisets can be implemented using a limited set of multiset operators, and applying other restrictions.

6.2 Extension of the program

We allow the following multiset operators:

- $V_1 \subseteq V_2$

  Where V_1, and V_2 are multiset variables. With this operator we can check whether one multiset variable is a subset of another multiset variable or not.

- $V = M$

$V_1 == V_2$	$V_1 \subseteq V_2, V_2 \subseteq V_1$
$V \subseteq E$	$V_2 = E, V \subseteq V_2$
$V \supseteq E$	$V_2 = E, V \supseteq V_2$
$V == \emptyset$	$V_2 = \emptyset, V \subseteq V_2$
$V == E$	$V_2 = E, V \subseteq V_2, V_2 \subseteq V$

Table 6.1: Other operators which can be expressed

Where V is a multiset variable, and M is a concrete multiset. With this operator we can change the value of the multiset variable.

- $V_1 = V_2 - V_3$

  Where V_1, V_2, and V_3 are multiset variables. The value of V_1 is calculated using the already known value of V_2 and V_3.

Although we allow only these three operators, some other operators can be expressed with these operators. Table 6.1 shows operators which can be expressed using the basic operators. In the table V_i's are multiset variables, E is a multiset constant.

All the multiset variables and constants are denoted with a '*' sign in the input file. There are also other restrictions related to multisets:

- At most one multiset variable in every relation.
- Only the first variable can be a multiset

6.3 An example

6.3.1 The problem

The next example is also related to the genome maps, and described in [5]. In this previous genome map example we cut the DNA with an enzyme, and get clones, after this used an other enzyme to digest the clones. In this example at the beginning we have two enzymes. We cut the original DNA with one of them and get the so called row clones, cut the original with the other enzyme and get the so called column clones. Neither the row clones nor the column clones can overlap each other. After this we use the same enzyme to digest both the row and column clones.

6.3.2 The solution

Because of the genetic difference, we know the first row, and the first column clone. The structure of the row and column clones implies that one of these two first clones is a subset of the other one. Assume the the first column clone is a subset of the first row clone. We also know that at the beginning of the DNA, there are the fragments of this column clone (which are fragments of the first row clone also), and following this, that fragments of the first row clone which are not in the first column clone. Let S be the set of these fragments. The algorithm should find an other column clone, which is either a subset of S, or a superset of S. If the column clone is a subset of S, it means that we still have more fragments from the row clones than from the column clones, hence we need to find an other column clone. If the column clone is a superset of S, it means that we have more fragments from the column clones than from the row clones, hence we need to find a row clone now. The difference of the column clone and S will be the new value of S. We can repeat this step, until S equals the empty set, which means, that we find the same fragments in both the row and the column clones. S will be empty at the end of the DNA, although there is a small possibility that S will be empty before that.

The *Datalog with Boolean Constraints* program, which can solve this problem:

```
down(*SS,UC,UR) :- initialc(*SS,UC,UR).
right(*SS,UC,UR) :- initialr(*SS,UC,UR).
down(*SS,UC,URR) :- down(*S,UC,UR), row(*R,M), *R <= *S,
                    *SS = *S - *R, pick(M,UR, URR).
down(*SS,UCC,UR) :- right(*S,UC,UR), column(*C,N), *S <= *C,
                    *SS = *C - *S, pick(N,UC,UCC).
right(*SS,UCC,UR) :- right(*S,UC,UR), column(*C,N), *C <= *S,
                     *SS = *S - *C, pick(N,UC,UCC).
right(*SS,UC,URR) :- down(*S,UC,UR), row(*R,M), *S <= *R,
                     *SS = *R - *S, pick(M,UR,URR).
halt(X) :- down(*X,@,@), *Y == *@, *X <= *Y.
halt(X) :- right(*X,@,@), *Y == *@, *X <= *Y.
```

6.3.3 Concrete example

In this example the DNA contains 28 fragments, ten row and nine column clones. Figure 6.1 shows the fragments of the DNA string, the row and column clones.

Table 6.2 shows the process of the solution. The first column shows the expression of the new value of S, the second column shows the new value of S. The third and fourth columns show the unused row and column clones.

Table 5.1 shows the execution time of this example. The Semi-Naive evaluation

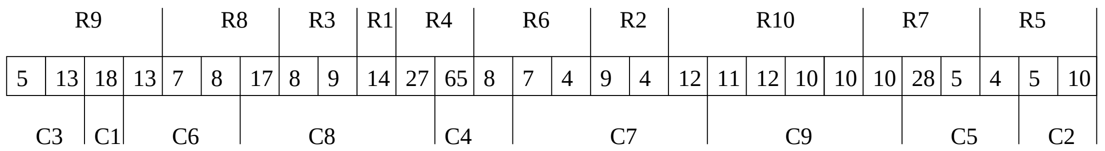

Figure 6.1: The correct order of clones

	S	Row clones	Column clones
C3	5,13	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	1, 2, 4, 5, 6, 7, 8, 9
R9 - S	13,18	1, 2, 3, 4, 5, 6, 7, 8, 10	1, 2, 4, 5, 6, 7, 8, 9
S - C1	13	1, 2, 3, 4, 5, 6, 7, 8, 10	2, 4, 5, 6, 7, 8, 9
C6 - S	7, 8	1, 2, 3, 4, 5, 6, 7, 8, 10	2, 4, 5, 7, 8, 9
R8 - S	17	1, 2, 3, 4, 5, 6, 7, 10	2, 4, 5, 7, 8, 9
C8 - S	8, 9, 14, 27	1, 2, 3, 4, 5, 6, 7, 10	2, 4, 5, 7, 9
S - R3	14, 27	1, 2, 4, 5, 6, 7, 10	2, 4, 5, 7, 9
S - R1	27	2, 4, 5, 6, 7, 10	2, 4, 5, 7, 9
R4 - S	65	2, 5, 6, 7, 10	2, 4, 5, 7, 9
C4 - S	8	2, 5, 6, 7, 10	2, 5, 7, 9
R6 - S	4, 7	2, 5, 7, 10	2, 5, 7, 9
C7 - S	4, 9, 12	2, 5, 7, 10	2, 5, 9
S - R2	12	5, 7, 10	2, 5, 9
R10 - S	10, 10, 11, 12	5, 7	2, 5, 9
C9 - S	10	5, 7	2, 5
R7 - S	5, 28	5	2, 5
C5 - S	4	5	2
R5 - S	5, 10	—	2
C2 - S	—	—	—

Table 6.2: The solution

is a great improvement in this example too (it needs only about 5 % of the time necessary for the Naive evaluation). However the optimization of Relational Algebra formulas has no important effect on the execution time of this query.

Chapter 7

Further Work

- **GUI:** Currently the program has a character based interface, which can make difficult for the user to handle the program. The biggest disadvantage of the character based interface is that sometimes it is too difficult to interpret the results, because the result is only a set of formulas.

  Currently a student (Song Liu) in the Department is working on a Graphical User Interface which will make easier to understand the results.

- **Approximation:** Every elimination method have some restrictions on the input database. Sometimes we cannot use any quantifier elimination methods. In these cases approximation may help, when we cannot compute the correct quantifier-free formula, rather only create a formula which approximate the result. A possible way of approximation is described in [3].

- **Multiset:** Chapter 6 describes the an extension of the system, which makes possible to use multisets. Only a small subset of multiset operators are used in this extension, it would be possible to implement more multiset operators.

- **Indexing:**

  Indexing can improve the speed of the database systems. However indexing is not too difficult in traditional relational database systems, it is much more difficult in constraint database systems. A good indexing method would improve the speed of this system.

Bibliography

[1] B. H. Arnold, *Logic and Boolean Algebra,*
Prentice-Hall, INC. (1962)

[2] J. Byon, P.Z. Revesz, DISCO: A constraint database system with sets, *Proc. Workshop on Constraint Databases and Applications,* pages 68–83 (1995), Springer-Verlag, Berlin/Heidelberg/New York.

[3] Richard Helm, Kim Marriott, Martin Odersky, Spatial Query Optimization: From Boolean Constraints to Range Queries, *Journal of Computer and System Sciences 51*, 197–201 (1995)

[4] P.C. Kanellakis, G.M. Kuper, P.Z. Revesz, Constraint query languages. *Journal of Computer and System Sciences* 51:26–52. (1995)

[5] Peter Z. Revesz, *CSE 913 (Advanced Database Systems) class material,* University of Nebraska-Lincoln, Spring 1998.

[6] Peter Z. Revesz, The Evaluation and the Computational Complexity of Datalog Queries of Boolean Constraint Databases, *International Journal of Algebra and Computation, to appear.*

[7] Peter Z. Revesz, Refining Restriction Enzyme Genome Maps, *Constraints 2, 361–375 (1997)*

[8] J. D. Ullman, *Principles of Database and Knowledge-base Systems* Computer Science Press (1988).