

# DDB trees: a basis for deductive database explanations

Sarah Mallet and Mireille Ducassé

IRISA/INSA  
Campus Universitaire de Beaulieu  
F - 35042 Rennes Cedex, France  
{smallet, ducasse}@irisa.fr

Keywords: debugging, explanations, deductive databases, logic programming

## Abstract

The power of deductive systems in general is that programs express what should be done and not how it should be done. Nevertheless, deductive systems need debugging and explanation facilities. Indeed, their operational semantics is less abstract than the declarative semantics of the programs. If users have to understand all the low level details of the operational semantics much of the benefits of using a deductive system is lost.

Existing explanation systems for deductive databases produce proof trees to be shown to users. Although useful, proof trees give a fragmented view of query evaluations, and users face a, most of the time large, forest of proof trees.

We propose a new data structure, called the DDB tree, which merges the information of a proof tree forest into one concise tree. A DDB tree gives a global picture of a query evaluation in a dramatically reduced structure with no loss of information. DDB trees can be shown to users or can be analyzed further by an explanation system.

## 1 Introduction

Deductive databases are based on a logical data model, in contrast with relational data models of more classical databases. Roughly speaking, a deductive database can be seen as a relational database (referred to as the database in the rest of the introduction) which contains facts, together with a deductive engine which can deduce new facts from the actual ones and a set of rules. A significant advantage of deductive databases is that they allow the full power of recursivity to be enacted [9].

The advantage of deductive systems in general is that programs express **what** should be done and not **how** it should be done. Hence people face more abstract programs and can better concentrate on the essential features of their applications. Nevertheless, deductive systems need debugging and explanation facilities. Indeed, the deductive engine works with an opera-

tional semantics less abstract than the declarative semantics of the programs. If users of deductive systems have to understand all the low level details of the operational semantics much of the benefits of using a deductive system is lost.

The existing explanation systems for deductive databases, proposed by Wieland [15], Specht[12] and Arora and al.[1], produce proof trees to be shown to users. A proof tree recapitulates the steps which enabled to deduce a given fact, showing the rules and the database facts used in the process.

To our point of view, proof trees provide useful information, but they are not quite accurate. First of all, they give a fragmented view of the accesses to the database, pretending that data is retrieved from the database one piece at a time. Furthermore, for each deduced fact a proof tree has to be shown, whereas much of the deduction can be common between several deduced facts. Thus people face a forest of proof trees, which can be very large and not easy to analyze.

Moreover, the already cited systems present proof trees as the end result of an explanation system. To our point of view there is much more analysis to be done by a system before showing information to users.

Our ultimate aim is to build a “trace” analyzer for deductive database systems based on the same principles as Opium, a trace analyzer for Prolog [3]. In Opium, a trace models a Prolog execution and is a representation of an SLD tree.

For deductive databases, as discussed earlier, proof trees are not satisfactory to model query evaluations. Hence we defined a new data structure, called the DDB tree. A DDB tree is similar to an SLD tree, where all branches lead to a success (there is no backtracking in deductive database evaluations) and *substitution sets* represent the accesses to the database.

The main advantage of DDB trees over proof trees is that they avoid the forest of trees, merging the common deduction steps with no loss of information. As will be shown on the examples, the size of the resulting information is dramatically reduced with no loss of information. Furthermore, as the image of the database accesses are more faithful to the reality, the information should be more helpful to users.

In a first part, we briefly present the context of deductive databases. We, then, describe the existing explanation systems. In the last part we introduce the definition of substitution sets, DDB derivations and DDB trees.

## 2 The context of deductive databases

In this section, we briefly describe Datalog, the main language of deductive databases. Finally we introduce the notion of proof tree, a data structure used in explanation systems.

## 2.1 Datalog

A deductive database is composed of three parts: a logic program, a database of facts and a set of constraints. The predicates defined in the logic program make up the **intentional** database, those defined in the database make up the **extensional** database. The implementation of the database part is not detailed in what follows; we will suppose that facts of the database are totally instantiated.

The languages used in deductive databases are very often extensions of *Datalog*. A Datalog program is a set of clauses of the form :

$$L : -L_1, \dots, L_n.$$

L is the head of the clause, the  $L_i$  compose the body. In pure Datalog,  $L_i$  is a positive literal of the form  $P(t_1, \dots, t_n)$  where P is a predicate symbol and the  $t_i$ , called terms, are constants or variables. By convention, terms beginning with an upper case letter are variables. Goals of the form  $:- B_1, \dots, B_n$ . are used to query the program. Note that in pure Datalog there are no functors but they have been introduced as extensions. A precise description of Datalog and its extensions can be found in Ullman's book [13].

*Program (intentional database)*

```
(r1)  trip(begin(X), end(Y), cost(P)) :-
      transit(X,Y,P).

(r2)  transit(X, Y, P) :-
      direct(X,Y,P),
      ok(P).

(r3)  transit(X, Y, P) :-
      direct(X,Z,PA),
      ok(PA),
      transit(Z, Y, PB),
      P is PA + PB.

(r4)  ok(P) :-
      P < 15.
```

*Database (extensional database)*

```
direct(a,b,6).    direct(b,c,3).    direct(e,f,2).
direct(f,b,3).    direct(b,l,7).
                  direct(b,k,18).
```

Figure 1: An example of extended Datalog program (with functors)

Figure 1 shows an example of a deductive program and its associate database. There exists a *trip* with a cost P from a begin point X to an end

point Y, if a *transit* between these two points is possible; that is to say either the trip is *direct* and the cost of the trip is *ok*, that is to say that the cost is lower than 15, either there exists an intermediate point with a cost to go there *ok*. The total cost P is the sum of cost of each stage.

With respect to Datalog terminology, this program contains four clauses which define three predicates. The first clause defines *trip/3*. The next two clauses define *transit/3*. The last clause defines *ok/1*. The extensional database is composed of one predicate *direct/3* defined by six tuples. The evaluation of extensional predicates leads to database accesses.

If the query  $trip(D,F,C)$  is asked, the system produces the set of facts:  $trip(begin(a), end(b), cost(6))$ ,  $trip(begin(b), end(c), cost(3))$ ,  $trip(begin(b), end(l), cost(7))$ ,  $trip(begin(e), end(f), cost(2))$ ,  $trip(begin(f), end(b), cost(3))$ ,  $trip(begin(a), end(c), cost(9))$ ,  $trip(begin(a), end(l), cost(13))$ ,  $trip(begin(e), end(b), cost(5))$ ,  $trip(begin(f), end(l), cost(10))$ ,  $trip(begin(f), end(c), cost(6))$ ,  $trip(begin(e), end(c), cost(8))$ ,  $trip(begin(e), end(l), cost(12))$ .

## 2.2 Proof trees

There exists different deductive database evaluation methods (cf Ullman [14]). Independently from these methods, proofs that a fact can be deduced from the program and data can be represented by proof trees. The nodes of a proof tree are goals used during query evaluation. The sons of a node are connected by an AND relation. To be proven, a goal needs to have all his sons proven. The leaves are, in the case of deductive databases, tuples of the database, facts of the program or built-in predicates. Figure 2 represents the proof tree related to the fact  $trip(begin(a),end(b),cost(6))$ .

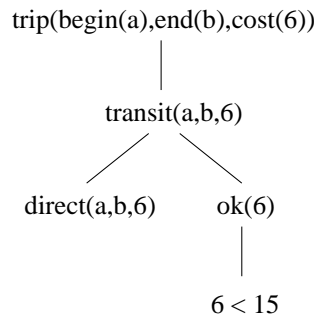


Figure 2: Proof tree related to the fact  $trip(begin(a),end(b),cost(6))$

In this tree,  $transit(a,b,6)$  is deduced from the conjunction of  $direct(a,b,6)$  and  $ok(6)$ , thanks to the second rule of the program of Figure 1.

### 3 Explanations and deductive databases

One of the first debugging systems for deductive databases has been proposed by Shmueli and Tsur [10]. They describe a *diagnosis system* for the deductive database LDL [7]. The principle of this system is to consider the user as an oracle who has an intended interpretation of the deductive program. By querying the user, the system will determine why a fact is missing or why it is wrong. In the case of a wrong fact, the system uses proof trees to direct the diagnosis.

Other *explanation systems* for deductive databases have a different point of view: they construct a trace of the query evaluation that the user can consult.

The first system has been developed for Dedex [6] by Wieland [15]. The proposed explanations are based on proof trees that can be visualized totally or partially. The implementation consists of a dedicated execution mode of Dedex that produces explanations. It decomposes the query evaluation according to the choice of the user.

The *Explain* system of Arora and al. [1] has been developed for CORAL [8]. As the previous system, the chosen structure is a proof tree. *Explain* proposes visualization with zooming possibilities on the different proof trees. *Explain* system consists of adding in CORAL a program which stores information on derivations during query evaluation.

The third system, proposed by Specht [11] for LOLA [4], also uses proof trees as explanations. The implementation principle is a transformation of the user program to insert trace generation as opposed to the other systems which modify deductive engine. The transformed program is queried as usual with the deductive database.

Most of the mixed evaluation and optimization methods, are implemented by program transformations. The queried program is then the optimized one. Sometimes it can be interesting to have explanations on the transformed program, to understand, for example, the applied transformations. But, most of the time, users prefer seeing a trace of their initial program. The system Explain allows only to construct the trace of the transformed program. Wieland's and Specht's system show a trace in terms of the original program, more interesting for users.

The implementations of Explain and Wieland's system modify the deductive database evaluation system. They are not portable. Specht's system transforms the program before optimizations take place. It is therefore independent of the implementation of the deductive database system, and more portable. There are, on the other hand, no information on performance.

A common feature of the three explanation systems is the structure of the traces (or explanations): they all use a forest of proof trees. For each produced fact, there are one or several associated proof trees. Figure 3 shows the forest of the query  $trip(D, F, C)$ .

The whole presented information is interesting but, this form is hardly

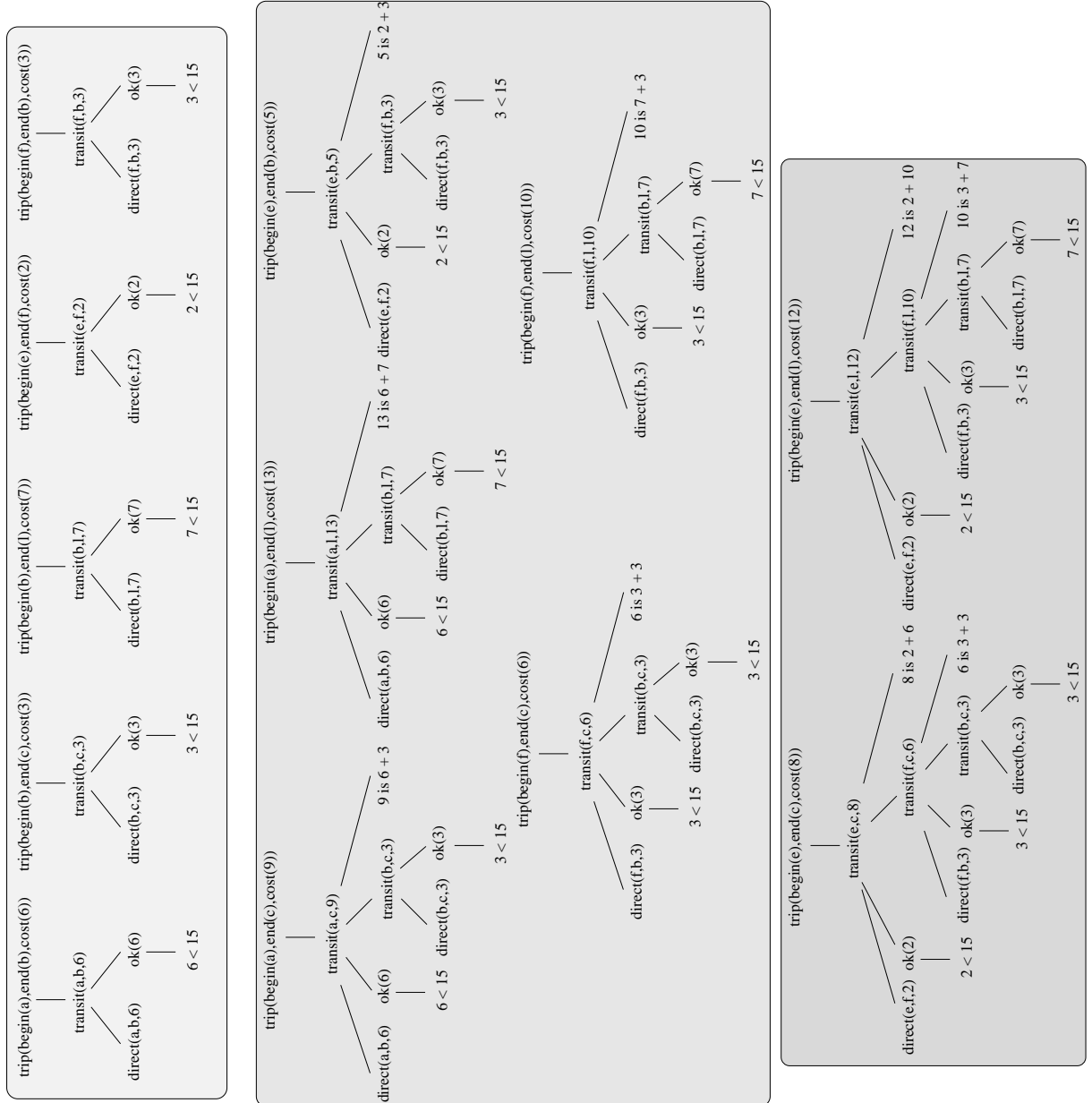


Figure 3: Forest of proof trees for the query  $trip(D, F, C)$  grouped by proof structure



















