Surveys
in Computer
Science

S. Ceri G. Gottlob L. Tanca

Logic Programming and Databases

逻辑程序设计和数据库 [英]

Springer-Verlag
World Publishing Corp

S. Ceri G. Gottlob L. Tanca

Logic Programming and Databases

With 42 Figures

Springer-Verlag
World Publishing Corp

Stefano Ceri

Dipartimento di Matematica
Università di Modena
Via Campi 213
I-41100 Modena

Georg Gottlob

Institut für Angewandte Informatik
und Systemanalyse
Abteilung für Verteilte Datenbanken
und Expertensysteme
Technische Universität Wien
Paniglgasse 16/181
A-1040 Wien

Letizia Tanca

Dipartimento di Elettronica
Politecnico di Milano
Piazza Leonardo Da Vinci 32
I-20133 Milano

ISBN 3-540-51728-6 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-51728-6 Springer-Verlag New York Berlin Heidelberg

Library of Congress Cataloging-in-Publication Data.
Ceri, Stefano, 1955-
Logic programming and databases / S. Ceri, G. Gottlob, L. Tanca. p. cm. -
(Surveys in computer science)
Includes bibliographical references.
ISBN 0-387-51728-6 (U.S.)
1. Logic programming. 2. Data base management. I. Gottlob, G. (Georg).
II. Tanca, L. (Letizia). III. Title. IV. Series. QA76.63.C47 1990 005.74-dc20
89-28960 CIP

Reprinted by World Publishing Corporation, Beijing, 1992
for distribution and sale in The People's Republic of China only
ISBN 7-5062-1127-0

Preface

The topic of logic programming and databases has gained increasing interest in recent years. Several events have marked the rapid evolution of this field: the selection, by the Japanese Fifth Generation Project, of *Prolog* and of the relational data model as the basis for the development of new machine architectures; the focusing of research in database theory on logic queries and on recursive query processing; and the pragmatic, application-oriented development of expert database systems and of knowledge-base systems. As a result, an enormous amount of work has been produced in the recent literature, coupled with the spontaneous growth of several advanced projects in this area.

The goal of this book is to present a systematic overview of a rapidly evolving discipline, which is presently not described with the same approach in other books. We intend to introduce students and researchers to this new discipline; thus we use a plain, tutorial style, and complement the description of algorithms with examples and exercises. We attempt to achieve a balance between theoretical foundations and technological issues; thus we present a careful introduction to the new language *Datalog*, but we also focus on the efficient interfacing of logic programming formalisms (such as *Prolog* and *Datalog*) with large databases.

The book is divided into three parts, preceded by two preliminary chapters. Chapter 1 offers an overview of the field. Chapter 2 discusses those aspects of the relational model and *Prolog* which are required for understanding the rest of the book. Of course, Chapter 2 is not a complete tutorial on these fields; it just redefines terminology, notation, and basic concepts in order to keep the book self-contained. However, in order to fully understand the problems, the reader should have some background in these subjects.

Part I is devoted to the coupling of *Prolog* with relational databases. Chapter 3 presents *Prolog* as a query language, applied to the formulation of two classical problems, the ***anti-trust*** and the *bill-of-materials* problems. Chapter 4 describes the alternative architectures and techniques for coupling a *Prolog* system to a relational database. Chapter 5 presents a review of the major current projects and prototypes for coupling *Prolog* and relational databases.

Part II is devoted to the precise definition of the *Datalog* language. Chapter 6 defines formally the syntax and semantics of *Datalog*. The semantics of *Datalog* is described by a nonprocedural, model-theoretic approach. Chapter 7 presents the proof theory of the language, by introducing an algorithm for evaluating *Datalog* goals, and by showing that the method is sound and complete with respect to the model-theoretic semantics. Chapter 7 also introduces two other paradigms for the evaluation of *Datalog* programs: *fixpoint theory* and *backward chaining*. In particular, resolution and SLD-resolution are defined in the context of *Datalog*.

Part III is devoted to the description of query optimization techniques for *Datalog*. Chapter 8 presents a general classification of the optimization techniques; we distinguish *rewriting methods*, which assume as input a *Datalog* program and produce as output an optimized *Datalog* program, from *evaluation methods*, which assume as input a *Datalog* program and produce the result of the query. Furthermore, we show a simple translation from *Datalog* programs to systems of algebraic equations. This translation enables us to describe a class of algebraic methods for query optimization. These arguments are then studied in the subsequent Chapters 9 and 10. Chapter 9 deals with evaluation methods, and presents both bottom-up and top-down evaluation methods (including the Naive, Semi-naive, and Query-Subquery methods). Chapter 10 deals with rewriting methods, and presents the logical rewriting methods (including the Magic Set, Counting, and the Static Filtering methods) and the algebraic rewriting methods.

Chapter 11 deals with extensions to pure *Datalog*, such as sets and negation. This chapter should be considered as an introduction to the subject, rather than a full treatment. Finally, Chapter 12 presents an overview of the main projects on the integration of logic programming and databases, including *Nail*, *LDL*, and the Fifth Generation Project.

The book is organized so that the three parts can be read independently by different readers. In fact, the various chapters are rather independent.

This book does not present a full overview of all topics which belong to the field of deductive databases. For instance, it does not deal with incompleteness, disjunctive data, or the validation of integrity constraints. We apologize to those who find their favorite topics missing, in particular the community of logicians who work in the area of deductive databases. In the choice of arguments, we have concentrated our attention on the use of *large* databases; loyal to the tradition of the database community, we are mainly concerned with the efficiency of database access, even when we use logic programming as a query language.

This book is primarily the outcome of research work conducted by the authors in cooperation with other colleagues. We wish to thank Gio Wiederhold for his contribution to the CGW approach, which was developed in the framework of the KBMS project at Stanford University; Stefano Crespi-Reghizzi, Gianfranco Lamperti, Luigi Lavazza, and Roberto Zicari, for their contribution to the development of the algebraic approach to logic queries within the framework of the ALGRES project; Silvia Cozzi, Fabrizio Gozzi, Marco Lugli, and Guido Sanguinetti, who developed the PRIMO system as part of their diploma theses at the University of Modena; and Roberta Cantaroni, Stefania Ferrari, and Franca Garzotto, who have addressed with us problems related to logic databases.

Many colleagues and students have produced useful comments in extending, reviewing, and correcting the manuscript; among them, we wish to thank Maurice Houtsma, who has made a very careful reading, suggesting several corrections and improvements; Hervé Gallaire, Jean-Marie Nicolas, Johann Christoph Freytag, and François Bry, who have provided useful information concerning the entire manuscript and more specifically about the projects developed at ECRC; Shamin Naqvi has also made specific comments about the LDL project developed at MCC; Wolfgang Nejdl has provided us with material about the QSQ method and its modifications. Werner Schimanovich and Alex Leitsch have helped us with encouragement and interesting discussions.

Particular appreciation is given to Gunter Schlageter and to his PhD students and to Renate Pitrik, Wilhelm Rossak, and Robert Truschnegg, whose careful reading and critical review has improved the quality of the book. Remaining errors and omissions are, of course, the responsibility of the authors.

We would like to thank our home institutions for providing support and equipment for editing this manuscript: the University of Modena, the Politecnico di Milano, the Technical University of Wien, and Stanford University. During the preparation of the manuscript, Letizia Tanca was supported by a grant from C.I.L.E.A. The accurate and fast final preparation of this

manuscript has been supervised by Dr. Hans Wössner, Ingeborg Mayer, and the copy editor Dr. Gillian Hayes, from Springer-Verlag.

The "Progetto Finalizzato Informatica e Calcolo Parallelo" of the Italian National Research Council, starting in 1989, will provide us with a research environment for realizing many of the ideas presented in this book.

October 1989

Stefano Ceri
Georg Gottlob
Letizia Tanca

Table of Contents

Chapter 9

Evaluation Methods 145

Chapter 10

Rewriting Methods 163

Chapter 11

Extensions of Pure Datalog 208

Chapter 1

Logic Programming and Databases: An Overview

This book deals with the *integration of logic programming and databases to generate new types of systems,* which extend the frontiers of computer science in an important direction and fulfil the needs of new applications. Several names are used to describe these systems:

a) The term *deductive database* highlights the ability to use a logic programming style for expressing deductions concerning the content of a database.
b) The term *knowledge base management system (KBMS)* highlights the ability to manage (complex) knowledge instead of (simple) data.
c) The term *expert database system* highlights the ability to use expertise in a particular application domain to solve classes of problems, but having access over a large database.

The confluence between logic programming and databases is part of a general trend in computer science, where different fields are explored in order to discover and profit from their common concepts.

Logic programming and databases have evolved in parallel throughout the seventies. *Prolog*, the most popular language for PROgramming in LOGic, was born as a simplification of more general theorem proving techniques to provide efficiency and programmability. Similarly, the relational data model was born as a simplification of complex hierarchical and network models, to enable set-oriented, nonprocedural data manipulation. Throughout the seventies and early eighties, the use of both *Prolog* and relational databases has become widespread, not only in academic or scientific environments, but also in the commercial world.

Important studies on the relationships between logic programming and relational databases have been conducted since the end of the seventies, mostly from a theoretical viewpoint. The success of this confluence has been facilitated by the fact that *Prolog* has been chosen as the programming language paradigm within the Japanese *Fifth Generation Project.* This project aims at the development of the so-called "computers of the next generation", which will be specialized in the execution of Artificial Intelligence applications, hence capable of performing an extremely high number of *deductions per time unit.* The project also includes the use of the relational data model for storing large collections of data.

The reaction to the Japanese *Fifth Generation Project* was an incentive to research in the interface area between logic programming and relational databases. This choice indicated that this area is not just the ground for theoretical investigations, but also has great potential for future applications.

By looking closely at logic programming and at database management, we discover several features in common:

a) *DATABASES.* Logic programming systems manage small, single-user, main-memory databases, which consist of deduction rules and factual information. Database systems deal instead with large, shared, mass-memory data collections, and provide the technology to support efficient retrieval and reliable update of persistent data.
b) *QUERIES.* A query denotes the process through which relevant information is extracted from the database. In logic programming, a query (or *goal*) is answered by building chains of deductions, which combine rules and factual information, in order to *prove* or *refute* the validity of an initial statement. In database systems, a query (expressed through a special-purpose data manipulation language) is processed by determining the most efficient access path in mass memory to large data collections, in order to extract relevant information.
c) *CONSTRAINTS.* Constraints specify correctness conditions for databases. Constraint validation is the process through which the correctness of the database is preserved, by preventing incorrect data being stored in the database. In logic programming, constraints are expressed through general-purpose rules, which are activated whenever the database is modified. In database systems, only a few constraints are typically expressed using the data definition language.

Logic programming offers a greater power for expressing queries and constraints as compared to that offered by data definition and manipulation languages of database systems. Furthermore, query and constraint representation is possible in a homogeneous formalism and their evaluation requires the same inferencing mechanisms, hence enabling more sophisticated reasoning about the database content. On the other hand, logic programming systems do not provide the technology for managing large, shared, persistent, and reliable data collections.

The natural extension of logic programming and of database management consists in building new classes of systems, placed at the intersection between the two fields, based on the use of *logic programming as a query language*. These systems combine a logic programming style for formulating queries and constraints with database technology for efficiency and reliability of mass-memory data storage.

1.1 Logic Programming as Query Language

We give an informal presentation of how logic programming can be used as a query language. We consider a relational database with two relations:

$$PARENT(PARENT, CHILD), \quad \text{and} \quad PERSON(NAME, AGE, SEX).$$

The tuples of the $PARENT$ relation contain pairs of individuals in parent-child relationships; the tuples of the $PERSON$ relation contain triples whose first,

PARENT	
PARENT	*CHILD*
john	*jeff*
jeff	*margaret*
margaret	*annie*
john	*anthony*
anthony	*bill*
anthony	*janet*
mary	*jeff*
claire	*bill*
janet	*paul*

PERSON		
NAME	*AGE*	*SEX*
paul	*7*	*male*
john	*78*	*male*
jeff	*55*	*male*
margaret	*32*	*female*
annie	*4*	*female*
anthony	*58*	*male*
bill	*24*	*male*
janet	*27*	*female*
mary	*75*	*female*
claire	*45*	*female*

Fig. 1.1. Example of relational database

second, and third elements are the person's name, age, and sex, respectively. We assume that each individual in our database has a different name. The content of the database is shown in Fig. 1.1.

We express simple queries to the database using a logic programming language. We use *Prolog* for the time being; we assume the reader has some familiarity with *Prolog*. We use two special *database predicates, parent* and *person* with the understanding that the ground clauses (facts) for these predicates are stored in the database. We use standard *Prolog* conventions on upper and lower case letters to denote variables and constants. For instance, the tuple <*john, jeff*> of the database relation PARENT corresponds to the ground clause:

$$parent(john,jeff).$$

The query: *Who are the children of John?* is expressed by the following *Prolog* goal:

$$? - parent(john, X).$$

The answer expected from applying this query to the database is:

$$X = \{jeff,\ anthony\}.$$

Let us consider now which answer would be given by a *Prolog* interpreter, operating on facts for the two predicates *parent* and *person* corresponding to the database tuples; we assume facts to be asserted in main memory in the order shown above.

The answer is as follows: After executing the goal, the variable X is first set equal to *jeff*; if the user asks for more answers, then the variable X is set equal to *anthony*; if the user asks again for more answers, then the search fails, and the interpreter prompts *no*. Note that *Prolog* returns the result one tuple at a time, instead of returning the set of all result tuples.

The query: *Who are the parents of Jeff?* is expressed as follows:

$$? - parent(X,jeff).$$

The set of all answers is:

$$X = \{john, mary\}.$$

Once again, let us consider the *Prolog* answer: After executing this goal, the variable X is set equal to *john*; if the user asks for more answers, then the variable X is set equal to *mary*; if the user asks again for more answers, then the search fails.

We can also express queries where all arguments of the query predicate are constants. For instance:

$$? - parent(john,jeff).$$

In this case, we expect a positive answer if the tuple *<john, jeff>* belongs to the database, and a negative answer otherwise. In the above case, a *Prolog* system would produce the answer *yes*.

Rules can be used to build an *Intensional Database (IDB)* from the *Extensional Database (EDB)*. The EDB is simply a relational database; in our example it includes the relations PARENT and PERSON. The IDB is built from the EDB by applying rules which define its content, rather than by explicitly storing its tuples. In the following, we build an IDB which includes the relations FATHER, MOTHER, GRANDPARENT, SIBLING, UNCLE, AUNT, ANCESTOR, and COUSIN. Intuitively, all these relationships among persons can be built from the two EDB relations PARENT and PERSON.

We start by defining the relations FATHER and MOTHER, by indicating simply that a father is a male parent and a mother is a female parent:

$$father(X,Y) \ :- \ person(X,\_,male),\ parent(X,Y).$$
$$mother(X,Y) \ :- \ person(X,\_,female),\ parent(X,Y).$$

As a result of this definition, we can deduce from our sample EDB the IDB shown in Fig. 1.2.

Note that here we are presenting the tuples of the IDB relations as if they actually existed: in fact, tuples of the IDB are not stored. One can regard the two rules *father* and *mother* above as *view definitions*, i.e., programs stored in the database which enable us to build the tuples of *father* starting from the tuples of *parent* and *person*.

The IDB can be queried as well; we can, for instance, formulate the query: *Who is the mother of Jeff?*, as follows:

$? - mother(X,jeff).$

FATHER	
FATHER	*CHILD*
john	*jeff*
jeff	*margaret*
john	*anthony*
anthony	*bill*
anthony	*janet*

MOTHER	
MOTHER	*CHILD*
margaret	*annie*
mary	*jeff*
claire	*bill*
janet	*paul*

Fig. 1.2. The IDB relations FATHER and MOTHER

With a *Prolog* interpreter, after the execution of this query X is set equal to *mary*. Notice that the interpreter does not evaluate the entire IDB relation MOTHER in order to answer the query, but rather it finds just the tuple which contributes to the answer.

We can proceed with the definition of the IDB relations GRANDPARENT, SIBLING, UNCLE, and AUNT, with obvious meanings:

$grandparent(X, Z)\ :-\ parent(X, Y),\ parent(Y, Z).$
$sibling(X, Y)\ :-\ parent(Z, X),\ parent(Z, Y),\ not(X = Y).$
$uncle(X, Y)\ :-\ person(X, \_, male),\ sibling(X, Z),\ parent(Z, Y).$
$aunt(X, Y)\ :-\ person(X, \_, female),\ sibling(X, Z),\ parent(Z, Y).$

Complex queries to the EDB and IDB can be formulated by building new rules which combine EDB and IDB predicates, and then presenting goals for those rules; for instance, *Who is the uncle of a male nephew?* can be formulated as follows:

$query(X)\ :-\ uncle(X, Y),\ person(Y, \_, male).$
$?-query(X).$

More complex IDB relations are built from *recursive rules*, i.e., rules whose head predicate occurs in the rule body (we will define recursive rules more precisely below). Well-known examples of recursive rules are the ANCESTOR relation and the COUSIN relation.

The ANCESTOR relation includes as tuples all ancestor-descendent pairs, starting from parents.

$ancestor(X, Y)\ :-\ parent(X, Y).$
$ancestor(X, Y)\ :-\ parent(X, Z), ancestor(Z, Y).$

The COUSIN relation includes as tuples either two children of two siblings, or, recursively, two children of two previously determined cousins.