# Advances in Database Technology-EDBT '90

# Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

## 416

F. Bancilhon C. Thanos
D. Tsichritzis (Eds.)



# Advances in Database Technology – EDBT '90

International Conference on Extending Database Technology
Venice, Italy, March 26–30, 1990
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo Hong Kong

**Editors**

François Bancilhon
ALTAIR
B.P. 105, F-78153 Le Chesnay Cedex, France

Costantino Thanos
I.E.I. – C.N.R.
Via Santa Maria 46, I-56126 Pisa, Italy

Dennis Tsichritzis
Centre Universitaire d'Informatique, Université de Genève
12, rue du Lac, CH-1207 Genève, Switzerland

CR Subject Classification (1987): D.3.3, E.2, F.4.1, H.2, I.2.1, I.2.4

ISBN 3-540-52291-3 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-52291-3 Springer-Verlag New York Berlin Heidelberg



Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210 – Printed on acid-free paper

# Foreword

This is the second EDBT conference. The first took place in March 1988 in Venice and was such a success that it was decided to create a biannual event and hold the second also in Venice. The intent in creating the conferences has been to provide an international forum appropriate for the exchange of results in research, development and applications which *extend* the scope of database technology. The conferences are designed to facilitate and extend communication among researchers and between academia and industry. They are also intended to be international level European conferences providing a high quality forum for the European database research community.

The program committee received 175 papers. These came from 28 different countries, the largest contributors being the USA, West Germany, France, Italy and the United Kingdom. A wide range of topics was covered, the most popular being object-oriented systems, database and AI, logic and databases, data structures and query processing. From these 175 papers, we had the difficult task of selecting 27 papers. We also selected an invited speaker, Carlo Zaniolo who addresses the issue of deductive databases and their transition from the technology development phase to the system implementation phase.

Two panels were chosen to address some of the most burning issues of the field, the impact of theory on systems in the past and in the future, chaired by David Maier, and the two "competing" fields of object-oriented databases and of deductive databases, chaired by Michele Missikoff.

We are extremely grateful to all the people who helped organize the conference: all the members of the organization committee, all the program committee members, Florence Deshors and Pauline Turcaud for their help in organizing the paper selection and program committee work, Philippe Richard and Paris Kanellakis who helped in the paper handling process.

François Bancilhon
Costantino Thanos
Dennis Tsichritzis

## Sponsorship

Promoted by the EDBT Foundation
Sponsored by AFCET - AICA - BCS - ENEA - GI - Comitati Nazionali di Consulenza del CNR
In cooperation with ACM and IEEE
Under the patronage of the Italian Ministry for the University, Research and Technology, the Italian National Research Council, the Commission of European Communities, the Regional Government of Veneto, the University of Venice and UNESCO-ROSTE.

## Organization

Conference Chairperson: *D. Tsichritzis (Université de Genève, Switzerland)*
Organization Committee Chairperson: *C. Thanos (IEI-CNR, Italy)*
Program Committee Chairperson: *F. Bancilhon (Altaïr, France)*
Technical exhibitions coordinator: *F. Bonfatti (CIOC-CNR, Italy)*
Tutorial coordinator: *D. Sacca (Universita della Calabria, Italy)*
Conference Secretariat: *M. Mennucci (IEI - CNR, Italy)*
CEC coordinator: *G. Metakides (CEC, Belgium)*
U.S. coordinator: *M. Brodie (GTE Laboratories, USA)*
Far East coordinator: *M. Yoshikawa (Kyoto Sangyo University, Japan)*
Local arrangements coordinator: *F. Dalla Libera (Universita di Venezia, Italy)*
Treasurer: *E. Ricciardi (IEI-CNR, Italy)*
Organization Committee Members : *M. Agostini (Italy), C. Freytag (FRG), K. G. Jeffery (United Kingdom), C. Rolland (France)*

## Program Committee

S. Abiteboul (France)
M. Adiba (France)
H. Ait-Kaci (France)
F. Bancilhon (France)
C. Batini (Italy)
C. Bauzer-Medeiros (Brazil)
A. Berre (Norway)
H. Boral (USA)
M. Casanova (Brazil)
S. Ceri (Italy)
K. R. Dittrich (FRG)
C. Freytag (FRG)
G. Grahne (Finland)
T. Hadzilacos (Greece)
M. A. W. Houtsma (The Netherlands)
M. Jarke (FRG)
K. G. Jeffery (United Kingdom)
P. Kanellakis (USA)
M. L. Kersten (The Netherlands)
H. Korth (USA)
R. Kowalski (United Kingdom)
M. Lacroix (Belgium)
M. Leonard (Switzerland)
U. Lipeck (FRG)
P. Lockeman (FRG)
D. Maier (USA)
A. Meier (Switzerland)
A. Mendelzon (Canada)
M. Missikoff (Italy)
R. Morrison (United Kingdom)
G. T. Nguyen (France)
I. Paredaens (Belgium)
F. Rabitti (Italy)
R. Ramakrishnan (USA)
P. Richard (France)
C. Rolland (France)
D. Sacca (Italy)
Y. Sagiv (Israel)
H. Schek (FRG)
J. W. Schmidt (FRG)
O. Shmueli (Israel)
Y. Tanaka (Japan)
J. Ullman (USA)
P. Valduriez (France)
L. Vieille (FRG)
M. Yoshikawa (Japan)
A. V. Zamulin (USSR)
R. Zicari (Italy)

## External Referees

C. V.D. Berg
C. Berrut
J. P. Bertrandias
M. F. Bruandet
S. Brass
A. L. Brown
H. J. Buikhardt
G. Cayres Magalhaes
C. Collet
R. C. H. Connor
M. Consens
B. Defude
A. Dearle
C. Delcourt
C. Delobel
U. Deppisch
D. DeWitt
N. Erol
T. Estier
G. Falquet
M. C. Fauvet
M. Freeston
J. P. Giraudin
O. Gruber
R. H. Guting
M. Hofmann
G. Hulin
Y. Ioannidis
M. Jeusfeld
P. Kipelainen
T. Kabas
M. Koubarakis
P. Lavency
C. Lécluse
E. Levy
C. de Maindreville
R. Manthey
H. Martin
F. Mc Cabe
C. Moss
H. B. Pane
M. Rafanelli
D. Rieu
A. Rifaut
T. Rose
J. Rosenberg
G. Saake
M. Sadler
F. Sadri
M. Sakkinen
L. Sbatella
M. H. Scholl
A. Siebes
E. Simon
S. Sippu
G. Speegle
S. Sripada
D. Srivastava
S. Sudarshan
M. Terranova
E. Ukkonen
M. Vanhoedenaghe
P. Velardi
F. Velez
V. Vianu
G. von Bueltzingsloewen
G. Weikum

# Contents

## Session 6: Complex Objects



## Session 7: Database Programming Languages



## Session 8: Panel



## Session 9: Object-Oriented Systems



## Session 10: Time, Object-Oriented and Active Systems

## Session 11: Rules



## Session 12: Panel

# Deductive Databases–Theory Meets Practice

*Carlo Zaniolo*

MCC
3500 West Balcones Center Drive
Austin, Texas 78759
USA

**Abstract**

Deductive Databases are coming of age with the emergence of efficient and easy to use systems that support queries, reasoning, and application development on databases through declarative logic-based languages. Building on solid theoretical foundations, the field has benefited in the recent years form dramatic advances in the enabling technology. This progress is demonstrated by the completion of prototype systems offering such levels of generality, performance and robustness that they support well complex application development. Valuable know-how has emerged from the experience of building and using these systems: we have learned about algorithms and architectures for building powerful deductive database systems, and we begin to understand the programming environments and paradigms they are conducive to. Thus, several application areas have been identified where these systems are particularly effective, including areas well beyond the domain of traditional database applications. Finally, the design and deployment of deductive databases has provided new stimulus and a focus to further research into several fundamental issues. As a result, the theory of the field has made significant progress on topics such as semantic extensions to Horn logic and algorithms for compilation and optimization of declarative programs. Thus, a beneficial interaction between theory and practice remains one of the strengths of Deductive Databases as the field is entering the '90s and the age of technological maturity.

## 1 Background

Deductive Databases are coming of age with the emergence of efficient and easy to use systems that support *queries, reasoning*, and *application development* on databases through *declarative logic-based* languages.

Interest in the area of Deductive Databases began in the '70s, with most of the early work focusing on establishing the theoretical foundations for the field. An excellent review of this work and the beneficial impact that it had on various disciplines of computing, and the database area in particular, is given in [GMN]. Throughout the '70s and the first part of the '80s, concrete

system implementations of these ideas were limited to few ground breaking experiments [Kell]. This situation contrasted quite dramatically with the significant system-oriented developments that were taking place at the same time in two fields very close to deductive databases. The first field was relational databases, where systems featuring logic-based query languages of good performance, but limited expressive power, were becoming very successful in the commercial world. The second field is Logic Programming, where successive generations of Prolog systems were demonstrating performance and effectiveness in a number of symbolic applications, ranging from compiler writing to expert systems.

A renewed interest in deductive database systems came about as a result of the flare-up of attention and publicity generated by the idea of Fifth Generation Computing. It was realized that the rule based reasoning of logic, combined with the capability of database systems of managing and efficiently storing and retrieving large amounts of information could provide the basis on which to build the next-generation of knowledge base systems. As a result, several projects were started that focused on extending Prolog systems with persistent secondary-based storage management facilities [RaSh] or on coupling Prolog with relational databases [JaCV, KuYo, Li, CeGW]. Several commercial systems are now available that support the coupling of SQL databases with Prolog or expert system shells. In particular, is the system described in [Boc, LeVi] provides close integration between Prolog and Database facilities, and smart algorithms for supporting recursive queries against the database.

Yet several other researchers were critical of the idea of using Prolog as a front-end to relational databases. In particular, it was noted that the sequential left-to right execution model of Prolog was a throw-back to navigational query languages used before relational systems. In relational systems, the user is primarily responsible for correct queries, and the system takes care of finding efficient sequencing of joins (query conjuncts), thus optimizing navigation through the database—a special module called the *query optimizer* sees to that [Seta]. In Prolog, instead, the programmer must carefully select the order of rules and of goals in the rules, since the correctness, efficiency and termination of the program depend on it. A second problem follows from the fact that efficient Prolog implementations are based on a abstract machine (WAM) and features (pointers) that rely on the assumption that data resides in main memory rather than secondary store [War]. Thus a number of research projects opted for an approach that builds more on extensions of relational database technology than on adaptations of Prolog technology. While several of these projects limited their interests to extending query languages with specific constructs such as rules and recursion, projects such as NAIL! [Meta] and $\mathcal{LDL}$ [Ceta1, NaTs] feature declarative languages of expressive power comparable to Prolog. This paper recounts and summarizes the author's experience in designing, developing and deploying the $\mathcal{LDL}$ system.

# 2 Overview

The motivation for designing and building the $\mathcal{LDL}$ system was twofold:

- To provide support for advanced database applications, with a focus on expert systems and knowledge based applications.
- To provide better support for traditional database applications by integrating the application development and database queries into one language—thus solving the impedance mismatch problem.

A serious problem with current database applications is due to the limited power of languages such as SQL, whereby the programmer has to write most of the application in a procedural language with embedded calls to the query language. Since the computing paradigm of a procedural language, such as COBOL, is so different from the set-oriented declarative computation model of a relational language, an *impedance mismatch* occurs that hinders application development and can also cause slower execution [CoMa]. Realization of this problem has motivated a whole line of database research into new languages, commonly called *database languages* [BaBu]. The typical approach taken by previous researchers in database languages consisted in building into procedural languages constructs for accessing and manipulating databases [Sch77, RoSh]. Persistent languages, where the database is merely seen as an extension of the programming language, represent an extreme of this emphasis on programming languages. In a sharp departure from these approaches, $\mathcal{LDL}$ focuses on the query language, and extends it into a language powerful enough to support the development of applications of arbitrary complexity. Rather than extending current database query languages such SQL, however, $\mathcal{LDL}$ builds on the formal framework of Horn clause logic—a choice that had less to do with the well-known shortcomings of SQL, than with the influence of Prolog (a language based on Horn clause logic). In fact, we were impressed with the fact that this rule-based language was effective for writing symbolic applications and expert applications as well as being a powerful and flexible database query language [Zan1].

A closer examination on why Horn clauses represent such a desirable rule-based query language reveals the following reasons:

- Horn Clauses are akin to domain relational calculus [Ull], which offer two important advantages with respect to tuple calculus on which languages such as SQL are based—but the two calculi are known to be equivalent in terms of expressive power. One advantage is that domain calculus supports the expression of joins without explicit equality statements; the other is that lends itself to the visualization of queries —both benefits vividly demonstrated by QBE [Ull].

- Horn clauses support *recursion* and *complex terms* (through function symbols) thus eliminating two important limitations of relational query languages and systems.

- Horn clauses have a declarative semantics based on the equivalent notions of minimal model and least fixpoint [Llo, vEKo].

- Horn clauses can also be used effectively as a navigational query language.

As the last two points suggest, Horn clauses can be used effectively as either a declarative query language or navigational one [Zan1]. In the declarative interpretation of Horn Clauses, the order of goals in a rule is unimportant (much in the same way in which the order of conjuncts in a relational query is immaterial). The navigational interpretation of Horn clauses follows from the operational semantics of Prolog. Under this interpretation, goals are executed respectively in a left-to-right order, and the programmer is basically entrusted with the task of using this information to write terminating and efficient programs. For instance, when the goals denote database relations, the order defines a navigation through the database records; the programmer

must carefully select the best navigation, e.g., one that takes advantage of access structures and limits the size of intermediate results.

A most critical decision in designing $\mathcal{LDL}$ was to follow the path of relational systems and build on the declarative semantics, rather than on the operational interpretation of Horn clauses. This approach was considered to be superior in terms of data independence and ease of use. Indeed this approach enables the user to concentrate on the meaning of programs, while the system is now entrusted with ordering goals and rules for efficient and safe executions. A further step toward declarative semantics was taken by freeing the user from the concern of whether forward chaining or backward chaining should be used in executing a set of rules. Current expert system shells frequently support only one of these two strategies; when they provide for both, they leave to the programmer the selection of the proper strategy for the problem at hand and its encoding as part of the program. In $\mathcal{LDL}$, the actual implementation is largely based on a forward chaining strategy which is more suitable for database applications [Ceta2]. But the compiler has also the capability of using rule rewrite methods, such as the magic set method or the counting method [BMSU, SaZ1, SaZ2], to mimic backward chaining through a bottom-up computation. Thus the $\mathcal{LDL}$ user is also provided automatically by the system with the functionality and performance benefits of backward chaining. This substantial progress toward declarative programming represents one of the most significant contributions to the technology of rule-based systems brought about by the research on deductive database systems in the recent years.

Another major area of progress for deductive databases is that of semantics. Indeed many other constructs beyond Horn clauses are needed in a language such as $\mathcal{LDL}$ to support application development. In particular, $\mathcal{LDL}$ includes constructs supporting the following notions:

- Negation [ApBW, Naq, Prz1],
- Sets, including grouping and nested relations [BNST, ShTZ],
- Updates [NaKr, KNZ]
- Don't-care non-determinism [KrN1].

Most of these constructs (excluding set terms) are also in Prolog—they were added because they were needed for writing actual applications. But, in Prolog, their semantics is largely based on Prolog's operational model. Therefore, a major challenge of the $\mathcal{LDL}$ research was to define a formal declarative semantics for these constructs, in a way that naturally extends the declarative semantics of Horn clauses. The problem of extending the power of declarative logic is in fact the second main area of recent advances promoted by research in deductive databases. Of particular interest is the fact that many open problems in knowledge representation and non-monotonic reasoning have been given a clearer definition, and in some cases brought close a solution by these new advances [MaSu, Prz2]

The combined challenge of designing a powerful and expressive language, with declarative semantics, and efficient techniques for compilation and optimization describes the whole first phase of $\mathcal{LDL}$ research. This began in mid 1984, and culminated in the implementation of the first prototype at the end of 1987. This prototype compile $\mathcal{LDL}$ into a relational algebra based language FAD for a parallel database machine [Bor]. Rule rewriting methods, such as magic set and counting,

were used to map recursive programs into equivalent ones that can be supported efficiently and safely by fixpoint iterations [BMSU, SaZ1, SaZ2]. A description of this system is given in [Ceta1].

The implementation of the first $\mathcal{LDL}$ prototype confirmed the viability of the new technology, but did little to transfer this technology from the laboratory to actual users, since FAD is only available on an expensive parallel machine. Seeking a better vehicle for technology transfer, a new prototype system was designed with the following characteristics:

- Portability,
- Efficiency,
- Open System Architecture.

This effort produced a portable and efficient $\mathcal{LDL}$ system under UNIX, called *SALAD*. [1] This implementation assumes a single-tuple, get-next interface between the compiled $\mathcal{LDL}$ program and the underlying fact manager (record manager). This provides for more flexible execution modes than those provided by relational algebra [Ceta1, Zan1]. The new design yields better performance, since the optimizer can now take advantage of different execution modes, and the compiler can cut out redundant work in situations where intelligent backtracking or existential optimization can be used [CGK2, RaBK]. *SALAD* includes a fact manager for a database residing in virtual memory that supports efficient access to the complex and variable record structures provided in $\mathcal{LDL}$. By using *C* as an intermediate target language and an open system architecture, *SALAD* ensures portability, support for modules, and for external procedures written in procedural languages—including controlled access by these routines to internal *SALAD* objects.

The *SALAD* prototype was completed in November 1988, and has undergone improvements and extensions during 1989. By the end of 1989, the system includes a fully functional optimizer, a powerful symbolic debugger with answer justification capability and an X-windows interface. The availability of *SALAD* led to the writing of significant applications and to the emergence of an $\mathcal{LDL}$ programming style. It was found that, in addition to supporting well database applications, $\mathcal{LDL}$ is effective as a rule-based system for rapid prototyping of applications in the *C* environment. Also in 1989, a complete description of the $\mathcal{LDL}$ language with sample applications appeared at bookstores [NaTs], and the first executable copies of *SALAD* were given to universities for experimentation.

I interpret these events as signs of a maturing technology. But, in the end, only the level of satisfaction experienced by users with *SALAD* or similar systems can confirm or disprove my claim that deductive databases are coming of age. To promote this goal, however, this paper will summarize the highlights of my experience with the $\mathcal{LDL}$ system, hoping that the readers will be enticed to experiment with it and then become enthusiastic users of the system. Therefore, the paper focuses on the functionality and usability aspects of the system. The reader interested in the architecture and enabling technology is referred to a recent overview [Ceta2]

# 3 Declarative Programming and Debugging

A declarative semantics for rules offer several advantages over the operational one, including the following ones:

[1] SALAD—System for Advanced Logical Applications on Data.

- Naturalness,
- Expressive Power,
- Reusability and Data Independence.

Frequently, the most natural definition of a programming object is inductive. For instance, the following $\mathcal{LDL}$ program defines all the integers between zero and $K$, using Peano's inductive definition (zero is an integer and if J is an integer, so is J+1).

```
int(K,0).
int(K,J) ← int(K,I), I< K, J = I+1.
```

The $\mathcal{LDL}$ compiler has no problem turning this definition into an efficient fixpoint iteration. This pair of rules, or any one obtained by scrambling the order of their goals, cannot be supported by any interpreter or compiler implementing the backward chaining strategy. For instance, in Prolog the user must be go through some interesting contortions to recast this program to fit the operational model.

As the next example, consider the situation where there is a binary tree of atoms. For instance a tree with leaves a and b will be represented by the complex term `tree(a, b)`. Associated with the leaf nodes of a tree there is a weight represented by facts such as

```
node(a, 1).
node(aa, 2).
node(ab, 3).
```

The weight of a tree is inductively defined as the sum of the weights of its two subtrees. We want now to define all the trees with weight less than a certain M. Immediately from these definitions we derive the following rules.

```
w(N, W, M) ← node(N, W), W < M.
w(tree(T1,T2), W, M) ← w(T1,W1), w(T2, W2), W = W1+W2, W < M.
```

This simple definition is not implementable with backward chaining and, unlike the previous example, we do not know of any set of rules that will support this predicate well in Prolog (assuming that the weights of the nodes are not known before hand). While forward chaining is the preferred strategy for these two examples, there are many situations where backward chaining is instead the only reasonable strategy. For instance, say that a tree is at hand and its weight must be computed, as per the the following query goal (where 10000 denotes a value high enough not to be a factor in the computation).

? w(tree(aa,tree(a,ab)), X, 10000).

In this situation, the $\mathcal{LDL}$ compiler simply mimics backward chaining by the use of a rewriting method—the efficient counting method in this particular case [SaZ3]. What is most important here is that the program has not changed. The same program works for different situations and the

compiler/optimizer takes care of matching the actual execution method to the problem at hand. The final result is a level of reusability of programs that is well beyond that of Prolog programs. The elimination of cuts, replaced by the `choice` and `if-then-else` constructs, is also very beneficial in terms of reusability [Zan1]. The concept of reusability for database programs is an extension of the notion of data independence, defined as the ability of queries and applications to survive changes in the database physical organization. In relational databases the key instrument in delivering data independence is the optimizer. In $\mathcal{LDL}$ the optimizer ensures data independence, reusability of code and economy of programming, since the user can call the same module or predicate with different set of bindings.

The previous example was inspired by an Alkane Molecules generation problem [Tsur] that was first proposed to illustrate the power of functional languages. The same problem was quite easily formulated in $\mathcal{LDL}$ due to the ability of expressing inductive definitions and to the ease of checking equivalent structures while avoiding cyclic loops, discussed next. Semantically the structures previously discussed are unordered trees. Thus, a given tree is equivalent to those obtained by recursively exchanging the left subtree with the right one. Equivalence can be expressed by following set of rules:

```
eq(T, T).
eq(tree(T1,T2), tree(T2,T3)) ← eq(T3, T1).
```

Thus two trees are equivalent, if they are equal or if their subtrees have been exchanged and possibly replaced with equivalent ones. The problem is that the composition of several exchanges can return the original structure, and the SLD-resolution will cycle. Thus in Prolog the programmer has to carry around a bag of previous solutions and check for cycles at the cost of inefficiency of programming and execution. In $\mathcal{LDL}$ instead, the system can deal with cycles automatically and efficiently. This feature is particularly important in situations involving negations, since it is the key to a complete realization of stratified negation which avoids the floundering problem of negation by failure [Prz1, Llo].

One of the most interesting aspects of programming with a declarative language is debugging. Any trace-based debugger would be a little use in $\mathcal{LDL}$ since the optimizer rearranges the rules to a point that they do not resemble the original program. On the other hand, the logical nature of the system makes it possible to explain and justify the answers, and thus support a truly logical debugger. The current $\mathcal{LDL}$ system provides logical debugging and answer justification capabilities as sophisticated as those of any expert shell or rule-based system available today.

The conceptual basis for the logical debugger consists in combined why and whynot explanation capabilities, whereby the system carries out a conversation with the user explaining why a certain answer was returned, and why another was not returned.

Thus to a user asking

```
why eq(tree(a, tree(b,c)), tree(a, tree(c,b)).
```

the system will return the instantiated rule that produced it:

```
eq(tree(a, tree(b,c)), tree(a, tree(c,b))) ← eq(tree(c,b),tree(b,c)).
```