Bottom-Up Query Evaluation in Extended Deductive Databases

Dem Fachbereich Mathematik
der Universität Hannover

zur Erlangung der Lehrbefugnis (venia legendi)
für das Fachgebiet Informatik

vorgelegte Habilitationsschrift

von

Stefan Brass

1996

My current address is:

Universität Hannover, Institut für Informatik,
Lange Laube 22, D-30159 Hannover, Germany,
`sb@informatik.uni-hannover.de`

This document was last updated on: June 12, 1996.

Contents

1 Introduction **1**

2 Horn-Clause Datalog **27**
2.1 Logic and Databases . . . 27
2.2 Syntax and Semantics of Datalog . . . 38
2.3 Bottom-Up Query Evaluation . . . 52

3 Goal-Directed Bottom-Up Evaluation **67**
3.1 The "Magic Set" Transformation . . . 71
3.2 An Improved Magic Set Technique . . . 83

4 Negation as Failure **101**
4.1 Datalog with Negations . . . 106
4.2 A Framework for Studying Semantics . . . 122
4.3 Bottom-Up Query Evaluation . . . 149

5 Reasoning with Disjunctions **167**
5.1 Extended Bottom-Up Query Evaluation . . . 169
5.2 Application: Computation of Stable Models . . . 180

6 Conclusions **187**

Bibliography **189**

Index **203**

Acknowledgements

First of all, I would like to thank my doctor father Udo Lipeck for giving me the opportunity to perform the research presented here and for numerous helpful comments. I believe that his influence can be seen in every chapter. For instance, there are notions from term rewriting used in Section 4.2, and he always recommended using meta-interpreters (which I did in Sections 3.2 and 5.1). Also Section 5.1 is based on joint research, and Section 5.2 is influenced by an earlier joint paper.

It is a pleasure for me to thank Jürgen Dix for two years of joint research, which has led to the results presented in Chapter 4 (and more). I really enjoyed working together with Jürgen, and our cooperation has been very fruitful.

I would also like to thank the students in my courses on deductive databases and logic programming. I learnt a lot by giving these lectures. Dirk Hillbrecht and Michael Salzenberg have implemented query evaluation based on the residual program and hyperresolution.

My brother Peter found a number of typing errors and made other valuable suggestions. Last but not least, many thanks to my parents for all kinds of support and finally getting me to this point (which was not always easy).

Chapter 1

Introduction

The goal of this work is to develop query evaluation algorithms for deductive databases extended by nonmonotonic negation and disjunctions. This topic lies in between the three fields of automated theorem proving, nonmonotonic reasoning, and databases. There are three specific questions treated here:

- First, even in the standard case of Horn clauses without negation, bottom-up query evaluation has not reached the efficiency of top-down query evaluation in practice [SSW94]. What are the reasons for this, and can the situation be improved?
- Second, there is a large number of proposed semantics for nonmonotonic negation. How different do query evaluation algorithms for them have to be, and are there any connections between semantical properties and possible ways to compute them?
- Third, reasoning with disjunctive rules is currently far less efficient than reasoning with Horn clauses. In fact, it seems that the communities studying the two topics are nearly disjoint. So, how far is it possible to use standard techniques known for Horn clauses in more general query evaluation algorithms?

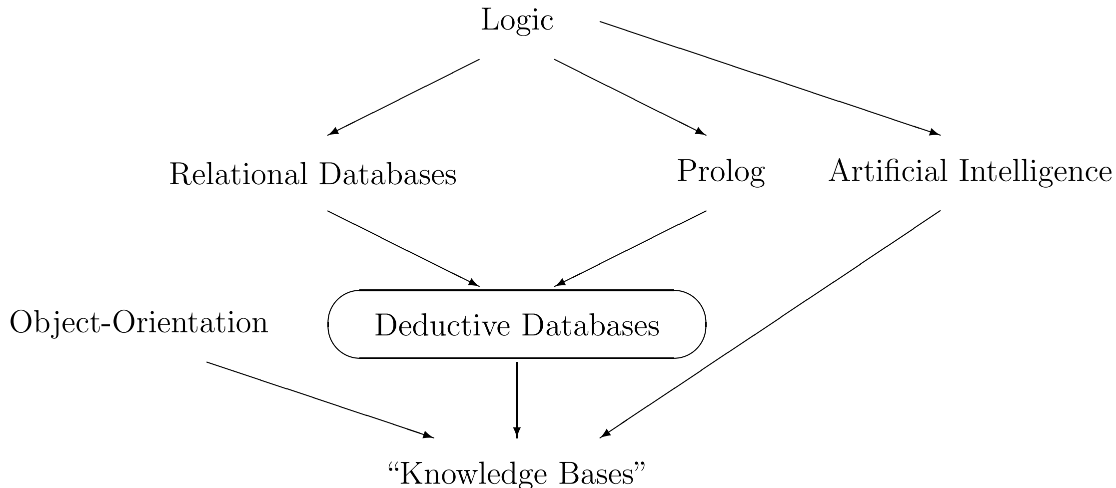
Some Bits of History

The field of mathematical logic has strongly influenced many branches of computer science, for instance the early results on formal languages and computability were developed by logicians (after all, there was no computer science by that time).

One of the main goals of mathematical logic is to represent knowlege in some formal way, suitable for algorithmic treatment. In computer science, databases were developed to store information. At the beginning, databases did not have much theory and were only a collection of subprograms to access files. It was a major step forward when CODD proposed 1970 the relational datamodel [Cod70], where a database state is nothing else than a first order interpretation (without function symbols) known in logic for a long time. This data model was first only theoretically defined, and was critized for being un-implementable. However, a decade later first commercial relational database systems appeared, and two decades later they are the state of the art.

In the programming language community, logic lead to the development of Prolog ("PROgramming in LOGic") by COLMERAUER, ROUSSEL and others in the early 1970s. KOWALSKI theoretically explained the possibility to use predicate logic as a programming language [Kow74, vEK76].

Now the idea of deductive databases is to integrate the possibilities of relational databases and Prolog:



So a deductive database consists of

- a relational database, which defines a number of relations (or predicates) "extensionally", by enumerating the tuples contained in the relation, and
- a logic program, which defines a number of relations/predicates "intensionally", by giving a set of defining rules (formulas of a restricted kind).

Of course, there are also alternative possibilities to describe deductive databases:

- One can say that a deductive database is simply a relational database with a new query language (Datalog instead of SQL), and with the view mechanism extended to allow recursive definitions.
- Or one can say that a deductive database is nothing else than a logic program with a large number of facts (corresponding to the tuples in the database), possibly treated in some special way in the implementation.
- Finally, a deductive database can be seen as an automated theorem prover, which allows only special kinds of formulas, but very many of them.

Of course it is difficult to single out one point in history, where the field of deductive databases "has started". Maybe we should mention the following events:

- A milestone in the development of automated theorem proving was the invention of the resolution method by ROBINSON 1963 [Rob92]. Subsequently, so called "question-answering systems" were developed (by GREEN [Gre69] and others [Min88b]), which tried to extract useful information (bindings of variables) from proofs. They can be seen as predecessors of deductive databases.
- VAN EMDEN and KOWALSKI introduced 1976 the minimal model of a set of definite Horn clauses [vEK76]. In logic programming it was superseded by CLARK's

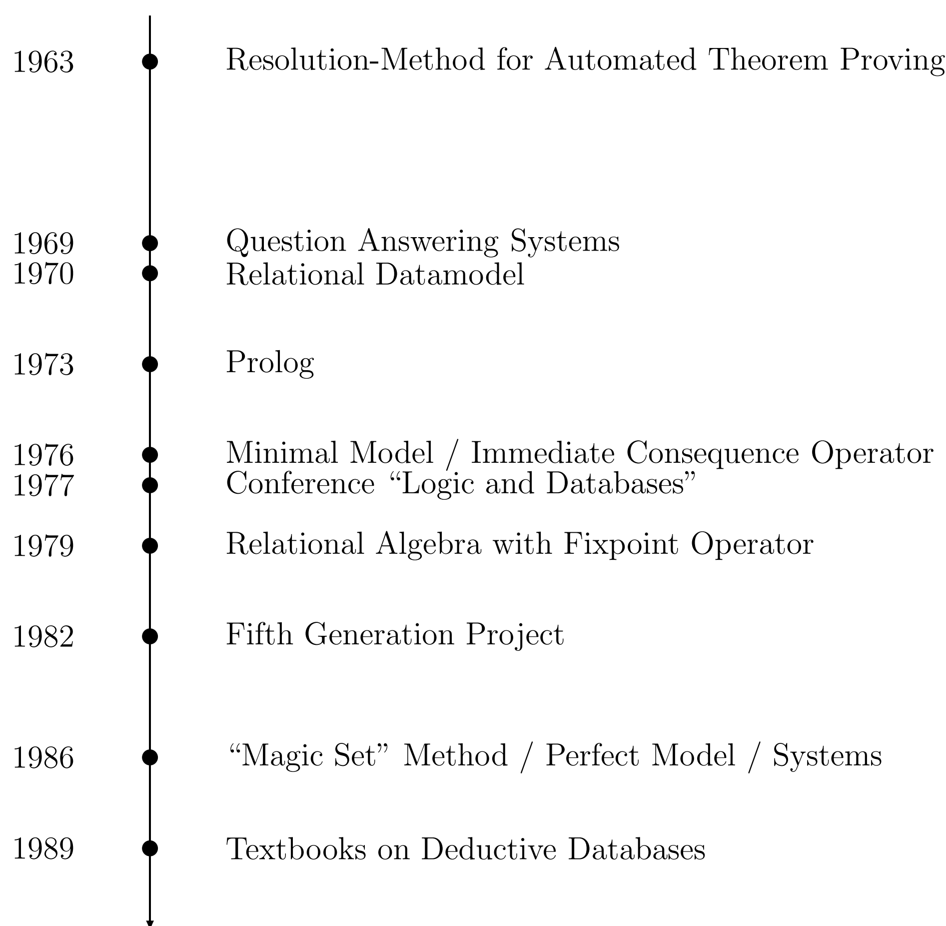


Figure 1.1: How the Field of Deductive Databases has Started

completion, but it revived later as the standard semantics of deductive databases and was generalized in various ways to handle negation. In [vEK76] also the immediate consequence operator and the fixpoint semantics were introduced, which are the foundation of bottom-up evaluation. As noted in [vEK76], the immediate consequence operator is nothing else than a special case of hyperresolution, which was introduced 1965 by Robinson.

- In 1977 Gallaire, Minker, and Nicolas organized a workshop on “Logic and Databases” in Toulouse and published subsequently a book with contributions from that workshop [GM78]. Among other important contributions, also two formalizations of nonmonotonic negation were presented at this conference: The completed database by Clark [Cla78] and the closed world assumption by Reiter [Rei78].
- In 1979, a paper by Aho and Ullman on relational algebra with a fixpoint operator appeared [AU79]. They also proved that the transitive closure cannot

be expressed in standard relational algebra. Although this paper itself did not examine the relation to logical rules, it was certainly one of the origins of bottom-up query evaluation.

It is difficult to say when naive bottom-up evaluation of Datalog was exactly introduced. As we already mentioned, it dates back to 1965, and there have been a lot of papers using some form of bottom-up evaluation. Naive bottom-up query evaluation as we know it today (based on relational algebra with fixpoint evaluation) seems to be first described in [CGL86].

- In 1982 the Fifth Generation Project started in Japan [Fur92]. It gave an important impetus to the development of logic programming, not only in Japan, but throughout the world.
- The Prolog-dialect for deductive databases is nowadays usually called "Datalog". It is not easy to find out who invented that name. In [CGT90] it is said:

  > The term "Datalog" was invented by a group of researchers from Oregon Graduate Center, MCC, and Stanford University.

  In [AHV95], the following is said:

  > It is difficult to attribute datalog to particular researchers because it is a restriction or extension of many previously proposed languages; some of the early history is discussed in [MW88]. The name *datalog* was coined (to our knowledge) by David Maier.

  I have found no papers before 1986 which use the name "Datalog", and 1986 it was used in [BMSU86], which refer to an unpublished memorandum of the Oregon Graduate Center 1985 (probably a predecessor of [MW88]). Also [MUVG86] contains the name "Datalog".
- In or before 1986 the work on three major implementations of deductive database technology was started: The LDL system at MCC [TZ86, Zan88, NT89], the NAIL! system at Stanford University [MUVG86], and the Smart Data System (SDS) at a commercial offspring of the Technical University of Munich [KSSD94].
- In 1986, Bancilhon, Maier, Sagiv, and Ullman developed the "Magic Set"-method [BMSU86], and Rohmer, Lescoeur, and Kerisit developed the closely related "Alexander"-method [RLK86]. These methods allow to combine the advantages of top-down and bottom-up query evaluation (see below). At that time many query evaluation algorithms for recursive rules had been proposed [BR86], and this was probably the true starting point where the field got its own methods and results. Today, deductive database systems usually use the Magic Set/Alexander technique, or variants of SLD-resolution with memoing, of which [TS86, Vie87a, Die87] were early developments. A similar method, which also integrates top-down and bottom-up evaluation, is the query-subquery approach [Vie87b, Nej87].
- In 1986, Minker organized another conference on deductive databases [Min88a], which was important for the development of the notion of stratified databases and their perfect model [VG86, ABW88, Prz88a, Naq89]. The treatment of

negation according to the perfect model, which became standard in deductive databases, is again a difference to logic programming, where CLARK's completion and similar approaches were dominant.

- From 1988 to 1990 a first generation of textbooks on deductive databases appeared. They were written by ULLMAN [Ull88, Ull89b], NAQVI and TSUR [NT89] and CERI, GOTTLOB, and TANCA [CGT90]. Furthermore, in the 5th edition of his best-selling textbook on databases, DATE added a chapter on deductive databases [Dat90].

With the event of a number of dedicated textbooks, it can be really said that the field is well established. Of course, many important things have been achieved since 1989. However it is still a little difficult to put them into historical perspective, since the distance is missing.

Why Deductive Databases?

A typical database application (see Figure 1.2) consists of three parts, which are usually coded in three different languages:

- At the top is the user interface, which manages the dialog between the user and the program. It defines menues, buttons, dialog boxes, and so on. Usually, it is either written in a special language like Tcl/Tk or HTML, or is constructed with a special editor ("resource construction set", "interface builder").
- Below that, there is a layer of code which performs data format changes, combines and aggregates data, and ensures that the integrity of the database is not violated. This code is often written in an imperative language like "C".
- Finally, at the bottom are the database accesses, written in SQL. The database has built-in algorithms for searching and sorting, it ensures the persistence of the data and manages concurrent accesses by different users.

Besides the problems for the user to know three different languages and to interact with three different compilers, it is well-known that between each two layers there is an "impedance mismatch problem". For instance, the database returns a set of answers to a query, but in the programming language it is necessary to write a loop which fetches every single tuple.

Deductive databases solve this problem because they are both, a programming language and a database, and they are usually so tightly coupled that it is not clear where one thing stops and the other starts.

This wish to integrate a database with programming facilities also led to the development of object-oriented databases. However, in deductive databases the set-oriented paradigm was extended to the programming part, while in object-oriented databases the tuple-oriented approach of the programming-language has "won".

Even commercial relational databases nowadays allow to store procedures within the database. This is a logical development, because one important goal of databases is the integration of different applications within some company. Before there were databases, every application program more or less had its own data files. Then databases allowed to share the data and avoid redundancy (and thus errors), but still

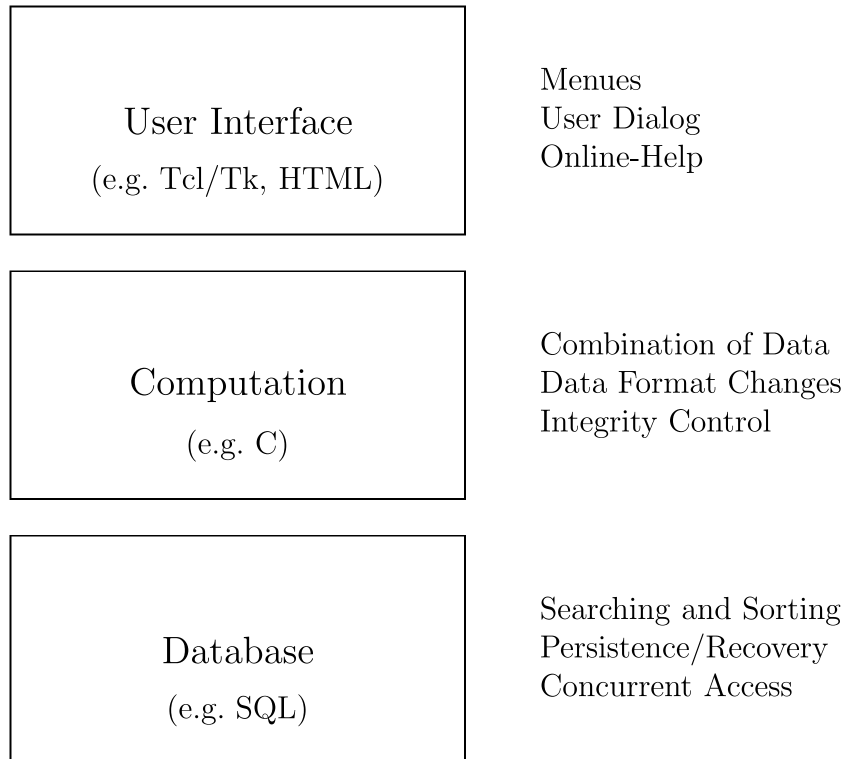


Figure 1.2: A Typical Database Application

the application programs were separate. There might have been some library procedures, just as some files were shared between some applications before databases were developed. But it was necessary to explicitly relink the application programs when a library procedure was changed. There was no controlled way of sharing code. This is what deductive databases as well as object-oriented databases and procedural extensions of relational databases aim at.

An additional advantage is that all the functionality of the database, such as persistency, data dictionary, access rights, etc., apply to the procedures as well, because they are stored within the database. Furthermore, the query optimization mechanisms of the database can now make some use of their knowledge of the procedural code. Before, no “global” query optimization was possible, because the database could not know at all how the sequence of queries executed by one program would look like.

Since deductive databases are a symbiosis of Prolog and relational databases, it is natural to ask what we gain with respect to each of the partners. If we compare a deductive database with a relational one, we have also the following advantages besides the above mentioned better integration with procedures:

- The queries which can formulated in SQL are restricted. For instance the transitive closure cannot be computed by a single query in standard SQL. Of course,

since this problem is practically relevant and often cited, some commercial versions of SQL now have a special clause for computing the transitive closure. But this is only a patch on the design of the SQL language: It complicates the language and solves one specific symptom, but not the deeper cause, namely that recursion is not available in SQL. This also means that recursive data structures such as trees or already lists cannot be manipulated adequately in SQL.

It might be argued that in practice recursion is always bounded, and for instance paths of length 3 or 4 can theoretically be computed in SQL. However, such queries are ugly to write in SQL and their size grows so fast (quadratically) that one soon reaches the limitations of the system. In combination with an application program in "C" or some other computationally complete language, every kind of query can be evaluated, but then we again have the above mentioned "impedance mismatch" problems.

- The notion of a view is not fully integrated in the relational model. Quite a number of systems restrict the queries which can be posed to views. One of the reasons for this is that views are often implemented by rewriting the query (so that it refers only to base relations), but SQL is not orthogonal, so the result is not necessarily a valid SQL query. However, in deductive databases, views are "first class citizens": The notion of a derived predicate is so essential that any restriction in their definition and usage would be totally unacceptable. Furthermore, corresponding to their greater importance in deductive databases, the implemention of views is usually more efficient there.
- The relational model is unsuitable for highly regular data. For instance a bus schedule can be better represented by rules than by facts. In a relational database such a bus schedule would probably be stored "extensionally" by enumerating the departure times. Maybe somebody even writes a "C" program to generate all these facts. But again the database has no knowledge of this program, while it could make good usage of the rules: For instance, the rules occupy much less memory than the facts, and thus can be more quickly loaded from external storage.

In comparison with Prolog, we should mention the following differences:

- Deductive databases are "more logical" and have less control, so they are closer to a theoretically ideal logic programming language. For instance, the order of rules or body literals does not matter in a deductive database, while it is essential in Prolog. The deeper reason for this is that deductive databases and Prolog programs are used very differently: A Prolog program usually is executed by calling one main predicate, so the programmer knows in advance which arguments of a predicate are bound or free, and therefore he/she can order the rules and body literals in an optimal way. In deductive databases this is not possible, because the user might query any predicate with any binding pattern for the arguments (more or less). It is therefore the responsibility of the system to determine a good execution order. This matches the tradition of the database community that a database should have a powerful query optimizer. Of course, such optimizers work well only for sets of relatively simple rules — they cannot

optimize all Prolog programs.

- Due to the bottom-up evaluation, the termination behaviour is much better than that of Prolog. In general, we would expect that termination can be guaranteed for query evaluation, while this certainly is not possible for arbitrary programs. Modern deductive database systems allow more or less every Prolog program, so termination cannot be guaranteed in general. But termination is a major issue in databases, and large subsets of queries/programs have been defined for which termination can be guaranteed.
- Prolog has insufficient support for external memory. Naturally, Prolog was not intended to work with large sets of facts, and many Prolog implementations will become at least very inefficient if the main memory is too small. An important property of external memory is its block-orientation: In order to get a single tuple of a few bytes we must read a complete block of several kilobytes. Therefore, set-oriented computations as done in deductive databases can make much better use of external storage than the standard tuple-oriented evaluation of Prolog. Of course, also other essential database features, like support for multiple users and recovery after a system crash, are missing in Prolog.

Let us conclude this section by looking at a number of typical applications of deductive databases [KG90, Tsu90, Tsu91a, Ram95]:

- **Expert systems:** Usually the expert knowledge is formalized by means of rules (not necessarily logic programming rules). Furthermore, there are often large sets of facts needed (even an expert has to look into a book from time to time). Thus, a deductive database seems to be a good tool for developing an expert system. Of course, an expert system shell usually has better support for creating a user interface and defining a structured user dialog, and better explanation facilities. But all this would be helpful in a deductive database as well.

  In [HR95], an expert system for querying a flights database is described. Since a user has several conflicting criteria for the best flight (cheap, not too early in the morning, not too much wait time on transit, not an obscure airline, etc.), this is a quite complex task. In [KSSD94] an expert system for the public transportation system of the Munich area is mentioned.
- **Decision Support Systems:** The task of these systems is to aggregate information form large data sets, or to find interesting cases in them. Often, the temporal development of the data should be displayed. Also, the system should be flexible and allow ad-hoc queries. Furthermore, it should have the ability to reason about future plans [RH94]. All this can be well supported by deductive databases.

  In [KSSD94] the following is said on the marketing strategy of the deductive database system SDS:

  > One main selling point of this technology is its strategic decision making capability. Database technology, enhanced by deductive rule-based capabilities, can assist enormously in condensing information to make

> good decisions (a major key in achieving a competitive advantage). In many corporate decisions, relevant information is spread over heterogeneous databases, and such an environment has to be addressed.

In [RRS95] a system for stock market analysis is described, which is based on the deductive database system CORAL [RSSS94]. The main advantage of using a deductive database system is the easy extensibility. It is also noted that

> The recursive query capabilities of CORAL are necessary for expressing many natural concepts (e.g. "bull market", "consecutive peaks") in this domain.

Finally, we would like to mention the following applications, which lie on the border to expert systems [RH94]:

> Medical analysis and monitoring can generate large amounts of data, and an error can have disastrous consequences. A tool to carefully monitor a patient's condition or to retrive relevant cases during diagnosis reduces the risk of error in such circumstances. Deductive database technology allows the analysis of these data to be performed more efficiently and with lower chance of error than by ad hoc methods. Such an intelligent tool allows the human experts to concentrate on the main problems, rather than being distracted by details. A similar example may be found in mineral exploration; a large amount of data may be generated, which can then be analyzed for clues suggesting the presence of the desired mineral.

- **Hierarchical Design:** Computer aided design of hierarchically structured objects is a good application area for deductive databases, because of their special support for hierarchies (through the computation of transitive closures). Of course, currently the performance of deductive databases is a problem, and therefore object-oriented databases are preferred. However, it is a nontrivial task to write a "C++"-program which checks whether a given object directly or indirectly needs some specific part, while this is possible in three short lines of Datalog.

  The "bill of materials"-problem has been used as an example in [CGT90, KG90], in [BK92] economists have discussed the problem and the usefulness of Datalog.

- **Complex Integrity Constraints:** In many design tasks complex integrity constraints have to be enforced. Since these integrity constraints are typically defined in logic, it is natural to use a deductive database to check them. Of course, there are again special additions to standard databases which allow the incremental checking of integrity constraints and thus make the process more efficient. But all this fits well into the framework of deductive databases while in other systems one can write only procedures or triggers which perform some checks and it is a difficult task to verify that these procedures really enforce a given set of constraints.

For instance, in [FSS+92] an application is mentioned, where the deductive database system LOLA is used for checking integrity constraints in the parts database of a car manufacturer. The deductive database systems EKS-V1 [VBK+92] and VALIDITY [FGVLV95] have special support for integrity checking.

- **Graph-Structured Data:** In general, applications of graphs (as known from combinatorics), are also good applications for deductive databases, because they often need to find paths. Some developers of deductive databases have tried to build good algorithms known from graph theory into their system (selectable with specific control statements for the optimizer). However, the graphs stored in deductive databases can be so big that they do not fit completely into main memory, so that "locality of access" is an issue here, which is usually not considered in graph theory.
- **Integration of Heterogeneous Databases:** Here the powerful view concept of deductive databases is very helpful. Currently many companies still have multiple databases and are not able to completely integrate them into a single system, but want at least a common view on all their data. As mentioned above, the deductive database system SDS [KSSD94] was specifically designed to work in such a heterogeneous database environment.
- **Parsing:** There is a strong relation between logical rules and grammar rules — in fact, Prolog was mainly invented for natural language analysis. But if we want to analyse natural language, then there are obviously large amounts of data (e.g. a lexicon), so we need database support. Definite clause grammars (DCGs) are standard material in any Prolog textbook. However, Prolog implementations of DCGs do not allow left-recursive rules, and backtracking involves the duplicate construction of syntactical structures. This is improved by using bottom-up evaluation, as done in deductive databases.

  In [FSS+92, SF95] a system for the morpho-syntactical analysis of Old Hebrew texts is described, which was implemented in the deductive database system LOLA and was used to analyse the grammar of the complete old testament (an obviously quite large set of data). Also the flexibility and support for incremental design of deductive databases was helpful in this application, since the grammar had to be refined via experiments with the data.

In summary, there is a large potential of applications which could profit from deductive databases. And, vice versa, these applications might suggest some important extensions of deductive databases.

Top-Down vs. Bottom-Up Evaluation

We will now give a first impression how a deductive database might work. The main task of a database system is of course to answer queries. Quite different query evaluation algorithms have been developed, all with their own features and problems.

Probably the first "deductive databases" were Prolog-interpreters or later compilers. Of course, they had no specific database support, one had to load all tuples (facts)

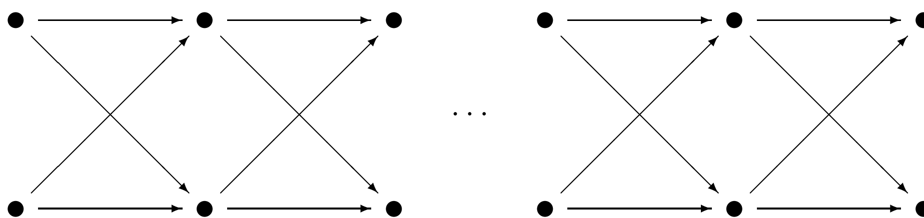
into main memory at the beginning of the Prolog session. This severely restricts the size of the "database", but otherwise a Prolog system can be seen as an implementation of Datalog. Prolog uses a theorem-proving method called SLD-resolution: Given a query $A \wedge \cdots$, it searches for matching rules of the form $A \leftarrow B_1 \wedge \cdots \wedge B_n$, and then recursively tries to prove $B_1 \wedge \cdots \wedge B_n \wedge \cdots$ as a subquery. Of course, if A is given as a fact, it can be simply deleted. Furthermore, variables can be instantiated by means of "unification". So the rules are used from head to body, and the process is goal-directed: Only rules are touched which are useful for proving the given query. This form of query evaluation is called "top-down", because one usually thinks of the query being at the top, and the given facts at the bottom. The advantage of top-down evaluation is that if it works well, then it is really fast. However, it might fail badly. First, it might not terminate, for instance in case of a rule like

$$p(X) \leftarrow p(X).$$

Of course, nobody would write such a strange rule (except perhaps a logician), but the following program for transitive closure is really standard:

$$\begin{array}{lcl} \mathit{path}(X, Y) & \leftarrow & \mathit{edge}(X, Y). \\ \mathit{path}(X, Y) & \leftarrow & \mathit{edge}(X, Z) \wedge \mathit{path}(Z, Y). \end{array}$$

It assumes that a directed graph is stored in the database relation *edge*, and computes pairs of nodes (vertices), which are connected by a path. Now this program runs into an infinite loop if the graph contains cycles. Of course, if the programmer knows this before, he/she can write a more complicated Prolog-program which detects cycles. But this is extra work and makes the rules less understandable. Furthermore, even if there are no cycles, and Prolog terminates, it might use exponential running time, for instance in case of the following graph:



There are exponentially many paths in this graph, and Prolog follows them all. But the number of connected node pairs is quadratic.

This is the advantage of the bottom-up query evaluation: It guarantees termination and a polynomial behaviour for any pure Datalog program (like the transitive closure above). Bottom-up evaluation works by applying the rules to the given facts, thereby deriving new facts, and repeating this process with the new facts until no more facts are derivable. The query is considered only at the end, when the facts matching the query are selected. Of course, it is very inefficient to compute a large number of facts which are irrelevant for the given query. In contrast, top-down evaluation performs only "relevant work", but might perform the same work again and

again — maybe infinitely often. So the result is that in many practical examples, top-down query evaluation runs much faster than "naive" bottom-up evaluation, but bottom-up evaluation is safer because of its guaranteed polynomial behaviour.

Naturally, people have tried to combine bottom-up and top-down query evaluation in order to have both advantages: being goal-directed and avoiding duplicate work. The standard method, used in many deductive database systems, is called the "magic set" transformation [BMSU86, RLK86, Bry90b]. It introduces new predicates which encode the queries occurring during top-down evaluation, and rewrites the rules in such a way that they are only applicable if the head literal is needed in order to answer the query (a formal definition is given in Chapter 3).

For instance, let us consider the following logic program which computes the Fibonacci numbers:

$$\begin{array}{lcl} \mathit{fib}(0,0). & & \\ \mathit{fib}(1,1). & & \\ \mathit{fib}(N,F) & \leftarrow & N > 1 \wedge \\ & & N_1 = N - 1 \wedge \mathit{fib}(N_1,F_1) \wedge \\ & & N_2 = N - 2 \wedge \mathit{fib}(N_2,F_2) \wedge \\ & & F = F_1 + F_2. \end{array}$$

Here, $\mathit{fib}(N,F)$ means that F is the N-th Fibonnaci number. Let us assume that the query is for instance $\mathit{fib}(10,X)$ ("What is the tenth Fibonacci number?").

Because this program contains arithmetic predicates, it is not a pure Datalog program. And in this case bottom-up evaluation does not work: It computes more and more Fibonacci numbers and does not terminate because the facts matching the query are selected only at the end. Top-down evaluation terminates, but it is also very inefficient: For instance to compute the tenth Fibonacci number, it evaluates 177 calls to fib, because it does not remember which numbers it has computed already. In general, the time top-down evaluation needs to compute the N-th Fibonacci number grows exponentially, while there is a simple linear time algorithm.

Now the magic set transformation introduces a predicate $m_\mathit{fib}^{\mathtt{bf}}$, which contains the arguments, for which the Fibonacci function has to be evaluated. Then the rules are only applicable if the resulting Fibonnaci-number is really needed:

$$\begin{array}{lcl} \mathit{fib}(0,0) & \leftarrow & m_\mathit{fib}^{\mathtt{bf}}(0). \\ \mathit{fib}(1,1) & \leftarrow & m_\mathit{fib}^{\mathtt{bf}}(1). \\ \mathit{fib}(N,F) & \leftarrow & m_\mathit{fib}^{\mathtt{bf}}(N) \wedge N > 1 \wedge \\ & & N_1 = N - 1 \wedge \mathit{fib}(N_1,F_1) \wedge \\ & & N_2 = N - 2 \wedge \mathit{fib}(N_2,F_2) \wedge \\ & & F = F_1 + F_2. \end{array}$$

Of course, the predicate $m_\mathit{fib}^{\mathtt{bf}}(0)$ has to be defined in such a way that it contains those arguments to the Fibonacci function which occur during the computation:

$$\begin{array}{lcl} m_\mathit{fib}^{\mathtt{bf}}(10). & & \\ m_\mathit{fib}^{\mathtt{bf}}(N_1) & \leftarrow & m_\mathit{fib}^{\mathtt{bf}}(N) \wedge N > 1 \wedge N_1 = N - 1. \\ m_\mathit{fib}^{\mathtt{bf}}(N_2) & \leftarrow & m_\mathit{fib}^{\mathtt{bf}}(N) \wedge N > 1 \wedge N_2 = N - 2. \end{array}$$

If this "transformed" program is evaluated bottom-up, it computes only Fibonacci numbers needed for the query, and it computes every Fibonacci number only once. The goal-direction is inherited from top-down evaluation, while the memoing of already computed facts is from bottom-up evaluation. By the way, it is a standard technique in functional programming to create a list of previous calls to a function and to search in this list before really executing the function.

It is by now folklore that bottom-up evaluation after the "magic set" transformation is at least "as efficient as" top-down evaluation. This is for instance what the title "Bottom-Up beats Top-Down for Datalog" of ULLMAN's paper [Ull89a] suggests. The result was very important because top-down evaluation was well-known in logic programming, and had been successfully used in many practical applications. With this result it seemed that deductive databases could be equally successful — or even more, because of the additional functionality. Of course, the implementation techniques of logic programming were better developed, and databases might introduce some overhead, but the result caused much optimism. It seemed achievable that good implementations of deductive databases will eventually beat Prolog systems.

However, ULLMAN in fact has not used the top-down algorithm of Prolog as a measure for comparison, instead he used an algorithm called QRGT-resolution (developed by himself). And in fact it is easy to see that the same result does not hold for Prolog's SLD-resolution. It is already wrong for the standard transitive closure program (see above), applied to a graph which is a simple straight line:

Here, the running time of Prolog's SLD-resolution is $O\big(n * \log(n)\big)$, while magic sets need more than $O(n^2)$. To be fair, already ULLMAN noted in a footnote of [Ull89a]:

> "However, Prolog *implementations* usually use a form of tail recursion optimization that, for certain examples, such as the right-linear version of transitive closure, will avoid rippling answer tuples up the rule/goal tree, and thus can be faster than QRGT."

Although it is true that Prolog implementations have a tail recursion optimization, the main goal of this optimization is to save memory; the improvement of the running time is only a side effect. And in fact, the difference in performance can already be understood on the abstract level of SLD-trees, we do not have to look at the internal data structures of a Prolog system. We will investigate this example more formally in Chapter 3, and show that the problem indeed occurs only in tail-recursive programs, and that the only reason is the "materialization of lemmas" during bottom-up evaluation of the magically rewritten program. These questions have been investigated several times in the literature, but we believe that our formalizations and proofs (given in Chapter 3) are especially useful, simple, and clear. Furthermore, we will show that by using BRY's idea of deriving magic sets from a metainterpreter [Bry90b], we can get (at least for tail-recursive programs) a rewriting technique which directly mimicks SLD-resolution.

The Need for Nonmonotonic Negation

Above, we said that a deductive database system can also be seen as an automated theorem prover for some simple subset of logic. However, there is one important difference: Negation is usually not treated as the negation of classical logic, but as some form of "negation as failure to prove". In this way deductive databases violate the principle of monotonicity, which is fundamental to classical logic: If we add further facts to the database, it might happen that previous answers become invalid. Such a behaviour is impossible in classical logic.

Naturally, if one changes the logic in such a way, one should have good reasons for this. In order to see the need for nonmonotonic negation, let us first consider a simple example. Suppose that we want to create a small database with information about the computers of our institute. In the tabular representation usually used in relational databases, this would look as follows:

Name	*Type*	*Bench*	*Date*	*Price*
wega	*SPARCserver 330*	89	11/89	84 425
pollux	*SPARCstation 1*	113	11/89	28 860
sirius	*SPARCstation ELC*	72	11/91	9 967
regulus	*SPARCstation 10 Mod. 30*	33	12/92	26 443
spica	*SPARCstation 10 Mod. 20*	42	12/92	18 126
polaris	*SPARCstation 10 Mod. 20*	36	12/92	18 126
krypton	*IBM RS/6000 Mod. 220*	110	12/92	15 972
prokyon	*SPARCclassic*	88	8/94	8 973
capella	*Linux 486/66*	42	11/94	4 500
deneb	*SPARCstation 5 Mod. 110*	32	10/95	17 478
antares	*Linux Pentium /100*	16	10/95	6 900

(The value in column *Bench* shows the number of seconds needed to format (with LaTeX) this thesis, so lower values mean higher performance.)

Usually, such a table is represented in logic as a set of facts:

$$\begin{aligned}&\mathit{computer}(\mathit{wega},\ \text{'}\mathit{SPARCserver\ 330}\text{'},\ 89,\ 11/89,\ 84\,425).\\&\ldots\\&\mathit{computer}(\mathit{antares},\ \text{'}\mathit{Linux\ Pentium/100}\text{'},\ 16,\ 10/95,\ 6\,900).\end{aligned}$$

Now if we ask the query, "Does the institute have a computer called *atair*?",

$$\mathit{computer}(\mathit{atair},\ _,\ _,\ _,\ _,\ _),$$

we expect of course the answer "no". But this answer is not logically correct: Given only the above set of facts, the system must logically answer "I don't know". The reason is that we have specified explicitly only what is true about the relation *computer*, and not what is false.

The nonmonotonic logic used in deductive databases automatically assumes that everything else is false, so we get the intended answer. However, it might seem at first

that changing the logic is much too drastic. So, how can we specify this in standard first order logic? Obviously, it is impossible to explicitly enumerate all false facts like

$$\neg computer(atair,\ \text{'}\mathit{SPARCstation\ IPC}\text{'},\ 74,\ 8/91,\ 12000).$$

(By the way, even if there were only finitely many constants, we would need a "domain closure axiom" in addition in order to conclude that the above answer is "no".) A much better solution seems to be an explicit definition of the predicate "*computer*" by means of a completion-formula [Cla78] like

$$\begin{aligned}
&\forall\, N,\ T,\ B,\ D,\ P:\\
&computer(N,\ T,\ B,\ D,\ P)\\
&\leftrightarrow\quad (N = wega \land T = \text{'}\mathit{SPARCserver\ 330}\text{'} \land B = 89 \land D = 11/89 \land P = 84\,425)\\
&\quad\ \lor \ldots\\
&\quad\ \lor (N = antares \land T = \text{'}\mathit{Linux\ Pentium/100}\text{'} \land B = 16 \land D = 10/95 \land P = 6\,900).
\end{aligned}$$

However, even with this prerequisite, the answer "no" is not justified, because it is not clear to a purely logical reasoner that *atair* and for instance *wega* are really two different objects — it might be only different names for the same object, such as "the butler" and "the murderer" in some thrillers. What we need are the "unique name axioms", such as "$atair \neq wega$". Of course, if we have infinitely many constants, these axioms cannot be written down explicitly. And even if there are only finitely many constants, the set of UNA-axioms grows quadratically, so it is very impractical to work with them.

Of course, it might be possible to construct a theorem proving algorithm which uses the unique name axioms implicitly. However, this is in fact already a change of the logic. But it is not such a big change as we propose, because the logic remains monotonic. This solution would suffice for relational databases, but it does not suffice for recursive rules. For instance, the transitive closure has no explicit definition like the one given for *computer* above. In (nonmonotonic) Datalog, pairs of connected nodes in a graph can be defined as follows:

$$\begin{aligned}
path(X,\,Y) &\;\leftarrow\; edge(X,\,Y).\\
path(X,\,Y) &\;\leftarrow\; edge(X, Z) \land path(Z,\,Y).
\end{aligned}$$

Again, the intention is that *path* contains only those tuples which are derivable by these rules. It might seem at first that the following axiom would constrain *path* in the correct way:

$$\begin{aligned}
\forall\, X, Y:\ path(X,\,Y) \;\leftrightarrow\; &\quad edge(X,\,Y)\\
&\lor\ \exists\, Z:\ \big(edge(X, Z) \land path(Z,\,Y)\big).
\end{aligned}$$

However, this does not work for cyclic relations *edge*, i.e. it does not enforce that *path* is only the transitive closure. For instance, if *edge* consists of the single tuple (a,a), also the following relation would be a model of the above formula:

$$path := \big\{(a,a),\ (a,b)\big\}.$$

In fact, it can be proven that no set of first order formulas works if *edge* can be any finite relation, because the transitive closure is not first order definable (this is also the deeper reason why it is not expressible in SQL).

At this point, we could still solve our problems by simply translating the “I don’t know”-answer of the first order theorem prover into “no”. However, this works only as long as we do not use negation explicitly. For instance, suppose that the computers *pollux* and *capella* got faulty. We represent this in another relation:

faulty
pollux
capella

It is now natural to ask, “Which computers are currently available?”. Of course, we could first query all computers, write the result down, and then query all faulty computers and take the set-difference. But a fundamental principle of query language design is that such simple combinations of queries should again be a valid query. For instance, in Datalog we get the available computers by means of the following query:

$$computer(X,\ _,\ _,\ _,\ _) \wedge \mathbf{not}\, faulty(X).$$

Note that here “**not**” does not denote the negation of first order logic, but “it is not provable that”. Of course, it is also possible to define a derived relation (a “view” in database terms), which contains the available computers:

$$available(X) \leftarrow computer(X, \ldots) \wedge \mathbf{not}\, faulty(X).$$

This is not as simple as it seems at first, because now the theorem prover has to reason about something not being provable while it is performing a proof. For instance, a typical paradoxy is

$$p \leftarrow \mathbf{not}\, p.$$

Cases like this have lead to quite a number of semantics for nonmonotonic negation, we will consider this in greater detail in Chapter 4.

Returning from this technical discussion, it is also important to note that human beings reason nonmonotonically in such a way. For instance, it is much simpler for us to write down the positive facts than to create correct $\leftrightarrow$-definitions (if at all possible). Since deductive databases are intended to be used not only by logicians, it is important to find a formalism which is similar to the way people think. For instance, it happens quite often that we say “if”, when we really mean “if and only if”. We like to concentrate on the important things and leave the rest to be understood without saying.

The field of artificial intelligence is investigating how to formalize “common sense”. For instance, John McCarthy has considered in [McC80] the well-known puzzle of missionaries and cannibals:

> Three missionaries and three cannibals come to a river. A rowboat that seats two is available. If the cannibals ever outnumber the missionaries on either bank of the river, the missionaries will be eaten. How shall they cross the river?

Of course, in order to find a solution algorithmically, one usually describes the problem space as a set of states with the allowed transitions between them. But, as McCarthy argues, it is really a big step from the above problem description in natural language to the proper formalization:

> The second reason why we can't *deduce* the propriety of Amarel's representation is deeper. Imagine giving someone the problem, and after he puzzles for a while, he suggests going upstream half a mile and crossing on a bridge. "What bridge," you say. "No bridge is mentioned in the statement of the problem." And this dunce replies, "Well, they don't say there isn't a bridge." You look at the English and even at the translation of the English into first order logic, and you must admit that "they don't say" there is no bridge. So you modify the problem to exclude bridges and pose it again, and the dunce proposes a helicopter, and after you exclude that, he proposes a winged horse or that the others hang onto the outside of the boat while two row.
>
> You now see that while a dunce, he is an inventive dunce. Despairing of getting him to accept the problem in the puzzler's spirit, you tell him the solution. To your further annoyance, he attacks your solution on the grounds that the boat might have a leak or lack oars. After you rectify that omission from the statement of the problem, he suggests that a sea monster may swim up the river and may swallow the boat. Again you are frustrated, and you look for a mode of reasoning that will settle this hash once and for all.

We see that the problem is again one of implicit negation: First, there are no other means of transportation besides the one mentioned in the puzzle, and second, there are no other problems, besides that the cannibals might eat the missionars.

Applications of Disjunctive Information

The above kind of deductive databases with some form of nonmonotonic negation is more or less standard now (although the allowable uses of negation are quite restricted, we will investigate this further in Chapter 4).

In this thesis, we go one step further and also allow to represent disjunctive information in the database.

Usually, a deductive database has only one intended model corresponding to the completely known state of the real world. However, there are many applications where we do not know exactly which of some possible states is the correct one. Examples are:

- Null values: For instance, an age "around 30" can be 28, 29, 30, 31, or 32.
- Legal rules: The judge always has some freedom for his decision, otherwise he/she would not be needed. So laws cannot have a unique model.
- Diagnosis: Only at the end of a fault diagnosis we know exactly which part of some machine was faulty. But as long as we are searching, there are different possibilities.

- Biological inheritance: E.g. if the parents have blood groups A and 0, the child must also have one of these two blood groups (example from [Lip79]).
- Natural language understanding: There are many possibilities for ambiguity here, and this is represented most natural in multiple intended models.
- Conflicts in multiple inheritance: If we want to keep as much information as possible, we would assume the disjunction of the inherited values [BL93].
- Reasoning about conflicts in concurrent updates: If we do not know in which sequence two processes are executed, we can assume only the disjunction of the two values they assign to some variable.

It is often possible to code these situations somehow in Horn clauses, but this is difficult, indirect, and therefore rather error-prone. At least in the first phases of database design we should use the most powerful tools available to be as near to the real world as possible. Later, it might be useful to apply some transformation to the database in order to increase efficiency. Naturally, it would be very useful if the same deductive database system could be used in every step of the transformation and we would have the possibility to stop whenever we have reached a sufficient degree of efficiency. Our goal is a system which allows gradual transitions between the disjunctive and the standard Horn case. For instance, if the database is "mostly Horn", i.e. there are only a few disjunctive rules which are not used too heavily, then the efficiency should also be nearly that of a standard deductive database.

However, it must be said at this point that disjunctive databases are really more powerful than standard deductive databases: A translation from a set of disjunctive rules into Horn clauses is not always possible, at least not without increasing the number of objects, for instance by introducing lists. This follows from complexity-theoretic considerations: For instance, it is known that a satisfiability-test for a set of propositional clauses is NP-complete. Such a satisfiability-test can easily be done by a disjunctive deductive database. Clauses with at least one positive atom, such as $p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4$, are written down in the form

$$p_1 \vee p_2 \leftarrow p_3 \wedge p_4,$$

while clauses without a positive atom, such as $\neg p_1 \vee \neg p_2$, are represented as

$$\mathit{false} \leftarrow p_1 \wedge p_2.$$

Now the set of clauses is unsatisfiable iff *false* follows from this database. So it is possible to specify this NP-complete problem in a disjunctive deductive database, while bottom-up evaluation of a standard Horn database is always polynomial.

So disjunctions really increase the expressive power of a deductive database, and make it more similar to a general automated theorem prover. However, an important difference is that we require the clauses to be "range-restricted", i.e. every variable must appear in a positive body literal. So a deductive database can reason only about the objects explicitly known to it, usually not about all integers and so on. Such tasks, which are required for instance in program verification, need an automated theorem prover, not a deductive database.

If we wanted to relax this restriction, as even some standard deductive databases do, we would loose a lot of efficiency, because database techniques are no longer applicable. So we would have to use also the implementation techniques for theorem provers, and this is a different area, which is not treated here. Maybe, we can say that a database in contrast to a theorem prover is always able to give specific objects as answers to queries — not only the information that there is such an object, or that all objects have this property.

As can be seen from the above discussion, we have now left the realm of standard deductive databases, and it is quite a big step to disjunctive databases. In fact, some people believe that disjunctive databases will never reach an acceptable degree of efficiency, such that they can be used in practice. They say that if already standard deductive databases have efficiency problems, and are currently seldom used in real applications, why look at something more general? Well, the answer is that disjunctive databases can really solve problems for which standard deductive databases are not applicable. Naturally, disjunctive databases will always be used only in more specialized application domains (maybe niches), and it will need more time until there are powerful implementations. However, for us it is also interesting to see how techniques developed for the Horn case are really applicable in a more generalized setting. Furthermore, automated theorem provers, which are even more general, have been successfully used for solving practical problems. So there is hope that disjunctive logic programming will also become useful for applications which do not need the full power of automated theorem proving, but have a fact base of medium size, which cannot be handeled by standard theorem provers.

For Horn clauses, it is by now generally accepted that top-down and bottom-up query evaluation techniques both have advantages of their own and that none is superior to the other for all applications. Furthermore, the cross-fertilization of both approaches was very successful [Bry90b]. But for disjunctive rules, up to now top-down approaches were dominant [LMR92]. Although it is known that the bottom-up immediate consequence operator T_P can be directly generalized to disjunctive rules [MR90], this is usually considered only as a means to define the semantics, not as something amenable to implementation.

One important reason for this is that T_P as defined in [MR90] allows very often the derivation of exponentially many disjunctive facts (although this is not made explicit in that paper, see Chapter 5 for an example). Now our contribution is an optimization of T_P which makes the resolvable literal in a disjunctive fact unique. In many cases, this reduces an exponential behaviour to a polynomial one. We thereby improve an optimization which is already known for positive hyperresolution [CL73] (the theorem-proving counterpart of the disjunctive T_P) for quite a long time.

By applying these ideas, disjunctive rules can be naturally translated into Horn clauses with lists. This allows a direct implementation on standard deductive database systems. Of course, more specialized data structures would be very useful for a really efficient implementation. However, by the inverse translation, it is also possible to generalize standard implementation techniques developed for Horn clauses to the disjunctive case. This is our main point.

For instance, one of the very basic things a standard deductive database does is that it applies the rules in the order of the predicate dependencies and iterates only (mutually) recursive rules. However, prototype implementations for disjunctive databases usually iterate all rules until nothing changes. The reason for this is that a seemingly innocent rule like $p(X) \leftarrow q(X)$ needs to be applied two times if there is e.g. the disjunctive fact $q(a) \vee q(b)$. The notion of disjunction types developed in Section 5.1 does allow to determine an evaluation order for disjunctive rules.

In fact, up to now such an analysis was nearly impossible, because there are so many ways to derive a disjunctive fact. With our optimization, the possibilities to resolve with a disjunctive fact are drastically reduced. In the case of positive disjunctive databases, where one is only interested in disjunctions of answer-literals, the resolvable literal within a disjunction can be made unique. However, even in the general case (where we need more information in order to evaluate negative body literals) our technique reduces the the number of different derivations of the same disjunctive fact.

By now, there is quite a lot of research on disjunctive information, e.g. [Lip79, RT88, MB88, Dem91, LMR92, SA93, EGM94]. However, there are still very few actual implementations (e.g. [SA93]), and even less using database techniques. We believe that disjunctive databases will only get away from the (early?) prototype state if we can make the best possible use of what is already known for standard deductive databases. This motivated the work presented in Chapter 5.

The Crisis of Deductive Databases

It seems that at the moment quite a number of deductive database researchers are a little frustrated: There is a nice theory, and there are some larger prototype implementations (see Figure 1.3 and [Ram94, RU95]), but deductive databases have not yet made their way into industry: No software company is currently developing (let alone selling) a deductive database system, and the available prototype systems are also used nearly exclusively in the universities themselves.

In fact, there has been a commercial deductive database system, called SDS [KSSD94], but it did not sell well enough. The development began 1986 and ended 1990. The system was quite ahead of the time, for instance by being especially designed for the integration of heterogeneous systems. At the beginning, it ran only in a Lisp environment, which might have deterred possible customers. Also, some companies had just moved to a relational database, and were not ready for another change. Finally text books on deductive databases were missing at that time, and application programmers did not know Datalog.

One reason, why deductive databases did not (yet) get their way is also that currently object-oriented databases are more fashionable. There are a number of companies producing object-oriented database systems, and also quite a lot of industrial users. However, one of the achievements of relational databases was their declarative query language, namely to say only *what* is wanted, and not *how* to compute it. From this point of view, object-oriented databases (in their current development state) are a step backward: It is often necessary to program in C++ to get the desired result.

System	Start	Group	References
LDL	1984	MCC	[Zan88, NT89]
NAIL!	1985	Stanford University	[MUVG86, Ull89b]
SDS	1986	MAD Intelligent Systems	[KG90, KSSD94]
ConceptBase	1986	RWTH Aachen	[JGJ+94]
Aditi	1988	University of Melbourne	[Ram93, VRK+94]
CORAL	1988	U. Wisconsin at Madison	[RSS92, RSSS94]
XSB	1989	SUNY at Stony Brook	[SSW94]
LOLA	1988	TU München	[FSS+92]
EKS-V1	1989	ECRC	[VBK+92]
LDL++	1990	MCC	[Zan92, ZAO93]
Glue-NAIL!	1991	Stanford University	[PDR91, DMP94]

Figure 1.3: Some Implementations of Deductive Database Technology

The idea of declarative programming is summarized in KOWALSKI's equation: 'Algorithm = Logic + Control". In standard imperative programming, the control part is explicit and the logic implicit, while in declarative programming, it is the other way round. The advantages of declarative programming are:

- **Enhanced Productivity:** It is not unusual that an equivalent formulation in Prolog or Datalog is ten times shorter than in C or C++. Since there is also reason to assume that the time an experienced programmer needs for one line of code is more or less independent of the language, this can lead to drastic savings in time and money.
- **More Powerful Optimization:** Since no fixed execution algorithm has influenced the language design, the space of possible optimizations is much bigger. On the other hand, optimization is not only possible, but also necessary, since a naive evaluation algorithm would be to inefficient.
- **Simpler Parallelization:** Imperative programming languages are often influenced by classical machine models, and a not very high abstraction of the machine language. Therefore, programs written in such languages cannot make full use of new machine architectures. Thus, programs written in declarative languages will probably live longer.
- **One Logic, Many Algorithms:** The dream of declarative programming is that the system automatically selects an optimal algorithm for the specified problem. Of course, this is impossible. However, it is possible to have an extensible optimizer: If the algorithm used for the general case should be too slow, the programmer can add annotations to guide the optimizer, or integrate new algorithms. In fact, the optimizer can have a whole library of possible algorithms at its disposal, similar to the different algorithms for joins and other operations in relational databases. Another simple example for "one logic, many algorithms" is the possibility to create or delete indexes in relational databases without hav-

ing to change application programs. This gives a good way to adapt to changing usage profiles.

- **Simpler Verification:** Since the semantics of the language is simpler and better formalized, also the verification of programs in this language is simpler. For instance, there exists a system for proving the termination of nontrivial Prolog programs, which is able to prove its own termination [Plü90]. I do not know of an equally powerful system for imperative programming languages.

Of course, object-oriented databases have also certain advantages, for instance an extensible type system, the clustering of whole objects on external memory, and a module concept. But all this can also be integrated in deductive databases, and there is currently a lot of research on deductive and object-oriented databases (DOOD systems). This is a very promising development for the future, and it seems that also the development of prototype systems is going this way.

In 1994, it became known that BULL was developing such a deductive object-oriented database system [RH94, FGVLV95]. However, in 1995 the whole group was closed down. This management decision is hardly understandable.

It seems to me that more research is needed before deductive database technology can really redeem its big promises. It is also necessary to look more at the needs of real applications, and to extend Datalog accordingly without sacrificing its cleanness and simplicity. Probably still more patience is needed than a company can have. Some scientists believe that it is no longer a problem of research, but only of having better implementations. For instance, on ICDE'93 there was a panel discussion "Are we Polishing a Round Ball?" [Sto93], and the topic of recursive query evaluation got especially bad marks for the potential of innovative research. Also Ramakrishnan writes in the preface to [Ram95]:

> These limitations can mostly be addressed by more careful implementations, and are perhaps understandable in research prototypes; it does not seem that fundamental new techniques are required to resolve them.

I believe that Chapter 3 of this thesis shows that there are essential problems not adequately treated in the literature. There have been quite a number of prototype implementations. If there was still no real breakthrough, this is not only a problem of implementation.

Let us also say some words on efficiency. This is often considered as very important by industrial users, whether it is really needed for the application at hand or not. It is a shame that the currently most efficient "deductive database system", namely XSB [SSW94], uses mainly Prolog technology, and not the algorithms developed in the field of deductive databases. Of course, we must say that the comparison is a little unfair, since XSB (as well as many other prototypes) are main memory systems. And database technology is intended for the management of large amounts of data, which can only be kept on external storage. However, we should be able to make efficient use of available main memory, and it is certainly an interesting and important research problem to find out why Prolog technology is more efficient here and how we could improve the techniques of deductive databases to catch up with it. There are also

certain very typical operations in deductive databases, such as the computation of transitive closures, where deductive databases should be as efficient as a hand-crafted implementation in C. In the preface of [Ram95] it was noted that for an application in program analysis, an equivalent program in Datalog was much shorter than a C-program ("about a hundred lines vs. a few thousand lines"), but also 5-10 times slower when executed on CORAL. Although this already saves money, because the man power for the additional programming time is much more expensive than buying a ten times faster computer, we must try to make the efficiency loss smaller.

It is also an interesting question, whether the deductive database researchers and implementors believe themselves in deductive database technology. In Prolog implementations it is very common to have a large part implemented in Prolog itself (e.g., the whole compiler into WAM-code). In current deductive database systems, this does not seem to happen at all. It is not an acceptable excuse that a deductive database system is mainly a database system, and cannot be used for programming tasks. One of the main selling arguments for deductive database technology is that it is an integrated programming and database environment. For example, since deductive database systems are especially well-suited to process graph-structured data, this should also apply the the predicate dependency graph extracted from the rules. If we have problems to get applications from the outside, we might start by looking at the deductive databases themselves. In an invited talk at DOOD'93, Rainer Manthey gave an overview of the possibilities to use deductive databases in their own implementation [Man93].

Currently, quite a lot of manpower has to be invested into the development of nice user interfaces. Object-oriented languages have special support for this since a long time, and it would certainly increase the acceptance of deductive databases if there would be an easy and powerful way to build user interfaces in Datalog. It seems that in order to process events triggered by the user, an integration with the concepts of active databases would be useful. This means that rules can also work as production rules, not only as deduction rules.

In this thesis, we will clarify and improve the foundations of deductive databases, since only well-understood concepts can be successfully implemented. We will return to the roots of deductive databases, namely the relation to logic and automated theorem proving, and further develop deductive databases by features which imperative languages will never have, namely powerful concepts for optimization, nonmonontic negation, and disjunctions.

What is New in This Thesis?

This thesis is structured by considering different classes of Datalog programs, one in each chapter.

Chapter 2 is devoted to the standard case of Horn-clause Datalog, but with built-in predicates (such as $<$, *sum*, and so on). Pure Datalog is well known in the literature, but it seemed necessary to review it to have a basis for later extensions and comparisons. The inclusion of the standard case makes this work more self-contained and precise, since the field has not yet reached completely canonical definitions. How-

ever, Chapter 2 is not simply a compilation of parts from other textbooks and papers. I had fun in writing it and believe that some degree of originality was reached.

Chapter 3, "Goal-Directed Bottom-Up Evaluation" also considers the case of pure Horn-clause programs, but is devoted to the special problem of simulating the goal-directed SLD-resolution by bottom-up evaluation. This chapter is based on my paper [Bra95], but contains an important improvement in the proposed method. In Section 3.1, we compare the efficiency of bottom-up evaluation after the standard "Magic Set" transformation with the efficiency of SLD-resolution. As shown by Ross [Ros91], bottom-up query evaluation with magic sets can be much slower than SLD-resolution for tail-recursive programs. We show that this happens only for tail-recursive programs, and that the only problem of magic sets is the "materialization of lemmas" (which is also done in variants of SLD-resolution which ensure termination). In any case, "magic sets" are "as goal-directed as" SLD-resolution. It is difficult to say how new these results are, certainly they are not very surprising. There are a number of similar approaches which compare variants of magic sets and/or variants of SLD-resolution [Ull89a, Sek89, RS91]. We compare the original magic sets with the real SLD-resolution, and give simple formalizations and proofs, suitable for classroom usage.

In **Section 3.2**, we show that it is possible to exactly simulate SLD-resolution for tail-recursive programs and to combine it with magic sets for other programs. A variant of magic sets with tail-recursion optimization has already been proposed by Ross [Ros91], but our goal is not only to solve this problem but to simulate SLD-resolution by bottom-up evaluation as far as we can. Formally, our method is different from the method of Ross since we work with lists of literals while he uses only pairs, and we produce range-restricted Datalog, while his method yields HiLog. However, the main reason why we believe that we can make an important contribution is that our method is based on a very simple idea, namely to simulate SLD-resolution by means of a meta-interpreter in the style of Bry's approach [Bry90b], but it gives many optimizations for free:

- If the given program is nonrecursive, then the transformed program is nonrecursive, too. For the magic set transformation, this is not necessarily the case, and this is an important problem. For instance, the magic set transformation can also be useful in standard relational databases [GM93], where the query evaluation algorithm is unable to handle recursions.
- Values for anonymous variables never get explicitly represented.
- In contrast to the "magic set" transformation, we need no extra rectification. It is automatically included in a simulation of SLD-resolution.
- Constants are "pushed downward" as far as possible already during the transformation (at "compile time").
- Also other constraints, such as $X < 100$, can be pushed into the called rules in order to abort inconsistent paths early.
- To some degree, also incomplete bindings can be passed to the called rules. In contrast, the standard magic set transformation usually assumes that every argument is either a ground term or a free variable. However, our transformation

also needs to "abstract" the binding information over some fixed depth limit.

- Finally, the idea to simulate SLD-resolution allows also a better understanding of the counting method and its generalizations [GZ92].

Probably Section 3.2 is the most important contribution of this thesis from the practical viewpoint, and Section 4.2 is the most important contribution from a more theoretical viewpoint.

In **Chapter 4**, "Negation as Failure", we extend the class of Datalog programs to allow nonmonotonic negation. In Section 4.2, we characterize the two most important negation semantics by simply requiring that certain natural elementary program transformations preserve the semantics of the program. For example, it should be possible to delete a tautological rule like $p(X) \leftarrow p(X)$ or to "unfold" a body literal, i.e. replace it by the bodies of the rules with matching head literal. In Section 4.3, we present a general framework for the computation of different negation semantics based on the idea of conditional facts [Bry89, Bry90a, DK89a, DK89b]. We show that with very weak assumptions on the semantics (namely the possibility to delete tautologies and to perform unfolding), the derivable conditional facts are equivalent to the original program. Our approach can be described as a source-level transformation of the program, and therefore is easy to understand and to verify. We believe that it also helps to understand other proposed query evaluation algorithms for specific semantics. The results of this chapter are based on my joint work with Jürgen Dix [BD94b, BD94a, BD95b, BD95a, BD95c]. Here, I specialized the results to the case of non-disjunctive Datalog, and was able to develop them a little further. The computation of the residual program and the well-founded and stable model semantics has been implemented by Dirk Hillbrecht and Michael Salzenberg [HS96] (even in the disjunctive case).

In **Chapter 5**, "Reasoning with Disjunctions", we consider another extension of pure Datalog, namely Datalog with disjunctions in the heads. We exclude nonmonotonic negation here, since the ideas presented in Chapter 4 can easily be extended to the disjunctive case [BD94b, BD94a, BD95b, BD95a, BD95c] and it seems better to consider the problems of negation and disjunctions first separately. In **Section 5.1**, we propose an optimization of the immediate consequence operator T_P which resolves only with a single literal of every disjunction fact and still is complete for deriving "answer clauses". This utilizes an idea known for "positive hyperresolution" (the theorem-proving counterpart of the disjunctive T_P) for a long time [CL73], but we show that for range-restricted clauses, the optimization can be descisively strengthened [Bra94a]. By applying this idea, disjunctive rules can be naturally translated into Horn clauses with lists. This allows a direct implementation on standard deductive database systems.

We then introduce the notion of disjunction types, which allow us to generalize techniques based on the predicate dependency graph (such as determining an evaluation order) to disjunctive rules. By specializing the meta-interpreter for disjunctive rules with respect to these disjunction types, we get a quite reasonable Horn-clause implementation of a disjunctive program. By the way, the notion of disjunction types is very similar to the "node types" used in Section 3.2. This work has not been previously published.

In **Section 5.2**, we use the reasoning algorithm developed in Section 5.1 for query evaluation under the stable model semantics. The stable model semantics is inherently disjunctive, and is often used to express disjunctive knowledge in non-disjunctive logic programs. So it is quite evident that for computing stable models, we need to be able to reason with disjunctions.

Bibliographical Notes

I would like to mention a number of textbooks and survey papers relevant to the topics treated in this thesis. Of course, this thesis is sufficiently self-contained so that it should not be necessary to consult a textbook. In fact, the first four chapters of this thesis may themselves be seen as a kind of textbook. However, if the reader should be interested to look at related material or alternative presentations of common material, we give the necessary references here.

Textbooks on deductive databases (or having a substantial part on them) are [Ull88, Ull89b, NT89, CGT90, Gog90, Das92, Nus92, CGH94, AHV95]. Survey papers on deductive databases are, e.g., [Llo83, GMN84, BR86, Min88b, HPRV89, CGT89, KG90, Zan90, Con91, Tsu91b, RU95]. There is also a special issue of the VLDB journal on actually implemented deductive database systems [Ram94], and there is a collection of papers on applications of deductive databases [Ram95]. General textbooks on databases with a chapter on deductive databases are, e.g., [GV89, Dat90]. For the history of deductive databases, see, e.g., [Min88b, RU95]. Textbooks on nonmonotonic reasoning are, for example, [Rei85, Bes88, Eth88, Łuk90, Bre91, MT93, GHR94]. Overview papers on the semantics of nonmonotonic negation are [She88, PP90, AB94, Prz94, Prz95, Dix95a, Dix95b, Dix95c, DF96]. There is currently only one textbook on disjunctive logic programming, namely [LMR92]. A textbook on automated theorem proving is [CL73].