An Introduction to Deductive Database Languages and Systems

Kotagiri Ramamohanarao and James Harland

1 Introduction

One of the most fundamental uses of a computer is to store and retrieve information, particularly when there is a large amount of data to be stored, or there are complex manipulations that must be performed on it. There has been a large amount research on the most efficient techniques to store and retrieve data stored in computers, and the associated problems now have satisfactory solutions. However, the problem of understanding and interpreting this large amount of information remains, particularly when the amounts of data belong to complex domains, such as those involving mineral exploration and financial analysis.

In order to tackle this problem, a mechanism for reasoning about the stored information is necessary. Such a mechanism needs to be able to cope with large amounts of information as well as to perform sophisticated inferences, and to draw the appropriate conclusions. A framework in which these problems may be attacked is given by the field of deductive databases. Deductive databases not only store explicit information, in the manner of a relational database, but also rules, which enable inferences to be made based on the stored data. This area is an outgrowth of the field of logic programming, in which mathematical logic is used to directly model computational concepts. Together with techniques developed for relational databases, this basis in logic means that deductive databases are capable of handling large amounts of information as well as perform reasoning based on that information.

There are many application areas for deductive database technology. One such area is that of decision support systems. In particular, the exploitation of an organisation's resources requires not only sufficient information about the current and future status of the resources themselves, but also a way of reasoning effectively about plans for the future. The present generation of decision support systems are severely deficient when it comes to reasoning about future plans. Deductive database technology is an appropriate solution to this problem.

Another fruitful application area is that of expert systems. There are many applications of computing in which there are large amounts of information, from which the important facts may be distilled by a simple yet tedious analysis. For example, 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 retrieve relevant cases during diagnosis reduces the risk of error in such circumstances. Deductive database technology allows the analysis of this data to be performed more efficiently and with a 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; large amounts of data may be generated, which can then be analysed for clues suggesting the presence of the desired mineral.

Planning systems are another application area. For example, a student planning a course of study at a university, or a passenger planning a round-the-world trip often need to consider a large body of information, as well as the ability to explore alternatives and hypotheses. A deductive database is able to advise students about pre-requisites and regulations on the choice of subjects, or a traveller of the financial implications of a given change in itinerary.

Deductive database systems have been the subject of extensive research, and several prototype deductive database systems are now emerging, as evidenced by the descriptions appearing elsewhere in this Issue.

The rest of this paper is organized as follows. In Section 2, we discuss various language issues for deductive database systems, and in Section 3 implementation schemes for these systems are described. In Section 4 we briefly describe various implementations of deductive database systems, and in Section 5 we present our conclusions.

2 Deductive Database Languages

In this section we briefly discuss some language issues relevant to deductive databases. For more details, the reader is referred to [27].

The field of deductive databases has had close links with the logic programming community, and hence much of the development of deductive database systems has centred around languages based on Horn clauses. This class of formulae form the basis of Prolog, and are powerful enough to encode Turing machines [40].

A *Horn clause* is generally written as

$$p(\tilde{t}) \text{ :- } q_1(\tilde{t_1}), \ldots, q_n(\tilde{t_n})$$

where p and $q_1, \ldots q_n$ are predicate letters, $n \geq 0$, and all variables which occur in the terms $\tilde{t}$, $\tilde{t_1}$, …, $\tilde{t_n}$ are considered universally quantified at the front of the clause.

Note that n may be 0, in which case we refer to the clause as a *fact*. Otherwise, we refer to the clause as a *rule*.

The atom $p(\tilde{t})$ is referred to as the *head* of the clause, and $q_1(\tilde{t_1}), \ldots, q_n(\tilde{t_n})$ as the *body* of the clause.

A *logic program* is a set of Horn clauses.

The terms $\tilde{t}$, $\tilde{t_1}$, …, $\tilde{t_n}$ may, in general, be arbitrary (first-order) terms, and hence may contain variables and/or function symbols.

It is often useful to consider sub-classes of this class of programs. A common restriction that is made is to only allow terms to be either a variable or a constant. Such programs are known as *Datalog* programs. An important property of such programs is that the problem of determining whether a given query is logically entailed by a Datalog program is decidable. Hence it is reasonable to expect that a deductive database system should terminate on all Datalog programs.

Not all deductive database systems restrict programs to be Datalog programs. Datalog programs are somewhat restrictive; for example, the append program is not a Datalog program, as it requires the use of function symbols.

In the deductive database field, a distinction is usually made between predicates defined by rules alone (referred to as the *intensional* database or IDB), and predicates defined by facts alone (referred to as the *extensional* database or EDB). It is not hard to see that any logic program may be rewritten so that all predicates are either IDB or EDB predicates. It is often useful to consider a given IDB for various EDBs.

Whilst Horn clauses are Turing complete [40], it is common to extend the language of Horn clauses so that the body of a clause is a conjunction of *literals*, i.e. an atom or the negation of an atom, rather than a conjunction of atoms alone. The negative literals are inferred by the use of the *Negation as Failure* rule [11]; a literal $\neg A$ succeeds if A fails. The addition of this feature gives the language more expressive power, but it can also confuse the semantics of the program somewhat. For example, consider the program below.

```
p :- ¬ q
q :- ¬ p
```

Here it is not clear whether we should interpret p as being true (and hence q being false) or vice-versa. As a result, negation generally has to be used carefully in logic programs to avoid problems of this kind. There has been a great deal of work on the semantics of negation in logic programs, and we give only a brief overview here. For more information, see papers such as [17, 16, 32, 24].

A useful class of programs in which the use of negation is restricted is known as *stratified* programs [2, 9]. Intuitively, a program is stratified if there is no recursion through negation. For example, the following program, which defines the acyclic part of a graph, is stratified.

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).

acyclic(X,Y) :- path(X,Y), ¬ path(Y,X).
```

Note that the definition of the acyclic predicate depends on the path predicate, but not vice-versa.

A more general class of programs, which is based on the same general idea, is the class of *locally stratified* programs [32]. Essentially, a locally stratified program allows recursion through negation provided that no atom depends on its own negation. Further extensions of this concept include modular stratification [35].

Another restriction that is often imposed is to consider only programs which are *range-restricted*: a program is range-restricted if every variable which appears in the head of a clause also appears in the body of the clause [6] (note that this definition can be simply extended when negative literals appear in the body of a clause). This implies that all facts in the program must be ground, i.e. contain no variables. The main advantage of this class of programs is that in the query evaluation process, full unification is not needed, but only matching, which is significantly more efficient. Also, all answers to a given query are ground, and hence there is no need to check for answers subsuming one another.

Several of the prototype systems described in this paper have implemented various combinations of the above features. Some systems only support Datalog range-restricted programs with stratified negation, some support modularly stratified programs, and/or function symbols. Some systems also do not impose any restrictions other than modular stratification. More details are provided in Section 4.

Many deductive database systems also include aggregate operators, such as sum, max, min and count. Whilst these operators allow the simple expression of many database programs, it is possible, just as in the case for negations, to write simple programs with a complicated semantics, and so many of the concepts introduced for negation such as stratification are also used for aggregate operators.

3 Implementation Schemes

There has been a significant body of research in the area of implementation of logic programming systems and deductive database systems, and a substantial body of theoretical work has been developed for such systems. In this paper we are interested only in implementation techniques for deductive databases. These implementation techniques can be broadly categorised into three main groups:

- Prolog systems loosely coupled to database systems
- Top-down evaluation with memoing
- Bottom-up methods

3.1 Prolog systems loosely coupled to database systems

Some of the early attempts to implement deductive databases were to interface a Prolog system to a database system (or a file store). These systems we refer to as Prolog database systems. These systems use Prolog computation and access appropriate database relations on a tuple-at-a-time basis. The benefit of this approach is that these systems can be implemented quickly and easily. The drawback of this approach is that the resulting system can be extremely inefficient, as access to the underlying database system is tuple-at-a-time and the resultant computation performed is similar to the nested loop join algorithm, but performing on several relations simultaneously. Several systems have been developed using this approach [34, 49, 21].

Prolog is based on the top-down computation method, which is also known as backward chaining or SLD-resolution. This method is also used in theorem proving. It starts at the query and applies the rules of the program until it arrives at the facts.

The main steps in SLD-resolution are as follows:

1. Initialize the *goal list* of literals to the query.
2. Choose a goal A_i from the goal list $A_1, A_2, \ldots, A_i \ldots A_n$. Find a rule $A :- B_1, \ldots, B_m$ such that $A\theta = A_i\theta$ for some most general unifier θ. Terminate with failure if there are no such rules.
3. Update the goal list to $(A_1, A_2, \ldots A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots A_n)\theta$.
4. If the goal list is not empty, go back to step 2. Otherwise, terminate with success; an answer to the query is contained in the substitutions.

Step 2 of the top-down algorithm has two forms of nondeterminism.

- The *computation rule* specifies which literal is to be selected.
- The *search rule* specifies the order in which the matching rules of the program are unified against the selected literal.

These two rules give the shape of the tree explored by the top-down algorithm.

In Prolog, the computation rule is to always use the leftmost literal in the goal list and the selected literal is replaced in the goal list by the body of the matching rule. The search rule is to always use the first matching rule.

Using this approach for deductive databases can result in a bottleneck, as large amounts of information can tend to "clog" the tuple-at-a-time nature of the computation. A significant development of this approach was pursued in the MegaLog system, developed at ECRC [21, 8]. MegaLog was designed to be similar to Prolog, but the main emphasis being on efficient database access. For example, MegaLog supports relational operations and indexing structures such as BANG files [13].

Note that in such systems it is possible for some Datalog programs not to terminate, and hence it is the programmer's responsibility to ensure that all queries terminate.

3.2 Top-down with memoing

To overcome the problem of the termination of top-down methods on Datalog programs, the technique of *memoing* is often used. The main problem for termination of SLD-resolution is that the refutation procedure does not recognise goals that it has previously called, and so may loop needlessly. Methods of incorporating such a check into the SLD-resolution procedure have been studied by many researchers [12, 41, 42, 43, 47], all of which may be considered variants of OLDT-resolution [39].

In its simplest form, top-down evaluation with memoing builds a tree similar to an SLD-tree except for the following restrictions and differences:

- Answers to subgoals are tabled (memoised) for future use: when the derivation proceeds from the goal list
$$A_1, \ldots, A_i, \ldots A_n$$
to a descendent goal list of the form
$$(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots A_n)\theta$$
the atom $A_i\theta$ is called an answer.

- If the subgoal A is an instance — possibly more instantiated — of a subgoal that occurred earlier in a left-to-right pre-order traversal of the tree, then A is not resolved using rules from the program, but is resolved against tabled answers.

- When a new answer is found, any subgoal that has been resolved using answers must be tested to see if it unifies with the new answer.

Although the above description is tuple-at-a-time in nature, it has been further developed to compute answers an efficient set-at-a-time manner in [43].

Essentially, the search procedure "remembers" each goal that it has called, so that the evaluation of a given goal does not repeatedly derive the same subgoal. This system is in some respects a mixture of top-down and bottom-up methods, as many of the characteristics of the system have direct counterparts in bottom-up evaluation methods.

3.3 Bottom-up methods

The bottom-up method is known also as forward chaining or fixpoint computation. It starts at the facts and applies the rules until it arrives at the query. This approach is often used in the study of the semantics of logic programs, and by many deductive databases.

This computation method can be characterised by the following steps: Let the query be $q(\tilde{Q})$.

1. Initialize M, the set of known facts, to the set of facts in the program, and add the following rule to the program:
$$\text{ans}(\tilde{Q}) \text{ :- } q(\tilde{Q}).$$

2. For each rule A :- $A_1, \ldots, A_n$, look for substitutions θ for which $A_1\theta, \ldots, A_n\theta \in M$. For each such substitution, add $A\theta$ to M.

3. If the set of known facts M has increased, go back to step 2.

4. The answer to the query is the set of ans facts in M.

The bottom-up approach naturally lends itself to the application of relational algebra techniques, as the conjunction of literals in the goal may be implemented by a sequence of join operations, for which many optimization techniques are known. However, the bottom-up method as described above completely ignores the values of any constants in a query, and therefore also derives facts which are irrelevant to the query. Relevant facts (including derived facts) are those which are used in the generation of answers to the query. The number of these irrelevant facts can be very large, and in general this can make bottom-up computation very expensive.

By contrast, the top-down method, with or without memoing, does not have this problem, as query evaluation makes use of the instantiated variables of the goal. In order to make bottom-up methods concentrate only on facts which are relevant to the query, techniques such as *magic sets* have been developed [5, 7, 4]. This is one of the most important optimization techniques for bottom-up methods.

This is a source-to-source transformation; it transforms the *program* (the rules of the database) into another program that can be evaluated more efficiently by the standard bottom-up computation we have presented.

The magic set transformation is a general transformation, and can be applied to all programs. However, special care is needed when dealing with programs containing negations. The transformation provides focus equivalent to top-down computation, so that only facts relevant to the query are generated.

For example, consider the program and query below.

```
?- partof(2, Y).

partof(X,Y) :- component(X,Y).
partof(X,Y) :- component(X,Z), partof(Z,Y).
```

Under the magic set transformation, the program and query become

```
?- partof_m(2, Y).

magic_partof(2).
magic_partof(Z) :- magic_partof(X), component(X,Z).
partof_m(X,Y) :- magic_partof(X), component(X,Y).
partof_m(X,Y) :- magic_partof(X), component(X,Z),
          partof_m(Z,Y).
```

The standard bottom-up evaluation of these rules produces the same result for this query as the evaluation of the unmodified rules would, but it looks at only the relevant facts. The `magic_partof` relation initially contains only the tuple $\langle 2 \rangle$, the input value for the first argument of `partof`. At each stage in the bottom-up evaluation of `magic_partof`, the computation adds to this relation the values of the first argument of `partof` that a top-down evaluation of the query would see at the corresponding depth in the search tree. At the end, `magic_partof` contains the *magic set*, i.e. all the values for the first argument of *partof* that the top-down evaluation of the query would ever see.

The modified rules of `partof` then use the magic set to avoid computing the parts of the `partof` relation that are not relevant to the query.

Below we provide a very simple form of the magic set transformation algorithm, which may be applied to any program.

- For each derived predicate of the program, create a magic predicate by prefixing *magic_* to the predicate name. The arguments of this new predicate are the bound arguments of the original predicate.
- For each rule, add a magic atom to the front of the rule; the arguments of this atom are the bound arguments of the rule head.
- For each modified rule of the program, create a new rule for each call to a derived predicate `p` whose bound arguments are $X_1, X_2, \ldots, X_n$. The head of this new rule is `magic_p`$(X_1, X_2, \ldots, X_n)$ and the body is the literals preceding the call.

There are several other optimization algorithms which have been developed, some for particular classes of programs such as linear recursions [23, 20], and various others which are more generally applicable [7, 38, 36, 37, 19, 25].

Several of the prototype systems described below have implemented various combinations of the above methods and techniques. Most systems implement either bottom-up evaluation or top-down with memoing, and various combinations of optimization techniques. Some allow the user control over the optimizations to be used, whilst others select optimization strategies automatically. More details are provided in the next section.

4 Prototype Deductive Database Systems

There have been a large number of prototype deductive database systems developed to date. Several of the systems implemented are memory-based. These systems assume that all the required permanent relations can be kept in main memory, and during the process of computation, any temporary relations generated can also be kept in memory. Although this method suffices for applications where the temporary relations generated are small enough to fit in main memory, for some applications this is an unreasonable expectation. When this assumption is false, these systems tend to behave poorly, and therefore techniques used in building relational database systems must be used.

Several of the implementations also assume that there is a single user of the database, and in general do not support transaction processing and crash recovery. In addition, many systems do not support essential database features such as integrity constraints and triggers. In spite of these limitations, substantial progress has been made towards demonstrating the feasibility of deductive database technology, and there are indeed some prototype systems that have been developed which do provide the expected features of a traditional database system. There are also commercial database systems under development, which have the capabilities of a deductive database.

Below we give an overview of the state of development of various prototype systems. This overview is not a complete survey of all efforts that have taken place in the development of deductive database systems. We concentrate on systems which have had significant developmental effort and have received significant attention in the literature. We refer interested readers to a forthcoming survey paper [33], which covers other issues.

RDL/C RDL/C is a programming language developed to integrate a rule-based language and the programming language C. RDL/C is derived from RDL1 [28]. This language has support for rules and abstract data types [15], and therefore the user can program at a higher level than is possible using the combination of SQL and C. In particular, the user does not have to manage temporary relations, which is done by the system. Programs written in RDL/C are compiled into an embedded database query language. This approach has the advantage of being simple to integrate into an object-oriented database system or a relational database system to provide a powerful and flexible database system. In many respects, this system is similar to LOLA (see below).

MegaLog MegaLog was developed at the European Computer Research Centre (ECRC) [21, 8]. This system is designed to provide support for manipulating large amounts of data while also providing standard Prolog features. One of the main contributions of this development is the support of a multi-dimensional grid file system called Balanced And Nested Grid file (BANG). Another important feature is its support for garbage collection and excellent facilities for dictionary management. As the behaviour of the system is similar to Prolog, the system does not guarantee termination even for Datalog programs. However, the system has proved to be a good development platform for data-intensive knowledge bases, such as the EKS system described below.

EKS The ECRC Knowledge base System (EKS) was developed at ECRC during the period 1989 to 1991 [45]. Like several deductive database systems, one of the goals of this project is to demonstrate the viability of deductive database technology for real-world applications. EKS is built on the MegaLog Prolog platform [21, 8]. The language of EKS is Datalog (and hence does not support function symbols). The main features of the EKS system include support for a very general form of integrity constraints, which may include references to recursive predicates and aggregate operations, rules which may contain recursion through aggregates, support for materialized views, and support for hypothetical query facilities. In this system, support for procedural definitions and updates is provided by the underlying MegaLog platform. The initial system was a single-user system. The computational model used in this system is derived from Query/SubQuery evaluation, a set-oriented top-down evaluation scheme with memoing [41, 44, 26]. One of the main advantages of this approach is that negation is handled in a top-down setting. This makes negation simpler to implement than bottom-up methods using the magic set transformation.

LDL The LDL system was developed at the Micro Computer Corporation [31]. One of the main features of this system is support for sets in the language. This system was built based on the bottom-up computation model, and uses several optimization techniques, such as magic sets. This system is a single user system, and all relations are memory-resident. The deductive part of this system is memory-resident. Later versions of the LDL system allow the interfacing of the system to traditional relational systems, thus providing traditional database features such as transactions. A second-generation version of LDL, known as LDL++ has been re-implemented [48]. Its main enhancements are the provision of interfaces to procedures written in C or C++, as well as the addition of abstract data types to the language.

LOLA The LOLA system was developed at the Technical University of Munich [14]. The system is implemented by compiling Horn clauses, which may contain lists, into a Relational Lisp program with embedded SQL statements. The system does optimizations to minimise the calls to the underlying SQL database system. The system's support for multiple users and transactions is mainly derived from the underlying system. This implementation approach is very similar to that of Declare & SDS, described below (and in an article in this Issue). The deductive part of this system is memory-resident, and hence this system is not scaleable to large databases when the intermediate relations are large.

CORAL The CORAL system was developed at the University of Wisconsin at Madison (see article in this Issue). CORAL uses bottom-up evaluation, with a wide variety of optimization strategies, which are specified by the programmer. One of the main features of CORAL is support for non-ground terms. The system is a single-user system and memory-resident. However the system can be connected to the EXODUS storage manager for access to permanent relations. It is not clear whether this kind of integration will scale up to large databases in performance terms.

Glue-Nail The Glue-Nail system was developed at Stanford University (see article in this Issue). An important feature of this system is the provision of two languages: one (Nail) for purely declarative statements based on Horn clauses, and another (Glue), which is procedural and used for I/O, updates and control constructs. The system also supports a form of higher-order syntax for the management of relation names. The system is a single-user system and memory-resident.

Aditi The Aditi system was developed at the University of Melbourne (see article in this Issue). Aditi uses a bottom-up approach using relational technology. In this system, both permanent and temporary relations can be disk-resident, and hence the system is scaleable to large databases. The system supports function symbols, negation, and aggregates (including recursively defined aggregates). The architecture of this system is based on the client-server model, and supports parallel query processing. The system is also a multi-user system. Another important feature of Aditi is that bottom-up and top-down computations can be interleaved. The user can declare that a particular predicate is to be evaluated in a top-down fashion. Aditi then makes a call to a Prolog system to execute such predicates. This mixing of top-down and bottom-up computation can improve performance by several orders of magnitude. However, for such predicates it is the responsibility of the user to ensure termination.

Declare & SDS The Declare & SDS project is one of the earliest deductive database projects to build a commercial deductive database system (see article in this Issue). This system has a lot of similarities to the LOLA system, although the Declare & SDS system is further developed. The language of this system is based on Horn clauses and supports lists, but with rules defining the same head predicate grouped together to form a virtual relation. The system is implemented using Relational Lisp, and is built on top of an extended version of the TransBase system. The system also provides support for types, as well as for distributed databases, and facilities for transactions.

XSB The XSB system [46, 47] was developed at Stony-Brook University. In many respects, this system has similar goals to the CORAL system in supporting non-ground terms and negation. However, the main distinction is the model of computation used in XSB, which is based on OLDT resolution, a top-down method with memoing [39, 12]. In this respect the XSB system resembles the EKS system. Like CORAL, this system is a single-user, memory-based system.

Starburst The Starburst system [18] was developed at IBM Almaden. This is a substantial project, with the main goal being extensibility of the database system, and with some interest in deductive capabilities. The system supports a restricted but useful class of recursive rules. Due to this restriction, the system is able to use efficient specialized algorithms for query evaluation. The usefulness of the magic set transformation for non-recursive programs is demonstrated in this system [30].

Commercial Systems In addition to Declare & SDS discussed above, there is also a commercial system currently under development at Groupe Bull. We believe that its main features include support for object-oriented features combined with the deductive facilities of EKS.

Some other interesting systems, such as ConceptBase [22], COL [1], LogicBase, Hy+ [10] and X4 [29], are discussed in the survey paper [33].

5 Conclusion

Deductive database technology has now reached a level of maturity that the commercial development of deductive database systems is feasible. There are several substantial prototype deductive database systems currently available from universities and research institutions, and so it is now possible to build real applications using this technology. These prototype systems have already demonstrated the potential for deductive database systems to perform as efficiently as relational systems (for those applications where relational systems are appropriate), and in addition deductive database systems provide significantly more expressive power, both for querying the database and modelling of data.

However, before deductive database technology is generally accepted in the database community, these systems will need to have the standard database facilities for transaction processing, crash recovery, multi-user access, integrity constraints, triggers and distributed database access. Unfortunately, several of the prototype systems do not have these facilities. We believe that systems such as Aditi and Declare & SDS are closer to this goal than many others. It is also encouraging to see that there are some commercial deductive database systems under development, which will include these standard database features.

References

[1] S. Abiteboul and S. Grumbach, A Rule-Based Language with Functions and Sets, *ACM Transactions on Database Systems*:16:1:1-30, 1991.

[2] K.R. Apt, H. Blair, and A. Walker, Towards a Theory of Declarative Knowledge, in *Foundations of Deductive Databases and Logic Programming* 89-144, J. Minker (ed.), Morgan Kaufmann, Los Altos, 1988.

[3] I. Balbin, D. Kemp. K. Meenakshi and K. Ramamohanarao, Propagating Constraints in Recursive Deductive Databases, *Proceedings of the North American Conference on Logic Programming* 16-20, Cleveland, 1989.

[4] I. Balbin, G. Port, K. Ramamohanarao and K. Meenakshi, Efficient Bottom-up Computation of Queries on Stratified Databases, *Journal of Logic Programming* 11:295-345, 1991,

[5] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, Magic Sets and Other Strange Ways to Implement Logic Programs, *Proceedings of the ACM Symposium on Principles of Database Systems* 1-15, Cambridge, 1986.

[6] F. Bancilhon and R. Ramakrishnan, Performance Evaluation of Data Intensive Logic Programs, in *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufmann, 1988.

[7] C. Beeri and R. Ramakrishnan, On the Power of Magic, *Proceedings of the ACM Symposium on Principles of Database Systems* 269-283, San Diego, 1987.

[8] J. Bocca, MegaLog — A Platform for Developing Knowledge Base Management Systems, *Proceedings of the Second International Symposium on Database Systems for Advanced Applications*, Tokyo, 1991.

[9] A. Chandra and D. Harel, Horn Clause Queries and Generalizations, *Journal of Logic Programming* 2:1:1-15, April, 1985.

[10] M. Consens and A. Mendelzon, Hy: A Hygraph-Based Query and Visualization System, *Proceedings of the ACM SIGMOD Annual Conference on Management of Data* 511-516, Washington DC, 1993.

[11] K. Clark, Negation as Failure, in *Logic and Databases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 1978.

[12] S. Dietrich, Extension tables: Memo relations in logic programming, *Proceedings of the Symposium on Logic Programming* 264-272, San Francisco, 1987.

[13] M. Freeston, Advances in the Design of the BANG File, *Proceedings of the Third International Conference on Foundations of Data Organization and Algorithms*, Paris, 1989.

[14] B. Freitag, H, Schütz, and G. Specht, LOLA — A Logic Language for Deductive Databases and its Implementation, *Proceedings of the Second International Symposium for Advanced Applications* 216-225, Tokyo, 1991.

[15] G. Gardarin, J. Cheiney, G. Kiernan, D. Pastre and H. Stora, Managing Complex Objects in an Extensible Relational DBMS, *Proceedings of the Fifteenth International Conference on Very Large Databases*, Amsterdam, 1989.

[16] A. van Gelder, K. Ross and J. Schlipf, The Well-Founded Semantics for General Logic Programs, *Journal of the ACM*:38:3:620-50, 1991.

[17] M. Gelfond and V. Lifschitz, The Stable Model Semantics for Logic Programming, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, 1988.

[18] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey and E. Shekita, Starburst Mid-Flight: As the Dust Clears, *IEEE Transactions on Knowledge and Data Engineering* 143-160, March, 1990.

[19] J. Harland and K. Ramamohanarao, Constraints for Query Optimisations in Deductive Databases, *Proceedings of the Second Far East Workshop on Future Database Systems*, Kyoto, April, 1992.

[20] J. Harland and K. Ramamohanarao, Constraint Propagation for Linear Recursive Rules, *Proceedings of the International Conference on Logic Programming* 683-699, Budapest, June, 1993.

[21] T. Horsfield, J. Bocca and M. Dahmen, MegaLog User Guide, Technical Report, ECRC, Munich, 1989.

[22] M. Jeusfeld and M. Staudt, *Query Optimisation in Deductive Object Bases*, in G. Vossen, J. Freytag and D. Maier (eds.), *Query Processing for Advanced Database Applications*, Morgan-Kaufmann, Los Altos, 1993.

[23] D. Kemp, K. Ramamohanarao and Z. Somogyi, Right-, Left-, and Multi-linear Rule Transformations that Maintain Context Information, *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, 380-391, Brisbane, August 1990.

[24] D. Kemp, D. Srivastava, and P. Stuckey, Query Restricted Bottom-up Evaluation of Normal Programs, *Proceedings of the Joint International Conference and Symposium on Logic Programming* 288-302, Washington DC, November, 1992.

[25] D. Kemp and P. Stuckey, Analysis Based Constraint Query Optimization, *Proceedings of the International Conference on Logic Programming* 666-682, Budapest, June 1993.

[26] A. LeFebvre and L. Vieille, On Deductive Query Evaluation in the Dedgin* System, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases* 95-112, Kyoto, 1989.

[27] J. Lloyd, *Foundations of Logic Programming* (2nd edition), Springer-Verlag, Berlin, 1987.

[28] C. de Maindreville and E. Simon, A Production Rule Based Approach to Deductive Databases, *Proceedings of the Fourth IEEE International Conference on Data Engineering* 234-241, Los Angeles, 1988.

[29] G. Moerkotte and P. Lockemann, Reactive Consistency Control in Deductive Databases, *ACM Transactions on Database Systems*:16:4:670-702, 1991.

[30] I. Mumick and H. Pirahesh, Implementation of Magic-sets in Starburst, to appear in the *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, May, 1994.

[31] S. Naqvi and S. Tsur, *A Logical Language for Data and Knowledge Bases*, Computer Science Press, New York, 1989.

[32] T. Przymusinski, On the Declarative Semantics of Stratified Deductive Databases, in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming* 193-216, Morgan-Kaufmann, Los Altos, 1988.

[33] R. Ramakrishnan and J.Ullman, A Survey of Research on Deductive Database Systems, to appear in the *Journal of Logic Programming*.

[34] K. Ramamohanarao, J. Shepherd, I. Balbin G. Port, L. Naish, J. Thom, J. Zobel, and P. Dart, The NU-Prolog deductive database system, *IEEE Data Engineering*:10:4:10-19, December, 1987.

[35] K. Ross, Modular Stratification and Magic Sets for Datalog Programs with Negation, *Proceedings of the ACM Symposium on Principles of Database Systems* 161-171, Nashville, 1990.

[36] D. Sacca and C. Zaniolo, The Generalized Counting Method for Recursive Queries, *Proceedings of the First International Conference on Database Theory* 31-53, Rome, 1986.

[37] D. Sacca and C. Zaniolo, Magic Counting Methods, *Proceedings of the ACM SIGMOD International Conference on Management of Data* 49-59, San Francisco, 1987.

[38] Y. Sagiv, Is There Anything Better than Magic?, *Proceedings of the North American Conference on Logic Programming* 235-254, Austin, October, 1990.

[39] H. Tamaki and T. Sato, OLD Resolution with Tabulation, *Proceedings of the Third International Conference on Logic Programming* 84-96, London, 1986. Published as Lecture Notes in Computer Science 225, Springer-Verlag.

[40] S.-A. Tärnlund, Horn Clause Computability, *BIT* 17:215-226, 1977.

[41] L. Vieille, Recursive Axioms in Deductive Databases: The query-subquery approach, *Proceedings of the First International Conference on Expert Database Systems* 179-193, Charlston, SC, 1986.

[42] L. Vieille, Database Complete Proof Procedures based on SLD-resolution, *Proceedings of the Fourth International Conference on Logic Programming* 74-103, Melbourne, 1987.

[43] L. Vieille, From QSQ towards QoSaQ: Global Optimization of Recursive Queries, *Proceedings of the Second International Conference on Expert Database Systems* 421-434, Tysons Corner, 1988.

[44] L. Vieille, Recursive Query Processing: the Power of Logic, *Theoretical Computer Science*:69:1, December, 1989.

[45] L. Vieille, P. Bayer, V. Küchenhoff, A. LeFebvre, EKS-V1, A Short Overview, *AAAI-90 Workshop on Knowledge Base Management Systems*, 1990.

[46] D. Warren, The XWAML A Machine that Integrates Prolog and Deductive Database Query Evaluation, Technical Report 89/25, Department of Computer Science, SUNY at Stony Brook, October, 1989.

[47] D. Warren, Memoing for Logic Programs, *Communications of the ACM* 35:3:93-111, March, 1992.

[48] C. Zaniolo, N. Arni and K. Ong, Negation and Aggregates in Recursive Rules: the LDL++ Approach, *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, Phoenix, 1993.

[49] Justin Zobel and Kotagiri Ramamohanarao, Accessing existing databases from Prolog Technical Report 86/17, Department of Computer Science, University of Melbourne.