A Data Base Extension of Prolog and its Implementation

Tore Risch
18 August 1982

UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computer Science, University of Uppsala
750 02 Uppsala, Sweden

ABSTRACT

A method is presented to extend the programming language Prolog with a capability to access a relational data base system. Since the data type 'relation' is basic in Prolog, and since Prolog is based on interpretation of a subset of predicate calculus, such an interface becomes simpler in Prolog than in most other languages, and the same language, extended Prolog, may be used both for programming and as a very powerful query language.

From the Prolog programmers' point of view, the approach has the advantages that conventional data bases can be concurrently accessed from Prolog programs, and that the sizes of certain types of Prolog relations are not limited by the primary memory size.

A portable pilot implementation has been done, using a Prolog interpreter coded in Lisp, a portable Lisp interpreter, and a portable relational data base system. The implementation and its primitives will be overviewed. The approach presented takes advantage of optimization methods of the underlying relational data base system.

CHAPTER 1

INTRODUCTION

Relational data base systems have become very popular during the last decade. Overviews of relational data base systems can be found in, e.g., <Kim>. Relational data bases allow the programmer to see his/her data as a set of tables, or relations. For manipulating these relations, there is normally a query (and update) language available. The query language can either be used interactively for data base manipulations from a terminal, or imbedded in some programming language. The query language is often non-procedural, i.e. the programmer declares to the system what he/she wants to do with the data base, and the query language handler decides on its own how the task is accomplished. For example, access path optimizations are of-ten done by the query language handler. By using a relational query language when developing data base application programs, one may separate the physical representation of the data from the way data is seen from the application program, which simplifies the programming and makes programs more independent of data base updates.

Since a query language can be used only for specifying data base manipulations, it needs to be connected to a general purpose language if it is to be used for development of application programs.

Therefore the query language is often imbedded into a general purpose programming language, e.g. PL/I, C, or COBOL, by specially marking query language statements, and using a pre-processor, which translates query language statements into calls to data base access routines. This method is, e.g., used in System R <Astrahan> and Ingres <Stonebraker>.

The advantage with this method is that the application programs can easily be integrated with other software in the general purpose language. However, a problem is that the programming environment of the query language is different from that of the general purpose language. For example, the data type 'relation' is not available in most general purpose languages, the error handling may be difficult to integrate, as well as data type handling. One will have a two-level language with features not easily integrated, and where the general purpose language has quite a different philosophy than the general purpose language. For example, the general purpose language is procedural (or imperative) while the query language is non-procedural.

Of course, it one makes a completely new language, these difficulties

may be avoided.

This paper will show that Prolog is a very good host language for a relational query language, because its philosophy is very similar to that of query languages. The data type relation and its handling is very basic in Prolog. Therefore an extension of Prolog can be directly used as a relational query language, thus giving a one level language for data base applications, i.e. extended Prolog can be used both for data base manipulations and for application programming.

Furthermore, Prolog as a query language is in some caes more powerful than conventional query languages. There are some practical problems which are not easily programmable using conventional query languages.

Another advantage is that Prolog adds deductive capabilities to the query language, i.e. one may represent and use implicit facts like, e.g., 'A person is a grandfather if he has a son which in his turn has a son'. This deductive capability may be regared as a generalization of views in relational database systems.

An important reason for the work presented here is to allow for the Prolog programmer to access relational data bases. In this way Prolog programs can easily access data stored in conventional data bases, thereby sharing data with other systems. The data base system will allow Prolog programs to update these data bases concurrently with other programs.

Finally, by using a relational data base system for storing Prolog relations one may relax limitations on the size of a Prolog program because of limitations on the primary memory.

There are also some limitations on the method presented here:

First, Lisp is used for the implementation of the Prolog system and its extension, while the relational data base system is coded in FORTRAN. This implementation method is slow compared to a system entirely implemented in assembler, Fortran, or some other efficient language. However, Lisp is very useful for making experiments with Prolog implementations because of its flexibility. Efficiency is not critical for showing the ideas. Furthermore the marriage of Lisp and Prolog is very useful (see <Robinson82>, <Komorowski>, and <Carlsson>).

Secondly, Prolog relations are more general than normalized data base relations. (The word tables will hereafter be used for external data base relations). Thus, only certain types of Prolog relations may be stored as tables, the other have to be stored in the Prolog memory. As we will see, this is not a serious limitation, since it can be assumed that the relations representable as tables are much larger than other Prolog relations.

CHAPTER 2

PROLOG AS A DATA BASE LANGUAGE

In order to make the paper self contained, this sections starts with a short description of the programming language Prolog. The reader familiar with Prolog can skip the beginning of this section. The section ends with some examples of data base queries formulated in Prolog compared to the same queries formulated in a relational calculus query language, QUEL <Stonebraker>.

The implementation on which this paper is based uses a Prolog interpreter written in Lisp, YAQ <Carlsson>. YAQ has Lisp's S-notation for Prolog expressions, but for making this paper more easily readable I will use a more conventional Prolog syntax.

2.1 CONCEPTS OF PROLOG

A logic program can be defined as a set of Horn clauses, which are expressions

```
B <= A1,...,Ak. (1)
```

where B and Ai are atoms. An atom is a predicate followed by a list of arguments. Each argument is a term, i.e. either a constant, a variable, or a functional expression. All the variables appearing in a clause are assumed to be universally quantifyed. I distinguish constants from variables by capitalizing variables and using lower case letters for constants.

The declarative interpretation

A set of Horn clauses is interpreted as the conjunction of its right hand side atoms. Variables are local to each clause. The Horn clause above is interpreted as "for all values of its variables, B holds if A1 and ... and Ak". For example, the clause

```
grandmother(X,Y) <= mother(Z,Y), parent(X,Z). (2)
```

can be read "for every X,Y, and Z it holds that Y is a grandmother of X if Y is the mother of Z and Z is the parent of X".

A Horn clause with an empty right-hand side is interpreted as an unconditional assertion of fact, e.g.:

```
mother(KAIN,EVE) <=.                    (3)
```

which thus should be read as "(It is always true that) the mother of KAIN is EVE".

The computational interpretation

A Horn clause of the above form is called a procedure and is interpreted as a procedure declaration, where B is the procedure name and formal parameters, and A1,...,Ak is the set of procedure calls constituting the procedure body. Assertions can be regarded as a special case of a procedure declaration, in which the body is empty.

A Horn clause of the form

```
<= A1,_,Ak.                             (4)
```

is called a query and is interpreted as a request to find a constructive proof of "There exist X1,...,Xk such that Al and ... and Ak hold", where X1,...,Xk are the variables in the query.

A Prolog interpreter constructs such a proof through resolution <Robinson65>, a kind of a pattern matching method. During this process, the variables in the query are instantiated, i.e. substituted for calculated expressions. The substitution of variables for terms thus obtained is called an answer to the query. Alternative proofs may result in alternative answers or the same query.

```
manager2(X,Y) <= manager(X,Y), manager(Z,Y).
manager(olle,kalle)<=.
manager(eve,elsa)<=.
manager(kalle,ludwig:)<=.
manager(elsa,ludwig)<=.
```

Fig. 1: Prolog examples

For example, given the Prolog program of figure 1 and the query

```
<=manager2(X,ludwig).                   (5)
```

a Prolog interpreter would find the two answers X=olle and X=eve.

A set of procedure declarations whose left hand sides have the same predicate predicate symbol, P, is called a predicate definition for P.

2.2 PROLOG AS A DATA BASE LANGUAGE

Throughout this section I will use as an illustration the relational data base table of figure 2, which describes a number of employees, their managers and their salaries. Top level managers have empty MANAGER columns.

EMPLOYEE	NAME	MANAGER	SALARY
	JONES	JAMES	28000
	JAMES		54000
	ADAM	JONES	30000
	CARL	JONES	25000
	EVE	CARL	26000
	ELIZA	CARL	27000

Fig. 2: A sample relational database table

A Prolog program can be regarded as a data base consisting of two parts, the extensional data base (EDB), and the intentional data base (IDB).

The extensional data base is represented by Horn clauses without right hand sides with only constants in their left hand sides. An example of an EDB is the clauses in figure 3 which represent the data base of figure 2. (Empty fields are represented with 'nil').

```
employee(jones,james,28000)<=.
employee(james,nil,54000)<=.
employee(adam,jones,30000)<=.
employee(carl,Jones,25000)<=.
employee(eve,carl,260001)<=.
employee(eliza,carl,27000)<=.
```

Fig. 3: A relational data base represented in Prolog.

We can make the observation that Horn clauses directly correspond to a relational table. The difference is that Prolog allows structured objects among the atoms, and that the assertions are of different lengths. For example, the clauses

```
r(f(hans),[a,b,c])<=.
r(eve)                                    (6)
```

is a legal EDB in Prolog, but it cannot be directly represented in a normalized relational data base table. ([a,b,c] denotes a list of three elements a, b, and c).

The IDB is a data base containing facts which are inferable by using the EDB and a number of predicate definitions. The MANAGER and MANAGER2 relations of figure 4 are examples of Prolog predicates in the IDB using the EDB of figure 3. (The underline (_) denotes an anonymous, unique

variable name). One may say that Prolog predicates with variables specifies rules how to construct one relation from other relations, which are either in the EDB or IDB. We can here observe that IDB-relations correspond to views in relational data bases, with the difference that a predicate definition can represent more powerful rules than relational data base views, e.g. with recursion.

```
manager(OF,IS)<=employee(OF,IS,_).
manager2(OF,IS)<=manager(OF,X),manager(X,IS).
```

Fig. 4: An example of an IDB

Another observation is that Prolog queries, i.e. Prolog statements without left hand sides, correspond to relational query language statements. For example, figure 5 shows some relational calculus queries in QUEL on the table of figure 2, and their corresponding Prolog queries over the data base of figure 3.

Find all employees of CARL:

```
QUEL:   RANGE OF E IS employee;
        RETRIEVE E.NAME WHERE E.manager = CARL;

Prolog: <=employee(RESULT,carl,_).
```

Find the second level manager of ELIZA:

```
QUEL:   RANGE OF Ej IS employee;
        RETRIEVE F.manager WHERE F.NAME = E.manager AND
                                 E.NAME = CARL;

Prolog: <=employee(carl,X,_),employee(X,RESULT,_).
          or
        <=employee(carl,X,_),employee(Y,RESULT,_),eq(X,Y).
```

Fig. 5: Some relational calculus queries and their corresponding Prolog queries.

Figure 5 illustrates how any relational data base query can be expressed as a Prolog query. Notice that Prolog queries refer to columns by positions, while relational data base queries use column names. In commercial data bases this may be a disadvantage, since the tables there normally have a large number of columns.

The second example of figure 5 illustrates an equi join in Prolog. It can either be done by using common variables in procedure calls, or by using the Prolog predicate for equality. The latter method corresponds more closely to the relational data base query, while the former method is more convenient. We also notice that Prolog queries are often more compact than relational data base queries.

There is a well known problem with relational data base queries known as

the 'bill of material problem', see <Chamberlin>. In our example it is analogous the problem of finding all lower level employees of some manager. In Prolog this query is expressed as in figure 6, while it cannot be expressed in most relational query languages. Therefore, to solve this problem, one has to use the high level language in which the relational query sublanguage is imbedded. For example, <Chamberlin> shows how to solve the problem in SQL, illustrating that the code becomes rather complicated.

```
manager(OF,IS)<=employee(OF,IS,_).
descendants(OF,IS)<=manager(OF,IS).
descendants(OF,IS)<=manager(OF,X),descendants(X,IS).
<=deseendants(carl,RESULT).
```

Fig 6: Retrieving all lower level employees of CARL.

In order to solve the query in Prolog one uses recursive relations, which would correspond to recursive views in a relational data base language, a normally missing feature. Notice that the recursive relation (view) descendants also can be used for retrieving all higher level managers of, say, 'eliza' with the query

```
<=deseendants(RESULT,eliza).                    (7)
```

In summary one may conclude that Prolog may be used as a powerful and compact query language, even though it may be inconvenient when there are many formal parameters (=columns) in the relations.

2.3 EVALUATION ORDERS IN PROLOG AND QUERY LANGUAGES

The relational data base query languages are normally non-procedural, which in practice means that a query language optimizer decides how the relations involved in a query shall be accessed. Prolog, on the other hand, has a defined evaluation order, by using unification and backtracking. Prolog will scan the first relation in a join, and for each tuple retrieved it will scan the second relation in the join for matching tuples. For efficiency Prolog often has built-in indexes on the first argument of large relations. A similar method is often used when joining relational tables as well. <Blasgen> calls it "Indexes on join columns" and <Arnborg> calls it back-track-programming (!). One difference is that relational data base query languages normally optimize the order of the nested loops. Both Prolog and relational data bases use indexes to quickly access a relation when a column (variable) value is known, even though Prolog normally only have indexes on the first variable (column). We should also notice that relational data base optimizers use other access methods as well, e.g. by sorting intermediate results <Blasgen,Arnborg>.

The disadvantage of not optimizing queries is more serious in relational data bases than in Prolog programs, since commercial data bases may be very large, and since real life applications require very fast access times. Prolog relations, on the other hand, are normally of limited size and are stored in the primary memory, which makes the access time less critical. Some Prolog dialects (e.g. <Clark>) allow relations to be externally stored.

The way in which views are handled is quite different in Prolog than in relational data bases. In relational data bases views are handled by modifying queries, which is equivalent to expand the view definitions. The expanded queries are then optimized as usual.

Prolog, on the other hand, uses unification and backtracking to 'invoke' the views, with no optimization.

Relational query languages do not allow recursive views, since expansion of recursive views would result in unlimited code. On the other hand, the Prolog method may result in very inefficient code, e.g. in the query descendants(carl,X), since the relations will be accessed in every recursive call.

In summary one may conclude that relational data base languages optimizes the table accesses while Prolog does not, and that there are special problems with optimizing accesses to recursive views. We also notice that the optimization problem is less critical in Prolog programs than in relational data bases.

Later I will show how the optimization problem is handled in my implementation.

CHAPTER 3

INTERFACING A RELATIONAL DATA BASE SYSTEM WITH PROLOG

This section starts with a historical review of the projects leading to this work (see 3.1). Section 3.2 is an overview of the primitives added to YAQ for the handling of tables. Section 3.3 gives some motivations for the implementation, and section 3.4 describes some implementation details.

3.1 HISTORY

The implementation on which this paper is based relates to several other research projects at the department of computer science in Uppsala and Uppsala university data center.

The author's dissertation work, Lidam, <Risch78,Risch80a> was based on the construction of a DBMS independent data dictionary system, having a program generator which was able to generate general purpose language code from specifications of data base retrievals in a non-procedural relational query language.

In parallel with the development of Lidam, Uppsala university data center was developing a relational data base system, Mimer <Mimer>. Mimer is largely machine independent, since it is written in Fortran with a few machine dependent routines in assembler.

Another system developed at the department of computer science at Uppsala University is a Lisp interpreter, Lispf3 <Nordstrom>, written in standard Fortran with only two small machine dependent routines.

Since both Mimer and Lispf3 are written in standard Fortran, it was a rather simple task to connect the two, making an interface between Lisp and the relational data base system Mimer. This Interface and parts of Lidam was used for making a combined query language handler and program generator for Mimer <Risch80b>.

A Prolog interpreter, YAQ <Carlsson> has been implemented to run under Lispf3. YAQ uses a lispish S-notation for Prolog expressions.

YAQ, Lispf3, Mimer, and parts of Midam was used to construct an interface between Prolog (i.e. YAQ) and a relational data base system

(i.e. Mimer). The system is thus largely machine independent as a large Fortran program, but still Lisp is used for the implementation of Prolog and parts of the relational data base interface.

The remainder of this report will describe some design principles and experiences made during the implementation.

3.2 PROLOG PRIMITIVES FOR THE RELATIONAL DATA BASE INTERFACE

Three types of primitives have been added to Prolog, namely

First, some data dictionary primitives, which are needed for creating and removing data base tables.

Second, a few primitives are used for declaring Prolog predicates as tables.

Third, Prolog needs some extensions for accessing the data base.

The data dictionary primitives include Prolog predicates for creating, formatting, and removing relational data base tables, predicates for retrieving the names of tables and columns, predicates for adding and deleting secondary indexes, etc.

The main primitive for declaring external Prolog predicates is

```
external(P,DB,TABLE)                    (8)
```

where P is the name of the Prolog predicate, and TABLE is the name of the table in the data base DB. The system requires the data base to exist when external is called, so that it can look up the data dictionary at declaration time for correct connection of the Prolog predicate to the external table.

Once P has been declared external all assertions of P will be done to the external table, and all accesses to P will be taken from the table. For example, if one asserts

```
f(adam,olle)<=.
f(kalle,olle)<=.                        (9)
```

two tuples will be inserted to the data base. If one later gives the Prolog query

```
<=f(X,olle).                            (10)
```

The Prolog system will access the external data base, yielding the answers

```
    X=adam    and
    X=kalle                                          (11)
```

If one wants to get all records in the external table, one may write

```
    <=f(X,Y).                                        (12)
```

and the system will write

```
    X=adam,  Y=olle  and
    X=kalle, Y=olle                                  (13)
```

Real life data base tables often have a large number of columns, e.g. one often store one line for each person, and each line contains several properties of the person. Therefore it is practical to use a feature of YAQ to allow a variable number of arguments, when listing an entire relation For example

```
    <=f L.                                           (14)
```

will print the answers

```
    L=[adam,olle] and
    L=[kalle,olle]
```

This section has shown how a simple connection of Prolog predicates to external tables may be implemented by basically the external declaration. In the next section I will show some disadvantages with the solution and outline a better implementation, as done in my implementation.

3.3 IMPROVING THE INTERFACE

The simple interface can be used for querying external tables by calling an external predicate and filtering the result with Prolog predicates. For example, suppose that we have the external table employee in figure 2, and have done the external declaration

```
    external(e,db1,employee)                         (15)
```

The query

Give me all persons who earn more than 20000.

can be expressed in Prolog as

```
     <=e(X,...,S), gt(S,20000).                      (16)
```

However, this query will be very inefficiently executed by Prolog. What

will happen is that the predicate e(X,_,S) will force a scan of the entire employee relation. For each new value of S the system will apply the gt(S,20000) test, and, if the test succeeds, the value of X will be printed. For every 'gt' call the system will backtrack for the new tuple from employee. The 'gt' predicate is furthermore called using unification. A much more efficient method would be to let the data base handling system do the filtering of the 'gt' predicate, and only return the tuples satisfying the 'gt' predicate. Relational data base handling systems normally have primitives for filtering basic predicates, as 'gt', which are more efficient than Prolog's unification.

Furthermore, a relational data base system may use (secondary) indexes for efficient data base access of certain filters. For example, Mimer uses B-trees for efficient handling of equality and interval queries.

So one improvement could be to allow for the data base system to do some access optimization and filtering, instead of letting Prolog do this with unification. The approach I have used in my implementation is to introduce higher level primitives to Prolog than the external declaration. These primitives take filter predicates on external tables as arguments. This means that Prolog programs accessing external data bases will differ slightly from equivalent programs accessing in-core Prolog data bases. However, programs for automatic transformations may be constructed.

The basic primitive for accessing external tables is named SELECT, and it is inspired by the SELECT statement in SQL <Astrahan>. In terms of SELECT, the above query would be written as:

```
<=select(e(name),gt(salary,32000),X).          (18)
```

The general format of the select predicate is

```
select(EXT(COLS),FILTER,Vl,...,Vk)             (19)
```

where EXT is the name of an externally declared predicate definition, COLS is a list of column names to be retrieved (projected). This list must have the same length as the number of variables, V1,_,Vk, to be bound at each call to select (k=l in the example above). FILTER is a selection rule on the table. It may contain only predicates supported by the data base system, as well as AND and OR. 'select' will retrieve values of the columns in COLS, and, for each tuple retrieved, the corresponding variable, Vi, will be bound. So, if one wants to retrieve the persons who earn more than 32000 and there fathers, one may write as in figure 6.

```
<=select(e(name,father)gt(salary,29000),X,Y).
and the system will print
X=james, Y=nil
X=adam, Y=Jones
```

Fig. 6: An example of how to use the predicate 'select'.

A limitation for the moment is that a select query may only range over one external table. However, this limitation will be removed in later versions of the system.

The system can use optimization primitives of Mimer to optimize select with respect to data base accesses and access path selection. 'select' expressions may be compiled into efficient specialized access code, by using techniques for relational database query compilation.

Note that a program transformator can be used to transform automatically the program (17) into (18)

3.4 OTHER PRIMITIVES

If a table is declared external, retract will remove tuples from it in exactly the same way as if the relation would be in-core, i.e. records are deleted from the external table. For example

```
<=retract(e(olle,X,_)).                    (20)
```

will delete all records in employee where the column name has the value 'olle'. At the same time X is bound to the father of 'olle'. This could be viewed as 'the last read'.

The declaration external is actually a special case of the declaration of external views, with the format of figure 7.

```
extview(NAME,EXT(COLS),FILTER,V1,...,Vk) for example
<=extview(rich,e(name),gt(salary,30000),X).
```

Fig. 7: The format of the built-in predicate 'extview'

extview will declare 'rich' to be an external relation containing the column 'name' of employee for lines where the column 'salary'>30000. 'rich' is an example of an external view. In difference to views in relational query languages, as SQL, I only allow external views to refer to exactly one external table each, and thus they define an external relation as a subset (horizontal and vertical) of the external table. The reason for this limitation is that is it shown that relational views may be updated without complications only if they refer to exactly one table, as in my case. In other words, since I have this limitation, 'assert' and 'retract' may be used freely on external views. If a tuple is inserted to an external view with only some of the columns of a table, the other columns of the table will have null values.

3.5 INDEX SELECTION FOR CONSTANTS

'select' solves some optimization problems for many programs. There is also another situation when index optimization is very important, namely when a constant occurs in a relation call, for example

```
<=e(olle,X,_).                                    (21)
```

With only the primitives above available, also this query would scan the entire employee table and unify each tuple retrieved to see if the first element is 'olle'. The query may be alternatively expressed as

```
<=select(e(father),eq(name,olle),X).              (22)
```

in which case the access optimization may be used efficiently. It is always possible to use select instead of the more simple method. To simplify this type of queries my implementation uses dynamic compilation of calls to external relations involving constants. So the query above is dynamically transformed into the select call and then executed. Such dynamic transformations will occur at any time the system internally generates calls to external relations with constants.

3.6 IMPLEMENTATION OVERVIEW

I only give a brief overview of the implementation. YAQ was extended with the previously described external data base primitives. The backtracking algorithm of YAQ was modified so that it cheeks whether a relation is external which is specially marked in its definition. If so, the backtracking mechanism will open the external table when the relation is entered. The system maintains a file cursor which is moved forward each time YAQ backtracks on an external relation. When end of file is reached the external table is closed. Several file cursors can be opened for the same external table. To allow for tail recursion optimization <Warren80>, the system actually looks ahead one line in the table. The lines are delivered to YAQ in the same internal format as is suitable for unification in YAQ. Thus the remainder of YAQ cannot notice any difference between accessing external or in-core relations.

CHAPTER 4

SUMMARY

I have shown some methods to extend Prolog with primitives for manipulation of external relational data bases.

For making external and normalized relational data bases useful, it is assumed that the extensional data base, EDB, ('records') is much larger than the intentional data base, IDB, (‘views'), so that conventional Prolog methods can be used for accessing the IDB, while a relational data base system handler can be used for accessing the EDB.

The approach assumes that the underlying relational data base handler can optimize data base queries, by choosing proper access paths.

I have made a pilot implementation showing that the approach works. The pilot implementation is made in Lisp and Fortran, it is machine independent, and it uses a commercial relational data base system, Mimer.

One important conclusion is that it is better to extend Prolog with some rather high level primitives for querying external data bases, than directly using Prolog for querying. Transformation programs may be constructed to transform standard Prolog programs into programs with calls to these high level constructs.

The basic primitive in the system is a feature to declare predicate definitions to be external, and a view definition capability. The Prolog predicates 'assert' and 'retract' are made transparent to predicates declared external.

CHAPTER 5

REFERENCES

<Astrahan> M.M.Astrahan et. al.: "System R: A Relational Approach to Database Systems", ACM Transactions on Database Systems, June 1976.

<Arnborg> S.Arnborg, P.Svensson: "Fast Multivariable Query Evaluation", FOA Report C20189-D8, Swedish National Defence Institute, Stockholm, 1977.

<Blasgen> M.W.Blasgen, K.P.Eswaran: "Storage and Access in Relational Databases", IBM Systems Journal, Vol. 16, No 4, 1977.

<Carlsson> M.Carlsson: "(Re)implementing Prolog in Lisp or YAQ ~ Yet Another QLOG", UPMAIL Technical Report NO 5, UPMAIL, Uppsala Univ. Comp. Science Dept., Box 2059, S-750 02 Uppsala, Sweden, 1981.

<Chamberlin> D.D.Chamberlin: "A Summary of user Experiences with the SQL Data Sublanguage" Proc. Int. Conf. on Data Bases, Univ. of Aberdeen, Scotland, July 1980.

<Chang> "DEDUCE 2: Further Investigations of Deduction in Relational Data Bases", in: H.Gallaire, J.Minker (eds.): "Logic and Data Bases", ISBN 0-306-40060~X Plenum Press, New York and London, 1978.

<Dahl> V.Dahl: "On Database Systems Development Through Logic", ACM Transactions on Database Systems, Vol. 7, No. 1, March 1982.

<Clark> K.L.Clark, F.G.McCabe, S.Gregory: "IC-Prolog Language Features", in K.L.Clark & S.-A. Tarnlund: "Logic Programming", Academic Press, ISBN 0-12-175520-7, 1982.

<Kim> Won Kim: "Relational Database Systems", Computing Surveys, Vol. 11, No. 3, Sept. 1979.

<Komorowski> H.J.Komorowski: "QLOG - The Software for Logic Programming", in: S.-A.Tarnlund (ed.): Proc. Logic Programming Workshop, Budapest, July 14-16, 1980.

<Minker> J.Minker: "An Experimental Relational Data Base System Based on Logic", in: H.Gallaire, J.Minker (eds.): "Logic and Data Bases", ISBN O-306-40060-X Plenum Press, New York and London, 1978.

<Mimer> Mimer Reference Manual, Uppsala University Data Center, Box 2103, S-750 02 Uppsala, Sweden.

<Nordstrom> M.Nordstrom: "Lisp F3 User's Guide", Technical Report DLU 79/1, Uppsala Univ. Comp. Science Dept., UPMAIL, Box 2059, 750 02 Uppsala, Sweden, 1979.

<Risch78> T.Risch: "Compilation of Multiple File Queries in a Meta-database System", PhD Thesis, Dept. of Mathematics, Linkoping univ., 581 83 Linkoping, Sweden, 1978

<Risch80a> T.Risch: "A Flexible and external Data Dictionary System for Program Generation", Proc. Int. Conf. on Data Bases, Univ. of Aberdeen, Scotland, July 1980.

<Risch80b> T.Risch: 'Troduction Program Generation in a Flexible Data Dictionary System", Proc. Conf. on Very Large Data Bases, Montreal, 1980.

<Robinson65> J.A.Robinson: "A Machine-oriented Logic Based on the Resolution Principle", Journal of the ACM 12, 1965.

<Robinson82> J.A.Robinson, E.E.Sibert: "LOGLISP: Motivation, Design and Implementation", in K.L.Clark & S.-A. Tarnlund: "Logic Programming", Academic Press, ISBN 0-12-175520-7, 1982.

<Stonebraker> M.Stonebraker, E.Wong, P.Kreps, G.Held: "The Design and Implementation of INGRES", ACM Transactions on Database Systems, Sept. 1976.

<Warren80> D.H.D.Warren: "An Improved Prolog Implementation which Optimises Tail Recursion" in: S-A. Tarnlund (ed.): Proc. from Logic Programming Workshop, Budapest, July 14-16, 1980.

<Warren8l> D.H.D.Warren: "Efficient Processing of Interactive Relational Data Base Queries Expressed in Logic", Proc. Conf. on Very Large Databases, Cannes, France, 1981.