Storing and evaluating Horn-clause rules in a relational database

by Ghica van Emde Boas
Peter van Emde Boas

This paper describes a practical approach to storing and evaluating Horn-clause rules in a relational database system. The intention is to give a complete outline of what needs to be added to an existing relational database system to allow it to support full logic programming functions. Implementation issues for each new function are discussed. We show how Horn-clause rules can be translated into database commands without recourse to semantics and how their evaluation can be performed in the database itself. This brings the complete logic programming environment within reach of the database management system, allowing data and rule sharing, concurrency control, recovery procedures, etc., to be used. New is that the complete logic programming environment is incorporated into the database system. IBM Business System 12, extended in this way, may be a suitable vehicle for expert system applications.

Introduction

As the art of building expert systems matures, more sophisticated tools to hold their knowledge base and better means to reason with this knowledge will be required. Large-scale, multi-user expert systems will benefit from functionality which has been until now almost exclusively the domain of database systems. This includes the primary database functions, such as storing large amounts of data and optimizing relational queries. Also, secondary database functions will be necessary: *data sharing*, *data integrity*, *authorization control*, *concurrency control*, and *backup and recovery procedures*. Most of these are not visible in a single-user environment but are indispensable for proper functioning of a database in a multi-user environment.

Considerable effort is being directed towards designing a system incorporating both logic programming functions to provide the inferencing capabilities for an expert system and the traditional database functions to be used for its knowledge base. As a short-term goal, a simple interface could be made available between Prolog and an existing relational database system providing tuple-at-a-time access. Such interfaces have been built. The main problem with them is that the query and optimization facilities of the database cannot be fully exploited.

Research aimed at using a database intelligently from a foreground (Prolog) interface was conducted by Chang [1], by Reiter [2], and more recently by Maier [3], Naqvi [4], Jarke et al. [5], Ullman [6], and Vassiliou et al. [7]. Their

purpose was to find preprocessing techniques for Prolog or other logic programs and translate them into database programs involving selects, projects, and sometimes joins. This approach then led to the problem of dealing with recursive Prolog rules, which goes beyond the relational database framework. See Chang [8], Naqvi and Henschen [9], and more recently Ullman [6], and also the interpretative approach in Walker [10, 11].

Much of the work reported in these papers has goals similar to ours. However, a disadvantage of the preprocessor approach is that, as long as data and rules are kept and managed at the preprocessor side, the system must be regarded as a distributed system, requiring that secondary database facilities be implemented at both sides.

Another issue is that in Prolog the capability exists of using structured or composite objects. Incorporating these into a relational database requires extending its functionality. Lorie and Plouffe [12], Zaniolo [13], and Tsur and Zaniolo [14] did similar work. However, their extensions seem not to have been considered with the intent of supporting logic programming functions.

What we describe in this paper is a new approach to adapting a relational database so as to allow it to perform logic programming functions. Rather than building database facilities into a Prolog environment, a Prolog environment is constructed, starting with an existing database system. In this regard our architecture resembles the approach of [14], where a high-level database system is implemented around an existing system.

Our work combines many of the ideas reported in the references cited above, giving an integrated picture of the functional extensions needed to provide a database with logic programming capabilities. Moreover, we introduce a new method for handling recursion within the database.

Our approach has several advantages: Duplication of the huge effort involved in developing the secondary database functions is avoided. Since rules are also stored in the database, *rule sharing* is now possible. All data, including rules, also become subject to concurrency control and recovery procedures. And global optimization of queries is feasible, including recursive ones, taking into account knowledge which only the database itself can have, such as table sizes, usage statistics, location of data, etc. In short, the full power of the database management system becomes available in the logic programming environment.

The project described in this paper originated from the observation that the backtracking performed by a Prolog interpreter when calculating the conjunction of two or more predicates is equivalent to a rather inefficient method of computing a join of two or more relations, at least in the simple situations used for explaining this mechanism in introductory texts such as [15]. Clearly, this observation is not new. The connection between logic programming and relational algebra is made self-evident when relational databases are discussed from a purely algebraic perspective, such as the cylindric algebras described in [16]. It is precisely this connection on which the compiled approach in [1] and [2] is based. See also the survey in [17].

There is also a point which some may consider a disadvantage: Due to the architecture of most relational systems, in which the order of result rows from a query is not predefined and is very much dependent on the optimization method chosen for that query by the database system, flow of control through the use of *cut* cannot be implemented in the same way as in conventional Prolog.

As a vehicle for our extensions, we use IBM Business System 12. Assuming that this system is not well known outside Europe, we briefly describe available functions when needed. Reference to possible future enhancements of IBM Business System 12 must not be construed to mean that IBM intends to implement these enhancements. The ideas presented here are entirely the responsibility of the authors and reflect their personal opinions.

Business System 12 and the relational model

Business System 12 is a new relational database information system offered by IBM Information Network Services to time-sharing users, mainly within Europe.

The main design philosophy behind the Business System 12 implementation is that the system be suitable for a time-sharing environment. High emphasis is given to data security, data sharing control, database integrity, recovery, etc., as well as to the need to prevent users from disturbing each other's operations by, for example, locking important resources. This is necessary because customers using the same database could be competitors.

Business System 12 is a full relational database management system, as specified by Codd in [18]. This implies that in Business System 12 we deal with objects such as *relations*, *tuples*, *attributes*, and *domains*. There are 12 relational operators, which include the usual *select*, *project*, and *join*; further, we need *union* and *calculate*.

First we introduce the syntax used in our paper. Next, we describe briefly the Business System 12 relational query evaluation function, with emphasis on and restriction to what we need for our logic programming extension. A more comprehensive description of this syntax and of the relational operators can be found in [19]. More information about Business System 12 itself is available in [20]. Our syntax is different from the actual Business System 12 syntax, but it allows for a concise description. The semantics remains unaffected by this change of notation.

domain A set of values. The user can specify subsets of an underlying data type which can be *character*, *numeric*, *name*, *bit*, or *timestamp*.

attribute	*A*. A named domain.
tuple	*u*. A set of attribute values $(u1,u2,\cdots,uk)$ such that *ui* is in *Ai*. We denote a tuple by $[A1: u1,\cdots,An: un]$.
relation	*r*. A set of tuples of identical type. We denote a relation together with its set of attributes as $r\{A1,\cdots,An\}$, where $A1,\cdots,An$ are the names of the attributes. The set of attributes is denoted as $R(r)$.
common attribute	A common attribute of two relations *r* and *s* is an attribute which belongs to both $R(r)$ and $R(s)$.
Project	***PR*** (r,Y) is the restriction of the relation *r* to the attributes in *Y*, where *Y* is a subset of $R(r)$.
Select	***SL*** (r,F) is the set of tuples *u* of *r* for which *F* evaluates to *true*. The expression *F* has the attributes of *r* as arguments.
Union	The union *r* ***UN*** *s* is the set of tuples which are in *r* or *s* or both, projected to their set of common attributes.
Join	The join operator ***JN*** defined in Business System 12 is generally known as the *natural join*. Tuples from *r* are combined with tuples from *s* to form a tuple over the set of attributes $R(r) \cup R(s)$ if their values are equal for all common attributes of *r* and *s*.
Calculate	The calculate operator $\boldsymbol{CL}(r,X \leftarrow F)$ adds a new attribute *X* to *r*. For each tuple in *r* the value of *X* is obtained by evaluation of the expression *F*.
table	The instantiated form of a relation. It is stored in the database. Attributes and tuples for tables are sometimes called *columns* and *rows*.
rename	The *rename operator* in Business System 12 is actually implemented as part of *project*. $\boldsymbol{RN}(r,A1/A)$ changes the name of attribute *A* in *r* to *A1*.

Frequently we encounter expressions in which there is a combination of rename and project; some attributes are renamed while others are projected away. In these cases we use a shorter notation which is best explained by an example: Let *r* be a relation with attributes $R(r) = \{A,B,C,D\}$. Then we write $\boldsymbol{PR}(\boldsymbol{RN}(r,A1/A,C1/C), \{A1,B,C1\})$ as $r\{A1/A,B,C1/C\}$.

• *Views*

In the relational model it is evident that the result of a relational operation on relation(s) is again a relation. In database terminology this result is called a ***view***, and the expression which led to this result is called a ***relational expression*** or a ***view definition***.

Many relational DBMSs have facilities to store view definitions and allow some usage of view names as if they were table names. In Business System 12, this is also possible. View definitions are stored in data tables of a special type. This allows for easy updating or changing of view definitions, and with some additions (views can also be shared for execution only, for example), the same mechanism is used to control sharing and authorization as for ordinary data tables.

This raises an important performance question. As we will see, Horn-clause rules are stored as view definitions in Business System 12. Such rules can be very dynamic, and many different rules may exist. Therefore, it must be possible to quickly add, delete, or change view definitions without impacting other users. This can indeed be done in Business System 12, because view definitions are stored in tables, not in the catalog. Further, a global catalog does not exist; rather, there is a separate one for each user. When a user asks for access to a table or view owned by another user, the database management system looks in the catalog of the owner to determine whether the requested type of access is authorized and where the table can be found. By using historical versions, the database management system is even able to see the latest consistent data while the owner is updating his catalog through adding or deleting tables.

In some relational database systems (we refer to DB2 by way of example; see [21]), rule-to-view translation would be more difficult due to more restrictive view handling. In DB2 view definitions are not allowed to contain *union*s. But the capability of forming unions is essential for our translation of Prolog rules into view definitions. Also, in DB2 the catalog is global, and special authority is needed to define or delete new database objects, leading to the performance bottleneck mentioned above.

• *A database example*

The example in **Figure 1** is intended to show what a view definition would look like in a simple case and how one definition could be used in another. Later, a second objective will become clear: It will be seen how close these "pure" database queries are to logic programming queries. The example involves family relations in a traditional Dutch environment (see [22]). We define three relations, *person*, *children*, and *marriage*, stored as tables in the database. Further, some "rules" are defined using view definitions, indicating who the *females* are in this small world, and who is a *sister*, a *parent*, or an *aunt*.

• *Compiling, optimizing, and executing queries*

To be able to discuss implementation issues later, we need some knowledge of the query execution process in Business System 12. We use the above example to illustrate it. Suppose we would like to know who the aunts of 'ruud' are. This query can be written as

SL(aunt,N='ruud')

The Business System 12 relational compiler finds that *aunt* is the name of a view definition and substitutes for *aunt* the relational expression it represents. This relational expression contains new view names, in our case *sister* and *parent*. These are also substituted until only the names of base relations remain.

At the same time the compiler forms an access tree for the query. The root of the tree represents the query result, the nodes are the relational operations, and the leaves represent the base relations, stored as *tables* in the database. For our example the tree would look as shown in **Figure 2**.

The result of a query is formed by so-called "pipelined" execution. The root node starts asking for rows from its subsidiaries; these next nodes do the same until the leaf nodes are reached, and respond by sending the rows up. Each parent node performs its designated operations, such as matching rows when it is a join node. Rows not needed in the result are discarded on their way up. The actual rows are not formed at intermediate nodes; the complete row may not even be in storage yet. Only (parts of) rows which appear at the top are put in the appropriate data buffers and sent across the interface to the requestor. This is a general description of tree execution. In practice, there may be intermediate results due to optimization, sorting, etc.

The access tree is optimized by Business System 12 both locally and globally. Locally it might be done, for example, by deciding to use an index or a spool file to hold an intermediate result; globally, by changing the shape of the tree. This is very important when there is a *select* somewhere in the tree. By moving the select down in the tree as far as possible, the number of rows moving up during execution can sometimes be limited tremendously. In the case of our example, the select on *aunt* is moved down to the *children* tables at the right-hand side of the tree. The paper by Blaauw et al. [19] contains a description of optimization methodology which transforms access trees into the most cost-effective ones. Part of this method is implemented in Business System 12.

Logic programming in Business System 12

We consider logic programs consisting of Horn-clauses:

A :- B1, B2, B3, · · · Bn.
A :- C1, C2, C3, · · · Cm.

A is called a *predicate* or a *goal*. The propositions *Bi* and *Cj* are called *subgoals*. *A* is considered true if the *conjunction* of *B1, B2, · · · Bn* is true ***or*** if the *conjunction* of *C1, C2, · · · Cm* is true.

If there are no subgoals ($n = 0$), the left-hand predicate is always true. Horn clauses of this shape are called *facts*. A fact is a *simple fact* if it contains no variables as arguments.

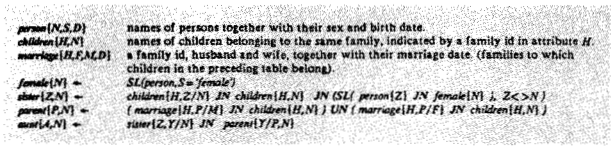

Figure 1

Family relation database views.

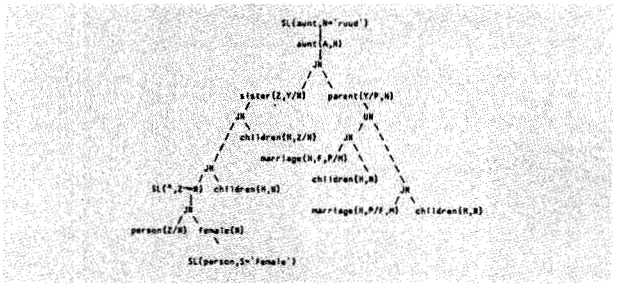

Figure 2

Access tree for query: SL(aunt,N = 'ruud').

For instance, *likes(john,mary)*. is a simple fact. Horn clauses with an empty goal are called *queries*.

• *Mapping of Horn-clause rules to relational algebra*

We proceed now by describing in more detail the process of translating Horn-clause rules to relational algebra:

- A predicate *P* maps to a relation *r*.
- If *P* has arity *n* (i.e., *n* arguments), *r* has *n* attributes.
- Each positional argument of *P* maps to a distinct attribute of *r* (this is possible because all arguments are assumed to be distinct).
- All rules *Di* for *P* map to distinct relations *ri*.
- *r* is the union of the *ri*'s.
- A Horn-clause rule which is a *simple fact* maps to a single row in a relation. The row contains in its attribute fields constant values, which are identical to the corresponding argument values of the simple fact.
- The conjunction of two Horn-clause-rule subgoals maps to the join of the corresponding relations. Equal variables of different subgoals map to equal attribute names for the join.
- A (sub)goal *S*, with constant arguments, maps to relation *rs* obtained as follows: Let *r* be the relation corresponding to the predicate associated with *S*. Then *rs* is formed from

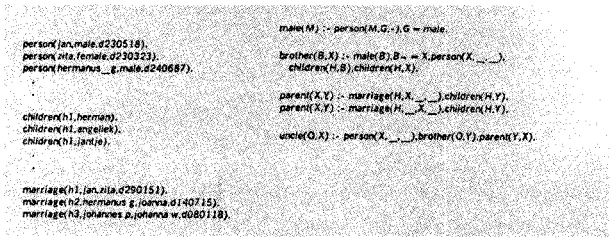

Figure 3

The persons database in Prolog.

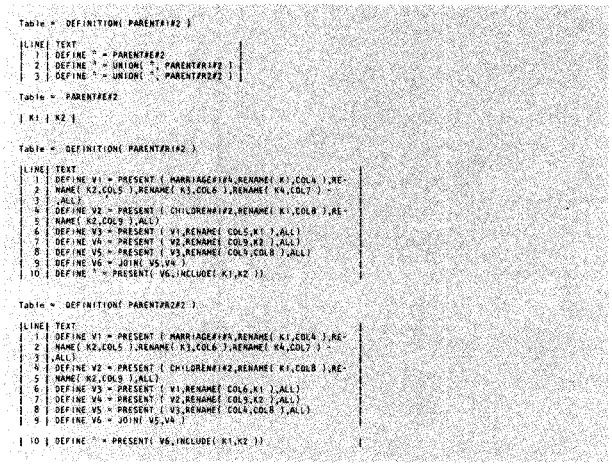

Figure 4

VIEWs for PARENT, made by NFP.

r by selecting only those rows that contain, in their corresponding attributes, the corresponding constant argument values.

- A Horn-clause rule *P* is mapped to a relation *r* obtained as follows: Let *P* consist of head *H* and subgoal(s) *S*. Let *rs* be the relation corresponding to *S*. Let *c* be a relation consisting of one row, which contains the constant arguments of *H* that do not occur in *S*. Let *rc* be the relation formed by taking the Cartesian product of *rs* and *c*. Then *r* is formed by presenting only those attributes from *rc* that correspond to the arguments of *H*.

This mapping does not cover *lists*, *structures*, and the Prolog *cut* operator. No distinction is made between built-in and user-defined predicates.

To see how this translation works in practice, we show in **Figure 3** an example similar to one we used earlier to demonstrate the view definition facilities of Business System 12 in a Prolog version. The translation of Prolog clauses to database views is obvious. *Note:* The Prolog syntax used in this paper is the Clocksin-Mellish syntax [15].

- *The prototype interface*

As part of his Master's thesis work [23], C. F. J. Doedens implemented a prototype for a logic programming interface written in PSC Prolog (a Prolog interpreter developed at the IBM Scientific Center in Paris). The functionality of this prototype is limited, partly because of the restricted scope of his project, partly because some new functions in Business System 12 are needed. Since Business System 12 is a commercial system, it is not possible to add functions to it just for research purposes.

In the Prolog subset supported by the prototype, it is possible to define facts and rules and to make queries, but no flow of control by means of a *cut* operator is provided. In fact, the prototype covers the mapping defined above, except that recursive queries, which are nicely mapped by the prototype, cannot be executed in Business System 12 in that form.

In addition to translation of logic programming queries, the prototype front end offers some user-friendly operational facilities, such as full-screen presentation of results. It is possible to use relations which are created outside the logic programming environment in a query, and it is possible to access relations made by the prototype through other Business System 12 interfaces (PL/I, APL, conversational facilities).

Due to the administrative technicalities involved, such as ordering and renaming attributes, the mechanical translation done by the prototype is not so trivial. To give an idea of the output produced by the prototype, a translation of *parent* is printed in **Figure 4**. We do not explain syntactical details.

Although the prototype interface has limited function, it gave us sufficient insight to enable us to make a fairly detailed list of the enhancements needed either in the prototype or in Business System 12 to arrive at a full logic programming interface. The next sections are devoted to describing these enhancements.

Towards a full logic programming interface

Our objective was to have all rules stored and evaluated inside the database. The preprocessor should remain a "dumb" syntax translation program which has no knowledge of semantics. We found, however, that by making the preprocessor slightly more intelligent, we could use considerably more of the functions available in Business System 12, and in that way provide important functions, such as *built-in predicates*, *recursion*, and *arbitrary structures*, while still maintaining our main objective. This enables us to see how logic queries behave in a relational database and will allow us to build a sizable application in the future. At a later stage, we can program more

understanding of the syntax of logic queries into the database itself, which will also allow for suitable optimization.

In the next sections, we describe the proposed enhancements to the prototype and indicate where changes to Business System 12 are necessary or profitable.

- *Predicates without arguments*

A naive understanding of our strategy tells us to translate facts without arguments into a base relation with one row and no attributes. An example is *true*. Such a construct is not externally available in Business System 12. Therefore, we add to all data relations a 0th attribute which need be only one bit wide. The question ?- *true*. will now indeed give a valid answer, and also *join*s with other relations will give the expected results. An additional advantage is that all relations now have at least one common column, and since this is already a requirement for using *join*s in Business System 12, it facilitates translation.

- *Multiple occurrences of a variable in a predicate*

In Prolog it is possible to have multiple occurrences of a single variable in different positions in a predicate. The *join* operator, which enables us to enforce equality of attribute values named by equal variables that occur in distinct predicates, cannot be used. However, using a *select* provides a suitable alternative (compare [24, Section 5.3]). The only requirement is that the preprocessor recognize this case.

- *Parallelism and flow of control*

The result of a relational database query is basically a set. This means that the order in which the tuples appear in the result relation is not predictable. (Of course, the user is given the opportunity to sort his rows if he wants to.)

By contrast, in conventional Prolog, solutions to queries are formed through a sequential search mechanism. Thus, the order in which predicates, with or without side effects, are executed is strictly determined. In Prolog there exists an operator called *cut* which influences this flow of control. When a *cut* is encountered, all choices after the parent goal is invoked become committed and no further solutions are attempted; i.e., all variables are assigned their last found value, and no further backtracking is done. See [15].

Something analogous can be implemented in Business System 12 by introducing a new relational operator, which we call the "breakpoint" operator. In database terms the function of a *cut* is equivalent to finding the first tuple of a view as defined by the rule(s) before the *cut* occurred. Inserting a breakpoint in the relational compile tree, which has the effect of realizing an intermediate result, and then taking the first tuple of this result, provides the required facility. The difference from the traditional *cut* is that the choice of the first row of the solution to a subquery is nondeterministic. Also, the use of a breakpoint operator inside the tree structure of a relational expression allows greater freedom in locating the commits than is permitted by the traditional *cut* operator, which is located in a sequentially ordered rule. It is possible, for example, to have two breakpoints located at independent branches in the tree, whereas two *cuts* inside a single rule are always related in the sense that one precedes the other. As a consequence, the semantics of the breakpoint operator will differ from the usual sequential interpretation, but could be compared to the guarded commands of Dijkstra, which are also used in concurrent Prolog as designed by Shapiro [25].

- *Not*

In the relational algebra, negation corresponds to complementation. If a negated clause occurs within a conjunction with a positive one, the complementation can be expressed by the relational *difference* operator, which is available in Business System 12. If a negated clause occurs in isolation, its meaning denotes complementation with respect to the universe, which leads to a possibly infinite relation. Since this form of complementation is not available in Business System 12, we do not support this type of negation, unless the complementation is restricted to a finite domain or the negated predicate has a positive equivalent by definition. For example, predicates expressing (in)equalities of arithmetic values can be negated.

It is likely that an implementation based on the difference operator in some circumstances will behave differently from the negation by failure implementation required by Prolog. It is unlikely that negation by failure is a reasonable aim for a database-oriented parallel evaluator, since in Prolog it is highly intertwined with the sequential evaluation strategy.

- *Built-in predicates*

The predicates which are built into most Prolog interpreters can be divided into two categories:

- Predicates causing side effects, such as *write* and *assert*.
- Arithmetic predicates, such as *sum* or *substring*.

Depending on the specific side effect under consideration, the intended result of a built-in predicate of the first type can be either innocent or highly detrimental to the contents of a database. Consequently, a uniform treatment of side effects does not exist. For example, it seems reasonable to print values encountered, but the order in which results will be printed is unpredictable. In the case of an assert or retract, the intended meaning is a modification of a database at the very same time this database is queried. Whether this is possible is dependent on the relations affected. We abstain from further comments on this topic.

In practice, arithmetic predicates describe a functional dependency between attribute values in an already bound

domain. These dependencies can conveniently be expressed in the Business System 12 query language, using the *calculate* relational command.

- *The unbounded view problem*

There is a class of rules which have no obvious translation to Business System 12 query language, such as

```
are_the_same(X,X).
dead(All) :- no_air_on_earth.
```

These facts and rules introduce two problems from the perspective of our interface:

- The data type of the values of the variables is unknown.
- The intended relation becomes infinite in general.

The first problem does not arise in Prolog, since that is an untyped language. But in Business System 12, all attributes are required to have an associated domain. This means that we should specify somehow explicitly the domains for variables which occur on the left-hand but not on the right-hand side of a view definition. A natural place would be within the view definition, but this is difficult in the present Business System 12 syntax. Instead, we can prevent the problem by always requiring the presence of some predicate on the right-hand side whose only purpose is to specify the type of those variables which have no other occurrences on the right-hand side of a view definition. For this purpose we provide in the enhanced system for every domain a corresponding built-in predicate with one attribute whose name equals the name of the corresponding domain. Its meaning is to be the characteristic predicate for this domain. The above two rules are now rewritten to

```
are_the_same(X,X) :- integer(X).
dead(All) :- human(All) , no_air_on_earth.
```

Obviously *integer* and *human* are the domains for *X* and *All*.

The examples above suggest that this type of rule leading to unbounded views has rather limited expressive power. Universally quantified assertions can be expressed, and it is also possible to enforce some equalities within these universally quantified assertions, as illustrated by the *are_the_same* example.

The second problem, which concerns having to deal with possibly infinite relations, is more interesting. Infinite relations will result if we interpret the built-in predicates describing arithmetic relations as ordinary database relations. In both cases the problem is not that the relational semantics is inadequate to provide the intended meaning, but that the database does not allow us to store infinite relations in tables. Database views traditionally are composed from a finite collection of finite base relations. Even for an extended database where recursive views are allowed, these recursive views are defined in terms of finite base relations.

A relational expression which involves infinite arguments must produce a finite result to be meaningful as a query. When the result is finite, it should also be possible to execute the query, and the check whether this condition is fulfilled should be a syntactic one. For example, consider the following Prolog rules:

```
earnsless(X,Y) :- salary(X,A),salary(Y,B), less(A,B)
netincome(X,A) :- salary(X,B), withholdings(X,C),
  sum(A,C,B)
```

In both cases, the resulting join describes an ordinary finite relation. In the first example, the *less* predicate is a restriction on the bounded product of two copies of the finite *salary* relations. In the second relation, one uses the fact that each of the three argument positions in the sum predicate is functionally dependent on the other two. So, as soon as two arguments are restricted to a finite domain, the third argument is bounded as well. And in both cases, a relational query producing the correct answer can be constructed based on a *select* or *calculate* operator.

These examples indicate that a system dealing with built-in predicates of the above type can indeed be designed. It will be based on a boundedness calculus, which will include among others the following rules:

1. Base-relation attributes are bounded; some attributes in general facts and specific built-in predicates can become unbounded.
2. An unbounded attribute becomes bounded if joined with a bounded attribute having the same name.
3. An attribute that is functionally dependent on bounded attributes is bounded.

Justification is clear for the first rule, which describes the finiteness of the base relations. For the second rule, we can specify an evaluation strategy based on semi-joins, whereas for the final rule a strategy based on the *calculate* operator will work.

The result of a query bounded in all attributes on the basis of these rules is indeed finite, and the rules also provide us with an evaluation strategy which does not require infinite intermediate results.

At this point it should be observed that this problem is very similar to the problem of how to plan query evaluations making optimal use of specified arguments, which was investigated by Ullman in his *capture rule* work [6]. The difference is only in the interpretation of the words *bounded* and *free*. Ullman assumes that his relations are *safe* [24], which means that our unbounded view problem does not arise. In his terminology, *bounded* denotes bounded to a single value, whereas for us it means bounded to a finite domain. We observe that the type of rules described by Ullman for obtaining efficient evaluation strategies can be

used in the extension of our system for checking boundedness.

In the full logic programming interface, the compiler will test the query submitted to the system for formal boundedness. If the query is found to be bounded on the basis of the rules of the calculus defined above, the system will produce an evaluation strategy at the same time. If not, the user will get an error message informing him of the source of the problem.

We will not require our system to be omniscient. For example, our system will not know the difference between a function such as the cosine and a nontrivial polynomial, which is expressed by the mathematical result that the polynomial has only finitely many real zeros, whereas the cosine has an infinite number of them. Neither has it the mathematical knowledge needed for building equation solvers. For example, in the first rule below, the boundedness of *X* and *Y* can be inferred, given that *A* and *B* are bounded, while this is impossible in the second rule because boundedness here is not based on functional dependencies but on linear algebra:

?- sum(A,B,X), sum(A,Y,B)
?- sum(X,Y,A), sum(X,B,Y)

It should be clear that handling unboundedness requires an intelligent preprocessor/compiler, which must have access to semantical information on built-in predicates (boundedness and functional dependencies).

• *Arbitrary structures*

In Prolog it is possible to instantiate variables, not only to constant values as in the relational model, but also to tree- or list-like structures.

To map these structures onto the database, we are forced to depart from the relational model in which nonatomic attribute values are not allowed. In return, however, we will be able to describe hierarchies, apart from providing functional capabilities similar to those in Prolog. To many, the impossibility of dealing with hierarchies is felt to be a severe functional restriction in relational databases today, preventing engineering and expert system applications.

The mapping we use is based on the foreign key concept and an extension of the prototype mapping described in the subsection "Mapping of Horn-clause rules to relational algebra." A *foreign key* is a set of attribute values which occur as *key attribute* values in another or the same relation.

We want to stress that the mapping we are going to describe is a *conceptual* solution. It can be used in a prototype implementation, using current database facilities, to check its functionality and performance. A more practical and less storage-consuming implementation will require database changes to make what we describe transparent to a user or preprocessor, but that does not change the applicability and feasibility of what we propose.

Consider the example

owns(john,table).
owns(john,book(odyssey,homer)).

Our translation algorithm suggests that this be put in the database in a relation *owns{A1,A2}* as

[A1: john,A2: table] ∈ owns.
[A1: john,A2: book(odyssey,homer)] ∈ owns.

The attribute *A2* now contains the nonatomic value *book(odyssey,homer)* in one of its tuples. In itself this item suggests translation to a relation *book{B1,B2}* as

[B1: odyssey,B2: homer] ∈ book.

This changes the *book* structure into a tuple with nice atomic values, but how can we find the row for this book back in the *owns* relation? Let us assume that we assign an identification *h1* to the tuple

[B1: odyssey,B2: homer] ∈ book.

Such a unique identifier can always be found, using common practice hashing techniques. We now change the *owns* table to

[A1: john,A2: table] ∈ owns.
[A1: john,A2: refto(h1,book)] ∈ owns.

The value of *refto(h1,book)* can be made atomic by using two separate attributes to store *h1* in the first, and a unique identification of the relation to which it belongs in the second.

In general, if

$p(A1,A2,\cdots,An).$

maps to

$p\{A1,A2,\cdots,An\},$

in the prototype translation, then we will extend the mapping to

$p\{H,R1,T1,A1,R2,T2,A2,\cdots,Rn,Tn,An\},$

where

H contains a unique identifier (a *key* in database terms) for each tuple, as described above.
Ri contains an identifier for a relation.
Ti contains a foreign key value to a relation whose name is in *Ri*.
Ti and *Ri* provide together the *refto* function as described above.
If *Ri* and *Ti* do not contain identifiers (but zeros or blanks instead), then *Ai* contains an atomic value for that tuple.

The translation of conjunctions to joins and multiple rules for the same predicate into unions carries through unchanged for the new mapping. Instead of single attributes,

triplets of attributes have to be taken care of now. That may be cumbersome to humans, but it should not be to a mechanical preprocessor.

Note: In Business System 12, there actually are already *H* attributes present for every relation, hidden from the user. Therefore, it will be easier to implement a user-hidden triplet structure than the above description might suggest.

As an example, consider the question

?- *owns*(*john*,*book*(*X*,*homer*)).

In our implementation, this query will be translated into

SL(*owns*,*OA1*='*john*' & *OR2*='book') ***JN***
SL (*book*{*OT2*/*H*,···},*BA2*='*homer*').

(We made attribute names unique by prefixing them with the first letter of the table name; *OT2* is the common attribute on which the join is performed.) Before being shown to the user, the remaining references should be transformed into the corresponding rows. The *process function*, a user-definable database procedure operating on every row in a table or view, can perform this task in a convenient way. Note that inside a query, references need not be expanded to do conjunctions (joins), because equality of a reference is sufficient.

We still have to answer some questions:

1. Can we indeed represent arbitrarily complex structures?
2. Does unification work as it should?
3. How does this proposal compare with other approaches to composite objects?

With respect to the first question, we observe that the problem of aliasing (having two different access paths to a single structure) does not arise. By definition of the foreign key concept, it is impossible for two references with different values to refer to the same row in some table. Thus, if John and Mary have the same book, they will have the same reference to it. This suffices to prove by induction that two structures are equal if and only if their representations as proposed are equal.

There is a problem in the sense that our proposal implements too many structures; a foreign key can refer directly or indirectly to the row in which it is stored. Prolog does not support such infinite recursive structures.

Regarding the second question, if we restrict ourselves to considering only those structures that are also supported by Prolog, the structures are equivalent to general directed acyclic graphs. This is exactly the domain in which the unification algorithm proceeds (see, for example, [26]).

Concerning the third question, we compare our approach to the proposals in [12] and [13]. In Zaniolo's proposal, a reference is bounded to a single table only. This makes it impossible for John to own both books and cars, if both books and cars are structured objects. In his semantics, the value of a reference is simply the value of the tuple it refers to. This has the required uniqueness property but will probably never be implemented in this way. Thus the question of whether two different references can refer to the same row returns to the implementation. Moreover, this semantics precludes circular structures. Finally, in Zaniolo's proposal, it is not clear whether a join can be performed on a reference column. References cannot be listed or shown to the user; whether they can be tested for equality is not clear. The proposal also supports sets as attribute values; it is not clear whether this has any usefulness for our logic programming interface.

The proposal of Lorie and Plouffe seems to provide the required flexibility; references can refer to different tables, but in this case John cannot both own books and chairs when a chair is a plain attribute value. In their semantics, references are keylike identifiers. No two different identifiers can refer to the same row. References can be tested for equality and for membership in some particular table. Cyclic structures are possible; updating is very restricted.

The mapping we propose is simple but effective, and we see no impact on performance of the evaluation of queries. Structures can be unified (or joined in database terms) by using only their references.

• *Lists*

Lists are a specific form of a structure. Therefore, we can translate lists by using their structural form.

A more convenient approach is to use a different but similar mapping. Instead of referencing a row, reference is made to a *serial table.* A serial table is an ordinary data table in Business System 12; however, rows are kept in arrival order, not randomly. Elements of the list will be stored as rows in the serial table. Since each element of a list can be a structured item, another list, or just a value, further referencing can occur in any row of the serial table.

This construction may lead to a large number of tables, one for each list. Although Business System 12 can create these dynamically, as described earlier, it may still provide performance problems. This seems not to be a severe difficulty, because in database programming lists are not used as often as they seem to be in logic programming. The reason is that in a database *sets* are more readily available. *Set* operators could easily be provided to the Prolog programmer by implementing some built-in predicates.

For serial tables, *head* and *tail* operations can be implemented in a simple way. Curiously enough, the *head* operator is very similar to *breakpoint*, the database equivalent of *cut*.

• *Recursive views*

It is well known that recursive queries can be replaced by an iteration of nonrecursive queries. With some restrictions, the results of these queries increase monotonically, tending to a

fixed point. Theoretical background can be found in books by de Bakker [27] or Manna [28].

A scheme to implement iteration within a database could be as follows. Let

a :- ···
a :- ···, *a* , ···

be a recursive Prolog query. We translate this into a database query:

$a(n) \leftarrow \cdots \; \textbf{\textit{UN}} \cdots \textbf{\textit{JN}}\; a(n-1)\; \textbf{\textit{JN}} \cdots$
$a(0) \leftarrow$ *empty* (*contains no rows*)

where $a(n)$ is a base table with the same set of attributes as the recursive view. To find the result of the recursive query a, we successively execute the queries $a(0), \cdots, a(n)$. The result of query $a(n-1)$ is stored as a base table and used as such to execute query $a(n)$. If the result of $a(n)$ is equal to the result of $a(n-1)$, the iteration is finished, and we have found the result of the recursive query a.

All processing can be done at the database side; except for the final result, no data must pass the interface, which is usually a slow performer. Compare routines, etc., requiring knowledge of the internal data structure in the database, do not have to be implemented in the foreground processor. Some communication is nevertheless necessary, because the foreground processor needs to be able to restrict the size of the result in case it is very large or infinite.

There is also room for optimization. In the simple case of a single recursive view, which occurs only once in its recursive definition, we need to store as the result of $a(n)$ only the tuples which are not in $a(0)\; \textbf{\textit{UN}}\; a(1)\; \textbf{\textit{UN}} \cdots \textbf{\textit{UN}}\; a(n-1)$. When no new tuples result for an iteration, we are finished. The total result is now the *union* of the results of all executed iterations.

Further optimization can be achieved by keeping intermediate results of the nonrecursive parts of the query (when they are not in the same branch of the tree).

Note that it is not sufficient to keep intermediate results for the whole query. If the recursive view is not also the goal of the query or if there is more than one recursive view involved, then it is necessary to keep an intermediate result at the root of each recursive view invocation in the relational access tree.

It was found by Naqvi and Henschen [9] that this approach to recursion can become quite inefficient if the recursive query is subjected to some selection elsewhere in the expression, since it is not possible in many cases to propagate knowledge about specified argument values into the fixed-point iteration results. These optimization problems are also investigated in recent work of Ullman [6].

Let us clarify this with the ancestor example. In Prolog, it is represented by

ancestor(*X*,*Y*) :- *parent*(*X*,*Y*)
ancestor(*X*,*Y*) :- *parent*(*X*,*Z*), *ancestor*(*Z*,*Y*)

This translates into the following recursive view definition:

$ancestor\{X,Y\} \leftarrow (parent\{X,Y\})\; \textbf{\textit{UN}}(parent\{X,Z/Y\}\; \textbf{\textit{JN}}\; ancestor\{Z/X,Y\})$

For example, one might ask who the ancestors of Mary are:

$\leftarrow \textbf{\textit{SL}}(ancestor, Y='Mary')$

In this case, it is possible to push the select down in the relational tree, a common optimization technique in relational databases. However, when it is asked to whom Mary is an ancestor,

$\leftarrow \textbf{\textit{SL}}(ancestor, X='Mary')$

this optimization is not allowed because the attribute X is renamed before the recursive invocation of the view. See also [5], where the same problem is observed. It is even more difficult to determine when optimization is allowed when the restriction is in the form of another query, such as "Provide the ancestors of all speed-skating world champions of the last five years."

• *Iterative compilation of recursive views*

As a step towards solving the performance problems mentioned, we introduce a new, more general approach to optimizing recursive relational queries. We refer to it as the *iterative compile* approach, as opposed to the *iteration on results* approach described in the previous section. It is based on the following unfolding method. Let a be a recursive query as before. Its translation to an iterative query is also syntactically the same as before. The basic difference is that instead of using the *result* of the previous iteration, the *definition* of the previous iteration is used to find the *definition* of the next iteration. Each successive query has to be compiled, optimized, and executed, and its result is saved. The results are compared at each step. When the results are equal, we are ready.

The important advantage of this approach is that *a recursive query is reduced to stepwise queries, which a relational database can handle as effectively as it is capable of.* In many cases, separate steps of the iteration can be optimized effectively, whereas this was impossible with the iteration on results method. The *same generation* query which we describe later is a good example.

In simple cases where stepwise compilation does not provide additional optimization, the iterative compile approach does not necessarily perform worse than the iteration on results method. Both methods have some overhead between steps to compare results and to readjust access trees. There is a slight increase in the amount of main storage required for the compiled method due to the larger access trees; however, the most storage is usually required to keep information about stored tables, and their number does not increase after the first step.

The next iteration step can be compiled effectively by keeping a copy of the tree for the compile step of $a(1)$.

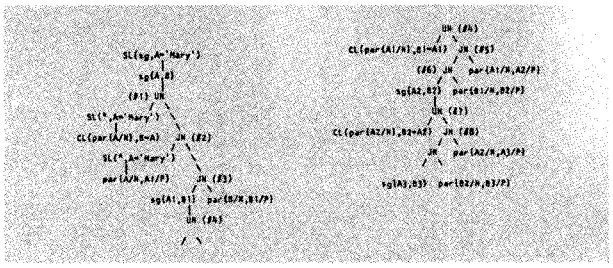

Figure 5

Access tree for query. SL(sg,N = ' Mary ')—three levels deep.

Replacing $a(0)$ with a copy of the tree for $a(1)$ in the tree of the nth step (after column renamings, if necessary), the $(n+1)$th step is easily obtained. In Business System 12, the output of the compiler is executable, and the optimizer transforms an executable access tree into another executable access tree; therefore, we can re-apply the optimization process to the compile tree for the $(n+1)$th step, which we obtained in the way just described.

Further performance improvements are left as a future research topic. We note:

- There is no reason to use the compile steps of 1. In cases where the recursion depth can be estimated, as may be possible in a database environment where queries are often of a repetitive nature, larger step sizes can be used and initially more steps can be compiled. This results in a significant reduction in the number of iteration steps required.
- It may be possible to find subexpressions which do not change between iteration steps. These can be kept as intermediate results, avoiding recomputation at each step.
- It seems that the execution iteration is just an optimized special case of the compiled iteration, considering the previous point. We do not discuss this further.

We clarify the iterative compilation method and its advantage using a more involved example (due to Ullman). Assume that we would like to determine people who belong to the same generation in a family database, as expressed by this Prolog query:

sg(A,B) :- A == B.
sg(A,B) :- par(A,AX) , par(B,BX) , sg(AX,BX).

We translate these rules, using our abstract relational database syntax. The child-parent relation is defined as *par{P,N}*. The database view is defined as

sg{A,B} ←
***CL**(par{B/P},A←B) **UN** (par{A/N,AX/P}*
***JN** par{B/N,BX/P} **JN** sg{AX/A,BX/B})*

The first clause, $A == B$, is translated using a *calculate* relational operator. See **Figure 5** for a pictorial representation of an expansion up to the third iteration for this recursive view.

Notice that the order of the arguments for some joins and unions is reversed and that the select is pushed down in the tree (which was not possible for the iteration on results case). This optimization is done automatically by the database query optimizer.

Keeping in mind the pipelined execution method described under "Compiling, optimizing, and executing queries," we see that the query executor indexes itself through the relational access tree, accessing only those rows from the *par* table which are necessary to form the result. Only generally applicable optimization methods are used to achieve this. A requirement is that indexes be available on both N and P attributes, which certainly will be true for larger database applications; otherwise, the optimizer will most likely decide to make them anyway.

As a consequence, the execution time will only be dependent on the number of recursion steps required, the complexity of the query itself, and the size of the result, but not on the size of the tables involved.

Currently, queries can be executed in Business System 12 in the way described above. This can be done by using a database procedure and a view definition which accept arguments to control the recursion depth and the necessary column renaming. The contents of the view definition and the procedure are shown in **Figure 6**, just to give an idea what this might look like. We do not explain the syntax in more detail.

SG0 is a predefined empty table (it contains no rows). The query can be executed by typing *run samegen('Mary')*. The result can be found in either TTAB1 or TTAB2. These can be used in further queries or can be displayed with *display ttab1*.

Because it is not yet possible to define indexes on both columns in the *par* table in Business System 12, performance is not as good as theory suggests.

Using the *same generation* query, we did some crude performance measurements. We used a *parent* table with a small set of rows in Business System 12 and a set of the same facts in Prolog. Next we added a variable number n of noise rows, with n ranging between 100 and 1000. These rows contained data which did not contribute to the result in any way. We also added the equivalent facts in Prolog. In both cases, we put the relevant data somewhere in the middle. For Business System 12, we used the code shown above, which is not optimal because the query must be completely recompiled at each step. For Prolog, we added a rule to obtain all results.

For both experiments we observed a running time of polynomial order in n with exponent approximately 2 for Prolog and exponent approximately 1.5 for Business System

12. We are aware of the fact that in this experiment the performance of Business System 12 is suboptimal; much of the observed growth can be avoided when the optimizer becomes capable of better indexing support.

Summary and conclusions

As a main topic, this paper has described how today's relational database management systems can be made more intelligent. Combining ideas put forward by other researchers in the database and logic programming fields, we have outlined an integrated approach to arrive at a logic programming system, beginning with a state-of-the-art relational database, Business System 12. For each function which requires changes to the database, we have described how it could be implemented and, if appropriate, how others viewed the problems.

Currently, we have only implemented a restricted prototype. We have shown, however, using examples of queries which are executable now in Business System 12, how the prototype can be made more intelligent and that it can support most logic programming functions. Support for structures, the iterative compile approach to recursion, the arithmetic built-in predicates, and some minor points such as a 0th column can simply be added. Meanwhile, we have kept our objective that all inferencing should be done by the database manager.

We have also indicated how at a later stage Business System 12 could be improved in a few areas, such that a logic programming interface could be supported in a more transparent manner and performance could be enhanced. This included a review of the features just mentioned for the prototype, support for which should then be removed. Also included are the breakpoint operator, built-in functions causing side effects if appropriate, iteration by execution of recursive queries, and optimizations for the compiled iteration.

New in our approach are that the whole logic programming environment (both rules and facts) is stored inside the database and that all inferencing is done inside. Apart from showing that and how it is possible, we have argued these advantages:

- All database functions, including *sharing*, *authorization*, *concurrency control*, etc., become available to the logic programmer, while avoiding the huge duplicated implementation effort of these functions which is inevitable in any distributed setup.
- Since whole queries are inside the database, a generalized and global approach to optimization of queries is possible. As a consequence, we were able to show a new method of evaluating recursive queries, the *iterative compile* approach, which gave promising performance measurement results. We pointed out several open problems in this area towards which future research can be directed.

```
  view SG
PARAM 1 SYNTAX
PARAM 2 SYNTAX
PARAM 3 NUM
PARAM 4 NUM
!CL1=SYNTAX("N"||CHAR(!3))
!CL2=SYNTAX("N"||CHAR(!3))
!N=!3+1
IF !3 = 14
 THEN
  * = PRE(SGO,REN(G1,!1),REN(G2,!2))
 ELSE
  * = UNI(CAL(PRE(PAR,REN(M,!2)),!1=!2), -
      JOI(PRE(PAR,REN(P,!CL1),REN(M,!3)), -
      JOI(PRE(PAR,REN(P,!CL2),REN(M,!2)), -
      SG(!CL1,!CL2,!M,!N))))
 ENDIF
RETURN

  clist SAMEGEN
PARAM 1 CHAR
LOOP !1T UNTIL EMPTY("EXCL(TTA01,TTA02)")
IF MODULO(!1T,2)=0
  THEN STORE SEL(SG(G1,G2,0,!1T),G2=!1), -
                TTA01,REPLACE
  ELSE STORE SEL(SG(G1,G2,0,!1T),G2=!1), -
                TTA02,REPLACE
 ENDLOOP
RETURN
```

Figure 6

The same generation query translated to Business System 12 procedure and view definition langauge.

The database-logic programming interface will have semantics which are different from conventional Prolog to allow for database parallelism, but it will adhere to the declarative features of pure Prolog.

We feel that the described additions to a relational database are a prerequisite to enable the development of larger, more complicated, and multi-user interactive expert systems.

Acknowledgments

This work would not have been possible without the continuing, enthusiastic, and inspiring support of A. J. du Croix, Business System 12 development manager. Also, we appreciate the excellent contribution of C. F. J. Doedens for the purpose of his Master's thesis study. He designed and implemented the prototype interface and pointed out many problems. We are grateful for the opportunities we had during our stay at the IBM Research Center in San Jose, California, to work further on our project and to discuss it with various people in IBM Research and at universities. Finally, we acknowledge the anonymous referees for the additional background references and the suggestions they provided in their reports.

References

1. C. L. Chang, "DEDUCE 2: Further Investigations of Deduction in Relational Databases," *Logic and Databases*, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp. 201–236.
2. R. Reiter, "Deductive Question-Answering on Relational Databases," *Logic and Databases*, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp. 149–177.
3. D. Maier, "Is Prolog a Database Language?", Oregon Graduate Center (Draft, 1983).
4. S. A. Naqvi, "Prolog and Relational Databases: A Road to Data Intensive Expert Systems," Bell Laboratories, Murray Hill, NJ, 1983.
5. M. Jarke, J. Clifford, and Y. Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System," Proceedings of SIGMOD'84, *ACM SIGMOD Rec.* **14**, No. 2, 296–306 (1984).
6. J. D. Ullman, "Implementation of Logical Query Languages for Databases," *ACM Trans. Database Syst.* **10**, No. 3, 289–321 (1985).

7. Y. Vassiliou, J. Clifford, and M. Jarke, "How Does an Expert System Get Its Data?" *Report No. CRIS#50, GBA#83-26(CR)*, New York University, New York, 1983.
8. C. L. Chang, "On Evaluation of Queries Containing Derived Relations in a Relational Database," *Advances in Database Theory*, Vol. 1, H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1981, pp. 201–236.
9. S. A. Naqvi and L. J. Henschen, "On Compiling Queries in Recursive First-Order Databases," *J. ACM* **31**, No. 1, 47–85 (1984).
10. A. Walker, "Syllog: An Approach to Prolog for Non-Programmers," *Research Report RJ-3950*, IBM Research Laboratory, San Jose, California, 1983.
11. A. Walker, "Syllog: A Knowledge-Based Data Management System," *Report No. 34*, Department of Computer Science, New York University, New York, 1981.
12. R. Lorie and W. Plouffe, "Complex Objects and Their Use in Design Transactions," Proceedings of IEEE Database Week, *ACM SIGMOD Rec.* **13**, No. 4, 115–121 (1983).
13. C. Zaniolo, "The Database Language GEM," Proceedings of IEEE Database Week, *SIGMOD Record* **13**, No. 4, 207–218 (1983).
14. S. Tsur and C. Zaniolo, "An Implementation of GEM—Supporting a Semantic Data Model on a Relational Back-End," Proceedings of SIGMOD'84, *ACM SIGMOD Rec.* **14**, No. 2, 286–295 (1984).
15. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag New York, 1981.
16. T. Imielinski and W. Lipski, Jr., "The Relational Model of Data and Cylindric Algebras," *J. Comput. Syst. Sci.* **28**, 80–102 (1984).
17. H. Gallaire, J. Minker, and J.-M. Nicolas, "Logic and Databases: A Deductive Approach," *Comput. Surv.* **16**, No. 2, 153–185 (1984).
18. E. F. Codd, "Extending the Database Relational Model to Capture More Meaning," *ACM Trans. Database Syst.* **4**, No. 4, 397–434 (1979).
19. G. A. Blaauw, A. J. W. Duijvestijn, and R. A. M. Hartmann, "Relational Expression Optimisation," *Report No. TR 13.190*, IBM INS-DC Uithoorn, the Netherlands; "Optimization of Relational Expressions Using a Logical Analagon," *IBM J. Res. Develop.* **27**, No. 5, 497–519 (1983).
20. *Business System 12 User's Guide*, Order No. SH19-6364, available through IBM branch offices.
21. C. J. Date, *A Guide to DB2*, Addison-Wesley Publishing Co., Reading, MA, 1984.
22. R. P. van de Riet, "Knowledge Bases" (de databanken van de toekomst), *INFORMATIE* **25**, 16–23 (1983).
23. C. F. J. Doedens, "Logic Programming and Business System 12," *Report No. TR 13.198*, IBM INS-DC Uithoorn, the Netherlands, December 1984.
24. J. D. Ullman, *Principles of Database Systems*, 2nd ed., Computer Sciences Press, Rockville, MD, 1982.
25. E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," *Report No. TR-003*, Weizmann Institute of Science, Israel, February 1983.
26. Cynthia Dwork, Paris C. Kanellakis, and John C. Mitchell, "On the Sequential Nature of Unification," *J. Logic Program.* **1**, 35–50 (1984).
27. J. W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.
28. Z. Manna, *Mathematical Theory of Computation*, Computer Science Series, McGraw-Hill Book Co., Inc., New York, 1974.

Received June 14, 1985; revised August 8, 1985

Ghica van Emde Boas-Lubsen *IBM Information Network Services Support Center, P.O. Box 24, 1420 AA Uithoorn, the Netherlands.* Mrs. van Emde Boas received a B.S. in mathematics from the University of Amsterdam, the Netherlands, in 1968. Since joining IBM in 1969, she has worked at the laboratory in Uithoorn. For several years she participated in testing various releases of the DOS/VS operating system, receiving an Outstanding Contribution Award related to this work. For the last six years she has been part of the Business System 12 development team, involved primarily in relational query compilation, locking, and transaction management. Mrs. van Emde Boas spent the first eight months of 1985 at the IBM Research laboratory in San Jose, California.

Peter van Emde Boas *Departments of Mathematics and Computer Sciences, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, the Netherlands.* Prof. Dr. van Emde Boas has been head of the theoretical computer science group at the University of Amsterdam since its creation in 1983. Since 1977 he has been a professor in the field of mathematical computer science. He received an M.S. in mathematics in 1969 and a Ph.D. in sciences in 1974, both from the University of Amsterdam. Between 1964 and 1977 he was a member of the Department of Pure Mathematics of the Mathematical Center in Amsterdam, and since 1969 also of the Mathematics Department of the University. In the fall of 1974 he was a research associate in the Department of Computer Science of Cornell University, Ithaca, New York, and he spent the first eight months of 1985 as a visiting scientist at the IBM Research laboratory in San Jose. His current research interests are computation theory, semantics, and database theory.