Algebraic optimization of recursive queries*

Maurice A.W. Houtsma and Peter M.G. Apers

University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract

Houtsma, M.A.W. and P.M.G. Apers, Algebraic optimization of recursive queries, *Data & Knowledge Engineering* 7 (1992) 299–325.

Over the past few years, much attention has been paid to deductive databases. They offer a logic-based interface, and allow formulation of complex recursive queries. However, they do not offer appropriate update facilities, and do not support existing applications. To overcome these problems an SQL-like interface is required besides a logic-based interface.

In the PRISMA project we have developed a tightly-coupled distributed database, on a multiprocessor machine, with two user interfaces: SQL and PRISMAlog. Query optimization is localized in one component: the relational query optimizer. Therefore, we have defined an eXtended Relational Algebra that allows recursive query formulation and can also be used for expressing executable schedules, and we have developed algebraic optimization strategies for recursive queries. In this paper we describe an optimization strategy that rewrites regular (in the context of formal grammars) mutually recursive queries into standard Relational Algebra and transitive closure operations. We also describe how to push selections into the resulting transitive closure operations.

The reason we focus on algebraic optimization is that, in our opinion, the new generation of advanced database systems will be built starting from existing state-of-the-art relational technology, instead of building a completely new class of systems.

Keywords. Deductive databases; relational databases; recursive queries; algebraic optimization; transitive closure.

1. Introduction

As a result of the clear need for database systems that provide more reasoning power than traditional relational database systems, a lot of attention has been paid to so-called deductive databases. These deductive databases offer a logic programming interface, called Datalog, that allows an easy, declarative specification of complex problems. Examples of such systems are NAIL! [31] and LDL [32]. Declarative query languages are, of course, since long associated with relational databases, but the new aspect of deductive databases is their support of recursive queries.

Traditionally, queries are optimized by a relational query optimizer. This optimizer produces a schedule to manipulate the huge sets of data stored in the database in such a way that the answer to a query is computed efficiently. To reach this goal, it uses a number of

* The work reported in this document was partly conducted in the context of the PRISMA project, a joint effort with Philips Research Laboratories Eindhoven, partially supported by the Dutch "Stimuleringsprojectteam Informaticaonderzoek Nederland (SPIN)"

strategies such as pushing selections into a query tree, determining join order, and introducing semijoins. These strategies do not suffice when dealing with recursive queries; therefore, much research has been focused on optimization strategies for recursive queries.

In the optimization of recursive queries we may discern two approaches: a logic approach and an algebraic approach. Researchers from the field of logic programming are mostly concerned with a logic-based approach, focusing on issues relevant in the context of logic programming, whereas researchers from the database field are mostly concerned with algebraic approaches, focusing on set-oriented optimization as usually studied in the database field. However, in [14] it is clearly shown that Datalog can be translated into equations of positive Relational Algebra and vice versa. So, we may first optimize in a Datalog context, then translate the resulting program into Relational Algebra and execute it on the database. But we may also first translate from Datalog into Relational Algebra, optimize the resulting program, and then execute it on the database. We may even go back and forth between Datalog and Relational Algebra, optimizing in each subsequent step (however, in this case care should be taken that optimization performed in one formalism is not destroyed by another optimization in the other formalism). It shall be no surprise that, given the resemblance between Datalog and Relational Algebra, optimization strategies in both contexts have a striking resemblance. We will go into this aspect later.

A disadvantage of deductive databases is that there is not yet a satisfactory way of dealing with updates in a logic query language. Although a logic query language is an appealing tool for formulating complex queries, it is, as such, not sufficient as user interface to a database system. Hence, there still is a need for conventional user-interfaces to a database system, besides a logic programming one.

In the PRISMA project we have developed a tightly-coupled, distributed database on a multiprocessor machine [4, 29]. And following the observations above, we have implemented two user-interfaces on the system. A standard SQL interface to deal with updates and to be able to run existing applications, and a logic query interface for easy formulation of complex queries and support of recursive queries. The logic query language is called PRISMAlog [27]; it is an extension of standard Datalog. The semantics of PRISMAlog is purely declarative, which means that the ordering of rules and predicates is not important. So, only the specification of the problem is given in PRISMAlog, how to solve it is left as a task for the database system — more specifically the query optimizer.

In the PRISMA project we thus have two interfaces that provide us with queries that have to be optimized and executed. To facilitate this process, it seems sensible to translate both query languages to *one* intermediate language that can be used for optimization and execution. This intermediate language is called eXtended Relational Algebra (XRA), which is essentially the Relational Algebra extended with constructs to support recursive query processing. This has as a big advantage that there are many algorithms available for relational query optimization, which is particularly important in a distributed environment. And since we now have all optimization concentrated in *one* component — the relational query optimizer — we also have to develop algebraic optimization strategies for recursive queries. We feel that this approach is more appropriate than to put the optimization in the translation from PRISMAlog to XRA. Particularly, since we are dealing with a distributed environment, where information about fragmentation and replication of data may be very important in the optimization [10].

The structure of this paper is as follows. In Section 2 we give a short overview of existing logic and algebraic optimization strategies. In Section 3 we describe the extension made to Relational Algebra that enables us to express recursive queries. In Section 4 we distinguish several types of recursion. In Section 5 we describe an optimization strategy for the most common type of recursion that we call regular. In Section 6 we describe how to deal with

recursive queries that do not belong to this class. In Section 7 we discuss the transitive closure operation, which is crucial to our optimization, and how to push selections into this operation. The use of distributed computation for this operation is also discussed. Finally, Section 8 concludes this paper. Part of this work has been described before in [2, 3, 22].

2. Approaches to recursive query optimization

In this section we give a short overview of some optimization strategies, both in the context of logic and in the context of algebra.

2.1 *Coupling Prolog and databases*

Coupling Prolog and a relational database leads to an impedance mismatch between these two systems. The Prolog inference engine applies a depth-first search strategy on the facts stored in main memory, and uses backtracking. The database system is tuned to retrieving sets of tuples from secondary memory. When the Prolog engine encounters a database predicate in the program this leads to a single-tuple retrieval from the database, thereby reducing the database to a simple storing device. To overcome this impedance mismatch, several techniques have been developed to come to a more efficient program execution.

A first technique is to retrieve a set of tuples corresponding to a database predicate from the database and store it in a sort of cache; this set can, for instance, be a page on disk. Whenever the Prolog inference engine asks for a new tuple, first this cache is searched, and whenever it is empty a new retrieval request is sent to the database.

A second technique is to modify the original Prolog program. By rewriting the Prolog rules one may ensure a more efficient use of the database. Changing the order of the predicates in a rule may lead to queries that are more selective, and to a reduction of the amount of tuples to be transported from the database to the Prolog system. For instance, instead of sending a query concerning a single database predicate to the database, one can send a conjunction of predicates to the database [16]. This amounts to computing a join in the database and sending the results to the Prolog system.

A third technique is to cache both data and queries. Whenever a database predicate is queried, not only the data retrieved is stored in the Prolog system, but also the query that was executed. This may help to discover that a query has already been solved once — for instance, in the context of another rule — and that the result is still available. Therefore, the database does not have to be approached again. This technique may further be improved by the use of subsumption. Whenever a query is posed that has already been solved in a more general form (by having variables instead of constants as arguments) there is no need to approach the database, since the result has already been retrieved and has been stored in Prolog's main memory [11].

Although several systems have been built using the techniques mentioned above (for an overview see [14]), coupling Prolog to a database does not lead to very efficient systems. The database system is mainly used as a large data storage device, and a caching and storing mechanism for relevant data is built inside the Prolog system. Especially for recursive queries, this leads to an inefficient way of computation. A quite different approach is to translate Prolog rules directly into Relational Algebra definitions, and solve recursion by unfolding the defining rules until no more new results are obtained [17]. This overcomes some of the problems, but has as a drawback that recursion is still handled outside the database system.

2.2 Deductive databases

In deductive databases there are two main optimization strategies for recursive programs. One can rewrite a logic program into one that, hopefully, will lead to a better performance; this is a purely syntactic way. Or one can develop an evaluation strategy that improves performance by executing queries/subqueries in a suitable order; this is a more semantic way. Both strategies focus on queries that have a variable bound to a constant value. In Relational Algebra this amounts to a selection on the eventual result. Let us first concentrate on the syntactic rewriting strategy.

2.2.1. Magic sets

One of the most popular rewriting methods in a logic context is that of ‘magic sets’ [5, 39]. It is based on the idea of sideways information passing. This means that by first solving a predicate with some variables bound, one can obtain bindings for other variables; which in their turn may transfer to other predicates in the same rule. The magic set method works by inserting a ‘magic’ predicate in the body of a recursive rule, which models the bound variables to be transferred. This may restrict the computation of the recursive predicate to a small part of the data. It is illustrated by the following example.

Example 1. Given the following ancestor program

```
anc(X, Y):−par(X, Y) .
anc(X, Y):−anc(X, Z), par(Z, Y) .
? anc(a, Y).
```

the result of applying the magic set method is the following program:

```
magic(a) .
anc(X, Y):−par(X, Y).
anc(X, Y):−magic(X), anc(X, Z), par(Z, Y) .
? anc(a, Y).
```

One may see that when a left-to-right order of evaluation of the predicates is assumed, the evaluation of $\text{anc}(X, Z)$ is limited to the parents, grandparents, great-grandparents, etc. of a ; this is the so-called ‘cone’ of a .

Translated into algebraic terms, the method turns out to lead to a repeated computation of a semijoin. The introduction of the *magic* predicate results in the subsequent computation of:

$$\sigma_{1=a} \text{PAR}, \text{PAR} \bowtie_{1=2} (\sigma_{1=a} \text{PAR}), \text{PAR} \bowtie_{1=2} (\text{PAR} \bowtie_{1=2} (\sigma_{1=a} \text{PAR})), \text{etc.}$$

instead of the complete relation *ancestor*. Although the magic set method may lead to an improvement of the original program, it does not necessarily lead to a more efficient program. For some queries, the binding of a variable to a constant value does not allow a program to be rewritten in such a form that it can be computed more efficiently. The binding of the variable will, in that case, not restrict the computation; the rewritten program will then be larger (in terms of rules and predicates), but not more efficient.

A variant of the magic set method is called magic counting [35]. Besides restricting the computation to the ‘cone’ of a , it also stores the distance of an ancestor to the root of the tree, which is a . A major drawback of this approach is that it only works for non-cyclic

relations, and cannot handle situations where there are several paths between any two points. This makes the method only applicable for relations with a tree structure, and limits its applicability. Now that syntactic rewriting of programs has been discussed, let us study evaluation strategies for logic programs.

2.2.2. Query/subquery

An evaluation strategy for recursive programs in a logic context that has received a lot of attention is called the query/subquery approach [42, 43]. Its purpose is to evaluate a logic program in such an order that the computation is most efficient. By efficiency one here means that the total amount of data used in the computation (the number of tuples) is minimized.

The query/subquery approach assumes the presence of a selection function, which indicates the specific predicate in the body of a rule to be considered next for evaluation. This selection function itself is not specified, but can be an arbitrary function: most instantiated predicate, left-most predicate, etc. If the chosen predicate is a database predicate the answers are retrieved from the database (set-at-a-time), but if it is an intensional predicate — defined by a rule — it has to be evaluated. This is done by recursively applying the query/subquery approach on the rule defining the predicate. At each stage of the evaluation process one has to keep track of the solutions and subqueries generated thus far.

The main benefit of the approach is that it focuses on relevant data. There are two aspects to this issue. The first aspect is that a tuple that is produced during evaluation of the program should be used only once in the further evaluation process, no matter how many times it is produced. This is equivalent to taking a union of the produced tuples in Relational Algebra: when a tuple is produced twice it will be present only once in the intermediate result relation that is used in the subsequent computations. The second aspect is that values that are produced for variables are used only once in the further evaluation process. This approach is illustrated by the following example.

Example 2. Let us assume the following rule to be part of the program:

$$\text{pred}(Z) :- p(X, Y), r(Y, Z).$$

When we assume a left-to-right order of evaluation, the evaluation of the predicate p will provide values for Y that can be used to restrict the evaluation of r . The values of X have no influence on the evaluation process. In the query/subquery approach, great care is taken to use each value for Y only once in the evaluation of r . Hence, if p is a base relation with tuples (a, c) and (b, c) , there will be only one evaluation process for the subquery $r(c, Z)$.

From this description, it may be clear that this technique is equivalent to applying a semijoin in Relational Algebra. Using the same example, the equivalent Relational Algebra strategy would be to compute $R \bowtie_{1=2} P$.

A first problem with the query/subquery approach is that the selection function, which determines the performance of the approach, is not specified. Actually, this selection function determines the join order of the relations in the rule (without, however, the possibility of specifying which join algorithm should be used). A second problem is that the approach leads to a complex overhead for keeping track of queries/subqueries and answers. It is not clear if the database system can be used for storing and manipulating this complex administration. A third problem is that the approach leads to ‘single predicate’ queries to the database, instead of joins between several relations. And since at least part of the answers is

probably stored outside the database system, it will encourage the use of a nested-loop join method. Whereas the database system itself would perhaps have been able to use a more efficient join method, such as sort-merge.

2.2.3. Other approaches

Several other methods besides the ones mentioned in the foregoing have been developed. In [15] the translation of a logic program into an iterative relational program is studied. This method only works for a very restricted class of rules — strictly right or left linear rules. (Terminology is treated later, in Section 4.) One of the first evaluation strategies developed is described in [20]. It only works for a restricted class of rules. In [38] a framework is developed for studying the evaluation of recursive programs. Several optimization strategies have been described that were developed in the context of that framework [33, 41]. For a good overview of the large number of strategies that have been developed over the years, see [14] or [34].

In [6] the performance of several of these strategies is compared. In this comparison the data is assumed to be non-cyclic, and duplicate paths between any two points are excluded if they do not have the same length. This results in relations that mainly model trees and lists; the results of the performance comparisons are, therefore, not easily generalizable and have only limited value. The criteria on which these strategies are compared are rather surprising in a relational environment. They measure the number of successful firings of rules, which leads, for instance, to choosing the size of the result relation as a measure of complexity for the join or selection operation. Whereas in a relational environment the number of page accesses is usually taken as the relevant factor to assess the suitability of an optimization strategy.

2.3 Algebraic optimization

Although recursive query optimization has received most of its attention in a logic context, it has also been studied in an algebraic one. By recognizing that Datalog-like programs can be translated into equations of positive Relational Algebra (Relational Algebra without the set-difference operator), one may perform this translation and optimize the resulting algebraic equations. This approach is taken in [12] and [13]. Here, a Datalog program is translated into a system of algebraic equations, and optimization techniques are studied for queries containing a constant value (i.e. a selection on the result).

In this approach, first the system of equations is put into some normal form. This normal form allows the detection of common subexpressions, which are factored out to reduce the amount of computation. Given the remaining system of equations, one can identify the strongly connected components and determine an order of evaluation of these components. A strongly connected component corresponds to a subsystem of equations that are mutually recursive. It can, therefore, be solved independently once the results of the other components it uses are known.

For the evaluation of strongly connected components, two optimization strategies have been developed. They are called reduction of variables and reduction of constants. Reduction of variables is a way of propagating selections on the result (a variable in the system of equations) to base relations (or constant relations) that are used in the equations for the variable. This is similar to pushing selections down into a query tree in the case of non-recursive queries. Reduction of constants is a more intricate technique than reduction of variables. It uses the idea of sideways information passing by introducing a repeated semijoin that avoids using a complete base relation in the evaluation process. The next example illustrates this optimization technique.

Example 3. Given the following Datalog program:

$$r(X, Y) :- c_1(X, Y) .$$

$$r(X, Y) :- r(X, Z), c_2(Z, Y) .$$

where c_1 and c_2 denote base relations, its translation in algebraic equations is given by the following:

$$R = N_1 \cup C_1$$

$$N_1 = R \circ C_2$$

The notation $R \circ C_2$ stands for $\pi_{1,4}(R \bowtie_{2=1} C_2)$; the operator is called composition operator. When we consider a selection on the variable R , for instance, the second attribute should be equal to a , the system is translated into the following system of marked equations:

$$(R : 2) = (N_1 : 2) \cup (C_1 : 2) .$$

$$(N_1 : 2) = R \circ (C_2 : 2)$$

$$R = N_1 \cup C_1$$

$$N_1 = R \circ C_2 .$$

The notation $(R : 2)$ means that there is a selection on the second attribute of R . As we can see, the system is now larger than it was before; but by a reduction of constants it is made more efficient than the original one. The base relation C_1 is used in its marked form in the first equation. In the third equation it is used entirely, but because it is only needed in the second equation for a join with C_2 , it may be reduced by a semijoin with C_2 . Similarly, C_2 may be reduced. This results in the following equations, in their executable Relational Algebra form.

$$C'_1 = \sigma_{2=a} C_1 \cup C_1 \bowtie_{2=1} C'_2$$

$$C'_2 = \sigma_{2=a} C_2 \cup C_2 \bowtie_{2=1} C'_2$$

These reduced constants may now be used instead of the original ones.

It may be noticed that there is some resemblance between these reductions and the ‘magic sets’ method. In fact, reduction of variables turns out to compute a generalized cone of a [14].

Once the algebraic rewriting methods as described have been used, the resulting system of equations has to be evaluated. This done by iterating over this set of equations until a fixpoint is reached. For this iteration one can use a simple iteration (a naive approach), or an iteration that uses in each step only those newly produced tuples that can really lead to any new results (a semi-naive approach) [12].

2.4 Conclusions

In this section we showed that recursive queries can be optimized in a logic, as well as in an algebraic context. Techniques used in deductive databases, such as ‘magic sets’, resemble well-known algebraic optimization techniques. Sideways information passing and focusing on relevant data in a logic approach, mimic semijoin techniques and set-oriented processing in an algebraic approach. Although logic is a good formalism for expressing recursion, it often encourages one to take a tuple-oriented approach—or even way of thinking—when studying optimization of queries. This is mainly due to the large influence of resolution-based computation, especially Prolog (although one tries to avoid this in most deductive database approaches, by adopting different compilation techniques for clauses). Since optimization can also be done in an algebraic way, we believe this to be preferred. It naturally leads to a set-oriented view on optimization, which is most efficient in a relational database context. Moreover, there exist many well-known query optimization techniques in the context of relational databases.

The overall goal is to support query optimization in a data and knowledge base management system (DKBMS). In such a system both recursive and non-recursive queries will have to be supported. It is to be preferred to have *one* query optimizer that handles both cases. And since query optimization is well-developed in the context of relational databases, this seems a good starting point. Moreover, when the database system to be used is a *distributed* one, query optimization becomes even more important. And while there exist well-known techniques for query optimization in distributed databases in a relational environment, this issue has not yet emerged in the context of deductive databases. Since we foresee a huge increase in the use of distributed database systems (especially to enable parallel computation), it is almost mandatory to start from a relational query optimizer and optimize queries in an algebraic way.

3. Expressing recursion in algebra

In Section 1 we argued that it is to be preferred to have one query optimizer for all optimizations. This query optimizer should optimize queries from fourth-generation languages such as SQL and from logic languages such as PRISMAlog. Therefore, we need an intermediate language with the same expressive power as SQL and PRISMAlog that can be optimized and executed on a relational database. For this intermediate language we propose eXtended Relational Algebra (XRA). This language is essentially Relational Algebra extended with a fixpoint construct called μ -calculus expression.

In this section we describe μ -calculus expressions, which allow us to express arbitrary recursion in an algebraic way. In Section 5 we describe how to optimize these expressions by means of a rewriting mechanism. This rewriting involves removing the fixpoint construct and introducing transitive closure operations on relational algebra expressions. Note that XRA thus serves two purposes: it is an intermediate query language, and it is a language to express executable schedules on a relational database. This is illustrated in Fig. 1; on the level of the query language we have μ -calculus expressions to express recursion, on the level of the schedule language we have transitive closure operations and an iteration construct, and to

query language:	Relational Algebra, μ -calculus expressions	↓ rewriting strategy
schedule language:	Relational Algebra, Transitive Closure, Iteration	

Fig. 1. The two levels in XRA.

transform one into the other we have a query optimizer that is equipped with a rewriting strategy.

3.1 μ -calculus expressions

To be able to handle recursive programs in a relational DBMS, it should first of all be possible to express them in a relational language. A construct is, therefore, necessary to express recursion. It should have a precise syntax and a precise semantics. For this purpose a new concept is introduced in this section: the μ -calculus expression. It is a least fixpoint construct that enables the expression of arbitrary recursive programs.

The μ -calculus is a concept adopted from theoretical computer science [36], where it is used to express the semantics of sequential programs. Informally speaking, a μ -calculus expression consists of a μ followed by a variable and a bracketed part consisting of several relational expressions that are separated by unions. The syntax of μ -calculus expressions is shown in Fig. 2. To have a recursive program, the $\langle \mu\text{-variable} \rangle$ should appear at least once in the $\langle \text{union of expressions} \rangle$. One may compare a μ -calculus expression with the definition of a recursive function. The union of expressions being the body of the function and the μ -variable being the function identifier, which is used inside its own body. Note that since the union operation has its normal Relational Algebra semantics, the expressions in the union of expressions should be union compatible.

Now that the syntax of a μ -calculus expression is explained, its semantics can be discussed.

Consider the following example expression:

$$\mu X[B \cup \pi_{\text{attributes}}(A \bowtie_{\text{conditions}} X)],$$

where *attributes* stand for the set of attributes one is projecting on and *conditions* for the conditions associated with the join. (Note that a join is an abbreviation for a selection on a Cartesian product, and, thus, confirms to the syntax.) The *meaning* of this expression is obtained by an iterative substitution. First the empty relation is substituted for the variable X , and the result of the $\langle \text{union of expressions} \rangle$ is computed. Then, in each subsequent step the result from the previous step is substituted for the variable X . Hence, when the expression would be evaluated in this naive way, the generated output would be as shown in Table 1. In this figure A^1B represents $\pi(A \bowtie B)$, similarly A^2B represents $\pi(A \bowtie \pi(A \bowtie B))$, etc., all with the appropriate subscripts. As may be seen, the output after n iteration steps is $B \cup \bigcup_{i=1}^{n-1} (A^i)B$. Remember that the result of $\pi(A \bowtie X)$ should be union compatible with the relation B , which is why the projection is necessary.

Now that the syntax and semantics of μ -calculus expressions have been introduced, we may make the following observations. A relational database may be viewed as a finite lattice, where the partial order is defined by the set-inclusion relation. A μ -calculus expression then

$$\begin{aligned} \langle \mu\text{-calculus expression} \rangle &::= \mu \langle \mu\text{-variable} \rangle [\langle \text{union of expressions} \rangle]. \\ \langle \text{union of expressions} \rangle &::= \langle \text{expression} \rangle \\ &\quad | \langle \text{expression} \rangle \cup \langle \text{union of expressions} \rangle. \\ \langle \text{expression} \rangle &::= \langle \mu\text{-variable} \rangle \\ &\quad | \langle \text{relational variable} \rangle \\ &\quad | \pi \langle \text{expression} \rangle \\ &\quad | \sigma \langle \text{expression} \rangle \\ &\quad | \langle \text{expression} \rangle \times \langle \text{expression} \rangle \\ &\quad | \langle \mu\text{-calculus expression} \rangle. \end{aligned}$$

Fig. 2. Syntax of μ -calculus expression.

Table 1. Semantics of μ -calculus expression.

Iteration step	Output
1	B
2	$B \cup A^1 B$
3	$B \cup A^1 B \cup A^2 B$
$\vdots$	
n	$B \cup \bigcup_{i=1}^{n-1} (A^i) B$

may be viewed as a transformation on this lattice. Since all operations in a μ -calculus expression are monotone, this transformation is guaranteed to have a least fixpoint [30]. Moreover, since all relations in the database are finite, this least fixpoint is a finite set. The iterative substitution describing the semantics of a μ -calculus expression thus has a finite number of steps; and if the meaning of a μ -calculus expression would indeed be computed by a naive iteration as described by the semantics, this computation would be guaranteed to terminate.

Note, if none of the relational expressions in the μ -calculus expression contains a join it concerns a trivial μ -calculus expression. The join is the only really significant way of generating new tuples in an iterative process. It describes how to combine two separate paths into a new one. If a relation is viewed as a representation of a graph, with tuples denoting the arcs, the join operation describes how to concatenate two paths to create a new one. Trivial μ -calculus expressions are not considered in the sequel, they should be expressed in standard Relational Algebra.

Example 4. Let us take as an example the relation AIR with attributes *departure*, *arrival*, *dep_time*, and *arr_time*. This relation contains information about possible flights and their respective departure and arrival times. Only direct flights are considered. A possible extension of the relation AIR is shown in Table 2. To express all possible connections by air, the following μ -calculus expression may be written down:

$$\mu X[\text{AIR} \cup \pi_{1,6,3,8}(\text{AIR} \bowtie_{2=1 \wedge 4 < 3 \wedge 1 \neq 2} X)].$$

This expression results in the relation shown in Table 3.

We may make the following observations about the example expression. The first join condition states that *arrival* of the first tuple should be equal to *departure* of the second, and thus a connection is possible. The second join condition states that one has to arrive before one can depart again. The third join condition states that cycles should be avoided: one should not depart from and arrive in the same place. In every iteration step a tuple representing a direct flight is joined with a tuple representing a multi-leg flight, and because

Table 2. Relation AIR

Departure	Arrival	dep_time	arr_time
Amsterdam	Paris	9.00	10.30
Amsterdam	Rome	8.30	10.30
Paris	Tokyo	11.00	20.00
Rome	Seoul	12.00	18.00
Rome	London	11.30	13.30
Seoul	Tokyo	19.00	22.30
London	Amsterdam	13.45	14.50

Table 3. Result of example expression.

Departure	Arrival	dep_time	arr_time
Amsterdam	Paris	9.00	10.30
Amsterdam	Rome	8.30	10.30
Amsterdam	Tokyo	9.00	20.00
Amsterdam	Seoul	8.30	18.00
Amsterdam	Tokyo	8.30	22.30
Amsterdam	London	8.30	13.30
Paris	Tokyo	11.00	20.00
Rome	Seoul	12.00	18.00
Rome	Tokyo	12.00	22.30
Rome	London	11.30	13.30
Rome	Amsterdam	11.30	14.50
Seoul	Tokyo	19.00	22.30
London	Amsterdam	13.45	14.50

of the join condition it is not possible to generate a cycle. Since in every step a union is taken between the new result tuples and all the previous result tuples, a path from x to y will never be added to the solution if was already part of the result.

Note that in Relational Algebra there is no need to restrict oneself to binary relations and equijoins, as is usually done in logic approaches. In Relational Algebra comparisons are part of the join condition, and the optimizer knows how to handle them. In Datalog, predicates have to be introduced for comparisons between variables. The selection function that chooses which predicate to evaluate (see Section 2.2.2) should also take into account these comparisons, and use them if possible as part of a join condition. This aspect will complicate the development of selection functions, and has not yet been discussed in papers on that subject.

For the sake of simplicity, the example introduced above just expressed a simple transitive closure problem. It is, however, possible to express arbitrary recursive programs with μ -calculus expressions. Because the result of a μ -calculus expression is a normal relation it may be used in other relational expressions. In contrast to the normal Relational Algebra, relational variables are allowed to stand for relational expressions that are evaluated during run-time. These relational variables are introduced for convenience only, and are called *virtual relations*. For example, we can introduce a virtual relation M to stand for the expression that has just been discussed.

$$M = \mu X[AIR \cup \pi_{1,6,3,8}(AIR \bowtie_{2=1 \wedge 4 < 3 \wedge 1 \neq 2} X)] .$$

Since virtual variables may also be used to stand for a recursive definition, they allow an easy formulation of more complex recursive problems. For instance, the following system consists of two recursive definitions that use each other in their definition.

$$S = \mu X[A \cup \pi(B \bowtie X) \cup \pi(C \bowtie T)]$$

$$T = \mu Y[D \cup \pi(A \bowtie Y) \cup \pi(E \bowtie S)] .$$

Note that the use of virtual relations does not have any influence on the meaning of the expressions. The above-mentioned expressions can easily be rewritten into a normal μ -calculus expression without any virtual relations. To achieve this, a virtual variable V_i has to be chosen and all virtual variables used in its definition have to be replaced by their

respective definitions. This process should then be iterated over the resulting expression. Care should be taken to replace a virtual variable that appears in the scope of its mu-variable by this mu-variable, and not by its definition. When the virtual variable S from the foregoing expressions is chosen as V_i , this results in the following rewritten expression.

$$S = \mu X[A \cup \pi(B \bowtie X) \cup \pi(C \bowtie (\mu Y[D \cup \pi(A \bowtie Y) \cup \pi(E \bowtie X)])]] .$$

Let it be clear that in this section only descriptions of the syntax and semantics of μ -calculus expressions have been given, together with some illustrative examples. We have only discussed the query language part up to now (see Fig. 1). This section did, however, *not* describe how the result of a μ -calculus expression should be computed. Although the description of the semantics provided a way of computing the result, this is a naive way that, in general, will not be very efficient. The optimization strategy that rewrites the query language part into the schedule is discussed later, in Section 5.

4. Types of recursion

The μ -calculus expression introduced in the previous section enables the formulation of arbitrary recursive programs. Let us now consider a classification of recursive rules. This classification allows us to distinguish two types of recursion: *regular* and *non-regular*. Of these two types, regular recursion is by far the most important.

The terms regular and non-regular recursion stem from formal languages. They can be illustrated by writing down corresponding grammar production rules for a μ -calculus expression. The mu-variable of a μ -calculus expression represents a non-terminal symbol, whereas the base relations represent terminal-symbols. There is a direct correspondence between a μ -calculus expression and a grammar production rule. For a μ -calculus expression, as many grammar production rules as there are expressions in the $\langle \text{union of expressions} \rangle$ are created. Each of these production rules has as left hand side the mu-variable of the μ -calculus expression, and as right hand side the $\langle \text{expression} \rangle$ with omission of the relational operators.

Example 5. Let us consider the following example expression:

$$\mu X[B \cup \pi_{\text{attributes}} \sigma_{\text{conditions}}(A \times X)] .$$

The corresponding grammar production rules are:

$$X \rightarrow B$$

$$X \rightarrow AX .$$

As we may see, there is one production rule for the expression B and one for the expression $\pi\sigma(A \times X)$.

A μ -calculus expression is called regular recursive if the corresponding grammar production rules can be transformed into a regular grammar. In the foregoing example the rules already form a regular grammar, so the corresponding μ -calculus expression is regular. A query is called regular recursive if the grammar production rules corresponding to the set of virtual relations the query refers to can be transformed into a regular grammar. In the sequel all regular expressions are assumed to be in a right-recursive form; this is no limitation, since right-recursive and left-recursive forms are equivalent (be it in Datalog or XRA).

Example 6. An example of a non-regular recursive μ -calculus expression is the following:

$$\mu X[(A \times B) \cup \pi\sigma(A \times X \times X)] .$$

The corresponding grammar production rules are:

$$X \rightarrow AB$$

$$X \rightarrow AXX .$$

The grammar production rules corresponding to this μ -calculus expression cannot be transformed into a regular grammar, because the non-terminal symbol occurs twice in the body of a rule.

A subclass of regular recursive rules that is often considered, especially in the context of logic optimization, is that of *linear* recursive rules. Linear recursive rules are rules with exactly one recursive predicate (namely the same as the head of the rule), and no other recursive predicates. Moreover, there is only one defining rule containing a recursive predicate allowed for each head. This means that in the system of corresponding grammar production rules, for every non-terminal only one production rule is allowed that contains another non-terminal, and this non-terminal should be equal to the non-terminal defined by the rule itself. In fact, the first example of this section was an example of a linear recursive rule. There is only one production rule for X that contains a non-terminal, and this non-terminal is equal to X . In [15] the term regular rules was used to denote these linear rules, however, this is not in line with common terminology in formal languages.

Note that because of the associativity and commutativity of the join operation, a linear recursive rule can always be put into a regular right-recursive form. For instance, the expression $A \bowtie X \bowtie B$ may be written as $A \bowtie B \bowtie X$. However, this is not guaranteed to lead to a more efficient computation; the joins may degenerate into Cartesian products.

Another notion that is often used in the context of recursive rules is that of *mutual recursion*. A collection of rules is said to be mutually recursive if they are used in each other's definition. Recollect from the previous section the following example.

$$S = \mu X[A \cup \pi(B \bowtie X) \cup \pi(C \bowtie T)]$$

$$T = \mu Y[D \cup \pi(A \bowtie Y) \cup \pi(E \bowtie S)] .$$

These expressions are mutually recursive because S and T are used in each other's definition. The corresponding grammar rules for this example are:

$$X \rightarrow A$$

$$X \rightarrow BX$$

$$X \rightarrow CY$$

$$Y \rightarrow D$$

$$Y \rightarrow AY$$

$$Y \rightarrow EX .$$

As may be seen from the grammar production rules, this is also an example that falls into the category of regular recursion. To conclude this section, let us study one more example of a regular, mutually recursive program.

Example 7. Imagine a sparse public transport network in which it is possible to take either the train or the bus, but not all cities can be reached directly through one means of transport. Hence, we have to take a, probably alternating, sequence of train and bus rides. We thus have relations $\text{TRAIN}(\text{departure}, \text{arrival}, \text{dep_time}, \text{arr_time})$, $\text{BUS}(\text{departure}, \text{arrival}, \text{dep_time}, \text{arr_time})$ and $\text{TRANSFER}(\text{city})$. The relation TRANSFER represents that one can transfer from train to bus and *vice versa* in the named city. For the sake of the example, the relation TRANSFER has a simple structure. One can imagine it to consist of more attributes, for instance, to indicate the minimum time that is required for the transfer. The μ -calculus expressions that model this problem are the following.

$$Q_1 = \mu X[\text{TRAIN} \cup \pi_{1,6,3,8}(\text{TRAIN} \bowtie_{2=1 \wedge 1 \neq 2 \wedge 4 < 3} X) \cup \pi_{1,6,3,8}(\text{TC} \bowtie_{2=1 \wedge 1 \neq 2 \wedge 4 < 3} Q_2)]$$

$$Q_2 = \mu Y[\text{BUS} \cup \pi_{1,6,3,8}(\text{BUS} \bowtie_{2=1 \wedge 1 \neq 2 \wedge 4 < 3} Y) \cup \pi_{1,6,3,8}(\text{BC} \bowtie_{2=1 \wedge 1 \neq 2, 4 < 3} Q_1)]$$

$$\text{TC} = \pi_{1,2,3,4}(\text{TRAIN} \bowtie_{2=1} \text{TRANSFER})$$

$$\text{BC} = \pi_{1,2,3,4}(\text{BUS} \bowtie_{2=1} \text{TRANSFER}).$$

In this example Q_1 models all trips that start with a train ride, and are followed by an arbitrary number of train rides (the join with X) and possibly a transfer to a bus trip (the join with Q_2). Similarly, Q_2 models all trips that start with a bus trip.

Now that a classification of recursive rules has been given, the problem of efficiently solving recursive programs will be studied. In the next section a rewriting strategy is described for regular recursive programs. This strategy enables an efficient computation of, for instance, the last example, which was regular mutually recursive.

5. Algebraic optimization of regular recursion

In Section 3 we discussed the query language level of XRA and showed how to express recursion in algebra by means of a fixpoint construct called μ -calculus expression. Now that we have a construct in our language to express recursion, we can develop optimization strategies that deal with such a construct.

Although μ -calculus expressions provide a convenient way of expressing recursion, it is rather hard to see the meaning of a set of mutually recursive rules. Therefore, in Section 5.1 a graphical way of depicting regular recursive queries, called transition graphs, is introduced. This may increase the insight in the answers a query should produce. After that, in Section 5.2 the evaluation of mutually recursive rules is studied in the context of these transition graphs. In Section 5.3 an algorithm is described that rewrites regular mutually recursive rules. This rewriting algorithm is based on the evaluation in the context of transition graphs. It produces a sequence of relational expressions and transitive closure operations, in which duplicate computations are avoided as much as possible. This rewriting algorithm takes us

from the query language level to the schedule language level, as was shown in Fig. 1. It is this rewriting algorithm that incorporates the major optimizations: computing common expressions only once, introducing transitive closure, and reducing the number of join operations. Finally, in Section 5.4 we discuss extra optimization possibilities that arise when we are dealing with mutually recursive systems that are not fully connected.

5.1 Transition graphs

Since we suppose that the system of regular mutually recursive rules is already in a right-recursive form, we can depict it directly by a finite automaton or by an equivalent graph formalism. Here the graph formalism, called a *transition graph*, is used. In this transition graph, the grammar production rules corresponding to the queries are depicted. For every non-terminal symbol there is a corresponding node in the graph; one of these nodes is designated as the start node—this can be any node, depending on the query at hand. For convenience, there is also a node in this graph representing the end of the computation; this node is called the end node. The nodes in the transition graph are connected by arcs, denoting the terminal symbols in the grammar production rules. For every grammar rule there is an arc in the transition graph. The formal definition of a transition graph is as follows.

$$G = (V, A), V = \{0, 1, \dots, n\}, n \text{ the number of non-terminal symbols,}$$

$$A = \{(v_1, v_2, l) \mid v_1, v_2 \in V, l \in L\},$$

$$L = (\pi, \sigma, R) \text{ with } R \in \text{relational expressions.}$$

The translation from a μ -calculus expression and its corresponding grammar production rules to a transition graph is straightforward, and not shown here.

As we may see, the labels alongside the arcs consist of a projection-list, a selection-list, and a relational expression consisting of base relations. Every possible path from the start node to the end node depicts a set of solutions to the query. The expression giving rise to this set of solutions can be constructed from the labels alongside the arcs that are traversed in this path. This construction is as follows. Suppose there is a path from start node n_1 to end node n_3 , consisting of arcs (n_1, n_2, l_1) and (n_2, n_3, l_2) , where $l_1 = (\pi_1, \sigma_1, R_1)$ and $l_2 = (\pi_2, \sigma_2, R_2)$. The resulting expression is then formed by $\pi_1 \sigma_1 (R_1 \times (\pi_2 \sigma_2 R_2))$. Given the construction of the graph, this is equivalent to $\pi_1 (R_1 \bowtie (\pi_2 \sigma_2 R_2))$.

For convenience, the projection-list and selection-list will not be shown in the graphical representation of transition graphs. Also, when the expressions corresponding to a transition graph are discussed, only projection and join symbols are used, without the appropriate subscripts and conditions. This will help the reader to focus on the important issues, without being distracted by an overly complex notation. In the graphical representation of transition graphs the start node is always numbered 1 and the end node is always numbered 0. Since every path from the start node to the end node gives rise to a set of solutions, the answer to a given query is found by taking the union of all possible paths from the start node to the end node. This operation is known as a Kleene closure in the context of regular expressions [21].

The graphical representation of the transition graph corresponding to the example that we have encountered a few times before is shown in Fig. 3. From this graph we may see that solutions are obtained by computing, for instance, the result of the expressions B and $\pi(A \bowtie B)$. In Fig. 4, the transition graph that corresponds with the public transportation problem from Example 7 is depicted. Again, we may find solutions by traversing the various paths in the graph.

Fig. 3. Transition graph for example expression.

Fig. 4. Public transportation problem.

In the next section the evaluation of transition graphs is studied. It will turn out that by transforming these transition graphs—i.e. rewriting the corresponding algebraic equations—a more efficient computation may be obtained than by a naive evaluation strategy. Although strategies from formal languages [21] might be used for evaluating transition graphs, this would result in many redundant computations that are avoided by the transformation process to be described.

5.2 Evaluation of a transition graph

We have seen that in a transition graph, every path from start node to end node corresponds with an expression that yields a set of solutions to the query. The union of all paths from start node to end node corresponds with the complete solution to the query. For the transition graph shown in Fig. 3, we may thus conclude that the complete set of solutions is generated by the following expression.

$$B \cup A^1 B \cup A^2 B \cup \dots = B \cup \bigcup_{i=1}^{\infty} A^i B.$$

As before, $A^1 B$ denotes $\pi(A \bowtie B)$, $A^2 B$ denotes $\pi(A \bowtie \pi(A \bowtie B))$, etc. Since we have already seen that because of the finiteness of the relations in the database and the monotonicity of the operations a least fixpoint for the query exists, we can be sure that there is some finite n such that the solution is equivalent to the following expression.

$$B \cup \bigcup_{i=1}^n A^i B = B \cup (A^T \bowtie B).$$

Here, A^T denotes the transitive closure $A \cup \pi(A \bowtie A) \cup \dots$ of A . This allows us to transform the transition graph for this problem, resulting in the graph shown in Fig. 5. As can be seen, the cycle from node 1 to node 1 has been replaced by a new type of arc. This is a graphical convenience to indicate that this arc may only be traversed once in each path. Since it is but a graphical convenience, it does not change the definition of a transition graph, and does not influence the algorithm to be developed. From the transformed transition graph we immediately see that the solution is given by the expression $B \cup \pi(A^T \bowtie B)$.

Fig. 5. Transformed transition graph.

The transition graph in *Fig. 5* is the simplest graph depicting recursion that one can imagine. The solution consists of the two paths from start node to end node, and the evaluation is straightforward. When we study the transition graph from *Fig. 4* we see that it is much more complex. It is difficult to come up with all possible paths from start node to end node. This is due to the three cycles in the graph. We have already seen that a cycle from node i to itself may be replaced by a special arc, with the transitive closure of the old label as its new label. The following step is to find a way to remove a cycle between differently numbered nodes. This may be done by removing the upward going arc (from the lower numbered node to the higher numbered node), which disconnects the higher numbered node. Just removing the arc, however, will cause the loss of several paths, and thus solutions. The remedy for this is to incorporate the paths that are lost in the labels of arcs that can still be traversed. This is done by incorporating in label l_{ij} all paths that went from node i to node j via the removed node k in the old transition graph. In this way, no information is lost. This process is illustrated in the next example.

Example 8. Let us take the transition graph from *Fig. 4*, which depicts the public transportation problem, as the graph to be transformed. For brevity, the label alongside the arc from node i to node j is denoted by l_{ij} in this example. First the cycle from node 2 to itself is removed, by taking the transitive closure of the old label:

$$l'_{22} := l_{22}^T.$$

This results in the transition graph shown in *Fig. 6*. The next step is to remove the cycle between nodes 1 and 2. This is done by removing the arc from node 1 to node 2. Hence, some of the labels have to be adjusted. Label l_{11} — denoting all paths that go from node 1 to node 1 — now has to incorporate all paths that go from node 1 via node 2 back to node 1 in the graph from *Fig. 6*. Similarly, label l_{10} now has to incorporate all paths that go from node 1 via node 2 to node 0 in the transition graph from *Fig. 6*. This leads to the following expressions:

$$l'_{11} := (l_{11} \cup \pi(l_{12} \bowtie \pi(l'_{22} \bowtie l_{21})) \cup \pi(l_{12} \bowtie l_{21}))$$

$$l'_{10} := l_{10} \cup \pi(l_{12} \bowtie \pi(l'_{22} \bowtie l_{20})) \cup \pi(l_{12} \bowtie l_{20}).$$

And since the cycle in node 1 is the only one remaining, we may take the transitive closure of the label and remove the cycle:

$$l''_{11} := l'_{11}^T.$$

This results in the transition graph shown in *Fig. 7*. Node 2 is now unreachable from node 1 and this graph is, therefore, equivalent to the one shown in *Fig. 5*.

Fig. 6. Graph after first transformation step.

Fig. 7. Graph after second transformation step.

```

current node := highest numbered node;
remove local cycle (arc from current node to current node);
repeat
  current node := currentnode - 1;
  remove upward going arcs;
  incorporate lost paths into labels;
  remove local cycle
until currentnode = startnode

```

Fig. 8. Outline of rewriting algorithm.

In the following section, an algorithm is described that uses the techniques just illustrated to rewrite regular mutually recursive rules.

5.3 Rewriting algorithm

The technique used in the previous section to evaluate the public transportation problem — a good example of regular mutual recursion — may be generalized for arbitrary complex transition graphs. First, the local cycle (i.e. the arc from a node to itself) in the highest numbered node is removed — provided, of course, that there is such a cycle. Then, the highest numbered node but one is considered. For this node, all upward going arcs are removed, and any paths that are lost are incorporated in the appropriate labels of arcs that can still be traveled. Finally, the local cycle in this node is removed. This process is repeated until the end node is the only node that can still be reached from the start node. The outline of this algorithm is shown in Fig. 8. The rewriting algorithm is described in full detail later on, but it seems a good idea to study one more example first. This may enable us to discover some optimizations that can improve the algorithm.

Example 9. Let us take as an example the following regular mutually recursive rules, where R_1 is the rule corresponding with the query:

$$R_1 = \mu X[l_{10} \cup \pi(l_{11} \bowtie X) \cup \pi(l_{12} \bowtie R_2) \cup \pi(l_{13} \bowtie R_3)]$$

$$R_2 = \mu Y[l_{20} \cup \pi(l_{21} \bowtie R_1) \cup \pi(l_{22} \bowtie Y) \cup \pi(l_{23} \bowtie R_3)]$$

$$R_3 = \mu Z[l_{30} \cup \pi(l_{31} \bowtie R_1) \cup \pi(l_{32} \bowtie R_2) \cup \pi(l_{33} \bowtie Z)] .$$

The first step, according to the algorithm, is to remove the cycle in node 3. Hence, l_{33} is replaced by the transitive closure of this label. There is no need to change any of the other labels since no paths have been lost. The first step thus results in the following assignment statements:

$$l'_{33} := l_{33}^T$$

$$l'_{32} := l_{32}$$

$$l'_{31} := l_{31}$$

$$l'_{30} := l_{30} .$$

In the next step, the arc from node 2 to node 3 has to be removed. By this removal several paths are lost, and some of the labels have to be adjusted to prevent such a loss. This is

completely in line with the changes made in the previous section. The resulting assignment statements are as follows:

$$\begin{aligned} l'_{22} &:= l_{22} \cup \pi(l_{23} \bowtie \pi(l'_{33} \bowtie l'_{32})) \cup \pi(l_{23} \bowtie l'_{32}) \\ l'_{21} &:= l_{21} \cup \pi(l_{23} \bowtie \pi(l'_{33} \bowtie l'_{31})) \cup \pi(l_{23} \bowtie l'_{31}) \\ l'_{20} &:= l_{20} \cup \pi(l_{23} \bowtie \pi(l'_{33} \bowtie l'_{30})) \cup \pi(l_{23} \bowtie l'_{30}) . \end{aligned}$$

The local cycle in node 2 may now be removed too, by taking a transitive closure of the label:

$$l''_{22} := l'_{22}{}^T .$$

Finally, the start node has been reached. Now the arcs from node 1 to nodes 3 and 2 have to be removed, and a transitive closure of the label has to be taken:

$$\begin{aligned} l'_{12} &:= l_{12} \cup \pi(l_{13} \bowtie \pi(l'_{33} \bowtie l'_{32})) \cup \pi(l_{13} \bowtie l'_{32}) \\ l'_{11} &:= l_{11} \cup \pi(l_{13} \bowtie \pi(l'_{33} \bowtie l'_{31})) \cup \pi(l_{13} \bowtie l'_{31}) \\ l'_{10} &:= l_{10} \cup \pi(l_{13} \bowtie \pi(l'_{33} \bowtie l'_{30})) \cup \pi(l_{13} \bowtie l'_{30}) \\ l''_{11} &:= l'_{11} \cup \pi(l'_{12} \bowtie \pi(l''_{22} \bowtie l'_{21})) \cup \pi(l'_{12} \bowtie l'_{21}) \\ l''_{10} &:= l'_{10} \cup \pi(l'_{12} \bowtie \pi(l''_{22} \bowtie l'_{20})) \cup \pi(l'_{12} \bowtie l'_{20}) \\ l'''_{11} &:= l''_{11}{}^T . \end{aligned}$$

These expressions may be combined by doing all replacements at once:

$$\begin{aligned} l'_{11} &:= (l_{11} \cup \pi(l_{13} \bowtie \pi(l'_{33} \bowtie l'_{31})) \cup \pi(l_{13} \bowtie l'_{31}) \\ &\quad \cup \pi(l'_{12} \bowtie \pi(l''_{22} \bowtie l'_{21})) \cup \pi(l'_{12} \bowtie l'_{21}))^T \\ l'_{10} &:= l_{10} \cup \pi(l_{13} \bowtie \pi(l'_{33} \bowtie l'_{30})) \cup \pi(l_{13} \bowtie l'_{30}) \\ &\quad \cup \pi(l'_{12} \bowtie \pi(l''_{22} \bowtie l'_{20})) \cup \pi(l'_{12} \bowtie l'_{20}) . \end{aligned}$$

By changing the labels as described, the result is again a transition graph that is as simple as the one shown in *Fig. 5*. The solution—the union of all paths from start node to end node—is again given by the following expression:

$$\pi(l'_{11} \bowtie l'_{10}) \cup l'_{10} .$$

From the example just described, we may see that the rewriting strategy works by starting from the highest numbered node and going down to the start node. While going downwards, every label l_{ij} is changed in the following way:

$$l'_{ij} := l_{ij} \cup \bigcup_{k=\max(i+1, j+1)}^n \pi(l'_{ik} \bowtie \pi(l'_{kk} \bowtie l'_{kj})) \cup \pi(l'_{ik} \bowtie l'_{kj}) .$$

And for the elements l_{ii} a transitive closure is computed of the foregoing expression to remove the local cycle.

The expression for l'_{ij} may be rewritten to reduce the number of join operations. As can be seen, the expression contains three join operations. The following, equivalent expression contains only two join operations and is, therefore, likely to be computed more efficiently:

$$l'_{ij} := l_{ij} \cup \bigcup_{k=\max(i+1, j+1)}^n \pi((\pi(l'_{ik} \bowtie l'_{kk}) \cup l'_{ik}) \bowtie l'_{kj}).$$

The procedure for evaluating a transition graph as described until now, can be put into a form that is easy to implement on a computer. A transition graph is represented as a matrix. There is a row for every node but the end node, and a column for every node. (Since there are no arcs leaving the end node, there is no need for a row corresponding with the end node.) Each matrix element l_{ij} contains the label of the arc between node i and node j . The data structure thus consists of an $n \times (n+1)$ matrix. And just like we started from the highest numbered node and then worked downwards, the computation has to start at row n , and go downwards until row 1 is entirely computed. This may be noticed too from the expression for element l'_{ij} , where elements l'_{kk} and l'_{kj} , with $k > i$, are used. Similarly, in the computation of l'_{ij} the element l'_{ik} , with $k > j$, is used. We should, therefore, start at element l_{in} in each row, and work downwards until l'_{i1} has been computed.

It may be noticed that both in the computation of l'_{ij} and $l'_{i(j-1)}$ the sub-expression $\pi(l'_{ix} \bowtie l'_{xx}) \cup l'_{ix}$ appears, with $x > i$; the element l'_{ix} is never used as such. An optimization can, therefore, be achieved by computing $\pi(l'_{ix} \bowtie l'_{xx}) \cup l'_{ix}$ instead of l'_{ix} . The formula for l'_{ij} then changes as follows:

$$l'_{ij} := l_{ij} \cup \bigcup_{k=\max(i+1, j+1)}^n \pi(l'_{ik} \bowtie l'_{kj}).$$

The algorithm is now optimized both with respect to the number of join operations and the amount of redundant computations. It is shown in Fig. 9. Note that optimization with respect to the amount of redundant computations is always beneficiary, since it will save a considerable number of relational operations. The reduction of the number of join operations does not necessarily lead to a more efficient computation. Besides the number of join operations the cardinality of the operand relations is important too, and this cardinality is hard (next to impossible) to estimate beforehand. Since an estimation of the actual cost of computation is so hard to get, the minimization of the number of join operations seems to be a good heuristic.

```

for  $i := n$  downto 1 do
  begin
    for  $j := n$  downto  $(i + 1)$  do
      begin
         $l'_{ij} := l_{ij} \cup \bigcup_{k=j+1}^n \pi(l_{ik} \bowtie l_{kj});$ 
         $l'_{ij} := \pi(l'_{ij} \bowtie l'_{jj}) \cup l'_{ij}$ 
      end;
      for  $j := 0$  to  $i$  do  $l'_{ij} := l_{ij} \cup \bigcup_{k=i+1}^n \pi(l_{ik} \bowtie l_{kj});$ 
       $l'_{ii} := l_{ii}^T$ 
    end;
  return  $\pi(l'_{11} \bowtie l'_{10}) \cup l'_{10}$ 

```

Fig. 9. Rewriting algorithm.

5.4 Optimization of partially connected systems

The rewriting strategy we developed has thus far only been discussed for fully connected systems of equations. In all examples, every node was connected to every other node. Of course, recursive problems may be much simpler than this, and we may also use our strategy to solve them. Let us change the equations from Example 9 in the following way:

$$R_1 = \mu X[l_{10} \cup \pi(l_{11} \bowtie X) \cup \pi(l_{12} \bowtie R_2)]$$

$$R_2 = l_{20} \cup \pi(l_{23} \bowtie R_3)$$

$$R_3 = \mu Z[l_{30} \cup \pi(l_{32} \bowtie R_2) \cup \pi(l_{33} \bowtie Z)] .$$

This system is much simpler than the one of Example 9; it is shown in Fig. 10. As may be seen, R_2 and R_3 form a strongly connected component (SCC). We may therefore use our rewriting algorithm to optimize the evaluation of this SCC, and then substitute the result in the expression for R_1 .

The detection of strongly connected components in a graph, and ordering them in a linear way for evaluation, is a well-known problem. Algorithms to do this can, for instance, be found in [14]. We may use these algorithms to detect strongly connected components, and then use our rewriting strategy on these components. Thereby combining the two, to come to a more efficient computation.

5.5 Conclusions

The rewriting strategy developed in this section is applicable to arbitrary, regular mutually recursive programs; this has been illustrated by some examples of fully connected systems of equations. The rewriting strategy is applicable to fully general queries. The rewriting algorithm itself is optimized with respect to the number of join operations and the number of redundant computations, thus leading to a more efficient computation than a naive evaluation of the algebraic equations. The output of this rewriting algorithm is a sequence of standard Relational Algebra expressions and transitive closure operations. Extra optimization possibilities that arise when dealing with systems that are not fully connected have also been discussed.

The advantage of our rewriting strategy is that the transitive closure operation is a well-known standard operation, for which special algorithms can be developed. If a query contains a selection on a recursive definition, this selection can partly be pushed into the computation of the transitive closure. In Section 7 several algorithms for the computation of the transitive closure of a relation are described, and it is discussed how to push a selection into the computation of the transitive closure.

Fig. 10. A not fully connected system.

6. Evaluation of non-regular recursion

As already stated in Section 4, regular recursion is by far the most prominent type of recursion. The optimization of non-regular recursion is, therefore, not studied in depth. It is assumed that non-regular recursion is solved by an iterative process, by algorithms such as the ones in Fig. 11 [2].

As may be seen, the algorithms implement a straightforward iteration over the μ -calculus expression in the query. The function μ -evaluate starts by substituting the empty relation for the variable X , computing the result of the expression, and substituting this result for X until the result of successive iterations does not change anymore. The evaluation of the relational expressions in the μ -calculus expression ($f_i(X)$) is done by a call to the function *evaluate*. This function solves nested μ -calculus expressions by a recursive call to the function μ -evaluate, and finally solves the remaining standard Relational Algebra expression by a call to the function *process*. The function *process* should be regarded as a call to the conventional relational query optimizer.

The algorithms may be improved by using only really new tuples produced in the previous iteration in each successive iteration step. This type of evaluation is called semi-naïve, and described in several papers [6, 12].

7. Transitive closure

In the previous sections a rewriting strategy has been introduced that transforms a regular recursive query (i.e. a system of linear equations) into a sequence of Relational Algebra expressions and transitive closure operations. It is, therefore, crucial to have some ‘good’ algorithms available for computation of the transitive closure of a relation. And since the performance of transitive closure algorithms very much depends on the type of relation (structure, connectivity, etc., see [23]), we foresee a whole range of algorithms supported by the database system, with the query optimizer deciding which algorithm is most appropriate.

Because the transitive closure of a relation is a common and simple example of a recursive query (think of the well-known ancestor problem), there is much interest in algorithms for its computation. A body of research has been performed on this subject [3, 7, 23, 28], and

```

function  $\mu$ -evaluate( $M$  :  $\mu$ -calculus expression) : Relation;
(* $M = \mu X\{f_1(X) \cup f_2(X) \cup \dots \cup f_n(X)\}^*$ *)
var  $X\_new$ ,  $X\_old$  : Relation;
begin
     $X\_new := \emptyset$ ;
    repeat
         $X\_old := X\_new$ ;
         $X\_new := \bigcup_{i=1}^n \text{evaluate}(f_i(X\_old))$ 
    until  $X\_new = X\_old$ ;
    return  $X\_new$ 
end
function evaluate( $Q$  : Query) : Relation;
begin
    if  $Q$  contains  $\mu$ -calculus expression
    then  $Q :=$  replace all  $\mu$ -calculus expressions  $M$  not contained in
        another one by  $\mu$ -evaluate( $M$ );
    return process( $Q$ )
end;

```

Fig. 11. Evaluating non-regular recursion.

transitive closure is being supported by some database systems [37]. Recently, one has started to study the use of distributed computation for transitive closure queries [19, 40, 44]. This is, however, still a difficult topic, and it seems that one should have some knowledge about the application domain to really benefit from parallel processing, as in [23, 24, 25]. Although in the context of PRISMA distributed transitive closure algorithms are very interesting, we will not go into this now. A good overview of parallel strategies for the transitive closure operation may be found in [8, 9].

A first way to compute the transitive closure of a relation is the naive approach. This is a simple iteration over the recursive expression, as we used to describe the semantics of μ -calculus expressions in Section 3.1. This way of evaluation leads to a great number of redundant computations. A widely used improvement upon the naive algorithm is the so-called semi-naive or delta algorithm. It has been widely applied under several names [6, 12], and is shown in Fig. 12. The idea behind the semi-naive algorithm is to restrict the variable *power* to paths that are really new, i.e. generated in the last iteration step and not generated before. When nodes are connected in several ways, only one of the connecting paths is used in the construction of new paths. So, the variable *power* only contains those connections, produced in the previous iteration step, that are really new. This is accomplished by introducing a minus operation in the algorithm.

If the longest path in a graph is n , the number of iterations required when applying the semi-naive algorithm is n . Hence, the number of joins and unions that are computed by applying the delta algorithm is n . The size of one of the join-operands has been reduced, at the cost of a minus operation. Because a minus operation is cheaper than a join, this is usually beneficiary. An index on R can be used to facilitate the remaining join. The semi-naive algorithm is a great improvement when there are many different paths between two nodes. When the graph represented by R is a tree, or only sparsely connected, the semi-naive algorithm does not represent an improvement to the naive algorithm.

Besides the semi-naive algorithm, a whole lot of other algorithms have been developed. These algorithms reduce the number of iterations required and the size of the join operands [7, 23, 28]. We will not discuss these approaches now, but instead concentrate on the problem of pushing selections into transitive closure algorithms.

7.1 Selection on closure

We have already observed that queries will often consist of a selection on the result of a recursive query. This means that after rewriting recursive queries into Relational Algebra and transitive closure operations, the selection has to be pushed into the computation of the transitive closure. This will help to speed up processing, by reducing the size of the join

```

k := 0;
delta := R;
union := R;
repeat
  k := k + 1;
  power :=  $\pi(R \bowtie \delta)$ ;
  (* power =  $R^{k+1}$  *)
  delta := power - union;
  union := union  $\cup$  delta
until delta =  $\emptyset$ ;
(* union =  $\bigcup_{i=1}^{k+1} R^i$  *)
return union

```

Fig. 12. Semi-naive algorithm.

operands. As a running example, the computation of the transitive closure of a binary relation R will be used. It is expressed by the following μ -calculus expression:

$$E = \mu X[R \cup \pi_{1,4}(R \bowtie_{2=1} X)].$$

Remember that this expression just states what has to be computed, and does not yet imply how such a computation will take place.

First let us assume that the selection concerns an attribute not involved in the join-condition. From [1] we may learn that a selection can be pushed into the computation of the transitive closure if the attribute it concerns does not change places during subsequent computations. Hence, a selection on the second attribute of R can be pushed into the computation of the transitive closure—the projection takes care that it does not change places. This is illustrated by the following expression:

$$\sigma_{2=c}E = \mu X[\sigma_{2=c}R \cup \pi_{1,4}(R \bowtie_{2=1} \sigma_{2=c}X)].$$

Assuming that the computation is carried out in the naive way, the selection reduces the amount of computation considerably. First R is restricted, and then this restricted R is substituted for X in the following join. Thereby, the cardinality of the join-operand gets considerably smaller. (Note that due to the semantics of the μ -calculus expression, the selection on X is, in fact, redundant.)

A selection on the first attribute of R cannot be pushed into the computation of the transitive closure without proper care. It does not abide by the rule just formulated: the selection concerns an attribute that changes place during the subsequent computations. In fact, it concerns a joining attribute that disappears because of the projection. It can easily be seen that the following inequality holds:

$$\sigma_{1=c}E \neq \mu X[\sigma_{1=c}R \cup \pi_{1,4}(\sigma_{1=c}R \bowtie_{2=1} X)].$$

When the naive way of computing the result of this expression is followed, it is clear that valid paths are missed. The restricted version of R is substituted for X , which only contains paths starting in c ; whereas this selection should only apply to the final result.

However, because the transitive closure of a relation represents a special case, the expression can be rewritten in such a way that the selection can be pushed down. If the transitive closure is considered in its graph-context, it may be noted that the original expression represents a computation where from a certain start node, arcs are added to the constructed paths until the end nodes are reached. But the computation can also be done the other way around, starting from the end node and following the arcs backwards to the start nodes. Therefore, the original right-recursive expression can always be rewritten into an equivalent left-recursive expression (or a non-regular recursive expression, see e.g. [34]). This gives the following result:

$$E' = \mu X[R \cup \pi_{1,4}(X \bowtie_{2=1} R)].$$

When the naive way of computing the result of the expression is followed, it shall be obvious that the selection can now be pushed into this expression, resulting in the following:

$$\sigma_{1=c}E' = \mu X[\sigma_{1=c}R \cup \pi_{1,4}(\sigma_{1=c}X \bowtie_{2=1} R)].$$

Again note that, due to the semantics of the μ -calculus expression, the selection on X is, in fact, redundant.


```

k := 0;
delta =  $\sigma_{2=c}R$ ;
plural_union :=  $\sigma_{2=c}R$ ;
repeat
    k := k + 1;
    power :=  $\pi(R \bowtie_{2=c}^{2=k} \text{delta})$ ;
    delta := power  $\stackrel{2=k+1}{-}$  plural_union;
    plural_union := plural_union  $\cup$  delta
until delta =  $\emptyset$ ;
return plural_union

```

Fig. 13. Selection on closure.

This rewriting of the original expression is only valid because the transitive closure is such a special case. The relations in the expression all have the same schema, and the join and projection operations are guaranteed to return a correct result. Although the example presented concerned a binary relation, these techniques are also valid for relations with more than two attributes and for non-equality join conditions.

An algorithm to compute a selection on the transitive closure of a relation can easily be given. In Fig. 13 an algorithm is presented that integrates a selection on the second attribute of the transitive closure of the relation R with a semi-naïve evaluation. As stated before, we foresee a whole range of transitive closure algorithms implemented on the database system. Each of these algorithms may easily be adapted to incorporate selections on attributes, as illustrated in this section for the semi-naïve approach.

8. Conclusions

In this paper we showed that recursive queries may be optimized in a logic context and in an algebraic context. We also showed that the techniques used in both contexts resemble each other very much. We explained that in our opinion new advanced database systems will be built starting from existing relational technology, and that it is therefore appropriate to study optimization of recursive queries in an algebraic context.

We then extended Relational Algebra with a fixpoint construct, called μ -calculus expression, that allows us to express arbitrary recursive queries in an algebraic way. Since we now had a construct for expressing recursive queries in algebra, we were able to define optimization strategies on this construct. We described an optimization strategy to rewrite regular (in the context of formal grammars) mutually recursive queries into Relational Algebra and transitive closure operations. We also described the extra optimizations incorporated in the algorithm, e.g. minimizing the number of join operations, and how to push selections into transitive closure operations.

Since transitive closure is such a well-structured simple recursive operation, special algorithms for its computation may be developed. This topic has received much attention lately. In the context of PRISMA, parallel algorithms for transitive closure computations are particularly interesting. We are currently investigating this topic [24, 25, 26], to allow distributed computation of the transitive closure operations that result from our optimization strategy.

Acknowledgements

We thank the reviewers for their comments, which have greatly improved the quality of the paper.

References

- [1] A.V. Aho and J.D. Ullman, Universality of data retrieval languages, in: *Proc. 6th Annual ACM Symp. on Principles of Programming Languages*, San Antonio, CA (Jan. 1979) 110–120.
- [2] P.M.G. Apers, M.A.W. Houtsma and F. Brandse, Extending a relational interface with recursion, in: *Proc. 6th Advanced Database Symp.* Tokyo, Japan (Aug. 1986) 159–166.
- [3] P.M.G. Apers, M.A.W. Houtsma and F. Brandse, Processing recursive queries in relational algebra, in: R.A. Meersman and A.C. Sernadas, eds., *Data and Knowledge (DS-2), Proc. 2nd IFIP 2.6 Working Conf. on Database Semantics*, Albufeira, Portugal (Nov. 1986) (North-Holland, Amsterdam, 1988) 17–39.
- [4] P.M.G. Apers, M.L. Kersten and H. Oerlemans, PRISMA database machine: A distributed main-memory approach, in: J.W. Schmidt, S. Ceri and M. Missikoff, eds., *Advances in Database Technology – EDBT’88, Lecture Notes in Computer Science* 303 (Springer, Berlin, 1988) 590–593.
- [5] F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman, Magic sets and other strange ways to implement logic programs, in: *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, Cambridge, MA (Mar. 1986) 1–15.
- [6] F. Bancilhon and R. Ramakrishnan, An amateur’s introduction to recursive query processing, in: *Proc. Internat. Conf. on Management of Data, ACM-SIGMOD*, Washington, DC (May 1986) 16–51.
- [7] J. Brehler, Transitive closure operation in a relational database, M.Sc. Thesis, University of Twente, The Netherlands, Mar. 1988.
- [8] F. Cacace, S. Ceri and M.A.W. Houtsma, A survey of parallel execution strategies for transitive closure and logic programs, Technical report no. 933, University of Twente, Enschede, The Netherlands, Nov. 1990; submitted for publication.
- [9] F. Cacace, S. Ceri and M.A.W. Houtsma, An overview of parallel strategies for transitive closure on algebraic machines, in: *Proc. Internat. Workshop on Parallel Database Systems*, Noordwijk, The Netherlands (Sep. 1990), *Lecture Notes in Computer Science* 503 (Springer, Berlin, 1991) 44–62.
- [10] S. Ceri and G. Pelagatti, *Distributed Databases, Principles & Systems* (McGraw-Hill, New York, 1985).
- [11] S. Ceri, G. Gottlob and G. Wiederhold, Interfacing relational database and Prolog efficiently, in: *Proc. 1st Internat. Conf. on Expert Database Systems*, Vol. 2, Charleston, SC (Apr. 1986) 141–153.
- [12] S. Ceri, G. Gottlob and L. Lavazza, Translation and optimization of logic queries: the algebraic approach, in: *Proc. 12th Internat. Conf. on Very Large Data Bases*, Kyoto, Japan (Aug. 1986) 395–402.
- [13] S. Ceri and L. Tanca, Optimization of systems of algebraic equations for evaluating Datalog queries, in: *Proc. 13th Internat. Conf. on Very Large Data Bases*, Brighton, UK (Sep. 1987) 31–42.
- [14] S. Ceri, G. Gottlob and L. Tanca, *Logic Programming and Databases* (Springer, Berlin, 1990).
- [15] C.L. Chang, On evaluation of queries containing derived relations in a relational database, in [18], 235–260.
- [16] D. Denoel, D. Roelants and M. Vauclair, Query translation for coupling Prolog with a relational database management system, in: *Proc. Workshop on Integration of Logic Programming and Databases*, Venice, Italy (Dec. 1986).
- [17] G. van Emde Boas and P. van Emde Boas, Storing and evaluating horn-clause rules in a relational database, *IBM J. Res. Develop.* 30 (1) (Jan. 1986) 80–92.
- [18] H. Gallaire, J. Minker and J.-M. Nicolas, eds., *Advances in Database Theory* (Plenum Press, New York, 1981).
- [19] S. Ganguly, A. Silberschatz and S. Tsur, A framework for the parallel processing of Datalog queries, in: *Proc. ACM-SIGMOD*, Atlantic City, NJ (May 1990) 143–152.
- [20] L.J. Henschen and S.A. Naqvi, On compiling queries in recursive first-order databases, *J. ACM* 1(31) (Jan. 1984) 47–85.
- [21] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, MA, 1979).
- [22] M.A.W. Houtsma and P.M.G. Apers, Processing regular recursive queries, in: *Proc. Computing Science in The Netherlands*, Utrecht, The Netherlands (Nov. 1988) 495–514.
- [23] M.A.W. Houtsma, Data and knowledge base management systems: Data model and query processing, Ph.D. Thesis, University of Twente, Enschede, The Netherlands, Nov. 1989.
- [24] M.A.W. Houtsma, P.M.G. Apers and S. Ceri, Distributed transitive closure computation: the disconnection set approach, in: *Proc. 16th Internat. Conf. on Very Large Data Bases*, Brisbane, Australia (Aug. 1990) 335–346.
- [25] M.A.W. Houtsma, P.M.G. Apers and S. Ceri, Distributed computation of transitive closure queries on a fragmented graph, in: *Proc. 3rd Internat. Conf. on Database Theory*, Paris (Dec. 1990), *Lecture Notes in Computer Science* 470 (Springer, Berlin, 1990) 470–484.
- [26] M.A.W. Houtsma, F. Cacace and S. Ceri, Parallel hierarchical evaluation of transitive closure queries, *Proc. 1st Internat. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL (Dec. 1991).
- [27] M.A.W. Houtsma, H.J.A. van Kuijk, J. Flokstra, P.M.G. Apers and M.L. Kersten, A logic query

- language and its algebraic optimization for a multiprocessor database machine, Technical report INF-88-52, University of Twente, The Netherlands, Dec. 1988.
- [28] Y.E. Ioannidis, On the computation of the transitive closure of relational operators, in: *Proc. 12th Internat. Conf. on Very Large Data Bases*, Kyoto, Japan (Aug. 1986) 403–411.
 - [29] M.L. Kersten, P.M.G. Apers, M.A.W. Houtsma, H.J.A. van Kuijk and R.L.W. van de Weg, A distributed, main-memory database machine, in: *Proc. 5th Internat. Workshop on Database Machines*, Karuizawa, Japan (Oct. 1987); and in M. Kitsuregawa and H. Tanaka, eds., *Database Machines and Knowledge Base Machines* (Kluwer, Dordrecht, 1988) 353–369.
 - [30] J.W. Lloyd, *Foundations of Logic Programming*, Second ed. (Springer, Berlin, 1987).
 - [31] K. Morris, J.D. Ullman and A. van Gelder, Design overview of the NAIL! system, Technical Report STAN-CS-86-1108, Stanford University, USA, May 1986.
 - [32] S. Naqvi and S. Tsur, *A Logical Language for Data and Knowledge Bases* (Computer Science Press, Rockville, MD, 1989).
 - [33] J.F. Naughton, Optimizing function-free recursive inference rules, Technical report STAN-CS-86-1114, Stanford University, USA, May 1986.
 - [34] D. Roelants, Recursive rules in logic databases, Technical report R 513, Philips Research Lab., Brussels, Belgium, March 1987.
 - [35] D. Sacca and C. Zaniolo, Magic counting methods, MCC Technical report DB-401-86, Austin, Texas, Dec. 1986.
 - [36] D.S. Scott and J.W. de Bakker, A theory of programs, unpublished seminar notes, IBM, Vienna, 1969; reprinted in J.W. Klop, J.-J.Ch. Meyer and J.J.M.M. Rutten, eds., *J.W. de Bakker, 25 jaar Semantiek*, CWI, Amsterdam, The Netherlands (April 1989) 1–30.
 - [37] Tandem Database Group NonStop SQL, a distributed, high-performance, high-availability implementation of SQL, Tandem report, Apr. 1977.
 - [38] J.D. Ullman, Implementation of logical query languages for databases, *ACM Trans. Database Systems* 10 (Sep. 1985) 289–321.
 - [39] J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. II (Computer Science Press, Rockville, MD; 1989).
 - [40] P. Valduriez and S. Khoskafian, Parallel evaluation of the transitive closure of a database relation, *Internat. J. Parallel Programming* 17 (1) (Feb. 1988).
 - [41] A. VanGelder, A message passing framework for logical query evaluation, Technical report STAN-CS-851088, Stanford University, USA, Dec. 1985.
 - [42] L. Vieille, Recursive axioms in deductive databases: the query/subquery approach, in: *Proc. 1st Internat. Conf. on Expert Database Systems*, Charleston, SC (1986) 179–193.
 - [43] L. Vieille, From QsQ towards QoSQ: global optimization of recursive queries, in: *Proc. 2nd Internat. Conf. on Expert Database Systems*, Tysons Corner, USA (Apr. 1988) 421–436.
 - [44] O. Wolfson, Sharing the load of logic program evaluation, in: *Proc. Internat. Symp. on Database in Parallel and Distributed Systems* (Dec. 1988) 46–55.

Maurice A.W. Houtsma studied computer science at the University of Twente where he obtained his M. Sc. in 1985. In 1989 he received his Ph. D. from the University of Twente on his thesis titled 'Data and Knowledge Bases Management Systems: Data Model and Query Processing'. After receiving his Ph. D. he worked as a post-doc researcher as part of his army

obligations.

In May 1991 he got a research position at the Faculty of Computer Science on the basis of a special 3-year research fellowship that was awarded to him by the Royal Dutch Academy of Sciences (KNAW). His current interests include: deductive databases, distributed computation, (semantic) query optimization, and object-oriented and semantic data models.

Peter M.G. Apers is a professor at the University of Twente in The Netherlands. He received his Ph. D. from the Vrije Universiteit in Amsterdam. His Ph. D. thesis was on distributed query processing and data allocation.

Before and after working on his thesis he was a visiting researcher at the University of California in Santa Cruz and Stanford University.

He was the European Program chairman of the Very Large Data Bases Conference in 1989 held in Amsterdam.

Currently he heads a group working on object-oriented data models, complex object databases, optimization of logical query languages, parallelism in database management systems, and database machines.

He serves on the editorial board of *Data & Knowledge Engineering* and *Distributed and Parallel Database Systems*.

He is a member of ACM and IEEE (CS).