Can we reach a uniform paradigm for
deductive query evaluation?

Rainer Manthey, Hervé Gallaire, and Jean-Marie Nicolas
ECRC, Arabellastr. 17, D-8000 München 81

1. Introduction

During the 1980s we have seen the rise of an exciting new discipline of computer science, called **logic programming**. The most prominent representative of this new programming paradigm is the language PROLOG, developed in the early 1970s by Colmerauer in Marseille and Kowalski in Edinburgh. Programming in PROLOG differs from conventional programming both stylistically as well as computationally, as it uses logic to represent knowledge and deduction to solve problems. Due to the success of PROLOG in the academic world, logic programming today slowly begins to find its way out of the research labs into advanced products and systems like expert system shells or knowledge-based systems.

From the very beginning, the development of logic programming has been closely associated with that of **deductive databases**. Both areas are based on similar formal grounds in first-order predicate logic. Both have been "nourished" by theoretical results and practical methods developed for automated theorem proving purposes during the 1960s (such as the resolution principle of logical inference and its various realizations). Rather early, pioneering researchers in both disciplines have exchanged their ideas and knowledge during a series of workshops held in Toulouse in 1977, 1979, and 1982. Well-documented in book form ([GM 78], [GMN 81], [GMN 84a]), these workshops have established a common formal basis for many activities in logic programming as well as in deductive databases (summarized in [GMN 84b]).

Meanwhile the relationship between the two areas has become so close, that often the notion ‘deductive database’ is regarded as synonymous with a PROLOG system connected in one or the other way with a standard database (or an appropriate file support), thus extending the working memory of logic programs by means of large secondary-storage resources.
Many research groups all over the world have been (and are still) experimenting with prototypes of such PROLOG-DB couplings, and even some commercially available PROLOG systems are nowadays offering access to existing relational databases.

All these systems can readily claim to be called deductive database systems, as they are extending the classical data representation and retrieval facilities of databases by means of rules and deduction. However, the scope of the concept 'deductive database' is much broader, and the intentions and ambitions of many research projects today considerably depart from the PROLOG-DB coupling view. Despite of the fact that such couplings have nowadays reached a fairly high standard (e.g., [Boc 89]), several reasons remain that motivate further research in alternative solutions: conceptual questions, like the desire for an even more declarative style of knowledge representation than that of PROLOG, are often mentioned. Performance considerations - acknowledging, e.g., the necessity to overcome certain limitations of PROLOG in presence of large amounts of data - are responsible for such activities as well.

Technology available in modern database systems (mainly those implementing the relational data model) has already influenced the way how DB-oriented extensions to logic programming have been designed and implemented: file systems, access methods or buffering strategies, e.g., have been adapted to the special needs of a PROLOG environment. However, evaluation and optimization strategies for database queries cannot so easily be exploited as long as the particular deduction strategy of PROLOG is "frozen" and thus restricting choices. As soon as alternative evaluation paradigms are considered, however, the influence of database terminology and solutions becomes constantly more intense and prominent. As a consequence, a kind of (fruitful) competition between a "PROLOG-oriented" and a "database-oriented" view of deduction can sometimes be observed today.

Recently, this development has somehow culminated in scientific contributions focusing on a particular sub-aspect of deductive query evaluation, namely the ability to efficiently handle recursive rules. Such rules enable a database designer to (partially) define a concept in terms of itself (e.g., to define paths in a graph inductively in terms of single edges and already constructed paths). The number of technical publications addressing this topic has dramatically increased during the last few years. This increase is not so much due to a particular practical importance of recursion (except for ubiquitous transitive closure problems), but rather reflects the ongoing debate about a theoretically satisfying and practically exploitable paradigm of deductive query evaluation in general. Recursion appears at the forefront of these discussions, because this feature introduces particular difficulties which are especially challenging research, of course.
The process of answering a retrieval request over a deductive database, can be considered from two main viewpoints. The one - traditionally advocated in the logic programming community - makes use of rules as a means for **problem reduction**. Those parts of a query that are defined in terms of rules are successively replaced by their defining expressions. When a fully expanded query has been reached, its component expressions can be directly evaluated over the base of facts, as they refer solely to explicitly stored concepts. The other view - more on the line of database traditions - regards rules as a means for **answer generation**, temporarily materializing derivable information until all the answers to a particular query have been obtained. Having in mind a tree-like representation of the deduction steps involved, these two viewpoints are preferably called the **top-down** and the **bottom-up** paradigm: either downwards from the original query to the stored facts, or upwards from the facts towards the query.

Each of these basic paradigms may be accomplished according to various strategies, each of them having particular merits as well as drawbacks. Regardless of the strategy chosen, there is a particular crucial problem for each paradigm. If working top-down, it is not obvious when and how to stop the expansion process and how to guarantee answer completeness in presence of recursive rules. If working bottom-up, the problem is how to avoid generation of (a sometimes tremendous amount of) redundant answers and intermediate results. Many attempts of curing the respective problems have been made and solutions been proposed during the last decade. Meanwhile a fairly stable collection of techniques have been identified that optimize each of the paradigms to such a degree that the basic drawbacks just mentioned may be overcome in principle. The search for efficient and elegant implementations, however, is still going on and will probably continue for a while.

As usual for scientific progress, it requires some mental and temporal distance from actual work until common features and unifying principles of competing methods and paradigms can be identified. With respect to the problem discussed here, a kind of key insight into the features shared by an improved top-down and an improved bottom-up evaluation by now seems to have emerged: the stepwise derivation of more refined subqueries from an initial top query can be interpreted (and consequently can be implemented) as a generation process as well. If queries (and intermediate subqueries) are treated in the same way as answers (and intermediate results), i.e., are stored and retrieved like database facts, then a uniform storage concept and a single generative mechanism is sufficient for both, problem reduction and answer generation. As a consequence, the selective capability of top-down problem reduction (avoiding construction of redundant answers) can be integrated with the simplicity and efficiency of a bottom-up accumulation of implicit information (guaranteeing termination and answer completeness).
In the remainder of this paper, we give a more detailed introduction into the particularities of the two evaluation paradigms and of the proposed solutions for overcoming their limitations. This introduction is based on a common terminology requiring only few technical notions. Thus we hope to provide an entry point for non-specialists as well. However, despite of its informal style of presentation, this paper does not simply summarize known results in a somewhat "popular" way. The unified view of improved top-down and improved bottom-up evaluations on which we are going to elaborate in the last section results from very recent work, which was reported in [Bry 89], aiming at an understanding of seemingly different techniques and methods on common grounds. This way of understanding is still unusual even for many researchers actively working in the field. We believe that such an achievement will put us on the way towards a positive answer to the question posed in the title.

2. Deductive and relational databases: an informal introduction

Traditionally, deductive databases are introduced as extensions of relational databases. This is mainly due to the fact that both, the relational model of data and the concept of a deductive database, have been developed approximately at the same time and with a similar motivation in mind. Both approaches are aiming at a uniform and precise formalization of data representation and manipulation in terms of predicate logic. We will follow this tradition here. However, we would like to point out that the decision to represent part of the information contained in a database by means of general rules rather than by means of facts is in principle orthogonal to the choice of a formalism for knowledge representation. Therefore, deductive databases may as well be based on other representation formalisms such as, e.g., functional or object-oriented models. In this section we will shortly and informally recall the main concepts and notions of relational and deductive databases in order to obtain a self-contained presentation accessible also to readers who are not so familiar with the field.

2.1 Relational databases

In a relational database, factual information is organized in tables, called relations. The columns of a relation are named by means of attributes and contain atomic values from a fixed domain. Every row in a relation corresponds to one fact (also called tuple). Information about the personnel in an enterprise, e.g., can be kept in a relation 'employee' with attributes 'name', 'address', and 'salary', where the first two columns contain character strings and the third column contains integer values, respectively.
There are mainly two ways how queries against a relational database can be formulated in predicate logic. In both cases queries are logical formulas containing variables which are replaced by (in logic terminology: are instantiated by) information from the database during query evaluation. One class of languages (called 'tuple calculus languages') uses variables for representing individual tuples of a relation. Relation names are unary predicates, whereas attributes are binary functions applicable to tuples and returning values. SQL, the most widely accepted language for relational databases, belongs to this class. A retrieval request like, e.g., "Find the names and addresses of employees earning more than 100000" could be formulated in SQL as

```
SELECT     name, address
FROM        employee E
WHERE       E.salary > '100000'
```

During query evaluation, the variable 'E' is instantiated by each of the different tuples in the 'employee' relation in turn, and the condition in the WHERE-part is evaluated by accessing the 'salary'-component of the respective tuple. In case the 'salary' value satisfies the condition, the 'name' and 'address' components belong to the set of answers.

A second class of relational query languages (called domain calculus languages) uses variables for representing attribute values rather than entire tuples. To each n-ary relation corresponds an n-ary predicate. Attributes are often completely omitted, assuming a fixed order of component values within a tuple. The example query could be formulated in an (SQL-like) domain-calculus style as

```
SELECT     Name, Address
WHERE       employee(Name, Address, Salary) AND
            Salary > 100000
```

Here 'Name', 'Address', and 'Salary' are variables which are instantiated during evaluation by the component values of each 'employee' tuple in turn. This time the WHERE-part consists of a conjunction of two atomic expressions (called literals). Only those instantiations that satisfy both literals are considered as answers to the query. PROLOG - if applied to a relational database - belongs to this second category of languages. Key words like 'SELECT' and 'WHERE' are usually omitted in PROLOG and variables contributing to the answer are not particularly distinguished. Thus, the PROLOG-representation of the query would simply consist of the condition-part of the above formulation (in addition using the convention that a comma is used instead of 'AND'):

```
?- employee(Name, Address, Salary), Salary > 100000.
```
As most of the literature about deductive databases uses this very compact style of expressing queries, we will do so, too.

2.2 Deductive databases

A deductive database is a relational database in which some of the relations may be virtual (or derived). The tuples in a virtual relation are not explicitly stored, but are implicitly represented by means of a pre-defined query which has to be automatically evaluated each time the virtual relation is accessed. We assume in the following that each relation is either virtual, or a base relation entirely consisting of stored tuples. Rules are expressions associating a virtual relation with a defining query. As an example consider the rule

\[
\text{top_employee}(\text{Name, Address}) \leftarrow \\
\text{employee}(\text{Name, Address, Salary}), \quad \text{Salary} > 100000.
\]

It defines a virtual relation 'top_employee' in terms of a base relation 'employee'. This virtual relation could occur in the internal information system of an enterprise, where the administrative personnel is allowed to access the salaries of all employees (thus using the base relation) whereas other employees only have access to names and addresses of those colleagues earning more than 100000 (by using the virtual relation). A duplication of the information visible to both classes of employees is avoided by choosing an implicit representation for one of the relations. The literal in front of the arrow (symbolizing a logical implication) is called the head of the rule, the query behind the arrow is called its body. Every answer obtained by evaluating the body of the rule instantiates the variables in the head thus defining a tuple of the virtual relation.

Virtual relations like 'top-employee' (restricting or modifying the visibility of a base relation) have motivated the notion of a relational view. Views are supported by many modern relational database systems, but are often subject to certain restrictions due to limitations of the algorithms implementing query evaluation in presence of views. In principle, any syntactically admissible query should be allowed as the body of a rule, provided it contains all the variables occurring in the head of the rule. In particular, a virtual relation may be defined in terms of any other virtual relation. In a deductive database, virtual relations may even be defined in terms of themselves. A managerial hierarchy, e.g., can be described by means of a base relation 'works_for' (connecting employees with their direct superiors) and a virtual relation 'manager_of' defined by means of the two rules
manager_of(X,Y) ← works_for(X,Y)
manager_of(X,Y) ← works_for(X,Z), manager_of(Z,Y).

The second rule, containing a 'manager_of' literal both in the head and in the body, is called recursive. Both rules together define 'manager_of' as the transitive closure of the 'works_for'-relation.

A very natural and convenient way of describing how a tuple in a virtual relation "descends" from the facts in base relations is by means of a derivation tree. The following constitutes a derivation tree for a tuple in the 'manager_of' relation:

```
  manager_of(phil,tom)
   /            \
  /              \         
works_for(phil,jim) manager_of(jim,tom)
   \              \            
   \            \            
  works_for(jim,bill) manager_of(bill,tom)
               \             \     
               \        \      
               \      \   \  
               \   \    \  \ 
               \  \     \ \ 
               \ \    \ \ 
               \ \   \ \ 
               \ \  \ \ 
               \ \ \ \ 
```

The leaves of a derivation tree are base facts. Nodes higher-up in the tree are derived facts obtained by applying a rule to the immediate predecessors of the respective node. The fact 'manager_of(jim,tom)', e.g., is derived from the base fact 'works_for(jim,bill)' and the derived fact 'manager_of(bill,tom)' by means of the recursive rule.

3. Deductive query evaluation: the bottom-up paradigm

The derivation tree representation very naturally leads to a derivation paradigm where rules are used as a generation mechanism for virtual relations. Derived facts are obtained by systematically evaluating rule bodies over base facts as well as over already constructed derived facts which have been temporarily memorized. This mode of derivation is usually called forward reasoning, as it constructs tuples for the relation in the head of a rule from tuples for the relations in the body of a rule - thus proceeding in the direction indicated by the implication arrow. By repeated application of forward reasoning steps, derivation trees are constructed bottom-up. Both notions, forward reasoning and bottom-up evaluation, are often used synonymously. We prefer to reserve 'forward reasoning' for speaking about in-
dividual rule applications, whereas 'bottom-up' characterizes the whole process of generating a derived fact. Using a rule in forward manner corresponds to a classical inference principle of mathematical logic, called 'modus ponens'.

When organizing derivation in a bottom-up manner, some strategical decisions are rather straightforward. As already mentioned, derived facts have to be temporarily materialized in a kind of "auxiliary" memory during query evaluation. For understanding the necessity of this step, consider the following example rules about employees working on certain subjects:

\[
\begin{align*}
\text{specialist_in}(\text{Empl}, \text{Subj}) & \leftarrow \text{works_on}(\text{Empl}, \text{Subj}), \text{experienced}(\text{Empl}) \\
\text{works_on}(\text{Empl}, \text{Subj}) & \leftarrow \text{assigned_to}(\text{Empl}, \text{Proj}), \text{concerned_with}(\text{Proj}, \text{Subj}) \\
\text{experienced}(\text{Empl}) & \leftarrow \text{employed_since}(\text{Empl}, \text{Years}), \text{Years} > 5
\end{align*}
\]

The virtual relation 'specialist_in' is defined in terms of two other virtual relations. The rule defining 'specialist_in' can be applied only if tuples for both 'works_on' and 'experienced' are available. Whatever order is chosen, at least the tuples for the relation that is generated first have to be intermediatedly stored while those for the second relation are constructed. In cases like this, forward reasoning steps cannot be linearly chained (i.e. one step consuming only tuples generated by the step before). In view of obtaining a uniform implementation, derived facts are usually stored in (temporary) base relations as well.

A particular derived fact often has more than one derivation tree, i.e., it can be constructed in different ways. In order to avoid redundancy, one usually checks whether a generated fact is "really" new (and has not been already generated before) by comparing it with the actual contents of the respective temporary relation. This additional duplicate elimination step might sometimes be expensive, but it is unavoidable in certain cases involving recursive rules. Consider another transitive closure case - paths within a graph, e.g. - based on a relation that is not hierarchical (such as 'manager_of'), but contains a cycle. In such a case the only way how bottom-up generation of the transitive closure can be prevented from looping is by avoiding to generate the paths in the cycle more than once. This requires duplicate elimination, at least for recursive predicates.

Both, materialization of intermediate results as well as duplicate elimination, are techniques available in most relational database systems. Therefore bottom-up evaluation is rather easily implementable using conventional database technology, as it can be regarded mainly as an iterated query evaluation over base relations (terminating if all derivable facts have been materialized). In particular, optimization techniques developed for base relation access become directly applicable. In addition, bottom-up evaluation relies on a sound theoretical basis in terms of fixpoint theory ([vEK 76]), which is well-established in computer science.
Despite of all these advantages - which have particularly attracted database researchers - the bottom-up paradigm in its pure form suffers from a serious drawback. It does not take into account the particular query to be answered. For each query, all virtual relations are materialized in order to guarantee answer completeness. This is redundant in most of the cases, as the virtual relations addressed in a query usually depend only on very few other virtual relations. In addition, if a query is partly or fully instantiated (such as "Who are the managers of Phil?" or "Is Jim a specialist in databases?"), only a subset of the tuples in the virtual relations involved is actually relevant as answers or intermediate results. In many cases the amount of implicit information represented by the rules of a deductive databases is so enormous that its unrestricted materialization is prohibitively expensive and has to be avoided under all circumstances. Two possibilities exist: either to modify and improve bottom-up processing (we will address this point in section 5), or to switch to a different paradigm.

4. Deductive query evaluation: the top-down paradigm

The trouble with a pure bottom-up evaluation is due to the fact that the actual query to be answered is used to filter answers only after all derived facts have been generated. A much better performance can be expected if the predicates and constant values occurring in the respective query are taken into account as early as possible in order to avoid any generation of unnecessary derived information. This can be achieved by using rules in a different way: instead of instantiating the variables in the body of the rule by matching them with suitable facts, one may as well instantiate the variables in the head of a rule by matching them against a query. Whereas in the first case new derived facts are inferred from stored facts and other derived facts, new subqueries are generated now from the original query or from other subqueries obtained earlier. If the query to be answered is, e.g.,

`?- specialist_in(jim,S)`

we can infer the subquery

`?- works_on(jim,S), experienced(jim)`

indicating that only a subset of the 'works_on'-tuples and only a single tuple from the 'experienced' relation are required for answering this particular query. By applying the same kind of inference to each of the literals in the subquery in turn, we reach further subqueries referring to base relations only. The subqueries thus obtained can be represented in a tree-like manner as well:
The variables still occurring in this tree can be instantiated in a second phase by evaluating
the leaf literals over base relations and propagating the instantiations found bottom-up to
higher nodes, thus obtaining complete derivation trees.

When inferring new subqueries from a given one, rules are traversed from the head towards
the body, opposite to the direction of the implication arrow. This mode of derivation is there-fore called **backward reasoning**. By repeated application of the backward reasoning prin-
ciple a partially instantiated derivation tree is constructed **top-down** from the root to the
leaves. Again we use 'backward reasoning' for the individual inference step and 'top-down
evaluation' for the construction of a whole tree. Backward reasoning corresponds to another
classical inference principle of logic, called 'modus tollens'. Top-down derivations of sub-
queries can be regarded as pre-processing steps which can be either performed entirely
before answers are generated, or interleaved with answer generation depending on the par-
ticular strategy chosen.

It is recursion which causes problems for top-down processing of rules. If a subquery refer-
ing to a virtual relation 'r' is matched against a recursive rule it may happen that the same
subquery (or a syntactic variant of it) is produced again, because the recursive rule contains
another occurrence of an 'r'-literal. As an example, consider once more the recursive rule
defining 'manager_of' and the query '?'- manager_of(X,bill)'. Backward reasoning yields a
subquery '?'- works_for(X,Z), manager_of(Z,bill)'. It contains the literal 'manager_of(Z,bill)'
which represents in fact the same query as 'manager_of(X,bill)', although both are not
strictly speaking identical due to the different variables occurring. If this similarity of sub-
queries remains undetected an infinite expansion of effectively identical subqueries is the
consequence.

In the bottom-up case, elimination of duplicate answers was chosen as a remedy. Sub-
queries may contain variables, however, such that it is not only duplicate queries that have
to be eliminated, but variants, as in the above example. Even more preferable would be an
elimination of subqueries that are instances of already derived ones: if looking for answers to
the query '?'- manager_of(X,Y)' it is superfluous to generate a less general subquery like
'?- manager_of(X,bill)', as every answer to the latter is already being computed for the former. Testing whether a newly derived subquery is a variant or an instance of a previous one is computationally much more expensive than a simple duplicate check. Many top-down implementations - including most PROLOG interpreters - therefore do not perform such a check, thus risking that for certain queries the top-down phase never terminates.

Even if termination of top-down processing can be achieved, it is not easy to guarantee that all answers to a recursive query are finally obtained. Answer completeness can be achieved only if the answers that have been generated for recursive (sub)queries are memorized "long enough" and remain accessible during iterated evaluation of these subqueries. Many top-down implementations do not store answers (i.e., derived facts) in temporary relations, but maintain additional temporary data structures for storing both subqueries and their answers together. These data structures are released very quickly in order to keep the temporary storage as small as possible. As subqueries depend linearly on each other, it is sufficient to store at any time only subqueries belonging to a single branch of a derivation tree. Thus the data structures storing subqueries are usually managed like a stack. However, the derived facts that are necessary for completely answering a recursive subquery cannot all be found on a single branch of a derivation tree. Therefore either the stackwise administration of the temporary data structures has to be given up, or answers have to be stored apart from subqueries.

Despite of the obvious conceptual superiority of the top-down paradigm due to its ability of exploiting queries and subqueries during evaluation, top-down solutions that are complete in case of recursion ([ITS 86], [Vie 87]) have not (yet) been fully accepted by many database researchers. This may be due to the problems sketched above which are encountered when trying to improve the top-down principle presupposing a PROLOG-like context (without variant-instance test for subqueries and based on a separate temporary store). In particular the coordination between top-down derivation of subqueries and bottom-up generation of answers appears rather intricate as compared to the simplicity of an unrestricted bottom-up processing.

5. Towards a closer integration of top-down and bottom-up reasoning

In the two previous sections we have briefly described the main paradigms of deductive query evaluation and discussed their main positive and negative characteristics. Numerous improvements of both paradigms have been proposed. If they are interpreted on a suitable
level of abstraction, it can be observed that the suggestions made for improving bottom-up evaluation are essentially typical for a top-down processing, whereas improvements proposed for a top-down evaluation are essentially characteristic of the bottom-up paradigm. Thus, the hope for a paradigm incorporating the best of both directions begins to emerge. Till now, contributions improving the one or the other evaluation scheme have been presented in various, seemingly incompatible forms under the "disguises" of the different formalisms. It is only recently that common general principles have been discovered.

The main contribution to improving the bottom-up paradigm is based on a transformation of the set of rules while preserving the simplicity of the evaluation paradigm ([RLK 86], [BMSU 86], [BR 87]). These transformations are such that only those answers and intermediate results are generated which are relevant for the actual query. For this purpose, additional auxiliary virtual relations are introduced and additional rules defining relations are added. Furthermore, each of the original rules is replaced by a set of rewritten rules referring to both, the original relations as well as the newly introduced auxiliary relations. The query to be answered in each particular case is entered as a fact initially supplied for one of the auxiliary relations. Depending on the respective rule system subject to the transformation, a more or less complex set of rewritten rules is obtained. The connection between original and rewritten rules is often not very obvious and may soon become fairly puzzling. Nevertheless, the rewriting leads to the same restriction of the set of generated virtual facts as obtained by means of a top-down pre-processing. In addition, facts for the auxiliary relations are generated as well, that serve as a means for controlling generation. If understood in terms of the formalisms originally chosen for describing the rewriting, it does not surprise that the transformation has been called "magic rewriting".

However, it is not by chance that the improvement with respect to derived fact construction obtained by means of top-down pre-processing is the same as the one obtained by simple bottom-up evaluation of the "magically" rewritten rules. Certain similarities have often been observed. In [Vie 89], different bottom-up methods have been classified according to a top-down formalism (called SLD-AL resolution). [Seki 89] has even established a one-to-one mapping between the individual computation steps performed by top-down and improved bottom-up evaluation. In [Bry 89] this isomorphism has now received an interpretation and explanation which shows that in fact the bottom-up processing of the rewritten rules directly implements a (particular strategy of) top-down evaluation of the original rules. The auxiliary facts generated by the rewriting approach have a natural interpretation as encoded representations of the subqueries that the top-down phase would have generated. Having this explanation in mind, a systematic understanding of the rewriting process can be obtained based on a uniform fixpoint-theoretic view of both, subquery construction and answer generation.
This development is complemented by contributions approaching deductive query evaluation from a top-down point of view, such as [VL 89], which on their part have reached implementations based on techniques that closely resemble bottom-up features such as:

- storing subqueries in base relations rather than in separate stacks
- encoding subqueries according to different instantiation patterns (thereby reducing the instance/variant test for subqueries to a simple duplicate test)
- organising the generation of subqueries and of answers in terms of a uniform iterative strategy

We would like to conclude this section with an illustration of the rewriting technique using an example already discussed earlier. We are conscious that such a comparatively simple example cannot fully account for the intricacies arising in more complex cases (in particular as it will be a non-recursive one). A more comprehensive treatment, however, is definitely beyond the scope of this paper.

Once more, we consider the three rules defining virtual relations 'specialist_in', 'works_on', and 'experienced' together with the query '?- specialist_in(jim,S)'. In order to illustrate the very principle of the implementation of a top-down evaluation by means of a "bottom-up machinery" we assume for the moment that only one auxiliary virtual relation is used, which we call 'query', as its tuples are in fact directly corresponding to the subqueries derived by top-down processing. The rewritten rules required for this example are as follows:

specialist_in(E,S) <--
query(specialist_in(E,S)), works_on(E,S), experienced(E)

works_on(E,S) <--
query(works_on(E,S)), assigned_to(E,P), concerned_with(P,S)

experienced(E) <--
query(experienced(E)), employed_since(E,Y), Y > 5

query(works_on(E,S)) <--
query(specialist_in(E,S))

query(experienced(E)) <--
query(specialist_in(E,S)), works_on(E,S)

If in addition, the initial query is represented by means of 'query(specialist_in(jim,E))', i.e., by an auxiliary fact, bottom-up evaluation of the last two rules will generate two more auxiliary facts, namely 'query(works_on(jim,S))' and 'query(experienced(jim))'. During application of the rewritten original rules the additional 'query'-literals in the rule bodies provide for an access to these auxiliary facts thereby instantiating those variables that would otherwise be
instantiated during a top-down pre-processing (compare with the partially instantiated derivation tree in section 4). Accordingly, only 'works_on'-tuples with first component 'jim' are generated. The only 'experienced'-tuple generated (if any) is 'experienced(jim))' due to the presence of the auxiliary fact 'query(experienced(jim))'. No 'query' facts have to be generated for the three base relations involved, because they are already (completely!) stored and don't have to be temporarily materialized.

If directly employing this "naive" style of rewriting, the auxiliary facts generated may still contain variables (like 'query(works_on(jim,S))' in the example) thus compromising the duplicate test which is correct only for fully instantiated facts. The problem can be overcome by using several different auxiliary relations for representing subqueries instead of a single one. For each relation involved and for each instantiation pattern occurring, one such auxiliary relation is introduced. Applied to our little example this leads to three auxiliary relations: 'query_specialist_in.1', 'query_works_on.1', and 'query_experienced.1', the postfix '.1' indicating that only the first argument of the subquery thus encoded is instantiated. Using this more detailed encoding we obtain the following rewritten rules:

\[
\begin{align*}
\text{specialist_in(E,S)} & \leftarrow \text{query_specialist_in.1(E), works_on(E,S), experienced(E)} \\
\text{works_on(E,S)} & \leftarrow \text{query_works_on.1(E), assigned_to(E,P), concerned_with(P,S)} \\
\text{experienced(E)} & \leftarrow \text{query_experienced.1(E), employed_since(E,Y), Y > 5} \\
\text{query_works_on.1(E)} & \leftarrow \text{query_specialist_in.1(E)} \\
\text{query_experienced.1(E)} & \leftarrow \text{query_specialist_in.1(E), works_on(E,S)}
\end{align*}
\]

The top query is now represented by the fact 'query_specialist_in.1(jim))', which does not contain a variable any more. The same is the case for the two auxiliary facts generated, namely 'query_works_on.1(jim)' and 'query_experienced.1(jim)'. Both principles, realization of a top-down generation of subqueries by means of bottom-up generation of auxiliary facts as well as encoding of different instantiation patterns with different auxiliary relations, explain rather straightforwardly which effect the rewriting has on a bottom-up evaluation.
6. Conclusion

In this paper, we have tried to combine an informal introduction into the basic paradigms of deductive query evaluation with a discussion of advanced results progressing towards a more unified view of evaluation. Interpreting improvements developed independently for each of the paradigms, a common view appears to be emerging which incorporates features of both, the bottom-up as well as the top-down paradigm. Whether this convergence will help to improve clarity and uniformity of technical contributions to the field remains to be seen.

Acknowledgements

Ideas emerge from constructive interaction. This paper would not have been written without such an interaction within the Knowledge Base Group at ECRC, the work of Alexandre Lefebvre and Laurent Vieille on recursion, and the analysis by François Bry that provided the input for this paper.

References


