# Recursive Strategies for Answering Recursive Queries - The RQA/FQI Strategy

Wolfgang Nejdl

Technische Universität Wien
Institut für Angewandte Informatik und Systemanalyse
A-1040 Vienna, Paniglg.16, Austria
nejdl@tuhold.uucp

**ABSTRACT**
In this paper we will discuss several methods for recursive query processing using a recursive control structure. We will describe the QSQR method, introduced in [Vie86] and show that it fails to produce all answers in certain cases. After analyzing the causes of this failure we propose an improved algorithm - the RQA/FQI Strategy - which is complete over the domain of function-free Horn clauses. The new method uses a two step approach - recursive expansion + an efficient variant of LFP iteration - to evaluate recursive queries. A short comparison of these methods shows the efficiency of RQA/FQI.

## 1. INTRODUCTION

Recursive query processing has been an area of active research for the last five years. Many strategies for this problem have been developed ([Hen84], [Ban86a], [McK81], [Smi86], [Ull85], [Vie86], [Loz85], [Cer86], [Ion86], [Ras86], for comparisons of existing algorithms see also [Ban86], [Han86]).

Many methods have been proposed in these strategies: interpreted and compiled approaches, optimization and evaluation strategies, top down and bottom up, recursive and iterative. Application areas range from linear rules to the whole area of function free Horn clauses.



Our aim was to find a strategy which should be both efficient and general (applicable to a large area of recursive rules).

As shown by [Ban86], strategies using recursive control (for the main part of the evaluation) show a superior behaviour to iterative strategies by cutting down the set of facts to be searched and avoiding much duplicate work. We will therefore discuss briefly the properties of PROLOG being the main prototype of a recursive top down evaluation strategy.

Then we turn to QSQR which has recently been introduced for handling recursive axioms in deductive databases by [Vie86]. The application domain of this strategy (according to [Vie86]) are all kinds of recursion defined by means of function free Horn clauses. We will discuss QSQR in more detail since it is claimed both in [Vie86] and in [Boc86] to be complete over its application domain. Moreover, in a survey and comparison of strategies for handling recursive queries in [Ban86] QSQR is said to be one of the best methods available.

Unfortunately QSQR fails to find all answers in certain cases. After a short description of QSQR we will give such an example and discuss the causes for this failure.

Using the insights gained through this analysis, in the second part of this paper we propose an improved recursive strategy - the Recursive **Q**uestion **A**nswering / **F**rozen **Q**uery **I**teration (**RQA/FQI**) Strategy - which is complete over the domain of function free Horn clauses.

A short comparison with the existing strategies shows the efficiency of the new method.

The RQA/FQI Strategy has been implemented in the PROLOG-DB System which is described in [Nej86a].

## 2. PROLOG

Actually PROLOG is a complete programming language rather than just a question answering strategy (see e.g. [Clo81]). However viewed as an answering strategy PROLOG uses a recursive top down strategy for the evaluation of queries over Horn clauses. The selection of rules is determined by the rule order, the selection of subgoals is depth first and left-to-right.

Because of this simple strategy and its inability to recognize cycles its application domain is strongly dependent on the available data (no cycles!). Its efficiency is high only in those cases when goals can be proved only once and constants can be properly propagated into the subgoals. In the other cases PROLOG has to do much duplicate work and uses too many non-relevant facts in its evaluation.

A further problem arising when coupling PROLOG with a relational database is its tuple oriented approach which hinders some possible improvements and optimizations used by a RDBMS.

Thus, while giving a good prototype example, its use is limited to domains where relatively few data and not too complex recursions are needed.

## 3. QSQR

### 3.1. Description

QSQR is a recursive top down strategy for handling (almost) all kinds of function free Horn clauses. An own selection function (subgoal most instantiated) determines the evaluation order of subgoals and the propagation of values. It uses a set oriented approach and reuses already evaluated queries (and their answers) to avoid duplicate work and to recognize cycles.

We will briefly describe QSQR according to [Vie86]. For further details please refer to the descriptions in [Vie86], [Ban86] and [Boc86].

When answering a query $R_i$ on a set of function free Horn clauses, the main principle of QSQR is the recursive expansion of the search tree for $R_i$.

Additionally, certain set variables are associated with each recursive predicate $R_i$:

- a global set of tuples Ans_$R_i$ which is used during evaluation to store the answers already found for queries on $R_i$,
- global sets of instances Inst_M_$R_i$, for each query pattern M (or adornment, see [Ull85]) of $R_i$, containing the instantiated arguments of queries already executed on $R_i$, where M indicates which arguments these instances correspond to,
- a local set of instances Loc_M_$R_i$ for each subquery indicating the instantiated arguments of the subquery.

Bindings for arguments are propagated in a set-oriented manner (generalized queries).

The answers to recursive queries are found by the following procedures:

```
Procedure ANSWER:
/* main procedure for answering a query Ri */
/* Input: local set of instances Loc_M_Ri */
/* Output: answers to Ri */
{repeat
   for each clause Cj defining Ri:
      repeat
         choose the first/next predicate ac-
            cording to a selection function;
         generate the corresponding general-
            ized query Pj (propagation of
            arguments);
         if Pj is recursive
         then apply ANS2 to Pj
         else evaluate Pj by standard nonre-
            cursive methods
      until there are no more predicates in
         the body of Cj;
      infer new answers for Ri and add them
         to Ans_Ri
 until no new answers are added to Ans_Ri.
}
```

```
Procedure ANS2:
/* Input: local set of instances Loc_M_Ri */
/* Output: set of answers for Ri yielded */
/*         in step 1 and 3 */
{search the tuples in Ans_Ri matching an ele-
    ment of Loc_M_Ri;
 add Loc_M_Ri to the corresponding Inst_M_Ri
    and retain those instances which are new;
 call ANSWER with the remaining set of new in-
    stances as input.
}
```

At the beginning, the Ans_$R_i$'s and Inst_M_$R_i$'s are empty, except one Inst_M_$R_j$ corresponding to the initial query $R_j$. At the end, the Ans_$R_i$'s contain all derived answers to queries on $R_i$. The set Ans_$R_j$ corresponding to the original query predicate $R_j$ should contain all answers to the initial query.

### 3.2. Incompleteness of QSQR

Proposition 1: QSQR is not complete on its supposed application domain (function free Horn clauses).

Proof: by presenting the following counter example. The causes of QSQR's incompleteness will be discussed afterwards.

Example 1:

Axioms:

```
(1)  n(X,Y) :- r(X,Y).
(2)  n(X,Y) :- p(X,Z), n(Z,W), q(W,Y).
```

Facts:

```
p(c,d).   r(d,e).   q(e,a).
p(b,c).             q(a,i).
p(c,b).             q(i,o).
```

Query: ?- n(c,Y).

Expected answers: {(c,a),(c,o)}

The final sets produced by QSQR are:

Inst_bf_n={c,d,b}, Ans_n={(d,e),(c,a)}.

The answer (c,o) is not found.

Due to space limitations we will not give a complete evaluation trace in this paper, it can be found together with a more detailed discussion in [Nej86b]. A short note on the incompleteness of QSQR can also be found in [Vie87].

What is more important in this context are the causes of QSQR's failure to produce all answers. So we state the following proposition:

Proposition 2: The cases where QSQR does not find all answers to a query can be characterized by the following properties:

- the repeated query occurs more than one step below the original query in the derivation path, and
- an answer needed for further iteration is not yielded by other (often nonrecursive) clauses.

Proof (informally): QSQR satisfies repeated queries just by searching the corresponding global set of answers. Therefore, iterating over a set of clauses (inside of ANSWER) for the second time does not expand the subgoals, but uses the set of answers to yield results.

Therefore intermediate goals (direct subgoals) of a query often prevent deeper subgoals from being evaluated more than once. New answers produced for this query by iterating on its clauses cannot be used in a repeating subquery unless it is a direct subgoal of this query. To find answers which could maintain the iteration process, QSQR must derive tuples from other sources, i.e. another (nonrecursive) clause (cf. the given example in [Vie86]).

This is especially relevant, if cycles are present in the tuples controlling the recursion (cf. 'driver tuples' in [Hen84]) or in the case of mutual recursion.

### 3.3. Strategic Failures of QSQR

If we analyze the QSQR strategy we can differentiate between two steps which are interwoven during the evaluation.

The first step is the recursive expansion of the rule goal tree evaluating all non-recursive predicates as well as all predicates which contain no repeated subqueries in their body. Predicates containing repeated subqueries - queries which have occurred already earlier in the expansion - are answered as far as possible, using the answers already found.

The second step tries to complete the repeated subqueries by iterating on each recursion level by means of 'naive evaluation' (simple least-fixed-point iteration). However, as the iteration takes place always only on one level, answers cannot properly be propagated more than one level.

So, while the two steps are basically correct, they must not be interwoven in order for the LFP iteration to iterate over more than one level. Another point is that by storing only the instantiated arguments of queries, queries like P(a,X,X) and P(a,X,Y) are treated as the same query. This also leads to wrong results in some cases.

## 4. RQA/FQI STRATEGY

### 4.1. Overview

The RQA/FQI Strategy which we are presenting in this paper uses a two step approach:

In the first step of the algorithm we use a recursive evaluation strategy similar to QSQR, expanding the search tree top down, but doing no LFP iteration in the recursive procedure EXPAND. Answers already deduced are re-used. The expansion stops when a Repeated Incomplete Query is encountered or after a subquery is answered completely using basic facts and nonrecursive predicates.

In the second step we process all incomplete branches of the search tree (Frozen Queries) using an efficient variant of LFP iteration over the incomplete goals caused by Repeated Incomplete Queries.

We will describe the single steps in greater detail after some definitions in the next chapter.

## 4.2. Definitions

In order to describe the RQA/FQI Strategy we will define the new terms Repeated Incomplete Query, Repeated Complete Query, Derived Incomplete Query, Frozen Query, Propagation Subgoal and Critical Path.

Def. **RIQ** (Repeated Incomplete Query): A **RIQ** is a query which is subsumed by a previous query which has not yet been answered completely.

The RIQ's are the only nodes which cannot be expanded by the recursive strategy in the first step (in order to avoid cycles). However, cutting the execution path in the search tree at a RIQ may affect the completeness of any goal (even the initial goal) relying on this subgoal. If a RIQ is encountered only answers already produced can be used in the further expansion of the search tree.

Def. **RCQ** (Repeated Complete Query): A **RCQ** is a query which is subsumed by a previous query which has already been answered completely. If a RCQ is encountered all its answers can be taken from the global answer sets.

Def. **DIQ** Derived Incomplete Query: A Derived Incomplete Query (**DIQ**) is a query containing a RIQ or another DIQ as a subquery. As with RIQ's, DIQ's cannot be answered completely during the expansion phase.

Def. **FQ** Frozen Query: A Frozen Query (**FQ**) is a query containing a RIQ or a sub-query which is incomplete because of a RIQ on a deeper level (a DIQ). Together with a set of Propagation Subgoals it stores the current step of evaluation for a clause which cannot be evaluated completely in the first recursive step.

A **FQ** consists of an uninstantiated rule of the form

$FQ_i(Q_i \text{ :- } PSG_i, P_i, S_i.)$

corresponding to an original rule

$(Q_i \text{ :- } E_i, P_i, S_i.)$

where $PSG_i$ (a Propagation Subgoal) is used to propagate the arguments instantiated so far in the original rule (according to the different instantiations of the terms $Q_i$ and $E_i$)

and $P_i$ is a RIQ or a DIQ (i.e. $P_i$ is on a Critical Path).

Associated to each FQ is a set of Propagation Subgoals.

Def. **PSG** Propagation Subgoal: A Propagation Subgoal (**PSG**) is a special artificial subgoal added in front of each recursive subgoal in a rule. This has to be done manually or - as in the PROLOG-DB system - by a pre-compiler.

For each Frozen Query $FQ_i$ a set of $PSG_i$'s is used to propagate the different instantiations of the query $Q_i$ and the term $E_i$. The term $E_i$ has already been completely answered and is thus fully instantiated. Only instantiations of those arguments have to be propagated which are needed later for the evaluation of the Frozen Query.

Def. **CP** Critical Path: A Critical Path (**CP**) is a path of the search tree which cannot be completely evaluated in the first recursive expansion of the search tree. It is represented by Frozen Queries and Propagation Subgoals and has to be further evaluated in the iteration step.

A **CP** is generated on the current path from a RIQ up to the initial goal node or up to the intersection with an already existing **CP**. The part of a **CP** above an intersection is automatically generated only once due to the recursive control structure of the expansion procedure.

## 4.3. Algorithm

The following sets are used in RQA/FQI:

- global sets of tuples ANS, ANS1, ANS2 which are used to store the answers already found for queries on recursive predicates $R_i$, used to separate old, currently used and new answers
- a global set Query_Goals containing the instantiated heads of queries on recursive predicates $R_i$ already executed,
- a global set of Frozen Queries FQ_SET which is used to store the Frozen Queries generated during the expansion,
- global sets of Propagation Subgoals $PSG_i$_SET and NEW_$PSG_i$_SET for each Frozen Query which are used to store the different instantiations for the FQ's (currently used and new instantiations).

Bindings for arguments are propagated in a tuple-oriented manner. However, queries over database predicates or database views (database equivalent predicates - described by non-recursive predicates) are processed set-oriented against an underlying relational database and stored in the PROLOG database. Retrieving these tuples from the PROLOG database is done tuple-oriented again.

Additionally, the instantiations of the different FQ's are stored in a set-oriented manner (using the PSG facts). Thus each incomplete branch of the search tree is stored only once and is completed by instantiating it with all appropriate PSG's in its PSG set. This is especially useful, if a lot of basic facts (stored in the RDBMS) has lead to many different instantiations.

Therefore, the advantages of the set-oriented approach of RDBMS's for retrieval of non-recursive predicates (optimization of joins, selection first and storage of large amounts of data) as well as the tuple-oriented advantages of PROLOG (automatic constant propagation, unification, recursive control structure and backtracking) can be used.

We will describe the algorithm in a PROLOG like manner as its recursive control structure and backtracking lends itself more easily to the description of the recursive algorithm than an iterative description. However the algorithm could be implemented in a more procedural way using for each, repeat until and similar constructs instead of depth-first search and backtracking for iteration.

In the algorithm described below iteration is done by backtracking (fail, when no more answers can be found for one subquery). The search tree is expanded depth-first, the subqueries (subgoals) are ordered by a selection function (currently terms with less uninstantiated variables are processed first). Argument propagation is done automatically by shared variables.

The answers to recursive queries are found by the following recursively defined procedures:

```
RQA_FQI :-
% main procedure for answering a query Ri
% Input: a query Ri
% Output: answers to Ri
   EXPAND Ri,
   ITERATE on the generated FQ's,
   RETURN answers to Ri (from ANS).

EXPAND Ri :-
% expanding the search tree for Ri
% Input: an instantiated query Ri
% 1st part: produce all answers
   EXPAND_ONCE Ri,
   STORE_ANSWER Ri in ANS2
   FAIL.

EXPAND Ri :-
% 2nd part: return answers tuple-oriented
   RETURN answer to Ri.

EXPAND_ONCE Ri :-
% returns an answer to Ri
   ADD Ri to Query_Goals,
% iteration over all clauses defining Ri (by
% backtracking)
% OR node in the rule/goal graph
   GET_CLAUSE Cj defining Ri,
   EXPAND_ALL_SUBGOALS from Cj and return Ri
          instantiated.

EXPAND_ALL_SUBGOALS in Cj for Ri :-
% expands all subgoals and
% returns Ri instantiated with an answer;
% AND node in the rule/goal graph
% 1st part: more than one SG
   EXPAND_SUBGOAL first SG in Cj,
   EXPAND_ALL_SUBGOALS rest of SG's in Cj.

EXPAND_ALL_SUBGOALS :-
% 2nd part: last subgoal
   EXPAND_SUBGOAL.

EXPAND_SUBGOAL Si :-
% expands subgoal Si and
% returns answer resp. propagations for
% further subgoals
   IS_PSG Si,
% a PSG is not changed, but used later
% for instantiating the appropriate FQ
   RETURN PSG.

EXPAND_SUBGOAL Si :-
   IS_not_recursive Si,
   EVALUATE Si by standard non-recursive
          methods (first time set-oriented by
          database retrieval, but return
          answers tuple-oriented).

EXPAND_SUBGOAL Si :-
   IS_RIQ Si,
   GENERATE_FQ,
   MARK next higher goal (Ri) incomplete,
   RETURN old answer (from earlier step).

EXPAND_SUBGOAL Si :-
   IS_RCQ Si,
   RETURN answer.

EXPAND_SUBGOAL Si :-
% recursive expansion of search tree
   IS_recursive, not RCQ, not RIQ,
   EXPAND Si.

EXPAND_SUBGOAL Si :-
% end processing of DIQ
   IS_DIQ Si,
   GENERATE_FQ,
   MARK next higher goal (Ri) incomplete,
   FAIL.
```

```
GENERATE_FQ :-
% stores the incomplete part of the
% current rule (FQ) + the instantiation
% in form of a PSG,
% if the FQ is already stored,
% only the new PSG has to be asserted.

ITERATE :-
% procedure for iteration on FQ's
% Input: a set of FQi's (FQ_SET) +
% instantiations (PSGi_SET's) + a set
% of answers (ANS1) generated during
% the preceding iteration resp. the first
% recursive expansion.
% iterate over Frozen Queries
   GET_FROZEN_QUERY FQi(Qi:-PSGi,Pi,Si),
% instantiate FQi using all PSGi's from
% the appropriate set
   GET_INSTANTIATION for PSGi,
% iterate over all current answers for
% first subgoal
   GET_ANSWER for Pi from ANS1,
% expand rest of subgoals recursively
   EXPAND_ALL_SUBGOALS Si,
   STORE_ANSWER Qi into ANS2,
   FAIL.

ITERATE :-
% iterate until no new answers are found
   NEW_ANSWERS_FOUND,
   PSGi_SET = PSGi_SET + NEW_PSGi_SET,
   ANS = ANS + ANS1,
   ANS1 = ANS2,
   ITERATE.

ITERATE :-
% iteration finished
% all answers returned in ANS
   ANS = ANS + ANS1.
```

At the beginning of RQA_FQI, the sets ANS, ANS1 and ANS2 are empty. At the end, the ANS set contains all derived answers to recursive queries. The subset $ANS\_R_j$ corresponding to the original query predicate $R_j$ contains all answers to the initial query.

The first step expands (only) the necessary parts of the search tree and stores exactly those paths which cannot be evaluated completely in the first step (Critical Paths) as Frozen Queries and Propagation Subgoals. It thus avoids duplicate work and processing of non-relevant facts.

The second step - an efficient variant of LFP iteration - ensures the following two properties:

- Whenever new results are produced by a subquery, those tuples can be used by a supergoal (propagation of sub-results up to the top of the derivation tree), and
- New answers generated for a query can be used by any subquery (usage of new results by other (sub-) queries).

Note: A variant of the algorithm described above uses new answers already in the same step during which they are generated. This leads to some duplicate work as they have to be used in the next iteration step in any case. However, the number of iteration steps decreases as new answers can be propagated through several FQ's in one iteration step. If the processed relations are of only small size the amount of duplicate work is less than the decrease in iteration overhead. In this case this variant (similar to a Gauß-Seidel-iteration) is advantageous.

## 4.4. An Example Evaluation

Example 2:

Let us now consider the axioms and facts of example 1 and process the query "?-n(c,X1)" according to the strategy just described.

Step 1 : recursive expansion

```
EXPAND  : n(c,X1)
clause1 : n(c,X1) :- r(c,X1).
non_rec : r(c,X1) ... no answers found
clause2 : n(c,X1):-p(c,X2), psg(id1,c,X1,X2),
                 n(X2,X3), q(X3,X1).
non_rec : p(c,X2)
answer  : p(c,d)
propag. : psg(id1,c,X1,d)
rec     : n(d,X3)
  EXPAND  : n(d,X3)
  clause1 : n(d,X3) :- r(d,X3).
  non_rec : r(d,X3)
  answer  : r(d,e)
  answer  : n(d,e)
  NEW_ANSWER stored
  clause2 : n(d,X3):-p(d,X4),
                   psg(id1,d,X3,X4),
                   n(X4,X5), q(X5,X3).
  non_rec : p(d,X4) ... no answers found
answer  : n(d,e)
non_rec : q(e,X1)
answer  : q(e,a)
answer  : n(c,a)
NEW_ANSWER stored
answer  : p(c,b)
propag. : psg(id1,c,X1,b)
rec     : n(b,X3)
  EXPAND  : n(b,X3)
  clause1 : n(b,X3) :- r(b,X3).
  non_rec : r(b,X3) ... no answers found
  clause2 : n(b,X3):-p(b,X4),
                   psg(id1,b,X3,X4),
                   n(X4,X5), q(X5,X3).
  non_rec : p(b,X4)
  answer  : p(b,c)
  propag. : psg(id1,b,X3,c)
```

```
  RIQ     : n(c,X5)
  storeFQ : FQ(n(X6,X7) :-
              psg(id1,X6,X7,X8),
              n(X8,X9), q(X9,X7).
incompl : n(b,X3)

recursive expansion finished ...

answers: n(c,a)
         n(d,e)

frozen_query(qid3, n(X1,X2),
             psg(id1,X1,X2,X3),
             n(X3,X4), q(X4,X2)).

propagation facts: psg(id1,b,X5,c).
                   psg(id1,c,X5,b).
```

Step 2: iteration steps

```
iteration step 1 ...
trying   : FQ(n(X1,X2):-
             psg(id1,X1,X2,X3),
             n(X3,X4), q(X4,X2).
instant.: psg(id1,b,X2,c)
ans1     : n(c,a)
non_rec  : q(a,X2)
answer   : q(a,i)
answer   : n(b,i)
NEW_ANSWER stored

iteration step 2 ...
trying   : FQ(n(X1,X2):-
             psg(id1,X1,X2,X3),
             n(X3,X4), q(X4,X2).
instant.: psg(id1,c,X2,b)
ans1     : n(b,i)
non_rec  : q(i,X2)
answer   : q(i,o)
answer   : n(c,o)
NEW_ANSWER stored

iteration step 3 ...
trying   : FQ(n(X1,X2):-
             psg(id1,X1,X2,X3),
             n(X3,X4), q(X4,X2).
instant.: psg(id1,b,X2,c)
ans1     : n(c,o)
non_rec  : q(o,X2) ... no answers found

iteration phase finished ...

answers: n(c,a)
         n(d,e)
         n(b,i)
         n(c,o)
```

All answers for n(c,X1) are found, using the recursive expansion in the first step which produces one Frozen Query with two different instantiations (PSG's), and three iteration steps which produce the remaining answers.

In the iteration step no new Frozen Queries have been generated in this example. No further recursive expansion is needed, as the Frozen Queries do not contain recursive predicates besides the RIQ's resp. DIQ's. However, both of these activities are needed when processing more complex recursions and are performed by the call of EXPAND_ALL_SUBGOALS.

## 5. COMPARISON WITH OTHER STRATEGIES

According to [Ban86] the performance of a recursive query processing strategy is greatly influenced by the following factors:

- the amount of duplication of work
- the size of the set of relevant facts
- the use of unary vs. binary intermediate relations.

Comparing our strategy with QSQR and the other strategies along these lines, we come to the following results:

- less duplicate work than other methods
  - very goal oriented due to the recursive top down control structure
  - less iteration (only when necessary, on Frozen Queries - QSQR iterates on any level and any subquery))
  - more efficient iteration (no answers are filled in twice - QSQR uses naive LFP iteration)
  - generalized repeated queries through subsumption
- the size of relevant facts is minimal (only answers, which can be used for the original query are generated)
- slightly larger administration overhead than other general strategies
- larger overhead than more specialized query processing strategies (for linear queries etc.)

Furthermore our strategy is complete for all kinds of recursive definitions and data.

The idea of preserving the search tree to enable plugging in new results at a deeper level is also described in [Smi86] and [McK81]. As distinct from these methods, our strategy does not keep a branch (path) of the tree unless it is involved in the derivation of a RIQ. Only the branches necessary for answer completeness are preserved.

Furthermore, both methods are designed for a tuple-oriented transfer of facts, whereas our approach is also suitable for a set-oriented processing of facts during the retrieval from a database, which is especially efficient, if the deductive system is connected with a relational database management system.

Additionally, as shown by [Ban86], the recursive control strategy of QSQR and our method avoids much duplicate resp. useless work compared to the methods proposed by [McK81] and [Loz85].

[Smi86] presents a good theoretical discussion of the problem of repeating queries. He gives some completion results to certain classes of algorithms which also can be adapted to RQA/FQI and extensions thereof ([Nej87]).

## 6. FURTHER WORK

Although RQA/FQI is very efficient in most cases, its performance degrades when the answer to a query can be generated using many different intermediate results (see example 6 in [Ban86a]). This redundancy can be removed if the algorithm can detect that the search tree is being expanded along a path which has been explored earlier. We are currently extending RQA/FQI in this direction.

Another topic is the comparison of the set of methods used by existing strategies. We are currently trying to further analyze these optimization methods (use of relevant facts, avoidance of duplication for different reasons, etc.) in a common frame-work based on the definitions used by RQA/FQI ([Nej87]).

An interesting direction in the context of linear recursive queries is the connection to methods for traversing directed graphs, for a formal approach to this topic see [Mar86].

## ACKNOWLEDGEMENT

I would like to thank Gerhard Fleischanderl, Markus Stumptner and Erich J. Neuhold for their comments on an earlier version of this paper. Georg Gottlob was an important partner discussing various ideas later used in this paper.

## REFERENCES

[Ban86] ... F.Bancilhon, R.Ramakrishnan: An Amateur's Introduction to Recursive Query Processing Strategies, Proc. of SIGMOD'86, Washington, May 1986, pp.16-52.

[Ban86a] ... F.Bancilhon, et al.: Magic Sets and Other Strange Ways to Implement Logic Programs, Proc. of 5th ACM SIGMOD-SIGACT Symp. on Princ. of Database Systems, 1986, pp.1-15.

[Boc86] ... J.Bocca, L.Vieille, et al.: Some steps towards a DBMS based KBMS, Proc. IFIP 10th World Computer Congress, Dublin, Sept. 1986, pp.1061-1067.

[Cer86] ... S.Ceri, G.Gottlob, L.Lavazza: Translation and Optimization of Logic Queries: The Algebraic Approach, Proc. VLDB '86, Kyoto, August 86, pp.395-402

[Clo81] ... W.F.Clocksin, C.S.Mellish: Programming in Prolog, Springer, 1981

[Han86] ... J.Han, H.Lu: Some Performance Results on Recursive Query Processing in Relational Database Systems, Proc. of Intl. Conf. on Data Engineering, Los Angeles, Feb.1986, pp.533-541.

[Hen84] ... L.J.Henschen, S.A.Naqvi: On Compiling Queries in Recursive First-Order Databases, JACM 31/1, Jan. 1984, pp.47-85.

[Ion86] ... Y.E.Ioannidis: On the Computation of the Transitive Closure of Relational Operators, Proc. VLDB '86, Kyoto, August 86, pp.403-411

[Loz85] ... E.L.Lozinskii: Evaluating Queries in Deductive Databases by Generating, IJCAI 85, Los Angeles, Aug. 1985, pp.173-177

[Mar86] ... A.Marchetti-Spaccamela, A.Pelaggi, D.Sacca: Worst-case Complexity Analysis of Methods for Logic Query Implementation, ESPRIT Report 1117, 1986

[McK81] ... D.P.McKay, S.C.Shapiro: Using Active Connection Graphs for Reasoning with Recursive Rules, Proc. of 7th IJCAI, Vancouver, Aug. 1981, pp.368-374.

[Nej86a] ... W.Nejdl, E.J.Neuhold: The PROLOG-DB System: Integrating Prolog and Relational Databases, Technical Report, TU Vienna, and ÖGAI-Journal 5/1, June 1986

[Nej86b] ... W.Nejdl, G.Fleischanderl: QSQR Revisited - Incompleteness, Causes and Improvements, Technical Report, TU Vienna, December 1986

[Nej87] ... W.Nejdl: Describing Recursive Query Optimization in a Uniform Frame-Work, in preparation

[Ras86] ... L.Raschid, S.Y.W.Su: A Parallel Processing Strategy for Evaluating Recursive Queries, Proc. VLDB '86, Kyoto, August 86, pp.412-419

[Smi86] ... D.E.Smith, M.R.Genesereth, M.L.Ginsberg: Controlling Recursive Inference, Artificial Intelligence 30/3, Dec. 1986, pp.343-389

[Ull85] ... J.D.Ullman: Implementation of Logical Query Languages for Databases, TODS 10/3, 1985, pp.289-321.

[Vie86] ... L.Vieille: Recursive Axioms in Deductive Databases: The Query/Subquery Approach, Proc. of the 1st Intl. Conf. on Expert Database Systems, Charleston, S.C., April 1986, pp.179-193.

[Vie87] ... L.Vieille: Database-Complete Proof Procedures Based on SLD Resolution, Proc. of the 4th Intl. Conf. on Logic Programming, Melbourne, Australia, May 1987