On the Evaluation Strategy of EDUCE

Jorge Bocca

ECRC

Arabellastr 17

D-8000 Muenchen 81

West Germany

Abstract

Educe is a logic programming system for handling large knowledge bases It was constructed by fully integrating the logic programming language Prolog and the relational data base management system Ingres Educe uses a hybrid strategy for the evaluation of queries This strategy is based on two contrasting strategies The strategy known as *sets retrieval*, transforms recursive and non-recursive queries into a form suitable for evaluation by a relational data base management system The other strategy, known as *one-tuple-at-a time*, evaluates queries by imitating the evaluation strategy of the programming language Prolog In earlier versions of Educe, users selected the strategy by using two different query languages In order to remove this responsibility from the user, algorithms to map expressions from either of the languages into the other were implemented and added to Educe This paper briefly reviews the implementation of both evaluators and the mappings compares the basic strategies of evaluation, and then proceeds to explain Educe's own strategy

1 Introduction

This paper discusses the design and implementation of Educe a logic programming system capable of handling large knowledge bases Educe uses a hybrid strategy for the evaluation of queries This strategy is based on two contrasting strategies The strategy known as *sets retrieval* transforms recursive and non-recursive queries into a form suitable for evaluation by a relational data base management system (RDBMS) The other strategy, known as *one tuple-at a time* evaluates queries by imitating the evaluation strategy of the programming language Prolog prolog

A number of alternatives for coupling/integrating Prolog and a relational DBMS are presented and discussed in [kb9, Ston85, Vass84, Venk85, cpdb, Zaniolo, SciWarr] Educe was constructed by the coupling/integration of a deductive component and an external data base (EDB) component [kb9] The programming language Prolog is at the centre of the deductive component, and the relational DBMS Ingres ingres was used for the EDB component [Gall85]

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise, or to republish, requires a fee and/or specific permission

© 1986 ACM 0-89791-191-1/86/0500/0368 $00.75

At the top level, Educe offers users two different languages one following the non-procedural style of data manipulation language (DML) for RDBMSs, and one with a style close to Prolog We refer to these languages as *loose* DML and *close* DML, respectively Expressions in these languages can be freely mixed in Educe programs In terms of implementation, there is a close correlation between these languages and the evaluation strategies outlined in the previous paragraph It seems natural to use the *sets retrieval* strategy for the loose DML and the *one tuple-at-a time* strategy for the close DML

Initially, Educe used a coupling between a Prolog interpreter and a relational DBMS for the implementation of the loose DML Because of the problems that the evaluation of recursive queries causes [Bocca 85] in a coupled system, the close DML was implemented by integrating the low level access mechanism of the DBMS into the Prolog interpreter Although, these two approaches might be thought antagonistic to each other, in Educe they co-exist and co-operate

More recently algorithms to map expressions from either of the languages into the other have been implemented This allows Educe to decide for itself which is the (likely) best strategy for evaluation of a given query

The motivations for the two languages in Educe and their particular syntax are discussed in detail in [Bocca 85] This paper first presents a short review of the architecture of Educe, then describes its implementation and finally discusses issues of performance

The paper is divided into five sections Section 1 is this introduction In section 2 an outline of the architecture of Educe is given Section 3 discusses the implementation of the loose and the close languages, the handling of those rules stored in the EDB and the mapping algorithms An examination of the efficiency of Educe's particular implementation is undertaken in section 4 Finally in section 5 conclusions are presented and future work discussed

2 Architecture

In this section an outline of the chosen architecture for Educe and the motivations behind it are presented A detailed discussion of possible alternative architectures for Educe can be found in Bocca 85]

From the point of view of the implementor loose coupling presents itself as an obvious method for implementation Provided that recursion is not allowed a simple way to construct a loosely coupled system is by setting up two processes one for the deductive component and one for the EDB component These two processes exchange messages, i e queries and replies through a channel of communication Educe follows this approach for loose coupling, setting up one process for Prolog as the deductive component, and one process for Ingres as the EDB component Communication between the Prolog and the Ingres processes is by mean of two pipes [unix 83], one for queries and one for replies [Fig 1]

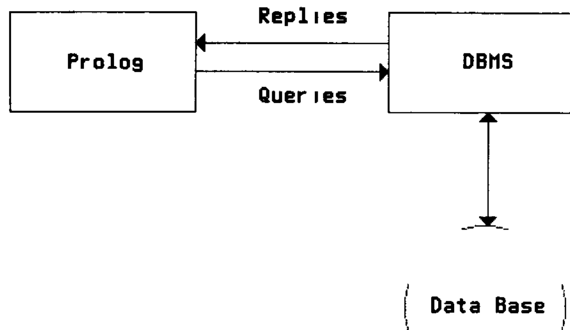

Figure 1

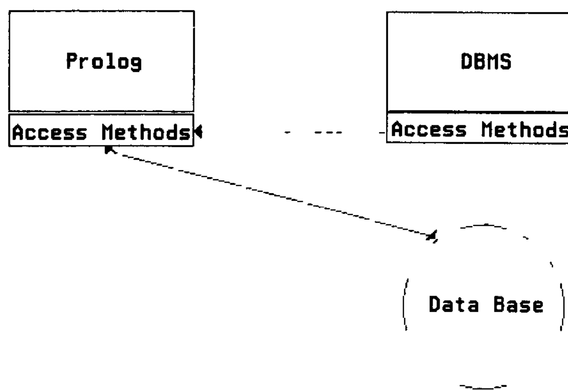

Figure 2

Unfortunately, the two processes in loose coupling would be very inefficient for an implementation of the close DML in Educe This is apparent in systems that have adopted this as a solution [frog 85, Naish 83] Because of this, we chose to integrate the deductive and the EDB components into one monolithic unit to handle the close DML For this, the ***access methods*** module of the DBMS was detached from it and attached to Prolog [Fig 2]

This allowed the multiple process configuration of loose coupling [Fig 1] to be merged with the close integration configuration [Fig 2] in a particularly coherent way To explain this, let us start by considering two concurrent processes, each of which runs the DBMS on a common data base [Fig 3]

When this configuration [Fig 3] is merged with the two previous ones [Figs 1 and 2] produces Educe's arhitecture [Fig 4] In the configuration depicted by Fig 3, one of the occurences of the DBMS is replaced by the Prolog+AM configuration [Fig 2] This is possible, since the Access Methods module of the Prolog+AM and the DBMS are identical replicas In other words, Educe appears as two concurrent DBMS's sharing access to a common data base

It is important to note that this architecture does not impose any restrictions on recursion On the contrary, it provides an efficient mechanism for the evaluation of multiple and recursive queries in either of the two languages, close and loose Recursive definitions which include expressions in loose form are evaluated by a hybrid strategy An evaluator has been implemented for this purpose The evaluator uses loose coupling for the non-recursive part of the definition, and then, for the recursive part, it uses the route provided by close integration for retrievals from the intermediate results Recently, a module which performs mappings from expressions in loose form into close form and vice versa, has been built This module allows Educe to select a route, either coupling or integration, entirely on the basis of expected performance

3 Implementation

The description of the implementation of Educe here presented follows the historical development of the system In order to avoid dismantling the deductive and the EDB components loose coupling was implemented in the first instance The integration phase was postponed until we had gathered sufficient detail of the construction of these components Rule storage and the transformation of expressions can be seen as important extensions of the basic capabilities of Educe

3 1 Loose Coupling

Let us begin with the discussion of a relatively simple part of the implementation the part that deals with loose coupling without recursion This part of Educe was implemented as two related processes [Fig 1] One process acting as a master runs the Prolog interpreter while the second process the server runs the DBMS The Prolog interpreter used is a derived product of the Mu-Prolog interpreter [Naish 83] The Ingres DBMS [Stonebraker 76] was chosen as the EDB component Thus, in this set up, whenever the evaluation of a goal requires access to the EDB, all expressions requiring some form of syntactic analysis are parsed, and code is generated for them by the Prolog interpreter The code generated is the equivalent QUEL expression This QUEL expression the query, is sent via a pipe to Ingres Ingres in turn evaluates the query and produces a reply This reply is piped back to Prolog which further processes it to bind variables to their respective values In this part of the implementation, the control of processes and communication between them was written in C, while the parsing of queries and code generation was all done in Prolog

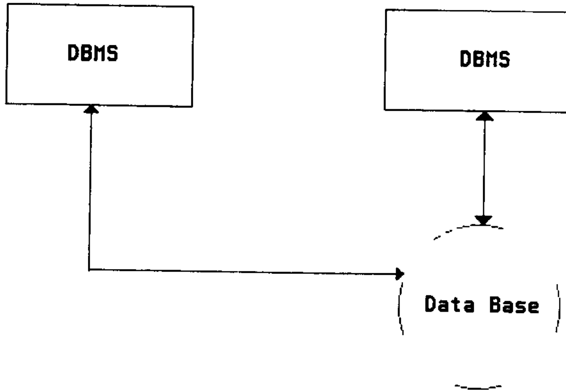

Figure 3

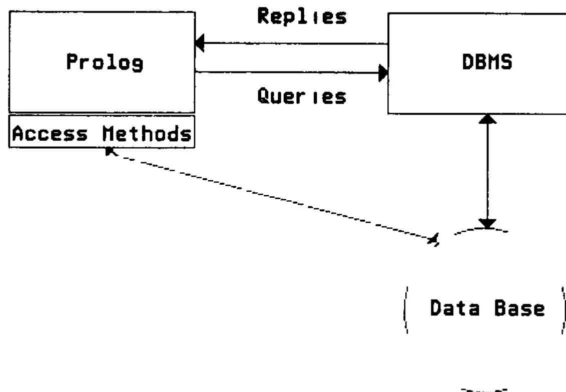

Figure 4

This scheme permits a more refined and efficient control of synchronization and communication between the processes

The predicate *helpdb* is perhaps the simplest example of the theory of operation described above. This predicate is defined by the Prolog clause

```
helpdb -
        query(' help ')
```

The predicate *query(anIngresQuery)* takes the atom *anIngresQuery* and sends the string of characters forming the name of the atom down a pipe to Ingres. Then it waits for the evaluation of the query by Ingres and on completion returns *true*. Thus, by means of this mechanism, any arbitrary Ingres query can be sent to the server for evaluation. The predicate *query* was written in C and has been integrated into the Prolog interpreter

A more complex situation develops when the mode of operation of Prolog differs from the mode of operation of the FDB. Take the case of *retrieve*

```
retrieve( Atts, Boolean )  -

    send_query( Rels, Atts, Boolean ),

    repeat,
    rel( P ),
    ( P = E,
       (  E = continue,
          !,
          fail
       )

       ,
       P =    [ | OutAtts],
          value( Atts, OutAtts )
    )
```

Following the parsing of Atts, a query is sent to Ingres via *send_ query*. Typically, Ingres produces a whole relation as an answer to the query. Since Prolog requires only one tuple at a time, some adjustments have to be made. Basically, Ingres pipes the result relation to Prolog while Prolog takes one tuple each time from the pipe. In other words the pipe acts as a queue. To take one tuple from the pipe *rel(P)* is called. *P* is compared against the atom *continue* to check for the end of reply from Ingres. If it is not the end of reply (*continue*), then the tuple *P* is passed in suitable form to *value* which binds variables to attribute values

It should also be mentioned that the syntax of *retrieve* allows uninstantiated variables in the conditions (*Boolean*) argument. Because of this, it is desirable to delay evaluation of the *retrieve* until the search criteria have been clearly established, so avoiding retrieval of unnecessary data

3 2 Close Integration

It is not only for syntactic convenience that variables are necessary in the condition part of *retrieves*. Without them, it would be impossible to express recursion. Take for example the relation *parent(X, Y)*, defined by

```
parent( X, Y)  -
    (  var(X), var(Y), ',
       retrieve([ father is_ = X,
          father of_ = Y],
       true)
    )
    ,
    ( atom(X), atom(Y), ',

    )

parent( X, Y)  -

       retrieve([ mother is_ = X,
          mother of_ = Y],
       true)
```

Then we could define the derived relation *ancestor(X, Y)* recursively by

```
ancestor( X, Y)  -
   parent( X, Y)
ancestor( X, Y)  -
   parent( X, Z),
   ancestor(Z, Y)
```

Unfortunately, the introduction of recursion and multiple queries brings new problems It becomes necessary to relate replies to their originating queries In order to evaluate a recursive definition such as *ancestor*, Educe first generates the relation *parent* (if virtual) and then proceeds to evaluate the recursive clause by using close access But, before we discuss the details of how this is done in Educe a description of the implementation of *close integration* is needed The particular reasons for having close integration in Educe are discussed in [Bocca 85]

Because of its linkage to the low level *access methods*, the close DML is implemented mainly in C At the top level, the Prolog predicate *retr* binds the C implemented parts together The implementation of this predicate is presented below

```
retr( R )  -
   R =    [Rel | Atts],
   openr(D, O, Rel),
   setsearch( D, Atts),
   repeat,
   getvals(D, Atts, NewTuple),
   (  (   /* failed */
          NewTuple = O,
          closer(D),
          ', fail
      )
      ,
      true
   )
```

An example of the use of *retr* is *? retr(employee(john, Salary))* In this example, the relation employee is searched for *john*'s *Salary*

The program above starts by transforming the (only) argument of *retr* into a list The head of the list is instantiated to the name of the relation (*Rel*) and the tail is instantiated to a list of attributes (*Atts*), some as variables and the others as constants, according to the particular retrieval condition In our example, this list becomes *[employee, john, Salary]* Once this is done, the relation *Rel* is opened by *openr* and the searching keys are set by *setsearch* Only then the first tuple (if any exists) is retrieved by a call to *getvals* The call to *repeat* is necessary to handle backtracking The predicates *openr*, *setsearch*, *getvals* and *closer* are all implemented in C The call to *openr* opens the file which contains the relation *Rel* and it also creates a descriptor, *D* If the file for relation *Rel* was already open (for reading) then only the new descriptor D is created

Descriptors not only keep static information about a relation, e g file name, degree, cardinality, etc, but they also maintain information of a dynamic type In particular, information about the last tuple accessed is kept by the descriptors (TID of last tuple [Stonebraker 76]) This is essential for an efficient implementation of backtracking Otherwise, Educe would need to re-access old tuples to get the next tuple This use of descriptors is essential in recursive cases Without the descriptors, recursive queries on a given relation would be restricted to as many levels of recursion as the numbers of files that the host operating system allows to keep opened at any particular time As an additional bonus to the scheme of operation described here, the overhead of opening and closing files is greatly reduced In fact, for recursive queries, this overhead is reduced to practically nothing

3 3 Rules

Once the ability to handle large numbers of facts by the EDB component had been installed in Educe, the next logical step was to introduce facilities to store and maintain large numbers of deduction rules in the EDB Thus a mechanism to serve this purpose was implemented In Educe, deduction rules are stored like the schema of the data base, in a relation This relation is named *rulerel*

Obviously, the storage of rules in the EDB is not in itself enough to achieve the desired effect Rules stored in the EDB must also be executed just like any other rule in main memory For this, the top level Prolog interpreter was modified Thus if, during the evaluation of a goal an appropriate clause head is not found for it in main memory then the *rulerel* relation is searched for it If a rule with such a head is found in the EDB, then Educe executes it If however the rule is not found in *rulerel* then Educe looks for a base relation to match the goal If such a relation is found then the rule

```
Relation  -
   retr( Relation)
```

is asserted More formally, to evaluate a given goal G the algorithm is

1 Search for rule fact in main memory

2 If 1 fails then search for rule in relation *rulerel*

3 If 2 fails then search for base relation with matching name and degree If such a relation is found then assert the rule *G retr(G)*

The algorithm above effectively makes the EDB component of Educe transparent to users of the close DML

Unfortunately, this scheme of operation is not free of some (minor) side effects The above algorithm implies an order of

evaluation for rules and facts In this implied order of evaluation, rules precede facts However, since facts are a special case of rules (no body), they can be treated like a rule if so wanted by the user

Still on the subject of evaluation precedence, a more important point is to note that Prolog inspects facts in a program in a top-down manner In relational data base terms, this implies an ordering in the tuples of a given relation This contradicts the definition of a relation To avoid the problem, Educe adopts the semantic of *assert* when inserting tuples in a relation Equivalences for *asserta* and *assertz* are purposedly excluded from Educe However, this is not sufficient in the case of general rules To keep close to Prolog semantic, users are asked to specify an order of evaluation for the rules kept by the EDB For example, to add the definition of *anc* to the EDB component of Educe, one should proceed as follow

```
?- nrule( 1,
    'anc(X,Y)  - parent(X,Y) ')
?- nrule( 2,
    'anc(X,Y)  - parent(X,Z), anc(Z,Y) ')
```

Once this is done, the EDB component of Educe becomes transparent to those users accessing the derived relation *anc*

Finally, a point that has some bearing on efficiency and integrity the evaluation algorithm described above retrieves rules from *rulerel* not only when the rule is activated for the first time, but also when backtracking takes place From an efficiency point of view, this is a serious drawback Also from the point of view of integrity, backtracking can cause some problems In particular, the answers to a query would not be correct if another user were allowed to update part of the necessary definitions while they were still being used Educe solves these two problems by pre-processing the top level query Thus, given a goal to evaluate, Educe builds the whole evaluation tree for this query Rules are then retrieved from the *rulerel* and the necessary *retr*'s rules are also asserted Only when all this information has been obtained from the EDB Educe proceeds to evaluate the query Effectively, the EDB is only consulted once for the necessary rules, and all the definitions needed are frozen during the evaluation of the goal Notice that with this scheme, other users are not prevented from updating the non-factual knowledge For the factual knowledge (base relations) the EDB uses normal data base techniques for concurrent access to relations

3 4 Mappings

Now we can go back to our example *ancestor* [sub section 3 2 and see how recursive (and multiple) queries in loose form are handled in Educe First, let us examine the program *$slowretr* below This program is a preamble to a simple but not very efficient implementation of a query evaluator for the loose DML However, this program is capable of handling multiple and recursive queries

```
/* $slowretr -
   it uses same syntax as retrieve
   in Atts and Boolean
*/

$slowretr( Rels, Atts, Boolean) :-

    /* Rels list is obtained
       from Atts and Boolean */

    $q_and_s( IntRes, Rels,
            Atts, Boolean),
    !,        /* never backtrack */
    $quickretr( IntRes, Atts)
```

In this program, once the list of base relations *Rels* has been extracted from *Atts* and *Boolean*, the call to *$q_and_s* prepares a query in loose form and executes it, saving the result in the intermediate relation *IntRes* We do not want to backtrack past this point, hence the cut (!) It is now up to *$quickretr* to produce the answer(s), one tuple at a time This is done by *$quickretr* by querying the intermediate relation *IntRes* using the close DML Finally, *$quickretr* matches the values in the returned tuple (close DML) to the non-positional projection specified by *Atts*

The first and obvious problem in this strategy of evaluation is one of efficiency In the case of recursive definitions, each time we backtrack on the non-recursive part of a definition, a new intermediate relation will be generated This is easily solved though, by labeling the queries already answered with the name of the intermediate relation generated for it Thus before proceeding to generate a new intermediate relation, we check whether the intermediate relation has been generated for the (intermediate) query The program for this version of *retrieve* is given below To stress the fact that this program also handles multiple relations and recursive definitions, we call it *mretrieve*

```
/* mretrieve -
   handles multiple relations and
   recursive definitions in the
   loose DML
   It uses same syntax as retrieve */

mretrieve( Atts, Boolean)  -
    $evaluated( Res, Atts, Boolean)
       -> $quickretr( Res, Atts)
       ,  $slowretr( Res, Atts, Boolean)
```

As can easily be imagined there are occasions when the above strategy to evaluate recursive queries can produce very slow responses This problem is addressed in detail in the next section

4 Efficiency

As was pointed out in [Bocca 85], users expect in the context of a Prolog interpreter to obtain a reply quickly This reply normally corresponds to just one tuple in a relation (base or derived) During backtracking the same still holds true By contrast, in a relational DBMS, users expect a whole relation, i e a set of tuples, to be generated as an answer This dichotomy between the two types of system leads to two different types of evaluation strategy for queries

Because DBMS evaluation techniques have been developed over a longer period of time, it is generally believed that they are more efficient than Prolog-type techniques This belief is questionable, particularly in the case of recursive queries A

typical DBMS evaluation strategy is akin to batch processing, e g the emphasis on the generation of a set of tuples (a relation) as the result In direct contradiction to this bias of DBMSs, a Prolog-type of system assumes an interactive environment In this latter case, answers (or part of them) should be delivered as small units, so that users can logically grasp them Hence, the smaller granularity of a "Prolog" answer, i e normally one tuple at a time

In order to compare these two different modes of evaluating a query, a number of tests were performed Equivalent queries in loose and close form were prepared and issued for evaluation by Educe The queries were graded according to expected difficulty and were run on relations containing 50000 randomly generated tuples The time taken by Educe to evaluate each query is given in Table 1

As the figures in Table 1 show, it is not unusual for the *one-tuple at a time* strategy (*method 3*) to outperform the *sets retrieval* strategy of DBMSs (*method 1*) This is particularly true in the case of recursive queries The cases in which the *sets retrieval* strategy is the winner are normally explained by the way in which Prolog selects sub-goals for evaluation [Bocca 85] This sometimes leads to the choice of the wrong index, which is then used until it gets exhausted, and only then the correct one might be chosen and used To demonstrate this, query B in run 7 was re-formulated so as to favour an early selection of the correct index in the evaluation of the close query

In loose coupling, queries are handed to the EDB component for evaluation This is in effect an evaluation of queries by a DBMS (*method 1*) In all cases of queries involving several relations and/or recursion and in many cases of queries involving a single relation, e g aggregation, a number of intermediate relations are generated during the evaluation of these queries It is the creation and manipulation of these intermediate relations that is the cause of major overheads in the evaluation of queries in loose form, particularly in the recursive case Even if large buffers in main memory were used for these intermediate relations, it would still be necessary to use secondary memory (slow) to store considerable parts of these relations Also there is an overhead attached to the creation and maintenance of the schemas for these relations

The *one tuple at a time* strategy of Prolog does not need to create intermediate relations (*method 3*) All intermediate results are kept at all times in main memory This is only possible because of the relatively small size of the intermediate results required by the *one tuple at a time* strategy In other words retrieval of data from secondary memory only occurs when base relations are consulted

However, as was discussed in [Bocca 85], there are good reasons for using a loose DML to express queries Also as shown by a significative number of cases in Table 1 there are situations where the *sets retrieval* strategy of DBMSs outperforms the *one tuple at a time* strategy of Prolog In Educe all of these reasons are considered to be important and hence queries in loose form are supported Moreover, a number of optimization techniques are applied to queries in loose form, in order to improve performance In addition to the optimization techniques of the DBMS, Educe uses its own techniques Particular attention is given to the recursive case, since this is an area outside the scope of conventional DBMSs Four significant cases are here discussed

The first case arises in queries involving one base relation and the boolean condition *true* For example, consider the relation *employee* with attributes *name, address and dept*, and the goal

```
?- retrieve( [employee name = Name,
         employee dept = Dept],
     true)
```

This query is transformed into the equivalent "query" in the close DML

```
employee( Name, Dept)  -
   retr( employee( Name, Dept, _))

?- employee( Name, Dept)
```

This example is generalized to the case of any base relation being queried with the boolean condition set to *true* The built-in predicate *$whole_ base_ r* makes the appropriate tests to decide on the applicability of this transformation rule Although the case described seems trivial and unlikely to be presented to Educe by users, it often arises as an intermediate step in the evaluation of a recursive definition For instance, in the loose DML version of our definition of *parent* [section 3 2], given the goal

```
/* see 3 2 */
?- ancestor( X, charles)
```

when the recursive clause is used, the goals to solve are *parent(X,Z)* and *ancestor(Z,charles)* Since *X* and *Z* are not instantiated, to solve *parent(X,Z)* the goal *retrieve([parent father is_ =X,father of_ =Y],true)* would have to be solved

The second case for efficiency improvements occurs again very frequently in recursive queries This is the case of intermediate queries that have already been evaluated, as discussed in 3 4, above It should also be said here that, although this situation often arises during the evaluation of recursive queries, it is not exclusive to them

The third case of importance occurs in conjunctive queries on a single relation Again, this is a very common situation during the evaluation of recursive queries (top level) In this case the conjunctive query is first transformed into a normal form and then from this normalized form, an equivalent query in close form is generated and evaluated

The last but certainly not the least major optimization step takes place during the transformation of loose form into close form and during the instantiation of variables at the top level For both of these processes it is necessary to access the schema of the relation involved These accesses are speeded up by maintaining the data base schema in buffers in main memory Obviously, some synchronization with the copies in secondary memory is necessary This is a common solution in a conventional DBMS and Educe has adopted it More seriously though whenever a tuple is retrieved from a base relation a number of variables have to be instantiated To do this, the list of values in the retrieved tuple has to be matched with a list of variables (typically, the variables in the projection part of a *retrieve*) The list of variables is normally shorter than the list of values and their sequences do not match For instance, a typical tuple in our relation *employee* might be *[john, munich, toys]* and the goal might be *retrieve([employee dept=Dept, employee name=Name],)* Obviously, the order and the length of the lists *[Dept, Name]* and *[john, munich, toys]* do not match In general to match the two lists every time a new tuple is retrieved is unnecessary The problem is avoided by, firstly, creating a bogus list of variables, say *[X1, X2, X3]*, then matching this list only once to the projection list in the *retrieve*, and finally, each time a

Query \ Run->	1	2	3	4	5	6	7	8
1Aa	260	258	268	279	339	275	425	274
3Aa	3	5	4	4	4	14	-	3
1Ab	240	339	235	219	343	240	223	231
3Ab	16	12	15	14	12	25	-	11
1Ac	256	264	252	257	314	329	224	259
3Ac	13	436	11	11	10	22	-	12
1Bc	527	475	502	506	12316	12932	11835	11853
3Bc	581	34	857	633	327	322	327	320
1Cc	10	235	10	11	319	250	248	236
3Cc	5	8	4	4	5	5	8	1
1Dc	-	254	236	228	233	243	241	280
3Dc	-	4	2	2	2	2	1	2

+ Times are given in 100th's of a second

Table 5-1 Comparisons of Methods DBMS and Prolog

Query \ Run->	1	2	3	4	5	6	7	8
1Aa	260	258	268	279	339	275	425	274
2Aa	6	6	6	7	6	17	4	5
3Aa	3	5	4	4	4	14	-	3
1Ab	240	339	235	219	343	240	223	231
2Ab	5	3	5	4	5	7	4	6
3Ab	16	12	15	14	12	25	-	11
1Ac	256	264	252	257	314	329	224	259
2Ac	3	3	4	3	4	3	4	3
3Ac	13	436	11	11	10	22	-	12
1Bc	527	475	502	506	12316	12932	11835	11853
2Bc	8	8	8	8	927	917	912	909
3Bc	581	34	857	633	327	322	327	320
1Cc	10	235	10	11	319	250	248	236
2Cc	6	4	4	4	4	3	4	4
3Cc	5	8	4	4	5	5	8	1
1Dc	-	254	236	228	233	243	241	280
2Dc	-	4	4	5	4	6	4	4
3Dc	-	4	2	2	2	2	1	2

+ Times are given in 100th's of a second

KEYS
****

```
        METHOD  1  Sets Retrieval (DBMS)
                2  Educe's for loose DML
                3  One-Tuple-At-A-Time (Prolog)

        TASK    A  Simple Selection
                B  Selection + Projection + Join
                C  Simple Recursion
                D  Difficult Recursion

        SIZE    a  Small -         10 tuples
                b  Fair -        1000 tuples
                c  Large -      50000 tuples
```

The test programs were run eight times on a VAX 785 Constant values were changed from one run to the next Also, in some runs the arguments were less (more) instantiated than in earlier runs For further details of the tests, see Appendix

Table 5-2 Comparisons of Methods DBMS, Educe and Prol

tuple is retrieved from secondary memory, this bogus list is used to instantiate the real variables A lot of unnecessary sorting is eliminated at a stroke!

The optimization steps described above have in fact led to a new strategy, the *Educe method* This method integrates all the above optimization steps into a harmonized strategy of evaluation for queries expressed in loose form The improvements obtained by the application of this method go well beyond the recursive case In fact, the *Educe method* always give a performance close to the best of the other two methods (DBMSs and Prolog) This is clearly shown by the performance figures obtained in comparative tests of the three methods [see Table 2]

5 Conclusions

From a logical point of view two contrasting styles of query language are used in Educe One is similar to query languages found in conventional DBMSs and the other follows the conventions of the programming language Prolog According to their closeness to Prolog, they are referred to as *loose* and *close* DML, respectively Both DMLs are necessary and complementary to each other

The form of each language strongly suggests its mode of evaluation The strategy known as *sets retrieval* used by conventional DBMS seems, at first, the most appropriate for the loose DML Similarly, the *one query-at a time* strategy of Prolog seems more suited for the close DML However, our work on Educe points to a mixed strategy, specially in the case of queries expressed in loose form

The *sets retrieval* strategy of conventional DBMSs produces a heavy overhead in the recursive case (and also in other non recursive cases) The main problem is the creation and maintenance of relatively large numbers of temporary relations By contrast, the *one tuple at a time* close integration strategy of evaluation does not require intermediate relations This acquires special importance in the case of recursive queries Precisely because of this a capability to map expressions from one language into the other according to performance requirements was built into Educe In particular recursive queries using conjunctive conditions benefit from the transformation of loose into close form Nevertheless one should be aware of the limitations to possible optimizations that the Prolog order of evaluation (for sub-goals) imposes This is indeed one more argument for a close DML based on pure Horn clauses where a more flexible evaluation order could be adopted

References

[Appelrath 85] Appelrath, H-J , Bense, H and Rose, T
CPDB - A Data Base based Prolog System Incorporating Meta-Knowledge
Unpublished , September, 1985
ETH Zurich, Dept of Computer Science, Switzerland

[Bocca 85] Bocca, J B
EDUCE - A Marriage of Convenience Prolog and a Relational DBMS
Internal Report KB-9, European Computer-Industry Research Centre, Munich, September, 1985

[frog 85] Ducasse, M , Faget, J and Gumbach, A
FROG - Implementation of a Language merging Functional Relational Programming Styles, via an Object Type Driven Evaluation
1985
Laboratories Marcoussis

[Gallaire 85] Gallaire, H
Logic Programming Further Developments
In *Proc 1985 Symposium on Logic Programming* pages 88-96 Boston, USA, July, 1985

[Naish 83] Naish L
MU PROLOG 3 0 reference manual
Melbourne University Computer Sc , Melbourne, Australia, 1983

[prolog 81] Clocksin, W F and Mellish, C S
Programming in Prolog
Springer-Verlag, Berlin-Heidelberg-New York, 1981

[SciWarr 84] Sciore L and Warren D S
Towards an Integrated Database-Prolog System
In *Proceedings First International Workshop on Expert Database Systems* pages 801 815 Kiawah Island South Carolina, USA October 1984

[Stonebraker 76] Stonebraker M Wong E Kreps P and Held G
The Design And Implementation Of Ingres
ACM Transactions on Database Systems 1(3) 189 222 September 1976

[Stonebraker 85] Stonebraker M
Inference in Data Base Systems Using Lazy Triggers
In *Proc of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, pages 295-310 Islamorada, Florida, USA, February, 1985

[unix 83] *Unix Programmer's Manual*
4 2 Berkeley Software Distribution (copyright 1979, Bell Telephone Laboratories, Inc) edition, Dept of Electrical Engineering and Computer Science, University of California, Berkeley, California, USA, 1983

[Vassiliou 84] Vassiliou, Y , Clifford, J and Jarke, M
Access to Specific Declarative Knowledge by Expert Systems The Impact of Logic Programming
Decision Support Systems 1(1), 1984

[Venken 85] Venken, R
The Interaction between Prolog and Relational Databases
Unpublished , Early, 1985
Report on ESPRIT Pilot Project 107

[Zaniolo 84] Zaniolo, C
Prolog a Database Query Language for All Seasons
In *Proceedings First International Workshop on Expert Database Systems*, pages 63 73
Kiawah Island, South Carolina, USA, October, 1984

```
/*********************************
   TEST PERFORMANCE OF EDUCE
*********************************

SCHEMA
*******

parent( anc, child)
  + entered by hand +
   pkey   -- ISAM on anc
   Tuples                -- 10
   Recursion Levels-- 9

parenta( name, father, mother)
  + random generator +
   pkey   -- HASH on name
   indexes
      x2parenta - ISAM
         on [father, name]
      x3parenta - ISAM
         on [mother, name]
   Tuples -- 1000
   Recursion Levels-- 3

parentb( name, father, mother)
  + random generator +
   pkey -- HASH on name
   indexes
      x2parentb - ISAM
         on [father, name]
      x3parentb - ISAM
         on [mother, name]
   Tuples                -- 50000
   Recursion Levels-- 10


Method
******
(Explicit use of mretr, fretr
and retr to force use of wanted
method of evaluation)

1  Uses straight DBMS techniques
   (Loose coupling)
2  Loose --> Close DML and optimizes
   new expression
3  Close Integration


Task
****

Query A   Simple Selection
Query B   Selection + Projection
          + Join
Query C   Simple Recursion
Query D   Difficult Recursion
          (slow in Prolog)


Size
****

Size a   10 tuples
Size b   1000 tuples
Size c   50000 tuples
*********************************/


/*
   Derived relations
```

```
Explicit use of indexes for better
control of test
*/

par1b( X,Y)  -
   /* derived from
   parenta relation
   1000 + 1000 tuples */

par1c( X,Y)  -
   /* derived from
   parentb relation
   50000 + 50000 */

par2b( X,Y)  -
   /* derived from
   parenta relation
   Method 2
   1000 + 1000 tuples */

par2c( X,Y)  -
   /* derived from
   parentb relation
   Method 2
   50000 + 50000 tuples */

par3b(X,Y)  -
   /* derived from
   parenta relation
   Method 3
   1000 + 1000 tuples */

par3c(X,Y)  -
   /* derived from
   parentb relation
   Method 3
   50000 + 50000 tuples */

brother1Bc( X, Y)  -
   par1c(X, P),
   par1c(Y, P),
   X ~= Y
brother2Bc( X, Y)  -
   par2c(X, P),
   par2c(Y, P),
   X ~= Y
brother3Bc( X, Y)  -
   par3c(X, P),
   par3c(Y, P),
   X ~= Y

anc1( X, Y)  -
   par1c(X, Y)
anc1(X, Y)  -
   par1c(X, Z),
   anc1(Z,Y)

anc2( X, Y)  -
   par2c(X, Y)
anc2(X, Y)  -
   par2c(X, Z),
   anc2(Z,Y)

anc3( X, Y)  -
   par3c(X, Y)
anc3(X, Y)  -
   par3c(X, Z),
   anc3(Z,Y)


/***********************
        QUERIES
```

```
***********************/
/***************
Case A - Select
***************/
/*
   Method 1
   mretr -- is basic DBMS method
      + obvious optimization steps
*/

query1Aa  -
   /* size a   10 tuples */
   write('1Aa -- '),
   etime(_),
   mretr([parent anc = Parent],
      parent child = mary),
   etime(X), writeln(X), '

query1Ab  -
   /* size b   1000 tuples */
   write('1Ab -- '),
   etime(_),
   mretr([parenta mother = Mother],
      parenta name = 235),
   etime(X), writeln(X), '

query1Ac  -
   /* size c  50000 tuples */
   write('1Ac -- '),
   etime(_),
   mretr([parentb mother = Mother],
      parentb name = 235),
   etime(X), writeln(X), '

/*
   Method 2
   fretr -- loose is mapped into close
      + high level optimization
*/
query2Aa  -
   /* size a */
   write('2Aa -- '),
   etime(_),
   fretr([parent anc = Parent],
   parent child = mary),
   etime(X), writeln(X), '

query2Ab  -
   /* size b */
   write('2Ab -- '),
   etime(_),
   fretr([parenta mother = Mother],
   parenta name = 235),
   etime(X), writeln(X), '

query2Ac  -
   /* size c */
   write('2Ac -- '),
   etime(_),
   fretr([parentb mother = Mother],
      parentb name = 235),
   etime(X), writeln(X), '

/*
   Method 3
   Close Integration (Prolog)
*/
query3Aa  -
   /* size a */
   write('3Aa -- '),
   etime(_),
   retr( parent( Parent, mary)),
   etime(X), writeln(X), '
```

```
query3Ab  -
   /* size b */
   write('3Ab -- '),
   etime(_),
   retr( parenta( 235, Father,
      Mother)),
   etime(X), writeln(X), '

query3Ac  -
   /* size c */
   write('3Ac -- '),
   etime(_),
   retr( parentb( 235, Father,
      Mother)),
   etime(X), writeln(X), '


/*****************
Case B  S + P + J
*****************/
/*
   A comparison of Methods 1, 2 and 3
   Join + Select + Project operators
   Only 1 size tested  50000 tuples
*/

query1Bc  -
   /* 1   basic DBMS method */
   write('1Bc -- '),
   etime(_),
   brother1Bc(248, 919),
   etime(X), writeln(X), '
query1Bc  -
   /* on failure */
   etime(X), writeln(X), '

query2Bc  -
   /* 2   Loose --> Close */
   write('2Bc -- '),
   etime(_),
   brother2Bc(248, 919),
   etime(X), writeln(X), '
query2Bc  -
   /* on failure */
   etime(X), writeln(X), '


query3Bc  -
   /* 3   Close DML */
   write('3Bc -- '),
   etime(_),
   brother3Bc(248, 919),
   etime(X), writeln(X), '
query3Bc  -
   /* on failure */
   etime(X), writeln(X), '


/***********************
Case C   Simple Recursion
***********************/
/*
   Compare Methods 1, 2 and 3
   Simple recursion
   Larguish size   50000 tuples
*/

query1Cc  -
   /* 1   DBMS method */
   write('1Cc -- '),
   etime(_),
   anc1(255, _),
   etime(X), writeln(X), '

query2Cc  -
   /* 2   Loose --> Close */
   write('2Cc -- '),
   etime(_),
   anc2(255, _),
   etime(X), writeln(X), '

query3Cc  -
   /* 3  Close DML */
   write('3Cc -- '),
   etime(_),
   anc3(255, _),
   etime(X), writeln(X), '


/*************************
Case D  Difficult Recursion
*************************/
/*
   Again compare methods 1, 2 and 3
   Hard recursion (for Prolog)
   Only largish size  50000
*/

query1Dc  -
   /* 1   DBMS method */
   write('1Dc -- '),
   etime(_),
   anc1( 255, 4095),
   etime(X), writeln(X), '

query2Dc  -
   /* 2   Loose --> Close */
   write('2Dc -- '),
   etime(_),
   anc2( 255  4095),
   etime(X), writeln(X), '

query3Dc  -
   /* 3   Close DML */
   write('3Dc -- '),
   etime(_),
   anc3( 255, 4095),
   etime(X), writeln(X), '
```