AN OPTIMIZING PROLOG FRONT-END TO A RELATIONAL QUERY SYSTEM

Matthias Jarke
James Clifford
Yannis Vassiliou

January 1984

Center for Research on Information Systems
Computer Applications and Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #65

GBA #84-24(CR)

Published in Proceedings ACM-SIGMOD International Conference on Management of Data, Boston, June 18-21, 1984.

AN OPTIMIZING PROLOG FRONT-END TO A RELATIONAL QUERY SYSTEM

Abstract

An optimizing translation mechanism for the dynamic interaction between a logic-based expert system written in PROLOG and a relational database accessible through SQL is presented. The mechanism makes use of an intermediate language that decomposes the optimization problem and makes the proposed approach target-language independent. It can either facilitate expert system - database interaction, e.g., when integrating expert systems into business systems, or augment existing database with (external) deductive capabilities.

1.0 INTRODUCTION

Efforts to bring together methods from artificial intelligence and database research have caused much interest in both communities. Cooperation between the areas can be fruitful in conceptual modelling [Brodie et al. 1984], in providing database data to expert systems, in supporting very high-level user interfaces for databases, and in improving database efficiency. In particular, the similarity between relational database concepts and logic-based deduction has focused the attention on integrating these two elements in various ways [Gallaire and Minker 1978; Gallaire et al. 1981; Nicolas et al. 1982].

In previous work [Jarke and Vassiliou 1984; Vassiliou et al. 1983, 1984], we investigated strategies for providing data management capabilities to expert systems. We presented a technique of 'tight coupling' between a logic-based expert system and a relational DBMS, employing delayed execution of database calls. Our implementation uses an amalgamation between a logic programming language (PROLOG) with a suitable meta-language of itself, expressed in a variable-free subset of the logic programming language [Bowen and Kowalski 1982]. We also postulated algorithms for the further processing, optimization, and translation into the database query language at hand of the generated database calls, without actually describing such algorithms.

In this paper, we generalize the notion of expert system--database coupling and describe an optimizing translation mechanism which allows for the continued efficient exchange of queries and/or data between a PROLOG-based expert system and a relational DBMS accessible through SQL. The mechanism is designed to enable portability to similar query languages such as QUEL [Stonebraker 1976] or PASCAL/R [Schmidt 1977]. It can be used regardless of whether one wants to provide database information to an expert system or to enhance the user interface or the efficiency of an existing conventional database system from 'outside' if one is unable or unwilling to extend the DBMS itself.

This work was carried out as part of a joint study being conducted with the IBM Corporation.

Our approach employs an intermediate language, DBCL. This language is a variable-free subset of PROLOG designed to be similar to tableaux as introduced in [Aho et al. 1979]. Attention in this paper will be focused on conjunctive, negation- and function-free queries but extensions to general DBCL predicates are also discussed.

The introduction of DBCL partitions the problem of efficient PROLOG-SQL translation into three major components: the translation of PROLOG data requests into DBCL statements; the syntactic and semantic optimization of such DBCL statements; and the translation of the optimized DBCL statements into queries expressed in the target language, in our case SQL.

Since we assume two independent subsystems to be coupled, our query optimization methods are somewhat different from those presented in previous research on PROLOG databases. For example, the kind of query optimization achieved by reordering PROLOG goals [Warren 1981] should be taken care of by the existing query processor of the DBMS. Our strategies focus more on DBMS-independent query simplification and multiple query optimization. The tableau-like structure of DBCL allows the application of results obtained by database theory, although it turns out that some of these results have to be extended for practical purposes.

The paper is organized as follows. Section 2 presents the global architecture of the proposed translation mechanism. After a definition of the DBCL subset to be used in this paper in section 3, section 4 briefly reviews the translation process from PROLOG to DBCL. The DBCL-SQL translation has been implemented in PROLOG using a syntax tree mapping approach (section 5). Section 6 demonstrates how syntactic (relational data structures) and semantic (integrity constraints) knowledge about the underlying database can be exploited for DBCL query simplification. Finally, section 7 presents an outlook of what can be done to support multiple query evaluation, including recursive database calls, through the creation and storage of suitable intermediate results.

2.0 ARCHITECTURE OF THE TRANSLATION MECHANISM

The basic problem in optimizing the interaction between PROLOG and a conventional relational query lanaguage is the translation of a series of tuple-oriented data requests, addressing parameterized and possibly recursive views in PROLOG, into (sequences of) set-oriented queries to base relations. Thus, an efficient translator must (a) collect tuple-oriented requests to form set-oriented queries, and (b) optimize the processing of parameterized views.

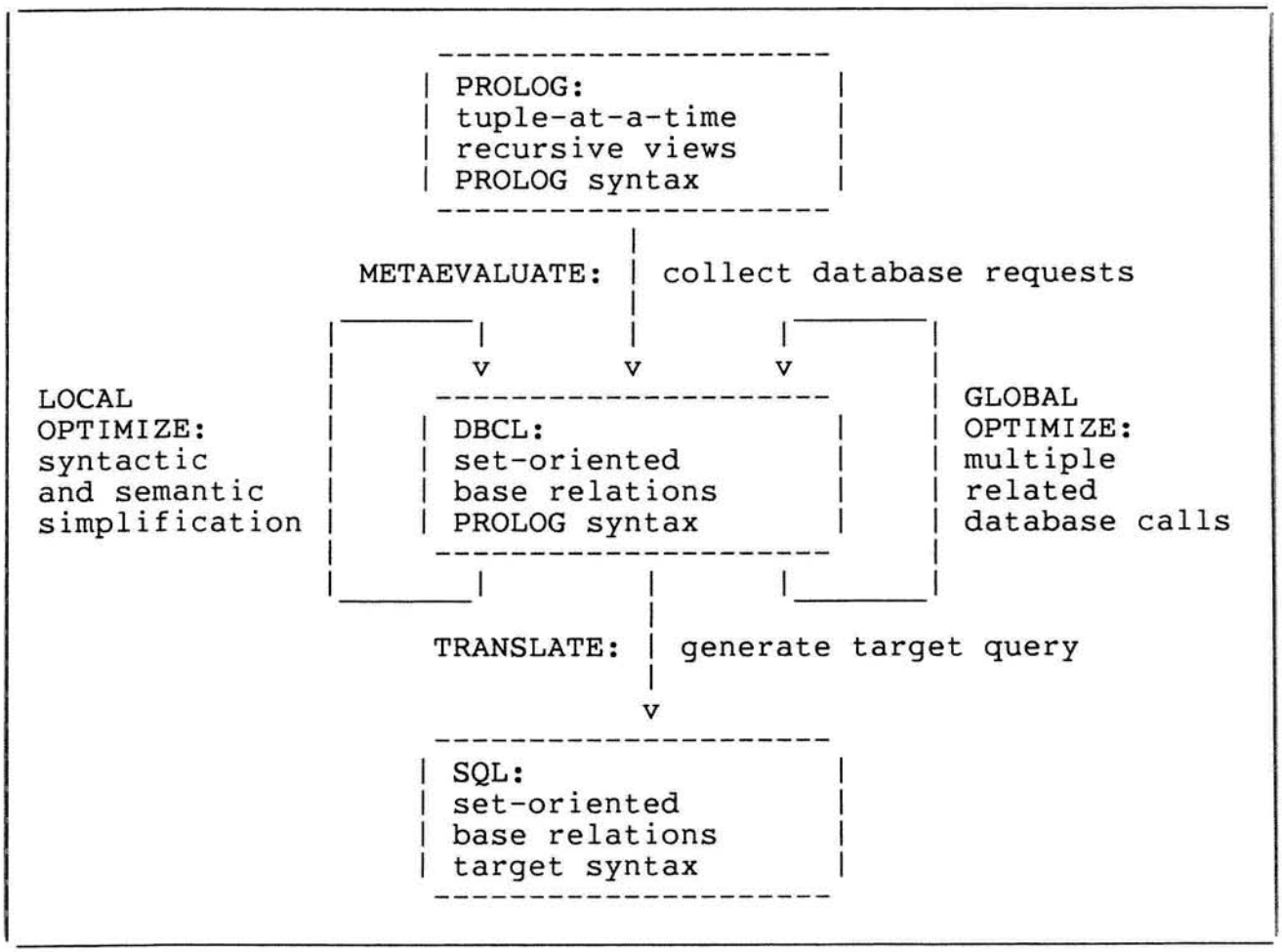

Figure 1: Architecture of PROLOG-SQL translation mechanism

The overall architecture of our approach is summarized in Figure 1. The central idea involves the introduction of an intermediate language of database calls (DBCL), which is set-oriented and uses base relations but is still expressed in PROLOG (to be precise: in a variable-free subset of PROLOG). The use of DBCL separates the two aforementioned tasks. Thus, phase (a) becomes independent of the target database query language and a large number of optimizations can be performed without reference to the target language. DBCL will be defined formally in section 3, followed by a description of the functions shown on the arcs of Figure 1:

1. Metaevaluate translates PROLOG statements into DBCL statements. It is described in [Vassiliou et al. 1983, 1984] for a slightly different intermediate language, and adapted for our current purposes in section 4.

2. Translate (see section 5) generates a set of queries in the target database language, SQL, from a DBCL predicate.

3. Local optimize (see section 6) removes redundancy from a DBCL predicate to eliminate the execution of unnecessary operations. Techniques used for this purpose are similar to view processing strategies as described, e.g., in [Ott and Horlaender 1982; Rosenthal and Reiner 1982].

4. Global optimize (see section 7) has two functions. First, it determines which parts of a DBCL expression can be evaluated using the internal PROLOG database, and for which parts external database queries have to be generated. Second, it decides whether query results should be stored for future reference, a feature of particular importance in processing recursive database calls.

We conclude this overview by briefly mentioning two crucial software components required to support the above functions, and some alternatives for their implementation.

An internal database system in the logic language can be used for storing query answers from the external database. Garbage collection may become an issue if some of these results are large and not reused. In addition, a merge procedure must be provided to combine internal and external database segments. An alternative strategy is provided by storing query results in the external database system, to keep a clean separation between database and logic program data. It is not clear which alternative is preferable in general. Our mechanism employs an internal DBMS because query results are expected to be fairly small.

An amalgamation procedure between logic programming language and metalanguage [Bowen and Kowalski 1982] is required since our approach requires a self-modifying program that uses a precompilation of itself. Alternatively, the view definitions could be translated at logic program design time [Reiter 1978]. However, if recursion appears, this "compiled" approach requires iteration constructs not readily available in PROLOG [Henschen and Naqvi 1984].

3.0 FORMAL DEFINITION OF DBCL

One of the main motivations for the introduction of DBCL is the collection, joint optimization, and therefore delayed execution of tuple-oriented PROLOG data requests. To prevent their immediate execution, PROLOG data requests must be manipulated in a metalanguage. Since PROLOG itself is used as this metalanguage (see section 4), the data requests must be converted into a variable-free representation to avoid instantiation of variables. This form is constructed from the original PROLOG predicate as follows. Constants are translated into themselves. Universally quantified variables of the original goal clause are preceded by a "t_" (these variables denote the target attributes of the query). Other variables are preceded by a "v_" and a number is appended to them to distinguish between different variables addressing the same attribute.

A BNF grammar of DBCL is provided in Figure 2. Note, that in general a DBCL statement may contain references to arbitrary PROLOG predicates as well as negation and disjunction (denoted by ";" in the grammar). In the remainder of this paper, we shall concentrate on a subset of DBCL that contains only metaterms without negation. The only predicate names allowed besides database relation names are the standard comparison operators. Essentially, this amounts to a reduction of the generated queries to conjunctive queries including inequality comparisons, but without embedded functions.

```
<metastatement> ::= <metaterm> |
                    <metaterm> ; <metastatement>
<metaterm>      ::= <metafactor> |
                    <metafactor> , <metaterm>
<metafactor>    ::= <relreference>  |
                    <predreference> |
                    not( <metastatement> )
<relreference>  ::= [<relname> , <parmlist>]
<predreference> ::= [<predname>, <parmlist>] |
                    <predname>(<metastatement>)
<parmlist>      ::= <parameter> |
                    <parameter> , <parmlist>
<parameter>     ::= v_<variable> |
                    t_<variable> |
                    <constant>
```

Figure 2: Grammar for full DBCL

A DBCL predicate for conjunctive queries takes the form

dbcl(Schema,Targetlist,Relreferences,Relcomparisons).

Relcomparisons contains predreferences, moved to the end of the predicate by goal reordering [Warren 1981]. The four DBCL components are explained below. They are designed in a way that makes DBCL similar to (tagged) tableaux [Ullman 1982].

Schema is a list of attributes of the underlying database schema together with the name of the database of interest.

Example 3-1:

Throughout this paper, we shall use a database, empdep, of two relations, describing employees (characterized by number, name, salary, and department number) and departments (characterized by number, function, and manager number).

empl (eno, nam, sal, dno)
dept (dno, fct, mgr)

The schema for this database is defined by the list

[empdep, eno, nam, sal, dno, fct, mgr].

When relations are used in tableau format, their definition follows the above schema, with a value of "*" specified for attributes that do not apply.

In addition to this syntactic specification of the schema, semantic integrity constraints can be specified. Since we assume the use of an existing database system, we cannot expect very sophisticated types of constraints to hold. In this paper, only three kinds of integrity constraints (we believe, the most frequent ones in practice) will be considered:

(1) value bounds for ordered attributes are expressed in PROLOG in the format

valuebound(R, A, L, U)

which means that L <= x <= U for all values x of attribute A in relation R.

(2) functional dependencies within relations are expressed in the format

funcdep(R, A1, A2)

which means that for all pairs x, y of elements of relation R:

x.A1 = y.A1 ---> x.A2 = y.A2,

where A1 and A2 are subsets of the attribute sets of R. An important subset of functional dependencies are key constraints.

(3) referential integrity constraints are denoted

refint(R1, A1, R2, K2)

which means that the set of values appearing in the attribute(s) A1 of relation R1 must be a subset of the set of key values appearing in R2.

Referential integrity constraints are a subset of the so-called inclusion dependencies [Fagin 1981]. They map the fact that each attribute, property, or relationship is based on the existence of (a unique combination of) underlying objects. This translates into the rules that (a) the right-hand side (or superset) of a referential integrity constraint always refers to the key of some relation (identifying exactly one of the underlying objects), and (b) no attribute may appear in more than one left-hand side of a referential integrity constraint.

Example 3-2:

In the example database, we assume the following integrity constraints to hold:

```
valuebound (empl, sal, 10000, 90000).
funcdep (empl, [nam], [eno]).
funcdep (empl, [eno], [nam, sal, dno]).
funcdep (dept, [dno], [fct, mgr])
funcdep (dept, [mgr], [dno]).
refint (empl, [dno], dept, [dno]).
refint (dept, [mgr], empl, [eno]).
```

It is easy to express inference rules over these dependencies in the same format, using variables instead of constants, e.g., funcdep(Rel,Attrset,Attrset) -- the reflexivity axiom. Such inference rules can be used for semantic query optimization, see section 6.

Targetlist has the same format as Schema and defines the schema of the result relation of the database call.

Relreferences is a list whose elements are lists, each corresponding to a row in a (tagged) tableau and having the same format as Schema, with "*" values for non-applicable attributes. In terms of SQL, each relreference corresponds to a relation variable. If symbols corresponding to variables (i.e., starting with t_ or v_) are duplicated in the Relreferences, each pair corresponds to an equijoin.

Relcomparisons is a list of lists, each corresponding to a relational comparison (e.g. less, greater). It may be empty if no such comparisons exist. Each sublist maps to either an inequality restriction or an inequality join.

We conclude this section by a comprehensive example illustrating the DBCL representation of general conjunctive database calls.

Example 3-3:

Let a PROLOG view called "works_dir_for" be defined as follows:

```
works_dir_for(X, Y) :-
        empl(_, X, _, D),
        dept(D, _, M),
        empl(M, Y, _, _).
```

The underscores represent distinct but irrelevant variables. Consider the query: "who works directly for Smiley for less than 40000?"

```
:- works_dir_for(X, smiley),
   empl(_, X, S, _), less(S, 40000).
```

This would translate to the tableau-like DBCL representation:

```
dbcl(
[empdep,   eno,    nam,    sal,    dno,    fct, mgr],

[works_dir_for,
             *,    t_X,      *,      *,      *,   *],

[[empl, v_Enol,    t_X, v_Sal1,    v_D,      *,   *],
 [dept,      *,      *,      *,    v_D, v_Fct2, v_M],
 [empl,    v_M, smiley, v_Sal3, v_Dno3,      *,   *],
 [empl, v_Eno4,    t_X,    v_S, v_Dno4,      *,   *]],

[[less, v_S, 40000]]).
```

4.0 TRANSLATION OF PROLOG INTO DBCL

PROLOG statements are translated into DBCL by the predicate metaevaluate, which is described in detail in [Vassiliou et al. 1983, 1984]. The function of metaevaluate is to delay the execution of database-related clauses in PROLOG, and to collect the related database calls for set-oriented processing. In order to perform this function, the database references are translated into DBCL using an amalgamation of PROLOG with a suitable meta-language as described in [Bowen and Kowalski 1982; Kunifuji and Yokota 1982]. If the original predicate involves recursion, a sequence of DBCL statements is generated.

Example 4-1:

Assume that the relations empl and dept are stored in an external database, whereas the internal PROLOG knowledge base contains the following facts and rules (the specific database-related predicates are not shown here, see [Vassiliou et al. 1984]):

```
specialist(jones, guns).
specialist(miller, driving).
specialist(smiley, thinking).
...

works_dir_for(X, Y) :- (... as in example 3.3 ...)

same_manager(X, Y) :- works_dir_for (X, M),
                      works_dir_for (Y, M),
                      neq(X,Y).

partner(W, X, Skill) :-
   metaevaluate(pr5,
      [same_manager(t_X, W)], no_optim, DBCL), !,
   same_manager(X, W), specialist(X, Skill).
```

That is, if employee W has to perform a specific task requiring a certain Skill, W can find a partner for that task by looking for employees X who have the same skill and work for the same manager. pr5 is a program name, no_optim indicates that query optimization is turned off, and DBCL is the variable that will hold the database query expressed in DBCL.
Assume that employee Jones looks for a partner who is a specialist in driving. The corresponding query

```
:- partner(jones, X, driving).
```

would be resolved partially using database data and partially within PROLOG. First, the evaluation of the metaevaluate predicate would result in the creation of instantiated same_manager predicates in the internal PROLOG database. Then, PROLOG would combine the same-manager data with the specialist information, using its normal tuple-at-a-time procedure. Note that the cut (!) after the metaevaluate predicate makes sure that it is evaluated only once (i.e., the queries resulting from its evaluation are submitted to the database only once).

For the purposes of this paper, the most important function of metaevaluate is the simulation of PROLOG's deduction procedure in order to translate the view

```
same_manager(t_X, jones)
```

into the DBCL predicate

```
dbcl(
[empdep,   eno,    nam,     sal,     dno,     fct,   mgr],

[same_manager,
              *,   t_X,       *,       *,       *,     *],

[[empl, v_Eno1,    t_X, v_Sal1,    v_D1,       *,     *],
 [dept,      *,     *,       *,    v_D1, v_Fct2, v_M1],
 [empl,   v_M1,    v_M, v_Sal3, v_Dno3,       *,     *],
 [empl, v_Eno4, jones, v_Sal4,    v_D2,       *,     *],
 [dept,      *,     *,       *,    v_D2, v_Fct5, v_M2],
 [empl,   v_M2,    v_M, v_Sal6, v_Dno6,       *,     *]],

[[neq, t_X, jones]]).
```

5.0 TRANSLATION OF DBCL INTO SQL

The second translation step of the proposed mechanism converts DBCL into the target language, here assumed to be SQL. Since only function-free conjunctive queries are considered, the generated queries do not require nesting [Kim 1982]. The algorithm just has to fill in the information from the DBCL tableau into the SELECT...FROM...WHERE... pattern according to the following rules:

1. Each row of the Relreferences section corresponds to a variable definition in the FROM clause.

2. Attributes with entries in the Targetlist appear in the SELECT clause, together with an appropriate variable name (number of the first row where the same entry appears).

3. Each constant in the Relreferences is translated into a restrictive condition (with an equality comparison operator) whose left-hand side is determined by the row (variable name) and the column (attribute name) of the appearance.

4. Each pair of equal symbols in the Relreferences starting with t_ or v_ is translated into an equijoin term. The components are again determined by their location as in the previous step.

5. Each row in Relcomparisons is mapped into a restrictive or join term. The names of the participating variables and attributes are determined by the location of the first occurrence of the same symbols in the Relreferences section.

6. Non-repeated variables do not appear in the SQL query.

Example 5-1:

Following the above rules, the query in example 4-1 is translated into:

```
SELECT v1.nam
FROM   empl v1, dept v2, empl v3,
       empl v4, dept v5, empl v6
WHERE  (v1.dno = v2.dno) AND (v2.mgr = v3.eno) AND
       (v4.dno = v5.dno) AND (v5.mgr = v6.eno) AND
       (v4.nam = 'jones') AND (v3.nam = v6.nam) AND
       (v1.nam ≠ 'jones')
```

More formally, the translation process can by described as a mapping from the DBCL syntax tree to an SQL syntax tree. An example is provided in the Appendix.

6.0 SYNTACTIC AND SEMANTIC QUERY SIMPLIFICATION

The DBCL and SQL examples presented so far are directly generated from the corresponding PROLOG predicates. Unfortunately, direct view translation tends to carry a large overhead of superfluous operations. Our mechanism does not rely on the database system but applies syntactic and semantic query simplification techniques within DBCL to remove such inefficiencies.

Syntactic methods attempt to reduce the number of joins in a query by removing redundant tuple variables, or by replacing joins with projections through constant propagation. The main task is the recognition and removal of common subexpressions. In a tableau representation, join minimization corresponds to the minimization of the number of rows [Aho et al. 1979]. Our algorithms for this syntactic step are based on proposals by Sagiv [1983] but extended to a multi-relation environment, in which variables may appear in more than one tableau column [Johnson and Klug 1983].

Often, syntactic simplification rules become applicable only after equivalence transformations based on semantic integrity constraints have been executed. Systems such as QUIST [King 1981] use semantic knowledge about single data objects or small groups of them for different kinds of efficiency-oriented query transformations; such heuristics usually require a large amount of detailed knowledge and sophisticated AI techniques to choose from applicable integrity constraints. In contrast, our algorithms only use general semantic integrity rules, applying to relations as a whole, for query simplification. In particular, the three types of integrity constraints introduced in section 3

(value bounds, functional dependencies, and referential constraints) are used as the knowledge base. We consider each type in turn.

6.1 Value Bounds

Value bounds can be added to Relcomparisons to check for contradictions or redundant comparisons. For example, if the value of 40000 were replaced by 200000 in the condition, less(S, 40000), of example 3-3, the inequality could be omitted since its satisfaction is already implied by the integrity constraint that all salaries must fall in the range between 10000 and 90000. On the other hand, a value of 2000 would yield the empty relation as a result to the whole query because of a contradiction with the same integrity constraint. Another opportunity for simplification may arise from certain combinations of inequality conditions. For example, in "A >= B and B >= C and A ≠ C", the last condition could be replaced by the sharper "A > C", and "A >= B and B >= C and C >= A" is equivalent to "A = B and B = C", which could be expressed more efficiently by renaming variables in Relreferences, discarding the inequalities. The PROLOG implementation of such inequality-based simplifications is based on a graph procedure described in [Rosenkrantz and Hunt 1980].

6.2 Functional Dependencies

One of the main reasons for designing DBCL in a tableau-like fashion is the availability of functional dependencies for tableau simplification, using variations of the chase process that have been widely studied since the original paper by Aho et al. [1979]. Since we consider functional dependencies only within relations, the Relreferences section of a DBCL predicate can be partitioned by relation names. The process then tries to equate rows within each partition and to remove duplicates, thereby simplifying the Relreferences. Care has to be taken for correct renaming since -- in contrast to normal tableaux -- we allow comparisons between different columns of a tableau (e.g., between mgr and eno). Our implementation employs a version of the fast chase algorithm proposed by Downey et al. [1980], adapted to the problem of query simplification rather than lossless join tests. In particular, our version does not only detect equivalence classes of tableau entries but actively removes duplicate rows.

Example 6-1:

Consider the three rows addressing empl elements in the Relreferences section of the DBCL predicate in example 3-3.

Applying the functional dependency,

```
funcdep(empl, [nam], [eno]),
```

we can replace all occurrences of v_Eno4 by v_Eno1 and consequently, since

```
funcdep(empl, [eno], [nam,sal,dno]),
```

the first and the last row can be equated, and one of them omitted leading to the simplified expression (note the renaming in the Relcomparisons section)

```
dbcl(
[empdep,   eno,    nam,    sal,    dno,    fct, mgr],

[works_dir_for,
             *,    t_X,      *,      *,      *,   *],

[[empl, v_Eno1,    t_X, v_Sal1,    v_D,      *,   *],
 [dept,      *,      *,      *,    v_D, v_Fct2, v_M],
 [empl,    v_M, smiley, v_Sal3, v_Dno3,      *,   *]],

[[less, v_Sal1, 40000]]).
```

6.3 Referential Integrity

The application of referential integrity constraints allows the deletion of certain 'dangling' rows from the Relreferences section and thus of unnecessary variables and join terms. A row r with tag R dangles if the set of |R| attributes can be partitioned into two sets containing variables RPi, i=1, ..., m, and RNj, j=m+1, ..., |R1|, such that (a) for all j, r[RNj] is a value starting with v_ that appears nowhere else in the DBCL predicate, and (b) there is a row r' with tag R' and attributes RP'i, i=1, ..., m, such that r[RPi] = r'[RP'i] for i=1, ..., m. A dangling row r is deletable if there is an referential constraint, refint(R', [RP'1, ..., RP'm], R, [RP1, ..., RPm]), for some row r' that satisfies condition (b).

Note, that -- due to condition (a) -- the deletion of a dangling row can cause other rows to become deletable. Therefore, row deletion due to referential integrity constraints is a recursive process.

We are aware of one proposed view optimizer that uses inclusion dependencies in this way [Rosenthal and Reiner 1982]. However, only directly applicable dependencies are considered. This is not surprising since the test whether a general inclusion dependency can be derived from a given set is known to be computationally

difficult [Casanova et al. 1982]. The restriction to 'key-based' dependencies proposed in [Johnson and Klug 1982] (i.e., that the right-hand side of each inclusion dependency must be a subkey and the left-hand side must not contain key attributes) amounts to the same fact, namely that inclusion dependencies are applicable either directly or not at all. The referential integrity constraints used in this paper do allow indirectly derived referential dependencies but there is a relatively simple and efficient method of drawing inferences since each attribute may appear only on the left-hand side of (at most) one referential integrity constraint. The algorithm is sketched below; it assumes an arbitrary but fixed numbering of attributes in the database schema.

Algorithm 1
(Inference Procedure for Referential Integrity):

Input: A pair ([Ra,[A1,...,Am]], [Rb,[B1,...,Bm]])
representing a hypothesized referential integrity constraint, and a set of given referential integrity rules, originally marked "unused".

Output: Success (the constraint is derivable from the stored referential constraints) or failure (it is not).

Procedure:

1. Initialize a variable CURRENT with the hypothesized referential constraint.

2. Sort the two attribute lists in CURRENT by ascending attribute numbers on the left-hand side.

3. By the inference axioms given in [Casanova et al. 1982], a referential constraint RC is applicable if the left-hand side of CURRENT is a subsequence of the left-hand side of RC. If no "unused" referential constraint is applicable, stop with failure: the hypothesized rule is not derivable from the existing referential integrity constraints.

4. Replace the left-hand side of CURRENT by the appropriate subset of the right-hand side of the applicable referential constraint RC. If now the right-hand side and the left-hand side of CURRENT match, stop with success: there is a (derived) referential integrity constraint between attributes A1, ..., Am of Ra and attributes B1, ..., Bm of Rb. Otherwise, mark RC "used" and return to step 2.

The correctness proof for this procedure is left to a forthcoming paper. Note, that by definition of the

referential constraints (see section 3), at most one rule will apply in each step, and because of the rule marking in step 3, no rule will be used more than once. Therefore, the algorithm not only terminates but can also be implemented quite efficiently. Its exact complexity depends on the matching procedure used for detecting applicable rules.

6.4 Summary Of The Simplification Algorithm

Our prototype simplification algorithm does not yet utilize semantic and syntactic query simplification methods in a fully integrated manner. For example, it does not take into account the interaction of functional dependencies with referential integrity constraints but applies them sequentially. Moreover, checking value bounds and functional dependencies could be integrated more efficiently. Nevertheless, the procedure sketched below covers a large class of possible improvements.

Algorithm 2 (DBCL Simplification Procedure):

1. Add value bounds to Relcomparisons for attribute variables appearing there and check whether all constants appearing in Relreferences are within their domains. If not, stop with an empty query result.

2. Set the Boolean variables REPEAT and FIRSTTIME to true.

3. Apply the inequality simplification algorithm (section 6.1); if a contradiction is detected, stop with an empty query result; if variables have to be renamed due to newly detected equality conditions or if FIRSTTIME, set REPEAT to true and FIRSTTIME to false, else set REPEAT to false.

4. If REPEAT then do the following:
   apply a functional dependency chase algorithm with deletion of duplicate rows (section 6.2); if a contradiction is detected, stop with an empty query result; if variables have been renamed return to 3.

5. Remove deletable dangling tuples from Relreferences recursively (section 6.3).

6. Minimize the remaining tableau by a syntactic algorithm (section 6.0).

Example 6-2:

Consider the query of examples 4-1 and 5-1. There are no applicable valuebounds in this case. However, knowledge about the functional dependencies allows the deletion of two

rows in the Relreferences section. The first functional dependency given in section 3 said that no two employees with the same name have different employee numbers. But eno is a key of relation empl by the second functional dependency. Therefore, the third and the last row of the predicate describe the same set of employees. Renaming row 6 permits the deletion of that row by the trivial syntactic simplification rule, A AND A <==> A. During this process, the mgr attribute in row 5 has been renamed to v_M1. Using the same reasoning as before, we can equate rows 2 and 5, rename again, and remove row 5. This ends step 4 of the algorithm; since still no valuebound applies, the tableau immediately before step 5 looks as follows:

```
dbcl(
[empdep,   eno,   nam,    sal,    dno,    fct,  mgr],

[same_manager,
             *,   t_X,      *,      *,      *,    *],

[[empl, v_Eno1,   t_X, v_Sal1,   v_D1,      *,    *],
 [dept,      *,    *,      *,   v_D1, v_Fct2, v_M1],
 [empl,   v_M1,   v_M, v_Sal3, v_Dno3,      *,    *],
 [empl, v_Eno4, jones, v_Sal4,   v_D1,      *,    *]],

[[neq, t_X, jones]]).
```

In this DBCL predicate, the third row dangles; it is also deletable since v_M1 appears in the second row and there is a (directly applicable) referential integrity constraint between mgr in dept and eno in empl. After the deletion of the third row, the second row dangles and is also deletable. Thus, the final DBCL predicate looks as follows:

```
dbcl(
[empdep,   eno,   nam,    sal,  dno, fct, mgr],

[same_manager,
             *,   t_X,      *,    *,   *,   *],

[[empl, v_Eno1,   t_X, v_Sal1, v_D1,   *,   *],
 [empl, v_Eno4, jones, v_Sal4, v_D1,   *,   *]],

[[neq, t_X, jones]]).
```

Informally speaking, the stored semantic knowledge allowed us to simplify the question: "who works (directly) for the same manager as jones?" to: "who works in the same department as Jones?" The above DBCL predicate translates into the SQL query:

```
SELECT v1.nam
FROM   empl v1, empl v2
WHERE  (v1.dno = v2.dno) AND (v2.nam = 'jones')
                         AND (v1.nam ≠ 'jones')
```

Comparing this with example 5-1, we note that four out of five join operations have been avoided by the application of semantic constraints.

7.0 EXTENSIONS

The procedures presented thus far deal only with function-free conjunctive queries. The extensions surveyed in this section have to handle disjunction, negation, general PROLOG predicates appearing in database requests, and recursive database calls. Details will be left to a forthcoming paper.

The simplest way to handle disjunction is converting the DBCL predicate into disjunctive normal form, and generating a query for each of these conjunctions. This is done in some existing DBMS (for instance, CCA's SDD-1 [Bernstein et al. 1981]) but may not be the most efficient solution [Grant and Minker 1981; Sagiv and Yannakakis 1980].

A problem with negation is that it is difficult to determine the meaning of it as soon as Relreferences extend over more than one relation. For example, consider a view definition

```
manager(X,Y) :- empl (X,_,_,D), dept (D,_, Y).
```

Should the query, :- not(manager(jones, M)) return all numbers of employees who are managers but do not manage Jones (to be retrieved from the dept relation), or should it also return the employees who are not managers at all (to be retrieved from the empl relation)? Note that the latter interpretation would utilize a referential integrity constraint. If it can be decided which query is meant, its evaluation involves first computing the positive result, and then its complement in the appropriate set. Instead of set difference, SQL's nested expressions (NOT IN (...)) can also be used.

If not all database references are lumped together in a view definition, there may be embedded predicates and PROLOG 'cuts' mixed with them to express certain relationships between the retrieved data. Some standard predicates can be handled within SQL, for instance inequality comparisons or built-in functions. If other predicates occur within the DBCL predicate several queries have to issued, and the

interaction between their results must be evaluated in PROLOG. Not only will this happen tuple-at-a-time but the partial query results may not even fit in main memory. Therefore, we are investigating a more structured approach based on the extensions of relational calculus proposed in [Klug 1982]. A first step in this direction that avoids the aforementioned space problems, at the expense of more computing time, is a step-wise evaluation process that evaluates the partial queries from right to left, using what amounts to a version of tuple substitution [Wong and Youssefi 1976].

Often, it is advantageous to process multiple database queries simultaneously by recognizing common subexpressions [Jarke 1984]. In particular, the problem of handling recursion in deductive databases has attracted considerable attention in the literature [Gallaire et al. 1981; Henschen and Naqvi 1984; Minker and Nicolas 1983]. Where these papers are concerned with optimization issues at all, they focus on the idea of preserving intermediate results for the next steps. The following example is meant to demonstrate this approach but also its limitations and the need for additional efficiency-oriented research.

Example 7-1:

Consider a recursive view definition that describes that someone (called Low) is working for someone else (called High) at any level.

```
works_for(Low, High) :-
   works_dir_for(Low, High).
works_for(Low, High) :-
   works_dir_for(Low, Medium),
   works_for(Medium, High).
```

Assume a query that asks for "Smiley's people":

```
:- works_for(People, smiley).
```

Naive processing of this would generate a sequence of increasingly complex queries:

```
1) works_dir_for(People, smiley).
2) works_dir_for(People, X1),
      works_dir_for(X1, smiley).
3) works_dir_for(People, X1),
      works_dir_for(X1, X2),
      works_dir_for(X2, smiley).
etc.
```

Each recursive step adds one condition to the query. For readability, the view representation of works_dir_for was used; in reality, the queries would address three database

relations for each view, making the duplication of effort even more obvious. Therefore, it would be useful to store the result of each step in an intermediate relation to be used in the following step, instead of re-executing the previous query as part of the new one. In essence, this can be achieved by augmenting the definition of works_for as sketched below.

```
works_for(Low, Boss) :-
   setrel(intermediate(Boss)),
   works_for_boss(Low, Boss, Boss).
works_for_boss(Low, Currenthigh, Boss) :-
   intermediate(Currenthigh),
   works_directly_for(Low, Currenthigh).
works_for_boss(Low, Currenthigh, Boss) :-
   intermediate(Currenthigh),
   works_directly_for(Medium, Currenthigh),
   setrel(intermediate(Medium)),
   works_for_boss(Low, Medium, Boss).
```

The predicate, setrel, creates a unary intermediate relation. Using this method,

```
:- works_for(People, smiley).
```

would first set the intermediate relation to containing just "Smiley", then all people who work for him directly, etc. Each generated SQL query would take the same form:

```
SELECT v3.ename
FROM   empl v1, dept v2, empl v3,
       intermediate v4
WHERE  (v1.dno=v2.dno) AND (v2.mgr=v3.eno) AND
       (v3.nam=v4.nam)
```

The final result would be the union of all these query results.

On first sight, this seems to be a nice solution to handling recursion efficiently. Unfortunately, just asking another query to the same view completely upsets our scheme. Consider the query, :- works_for(jones, Superior), requesting the names of Jones' managers at any level. Here, the proposed solution would still work. However, it would generate as the first intermediate relation all employee names, then all names of immediate employees of any manager (i.e., everybody except the top manager), and so forth until the hierarchy is exhausted. Although the final solution is smaller than in the first query, the intermediate results are much (and unnecessarily!) larger. A better solution would have to generate a more efficient original view definition that generates solutions bottom-up rather than top-down, namely:

```
works_for(Low, High) :-
   works_directly_for(Low, High).
works_for(Low, High) :-
   works_directly_for(Medium, High),
   works_for(Low, Medium).
```

It is open, how this kind of optimization can be detected by a query optimizer. (After the revision of this paper, we became aware of a new approach by Marque-Pucheu et al. [1984] which provides a partial solution. However, the question how to integrate the optimization of recursive queries with the type of optimization proposed in section 6 remains to be investigated.)

8.0 CONCLUDING REMARKS

A mechanism for coupling expert systems and database systems was presented. Our approach differs from integrated expert systems databases [Warren 1981], as well as from so-called deductive databases such as BDGEN [Nicolas and Yazdanian 1983], in that it provides a connection mechanism attached to the expert systems language, yet is independent of any particular application. We believe that the proposed method can contribute to a practical integration of expert systems into real-life business environments, and to more intelligent and powerful operation of existing relational database systems [Jarke and Vassiliou 1984].

The PROLOG-DBCL and DBCL-SQL translations, as well as initial versions of the query simplification procedures, have been implemented. We are still working on more efficient support for recursive queries, and on extensions to the local optimizer, covering disjunction, negation, and general embedded PROLOG functions.

REFERENCES

1. Aho, A.V., Sagiv, Y., Ullman, J.D., "Efficient optimization of a class of relational expressions", ACM-TODS 4, 4 (1979), 435-454.

2. Bernstein, P.A., Goodman, N., Wong, E., Reeve, C.L., Rothnie, J.R., "Query processing in a system for distributed databases (SDD-1)", ACM-TODS 6, 4 (1981), 602-625.

3. Bowen, K.A., Kowalski, R.A., "Amalgamating language and metalanguage in logic programming", in K.Clark and S.A.Tarnlund (eds.), Logic Programming, Academic Press, 1982.

4. Brodie, M., Mylopoulos, J., Schmidt, J.W. (eds.), On Conceptual Modelling, Springer 1984.

5. Casanova, M.A., Fagin, R., Papadimitriou, C.H., "Inclusion dependencies and their interaction with functional dependencies", Proc. First Symposium on Principles of Databases, Los Angeles 1982, 171-176.

6. Downey, P.J., Sethi, R., Tarjan, R.E., "Variations on the common subexpression problem", Journal of the ACM 27,4 (1980), 758-771.

7. Fagin, R., "A normal form for relational databases that is based on domains and keys", ACM Transactions on Database Systems 6, 3 (1981), 387-415.

8. Gallaire, H., Minker, J. (eds.), Logic and Databases, Plenum 1978.

9. Gallaire, H., Minker, J., Nicolas, J.M. (eds.), Advances in Database Theory, Vol.1, Plenum Press, 1981.

10. Grant, J., Minker, J., "Optimization in deductive and conventional relational database systems", in H.Gallaire, J.Minker, J.M.Nicolas (eds.), Advances in Database Theory, Plenum Press, New York, 1981, 195-234.

11. Henschen, L.R., Naqvi, S., "On compiling queries in recursive first-order databases", Journal of the ACM 31, 1 (1984), 47-85.

12. Jarke, M., "Common subexpression isolation in multiple query optimization", in Kim, W., Reiner, D., Batory, D., (eds.), Query Processing in Database Systems, Springer, to appear 1984.

13. Jarke, M., Vassiliou, Y., "Coupling expert systems with database management systems", in Reitman, W. (ed.), Artificial Intelligence Applications for Business, Ablex, Norwood, NJ, 1984, 65-85.

14. Johnson, D.S., Klug, A., "Testing containment of conjunctive queries under functional and inclusion dependencies", Proc. ACM Symposium on Principles of Database Systems, Los Angeles 1982, 164-169.

15. Kim, W. "On optimizing an SQL-like nested query", ACM-TODS 7, 3 (1982), 443-469.

16. King, J.J., "QUIST: A system for semantic query optimization in relational data bases", Proc. 7th VLDB Conf., Cannes 1981, 510-517.

17. Klug, A., "Access paths in the 'Abe' statistical query facility", Proc. ACM-SIGMOD Conf., Orlando 1982, 161-173.

18. Kunifuji, S., Yokota, H., "Prolog and relational databases for Fifth Generation Computer Systems", Proc. Workshop on Logical Bases for Data Bases, Toulouse, December 1982.

19. Marque-Pucheu, G., Martin-Gallausiaux, J., and Jomier, G., "Interfacing Prolog and relational data base management systems", in Gardarin, G., Gelenbe, E. (eds.), New Applications of Data Bases, Acadamic Press, to appear 1984.

20. Nicolas, J.M., Gallaire, H., and Minker, J. (eds.), Proc. Workshop on Logical Bases for Databases, Toulouse, December 1982.

21. Nicholas, J.-M., Yazdanian, K., "An outline of BDGEN: A deductive DBMS", in R.E.Mason (ed.), Information Processing 83, North-Holland 1983, 711-717.

22. Ott, N., Horlaender, K., "Removing redundant join operations in queries involving views", IBM Scientific Center Heidelberg Technical Report TR-82.02.003 (1982).

23. Reiter, R., "Deductive question-answering on relational databases", in Gallaire, H., Minker, J., Logic and Databases, Plenum 1978, 149-177.

24. Rosenkrantz, D.J., Hunt, M.B. "Processing conjunctive predicates and queries", Proc. 6th VLDB, Montreal 1980, 64-74.

25. Rosenthal, A., Reiner, D., "Querying relational views of networks", Proceedings IEEE COMPSAC, 1982.

26. Sagiv, Y., "Quadratic algorithms for minimizing joins in restricted relational expressions", SIAM Journal of Computing 12, 2 (1983), 316-328.

27. Sagiv, Y., Yannakakis, M., "Equivalences among relational expressions with the union and difference operators", JACM 27 (1980), 633-655.

28. Schmidt, J.W. "Some high-level language constructs for data of type relation", ACM-TODS 2, 3 (1977), 247-261.

29. Stonebraker, M., "The design and implementation of Ingres", ACM-TODS 1, 2 (1976).

30. Ullman, J.D., Principles of Database Systems, Computer Science Press 1982.

31. Vassiliou, Y., Clifford, J., Jarke, M., "How does an expert system get its data?", Proc. 9th VLDB Conf., Florence, October 1983, 70-72.

32. Vassiliou, Y., Clifford, J., Jarke, M., "Access to specific declarative knowledge by expert systems: the impact of logic programming", Decision Support Systems 1, 1 (1984).

33. Warren, D.H.D., "Efficient processing of interactive relational data base queries expressed in logic", Proc. 7th VLDB Conf., Cannes 1981, 272-282.

34. Wong, E., Youssefi, K., "Decomposition- a strategy for query processing", ACM TODS 1 (1976), 223-241.

APPENDIX

The execution of our prototype PROLOG programs is demonstrated by the example of the works_dir_for predicate introduced in section 3. The output shown is actual PROLOG output, run using DEC-20 Prolog under TOPS-20, but has been edited to fit the Proceedings format.

Consider the query: "who works directly for Smiley?" This would be expressed in Prolog as:

```
:- works_dir_for(Nam,smiley).
```

The metaevaluation of this query yields:

```
| ?- metaevaluate(pr5,
     [works_dir_for(t_nam,smiley)], no_optim, NEW).

NEW = [dbcall(empl,v_eno,t_nam,v_sall,v_dno),
       dbcall(dept,v_dno,v_fct,v_enol),
       dbcall(empl,v_enol,smiley,v_sal2,v_dno2)]
```

This metaevaluated query is further transformed into the tableau-like DBCL format:

```
dbcl(
[empdep,   eno,    nam,    sal,    dno,   fct,    mgr],

[works_dir_for,
            *,  t_nam,      *,      *,     *,      *],

[[empl,  v_eno,  t_nam, v_sall,  v_dno,     *,      *],
 [dept,      *,      *,      *,  v_dno, v_fct, v_enol],
 [empl, v_enol, smiley, v_sal2, v_dno2,     *,      *]],

[]).
```

Finally, it is translated into an equivalent SQL formulation of the query (sqltrans), and the SQL version is displayed (sqlprint):

```
| ?- dbcl(DBCL),sqltrans(DBCL,SYNTAXTREE),
     sqlprint(SYNTAXTREE).

SELECT   v12.nam
FROM     empl v12, dept v13, empl v14
WHERE    (v12.dno=v13.dno) AND (v14.nam='smiley')
         AND (v13.eno1=v14.eno1)

DBCL =
[[empdep,   eno,    nam,   sal,     dno,   fct,    mgr],

 [works_dir_for,
             *,  t_nam,      *,      *,     *,      *],

 [[empl,  v_eno,  t_nam, v_sal1,  v_dno,     *,      *],
  [dept,      *,      *,      *,  v_dno, v_fct, v_eno1],
  [empl, v_eno1, smiley, v_sal2, v_dno2,     *,      *]],

 []].

  SYNTAXTREE =
  select([v12.t_nam],
    from([(empl,v12),(dept,v13),(empl,v14)]),
    where([equal(dot(v12,v_dno),dot(v13,v_dno)),
           equal(dot(v14,nam),smiley),
           equal(dot(v13,v_eno1),dot(v14,v_eno1))]))
```