INTELLIGENT QUERY ANSWERING IN RULE BASED SYSTEMS

TOMASZ IMIELINSKI*

▷ We propose that in large knowledge bases which are collections of atomic facts and general rules (Horn clauses), the rules should be allowed to occur in the answer for a query. We introduce a new concept of the answer for a query which includes both atomic facts and general rules. We provide a method of transforming rules by relational algebra expressions built from projection, join, and selection and demonstrate how the answers consisting of both facts and general rules can be generated. ◁

1. INTRODUCTION

In a large knowledge base system, the data are represented in the form of both general laws (given as Horn clauses) and assertions representing specific facts (e.g., tuples of relations). We are going to argue here that it is frequently beneficial to structure the answer for a query in a similar way, i.e. both in terms of tuples, as is traditionally done, and in terms of general rules. This is simply a consequence of the general philosophy of logic programming, which imposes a uniform view on programs (in this case queries and rules) and data [6]. This is beneficial both from the conceptual and the computational point of view.

Conceptually, rules are often more informative and easier to comprehend than the corresponding sets of derived tuples. The fact that, for example, in the university environment all graduate students in computer science who specialize in a given area have to take all the courses offered in this area can be better represented by a rule than by the corresponding derived set of tuples, regardless of whether this information is part of the database or part of the answer for a query. Computationally, we can benefit even more. It is much less expensive to evaluate the rules over the result of the query than over the database state itself, since the result of the

*Research supported by NSF grant DCR 85-04140.

Address correspondence to Dr. T. Imielinski, Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903.

Received May 1986; accepted October 1986.

THE JOURNAL OF LOGIC PROGRAMMING

52 Vanderbilt Ave., New York, NY 10017 0743-1066/87/$3.50

query is much smaller than the database. Besides, rule transformation extends the algebraic spirit of query processing from purely relational databases to databases with rules and is also another example of the "lazy evaluation" known in the area of programming languages. If rules in the answer of a query are evaluated yielding a set of tuples as the final answer for the query, the rules become *derived integrity constraints* in the view of the database [3].

Finally, this new concept of the answer for a query can be helpful in *resource limited computation*, when we do not have enough resources to compute the full answer to a query. Indeed, we may interpret an answer with rules as a type of "abstracted" answer to a query. Such an answer can be further specified, depending on the user's request. For example, if in the answer we include the rule saying "all prerequisities of courses", a user may want to know what these prerequisities are (see the next example), and in such a case our rule has to be evaluated. The query formulation process is in this case not only the process of choosing a proper formula to express a query, but also choosing a *language* in which the answer for a query is acceptable to the user. Such a language could for example include predicates which can occur in the rules of the answer to a query. Needless to say, computing rules in answers is an option for the user and is applied only if he explicitly requests it. We would also prefer that in case such a request is made, the necessary additional computation should be cheaper than starting everything from scratch.

Formally a database consists of a *database extension*, represented in the form of relations, and a *database intension*, which includes the rules R. The rules mentioned above are really *inference rules*, that is, rules which will derive missing, implicitly represented information from the information explicitly stored as tuples of the database extension. Therefore, if we want to query such a database, we cannot ignore the database rules, but have to take them into consideration in the process of query evaluation. Formally the answer for the query in such circumstances could be computed by closing the database under the underlying set of rules and then evaluating the query. We will say that rules from the database can be transformed by the query if some of the rules can be evaluated *after* the evaluation of the query without affecting the final result:

Example 1. Let us assume that the set of all professors of a university is partitioned into research groups. Let the binary predicate Group(x, y) be true for x and y iff x and y belong to the same research group. In this case it is natural to assume (or require) that professors from the same research group be able to teach the same courses. This can be formally represented as the following formula:

$$r_1 = \text{Teach}(x, y) \wedge \text{Group}(x, z) \rightarrow \text{Teach}(z, y).$$

Besides, we may have rules expressing standard equivalence properties of the relation Group (reflexivity, symmetry, transitivity). We may also require that the professors should be able to teach prerequisites of all the courses they are able to teach, which can be expressed by the following formula:

$$r_2 = \text{Teach}(x, y) \wedge \text{Prerequisite}(x, y) \rightarrow \text{Teach}(x, z),$$

where additionally the predicate Prerequisite satisfies the transitive closure rule.

Let us consider now the query: "Give me all the courses that Imielinski may teach". This query can be expressed as the following relational algebra expression:

$$\pi_{\text{course}}\sigma_{\text{name}=\text{Imielinski}}(\text{Teach}).$$

It is easy to see that the following rule, related to r_2, can be included in the answer for the query:

$$Q(x) \wedge \text{Prerequisite}(x, y) \rightarrow Q(y),$$

where Q denotes the query predicate. Let us say that this rule is *transformed* from r_2 modulo the query Q. This rule has also a very natural interpretation—together with the set of courses which will be returned, it simply says "include prerequisites of the courses which were just printed out". If the user wants to know what these prerequisites are, he may request the evaluation of this rule. Notice that even in this case we will benefit computationally, since the new transformed rule will be evaluated over the result of the query, which is much smaller than the database.

It is also clear that the rule r_1 is not transformable modulo this query and would have to be evaluated prior to the query itself.

The situation changes when we consider a different query:

"Who can teach the Database course".

It is easy to see that the situation is totally reversed: now the rule r_2 has to be evaluated prior to the query evaluation, while the rule r_1 can be transformed into the rule:

$$Q(x) \wedge \text{Group}(x, y) \rightarrow Q(y).$$

In other words, the answer for the query can be described in natural language as "the groups of Imielinski, Smith, etc. can teach databases".

Again, if the user wants to know who are the members of a particular group, he should request the evaluation of this rule.

Formally, the rules in the answer for a query should be either expressed totally in terms of the *query predicate* i.e., the predicate defined by the query or by the query predicate and some other additional predicates which occur in the database. The choice of these additional predicates may be left to the user. The query predicate must appear both in the consequent and in the body of the rule (otherwise the query definition could trivially form such a rule). We will always denote the query predicate by Q.

In this paper we analyze conditions under which rule transformation is possible. First we describe conditions under which single rules can be transformed by single relational operations; then we generalize our discussion to the relational expressions and finally to the sets of rules. The transformation of the sets of rules is particularly difficult—we always attempt to decompose the problem of transformation of sets of rules into the transformation of individual rules. We also demonstrate that even when the particular set of rules cannot be transformed, we can frequently construct an equivalent set of rules which can be transformed.

The paper is organized as follows. In the second section, basic notions are introduced. General conditions for transformation of single rules are described in the third section. The sets of rules are discussed in the fourth section. In the last

section we talk briefly about the further generalization of the answer for a query to include predicates and function symbols.

2. BASIC NOTIONS

We assume that the reader is familiar with the relational model of data [7] and standard terminology of logic programming [6, 2]. $P(t)$ will denote an atomic formula where P is the predicate name and t is a tuple of variables and constants. *Rules* will have the form of function free Horn clauses:

$$P_1(t_1) \wedge P_2(t_2) \wedge \cdots \wedge P_n(t_n) \rightarrow P_{n+1}(t_{n+1}).$$

A rule with k literals in its body will be called a *k-rule*. We will say that a rule r *defines* a predicate p iff p occurs as a consequent of r. By *assertions* we mean atomic formulas of the form $P(t)$ where t is a constant tuple. With a set of assertions we will usually identify the relations built from the sets of tuples t such that $P(t)$ is an atomic assertion. These relations will have columns identified either by *attributes* [7] or by the letters referring to attribute names. For example, in the predicate $P(x_1, \ldots, x_n)$ first position will be referred to by the number 1 or by A, the second by the number 2 or by B etc. With sets of predicates we will associate *multirelations*, i.e., sets of relations each corresponding to an individual predicate. Further, we will identify sets of assertions with the corresponding multirelations.

Let our rule be $P(t_1) \wedge \cdots \wedge P(t_n) \wedge W \rightarrow P(t_{n+1})$, where W is a conjunction of literals with predicate names different from P.

A variable will be called *bound* iff it has more than one occurrence in the body (the left-hand side) of the rule. The rule is *active* iff it serves as an inference rule; it is *passive* if it is satisfied by the set of assertions. In the latter case, the rule does not yield new answers and becomes simply an *integrity constraint*. We will assume further in the text that our database intension is built from inference rules R and integrity constraints IC. The division of the rules into passive and active will usually be done at the compile time of the database and will hold for all possible database extensions.

Further, we will always assume that our sets of rules are free from mutual recursion. Since mutual recursion is disallowed here, the set of all predicates could be partially ordered in such a way that p precedes q if p occurs in the body of some rule r defining q (i.e., with q the consequent of r). We will say that q *uses* p and define the "use" relation formally as a transitive closure of the above partial order. In a similar way we will say that the rule r uses rule s if s defines a predicate which is used by some predicate occurring in the body of the rule r.

We will classify the rules we are going to consider into certain categories. First of all, the rules that we will consider are function free. We will consider *unirelational* rules, in which only one predicate occurs (possibly in the different literals), and *multirelational* rules, in which several predicates may occur. For example, all template dependences are unirelational. The rules in the initial example are multirelational, since they consist of occurrences of more than one predicate.

The database state will consist of two parts: the database *extension* and the database *intension*. The database extension will be simply a set of assertions S stored in the form of a relational database. The database intension will consist of

Horn rules without functions. The rules of database intension will generate additional, derived assertions from the database extension. Given a set of rules R, by $R^*(S)$ we denote the smallest multirelation containing S and closed under the set of rules R.[1]

We will say that the set of rules is *bounded* iff there exists k such that for any database state S (i.e. the set of database assertions), $R^*(S) = R^k(S)$. We will say that a set of rules R is p-bounded, where p is a predicate, iff there exists k such that for any $S = \langle S_1, \ldots, S_n \rangle$ we have $R^*(S)[p] = R^k(S)[p]$, where $S[p]$ denotes the relation corresponding to predicate p in the multirelation S. Such a predicate will be called a *bounded predicate*, and the smallest k with the above property is called the *limit* of the set of rules. Obviously a set of rules may in general be unbounded and at the same time p-bounded for some particular predicate p.

A relation S_i is *closed* with respect to a set of rules R iff the smallest fixpoint of R containing $S = \langle S_1, \ldots, S_i, \ldots, S_n \rangle$ has the same ith component as S (the fixpoint is a multirelation as well as a database extension). The rule r is *closed* in a database state S iff $r(S) = S$.

All the predicates other than the consequent predicate which occur in the rule are called *foreign* predicates.

We will say that the rule r is *strongly idempotent* iff $r(r(S)) = r(S)$, and *weakly idempotent* iff $r(r(S)) = r(S)$ for any multirelation S such that all foreign predicates in r are closed. In other words, if the rule r is evaluated after all the rules which are used by it are closed, then it does not have to be evaluated more then once to reach a fixpoint.

We are going to introduce now two subclasses of rules we are going to deal with further in the paper.

2.1 Strongly Linear Rules

We will say that a rule is *typed* with respect to a certain predicate if each variable can occur only in a fixed position in any occurrence of this predicate. A predicate which occurs in the consequent of a rule will be called a *consequent* predicate.

A rule will be called *strongly linear* if a consequent predicate occurs exactly once in the body of the rule and a rule is typed with respect to its consequent predicate. The body of a linear rule can be viewed as a conjunction of the literal generated by the consequent predicate and some formula which is the conjunction of literals generated by foreign predicates. The latter formula will be called the *writing formula* of the strongly linear rule. By an R-occurrence of a consequent predicate in the rule r we mean the consequent literal. By an L-occurrence we mean its occurrence (exactly one, since the rule is linear) in the body of r.

By $\mathrm{Arg}(r)$ we denote the set of positions of the L-occurrence of the consequent predicate which share variables with the writing formula. By $\mathrm{Res}(r)$ we denote the set of positions of the R-occurrence of the consequent predicate which share variables with the writing formula. Intuitively, $\mathrm{Res}(r)$ is the set of positions of the consequent predicate which are changed (written) by the rule r, while $\mathrm{Arg}(r)$ is the set of positions in the consequent predicate on which the changed positions depend.

[1] In other words, the smallest fixpoint of the set R of rules containing S.

2.2 CONST *Rules*

Let CONST be a family of unirelational rules which are typed with respect to their consequent predicates and such that the set of variables which occur in the body of the rule is a subset of the set of variables which occur in the consequent of the rule. We have named these rules in this way because the constants occurring there play an important role. In face, these rules can be viewed as describing properties and relationships between different constants in the database (see Example 11 in the appendix). We believe that although in logic programming the rules of this type are not very interesting, in the context of databases or expert systems they have important applications.

Any finite set of rules from CONST has, of course, finite breadth first resolution trees (BFT) which result from resolving the rules of the database against negations of each individual predicate.[2]

Let us now describe more formally the notion of a query. Traditionally, queries in databases are expressed using the two different formalisms: relational algebra and the relational calculus [7]. The relational calculus is a variant of the predicate calculus with queries being open formulas of the language. Formally, by the *answer* for a query $Q(x)$ in the database DB (which is viewed as a first order theory, a combination of database extension and intension) we mean the set of *all* substitutions of domain constants for the variable x such that the resulting formula is a semantic consequence of the database DB. If the database is relational—i.e., if the database state is a collection of relations—then for any query $Q(x)$ we can compute the answer for this query by a *relational algebra expression* built from the following relational algebra operations:

(1) *Selection*: Denoted by $\sigma_E(R)$ where E is a selection condition generated by descriptors of the form $(A = a)$ or $(A = B)$ (where A, B are the attribute names) and conjunction, disjunction and negation. For example, $E = (A = a) \wedge (B = b) \vee \neg(A \neq B)$ is a selection condition. It is easy to see that the selection conditions correspond to formulas expressed in the predicate calculus of equality without quantifiers.

(2) *Projection*: Denoted $\pi_X(R)$, where X is a subset of the set of attributes of R which is defined as $\{t | \exists s \in R$ such that $t[X] = s[X]\}$, where $t[X]$ denotes the restriction of tuple t to the set of attributes X.

(3) *Natural join*: Denoted by $R \bowtie S$, and defined as the set of all tuples t defined over the union of the set of attributes of R and the set of attributes of S and such that the projection of the tuple t over the set of attributes of R belongs to R and the projection of t over the set of attributes of S belongs to S.

(4) *Union and difference*: Standard set theoretical operations on relations.

We will define P-queries as queries built only from projections, PS-queries as queries built from projection and selection, and finally PSJ-queries as queries built from projection, join, and selection.

Traditionally, therefore, the answer for the query is defined as a set of tuples. In the next section we are going to modify this concept suitably.

[2] If we exclude trivial tautologies of the form $p(x_1, \ldots, x_n) \rightarrow p(x_1, \ldots, x_n)$.

3. RULES IN THE ANSWERS OF QUERIES—RULE TRANSFORMATION

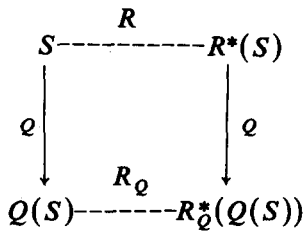
We will now describe the conditions under which we can include rules as part of the *answer* for a *query*. Although the logic programming approach does not make any distinction between the query and the data (both are parts of the same program), we will make such a distinction. We will refer to the query usually by a relational algebra expression. This is more convenient in our approach, which is essentially bottom up.

Formally, we will be interested in the following three situations:

i. The rules in the answer for a query are defined totally in terms of the query predicate and con be "postponed" after the evaluation of the query. Formally, this will correspond to the following formula:

For every set of assertions S, $Q(R^*(S)) = R_Q^*(Q(S))$, where R_Q is the new "postponed" or transformed set of rules.

This situation can be illustrated by the following diagram:



Example 2. Take the predicate Teach(x, y, x), meaning that a teacher x teaches a course y to a student z, and take the (CONST) rule

$$\text{Teach}(x, \text{Databases}, z) \rightarrow \text{Teach}(x, \text{File Systems}, z)$$

If the query Q is a simple projection $\pi_{\text{Teacher, Course}}$, we can get the following rule as the answer for this query:

$$r_Q = Q(x, \text{Databases}) \rightarrow Q(x, \text{File Systems})$$

It is easy to see that this rule r_Q satisfies the formula above—i.e., the original rule can be totally transformed.

ii. The rules in the answer for a query are defined totally in terms of the query predicate but may be only partially postponed, and some of them will have to be performed prior to the evaluation of the query. In this case we have:

For every set of assertions S, $Q(R^*(S)) = R_Q^* Q(R_0^*(S))$, where R_0 is the set of rules which have to be evaluated prior to the query itself (i.e., which cannot be postponed).

For example, take the CONST rule as before and the query

$$Q = \sigma_{(\text{Course} \neq \text{Database}) \vee (\text{Teacher} \neq \text{John})}(\text{Teaches}).$$

It is easy to see that the rule

$$r_0 = \text{Teach}(\text{John}, \text{Databases}, z) \rightarrow \text{Teach}(\text{John}, \text{Compilers}, z)$$

will have to be evaluated prior to the query, while

$$r_Q = (x \neq \text{John}) \wedge Q(x, \text{Databases}, z) \rightarrow Q(x, \text{File Systems}, z)$$

can be postponed till after the query evaluation. [Notice that we could really skip the condition $(x \neq \text{John})$ in this rule.]

iii. The rules in the answer for a query are defined not only in terms of the query predicate but also in terms of other database predicates. In this case again we may have both the case of total rule transformation as in situation i and the case of partial query transformation as in situation ii. Formally this situation could be described by the same formula as in the latter case. The only difference is that the user has first to define the set of *concept* predicates, i.e., predicates in terms of which he is going to accept the answer to the query. The query Q is now no longer just a single relational expression (or formula of predicate calculus). It is rather a pair (Q, C) consisting of the query Q and the set C of concept predicates.

The term "concept" is used here to reflect the fact that these predicates will be treated as "atomic" concepts which can then occur in rules in an answer for a query. We will assume that concept predicates do not occur in the query itself (otherwise we could allow trivially the query definition as the possible answer for a query). By *concept rules* we mean rules in which all foreign predicates are concept predicates.

In all these situations we assume that all the rules in an answer for a query are recursive, i.e., they contain the query predicate both in the consequent and in the body of the rule. This assumption prevents us from considering trivial rule transformations, such as for example including a query definition (in the form of a Horn formula with a query predicate as a consequent) in an answer for a query. Besides, since the recursive rules are the hardest to evaluate, their transformation could lead to significant computational gains.

Notice that in all the above cases the rule transformation is obtained independently of the set of assertions.

We must also point out here that we are interested in getting *active* rules in an answer for a query—that is, we want our rules still to play the role of inference rules, as in the database itself. Certainly, we may also talk about transforming rules into *passive* rules in an answer for a query; however, this more in the spirit of *derived* integrity constraints, considered e.g. in [3] and [5], and will not be considered here.

3.1 Single Rules

We start by describing the method for single rule transformation by a single relational expression. We will give precise conditions under which a single application of a given rule r can be transformed by a single relational operation. Later we will consider the cases of an arbitrary number of applications of a single rule and of a set of rules.

Projection. Let π_X be a projection operation over an instance of the predicate occurring in the consequent of the rule, and let l be any atom in which the

consequent predicate occurs. By $l[X]$ we denote the atom resulting from l by dropping all positions which correspond to the columns outside of X. We will say that the symbols which occur in these columns are *projected out* by the projection operation.

We have the following lemma:

Lemma. A rule r is π-transformable[3] *iff π_X does not project out either constants or bound variables. In this case, the result of the projection is the rule in which each occurrence of the literal l generated by the consequent predicate is replaced by its projection $l[X]$.*

PROOF. If the condition of the lemma is not satisfied, we can construct two different database instances which will have the same projection before the derivation of additional tuples by the rule r and different projections after the closure is computed, which demonstrates that the construction of the derived rule is impossible. If the conditions of the theorem are met, then by easy verification we obtain that the transformed rule satisfies the conditions of π-transformability.

Example 3. Let $r = p(x, y, z', w) \wedge p(x', y, z, w) \rightarrow p(x, y, z, w)$. Let $X = \{A, C\}$. Then r is not π_X-transformable, because the variable w is bound (it has two different occurrences). It is easy to see that for

$$S_1 = \begin{array}{cccc} A & B & C & D \\ \hline a & b & c & d \\ a' & b' & c' & d \end{array} \quad \text{and} \quad S_2 = \begin{array}{cccc} A & B & C & D \\ \hline a & b & c & d \\ a' & b & c' & d' \end{array}$$

where $a, b, \ldots, d'$ are domain constants, we will have $\pi_{AC}(S_1) = \pi_{AC}(S_2)$ but not

$$\pi_{AC}(r(S_1)) = \pi_{AC}(r(S_2)).$$

On the other hand, if we take any X such that $B \in X$ and $D \in X$, then the rule r is π_X-transformable. If $X = \{B, C, D\}$, then the rule

$$\pi_X(r) = Q(y, z', w) \wedge Q(y, z, w') \rightarrow Q(y, z, w)$$

is part of the result of the query $Q = \pi_{BCD}(r)$. In this case the rule r does not have to be evaluated on the operand R, but could remain active and be transformed independently into an active rule in the answer for the query. It simply does not pay to evaluate the rule r over R in this case. Notice also that rule transformation does not require any accesses to the assertional part of the database and may be even done at compile time in many cases. In a similar way, the rule $r = P(x, y, z) \wedge W(y, y') \wedge U(z, w) \rightarrow P(x, y', w)$ is not π_{AB}-transformable, because the variable z which is projected out is bound (by the predicate U). However, if we apply projection over the attributes B and C, then the rule can be transformed to the form

$$Q(y, z) \wedge W(y, y') \wedge U(z, w) \rightarrow Q(y, w),$$

where Q is a query predicate.

Let us now describe the join operation and the transformability of the rules by the join.

[3] That is, there exists a rule r' such that $\pi_X(r(S)) = r'(\pi_X(S))$.

Join. Let r_1 and r_2 be two rules defined for relations R and S.

In order to determine the transformability of these rules, we have to determine when the following holds:

$$r_1(R) \bowtie r_2(S) = r_Q(R \bowtie S).$$

Let $X = Y \cap Z$ be the common set of attributes for R and S. In order for the above equality to hold, the following necessary and sufficient conditions must be satisfied:

$$\pi_X(r_1(R)) = \pi_X(R) \text{ and } \pi_X(r_2(S)) = \pi_X(S) \qquad \text{for every } R \text{ and } S. \tag{1}$$

The condition (1) simply means that the rules r_1 and r_2 leave projections on X of R and S invariant. Indeed, otherwise the rules r_1 and r_2 could produce some tuples which would contribute to the join if evaluated prior to the join itself. In consequence it would be incorrect to postpone their evaluation till after the join. This above condition is very easy to test and can be done using standard methods of testing the equivalence of relational expressions.

The rules r_1 and r_2 in this case will transform to a set of rules rather than to just a single rule. The resulting set will contain three rules, denoted by r_1', r_2', and r_∞, constructed in the following way:

Let r_1 define the predicate p. By a p-literal we will mean a literal generated by a predicate symbol p. The rule r_1' will be formed from the rule r_1 by replacing all p-literals in the body of the rule as well as in its consequent by Q-literals, where Q is a new predicate with the arity corresponding to the arity of the join. Each p-literal l will be replaced by the new Q-literal and will have the same variables as l on the attributes corresponding to the attributes of p. The other positions of the new Q-literal will be formed by a vector of new variables $z_{i_1}, \ldots, z_{i_n}$. These variables will be the same for any new Q-literal replacing any p-literal either in the body or in the consequence of the rule. All the other literals in r_1 will remain unchanged. The rule r_2' will be formed in the identical way.

There will be also the third rule r_∞ in the join, which will be a unirelational recursive rule generated by the predicate Q. This rule will be a proper join dependency corresponding to the join. It will be formed in the following way:

There will be two Q-literals in the body, each corresponding to one of the arguments of the join operation. The consequent literal will have the form $Q(x_1, \ldots, x_n)$, and the body Q-literal corresponding to the argument R will have the same variables as the consequent literal on all attributes in Q corresponding to the attributes in R. The second literal will be formed in the identical way with respect to the second argument of the join. All the other variables in the rule will be pairwise different and different from the variables occurring in the consequent predicate.

The third rule always holds in the join of the two relations (with the sets of attributes Y and Z respectively). In other words, even join of two relations R and S over the sets of attributes Y and Z satisfies this rule. It is therefore a derived integrity constraint or a passive rule in our previous terminology. Can this rule be made active? Yes, if we perform the "simplified" join over the tables R and S and just add the rule r_∞ as an additional rule which computes the remaining tuples if necessary. This simplified join has the following form: For each tuple in R, find one tuple in S which can be joined with it, make a join of them, and include the tuple in

the result. Do the same thing for each tuple in S. What we are saving here is space, since we do not have to represent all the combinations of tuples of R and tuples in S which match on X; they can always be generated by the r_∞ "on demand". In such a case the size of the result is in the worst case $\#R + \#S$ instead of $\#R \cdot \#S$. In this case the rule r becomes the "active" part of the result. This method therefore constitutes a good alternative way of evaluating a join.

Example 4. Let $R[A, B]$ and $S[B, C]$ be two relational schemes. Let $r_1 = R(a, b_1) \to R(a, b_2)$ and $r_2 = S(b, c_1) \to S(b, c_2)$.

This set of rules is not transformable, because for $X = \{A, B\} \cap \{B, C\} = \{B\}$ the rule r_1 is not X-invariant, i.e., it does not satisfy the condition (1). However, the rule r_∞ will always hold in the result of this query.

The situation changes when instead of the rule r_1 we consider the rule $r_1' = R(a_1, b) \to R(a_2, b)$. The resulting set of rules is transformable. Indeed, the condition (1) is now satisfied and the transformed set of rules has the following form:

$$r_1' = Q(a_1, b, y) \to Q(a_2, b, y),$$
$$r_2' = Q(z, b, c_1) \to Q(z, b, c_2),$$
$$r_\infty = Q(x, y, z') \wedge Q(x', y, z) \to Q(x, y, z).$$

We can make all these three rules active in this case by evaluating a "simplified" join as described above.

Selection. This is the most important operation, both because it is frequently used and also because it is the most "restrictive" one in producing small answers even from large databases. It definitely pays to evaluate rules over the result of the selection instead of evaluating them over the original database. This leads to a rule transformation which intuitively speaking "instantiates" the rule and makes it "more specific".

Let E be a selection condition. If t is a tuple (possibly with variables), by $E(t)$ we will denote a formula such that

$$v(E(t)) = \text{true} \quad \text{iff} \quad E(v(t)) \text{ is true},$$

where v is the substitution of the constants for variables.

Let σ_E be the selection operation. We will have two conditions under which the rule r will be transformed into either an active or a passive rule.

Let the rule r have the form $P(t_1), \ldots, P(t_n) \wedge W \to P(t_{n+1})$, where W is a conjunction of all the other literals in which the consequent predicate does not occur. We assume here again (as in case of projection) that our rule is built only from concept predicates and the consequent predicate. We will distinguish the following three cases:

(1) The rule r is σ_E-transformable into an *active rule* iff $E(t_{n+1}) \leftrightarrow E(t_1) \wedge \cdots \wedge E(t_n)$. In this case we simply have $\sigma_E(r) = r$.

(2) The rule $r = P(t_1) \wedge \cdots \wedge P(t_n) \wedge W \to P(t_{n+1})$ is σ_E-transformable into a *passive rule* iff $E(t_1) \wedge \cdots \wedge E(t_n) \to E(t_{n+1})$ holds.

(3) The rule r is independent of the selection condition E iff $E(t_{n+1})$ is false. In this case the rule r does not have to be evaluated at all as far as the selection query is concerned.

(4) Indeed, in the case (1) we simply have that $\sigma_E(r(R)) = r(\sigma_E(R))$. It is easy to see that the condition (1) is a necessary and sufficient condition for this equation to hold. In the case (2) that is not true, but the rule r is still implied as the integrity constraint in the result.

In the situation when a rule cannot be transformed, we can decompose the rule into two parts, one of which is transformable while the other is not.

Definition. Let $P(t_1) \wedge \cdots \wedge P(t_n) \wedge W \rightarrow P(t_{n+1})$ be the rule. Let $E^* = E(t_1) \wedge \cdots \wedge E(t_n)$, and let

$$E' = E^* \wedge E(t_{n+1})$$

and

$$-E' = \neg(E^*) \wedge E(t_{n+1}).$$

By r^E we denote the rule $E' \wedge P(t_1) \wedge \cdots \wedge P(t_n) \wedge W \rightarrow P(t_{n+1})$, and by r^{-E} we denote the rule $-E' \wedge P(t_1) \wedge \cdots \wedge P(t_n) \wedge W \rightarrow P(t_{n+1})$. If either of the preconditions E' or $-E'$ is false, then the corresponding rule r^E or r^{-E} will be empty (the empty rule will be denoted by δ).

It is easy to see why the rule r^{-E} will have to be evaluated prior to the section. It takes into account the tuples which themselves do not satisfy the selection condition but *contribute* to the selection condition through the original rule. Therefore this modified rule will have to be evaluated prior to the query. On the other hand, the rule r^{+E}) can be evaluated after the selection condition—indeed, all P-literals in the body of the rule satisfy the selection condition, so no information will be lost by postponing this rule.

Example 5. Let our rule r have the form

$$P(x, y, z) \wedge W(z, u) \wedge U(y, v) \rightarrow P(x, u, v),$$

and let the selection condition $E = (A \neq a) \wedge [(B = b) \vee (C \neq c)]$. This rule can be decomposed into two rules:

$$r^E = P(x, y, z) \wedge W(z, u) \wedge U(y, v) \wedge E(x, y, z)$$
$$\wedge (u = b) \vee (v \neq c) \rightarrow P(x, u, v)$$

The descriptor $(x \neq a)$ which seems to be missing above is implied by $E(x, y, z)$:

$$r^{-E} = P(x, y, z) \wedge [(x = a) \vee (y \neq b) \wedge (z = c)]$$
$$\wedge W(z, u) \wedge U(y, v) \wedge (x \neq a) \wedge [(u = b) \vee (v \neq c)] \rightarrow P(x, u, v).$$

Since the first rule belongs to the answer to our query, we can replace it by

$$Q(x, y, z) \wedge W(z, u) \wedge U(y, v) \wedge (u = b) \vee (v \neq c) \rightarrow Q(x, u, v),$$

where Q is the query predicate.

Queries. Now we have to say a few words about the transformations of individual rules by whole relational algebra expressions. The ideal situation would be if we could simply apply our transformation rules for the individual relational operators in a recursive way. Unfortunately, it is not that simple, and we will have to restrict

the syntax of the queries (expressions) to effectively use the rules introduced above. The most problematic operation will be the natural join, since it generates a nonlinear recursive rule as the result of the rule transformation (i.e. a join dependency r_∞). If additionally the argument predicates of the join are defined by some other recursive rules (e.g. strongly linear rules), then after applying the join we will produce (in case all rules are transformable) the *set* of (not necessarily linear) recursive rules as the set of transformed rules. If there is a next operation in our expression which governs the join, it will not have a single rule as an argument, but rather a set of complex rules. As we will see in the next section, the transformation of the set of recursive rules is considerably harder than the transformation of individual rules and is not always possible. Therefore, if we want to transform the rules effectively, we should avoid expressions in which the join operation does not occur as the top (final) operation but is an argument of the other operation. Notice that neither selection nor projection suffers from the above problem; they always transform single rules into single rules.

Idempotent Rules. So far we have described conditions for transforming single applications of a single rule by individual relational operations. These conditions can be directly applied in a case of an arbitrary number of application of a single rule $r^*(S)$ in the case of projection, selection, and join. Unfortunately, the decomposition of the rules modulo the selection condition cannot be directly generalized to the case of an arbitrary number of applications of a single rule. Such a decomposition is only possible when our rule r is *idempotent*, i.e., no more than one application of the rule is necessary in order to reach a fixpoint (the limit of the rule is equal to one). This is the case for CONST rules and join dependencies, and it turns out that it will also be the case for strongly linear rules which we want to consider, if we make some additional assumptions about concept definitions. We will come back to this point when discussing the transformation of the sets of rules.

4. TRANSFORMING SETS OF RULES

Rules may be related to each other in a complicated way. In most cases we cannot simply apply the procedure described above to a set of rules on a rule by rule basis. Hull in his paper [3] demonstrated a simple family of rules (template dependencies) such that a simple projection of it is not finitely specifiable by Horn clauses. In other words, he demonstrated that the set of rules holding in a simple projection may be not finite. We provide here his example.

Example 6 [3]. Let the rules have the following form:

$$r_1 = P(a, b', c', d', e), P(a, b, c, d, e') \rightarrow P(a, b, c, d, e),$$

$$r_2 = P(a', b, c', d', e), P(a, b, c, d, e') \rightarrow P(a, b, c, d, e),$$

$$r_3 = P(a', b', c, d, e), P(a, b, c, d', e) \rightarrow P(a, b, c, d, e),$$

where $a, b, c, d, e, a', b', c', d', e'$ are variables. Let us consider the projection π_{ABCD}. Hull demonstrated that the set of all relations which are the projections on $ABCD$ of the relations satisfying this set of rules is not finitely specifiable by a set of Horn clauses. In our terms this means that the set of rules holding in the above projection is even not finitely specifiable.

Frequently it turns out that although a given set of rules is not transformable by the query, some other equivalent set of rules is. The following example illustrates the point:

Example 7. Given two rules $p(a) \rightarrow p(b)$ and $p(b) \rightarrow p(c)$ and a selection $\sigma_{(A=a) \vee (A=c)}$, we can get the implied rule $p(a) \rightarrow p(c)$ by the application of modus ponens to the above two rules. In this case only the rule $p(b) \rightarrow p(c)$ should be evaluated before the query; the implied rule could be part of the answer. Therefore the "closure" of this set of rules allows us to postpone one of the rules till after the query evaluation, which was not the case for the original set of rules.

In this section we are going to describe conditions under which sets of rules are transformable. In general we will be looking for the cases in which the transformation of a set of rules can be reduced to the transformation of a single rule or can be done on a rule by rule basis. In case the given set of rules is not transformable, we will be interested in the construction of sets of rules which are equivalent to the given set and in the same time "easier" to transform.

We will select here some properties of the sets of rules which make them easy to transform. All of these properties will require certain forms of modularity or independence.

Definition (Modular decomposability). We will say that a set of rules R is modularly decomposable into $\langle R_0, R_1 \rangle$ (where the union of R_0 and R_1 is equal to R) iff

$$R^*(S) = R_0^*(R_1^*(S))$$

for any set of assertions S.

Notice that the set R of rules may be modularly decomposable into $\langle R_0, R_1 \rangle$ but not into $\langle R_1, R_0 \rangle$.

Definition (Strong independence). A set of rules R is *strongly independent* iff for any set of assertions S

$$R^*(S) = \bigcup_{r \in R} r(S).$$

Definition (Independence). Independence is the same as strong independence, but with respect to the closures of individual rules, i.e. with respect to r_i^* instead of r_i for $i = 1, \ldots, n$.

All these definitions correspond to some kind of modularity, since they express the fixpoint of a set of rules in terms of the fixpoints of the subsets of this set of rules.

We will say that a set of rules R is *totally* transformable by the query Q iff all the individual rules in R are totally transformable.

We will start our description of transformations of the sets of rules from the situation when the set of rules is modularly decomposable. Later on we will analyze

the transformation of independent sets of rules and show how, given the set of rules, to construct an independent set of rules which is equivalent to it.

In the following subsections we will show two principal ways in which we are going to transform rules in a database. The first will be applied to the situation when the set of rules is modularly decomposable. In the second the set of rules will be transformed to a *single* rule. Finally, at the end of the section we will discuss the general methods of constructing an independent set of rules which is equivalent to the given one.

4.1 Transforming Modularly Decomposable Sets of Rules

By a *totally transformable* set of rules we mean a set of rules in which each individual rule is directly transformable (i.e., it satisfies our initial condition for situation i in section 3).

Let us start with a transformation of a totally transformable set of rules. We will assume that all these rule are concept rules; otherwise we have to eliminate nonconcept rules first. A totally transformable set of rules can be transformed rule by rule in the case of queries involving projections and selections. Indeed, such queries can be distributed over the union operator and our property directly follows. The join operator can be distributed too, but in this case the "total transformability" of the set of rules is more restrictive. It requires that no rule violate the join requirement (specified in the join transformation rule). The transformation rules in this case will require us to take each possible pair of rules, one from each argument of the join, and perform a transformation according to the previously described join transformation rule.

The case of total transformability of a set of rules enables us therefore to reduce the problem of the transformation of a set of rules to the problem of the transformation of single rules. What happens, however, if the set of rules under consideration is not totally transformable? In this case we have to do the following:

(1) Identify a modular decomposition $\langle R_0, R_1 \rangle$ of the set R such that R_0 is totally transformable (clearly therefore we require that all rules in R_0 are concept rules).

(2) Evaluate all nonconcept rules and all rules from R_1 or used by R_1 prior to the query Q, and transform the rules from R_0 on a rule by rule basis.

Indeed, since the rules R_1 will have to be evaluated prior to the query Q, so must be all the rules used by the rules from R_1, regardless of whether they are concept or nonconcept rules. Notice that in general we will be interested in the maximization of the subset R_0 which have the above properties.

It is clear that the modular decomposability of the set of rules allows the components of such decompositions to be treated as independent modules. As such, they can be transformed separately. It is also easy to see that even when the subset of rules is totally transformable but is not an independent submodule of the total set of rules, then the rule transformation is not possible.

Unfortunately we do not have sufficient and necessary conditions for the modular decomposability; we can provide only certain sufficient conditions. One of such

useful sufficient conditions is presented here:

Definition. Rule r_2 is *invariant* with respect to rule r_1 iff for any S

$$r_2(r_1(S)) = r_2(S).$$

In other words, the rule r_1 is not "affected" by the rule r_1. The condition given in the above definition can be easily checked using, for example, standard methods of testing the equivalence of relational expressions.

We have now the following sufficient condition for the modular decomposability:

Condition 1. Let $R = R_0 \cup R_1$. *If all rules in* R_1 *are invariant with respect to all the rules in* R_0, *then*

$$R^* = R_0^*(R_1^*).$$

The proof is a straightforward consequence of the definition.

The other condition is related to strongly linear rules.

Condition 2. Let R_1 *and* R_2 *be two disjoint sets of strongly linear rules defining the same predicate. If for some rule* $r \in R_1$ *and any rule* $s \in R_2$ *we have* $\mathrm{Arg}(r) \cap \mathrm{Arg}(s) = \varnothing$, *then*

$$R^*(S) = R_1^*(R_2^*(S)) = R_2^*(R_1^*(S))$$

for any S.

The proof is again trivial. Such two sets of linear rules will be called *orthogonal.* Obviously, when we can establish the total transformability of a subset S of R which is orthogonal to its complement in R, then we can simply transform S and evaluate all the other rules (i.e. from $R \setminus S$) prior to the query. This is simply because orthogonality implies the modular decomposability. We will now present an example of such a situation and the transformation procedure in this case:

Example 8. As an illustration of modular decomposability we can consider Example 1 with two rules: the first with the "Group" predicate and the second with the "Prerequisite" predicate. Both rules are strongly linear and orthogonal. Therefore for this set of rules $R = \{r_1, r_2\}$ we have that:

$$R^* = r_1^* r_2^*(S) = r_2^* r_1^*(S).$$

Clearly, as far as the first query is concerned, the rule r_2 it totally transformable, while the r_1 is not. This is why the first rule has to be evaluated prior to the query, while the second one could be transformed. For the second query the situation is exactly the opposite. In both cases we can transform the rules separately because of the possibility of modular decomposition.

An interesting situation occurs when a query is simply a selection operator (or is built from selection operators). When a rule r is not selection transformable, we still have a change, as the previous section indicates, to decompose the rule into two

parts, r^E which is transformable and r^{-E} which is not. This is the case, however, only when the rule r is idempotent. We would like to establish conditions under which the rule r can be decomposed in such a way when it is a member of the set of rules. A necessary condition is the idempotence of the rule. It is not however, (as it was for transformation of a single rule) a sufficient condition. Let R' be a result of replacement of r in the set R by $\{r^E, r^{-E}\}$. For sufficiency we need the existence of a modular decomposition of R' into modular components R'_0 and R'_1 such that r^E will belong to R'_0 and r^{-E} to R'_1. Indeed, otherwise rules from R'_1 would have to be evaluated at least one more time after R'_0, contradicting the correctness of the transformation of R'_0.

In the next subsection we demonstrate a situation when the set of linear rules can be replaced by a single linear rule.

4.2 Saturating Transformation

The following theorem is a key to our transformation of a set of linear rules to a single linear rule:

Theorem. Let Σ be a set of strongly linear p-rules. There exists a set of rules Σ' which is equivalent to Σ modulo p (i.e., both sets imply the same formulas about p) and such that p occurs only in one rule, which is strongly linear and weakly idempotent.

In other words, in the above theorem we claim that there exists a transformation of a set of strongly linear rules into a *single* idempotent rule. Clearly therefore, the problem of the transformation of a set of rules can be reduced here to the problem of transformation of a single rule.

PROOF. Let $\Sigma = \{r_1, \ldots, r_k\}$. Let $\{w_1, \ldots, w_k\}$ be a set of writing formulas for $r_1, \ldots, r_k$ respectively. By the *saturating* transformation T_Σ of Σ we mean the following one: Add one additional predicate symbol T to the language. The rank of this symbol (the number of columns) should be equal to the cardinality of ACTIVE $= \bigcup\{\text{Arg}(r_i) \cup \text{Res}(r_i) : r_i \in \Sigma\}$ times two. Intuitively, the predicate T will describe how the positions from ACTIVE will be changed by the rules, by showing their "new" values. That is why we need the predicate T with arity twice as large as the cardinality of ACTIVE.

Replace Σ as follows:

(1) a single linear rule

$$r_T = p(x_1, \ldots, x_n) \wedge T(x_{i_1}, \ldots, x_{i_p}, z_{i_1}, \ldots, z_{i_p}) \to p(u_1, \ldots, u_n),$$

where T is the predicate added to the language, ACTIVE $= \{i_1, \ldots, i_p\}$, and:

$$u_i = \begin{cases} z_{i_k} & \text{if } \ i = i_k \in \text{ACTIVE}, \\ x_i & \text{otherwise.} \end{cases}$$

Some of the variables z_{i_k} may be equal to x_{i_k}, for example in case the position i_k is not changed by any rule.

(2) Add two sets of rules defining this additional predicate symbol T:
Initialization rules. For each writing formula w_i introduce the following rule:

$$w_i(x_{i_1}, \ldots, x_{i_n}, u_{j_1}, \ldots, u_{j_m}, y_{k_1}, \ldots, y_{k_l}) \to T(x_1, \ldots, x_k, z_1, \ldots, z_k),$$

where the x-variable on the left hand side are the variables which occur both in the L-occurrence of the consequent predicate and in the writing formula, the u-variables are the variables which occur *only* in the writing formula (i.e. neither in L- nor in R-occurrences of the consequent predicate), and finally the y-variables occur in the writing formula and in the R-occurrence of the consequent predicate. Moreover,

$$z_i = \begin{cases} x_i & \text{if} \quad i \notin \text{Res}(w_i), \\ y_i & \text{if} \quad i \in \text{Res}(w_i), \text{ assuming that } y_i \\ & \text{occurs at the } i\text{th position} \\ & \text{of a consequent predicate.} \end{cases}$$

As it is easy to see, the effect of this added rule the predicate T will leave all positions unchanged with the exception of the ones written by w_i.

Continuation rule. This is the transitive closure of T, i.e.

$$T(x_1, \ldots, x_n, y_1, \ldots, y_n) \wedge T(y_1, \ldots, y_n, z_1, \ldots, z_n) \rightarrow T(x_1, \ldots, x_n, z_1, \ldots, z_n).$$

It is easy to see that the obtained set of rules is equivalent to the original set of rules with respect to p. Indeed, any sequence of appliations of original linar rules $r_1, \ldots, r_k$ can be "simulated" by the evaluation of a new single rule with the predicate T. In the same way, any sequence of appliations of a single new rule can be simulated by some sequence of applications of the original rules $r_1, \ldots, r_k$. □

Notice also another interesting interpretation of the above theorem: In order to compute the least fixpoint of strongly linear rules, it is sufficient to extend relational algebra only by a transitive closure operator (to compute one of transformed set of rules). The full power of least fixpoint queries is therefore much too large for strongly linear rules.

To benefit from such a transformation a user must include T in a set of his concept predicates, since otherwise a single new rule could not be a subject of transformation at all.

Example 9. Assume that we have the following set of rules:

$\text{MayTeach}(x, y) \wedge \text{Prefer}(x, y, z) \rightarrow \text{MayTeach}(x, z)$,

$\text{MayTeach}(x, y) \wedge \text{Better}(x, y, v) \rightarrow \text{MayTeach}(v, y)$.

The intuitive meaning of the first rule is the following: If a teacher is able to teach a given course he also has to be able to teach all the courses which he prefers [these courses depend on the individual teacher; therefore the predicate $\text{Prefer}(x, y, z)$ is ternary, meaning "Teacher x prefers course z to course y"]. The meaning of the second rule is as follows: If the teacher x is able to teach the course y, so must be any teacher v who is better then x in y (again, $\text{Better}(x, y, z)$ means that "z" is better than "x" on the subject "y".)

Suppose now we have a query: Give me all teachers and the undergraduate courses[4] which they may teach. It is easy to see that neither of these two rules is

[4]Assuming we have an additional unary predicate $\text{Undergraduate}(x)$ at our disposal.

transformable with respect to this query. Indeed, this query is analogous to a pure selection query with respect to which neither rule is transformable. If however we apply a saturation transformation with respect to this set of rules, we will be able to transform the resulting rule.

Indeed, let us introduce the additional predicate

$$\text{Betterknow}(x, y, u, v)$$

with the meaning "u knows v better then x knows y", and the rule

$$r_T = \text{MayTeach}(x, y) \wedge \text{Betterknow}(x, y, u, v) \rightarrow \text{MayTeach}(u, v)$$

together with the rules defining the Betterknow predicate:

$$\text{Prefer}(x, y, z) \rightarrow \text{Betterknow}(x, y, x, z),^5$$

$$\text{Better}(y, z, u) \rightarrow \text{Betterknow}(y, u, z, u),$$

$$\text{Betterknow}(x, y, u, v) \wedge \text{Betterknow}(u, v, x', y') \rightarrow \text{Betterknow}(x, y, x', y').$$

According to the above theorem the second set of rules is equivalent to the original set of rules with respect to a predicate MayTeach. If the user u accepts the Betterknow predicate as the concept predicate, then the answer for the query must include the following rule:

$$Q(x, y) \wedge \text{Betterknow}(x, y, u, v) \wedge \text{Undergraduate}(v) \rightarrow Q(u, v).$$

4.3 Transforming Sets of Bounded and Relatively Bounded Rules

In this subsection we are going to consider other situations in which a set of rules can be transformed on a rule by rule basis. This is the case when the set of rules is independent. That is a very rare property (though it will happen for example when the set of rules under consideration has a connection graph with no edges). Thus we will rather look for conditions under which a set of rules which is not independent can still be transformed to a logically equivalent independent form. It turns out, for example, that in a situation when a set of concept rules is p-bounded, where p is a predicate under consideration, we can always generate a finite set of rules which is equivalent to the original one and independent. This is so because, if the set of rules is bounded, only a finite bounded part of the BFT is relevant. An independent set R' which is equivalent to R can be generated by generating the p-closure for any predicate p. This p-closure can be generated simply by building a BFT with $:- p(x_1, \ldots, x_n)$ as the goal node. With each node of the BFT we can associate the (answer) substitution θ generated down to this node. Let the node of the BFT have the form

$$:- P_1(t_1), \ldots, P_m(t_m).$$

With this node we can associate the following rule:

$$P_1(t_1), \ldots, P_m(t_m) \rightarrow p(\theta(x_1), \ldots, \theta(x_n))$$

where p is the "goal" predicate.

[5] This name convention is based on philosophical assumption that if x prefers teaching y than to teaching z, then x knows y better than he knows z.

Obviously, this rule follows by the resolution inference rule from the original set of rules. Notice that in case the node is empty (we do not expect this to happen in our case unless we allow pure assertions to occur in R) we will get a "pure" answer substitution as the rule associated with the node. The set of all rules associated with any node of the BFT for a goal $:-p(x_1, \ldots, x_n)$ will be called a *p-closure* of the set of rules R. The p-closure of the set of rules can therefore be computed at compile time of the database by the "symbolic execution" of the set of rules (i.e., the execution of the set of rules of the database as a logic program). The p-closure (which simply corresponds to the logical closure of the set of rules by resolution) is of course logically equivalent to R. Moreover it is independent, as the following corollary shows:

Corollary 1. Let R be a p-bounded set of rules. The p-closure of R is independent.

The symbolic execution of a set of rules has to be done only once at the compile time for the database. The size of the resulting set of rules may still, in the worst case, be exponential (with respect to the size of the set of rules). We do not see any better alternative: after all, if we did not evaluate the rules symbolically at compile time, we would have to do possibly exponential compuations at run time, which would be much more expensive. An additional advantage of the closure computation at compile time is the possibility of the *optimization* of the resulting, independent set of rules in such a way that redundancy is avoided (for example, no two rules imply each other).

The above method will be useful only when the closure of a set of rules can be computed. That will be the case, for example, for sets of nonrecursive rules. Indeed, their BFT is finite; therefore the process of closure generation will always terminate in this case. This will be the case for CONST rules. Sets of bounded rules will be also transformable to an equivalent unbounded form. The trouble is that the problem of deciding boundness is undecidable [1]. The hope again is in finding special families of rules for which the boundness problem is decidable. The class of functional rules defined in [4] has this property. The transformation of such rules is considered in that paper too, so we will not elaborate on it any further here. The generation of the closure will however be used further in the case of transformation of the CONST rules.

Example 10. Example 7 clearly illustrates the case when the original set of rules cannot be transformed, while the closure (in this case obviously finite) is independent and can be transformed on a rule by rule basis.

4.4 Examples of Transformation of Set of Rules

Let database predicates have the form:

Supply(x, y, z), meaning that supplier x supplies parts y to a project z,

Subpart(u, v), meaning that part u is a subpart of v,

Related(x, y), meaning that projects x and y are related (similar area),

Location(x, y), meaning that supplier x has location y (he may have a number of locations).

Let our set of rules have a form

$$r_1 = \text{Supply}(x, y, z) \wedge \text{Subpart}(u, y) \rightarrow \text{Supply}(x, u, z)$$

with the interpretation that if a supplier supplies a project with a part, he must supply this project with all subparts of this part, and

$$r_2 = \text{Supply}(x, y, z) \wedge \text{Related}(w, z) \rightarrow \text{Supply}(x, y, w)$$

with the interpretation that given supplier supplies all related projects. Additionally we have the transitive closure rule for the Subpart relation. We do not assume anything about the predicate Related; in particular, we do not assume that it is transitive.

Let us assume that the predicates Related and Subpart are concept predicates. Notice first that again, as in the previous example, the rules r_1 and r_2 are orthogonal. Therefore they can be transformed independently.

Let now

$$Q = \pi_{\text{Supplier, Part}}(\text{Supply}) \bowtie \text{Location}$$

be our query or a view.[6] Intensionally this query defines a view which is a table ("report") listing all suppliers, parts, and locations of suppliers.

First of all we have to compute the result of the transformation of our set of rules by a subquery $Q' = \pi_{\text{Supplier, Part}}(\text{Supply})$. We can consider rules r_1 and r_2 separately because of their orthogonality. Clearly r_2 is transformable into the trivial tautology and r_1 is transformed into

$$Q'(x, y) \wedge \text{Subpart}(z, y) \rightarrow Q'(x, z).$$

This rule will be transformable by join in our query Q, since it does not change projection over the attribute "Supplier". We now have to apply the join transformation and finally get the transformed set of rules:

$$r_3 = Q(x, y, z) \wedge \text{Subpart}(v, y) \rightarrow Q(x, v, z),$$

$$r_4 = Q(x, y, z) \wedge Q(u, y, w) \rightarrow Q(x, y, w).$$

The rule r_4 is our r-rule for a join transformation (it is in fact the so-called join dependency). Notice the computational benefit stemming from the rule transformation here. The resulting query (or "view") could be economically stored by evaluating only the simplified join, as we described before, and leaving the above two rules unevaluated. The rule r_3 could generate a huge number of tuples (because of the transitivity of the subpart predicate); in this way these tuples do not have to be stored, since they can always be generated by the corresponding rules. It is also clear here that the answer for the query (view) is of the same type as a database (i.e., information is stored both in terms of tuples and in terms of rules).

In the next subsection we are going to describe rule transformations for the sets of CONST rules.

[6] A *view*, in the database literature [7], is a mapping defining a new "derived" database. Such a definition mapping is frequently a relational algebra expression.

4.5 *Transforming Sets of* CONST *Rules*

We will deal here with sets of unirelational CONST rules. As it is easy to see, these sets of rules will always have finite BFT; therefore the closure computation will be straightforward here.

The unirelational CONST rules have one more important feature: It turns out that if we deal only with rules from CONST, we do not have to compute the whole closure, but most frequently much less.

Fact. If the rules are CONST, *then the only rules which are* π_X*-transformable for any proper subset of the set of all attributes are* 1*-rules.*

PROOF. If a rule is a k-rule for $k > 1$, then each position at each literal in the body of the rule consists of either a constant or the same variable (different variables occur on different positions, though). Therefore, at each position we have either a variable which is bound or a constant. According to our previous criterion such a rule is not projection transformable. □

This fact has an important influence on the overall process of rule transformation in the case of the CONST rules. If any projection occurs in the query (which is most frequently the case), we need not even bother considering k-rules for $k > 1$. We may restrict ourselves to 1-rules and their closures, since according to the above fact no k-rule for $k > 1$ is going to be transformed into a rule in the answer for the query. In other words, we have only to symbolically evaluate the subset of 1-rules from R. This is a considerable simplification, since each rule in the closure of 1-rules is a 1-rule too, while in the general case the size of the rule (number of literals) grows exponentially with the number of times resolution is applied. Hence much space can be saved. Only in cases when we are dealing with the selection operation separately do we really have to compute the closure (at compile time, though).

As before, the general pattern of the transformation algorithm for a set of independent rules is that we apply the above projection, selection, and join on a rule by rule basis. Additionally, if a rule is not selection-transformable, we can always decompose it into two parts, one of which is decomposable and the other not.

If a set of rules R is independent, then the selection operation σ_E is simply performed by decomposing each rule. In this case, for any rule $r \in R$ the rule r^E is the element of the answer. Moreover, the resulting set of rules is independent too. In fact, for the set of CONST rules we can simplify the rule transformation by describing it "at once" for the expressions built from projection (on the proper subset of the set of attributes) and selection.

$$\pi_X\sigma_E(r)) = \begin{cases} \delta & \text{iff } r \text{ is the } k\text{-rule for } k > 1 \\ & \text{or } \pi_X(r) = \delta \text{ or } r^E = \delta, \\ \pi_X(r^E) & \text{otherwise.} \end{cases} \qquad (PS\text{-rule})$$

We can easily generalize these transformation rules to the case when the rule is of the form

$$E_r \wedge P(t_1) \wedge P(t_2) \wedge \cdots \wedge P(t_n) \wedge W \rightarrow P(t),$$

where E_r is the additional condition imposed on the variables in the rule. For

example, such a rule may have the form:

$$(x \neq c_1) \wedge (y \neq b_2) \wedge P(x, y, d) \rightarrow P(x, y, e).$$

In this case $E_r = (x \neq c_1) \wedge (y \neq b_2)$. It is straightforward to generalize our projection, join, and selection rules to capture this important modification. This simplified transformation rule illustrates the point which we made before about the decisive influence of projection (only 1-closure has to be taken into consideration, etc.).

In general, there are classes of queries and classes of rules for which the rule transformation will be very beneficial, and classes for which we do better to apply traditional methods, since we are not going to gain much by the rule transformation. Here is the classification, together with the proper modifications of the general "rule by rule" application of selection, projection, and join transformation rules:

(1) *The class of PS-queries, built only from projection (on the proper subset of the set of attributes) and selection.* This is a particularly good class of queries for the application of rule transformation. Indeed, in this case we only have to compute the closures of 1-rules (at compile time) by the symbolic execution of the set of rules. It is a good idea to label each rule with its derivation tree (i.e., from which rules it was derived and how). In such a case we may also be able to eliminate some redundancies after the rule transformation has been completed and possibly further decrease the number of rules which have to be evaluated prior to the query evaluation. In general the rules which cannot be transformed will have to be evaluated prior to the evaluation of the query. In this way elimination of redundancy could improve the overall computational benefit. In the case of CONST rules this is fairly inexpensive, since effectively we are going to deal only with 1-rules. Basically, if the given rule which is not transformable can be derived from some transformed rule, than it does not have to be evaluated at all, hence saving expensive computation. The main procedure of rule transformation has the following form:

I. Eliminate the rules which are independent of the query, that is, the rules for which either $\pi_X(r) = \delta$ or $E(t_r)$ is false.

II. Apply a PS-rule to the set of rules on a rule by rule basis to compute the set of transformed rules.

III. Select the rules which have to be evaluated prior to the query [the set R_0 from the formula $(**)$]. These are the k-rules for $k > 1$ and the r^{-E}-rules.

In the top down interpretation the rules computed in step III will have to be included as part of the program computing the answer for the query. The rules computed in step II will be included in the answer for the query. Notice that from the practical point of view this is a very important class of queries. It corresponds for example to the mappings in the query language SQUARE [7].

(2) *The class of PSJ-expressions, built from the joins of PS-expressions.* This is a similar case where the closure of 1-rules is sufficient. We apply the same procedure as in (1) followed by the application of the join transformation rules [see (3) below].

(3) *The class of S-queries, built purely from selection conditions.* These queries require in principle the computation of the full closure of the set of rules at

compile time of the database. Then the rule decomposition could be applied on a rule by rule basis. The same applies to the combinations of S-queries with join. This pays only if the closure of the set of rules is not very large, which happens when the limit of the set is small. The limit of the set of rules measures in fact "how close" the set of rules R is to the independent set.

The queries of the first two groups are more frequent, since selection is usually followed by projection.

In general the rule transformation can be applied when the set of rules is "loosely" connected, that is, when the closure of the set of rules is not too large, or when we do not have to compute the full closure (as for PS-queries). We claim that rule transformation should be applied every time it is possible because of the obvious benefits. In addition, let us point out that we have dealt here only with the conditions for obtaining the complete set of rules in the answer for the query. If we give up completeness, we can always apply our transformation (projection, selection, and join) rules directly to the set of rules, getting only some rules in the answer. This may still be beneficial and does not require dealing with the closure.

5. FUNCTIONS AND PREDICATES IN THE ANSWERS FOR QUERIES

The concept of the answer for a query could be further revised to allow function symbols and even whole predicates to occur in it. For example the term Brother(John) may occur in the answer for the query "Who lives with Mary?" even if we do not know who is the brother of John. Technically such an answer would result from aborting the unification (or semantic unification) in the query answering process. This way of representing answers could help in technically handling *infinite* answers resulting from the presence of arithmetic functions (e.g., successor) in the database formulas. For example, we could represent the answer to some query by listing a set of objects and adding the rule "all successors". In a similar way one can include predicates in the answer for a query. For instance as the answer for the query "who is teaching undergraduate courses" one may get the predicate "Assistant Professors". These answer may be further specified if such a request is made by the user. This "abstracted answer" is easier to obtain than the full one. Besides, it may fully satisfy the user's needs at the moment. The abstracted answer may be also given to the user if the system itself finds the query too time consuming to process. Indeed, computing the answers for queries (remember, we require *all* tuples which satisfy the query predicate to be included) may frequently be too hard. One way to cut the computational cost would be to approximate the answers to queries, i.e., not return all the tuples satisfying the answers. This seems rather arbitrary, however. The other option is the one provided above—to give the complete answer for the query but in more "abstracted terms", possibly provided by the user as an option. In this case we may talk about complexity tailored computation which tailors the level of detail of the answer of the query to the computational resources. This is a topic for further research.

6. CONCLUSIONS

We have proposed rule transformation as a technique for query processing which is beneficial both from the conceptural and from the computational point of view. We have established the method of transformation of single rules by relational oper-

ations of projection, join, and selection. We have demonstrated that the situation becomes much more complex when we want to transform sets of rules, but in some cases we were still able to give the algorithm of rule transformation. We have defined classes of queries and rules for which the rule transformation is advantageous. These include queries built from projections and selection and limited occurrences of join.

An interesting extension of this work would be to study the concepts of answers including unevaluated predicates and function symbols. An important future direction of research is to develop the concept of "complexity tailored" information systems in which a user will have a whole hierarchy of answers to a query at his or her disposal. The more abstracted answers would cost less, and the final choice of the type of answer would depend on the limitations on the computational resources of the system.

APPENDIX

In this appendix we will demonstrate examples of the transformation of CONST rules.

Example 11. Let our database scheme consist of two relation schemes:

TAKEN[Student, Program, Major, Minor, Course#]

and

DESCRITPION[Course#, Credits, Field].

The relation TAKEN describes the courses taken by a given student during his stay at the university. Only those students who graduated from the university are stored in this relation, which is a record of the curriculum of the student. The relation DESCRIPTION simply describes the number of credits for a given course. Notice that the relation TAKEN is not normalized—it may be interpreted as a view defined over two relations: one describing students (and their attributes), and the other describing the relationship between students and courses.

The domain of the attribute Program has two elements: "honors" and "regular", depending on whether the student was in an honor program or in a regular program. The attributes Minor and Major describe the corresponding minor and major fields (one each) for a given student. The relation DESCRIPTION is defined over three attributes; the last one, Field, describes the field of a given course and has the same domain as the attributes Major and Minor.

Let us consider the following rules defining instances of our database:

(1) r_1 = "All students who major in economics and take course #211 must also take course #241"

(2) r_2 = "All students who major in computer science and take course #111 must also take course #211"

(3) r_3 = "All honor students majoring in a given field must take all the 400 level courses in the field"

The first two rules represent the prerequisite information. They can be represented

by the following formulas:

(4) $\forall s, y, z$ TAKEN$(s, y,$ Economics, $z, 211) \rightarrow$ TAKEN$(x, y,$ Economics, $z, 241)$

(5) $\forall s, m, p$ TAKEN$(s, p, m,$ Computer Science, $111) \rightarrow$ TAKEN $(s, p, m,$ Computer Science, $211)$

Consider another set of rules about the predicate DESCRIPTION:

(7) "All 400 level courses have 4 credits"

(8) "All 300 level courses have 3 credits"

Let us have now the following query:

Q_1 = "Give me all the students s and courses c such that c is either #211 or #241 and that student s is taking c"

We could simply return the rule r_1 as part of the answer for the query. This is an illustration of the first case of our definition—when the rule can be *totally* transformed.

Now let's consider another similar query which, however, requires some more work:

Q_2 = "Give me all students who took either course #111 or #241"

The answer for this query also contains a rule. However, this rule is no longer one of the rules r_1 or r_2. It is the consequence of these two rules by means of modus ponens. Namely, the rule which is part of the answer has the following form:

r_{12} = "All the minors in computer science who are majors in economics and who took #111 had also taken #241"

Again, instead of evaluating both r_1 and r_2 before the selection condition, we may only evaluate r_2 and postpone the evaluation of the rule r_{12} or just include it in the answer without evaluating it. In the former case, the rule r_{12} will become a passive integrity constraint similar to those "derived constraints" holding in the view (1); in the latter case, when the rule r_{12} remains unevaluated, it will stay as an active, i.e. an inference rule.

Another query for which we could include rules as part of the answer is the following:

"Give me all the set of the tuples of the relation TAKEN describing the students who took 400 level courses"
In this case a modified form of r_3 will be part of the answer, i.e.,

"Honor students took all the courses"
since we are talking only about 400 level courses.

It is even more natural to keep the rules of the second scheme active. Indeed, it definitely does not pay to physically store all the courses and the credits associated with them separately, since we can always use the rules to derive the necessary data. It is helpful also for the queries to manipulate over the unevaluated rules. For example if we ask about the courses which have 3 credits, we could immediately include in the answer the statement "All 300 level courses", which can be later evaluated to the list of courses if the user wants to (see [6] for a similar argument). It could also be left as a meaningful part of the answer.

Example 12. Consider the following set R_S of rules defined over the relation $p(ABC)$:

$$r_1 = p(x_A, b_1, x_C) \to p(x_A, b_2, x_C),$$

$$r_2 = p(x_A, x_B, c_1) \to p(x_A, x_B, c_2),$$

$$r_3 = (x_C \neq c_2) \wedge p(a_1, x_B, x_C) \wedge p(a_2, x_B, x_C) \to p(a_3, x_B, x_C),$$

where $b_1, b_2, c_1, c_2, a_1, a_2, a_3$ are constants and x_A, x_B, and x_C are variables. By symbolic execution of this set of rules (not demonstrated here) we can obtain the following closure of 1-rules and full closure of this set of rules: The closure of 1-rules of R_S has the form

$$R_S \cup \{r_4\},$$

where

$$r_4 = p(x_A, b_1, c_1) \to p(x_A, b_2, c_2)\};$$

the full closure of R_S will additionally contain the following three rules:

$$(x_C \neq c_2) \wedge p(a_1, b_1, x_C) \wedge p(a_2, b_1, x_C) \to p(a_3, b_2, x_C),$$

$$(x_C \neq c_2) \wedge p(a_1, b_2, x_C) \wedge p(a_2, b_1, x_C) \to p(a_3, b_2, x_C),$$

$$(x_C \neq c_2) \wedge p(a_1, b_1, x_C) \wedge p(a_2, b_2, x_C) \to p(a_3, b_2, x_C).$$

Let us start from the S-queries. Given a selection query σ_E, we can decompose each rule of the closure according to E-decomposition. Two extremal situations occur when $r^{-E} = \delta$ for all rules r and $r^E = \delta$ respectively. In the first case all the rules can be transformed directly by the selection and no rule has to be evaluated prior to selection. In the second case no rule can be transformed. The first situation is equivalent to total transformability of all the rules, i.e., it occurs when for any rule r we have $E(t_r) \leftrightarrow \bigwedge_{t \in T_r} E(t)$. This is the case, for example, for any selection condition which is implied by $E' = (A \in X_A) \wedge (B \in X_B) \wedge (C \in X_C)$, where X_A, X_B, and X_C are subsets of the proper domains, $b_1, b_2 \in X_B$, $a_1, a_2, a_3 \in X_A$, and $c_1, c_2 \in X_C$. Indeed, as it is easy to verify for each rule r, $r^{-E} = \delta$. Therefore in this case all the rules are the part of the answer for the selection query, which can be evaluated directly on the extension of the database (no rules have to be evaluated).

This situation is certainly optimal. Let us now consider a less favorable situation, when

$$E = (B = b_1) \wedge (C = c_1) \vee (B = b_2) \wedge (C \neq c_1).$$

Here we have:

$$r_1^E = \delta,$$

since it is not possible to have both $E(t_1)$ and $E(t_r)$ true for this particular rule and this particular selection condition (t is the antecedent of the rule and t_r is the consequent of the rule). Thus

$$r_1^{-E} = (x_C \neq c_1) \wedge p(x_A, b_1, x_C) \to p(a_A, b_2, x_C).$$

For r_2 we have in a similar way

$$r_2^E = \delta$$

and

$$r_2^{-E} = (x_B = b_2) \wedge p(x_A, x_B, c_1) \rightarrow p(x_A, x_B, c_2).$$

Finally, for r_4 we have that

$$r_4^E = r_4 \quad \text{and} \quad r_4^{-E} = \delta.$$

Therefore the rule r_4 may become an element of the answer for the query, while the rules r_1^{-E} and r_2^{-E} will have to be evaluated prior to the query itself. The other rules of the closure of R_S can be transformed in a similar way. Notice the importance of the computaton of the closure (the rule r_4 is not in the original set of rules).

We now consider PS-queries (queries built from selection and projection). If we are dealing with the query involving projection, we may restrict ourselves to the 1-rules from the closure. For example, $\pi_{BC}(\sigma_E(R))$ for this set of rules can be computed by totally disregarding 2-rules in applying the global transformation rules for PS-expressions and CONST rules. So if E is the selection condition, considered above, which allows the total transformability of the rules, then the projected rules will have the form:

$$p(b_1, x_C) \rightarrow p(b_2, x_C), \tag{1}$$

$$p(x_B, c_1) \rightarrow p(x_B, c_2), \tag{2}$$

and

$$p(b_1, c_1) \rightarrow p(b_2, c_2), \tag{3}$$

which is the modus ponens consequence of the first two rules. On the other hand, for the selection condition E, considered above, only the rule (3) will be transformed:

$$p(b, c_1) \rightarrow p(b_2, c_2).$$

Finally let us consider queries built from join. Assume now that the relation R has four attributes $ABCD$ instead of three ABC. All the rules are changed by adding one more position D to the predicate p and putting the variable x_D into this position. Suppose than that we have the second reltion scheme $T[DEF]$ with the following rules:

$$R_T = \{T(x_D, e_1, x_F) \rightarrow T(x_D, e_2, x_F),$$
$$T(d_1, x_E, x_F) \rightarrow T(d_2, x_E, x_F)\}.$$

Let the query have the form

$$\pi_{BCD}(\sigma_{A \neq a_3}(R) \bowtie \pi_{DE}(T)).$$

From the join rules we immediately obtain that the second rule in R_T is not π_{DE}-transformable (it does change the projection on D). If we apply the CONST transformation rules, we obtain that the only transformable rules are r_1, r_2, and r_3. Moreover the rule r_3 does not have to be evaluated at all, since $E(t_{r_3})$ is false.

Finally then, the rules which are the result of the left subexpression $Q' = \sigma_{BCD}(\sigma_{1 \neq a_3}(R))$ are

$$Q'(b_1, x_C, x_D) \rightarrow Q'(b, x_C, x_D),$$
$$Q'(x_B, c_1, x_D) \rightarrow Q'(x_B, c_2, x_D).$$

The only rule which is join transformable in the right argument of join in the first rule is R_T. The second rule has to be evaluated prior to the join. Both rules about Q' are however join transformable.

According to the join transformation rules we finally obtain the set of the following rules in the answer for this query:

$$Q(b_1, x_C, x_D, x_E) \rightarrow Q(b_2, x_C, x_D, x_E),$$
$$Q(x_B, c_1, x_D, x_E) \rightarrow Q(x_B, c_2, x_D, x_E),$$
$$Q(x_B, x_C, x_D, e_1) \rightarrow Q(x_B, x_C, x_D, e_2),$$

and the rule corresponding to the join operation:

$$Q(x_B, x_C, x_D, u) \wedge Q(y, z, x_D, x_E) \rightarrow Q(x_B, x_C, x_D, x_E).$$

In the top down interpretation this would mean that the program computing the answer for this query would only have to include the second rule in the set R_T; all the other rules could just be included in the answer for the query. They could eventually be evaluated together with the set of answers obtained from the program computing the query. They could also be left in an unevaluated form, as a part of data.

The comments of anonymous referees, including the suggestion of a new title for the paper, are gratefully acknowledged.

REFERENCES

1. Gaifman, NAIL communication, 1986.
2. Gallaire, H. and Minker, J. (eds.), *Logic and Databases*, Plenum, 1978.
3. Hull, R., Finitely Specifiable Implicational Dependency Families, *J. Assoc. Comput. Mach.* 31 (Apr. 1984).
4. Imielinski, T., Functional rules, unpublished.
5. Klug, A. and Price, R., Determining view dependencies using tableaux, *ACM Trans. Database Systems* 7:361–381 (1982).
6. Kowalski, R., *Logic for Problem Solving*, North Holland, New York, 1979.
7. Ullman, J., *Principles of Database Systems*, Computer Science Press, 1982.