Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited

François Bry
ECRC, Arabellastr. 17, 8000 München 81, West Germany
uucp: ...!pyramid!ecrcvax!fb

ABSTRACT *Database applications often require to evaluate queries containing quantifiers or disjunctions, e.g., for handling general integrity constraints. Existing efficient methods for processing quantifiers depart from the relational model as they rely on non-algebraic procedures. Looking at quantified query evaluation from a new angle, we propose an approach to process quantifiers that makes use of relational algebra operators only. Our approach performs in two phases. The first phase normalizes the queries producing a canonical form. This form permits to improve the translation into relational algebra performed during the second phase. The improved translation relies on a new operator - the* complement-join *- that generalizes the set difference, on algebraic expressions of universal quantifiers that avoid the expensive division operator in many cases, and on a special processing of disjunctions by means of* constrained outer-joins. *Our method achieves an efficiency at least comparable with that of previous proposals, better in most cases. Furthermore, it is considerably simpler to implement as it completely relies on relational data structures and operators.*

1. Introduction

Evaluating expressions with quantified variables or disjunctions is often needed for database applications, e.g., for processing queries involving refined relationships between data, or for evaluating sophisticated views, or for handling integrity constraints that are more complex than dependencies. Although defining quantified expressions is usually considered difficult, natural language interfaces to databases considerably help inexperienced users, for example by allowing various grammatical forms that are less stringent than explicit prefixed quantifications. Furthermore with the advent of couplings between databases and artificial intelligence systems, one can expect from users more familiarity with logic - in particular with quantification. The capability of using quantified expressions with ease is often assumed from database users, for instance in the special area of scientific and statistic applications (see, e.g., [CCO 86]). The original definition of SQL [DAT 81] includes both universal and existential quantifiers, but constrains their use (explicit universal quantifications are strongly limited). Some high-level query languages, e.g., DAPLEX [SHI 81] and PASCAL/R [JS 82], permit explicit unrestricted quantifications.

Most research on efficient query evaluation however has focused on quantifier-free conjunctive expressions. The implementations of the most popular relational calculus languages do not permit general quantifications. Quel [ZOO 77] for instance does not have quantifiers. A variable not occurring in the target list is implicitly considered to be existentially quantified. Universal quantification is achieved in Quel by means of an aggregate function: In order to check if all tuples qualified by an expression Q also satisfy a property P, one has to pose a query comparing the numbers of tuples satisfying Q and P, respectively. In QBE [ZLO 77] and SQL [DAT 81], general universal quantifications are in addition expressable by means of set inclusions. These solutions can be criticized for two reasons. First, they are often inefficient because they impose to compute intermediate results - aggregates or relations - that are in principle not needed for answering the universally quantified query. Second, they compromise the declarative character of the query languages and many users consider them more complicated than explicit quantifications.

Since relational calculus is complete with respect to relational algebra [COD 72], any kind of quantified query can in principle be translated into relational algebra. Codd's completeness proof of the calculus defines a reduction algorithm for translating quantified calculus queries into relational algebra expressions. Quantified queries are processed by considering first their prenex disjunctive normal form and the cartesian product of the ranges of all variables occurring in the query. Existential quantifications are then expressed by means of projections; universal quantifications by means of divisions. Despite of its theoretical interest, this approach - even improved as proposed in [PAL 72, JS 82, CG 85] - turns out to be unreasonable for practical use. Because of the initial cartesian product of all ranges and the systematic use of divisions, the relational algebra expressions it generates are extremely inefficient. As shown in [DAY 83], this cartesian product usually retains much more tuples than needed and these tuples are eliminated too late, when divisions are finally performed.

```
evaluate(exists x in R: F(x), value):
value := false
for each x in R while value ≠ true
do      evaluate(F(x), v)
        if v = true then value := true
end
```

(a) Closed (i.e., yes/no) existential queries

```
evaluate(forall x in R: F(x), value):
value := true
for each x in R while value ≠ false
do      evaluate(F(x), v)
        if v = false then value := false
end
```

(b) Closed (i.e., yes/no) universal queries

```
evaluate(x1 in R1: [quantifier x2 in R2: F(x1,x2)], rel):
rel := ∅
for each x1 in R1
do      evaluate(quantifier x2 in R2: F(x1,x2), value)
        if value = true then rel := rel∪{x1}
end
```

(c) Open quantified queries

Fig. 1 Loop algorithms for evaluating quantified queries

A rather immediate alternative to Codd's processing of quantifiers is to process them by means of the loop algorithms of Figure 1, where R represents a range relation for the variable x - this concept is formally defined in Section 2.1 - and F(x) and $F(x_1,x_2)$ represent subqueries with free variables x and x_1, x_2, respectively.

It is important to note the symmetry of the procedures given in Fig. 1a and 1b. Both consist of very similar while loops in which the same subexpressions are evaluated. The algorithms differ in two points only: In the loop halting conditions and in the value finally returned. The truth (the falsity, respectively) of the evaluated subexpression stops the loop in the first procedure (the second procedure, respectively) and establishes the truth (the falsity, respectively) of the existential query (of the universal query, respectively). A logical formalization of this remark (Section 2.1: rewriting rules 4 and 5), and the definition of an operator that generalizes the set difference (Section 3.1: Definition 6) give rise to evaluating universal quantifications similarly to existential ones (Section 3.2: Proposition 4) - up to the above-mentionned symmetry. The efficient methods proposed in [DAY 83, DAY 87], as opposed, rely on special techniques for evaluating universal quantifications.

The algorithms of Fig. 1 process multiple quantifications with nested loop programs, the loop nesting reflecting the quantifier nesting. All operations are pipelined [SC 75, YAO 79] and performed one tuple at a time. Although avoiding temporary storage, this approach is in general inefficient. This is in particular the case when not all relations involved in the query can be simultaneously opened. Since algebraic operations are amenable to pipelining without imposing this technique, nor requiring to perform it on the whole of the query, an efficient quantifier processing method based on relational algebra operations is desirable.

In spite of their drawbacks, the loop algorithms of Fig. 1 have two noticeable properties: Each range relation is searched only once, and no more tuples are accessed than necessary. The main result achieved in this article is the description of a quantifier processing method based on relational algebra operators that has these two attractive properties. The method retains the 'logic' of the nested loop approach without keeping its one-tuple-at-a-time 'control' - in the sense of Kowalski's equation 'algorithm = logic + control' [KOW 79]. Our method can handle all types of quantified queries.

A particularity of our approach is to process queries in two phases. The first phase is a logical normalization of queries into a canonical form. The canonical form is based on concepts - such as the miniscope form - that are unusual in query optimization, though rather classical in logic. The canonical form is important because it gives rise to improving the algebraic translation performed during the second phase. The algebraic translation we describe improves over the classical ones [COD 72, PAL 72, JS 82, CG 85] since it avoids the initial cartesian product, and in most cases the expensive division operator for expressing universal quantifications. Instead, it relies on a new difference operator, that we call complement-join, on a special processing of quantified expressions, and on an improved processing of disjunctions by means of outer-joins.

Other efficient approaches [DAY 83, DAY 87], as opposed, depart from the relational model and rely on non-algebraic procedures. Dayal's first method [DAY 83] requires a considerable machinery and uses unnormalized data structures. Although having the two noticeable properties of the nested loop approach, it imposes to process a quantified query as a whole. Dayal's second method [DAY 87] is significantly simpler than the first one, as it is based on normalized data structures and uses algebraic operators. However, it handles universal quantification with special algorithms. The special translation into relational algebra we describe improves over the efficient processing of Dayal's methods. Furthermore, it is considerably simpler to implement as it completely relies on relational data structures and operators, especially for handling universal quantifications.

Logic based normalizations of quantified queries have already been proposed in [JK 83]. However, the normalizations described there are not related to any improved translation into relational algebra. Instead, a restriction to a special class of queries - called 'perfect queries' - is proposed, in order to ensure acceptable evaluation costs under conventional translations, e.g., [COD 72, PAL 72, JS 82, CG 85]. Although well motivated from an implementation viewpoint, this restriction is unnatural for database users.

Moreover, 'perfect queries' are in our opinion much too restricted for practical use. As opposed, the canonical form we propose combined with the algebraic translation we describe achieve an efficient processing of unrestricted queries.

The article is organized as follows. After this introduction, we define and motivate in Section 2 the canonical form of queries. Section 3 describes the improved translation of canonical form queries into relational algebra. In Section 4, we summarize the main points of the paper. In the remainder of this introductory section, we introduce a few definitions and notations.

Definitions and Notations

We assume that queries are expressed in a relational calculus with domain variables. A selection over an n-ary relation R is therefore represented in a query by means of an atom $R(t_1,...,t_n)$, where the terms t_i are constants or variables. The choice of a formal calculus intends to make the description of the method independent from any actual query language. Domain variables are chosen instead of tuple variables for making easier the reference to logic properties that are traditionally expressed in this manner. Although some of the logical concepts and properties we use are rarely mentioned in query optimization, they are classical in logic (see, e.g., the tutorial [MEN 79]).

Query languages usually have typed variables. We recall that typed quantifications '$\exists$x in R: F(x)' and '$\forall$x in R: F(x)' correspond to '$\exists x\ R(x) \wedge F(x)$' and '$\forall x\ R(x) \Rightarrow F(x)$', respectively, in untyped logic. In order to treat all parts of a query uniformly, we adopt the untyped formalism. In an existential subquery '$\exists x\ R(x) \wedge F(x)$' the outermost conjunction therefore distinguishes the variable range 'R(x)' from the rest of the subquery. Similarly, the implication distinguishes the range 'R(x)' in a universal subquery '$\forall x\ R(x) \Rightarrow F(x)$'. For simplifying the description of the method, the connective => will be used only for expressing ranges. In other contexts an expression '$F_1 \Rightarrow F_2$' is supposed to be written as '$\neg F_1 \vee F_2$', and an expression '$F_1 \Leftrightarrow F_2$' as '$(\neg F_1 \vee F_2) \wedge (\neg F_2 \vee F_1)$'.

Since '$\exists x_1 \exists x_2$ F' and '$\exists x_2 \exists x_1$ F' are logically equivalent whatever formula is F, we allow the shorthand notation '$\exists x_1 x_2 ... x_n$ F' - in which the order of the x_i's is irrelevant - in lieu of '$\exists x_1 \exists x_2 ... \exists x_n$ F'. Similarly '$\forall x_1 x_2 ... x_n$ F' will denote '$\forall x_1 \forall x_2 ... \forall x_n$ F'. Remember however that commuting *distinct* quantifiers in general does not preserve logical equivalence.

A subformula A has *positive polarity* in a formula F if A is embedded in zero or in an even number of negations in F (the left hand side of an implication being considered as an implicit negation). Similarly A has *negative polarity* in F if it is embedded in an odd number of - explicit or implicit - negations in F.

The governing relationship between variables is important for optimizing quantified queries (Section 2.2). We therefore recall its definition. Given a quantification [Qx S] where Q denotes either $\exists$ or $\forall$, the subformula S is called the *scope* of x. A quantified variable x *directly governs* a variable y if the following conditions are satisfied:

1. y is quantified within the scope of x
2. the quantification of y 'follows immediately' that of x (formally: y is not quantified within the scope of a variable quantified in the scope of x)
3. S contains an atom in which both x and y or a variable governed by y occur
4. x and y have distinct quantifiers

The governing relationship is the transitive closure of the above-defined relationship: A quantified variable x *governs* a quantified variable y either if x directly governs y, or if x directly governs a variable z that governs y. Intuitively, x governs y iff moving the quantification of y out of the scope of x could compromise logical equivalence. For example, x governs y but none of the z_i's in the formula:

$\exists x\ \{\text{student}(x) \wedge [\forall y\ \text{lecture}(y,db) \Rightarrow \text{attends}(x,y)] \wedge [\forall z_1\ \text{student}(z_1) \Rightarrow \exists z_2\ \text{attends}(z_1,z_2)]\}$

(there is a student attending all database lectures and each student attends at least one lecture)

2. The Canonical Form of Quantified Queries

This section defines a standardization of queries with quantifiers into a 'canonical form'. This form is motivated by efficiency reasons. The translation into canonical form preserves logical equivalence and produces calculus expressions. It is defined by means of 14 rewriting rules. This technique, stemming from Artificial Intelligence, is particularly well-suited for formalizing standardization processes. Rule defined rewriting systems can easily be compiled into deterministic procedures that are more amenable to practical use. However, the rule formalism is more convenient to simple descriptions.

Section 2.1 defines some concepts of 'range', 'restricted quantification', and 'restricted variables' corresponding to variable declarations in procedural languages. In Section 2.2, we show how to improve evaluations of quantified queries with a 'miniscope form'. In Section 2.3 we argue for keeping disjunctive subqueries in certain cases. Finally, we show in Section 2.4 that the rewriting system defining the canonical form is correctly defined, i.e., in formal terms, it is noetherian and has the Church-Rosser property.

2.1. Formulas with Restricted Variables

In a database, negations are interpreted by failure: Facts not known to be true are assumed to be false. Under this assumption - the Closed World Assumption - the evaluation of non-ground queries with negative polarities is only possible if domains of values are specified for all variables. In order to provide general queries with non-ambiguous semantics, it is assumed that there are no other values than those in the database. By this postulate - the Domain Closure Assumption - the set of all values in the database, called the *database domain*, can be assigned to all variables. A query $\neg p(x_1,...,x_n)$ is in consequence equivalent to $\text{dom}(x_1) \wedge ... \wedge \text{dom}(x_n) \wedge \neg p(x_1,...,x_n)$ where the view 'dom' describes the database domain. If variables in the query language are typed, the type relations can be used instead of 'dom'.

Since negation is interpreted by failure, logically equivalent open queries can be evaluated differently: p(x) for instance

can be used for producing values for x, the evaluation of its logically equivalent form $\neg \neg p(x)$ however does not return values for x. In order to permit a correct evaluation of queries involving nested negations, we use the following classical rewriting rules:

Rule 1: $\neg \neg F \rightarrow F$
Rule 2: $\neg (F_1 \wedge F_2) \rightarrow \neg F_1 \vee \neg F_2$
Rule 3: $\neg (F_1 \vee F_2) \rightarrow \neg F_1 \wedge \neg F_2$

Rules 1, 2 and 3 preserve logical equivalence. Note that they do not transform negated quantifications.

Explicit calls to the database domain or types can be avoided by declaring for each variable the attribute of a relation or of a view as range. The following definition formalizes this notion.

Definition 1
A *range* $R[x_1,...,x_n]$ for variables $x_1,...,x_n$ is recursively defined as follows:

1. $P(x_{\sigma(1)},...,x_{\sigma(n)})$ is a range for $x_1, ..., x_n$ if P is a relation or a view and if σ is a permutation of $\{1,...,n\}$
2. $R_1 \wedge R_2$ is a range for $x_1,..., x_n$ if R_1 is a range for $y_1,..., y_k$, if R_2 is a range for $z_1,...,z_h$ and if $\{y_1,...,y_k\} \cup \{z_1,...,z_h\} = \{x_1,...,x_n\}$
3. $R_1 \vee R_2$ is a range for $x_1,...,x_n$ if R_1 and R_2 are both ranges for $x_1,...,x_n$
4. $R \wedge F$ is a range for $x_1, ..., x_n$ if R is a range for $x_1,...,x_n$ and F is a (possibly quantified) formula with free variables in $\{x_1,...,x_n\}$
5. $\exists y_1...y_p\ R$ is a range for $x_1,...,x_n$ if R is a range for $x_1,...,x_n,y_1,..., y_p$

Existential quantifications in ranges correspond to projections: A range $\exists yz\ p(x,y,z)$ for a variable x expresses that x takes its values from the first projection of p. Classical ranges are those defined only by Conditions 1 and 5, i.e., corresponding to projections of a single relation or view. The more general definition given here may be seen as allowing view definitions local to a query.

Closed formulas with restricted quantifications are the counterparts in logic to yes/no queries in conventional query languages:

Definition 2
A formula is a *closed formula with restricted quantifications* if all its quantified subformulas SF have one of the forms $\exists x_1...x_n\ R[x_1,...,x_n]$, $\exists x_1...x_n\ R[x_1,...,x_n] \wedge F$, $\forall x_1...x_n \neg R[x_1,...,x_n]$, or $\forall x_1...x_n\ R[x_1,...,x_n] \Rightarrow F$ where $R[x_1,...,x_n]$ is a range for $x_1, ..., x_n$, and if variables free in SF are quantified outside SF.

Queries like F_1: $\exists x_1x_2\ [r(x_1) \vee s(x_2)] \wedge \neg p(x_1,x_2)$ are rejected by Definition 2. F_1 is unacceptable since certain evaluations of $[r(x_1) \vee s(x_2)]$ - such as $x_1 = a$, if r(a) holds - leads to unbound variables occurring in the negated subquery $\neg p(x_1,x_2)$: The expression $[r(x_1) \vee s(x_2)]$ is not a range for x_1 and x_2.

Consider a closed existential formula Q_1: $\exists x_1...x_n\ R[x_1,...,x_n] \wedge F_1$. An evaluation of Q_1 that minimizes the number of tuples accessed and the number of tuple comparisons performed can be informally described as follows. First, the open formula $R[x_1,...,x_n]$ is evaluated and for each returned binding σ of $x_1,...,x_n$, the closed expression $F_1\sigma$ is in turn evaluated. This evaluation can terminate as soon as a binding σ is found such that $F_1\sigma$ evaluates to true, Q_1 is known to be true. If no such binding is found, Q_1 is false. In the rest of the paper, an evaluation method is described that follows this evaluation scheme but permits other controls than the one-tuple-at-a-time strategy of the loop algorithms mentioned in the introduction.

The evaluation of a closed universal formula with restricted quantifications Q_2: $\forall x_1...x_n\ R[x_1,...,x_n] \Rightarrow F_2$ is very similar. First, the range $R[x_1,...,x_n]$ is evaluated. For each returned variable binding σ, $F_2\sigma$ is evaluated. Q_2 is known to be false if and only if there is a binding σ such that $F_2\sigma$ evaluates to false. Up to the truth value finally returned, this evaluation process is identical with that of $\exists x_1...x_n\ R[x_1,...,x_n] \wedge \neg F_2$. In other words, the logically equivalent formulas Q_2 and $\neg (\exists x_1...x_n\ R[x_1,...,x_n] \wedge \neg F_2)$ are evaluated in the same manner. Based on this observation, we introduce two rewriting rules that reduce the evaluation of universal expressions to that of existential ones:

Rule 4: $\forall x_1...x_n\ R[x_1,...,x_n] \Rightarrow F \rightarrow \neg (\exists x_1...x_n\ R[x_1,...,x_n] \wedge \neg F)$
Rule 5: $\forall x_1...x_n \neg R[x_1,...,x_n] \rightarrow \neg (\exists x_1...x_n\ R[x_1,...,x_n])$

Rules 4 and 5 preserve logical equivalence. This reduction of universal expressions to existential ones requires to be capable to evaluate existential formulas in which negations occur. An efficient translation of such formulas into relational algebra is proposed in Section 3.

If the existential closure of an open query is a formula with restricted quantifications, then the query can be evaluated similarly to existential formulas. Let Q_3: $R[x_1...x_n] \wedge F_3$ be an open query such that its existential closure $\exists x_1...x_n\ Q_3$ is a closed formula with restricted quantifications. The evaluation of the range $R[x_1...x_n]$ returns bindings for the x_i's. For each such binding σ, the expression $F_3\sigma$ is evaluated. If it evaluates to true, the σ defines an answer to Q_3. As opposed to closed formulas, all solutions to $R[x_1,...,x_n]$ have to be computed, in order to determine all answers to Q_3. Definition 3 formalizes the concept of open query.

Definition 3
Open formulas with *restricted variables* $x_1,...,x_n$ are recursively defined as follows:

1. F is an open formula with restricted variables $x_1,...,x_n$ if every x_i is free in F, if F has no other free variables and if $\exists x_1...x_n\ F(x_1,...,x_n)$ is a closed formula with restricted quantifications
2. $F_1 \vee F_2$ is an open formula with restricted variables $x_1,...,x_n$ if both F_1 and F_2 are open formulas with restricted variables $x_1,...,x_n$

Formulas with restricted variables and quantifications are definite [KUH 67] and domain independent [FAG 80], two equivalent properties characterizing the formulas whose valuations remain unchanged under updates on relations not occurring in the formulas. As opposed to domain independence, the restricted variable and quantification properties are defined syntactically. Other syntactical classes of domain independent formulas that have been proposed are, e.g., the range-restricted formulas [ND 83] and the various classes of allowed formulas [LT 86, VGT 87, SHE 88]. For each expression in one of these classes, it is possible to construct an equivalent formula with restricted variables and quantifications [BRY 89].

Well-formed formulas may contain 'useless' quantifications, i.e. quantifications applying to variables that do not occur in the rest of the formula. This is for example the case of the first quantification in the formula F: $\exists x\ [\forall y\ p(y) \Rightarrow q(y)]$. There are no chances to find ranges for variables with such quantifications! We therefore introduce two rewriting rules for avoiding these cases:

Rule 6: $\exists x_1...x_n \rightarrow F$
if none of the x_i's occur in F

Rule 7: $\exists x_1...x_n \rightarrow \exists x_{i_1}...x_{i_p}\ F$
if the x_i's distinct from $x_{i_1}...x_{i_p}$ do not occur in F

Rule 6 preserves logical equivalence. Rule 7 also preserves logical equivalence since $\exists x_1...x_n\ F$ is logically equivalent to $\exists x_{\sigma(1)}...x_{\sigma(n)}\ F$, for all permutations σ of {1,...,n} (we recall that $\exists x_1...x_n\ F$ denotes $\exists x_1 \exists x_2...\exists x_n\ F$).

2.2. The Miniscope Form

The methods proposed in the literature for evaluating quantified queries usually first compute their prenex form - obtained by moving the quantifiers in front of queries [COD 72, PAL 72, JS 82, DAY 83, CG 85, DAY 87]. Here we plead for a different syntactical form of queries, the miniscope form. Informally, this form is obtained by pushing all quantifiers inwards, reducing variable scopes as much as possible.

We first motivate the miniscope form with an example. Consider the query

Q_1: $\exists x$ student(x) $\wedge$ $\forall y$ [cs-lecture(y) => attends(x,y) $\wedge$ $\neg$ enrolled(x,cs)]

asking if one may find a student attending all lectures in computer science without being enrolled in this department. If Q_1 is evaluated as described above, for a given student 'a' the subquery $\neg$ enrolled(a,cs) is evaluated as many times as there are computer science lectures. This is of course neither necessary nor desirable. It has been proposed in [JK 83] to rely on more sophisticated query evaluation procedures for avoiding these redundant evalutions. A much simpler - and cheaper - solution consists in first transforming Q_1 into the equivalent formula

Q_2: $\exists x$ student(x) $\wedge$ [$\forall y$ cs-lecture(y) => attends(x,y)] $\wedge$ $\neg$ enrolled(x,cs)

and then to process Q_2 as described in the previous section. During the evaluation of Q_2, the subquery $\neg$ enrolled(x,cs) is evaluated only once for each possible student x because the variable x is no more in the scope of the universal variable y. Definition 4 formalizes this remark.

Definition 4
A formula is in *miniscope form* if and only if none of its quantified subformulas F contains an atom in which only variables quantified outside F occur.

A miniscope form is often obtained by simply moving subformulas out of the scopes of quantifiers according to the following rules, where θ denotes $\wedge$ or $\vee$.

Rule 8: $\exists x_1...x_n\ [F_1\ \theta\ F_2] \rightarrow F_1\ \theta\ [\exists x_1...x_2\ F_2]$
if none of the variables $x_1,...,x_n$ occur in F_1

Rule 9: $\exists x_1...x_n\ [F_1\ \theta\ F_2] \rightarrow [\exists x_1...x_n\ F_1]\ \theta\ F_2$
if none of the variables $x_1,...,x_n$ occur in F_2

These rules preserve logical equivalence. In certain cases Rules 8 and 9 are not sufficient for reaching a formula in miniscope form. Consider for example the formula:

F_1: $\exists x\ p(x) \wedge (q(y) \vee r(x))$

In order to move the atomic subformula q(y) out of the scope of the existential quantifier, one has first to move it out of the disjunction. This is done by first putting the matrix of F_1 in disjunctive normal form:

F_2: $\exists x\ [p(x) \wedge q(y)] \vee [p(x) \wedge r(x)]$

Then, since existential quantifiers distribute over disjunctions, F_2 is transformed into:

F_3: $(\exists x_1\ [p(x_1) \wedge q(y)]) \vee (\exists x_2\ [p(x_2) \wedge r(x_2)])$

Finally, applying Rule 5 results in:

F_4: $([\exists x_1\ p(x_1)] \wedge q(y)) \vee (\exists x_2\ [p(x_2) \wedge r(x_2)])$

Both disjuncts D_1: $[\exists x_1\ p(x_1)] \wedge q(y)$ and D_2: $\exists x_2\ [p(x_2) \wedge r(x_2)]$ of F_4 share a common subexpression, namely C: $\exists x\ p(x)$. It is therefore desirable that both evaluations of D_1 and D_2 use the evaluation of C as a sharable resource. To this aim, it is tempting to look for a rewriting of F_4 in which C occurs only once. Unfortunately, this results in a non-miniscope expression similar to the initial query F_1!

It is worth noting that the two requirements, miniscope form and sharing of common subexpressions, cannot always be reached by a unique syntactical form of the queries. However, answers to common subexpressions that are not syntactically shared in the query can be shared procedurally by refined evaluation procedures. Such methods have been investigated, e.g., in [SEL 86]. The following rules together with Rules 8 and 9 yield formulas in miniscope form.

Rule 10: $\exists x_1...x_n\,(F_1 \vee F_2) \wedge F_3 \rightarrow [\exists x_1...x_n\,(F_1 \wedge F_3)] \vee [\exists x_i...x_n\,(F_2 \wedge F_3)]$ if (†) F_1 or F_2 contains an atomic subformula in which none of the x_i's and none of the variables governed by some x_i occur

Rule 11: $\exists x_1...x_n\,F_1 \wedge (F_2 \vee F_3) \rightarrow [\exists x_1...x_n\,(F_1 \wedge F_2)] \vee [\exists x_i...x_n\,(F_1 \wedge F_3)]$ if (†) holds

Rules 10 and 11 preserve logical equivalence. They cannot be applied if F_1 or F_2 simply contains an atomic subformula in which none of the x_i's occur. The governing relationships between variables must be taken into account. Consider for example F_5: $\exists x\ p(x) \wedge [\forall y \neg q(y) \vee r(x,y)]$. Moving q(y) out of the universal quantification would not preserve logical equivalence. In fact, F_5 is in miniscope form.

2.3. Producers and Filters

It is generally considered beneficial to break queries into conjunctive subqueries in order to avoid - or to postpone as long as possible - the computation of unions. Such an approach permits indeed to delay the creation of intermediate results of larger size. The disjunctive normal form is in consequence usually considered as the most appropriate syntactical form of queries for an efficient evaluation. We propose to distribute conjunctions over disjunctions only in certain cases, namely when the superexpressions containing the disjunctions are all 'producers'. However, we advocate to keep those disjunctions that occurr in subexpressions we call 'filters'. As we show in Section 3.3, this approach permits to improve over the traditional ones, especially for evaluating range expressions.

Let us first informally introduce in an example the concepts of producer and filter. Consider a query:

Q_1: $\exists x$ [(student(x) $\wedge$ makes(x,PhD)) $\vee$ prof(x)] $\wedge$ [speaks(x,french) $\vee$ speaks(x,german)]

asking if there is a PhD student or a professor that speaks french or german. Evaluating the range R: [(student(x) $\wedge$ makes(x,PhD)) $\vee$ prof(x)] 'produces' bindings for the variable x. These bindings are then tested with the subexpression E: speaks(x,french) $\vee$ speaks(x,german). All variables occurring in E also occur in R: E therefore does not 'produce' variable bindings but 'filters' the values returned by the evaluation of R. The ranges in a query with constrained variables are producers. The expressions outside ranges are filters. As shown at the end of this paragraph, it is beneficial not to identify the concepts of range and of producer.

Definition 5

Given a conjunctive expression (P $\wedge$ F) (F $\wedge$ P, resp.) with free variables x_1, ..., x_n, P is a *producer* and F is a *filter* if P is a range for the x_i's. A filter which is a disjunction of subformulas is called a *disjunctive filter.*

In certain cases, both arguments of a conjunction may be considered as producers or as filters, respectively. Deciding which argument will be considered first is a major concern in query optimization. This is usually done by comparing the (estimated) costs of the various solutions. In order to estimate these costs or performances, one first has to know how a given ordering is processed. Such a processing is proposed below. However, no choice strategy is described here: Such a strategy requires the definition of a cost model - an issue out of the scope of this article.

Consider again the query Q_1 defined at the beginning of this section. Distributing the quantification and the range of Q_1 over the disjunction occurring in the filter (speaks(x,french) $\vee$ speaks(x,german)) yields the following disjunctive query:

Q_2: ($\exists x_1$ [(student(x_1) $\wedge$ makes(x_1,PhD)) $\vee$ prof(x_1)] $\wedge$ speaks(x_1,french)) $\vee$ ($\exists x_2$ [(student(x_2) $\wedge$ makes(x_2,PhD)) $\vee$ prof(x_2)] $\wedge$ speaks(x_2,german)).

A direct evaluation of Q_2 would independently search on the one hand for a PhD student or professor speaking french, on the other hand for a PhD student or professor speaking german. The set of individuals qualified by the range is therefore searched twice. Keeping the disjunction in the filter subexpression might permit a more efficient evaluation. It is shown in Section 3 how disjunctions occurring in filters can be expressed by means of outer-joins instead of unions. Intuitively, disjunctions occurring in filters do not require to store intermediate relations, since filters do not produce new tuples.

In the general case of a query $\exists x_1...x_n\ R[x_1,...,x_n] \wedge F$, F may contain quantified subformulas. This however does not prevent from keeping disjunctions in the filter F. A quantified formula, such as for example [$\forall$y roman-language(y) => speaks(x,y)] *(x speaks all roman languages)* expresses a property of the individual x rather similar to the quantifier-free property speaks(x,french).

Distributing the existential quantifier and the filter (speaks(x,french) $\vee$ speaks(x,german)) over the disjunction occurring in the producer R of Q_2 yields the equivalent query:

Q_3: $\exists x_1$ (student(x_1) $\wedge$ makes(x_1,PhD)) $\wedge$ (speaks(x_1,french) $\vee$ speaks(x_1,german)) $\vee$ $\exists x_2$ professor(x_2) $\wedge$ (speaks(x_2,french) $\vee$ speaks(x_2,german))

Evaluating Q_3 consists in independently searching on the one hand for a PhD student, on the other hand for a professor that speaks french or german. The form Q_3 of the query appears to be preferable as it permits not to compute the union of the relations of professors and of PhD students.

It is in certain cases preferable to keep disjunctions occurring in ranges. Consider, e.g., a query

Q_4: $\exists x$ [professor(x) $\wedge$ (member(x,cs) $\vee$ skill(x,math))] $\wedge$ speaks(x,french)

asking if there is a professor who speaks french, in the computer science department or with a skill in mathematics. Moving the disjunction out of the range gives the formula:

Q_5: $\exists x_1$ [professor(x_1) $\wedge$ member(x_1,cs)] $\wedge$ speaks(x_1,french) $\vee$ $\exists x_2$ [professor(x_2) $\wedge$ skill(x_2,math)]

$\wedge$ speaks(x_2,french)

A direct evaluation of Q_5 would require to search the professor relation twice. Sharing the search of this relation for the evaluation of both subqueries is a rather natural optimization. Such a sharing is precisely expressed in the compacted original form Q_4 of the query. This sharing is possible by considering professor(x) as a producer and [member(x,cs) $\vee$ skill(x,math)] as a filter in the range [professor(x) $\wedge$ (member(x,cs) $\vee$ skill(x,math))].

The desirable transformation consists therefore in 'moving out' disjunctions that are not contained in filter subexpressions, and only those disjunctions. It is defined by the following rules:

Rule 12: $(P_1 \vee P_2) \wedge F \rightarrow (P_1 \wedge F) \vee (P_2 \wedge F)$
if $(P_1 \vee P_2) \wedge F$ occurs in a range and if $(P_1 \vee P_2)$ is not a filter

Rule 13: $F \wedge (P_1 \vee P_2) \rightarrow (F \wedge P_1) \vee (F \wedge P_2)$
if $F \wedge (P_1 \vee P_2)$ occurs in a range and if $(P_1 \vee P_2)$ is not a filter

Rule 14: $\exists x_1 ... x_n (R1 \vee R2) \rightarrow$
$(\exists x_{j_1} ... x_{j_p} R_1) \vee (\exists x_{k_1} ... x_{k_q} R_2)$
if the x_j's (x_k's, resp.) are the x_i's occurring in R_1 (R_2, resp.)

Rules 12 and 13 express classical equivalence preserving transformations. Rule 14 preserves logical equivalence because logical quantifers distribute over disjunctions and for reasons similar to the motivation of Rule 7.

2.4. Correctness of the Rewriting System

In the previous paragraph, we have proposed to normalize queries into a canonical form. We have defined this normalization by 14 rewriting rules. We establish in the present paragraph the correctness of the rewriting system. We then show that the canonical form of a query is unique, up to the choice of the producers. More formally, we prove that the translation process defined by the rule system stops in all cases - the rewriting system is *noetherian* - and we show that the final result of a translation does not depend on the application order of rules - the rewriting system is *confluent* or has the Church-Rosser property.

Proposition 1 The rewriting system consisting of Rules 1 to 14 is noetherian.

[*Proof:* It is sufficient to remark that the number of times a given rule migth be applied during a translation process is bounded by a parameter that depends only on the considered rule and on the formula to translate. Rules 4 and 5, for example, are applicable at most as many times as there are universal quantifiers in the formula.]

Proposition 2 The rewriting system S consisting of Rules 1 to 14 is confluent.

[*Proof:* (sketched) Let us informally recall some concepts and refer to [SCH 87, HUE 80] for formal definitions. Two subformulas SF_1 and SF_2 form a 'critical pair' if there is a formula F and two distinct rewriting rules both applicable on F trough the subformulas SF_1 and SF_2, respectively. A normal form of a formula F is a final translation of F. Since the system S is noetherian, each formula has at least one normal form. Since S is noetherian, as shown in [HUE 80] it suffices to prove that for all critical pairs (SF_1,SF_2) and for the corresponding normal forms NF_1, NF_2 of a formula F we have $NF_1 = NF_2$ in order to establish Proposition 2. S being finite, there are finitely many critical pairs (SF_1,SF_2), that can be successively checked for the required property. Consider for example the pair ($\exists x_1 ... x_n F_1 \theta F_2$, $F_1 \theta F_2$), where F_1 does not contain any x_i, and where a variable x_{i_0} does not occur in F_2. Applying Rule 7 first (for removing the useless variable x_{i_0} from the quantification) results in $\exists x_1 ... x_{i_0-1} x_{i_0+1} ... x_n F_1 \theta F_2$. Applying Rule 5 on this expression (for moving F_1 out of the scope of the quantifier) yields $F_1 \theta \exists x_1 ... x_{i_0-1} x_{i_0+1} ... x_n F_2$. Applying Rule 5 first gives $F_1 \theta \exists x_1 ... x_n F_2$. The same normal form as before, namely $F_1 \theta \exists x_1 ... x_{i_0-1} x_{i_0+1} ... x_n F_2$, is obtained by applying now Rule 4. The reasoning is similar for the other critical pairs.]

3. Translation into Relational Algebra

Since queries in canonical form have restricted variables, the translation of canonical queries could be done in a conventional manner - existential quantifiers being processed by projections, universal quantifier by divisions, disjunctions by unions, conjunctions by joins and differences - as proposed in [COD 72, PAL 72, JS 82, CG 85]. In this section, we describe a more efficient translation.

We define in Section 3.1 a new operator, the complement-join, that generalizes the set difference. We show that the complement-join improves significantly the processing of certain conjunctions. In Section 3.2, we propose to add an emptiness test to relational algebra. This test permits a more faithful translation - and therefore a better processing - of closed existential queries. We then describe a translation of quantified queries into relational algebra that considerably restricts the use of the expensive division operator. Furthermore, this new translation does not systematically rely on the cartesian product of all variable ranges for expressing nested quantifications. We then propose in Section 3.3 a special processing of disjunctive filters by means of outer-joins. This processing of disjunctions is especially useful for quantified queries. It is of course also applicable to quantifier-free expressions.

3.1. The Complement-Join Operator

Certain conjunctions are inefficiently processed under the classical translations into relational algebra. In order to remedy to this undesirable situation, we propose to extend the relational algebra with a new operator, the complement-join. We first introduce informally the complement-join on an example.

Consider an open query Q_1 qualifying the members of some departments that have no skill in databases:

Q_1: $\exists z$ member(x,z) $\wedge \neg$skill(x,db)

The usual translation of Q_1 into relational algebra is π_1(member) − $\pi_1[\sigma_{2="db"}$(skill)]. In order to also get the names of the departments, one has to modify Q_1 into:

Q_2: member(x,z) $\wedge \neg$skill(x,db)

Though Q_1 and Q_2 are very similar, the conventional translation of Q_2 into relational algebra differs significantly from that of Q_1. Indeed, since the variable z does not occur in the second conjunct of Q_2, Q_2 cannot be directly translated with a difference. A conventional translation of Q_2 is:

$$\text{member} \underset{1=1}{\bowtie} (\pi_1(\text{member}) - \pi_1[\sigma_{2="db"}(\text{skill})])$$

This algebraic expression is more expensive to evaluate than that associated with Q_1 because it requires to compute not only a difference, but also a join.

The conjunction in Q_2 could however be evaluated similarly to a semi-join. For each tuple (m_i,d_i) in 'member', the relation 'skill' could be searched for a tuple $(m_i,"db")$. If such a tuple is found - i.e., if $(m_i,d_i) \in$ member $\underset{1=1}{\ltimes} \pi_1[\sigma_{2="db"}(\text{skill})]$ - then (m_i,d_i) is not an answer to Q_2. Otherwise, it is an answer to Q_2. In other terms, Q_2 corresponds to the complement of the relation 'member $\underset{1=1}{\ltimes} \pi_1[\sigma_{2="db"}(\text{skill})]$' in the relation 'member'. Like a semi-join, the complement-join of two relations is a subset of its first argument.

The following definition of the complement-join operator is based on the preceding observation.

Definition 6

Let P and Q be two relations with arities p and q, respectively. Let 'conj' be a conjunction of equalities i=j where $1 \le i \le p$ and $1 \le j \le q$.

The *complement-join* $P \underset{conj}{\overline{\ltimes}} Q$ is the p-ary relation defined as:

$$\{(c_1,...,c_p) \mid (c_1,...,c_p) \in P - \pi_{1...p}(P \underset{conj}{\bowtie} Q)\}$$

Using the complement-join, the example query Q_2 considered above is expressed as:

$$\text{member} \underset{1=1}{\overline{\ltimes}} \pi_1[\sigma_{2="db"}(\text{skill})]$$

The complement-join is easily implemented by modifying any semi-join algorithm, thus permitting an algebraic processing of queries like Q_2 much more efficient than the conventional one.

The following proposition motivates the name 'complement-join' and shows that this new operator generalizes the set difference.

Proposition 3

Let P and Q be two relations with arities p and q, respectively. Let 'conj' be a conjunction of equalities i=j, where $1 \le i \le p$ and $1 \le j \le q$.

The following equalities hold:

$$P = \pi_{1...p}(P \underset{conj}{\bowtie} Q) \cup [P \underset{conj}{\overline{\ltimes}} Q]$$

$$\emptyset = \pi_{1...p}(P \underset{conj}{\bowtie} Q) \cap [P \underset{conj}{\overline{\ltimes}} Q]$$

If p = q, then the following equality holds:

$$P - Q = P \underset{1=1\wedge...\wedge p=q}{\overline{\ltimes}} Q$$

[*Proof:* The equalities are immediate consequences of Definition 6.]

The similarity as well the difference between the semi-join and the complement-join can also be illustrated by the following equalities, that are directly implied by the definitions:

$$R \underset{1=1}{\ltimes} S = \{x \mid R(x) \wedge \exists y\, S(x,y)\}$$

$$R \underset{1=1}{\overline{\ltimes}} S = \{x \mid R(x) \wedge \neg \exists y\, S(x,y)\}$$

where R and S denote a unary and a binary relation, respectively.

3.2. Processing Quantifiers

The query standardization described in Section 2 reduces universal quantifications to existential ones. It is therefore sufficient to describe how existential expressions are translated into relational algebra.

A closed existential query Q: $\exists x_1...x_n\, F(x_1,...,x_n)$ is equivalent to the equation $\{\vec{x} \mid F(x_1,...,x_n)\} \neq \emptyset$. Provided the quantification in Q is restricted, i.e., $F(x_1,...,x_n) = R[x_1,...,x_n] \wedge G$ where $R[x_1,...,x_n]$ is a range for the x_i's, the set S: $\{\vec{x} \mid F(x_1,...,x_n)\}$ is definable by an algebraic expression. It is indeed not necessary to compute the whole set S for evaluating Q. It is therefore desirable to extend the relational algebra with a non-emptiness test. Allowing tests in algebraic expressions leads to allow boolean connectives as well. This is indeed needed for queries consisting of conjunctions or disjunctions of closed formulas. Consider for example the query:

$\exists x$ (student(x) $\wedge$ [$\forall$y lecture(y,db) => attends(x,y)])
$\wedge$ [$\forall$ z_1 student(z_1) => $\exists z_2$ attends(z_1,z_2)]

asking if there is a student attending all database lectures and if each student attends at least one lecture. It corresponds to the boolean expression:

{x | student(x) $\wedge$ [$\forall$y lecture(y,db) => attends(x,y)]} $\neq \emptyset$
$\wedge$ {z_1 | student(z_1) $\wedge \exists z_2$ attends(z_1,z_2)} = $\emptyset$

Pipelined evaluations [SC 75, YAO 79] are particularly convenient for performing such tests. Consider the query:

Q: $\exists$xy [enrolled(x,y) $\wedge$ y $\neq$ cs $\wedge$ makes(x,PhD)
$\wedge \exists z$ (lecture(z,cs) $\wedge$ attends(x,z))]

asking if there is a PhD student who is enrolled in another department than the computer science department and attends a lecture in computer science. A conventional evaluation of Q consists for instance in computing both sets:

S_1: $\pi_1[\sigma_{2\neq"cs"}(\text{enrolled}) \underset{1=1}{\bowtie} \sigma_{2="PhD"}(\text{makes})]$

S_2: $\pi_2[\sigma_{2="cs"}(\text{lecture}) \underset{1=2}{\bowtie} \text{attends}]$

and in performing the join $S_1 \underset{1=1}{\bowtie} S_2$. Such a processing in general performs much more tuple comparisons than a pipelined evaluation of Q.

Pipelining the evaluation of Q consists in fact in applying the loop algorithms of Fig. 1 (Section 1). With this approach, the relation 'enrolled' is first searched for a tuple (x,y) such that y ≠ "cs". As soon as such a tuple is found, it is checked if makes(x,PhD) holds or not. If it does hold, the relation 'lecture' is searched for a tuple (z,cs). As soon as such a tuple is found, it is checked if the tuple (x,z) is in the relation 'attends'. In case of failure, the next tuple of the last relation is considered. When convenient values for x, y and z are found, Q is known to be true: There is no need to pursue the search for other values. This one-tuple-at-a-time evaluation therefore minimizes the number of tuple comparisons. It is however often inefficient, in particular when the relations mentioned in the query cannot be altogether simultaneously opened. It is generally preferable not to pipeline the evaluation of a query as a whole, but to pipeline the evaluation of subexpressions only. A faithful translation of closed existential queries with non-emptiness equations gives rise to save useless computations.

The following proposition shows how to translate open calculus expressions with nested quantifications into relational algebra. It is worth noting that, among four syntactical forms, only one case makes use of the expensive division operator. Translations according to Proposition 4 are therefore more efficient than the conventional ones described, e.g., in [COD 72, PAL 72, JS 82, CG 85].

Proposition 4
Let R and T be two binary relations, and let S and G be two ternary relations. $Q_c \equiv Q_a$ denotes that the calculus query Q_c is equivalent to the algebraic query Q_a.

1. $\exists y\, R(x,y) \wedge \exists z\, [S(x,y,z) \wedge G(x,y,z)]$
   $\equiv \pi_1[R \underset{1=1\wedge 2=2}{\bowtie} \pi_{12}(S \underset{1=1\wedge 2=2\wedge 3=3}{\bowtie} G)]$

2a. $\exists y\, R(x,y) \wedge \exists z\, [S(x,y,z) \wedge \neg G(x,y,z)]$
   $\equiv \pi_1[R \underset{1=1\wedge 2=2}{\bowtie} \pi_{12}(S \underset{1=1\wedge 2=2\wedge 3=3}{\overline{\ltimes}} G)]$

2b. $\exists y\, R(x,y) \wedge \exists z\, [T(y,z) \wedge \neg G(x,y,z)]$
   $\equiv \pi_1[R \underset{1=2}{\bowtie} T] \underset{1=1\wedge 2=2\wedge 3=3}{\overline{\ltimes}} G$

3. $\exists y\, R(x,y) \wedge \neg (\exists z\, [S(x,y,z) \wedge G(x,y,z)])$
   $\equiv \pi_1[R \underset{1=1\wedge 2=2}{\overline{\ltimes}} \pi_{12}(S \underset{1=1\wedge 2=2\wedge 3=3}{\bowtie} G)]$

4. $\exists y\, R(x,y) \wedge \neg (\exists z\, [S(x,y,z) \wedge \neg G(x,y,z)])$
   $\equiv \pi_1[R \underset{1=1\wedge 2=2}{\overline{\ltimes}} \pi_{12}(S \underset{1=1\wedge 2=2\wedge 3=3}{\overline{\ltimes}} G)]$

5. $\exists y\, R(x,y) \wedge \neg (\exists z\, [T(y,z) \wedge \neg G(x,y,z)])$
   $\equiv \pi_1[R \underset{1=1\wedge 2=2}{\overline{\ltimes}} \pi_{12}(G \underset{3=1}{\div} \pi_3(T))]$

[*Proof:* The first five equivalences are immediate consequences of the definitions of the join and complement-join operators. The last one holds as well since:

a. $\neg (\exists z\, [T(y,z) \wedge \neg G(x,y,z)])$ is logically equivalent to $(\forall z\, [T(y,z) \Rightarrow G(x,y,z)])$

b. $G \underset{3=1}{\div} \pi_2(T) = \{(x,y) \mid (x,y) \in \pi_{12}(G) \wedge \forall z\, [z \in \pi_2(T) \Rightarrow (x,y,z) \in G]\}$
by definition of the division operator.]

In the fifth case, the division operator cannot be avoided, except rewritten in terms of difference or complement-join. Since the variable x does not occur in T(y,z), but only in G(x,y,z), a translation similar to that of case 4 is impossible. The impossibility to translate all quantified calculus expressions into relational algebra without using either division or difference was observed by Codd [COD 72]. It has motivated the introduction of the division operator into relational algebra.

Proposition 4 extends easily to more general expressions, e.g. with more than one free variable, or such that R, S, T or G denote complex expressions, or such that x or y does not occur in both S and G.

It is worth noting that this is because of the miniscope form, that cartesian products and divisions can be avoided. Instead of the miniscope form, the classical methods [COD 72, PAL 72, JS 82, CG 85] consider calculus queries in prenex form: An initial cartesian product of all variable ranges and a systematic translation of universal quantifications - or negated existential quantifications - into divisions are in consequence necessary. The methods [DAY 83, DAY 87] also consider queries in prenex form. Although not explicitly performing cartesian products and divisions, the method [DAY 83] rely on non-relational procedures that perform similary to these operators. The method [DAY 87] relies on cartesian products. It handles quantifiers in ways similar to that of [DAY 83].

3.3. Processing Disjunctive Filters with Outer-Joins

Let us first consider how a disjunctive filter is evaluated under a pipelining strategy. Consider for example the open query Q_1: $P(x) \wedge [T(x) \vee U(x)]$. According to Definition 5, P(x) can be considered as a producer and $[T(x) \vee U(x)]$ as a filter. In this case, the evaluation of P(x) produces values for x that are filtered through the expression $[T(x) \vee U(x)]$. A pipelined evaluation of Q_1 consists in successively checking for each P-tuple, first if it occurs in T, second if it occurs in U.

This evaluation strategy presents three main advantages over conventional algebraic approaches. First, the set $T \cup U$ is not constructed. Second, the relation P is searched only once. Third, it is possible not to search U for those tuples that are in T (or conversely). However, a major drawback of the approach is that interleaved access to all relations occurring in a disjunction are needed. This may introduce a considerable overhead in secondary storage access. In this paragraph, it is shown how a special processing of disjunctive filters with outer-joins retains the three above-mentioned advantages without imposing to simul-

taneously access all considered relations. In order to illustrate the approach on an example consider the relations P, T, and U of Fig. 2.

The unidirectional outer-join [LP 76] R_1: $P \overset{}{\underset{1=1}{\ltimes\!\bowtie}} T$ (Fig. 2) permits to recognize the P-tuples - like (a) - that are in T, without losing the P-tuples - like (c) - that are not in T. The former tuples correspond to R_1-tuples with non-null second attribute, the latter to R_1-tuples with null second attribute. The null symbol ∅ serves only internal purposes: It is not available in the user language.

P	T	U
a	a	a
b	b	c
c	e	f
d		

R_1: $P \underset{1=1}{\ltimes\!\bowtie} T$

a	a
b	b
c	∅
d	∅

Fig. 2

Since the outer-join R_1 'preserves' its left operand - $P = \pi_1(R_1)$ - the relation R_2: $R_1 \underset{1=1}{\ltimes\!\bowtie} U$ (Fig. 3) permits to distinguish the P-tuples occurring in U. A R_2-tuple with non-null third argument corresponds to a tuple in both P and U. The P-tuples occurring in at least one of T and U are the R_2-tuples with at most one null attribute. In other terms, the query:

$$Q_1: P(x) \wedge (T(x) \vee U(x))$$

can be translated into the algebraic expression:

$$\pi_1(\sigma_{2\neq\varnothing \vee 3\neq\varnothing}([P \underset{1=1}{\ltimes\!\bowtie} T] \underset{1=1}{\ltimes\!\bowtie} U))$$

R_2: $[P \underset{1=1}{\ltimes\!\bowtie} T] \underset{1=1}{\ltimes\!\bowtie} U$

a	a	a
b	b	∅
c	∅	c
d	∅	∅

Fig. 3

This translation presents the first two attractive properties of a pipelined evaluation: The union P ∪ T is not constructed and P is searched only once. However, it does not satisfy the third property: For computing R_2 the relation U is also searched for tuples occurring in T - as reflected by the presence of (a,a,a) in R_2. The useless search can be avoided by constraining the second outer-join with the condition 2 = ∅. Under this constraint, a tuple of R_1 with non-null second argument - like (a,a) - is not compared with U-tuples, but yields an R_2-tuple with null third attribute.

Fig. 3 shows an undesirable redundancy: Since P is a producer and T participates to a filter, there is no need to register the value 'b' as second argument in the outer-join $P \underset{1=1}{\ltimes\!\bowtie} T$. Instead, it would be sufficient to 'mark' with a special symbol, say ⊥, that 'b' has been found in T. The symbol ⊥ is somehow a counterpart to ∅. Like ∅, ⊥ is not available in the user language.

Definition 7 formalizes the concept of constrained outer-join.

Definition 7

Let P and Q be two relations with arities p and q, respectively. Let 'comp' be a boolean combination of P- and Q-attribute comparisons. Let

$$\text{const} = \wedge_{m=1}^{m=p} (i_m \, \varepsilon_m \, \varnothing)$$

be a *constraint* on attributes of P, where an ε_m denotes '=' or '≠'.

The *constrained outer-join* $P \overset{\text{const}}{\underset{\text{comp}}{\ltimes\!\bowtie}} Q$ is the relation with arity p+1 defined as:

$$\{(c_1,...,c_p,\perp) \mid (c_1,...,c_p) \in \pi_{1...p}(P \underset{\text{comp}\wedge\text{const}}{\bowtie} Q)\}$$

$$\cup \{(c_1,...,c_p,\varnothing) \mid (c_1,...,c_p) \in P - \pi_{1...p}(P \underset{\text{comp}}{\bowtie} Q)\}$$

$$\cup \{(c_1,...,c_p,\varnothing) \mid (c_1,...,c_p) \in \sigma_{\neg\,\text{const}}(P)\}$$

The constrained outer-join operator is easily implemented by modifying any join - or outer-join - procedure. According to Definition 7, the example query Q_1: P(x) ∧ (T(x) ∨ U(x)) corresponds to:

$$E: \pi_1[\sigma_{2\neq\varnothing \vee 3\neq\varnothing}((P \underset{1=1}{\ltimes} T) \overset{2=\varnothing}{\underset{1=1}{\ltimes}} U)]$$

It is worth noting that, by definition of a constrained outer-join, the projection in the expression E cannot induce duplicate tuples. Manipulation rules for outer-joins, that apply to constrained outer-joins as well, are given in [RR 84].

Disjunctive filters containing negated subexpressions can also be evaluated by means of constrained outer-joins. Consider the relations P, T and U of Fig. 2 and the query Q_2: $P(x) \wedge [\neg T(x) \vee U(x)]$. By definition, a tuple (c) is in the difference relation P – T if the outer-join R_1: $[P \underset{1=1}{\ltimes} T]$ (Fig. 2) contains the tuple (c,∅).

Conversely, (c_1) is not in P – T if R_2 contains a tuple (c_1,c_2) such that $c_2 \neq \varnothing$. For computing the constrained outer-join R_3: $[P \underset{1=1}{\ltimes} T] \overset{2\neq\varnothing}{\underset{1=1}{\ltimes}} U$ (Fig. 4) it is sufficient to

search U only for the P-tuples that are not in P – T. The tuples of R_3 relevant to Q_2 have a null second attribute or have a non-null third attribute. Q_2 is therefore described by the algebraic expression:

$$\pi_1[\sigma_{2=\varnothing \vee 3\neq\varnothing}([P \overset{1=1}{\ltimes} T] \overset{1=1}{\overset{2\neq\varnothing}{\ltimes}} U)]$$

R_3: $[P \overset{1=1}{\ltimes} T] \overset{1=1}{\overset{2\neq\varnothing}{\ltimes}} U$

a	⊥	⊥
b	⊥	∅
c	∅	∅
d	∅	∅

Fig. 4

The following proposition summarizes the processing of disjunctive filters with constrained outer-joins.

Proposition 5
Let P and T_1, ..., T_n be unary relations. Assume that a symbol Λ_i denotes either '¬' or the empty symbol ' ' ($\Lambda_i F$ denotes $\neg F$ if Λ_i = '¬', it denotes F if Λ_i = ' ').

The calculus query

$$P(x) \wedge [\Lambda_1 T_1(x) \vee ... \vee \Lambda_n T_n(x)]$$

is equivalent to the algebraic expression

$$\pi_1[\sigma_E(...(P \overset{1=1}{\ltimes} T_1) \overset{const(1)}{\overset{1=1}{\ltimes}} T_2)... \overset{const(i)}{\overset{1=1}{\ltimes}} T_{i+1} ... \overset{const(n-1)}{\overset{1=1}{\ltimes}} T_n))]$$

where E denotes the expression:
$\Lambda_1(2\neq\varnothing) \vee ... \vee \Lambda_i(i+1\neq\varnothing) \vee ... \vee \Lambda_n(n+1\neq\varnothing)$
and where, for i = 1, ..., n-1:
$const(i) = \wedge_{k=2}^{k=i+1} \Lambda_{k-1}(k = \varnothing)$

The initial projection does not induce duplicate tuples.

[*Proof:* From Definition 7, by induction on n.]

Proposition 5 extends easily to more complex producer-filter expressions involving, e.g., n-ary relations or general attribute comparisons.

4. Conclusion

Looking at query optimization from a logical viewpoint, we have proposed a new evaluation method for quantified and disjunctive queries. The method proceeds in two phases. The first phase consists in a logical standardization of the queries into a canonical form. In the second phase, canonical expressions are translated into relational algebra. The translation we propose is unconventional and improves over the usual ones. It relies on a new operator, the *complement-join*, that generalizes the set difference, on special translations of quantified expressions, and on an improved processing of certain disjunctions by means of *constrained outer-joins*.

The standardization into canonical form has been defined by a rewriting system. This system preserve logical equivalence, hence the semantics of queries. We have shown that the system is noetherian and confluent. The canonical form permits to considerably improve the translation into relational algebra.

Unlike the rather efficient methods described by Dayal in [DAY 83, DAY 87], our method process quantifiers by means of relational algebra operators only. Thanks to the canonical form, no special procedure is needed for handling universal quantifications, as opposed to these approaches. An advantage of our approach over that proposed by Dayal is its simplicity: Neither special data structures, nor particular procedures are needed. However, this simplicity is not obtained to the cost of efficiency. Instead, the method proposed in this article significantly improves over the previous proposals. Another advantage of the approach advocated in this article is that, unlike both methods of Dayal, quantified queries can be evaluated subexpression by subexpression. In other words, the method does not impose to simultaneously access all relations occurring in the scope of a quantifier, as opposed to the approaches [DAY 83, DAY 87].

The canonical form of queries we have defined permits to avoid the systematic cartesian products induced by the conventional translation into relational algebra [COD 72, PAL 72, JS 82, CG 85]. Furthermore, it permits in most cases to translate quantified queries without calling for the expensive division operator. Finally, the complement-join operator we have defined and the special processing of disjunctions by means of outer-joins improve considerably over the usual algebraic processings.

The techniques we have described for processing quantifiers and disjunctions rely mostly on variants of a same operator, namely the join operator. This is an additional interesting feature of our approach, apart from its efficiency and simplicity. Indeed, an algebraic translation basically relying on a unique operator give rise to simplifying the cost estimation model. Further research should be devoted to investigating this issue.

This research has been motivated by an on-going project at ECRC - the building of a knowledge base management system. It is complementary to previous work on integrity constraint processing [BDM 88].

5. Acknowledgement

I am grateful to Christoph Freytag and Rainer Manthey for the many discussions we had during the elaboration of this paper. I thank Jean-Marie Nicolas, Pierre Coste, Alexandre Lefebvre, and Mark Wallace for useful comments on a first version of this article.

6. References

[BDM 88] Bry, F., Decker, H. and Manthey, R. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *Proc. EDBT '88*. March, 1988.

[BRY 89] Bry, F. Logical Rewritings for Improving the Evaluation of Quantified Queries. In *Proc. 1st Int. Conf. on Mathematical Fundamentals of Database Systems*. Visegrad, Hungary, June 26-20, 1989.

[CCO 86] Cubitt, R., Cooper, B. and Ozsoyoglu, G. (editors). *Proc. 3rd Int. Workshop on Statistical and Scientific Database Management*. Eurostat, July 22-24, Luxembourg, 1986.

[CG 85] Ceri, S. and Gottlob, G. Translating SQL in Relational Algebra: Optimization, Semantics and Equivalence of SQL Queries. *IEEE Trans.* SE-11(4), April, 1985.

[COD 72] Codd, E. *Database Systems - Courant Computer Science Symp.* Prentice Hall, Englewood Cliffs, New Jersey, 1972, Chapter Relational Completeness of Database Sublanguages.

[DAT 81] Date, C. *An Introduction to Database Systems*. Addison Wesley, 1981.

[DAY 83] Dayal, U. Processing Queries with Quantifiers: A Horticultural Approach. In *Proc. PODS '83*, pages 125-136. ACM, Atlanta, March, 1983.

[DAY 87] Dayal, U. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proc. VLDB '87*, pages 197-208. August, 1987.

[FAG 80] Fagin, R. Horn Clauses and Database Dependencies. In *12th Ann. ACM Symp. on Theory of Computing*, pages 123-134. 1980.

[HUE 80] Huet, G. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Jour. of the ACM* 27(4):797-821, October, 1980.

[JK 83] Jarke, M and Koch, J. Range Nesting: A Fast Method to Evaluate Quantified Queries. In *Proc. SIGMOD '83*, pages 196-206. ACM, San Jose, Calif, May 23-26, 1983.

[JS 82] Jarke, M. and Schmidt, J. Query Processing Strategies in the PASCAL/R Relational Database Management System. In *Proc. SIGMOD '82*. ACM, June, 1982.

[KOW 79] Kowalski, R.A. Algorithm = Logic + Control. *Commun. ACM* , Aug., 1979.

[KUH 67] Kuhns, J.L. *Answering Question by Computer: A Logical Study*. Technical Report RM-5428-PR, Rand Corp., 1967.

[LP 76] Lacroix, M and Pirotte, A. Generalized Joins. *SIGMOD Records* 8(3):14-15, September, 1976.

[LT 86] Lloyd, J.W. and Topor, R.W. A Basis for Deductive Database Systems II. *Jour. of Logic Programming* 3(1):55-67, 1986.

[MEN 79] Mendelson, E. *Introduction to Mathematical Logic*. Van Nostrand, New York, 1979.

[ND 83] Nicolas, J.-M. and Demolombe, R. *On the Stability of Relational Queries*. Technical Report, ONERA-CERT, Jan., 1983.

[PAL 72] Palermo, F. A Data Base Search Problem. In *Proc. 4th Symp. on Computer and Information Sc.*. 1972.

[RR 84] Rosenthal, A. and Reiner, D. Extending the Algebraic Framework of Query Processing to Handle Outerjoins. In *Proc. VLDB'84*, pages 334-343. August, 1984.

[SC 75] Smith, J.M. and Chang, P.Y.T. Optimizing the Performance of a Relational Algebra Database Interface. *Commun. ACM* 18(10):568-579, Oct., 1975.

[SCH 87] Schmitt, P. H. A Survey of Rewrite Systems. In *Proc. of the 1st Workshop on Computer Science Logic (CSL '87)*, pages 235-262. Oct., 1987.

[SEL 86] Sellis, T. Global Query Optimization. In *Proc. SIGMOD '86*, pages 191-205. ACM, 1986.

[SHE 88] Shepherdson, J.C. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, CA, 1988, pages 19-88, Chapter Negation in Logic Programming.

[SHI 81] Shipman, D. The Functional Data Model and Data Language DAPLEX. *ACM Trans. on Database Systems* 6(1), March, 1981.

[VGT 87] Van Gelder, A. and Topor, R.W. Safety and Correct Translation of Relational Calculus Formulas. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 313-327. 1987.

[YAO 79] Yao, S.B. Optimization of Query Evaluation Algorithms. *ACM Trans. on Database Systems* 4(2):133-155, June, 1979.

[ZLO 77] Zloof, M. Query by Example. *IBM Systems Jour.* 16(4), 1977.

[ZOO 77] Zook W. et al. *INGRES Reference Manual*. University of California, Berkley, 1977.