Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions

RYOHEI NAKANO
NTT Communications and Information Processing Laboratories

Most of the previous translations of relational calculus to relational algebra aimed at proving that the two languages have the equivalent expressive power, thereby generating very complicated relational algebra expressions, especially when aggregate functions are introduced. This paper presents a rule-based translation method from relational calculus expressions having both aggregate functions and null values to optimized relational algebra expressions. Thus, logical optimization is carried out through translation. The translation method comprises two parts: the translation of the relational calculus kernel and the translation of aggregate functions. The former uses the familiar step-wise rewriting strategy, while the latter adopts a two-phase rewriting strategy via standard aggregate expressions. Each translation proceeds by applying a heuristic rewriting rule in preference to a basic rewriting rule. After introducing SQL-type null values, their impact on the translation is thoroughly investigated, resulting in several extensions of the translation. A translation experiment with many queries shows that the proposed translation method generates optimized relational algebra expressions. It is shown that heuristic rewriting rules play an essential role in the optimization. The correctness of the present translation is also shown.

Categories and Subject Descriptors: H.2.3[**Database Management**]: Languages—*query languages*; H.2.4[**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Aggregate function, null value, query languages, query optimization, relational algebra, relational calculus, translation

1. INTRODUCTION

Relational calculus is declarative and relational algebra is procedural [3]. Therefore, the former is suitable for user interface, while the latter is suitable for database operations. Recent database machines [8, 18, 21] execute relational algebra very quickly. Consequently, system architectures have become feasible in which relational calculus expressions (hereafter called *calculus expressions*) written by users are internally translated to relational algebra expressions (hereafter called *algebraic expressions*), the algebraic expressions are optimized, and

Author's address: Knowledge Systems Laboratory, NTT Communications and Information Processing Laboratories, 1-2356, Take, Yokosuka, Kanagawa 238-03, Japan.

the optimized algebraic expressions are evaluated by database machines. These three steps are called *query translation*, *query optimization*, and *query evaluation*, respectively. Moreover, query optimization can be divided into logical and physical aspects. In such architectures, a subsystem that translates calculus expressions to succinct (optimized) algebraic expressions is indispensable.

Bearing the above in mind, we focus on query translation and query optimization; as for the latter, only logical optimization is focused on. Here we introduce a new point of view to classify the methods of query translation and optimization. We call a translation that performs simultaneous logical optimization *translation with optimization*. An alternative called *optimization after translation* exists in which calculus expressions are translated in any way to algebraic expressions of some (probably high) level of complexity, and then a logical optimizer [22, 23, 24] works on them. Thus, the approach consists of two steps: *translation without optimization* and logical optimization. Almost all previous translations [1, 2, 3, 17, 19, 20, 23] can be classified as translations without optimization.

The main claim of this paper is that translation with optimization is more effective and more promising than optimization after translation, because it seems quite difficult to optimize complicated algebraic expressions.

Previous algorithms [3, 20, 23] for translating from relational calculus to relational algebra aimed at proving that the two languages have the equivalent expressive power. They presented the translation of the relational calculus kernel, which includes neither aggregate functions nor null values. They were not intended for actual execution of their translated results, so the algebraic expressions generated were very complicated. The reason they generated such complicated algebraic expressions was not due to the translation strategy but because they had only a minimum set of rewriting rules.

The introduction of aggregate functions produces open alphas, which seriously complicate translations. In [17] the scope of translation was extended by adding aggregate functions to the kernel. It showed that both relational calculus extended to include aggregate functions and relational algebra extended to include the aggregate formation have the same expressive power. However, the translation generated a highly complex algebraic expression even for a simple calculus expression.

The research cited in [1, 2, 7, 16, 19] on translating SQL queries is also relevant to the present paper. Although aggregate functions are treated by all of the above researchers, null values are investigated by only one of them.

In [1], a translation from SQL to relational algebra via relational calculus is presented. Null values incorporated in SQL are extensively considered in the translation. However, the translation can be regarded as an extension of Codd's algorithm [3]; hence, it is a translation without optimization. Moreover, the introduction of aggregate functions to relational calculus seems easygoing, that is, the same aggregate formation as introduced in relational algebra is adopted. This results in both the straightforward translation of aggregate functions and the disturbance of the declarative nature of relational calculus.

In [2], a translation from SQL to relational algebra having aggregate functions is presented. It introduces a new algebraic operation for aggregations, which can be regarded as a composite operation of the aggregate formation and the join.

The translation is performed in two steps. The first step transforms SQL queries into equivalent SQL queries acceptable by a restricted grammar, and the second step transforms them into algebraic expressions. There are few heuristic rules in the second transformation, for example, no heuristic rules for divisions, elimination of join, negative semijoin, or aggregate functions; hence the translation is one without optimization and the translator can be regarded as the front-end of an optimizer. Moreover, the impact of null values is not considered.

In [7], a translation from SQL to relational algebra is investigated, aiming at increasing the strategy space for an optimizer. It has in common with this paper that of using the approach of translation with optimization. It differs, however, from the present paper mainly in the following features: (1) The description is less formal, for example, the impact of null values is not formally considered; (2) heuristic rules are not sufficient, for example, about divisions, elimination of join, and aggregate functions; (3) aggregations for empty sets are treated by introducing the outer join, while we solve the problem by extending the aggregate formation; and (4) the feature of control of duplicates in SQL is faithfully considered, while we always eliminate duplicates.

In [16], a translation from SQL to SQL is presented, transforming arbitrarily nested queries into simpler ones. The algorithms can be seen to realize a translation with optimization. However, it has only basic rewriting rules and no heuristic ones, as proposed in this paper. Unfortunately, aggregates over empty sets are not taken into consideration here.

In [19], a translation from SQL to QUEL is presented. Since these two query languages are both calculus-type, the translation is rather straightforward. In addition, aggregates over empty sets are not treated properly, and null values are not considered because they were not available in QUEL at the time.

Recent works [9, 10, 11] on rule-based optimizers are also quite relevant to this paper. In common with this paper, they employ rule-based rewriting techniques. However, they differ from the present paper in two ways. First, they focus on query optimization from logical (operator-level) to physical (method-level), while this paper covers both query translation and logical optimization. Second, they find the (physically) optimized expression via generate-and-test methods, while this paper finds the (logically) optimized expression through the application of rewriting rules.

This paper presents a translation with optimization, which has enough rewriting rules, especially heuristic ones, and translates calculus expressions having aggregate functions to optimized algebraic expressions. *Heuristic rewriting rules*, into which human knowledge about constructing succinct expressions is incorporated, play an essential role in optimization through translation. The optimization accomplished in this paper can be classified into semantic optimization [14], since it uses the knowledge on query semantics. The present translation method comprises two parts: the translation of the relational calculus kernel and the translation of aggregate functions. The former uses the familiar step-wise rewriting strategy, while the latter adopts a two-phase rewriting strategy via the special pattern of expressions called *standard aggregate expressions*. Each translation proceeds by applying a heuristic rewriting rule in preference to a basic one. Then, after introducing SQL-type null values, their impact on the present translation methods is thoroughly investigated. A translation experiment with many queries shows that the present translation method generates optimized algebraic expressions. It should be noted that the present translation of aggregate functions produces consistent results even for aggregates over empty sets, a proper translation of which is a well-known knotty problem [15, 17].

In Section 2, we define the syntax and semantics of relational calculus and relational algebra extended to include aggregate functions. A new type of aggregate formation is adopted in relational algebra together with the one originally proposed in [17]. In Section 3, the significance of translation with optimization is discussed. Then, we present the translation of the relational calculus kernel in Section 4 and the translation of aggregate functions in Section 5. Both sections give the general flows of the translation method and detailed rewriting rules with proofs. Section 6 strictly defines SQL-type null values and comprehensively investigates their impact on the present translation along with the relevant extensions of relational calculus and algebra. Section 7 presents several translation examples, evaluates the optimization level, and compares the general aggregate formation with the outer join.

The examples used in this paper are based on the following schema [5]:

emp(name, sal, mgr, dept)
sales(dept, item, vol)
supply(comp, dept, item, vol)
loc(dept, floor)
class(item, type)

2. DEFINITIONS

The following formal definitions of the relational model, relational calculus, and relational algebra are mainly based on [17], but some extensions are performed.

2.1 Relational Model

A relation scheme has a relation name R together with a degree of R, referred to as $\deg(R)$. The set of attributes is associated with R and each attribute is identified by an attribute number $j \in \{1, \ldots, \deg(R)\}$. For simplicity, the domain of each attribute is assumed to be the set **N+** of natural numbers including the null value. The null value is inevitably introduced here in the definition of aggregate functions, but its treatment will be intensively described in Section 6. Until then, null values are not taken into consideration. A relation r of degree n is a finite set of n-tuples. Duplicates, tuples which have the same values for all corresponding attributes, are always eliminated due to the nature of a set. A schema is a set $\{R_1, \ldots, R_N\}$ of relation schemes. An instance I of schema $\{R_1, \ldots, R_N\}$ is a set $\{r_1, \ldots, r_N\}$, where r_i is a relation over R_i for each $i = 1, \ldots, N$. Throughout this paper, one fixed schema $\{R_1, \ldots, R_N\}$ is assumed.

2.2 Relational Calculus

The relational calculus kernel was originally defined in [3]. In [17] it is extended in two ways: the introduction of aggregate functions and the extension of the form of range formulas.

Calculus expressions are defined recursively by using several classes of objects: variables, terms, formulas, range formulas, and alphas.

(a) *Variables.* A variable is denoted by a lower case letter or a lower case letter with a subscript. For example, u, v_1, v_2, w, and x denote variables.

(b) *Terms.* A constant (an integer or a string) is a term. If u is a variable and A is an attribute number, then $u[A]$ is a term. If α is an alpha (defined below) and f is an aggregate function, then $f(\alpha)$ is a term. Given a relation evaluated as α, an aggregate function produces either a real number or the null value. It has the following form: count or fun$[A]$, where A is an attribute number and fun $\in$ {min, max, sum, avg}. If the relation to which an aggregate function is applied is empty, then count $= 0$ and fun$[A]$ = null.

(c) *Formulas.* If t_1, t_2 are terms and $\theta \in \{=, <>, <, <=, >, >=\}$, then $t_1 \ \theta \ t_2$ is a formula (called an atomic formula). If f and g are formulas, then $\sim f$, $f \vee g$ and $f \wedge g$ are formulas. If f is a formula and $r(u)$ is a range formula, then $(\exists r(u))f$ and $(\forall r(u))f$ are formulas.

(d) *Range formulas.* If $\alpha_1, \ldots, \alpha_k$ are closed alphas (defined below) and u is a variable, then $(\alpha_1 \vee \cdots \vee \alpha_k)(u)$ is called a range formula for u. A range formula $r(u)$ defines the range r where the variable u moves.

(e) *Alphas.* If R_i is a relation scheme, then R_i is an atomic alpha of $\deg(R_i)$ and corresponds to a basic relation. If $t_1, \ldots, t_n$ are terms, $r_1(u_1), \ldots, r_n(u_n)$ are range formulas for $u_1, \ldots, u_n$ and ψ is a formula, then

$$(t_1, \ldots, t_n){:}r_1(u_1), \ldots, r_n(u_n){:}\psi$$

is an alpha of degree n, where the list $(t_1, \ldots, t_n)$ is called the target list, and ψ is called the qualifier. This form of an alpha corresponds to a derived relation.

Only a closed alpha is called a *calculus expression.* A closed alpha and an open alpha are defined next. An alpha is closed if it has no free occurrences of any variable. Otherwise, an alpha is open. An atomic alpha is closed. An occurrence of a variable u is *free for an alpha* if it is neither within the scope of a quantifier ($\exists r(u)$ or $\forall r(u)$) in the qualifier of the alpha nor defined as a range formula of the alpha. Otherwise it is bound. Note that whether an occurrence of a variable is free or not is determined from the standpoint of an alpha. This is not explicitly pointed out in [17]. For instance, in the following example,

Example 1. For each item, find the number of companies which supply the item.

```
(u[1], count((v[1]):supply(v):v[3] = u[1])):class(u):()
```

the second occurrence of u in the target list is bound for the outer alpha but free for the inner alpha. Thus, the outer alpha is closed and the inner one is open.

With respect to a schema instance I, the calculus expression $(t_1, \ldots, t_n){:}r_1(u_1), \ldots, r_n(u_n){:}\psi$ can be interpreted without using any valuation, as follows:

$$\{(t_1(u_1, \ldots, u_n), \ldots, t_n(u_1, \ldots, u_n)) \mid u_1 \in r_1(I) \wedge \cdots \wedge u_n \in r_n(I) \wedge \psi(u_1, \ldots, u_n)\}.$$

Note that the above interpretation will be used in proofs presented later.

2.3 Relational Algebra

We extend the usual relational algebra in the following points to make it more practical: the formula-type selection, the positive and negative semijoins, the conjunctive join (or conjunctive semijoin), and the more general aggegate formation. Note that an attribute number is distinguished from an integer constant by attaching the prefix #.

With respect to schema instance I, an algebraic expression can be defined as follows, where e, e_1, and e_2 are algebraic expressions, A, A_1, and A_2 are attributes, F_s is a selection condition, F_j is a join condition, ϕ is an empty set, agg is an aggregate function that is either count or fun$[A]$ for fun $\in$ {min, max, sum, avg}, and agg's is one or more aggregate functions.

Projection

$$(e[A])(I) = \{t[A] \mid t \in e(I)\}$$

Selection

$$(e[F_s])(I) = \{t \mid t \in e(I) \wedge f_s(t)\}$$

Join

$$(e_1[F_j]e_2)(I) = \{t \circ s \mid t \in e_1(I) \wedge s \in e_2(I) \wedge f_j(t, s)\},$$

where $\circ$ denotes concatenation.

Cross Product

$$(e_1[\,]e_2)(I) = \{t \circ s \mid t \in e_1(I) \wedge s \in e_2(I)\}$$

Positive Semijoin

$$(e_1[\exists;\, F_j]e_2)(I) = \{t \mid t \in e_1(I) \wedge s \in e_2(I) \wedge f_j(t, s)\}$$

Negative Semijoin

$$(e_1[\sim\exists;\, F_j]e_2)(I) = \{t \mid t \in e_1(I) \wedge \{s \mid s \in e_2(I) \wedge f_j(t, s)\} = \varnothing\}$$

Division

$$(e_1[A_1/A_2]e_2)(I) = \{t[\sim A_1] \mid t \in e_1(I) \wedge \{s[A_1] \mid s \in e_1(I) \wedge s[\sim A_1] = t[\sim A_1]\} \supseteq \{r[A_2] \mid r \in e_2(I)\}\}$$

Otherwise,

$$(e_1[A_1/A_2]e_2)(I) = \{t[\sim A_1] \mid t \in e_1(I) \wedge s \in e_1(I) \wedge r \in e_2(I) \wedge \forall r \exists s (s[\sim A_1] = t[\sim A_1] \wedge s[A_1] = r[A_2])\}$$

Union

$$(e_1[+]e_2)(I) = \{t \mid t \in e_1(I) \vee t \in e_2(I)\}$$

Intersection

$$(e_1[*]e_2)(I) = \{t \mid t \in e_1(I) \wedge t \in e_2(I)\}$$

Difference

$$(e_1[-]e_2)(I) = \{t \mid t \in e_1(I) \wedge t \notin e_2(I)\}$$

Aggregate Formation

$$(e <A;\ \text{agg's}>)(I) = \{t[A] \circ f\text{'s} \mid t \in e(I) \wedge f\text{'s} = \text{agg's}(\{r \mid r \in e(I) \wedge r[A] = t[A]\})\}$$

General Aggregate Formation

$$(e_1[A_1/A_2;\ \text{agg's}]e_2)(I) = \{t[A_2] \circ f\text{'s} \mid t \in e_2(I) \wedge f\text{'s} = \text{agg's}\ (\{r \mid r \in e_1(I) \wedge r[A_1] = t[A_2]\})\}$$

Note that count(∅) = 0, fun[A](∅) = null.

The selection condition is usually atomic, for example, emp[#4 = toy]. Most intelligent disk systems can process several atomic selection conditions in parallel, however, thus efficiently processing formula-type selection, for example, emp[#2 >10000 ∧ (#4 = toy ∨ #4 = dress)]. Hence, formula-type selection is adopted, which includes the restriction.

The semijoin selects from one relation all tuples that have at least one partner tuple in the other relation; namely, it selects only nondangling tuples and directly reflects the existential quantifier. The semijoin described above is called a positive semijoin. As a counterpart, we can create the negative semijoin that selects only dangling tuples and directly reflects the negation of the existential quantifier. In general the semijoin has a smaller processing load than the join, so we should use the semijoin anywhere possible instead of the join.

The conjunctive join (or conjunctive semijoin) is the join (or semijoin) whose condition is a conjunctive form of atomic formulas. For example, emp[∃; #3 = #1, #2 > #2]emp exemplifies the conjunctive semijoin which finds employees who earn more than their managers.

The aggregate formation was originally introduced in [17]. We introduce the more general aggregate formation, which includes the original one as a special case. Example 1 is used to explain this new operation. The following algebraic expression using the original aggregate formation appears to represent correctly the Example 1 query:

((class[#1])[#1 = #2](supply[#1, #3])) ⟨#1; count⟩.

When (class[#1][−]supply[#3]) is not empty, however, this expression loses those items which were supplied in the past but not in the present. We should add the following correction (called *Klug's correction*) [17]:

((class[#1])[#1 = #2](supply[#1, #3])) ⟨#1; count⟩
[+]((class[#1][−]supply[#3])[ ]{0}).

Klug's correction as expressed above is rather artificial. The general aggregate formation can briefly express an aggregation with Klug's correction as shown below:

(supply[#1, #3])[#2/#1; count]class.

This new operation considers the set of items registered in the class[#1] as grouping-by keys. Then it groups each tuple of the supply[#1, #3] relation by considering the value of supply[#3] as grouped-by keys. Note that a grouping-by key that has a zero count still remains.

The original aggregate formation is a special case where the set of grouping-by-keys and the set of grouped-by-keys are identical. Such keys are simply called group-by-keys. When group-by-keys are omitted, an aggregate formation is a scalar function. Note also that an original or general aggregate formation can have several aggregate functions.

3. SIGNIFICANCE OF TRANSLATION WITH OPTIMIZATION

This paper presents a method for translating from relational calculus to relational algebra with simultaneous logical optimization, called *translation with optimization.* An alternative called *optimization after translation* exists in which calculus expressions are translated to algebraic expressions in any way and then a logical optimizer is applied to get optimized algebraic expressions. Figure 1 illustrates these two approaches. *Translation with optimization* starts from a calculus expression A and ends up with the optimized algebraic expression Z, while *optimization after translation* reaches the expression Z via a complicated algebraic expression P.

As shown in Figure 1, the translation starts from a pure calculus expression and terminates at a pure algebraic expression. In an intermediate process, an expression is a mixture of the two kinds of expressions. When necessary, we call such a mixture a *mixed expression.*

Note that we concern ourselves only with logical optimization [23]—transformations of algebraic manipulations with no concern, or little concern, for how the relations are stored. Algebraic operations can be classified into three categories in respect of processing time: light, middle, and heavy operations. When hashing algorithms and no indices are used [21], light, middle, and heavy operations require one attribute scan of data, more than one attribute scan, and a scan of all attributes, respectively. Each operation can be classified as follows:

light = {selection},
middle = {join, cross product, semijoins, aggregate formations},
heavy = {projection, division, union, intersection, difference}.

Roughly speaking, further optimization means, first of all, fewer heavy operations, next, fewer middle operations, and lastly, fewer light operations. More strictly speaking, even when the number and classes of operations are the same, the sequence of operations is known to be critical, for example, "perform selections as early as possible." Moreover, within the same class the processing loads are not the same, for example the semijoin is lighter than the join, or the aggregate formation is lighter than the general aggregate formation. All these should be considered in logical optimization.

Note that we do not consider physical optimization. Hence, for example, we cannot find the most efficient processing ordering of successive joins; taking into consideration physical organization of a database can cope with such cases.

3.1 Optimization of Complicated Expressions

This section shows how difficult it is to optimize very complicated algebraic expressions.

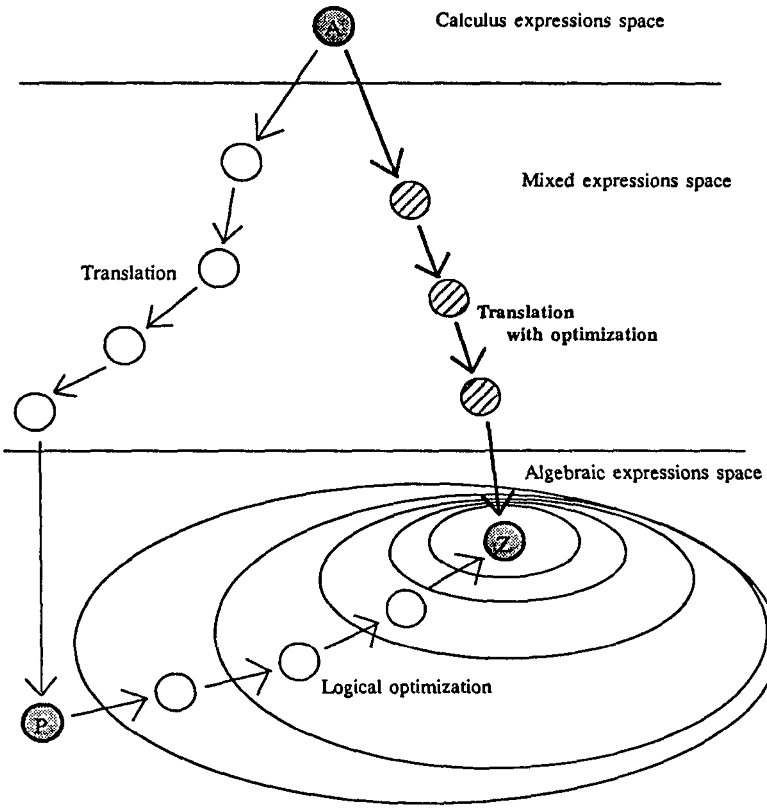
A
Calculus expressions space
Mixed expressions space
Translation
Translation with optimization
Algebraic expressions space
Z
Logical optimization
P

Fig. 1. Translation with optimization.

Example 2. Find those companies, each of which supplies every item.

(x[1]): supply(x): ∀class(y)∃supply(z) (z[1] = x[1] ∧ z[3] = y[1])

From the above calculus expression, a typical translation [3] without optimization generates the following algebraic expression:

((supply[ ]class[ ]supply)[#1 = #7][#5 = #9]
  [#1, #2, #3, #4, #5, #6][(#5, #6)/(#1, #2)]class)[#1].

On the other hand, after consideration we can arrive at the following optimized algebraic expression:

supply[#1, #3][#2/#1]class.

Although these two algebraic expressions do not appear to be equivalent, they are, as shown here. Starting from the complicated one, we get the following by combining selections with cross products.

((supply[#1 = #3](class[#1 = #3]supply))[#1, #2, #3, #4, #5, #6]
  [(#5, #6)/(#1, #2)]class)[#1]

Although in the division two attributes of the class relation are used, only the first one is essential for this query, as shown explicitly in the above calculus expression. Thus, even if we discard the second attribute, the equivalence of the expressions still holds. This can be seen by replacing "class" with "class[#1]" from the beginning, as shown below, and then eliminating the unnecessary projection of "class[#1]."

((supply[#1 = #3](class[#1][#1 = #3]supply))[#1, #2, #3, #4, #5]
    [#5/#1]class[#1])[#1]

If we consider only the division, the discarding of the second attribute appears impossible. Only global consideration can make the elimination possible. Hence we get

((supply[#1 = #3](class[#1 = #3]supply))[#1, #2, #3, #4, #5][#5/#1]class)[#1]

The dividend is the first attribute of the first occurrence of the class relation, which is set equal to the third attribute of the second occurrence of the supply relation. We replace the former with the latter, eliminating the relevant join. Although this elimination fails to remove dangling tuples, the equivalence of the expressions still holds, because in the end dangling tuples have no influence on the division. Again, the global consideration, the combination of the join and the division, can guarantee that the dangling tuples are harmless.

((supply[#1 = #1]supply)[#1, #2, #3, #4, #7][#5/#1]class)[#1]

It is easily seen that the join can be omitted by projecting only the relevant attributes. Thinking through the above query makes the omission possible.

(supply[#1, #3][#2/#1]class)[#1]

By removing the superfluous projection, we finally reach the optimized expression

supply[#1, #3][#2/#1]class.

The original complicated expression consists of two joins, two projections, and one division; on the other hand, the optimized expression consists of one projection and one division. Roughly speaking, the difference in the processing load is one heavy operation (projection) and two middle operations (joins).

Any previous optimization algorithm could not show the above equivalence, essentially because complicated algebraic expressions may lose essential structural information about a query, and directly because rewriting rules are not enough, in this case, the division is not usually taken into consideration. Note that the above sequence of transformations effectively uses the information specific to the query as the global considerations. The information is explicitly expressed in the original calculus expression but not in the complicated algebraic expression. This example suggests that the optimization of complicated algebraic expressions seems quite difficult and the difficulty is due to loss or obscurity of the information essential to get optimized expressions.

There is another reason why the optimization of complicated algebraic expressions seems difficult. Even if we could find out enough rewriting rules to meet the transformation of complicated algebraic expressions, we could not control the sequence of rule application, because there might exist many rewriting rules applicable to a calculus or mixed expression. That is, it seems difficult to find adequate rules, called *meta-rules* to distinguish them from rewriting rules, that control the sequence of rewriting rule application. Alternatively, we can adopt

Table I. Symbol Notation

$\{\}$	set
$\varnothing$	empty set
$\mathrm{len}(L)$	length of a list L
I	any schema instance
α	alpha
tl	target list
tl_y	target list which has only one variable y
TL	projection condition
$y, z, \ldots$	variables
$\mathrm{Y}(y), \ldots$	range formulas, where Y denotes a range and y denotes a variable
$\Psi(\psi), \ldots$	range formulas, where Ψ denotes ranges(vector) and ψ denotes variables(vector)
$f, g, \ldots$	formulas
f_s	selection-type formula
f_j	join-type formula
f_e	equijoin-type formula
F_s	selection condition
F_j	join condition
F_e	equijoin condition
y_f	set of terms $y[i]$ which appear in a formula f
y_z	set of terms $y[i]$ which appear in an alpha Z
y_{tl}	set of terms $y[i]$ which appear in a target list tl

the generate-and-test method to get the optimized expression, which means, however, a full search of a vast graph of expressions.

3.2 Rationale for Translation with Optimization

Translation with optimization is supported by the following consideration. Given a query, we humans can consider both the succinct calculus expression and the succinct (optimized) algebraic expression. Hence, for a set of various queries there are two sets: a set of calculus expressions and a set of algebraic expressions, with one-to-one mapping between them. We assume that the structural correspondence that exists in the mapping can be summarized into rewriting rules. From many mapping examples we have extracted 24 rewriting rules. Given enough rules, the translation with optimization is expected to work well.

Note that, in this approach, quite simple control will do for meta-rules; that is, the simple static priority control of rewriting rule application will suffice. This means a single-path search of the graph of expressions, which is quite efficient.

For instance, for such queries as Example 2 we have the following rewriting rule:

Rule D. Division

from $tl_x{:}X(x){:}\forall Y(y)\exists X[H_s](z)(f_e(x, z) \wedge g_e(y, z))$

to $tl_w{:}(X[H_s][z_f, z_g][z_g/y_g]Y)(w){:}(\,)$

where $Y(I) \neq \phi$, $x_f \supseteq tl_x$, and H_s denotes the selection.

By applying this rule to the calculus expression we get the following succinct mixed expression.

(w[1]):(supply[#1, #3][#2/#1]class)(w):( )

From this we can easily derive the optimized algebraic expression.

Table I shows the notation of symbols that will be used in the succeeding sections.

4. TRANSLATION OF RELATIONAL CALCULUS KERNEL

This section presents the translation of the relational calculus kernel, which includes neither aggregate functions nor null values.

4.1 Overview

Since in the relational calculus kernel every calculus expression is closed, we adopt the following simple translation strategy:

(a) for a nested alpha, translate the inner closed alpha first,

(b) within an alpha, translate in the order of range formulas, the qualifier, and the target list,

(c) within a qualifier (formula), translate along with the parse tree of the formula.

Although the strategy is simple, the enrichment of rewriting rules will guarantee optimization.

Rewriting rules can be classified into two categories: basic and heuristic. Basic rules can be regarded as rules for word-for-word translation, and are sufficient to complete a translation, but generally do not allow optimization. Heuristic rules play an essential role in the optimization; that is, they find idiomatic patterns in calculus expressions and rewrite them to succinct algebraic expressions in one stroke. Hence, enrichment of heuristic rules means improvement of translation system's intelligence.

Skillful rewriting of qualifiers requires *formula classification* as shown in Table II. Logical connectives $\wedge$ and $\vee$ in calculus expressions are usually replaced with the intersection and the union, respectively [3, 20, 23]. However, we often find instances where the logical connectives should be included in the conditions of the selection, join, semijoin, or division. Good use of the formula classification enables us to describe appropriate rewriting rules.

The general flow of the kernel translation is shown in Figure 2. We assume that a calculus expression is already parsed.

First, reduce the scope of each negation sign in a formula by making repeated use of the following substitutions until any negative sign finally disappears.

replace	$\sim(\sim f)$	by	f
replace	$\sim(f \wedge g)$	by	$(\sim f) \vee (\sim g)$
replace	$\sim(f \vee g)$	by	$(\sim f) \wedge (\sim g)$
replace	$\sim\forall X(x)f$	by	$\exists X(x)(\sim f)$
replace	$\sim\exists X(x)f$	by	$\forall X(x)(\sim f)$
replace	$\sim(t_1\theta t_2)$	by	$t_1(\sim\theta)t_2$

In the above substitutions, f and g are formulas, t_1 and t_2 are terms, $\theta \in \{=, \langle\,\rangle, <, <=, >, >=\}$, $\sim\theta$ denotes a complement of θ, and X is a closed alpha.

Next, classify a qualifier as shown in Table II. After the classification, apply a heuristic rewriting rule in preference to a basic one. Iterate this cycle until we finally get a pure algebraic expression. For a nested alpha, the iterative cycle is applied first to the innermost closed alpha, and then proceed to the outer ones.

Table II. Formula Classification

saf	selection-type atomic formula	atomic formula having a single variable
jaf	join-type atomic formula	atomic formula having different variables
eaf	equijoin-type atomic formula	jaf whose comparator is =(equal)
sf	selection-type formula	formula having a single variable
jf	join-type formula	jaf or conjunction of jafs where variable-pairs are the same
ef	equijoin-type formula	eaf or conjunction of eafs where variable-pairs are the same
mf	mixed formula	formula different from either of the above formulas

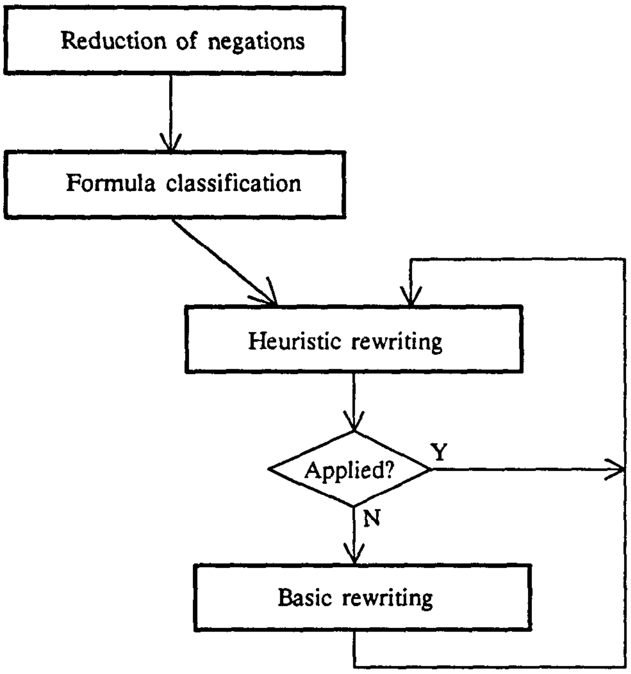

Fig. 2. General flow of kernel translation.

For the kernel translation there are eight basic rewriting rules and seven heuristic ones, as shown below:

Basic:	R (range),
	S (selection),
	PEQ (positive existential quantifier),
	NEQ (negative existential quantifier),
	J (join),
	UQ (universal quantifier),
	T (transformation),
	P (projection),

Heuristic:	JD (join and division),
	DS (division & semijoin),
	D (division),
	EJ (elimination of join),
	EP (elimination of projection),
	PS (positive semijoin),
	NS (negative semijoin).

A conflicting state can exist where more than one rewriting rule is applicable. Generally in such a conflict an application of a different rule generates a different result. However, a conflict seems to be unusual because we have decided to apply a heuristic rewriting rule in preference to a basic one. That is, conflicts are restricted to appear within basic rules or within heuristic ones. Within basic rules a conflict will be resolved if we give Rule S the highest priority and Rule T the lowest one. Within heuristic rules a conflict may exist among Rules JD, DS, and D. As described later, we give Rule D the highest priority and Rule JD the lowest one.

4.2 Basic Rewriting Rules

This section describes eight basic rewriting rules for the kernel translation.

Rule R. Range

from $tl{:}(X \vee Y \vee \ldots)(u),\ \Omega(\omega){:}f$
to $tl{:}(X[+]Y[+]\ldots)(u),\ \Omega(\omega){:}f$

PROOF. Omitted. □

Rule S. Selection

from $tl{:}\Psi(\psi),\ \Omega(\omega){:}f_s(\psi) \wedge g$
to $tl{:}\Psi[f_s](\psi),\ \Omega(\omega){:}g$

PROOF.

$$\begin{aligned}
&(tl{:}\Psi(\psi),\ \Omega(\omega){:}f_s(\psi) \wedge g)(I)\\
&= \{(\psi \circ \omega)[\mathrm{TL}] \mid \psi \in \Psi(I) \wedge \omega \in \Omega(I) \wedge f_s(\psi) \wedge g\}\\
&= \{(\psi \circ \omega)[\mathrm{TL}] \mid \psi \in \{\psi' \mid \psi' \in \Psi(I) \wedge f_s(\psi')\} \wedge \omega \in \Omega(I) \wedge g\}\\
&= \{(\psi \circ \omega)[\mathrm{TL}] \mid \psi \in \Psi[F_s](I) \wedge \omega \in \Omega(I) \wedge g\}\\
&= (tl{:}\Psi[f_s](\psi),\ \Omega(\omega){:}g)(I) \quad \Box
\end{aligned}$$

Comment. A selection-type formula is rewritten as a formula-type selection. To perform selections as early as possible, we rewrite the following formula patterns, in addition to that above: $g \wedge f_s(\psi)$, $(f_s(\psi) \wedge g) \wedge h$, $(g \wedge f_s(\psi)) \wedge h$, $h \wedge (f_s(\psi) \wedge g)$, $h \wedge (g \wedge f_s(\psi))$, $(\exists\Psi(\psi)(f_s(\psi) \wedge g))$, and $(\exists\Psi(\psi)(g \wedge f_s(\psi)))$.

Rule PEQ. Positive Existential Quantifier

from $tl{:}\Psi(\psi){:}(\exists\Omega(\omega)f) \wedge g$
to $tl{:}\Psi(\psi),\ \Omega(\omega){:}f \wedge g$

PROOF.

$$\begin{aligned}&(tl{:}\Psi(\psi){:}(\exists\Omega(\omega)f)\ \wedge\ g)(I)\\ &=\{\psi[\mathrm{TL}]\mid \psi\in\Psi(I)\ \wedge\ \exists\omega(\omega\in\Omega(I)\ \wedge\ f(\psi,\omega))\ \wedge\ g\}\\ &=\{\psi[\mathrm{TL}]\mid \psi\in\Psi(I)\ \wedge\ \omega\in\Omega(I)\ \wedge\ f(\psi,\omega)\ \wedge\ g\}\\ &=(tl{:}\Psi(\psi),\ \Omega(\omega){:}f\ \wedge\ g)(I)\quad\square\end{aligned}$$

Rule NEQ. Negative Existential Quantifier

from $tl{:}\Psi(\psi){:}(\sim\exists\Omega(\omega)f)\ \wedge\ g$
to $tl{:}(((\psi',\ \omega'_f){:}\Psi(\psi'),\ \Omega(\omega'){:}\sim f)[\omega'_f/\omega_f]\Omega)(\psi){:}g$

PROOF.

$$\begin{aligned}&(tl{:}\Psi(\psi){:}(\sim\exists\Omega(\omega)f)\wedge g)(I)\\ &=\{\psi[\mathrm{TL}]\mid\psi\in\Psi(I)\wedge g(\psi)\wedge\sim\exists\omega(\omega\in\Omega(I)\wedge f(\psi,\omega))\}\\ &=\{\psi[\mathrm{TL}]\mid\psi\in\Psi(I)\wedge g(\psi)\wedge\psi\notin\{\psi''\mid\psi''\in\Psi(I)\wedge\omega\in\Omega(I)\wedge f(\psi'',\omega)\}\}\\ &=\{\psi[\mathrm{TL}]\mid\psi\in\Psi(I)\wedge g(\psi)\wedge\psi\in\{\psi'\mid\psi'\in\Psi(I)\wedge\forall\Omega(\omega)(\sim f(\psi',\omega))\}\} &&(1)\\ &=\{\psi[\mathrm{TL}]\mid g(\psi)\wedge\psi\in\{\psi'\mid\psi'\in\Psi(I)\wedge\forall\Omega(\omega)(\sim f(\psi',\omega))\}\}\\ &=\{\psi[\mathrm{TL}]\mid g(\psi)\wedge\psi\in\{\psi'\mid\psi'\in\Psi(I)\wedge\forall\Omega(\omega)\ \exists\Omega(\omega')(\sim f(\psi',\omega')\wedge\omega_f=\omega'_f)\}\}\\ &=\{\psi[\mathrm{TL}]\mid g(\psi)\wedge\psi\in(((\psi',\omega'_f){:}\Psi(\psi'),\Omega(\omega'){:}\sim f(\psi',\omega'))[\omega'_f/\omega_f]\Omega)(I)\} &&(2)\\ &=(tl{:}(((\psi',\omega'_f){:}\psi(\psi'),\Omega(\omega'){:}\sim f(\psi',\omega'))[\omega'_f/\omega_f]\Omega)(\psi){:}g)(I)\quad\square\end{aligned}$$

Explanation. (1) Directly derivable from substituting $\sim\exists\Omega(\omega)f$ by $\forall\Omega(\omega)(\sim f)$. (2) Apply Rule JD.

Rule J. Join

from $tl{:}X(x), Y(y), \Omega(\omega){:}f_j(x,y)\wedge g$
to $tl'{:}(X[F_j]Y)(z), \Omega(\omega){:}g'$

where tl' and g' are the same as tl and g respectively, except that $x[i]$ and $y[j]$ are replaced with the corresponding $z[k]$.

PROOF.

$$\begin{aligned}&(tl{:}X(x), Y(y), \Omega(\omega){:}f_j(x,y)\wedge g)(I)\\ &=\{(x\circ y\circ\omega)[\mathrm{TL}]\mid x\in X(I)\wedge y\in Y(I)\wedge\omega\in\Omega(I)\wedge f_j(x,y)\wedge g\}\\ &=\{(z\circ\omega)[\mathrm{TL}']\mid z\in\{x\circ y\mid x\in X(I)\wedge y\in Y(I)\wedge f_j(x,y)\}\wedge\omega\in\Omega(I)\wedge g\}\\ &=\{(z\circ\omega)[\mathrm{TL}']\mid z\in(X[F_j]Y)(I)\wedge\omega\in\Omega(I)\wedge g\}\\ &=(tl'{:}(X[F_j]Y)(z),\Omega(\omega){:}g')(I)\quad\square\end{aligned}$$

Rule UQ. Universal Quantifier

from $tl{:}\Psi(\psi){:}(\forall X(x)f)\wedge g$
to $tl{:}\Psi(\psi){:}(\sim\exists X(x)(\sim f))\wedge g$

PROOF.

$$\begin{aligned}&(tl{:}\Psi(\psi){:}(\forall X(x)f)\wedge g)(I)\\ &=(tl{:}\Psi(\psi){:}(\forall x(\sim X(x)\vee f))\wedge g)(I)\\ &=(tl{:}\Psi(\psi){:}(\sim\exists x(X(x)\wedge\sim f))\wedge g)(I)\\ &=(tl{:}\Psi(\psi){:}(\sim\exists X(x)(\sim f))\wedge g)(I)\quad\square\end{aligned}$$

Rule T. Transformation

(1) *from* $tl{:}\Psi(\psi){:}(f \wedge g) \wedge h$
to $tl{:}\Psi(\psi){:}f \wedge (g \wedge h)$

(2) *from* $tl{:}\Psi(\psi){:}(f \vee g) \wedge h$
to $tl{:}\Psi(\psi){:}(f \wedge h) \vee (g \wedge h)$

(3) *from* $tl{:}\Psi(\psi){:}f \vee g$
to $(tl{:}\Psi(\psi){:}f)[+](tl{:}\Psi(\psi){:}g)$

PROOF. Omitted. □

Rule P. Projection

(1) *from* $tl{:}X(x){:}(\,)$
to $X[TL]$

(2) *from* $tl{:}X(x), Y(y), \Omega(\omega){:}(\,)$
to $tl'{:}(X[\,]Y)(z), \Omega(\omega){:}(\,)$

where tl' is identical to tl, except that $x[i]$ and $y[j]$ are replaced with the corresponding $z[k]$.

PROOF. Omitted. □

The following theorem has been established, but rewriting rules used in the proof are different from previous works [3, 20, 23].

THEOREM 1. *Within the relational calculus kernel, any calculus expression has an equivalent algebraic expression.*

PROOF. In the relational calculus kernel every expression is closed, but an alpha nesting may exist. For a nested alpha, if we translate the inner closed alpha first, it can be assumed that the alpha we have to translate is always closed and unnested.

Here we consider a closed unnested alpha. Every range can be translated to an algebraic expression, if necessary, using Rule R. Every possible form of a formula is to be rewritten by appropriate rules as follows: $f_s \wedge g$ (by Rule S), $(\exists\Omega(\omega)f) \wedge g$ (by Rule PEQ), $(\sim\exists\Omega(\omega)f) \wedge g$ (by Rule NEQ), $f_j \wedge g$ (by Rule J), $(\forall X(x)f) \wedge g$ (by Rule UQ), $(\sim\exists X(x)(\sim f)) \wedge g$ (by Rule UQ), $(f \wedge g) \wedge h$ (by Rule T), $(f \vee g) \wedge h$ (by Rule T), and $f \vee g$ (by Rule T). By making repeated use of the appropriate rule for each form, we eventually obtain a mixed expression whose qualifier has no formula; that is, always true. Then by applying Rule P we have the pure algebraic expression which is equivalent to the given calculus expression. □

4.3 Heuristic Rewriting Rules

Heuristic rules play an essential role in optimization through translation, finding idiomatic patterns in calculus expressions and rewriting them to succinct algebraic expressions in one stroke. The addition of the following heuristic rules has no effect on the correctness of Theorem 1, because they do not disturb the translation by basic rules.

As for the division, there are three kinds of heuristic rules. Rule JD is the most general and Rule D is the most specific and Rule DS is somewhere between them.

In general, more specific rules should be given higher priority. Hence the order applied is (1) Rule D, (2) DS, and (3) JD.

Both Rule EJ and EP play a remarkable part in the simplification of a result. Although the second and third cases of Rule EJ are much the same as the first, we present them so as to test Rule EJ in preference to Rules PS or NS.

In general, the semijoin has a smaller processing load than the join, so the semijoin should be used anywhere possible instead of the join. The positive semijoin maps directly to the existential quantifier; on the other hand, the negative semijoin maps to the negation of the existential quantifier. Rules PS and NS embody these mappings. Even though there are no negations of the existential quantifier in the original calculus expression, they may be generated by Rule UQ. Rule NS is valuable because without it the negation of the existential quantifier is rewritten by Rule NEQ, which is by no means simple.

Rule JD. Join and Division

from $tl_x{:}X(x){:}\forall Y(y)\exists Z(z)(f(x,z)\wedge g_e(y,z))$
to $tl_w{:}(((x,z_g){:}X(x),Z(z){:}f)[z_g/y_g]Y)(w){:}(\,)$

where $Y(I)\neq\phi$.

PROOF.

$$\begin{aligned}
&(tl_x{:}X(x){:}\forall Y(y)\exists Z(z)(f(x,z)\wedge g_e(y,z)))(I)\\
&=\{x[\mathrm{TL}]\mid x\in X(I)\wedge y\in Y(I)\wedge z\in Z(I)\wedge\forall y\exists z(f(x,z)\wedge g_e(y,z))\}\\
&=\{t[\mathrm{TL}]\mid t\in R(I)\wedge y\in Y(I)\wedge s\in R(I)\wedge\forall y\exists s(s[x]=t[x]\wedge g_e(y,s))\}\quad(1)\\
&=\{w[\mathrm{TL}]\mid w\in\{t[x]\mid t\in R(I)\wedge y\in Y(I)\wedge s\in R(I)\\
&\quad\wedge\forall y\exists s(s[x]=t[x]\wedge g_e(y,s))\}\}\\
&=\{w[\mathrm{TL}]\mid w\in\{t[x]\mid t\in R(I)\wedge s\in R(I)\wedge y\in Y(I)\\
&\quad\wedge\forall y\exists s(s[x]=t[x]\wedge s[z_g]=y[y_g])\}\}\quad(2)\\
&=\{w[\mathrm{TL}]\mid w\in(R[z_g/y_g]Y)(I)\}\\
&=(tl_w{:}(((x,z_g){:}X(x),Z(z){:}f)[z_g/y_g]Y)(w){:}(\,))(I)\quad\Box
\end{aligned}$$

Explanation. (1) $R(I)=\{(x,z_g)\mid x\in X(I)\wedge z\in Z(I)\wedge f(x,z)\}$. (2) Apply the definition of the division.

Rule DS. Division and Semijoin

from $tl_x{:}X(x){:}\forall Y(y)\exists Z(z)(f_e(x,z)\wedge g_e(y,z))$
to $tl_w{:}(X[\exists;F_e](Z[z_f,z_g][z_g/y_g]Y))(w){:}(\,)$

where $Y(I)\neq\phi$.

PROOF.

$$\begin{aligned}
&(tl_x{:}X(x){:}\forall Y(y)\exists Z(z)(f_e(x,z)\wedge g_e(y,z)))(I)\\
&=\{x[\mathrm{TL}]\mid x\in X(I)\wedge y\in Y(I)\wedge z\in Z(I)\wedge\forall y\exists z(f_e(x,z)\wedge g_e(y,z))\}\\
&=\{x[\mathrm{TL}]\mid x\in X(I)\wedge u\in\{t[z_f]\mid t\in Z(I)\wedge s\in Z(I)\wedge y\in Y(I)\\
&\quad\wedge\forall y\exists s(s[z_f]=t[z_f]\wedge g_e(y,s))\}\wedge f_e(x,u)\}\\
&=\{x[\mathrm{TL}]\mid x\in X(I)\wedge u\in\{t[z_f]\mid t\in Z(I)\wedge s\in Z(I)\wedge y\in Y(I)\\
&\quad\wedge\forall y\exists s(s[z_f]=t[z_f]\wedge s[z_g]=y[y_g])\}\wedge f_e(x,u)\}\\
&=\{x[\mathrm{TL}]\mid x\in X(I)\wedge u\in\{t[z_f]\mid t\in Z[z_f,z_g](I)\wedge s\in Z[z_f,z_g](I)\\
&\quad\wedge y\in Y(I)\wedge\forall y\exists s(s[z_f]=t[z_f]\wedge s[z_g]=y[y_g])\}\wedge f_e(x,u)\}\quad(1)\\
&=\{x[\mathrm{TL}]\mid x\in X(I)\wedge u\in(Z[z_f,z_g][z_g/y_g]Y)(I)\wedge f_e(x,u)\}\quad\Box\quad(2)
\end{aligned}$$

Explanation. (1) Apply the division definition. (2) Apply the positive semijoin definition.

Rule D. Division

from $tl_x{:}X(x){:}\forall Y(y)\exists Z(z)(f_e(x, z) \wedge g_e(y, z))$
to $tl_w{:}(Z[z_f, z_g][z_g/y_g]Y)(w){:}(\,)$

where $Y(I) \neq \phi$, $x_f \supseteq \mathrm{tl}_x$ and $Z = (X \text{ or } X[H_s])$. H_s denotes the selection.

PROOF.

$$(tl_x{:}X(x){:}\forall Y(y)\exists Z(z)(f_e(x, z) \wedge g_e(y, z)))(I)$$
$$= (tl_w{:}(X[\exists;\, F_e](Z[z_f, z_g][z_g/y_g]Y))(w){:}(\,))(I) \qquad (1)$$
$$= (tl_x{:}X(x),\, (Z[z_f, z_g][z_g/y_g]Y)(u){:}f_e(x, u))(I)$$
$$= (tl_w{:}(Z[z_f, z_g][z_g/y_g]Y)(w){:}(\,))(I) \quad \square \qquad (2)$$

Explanation. (1) Apply Rule DS. (2) Since the division is a kind of selection, we can apply Rule EJ.

Rule EJ. Elimination of Join

(1) *from* $tl{:}X(x),\ Y(y),\ \Omega(\omega){:}f_e(x, y) \wedge g$
to $tl'{:}Y(y),\ \Omega(\omega){:}g'$
(2) *from* $tl{:}X(x),\ \Omega(\omega){:}(\exists Y(y)(f_e(x, y) \wedge g_1)) \wedge g_2$
to $tl'{:}Y(y),\ \Omega(\omega){:}g'$
(3) *from* $tl{:}Y(y),\ \Omega(\omega){:}(\exists X(x)(f_e(x, y) \wedge g_1)) \wedge g_2$
to $tl'{:}Y(y),\ \Omega(\omega){:}g'$

where $g = g_1 \wedge g_2$, $X[x_f](I) \supseteq Y[y_f](I)$, $x_f \supseteq x_{tl}$, $x_f \supseteq x_g$,

tl' and g' are the same as tl and g, respectively, except that $x[i]$ is replaced with $y[j]$, which are made equal in f_e.

PROOF. We consider only the first case because the second and third are reducible to the first by applying Rule PEQ. The conditions $x_f \supseteq x_{tl}$ and $x_f \supseteq x_g$ guarantee that with variables x and y the attributes which influence the expression are restricted to x_f and y_f. Now consider any atomic formula, say $x[i] = y[j]$, from $f_e(x, y)$. The condition $X[x_f](I) \supseteq Y[y_f](I)$ guarantees that the value set of $x[i]$ always includes the value set of $y[j]$. Hence a value of $x[i]$ is discarded if it is not included in the value set of $y[j]$. That is, the range $X(I)$ is no longer necessary. $\square$

Comment. Since it is not easy to check whether the condition $X[x_f](I) \supseteq Y[y_f](I)$ holds, we recommend replacing it with a more practical condition: check whether $Y[y_f](I)$ has been derived from $X[x_f](I)$ or not.

Rule EP. Elimination of Projection

from $(x[1], \ldots, x[m]){:}X(x){:}(\,)$
to X

where m is $\deg(X)$.

PROOF. Omitted. □

Rule PS. Positive Semijoin

from $tl{:}X(x),\ \Omega(\omega){:}\exists Y(y)(f_j(x, y) \wedge g(y)) \wedge h$
to $tl{:}(X[\exists;\ F_j](y{:}Y(y){:}g))(x),\ \Omega(\omega){:}h$

PROOF.

$$\begin{aligned}
&(tl{:}X(x),\ \Omega(\omega{:}\exists Y(y)(f_j(x, y) \wedge g(y)) \wedge h)(I)\\
&= \{(X^x \circ \omega))[\mathrm{TL}] \mid x \in X(I) \wedge \omega \in \Omega(I)\\
&\quad \wedge \{y \mid y \in Y(I) \wedge f_j(x, y) \wedge g(y)\} \neq \phi \wedge h\}\\
&= \{(X^x \circ \omega)[\mathrm{TL}] \mid x \in X(I)\\
&\quad \wedge \{u \mid u \in \{y \mid y \in Y(I) \wedge g(y)\} \wedge f_j(x, u)\} \neq \phi \wedge \omega \in \Omega(I) \wedge h\} \qquad (1)\\
&= \{(x \circ \omega)[\mathrm{TL}] \mid x \in \{x' \mid x' \in X(\mathrm{I})\\
&\quad \wedge u \in \{y \mid y \in Y(I) \wedge g(y)\} \wedge f_j(x', u)\} \wedge \omega \in \Omega(I) \wedge h\}\\
&= \{(x \circ \omega)[\mathrm{TL}] \mid x \in (X[\exists;\ F_j](y{:}Y(y){:}g))(I) \wedge \omega \in \Omega(I) \wedge h\}\\
&= (tl{:}(X[\exists;\ F_j](y{:}Y(y){:}\ g))(x),\ \Omega(\omega){:}h)(I) \quad \square
\end{aligned}$$

Explanation. (1) Apply the positive semijoin definition.

Rule NS. Negative Semijoin

from $tl{:}X(x),\ \Omega(\omega){:}{\sim}\exists Y(y)(f_j(x, y) \wedge g(y)) \wedge h$
to $tl{:}(X[{\sim}\exists;\ F_j](y{:}Y(y){:}g))(x),\ \Omega(\omega){:}h$

PROOF. Much the same as above. □

5. TRANSLATION OF AGGREGATE FUNCTIONS

5.1 Overview

The introduction of aggregate functions produces open alphas, which seriously complicate translations. An expression having an aggregate function to be translated is called a *mother aggregate expression.* The basic idea is two-phase rewriting through a *standard aggregate expression* (defined below). The idea is illustrated in Figure 3. At the first phase, a standard aggregate expression is generated from a mother aggregate expression and these two expressions are connected by a *linkage formula.* In the second phase, the standard aggregate expression is translated to an algebraic expression using either the general aggregate formation or the original aggregate formation. Of course, when a mother aggregate expression is already a standard aggregate expression, only the second phase suffices.

A standard aggregate expression satisfies the following conditions: An aggregate function should appear only in its target list, other terms in the target list should make up the exact set of occurrences of free variables for the alpha to be aggregated, and its qualifier should have no formula.

A linkage formula is an equijoin-type atomic formula or equijoin-type formula that equates the corresponding attributes between a mother aggregate expression and the standard aggregate expression.

The following is a typical translation of an aggregate function.

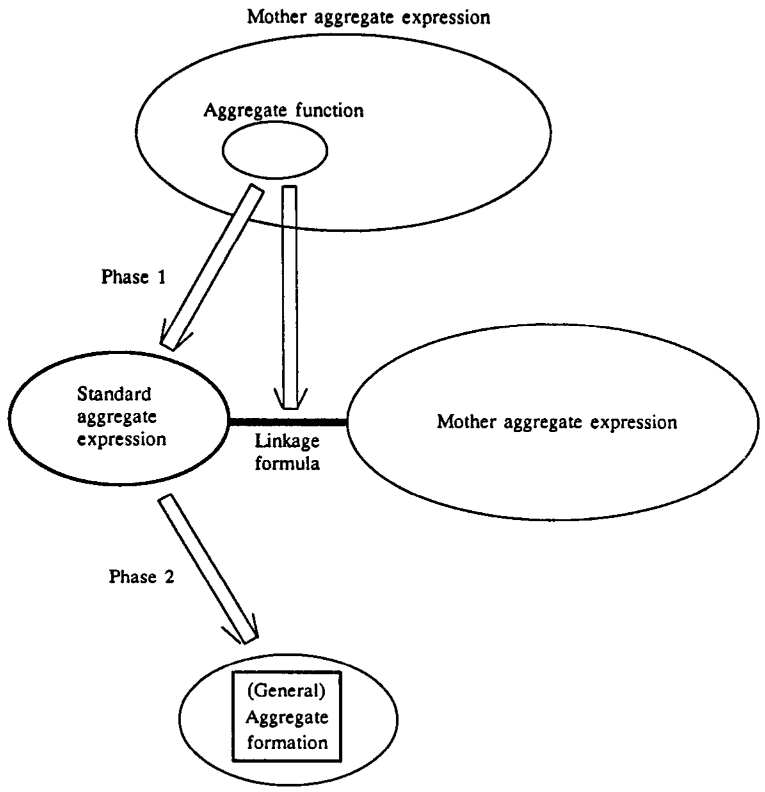
Mother aggregate expression
Aggregate function
Phase 1
Standard aggregate expression
Linkage formula
Mother aggregate expression
Phase 2
(General) Aggregate formation

Fig. 3. General flow of aggregate function translation.

Example 3. For each employee, find the name, salary and the average salary of his (or her) department.

(u[1], u[2], avg[2](v:emp(v):v[4] = u[4])):emp(u):( )

In the first phase (Rule SAT applied), the mother aggregate expression is rewritten as follows:

(u[1], u[2], s[2]):emp(u), S(s):s[1] = u[4]
S = (u[4], avg[2](v:emp(v):v[4] = u[4])):emp(u):( )

where S is the standard aggregate expression and $s[1] = u[4]$ is the linkage formula. In S, the alpha to be aggregated is $(v\text{:emp}(v)\text{:}v[4]) = u[4])$ and the occurrence of free variable u is $u[4]$. In the second phase (Rules EK2 and EJ applied), S is translated using an aggregate formation, and the mother aggregate expression is translated into the following mixed expression:

(u[1], u[2], s[2]):emp(u), (emp⟨#4; avg[2]⟩) (s):s[1] = u[4].

From now on, the kernel translation suffices.

For this translation there are four basic rewriting rules and four heuristic ones:

Basic: SAT (standardization of aggregate function in a target list),
SAQ (standardization of aggregate function in a qualifier),
RS (resolution of standard aggregate expression),
CAF (closed aggregate function),

Heuristic: EK1 (elimination of Klug's correction type 1),
EK2 (elimination of Klug's correction type 2),
EM (elimination of merge),
EK1M (elimination of both Klug's correction type 1 and merge).

Figure 4 illustrates in detail how each rewriting rule works in the total view of the aggregate function translation. Basic rewriting rules are represented by hatched arrows and heuristic rewriting rules by black arrows. Note that Rule EM works in more general cases than Rule EJ (Elimination of Join) does. Similarly, Rule EK1M works in more general cases than a simple combination of Rules EK1 and EJ does.

5.2 Basic Rewriting Rules

The first phase is executed by either Rule SAT or SAQ, depending on the location of an aggregate function. In either case both a standard aggregate expression and a linkage formula are generated. In SAT the standard aggregate expression is added as a range formula of the mother aggregate expression. On the other hand, in SAQ it is added as an existentially quantified range. The second phase is done by Rule RS, where a standard aggregate expression is translated in a rather straightforward way by using the general aggregate formation. Klug's correction, described in Section 2.3, is taken into consideration in this phase. In addition, when an alpha to be aggregated is closed, the rewriting is trivially simple (Rule CAF).

Rule SAT. Standardization of Aggregate Function in a Target list

from $\quad \text{tl}, \text{agg}(Z)) : \Psi(\psi), \Omega(\omega) : (\,)$
to $\quad (\text{tl}, s[n+1]) : \Psi(\psi), \Omega(\omega), S(s) : f_e(s, \psi)$

where $Z = Z(\psi)$, $S = (\psi_z, \text{agg}(Z)) : \Psi(\psi) : (\,)$, $f_e(s, \psi) = (s[1] = \psi_z[1] \wedge \cdots \wedge s[n] = \psi_z[n])$ and $n = \text{len}(\psi_z)$.

PROOF.

$$((\text{tl}, \text{agg}(Z)) : \Psi(\psi), \Omega(\omega) : (\,))(\text{I})$$
$$= \{(\psi \circ \omega)[\text{TL}] \circ \text{agg}(Z(\psi)) \mid \psi \in \Psi(\text{I}) \wedge \omega \in \Omega(\text{I})\}$$
$$= \{(\psi \circ \omega)[\text{TL}] \circ \text{agg}(Z(\psi')) \mid \psi \in \Psi(\text{I}) \wedge \omega \in \Omega(\text{I}) \wedge \psi' \in \Psi(\text{I}) \wedge \psi_z = \psi'_z\}$$
$$= \{(\psi \circ \omega)[\text{TL}] \circ s[n+1]) \mid \psi \in \Psi(\text{I}) \wedge \omega \in \Omega(\text{I}) \wedge s \in \{(\psi' z \circ \text{agg}(Z(\psi'))) \mid \psi' \in \Psi(\text{I})\} \wedge f_e(s, \psi)\}$$
$$= ((\text{tl}, s[n+1]) : \Psi(\psi), \Omega(\omega), S(s) : f_e(s, \psi))(\text{I}) \quad \square$$

Rule SAQ. Standardization of Aggregate Function in a Qualifier

from $\quad \text{tl} : \Psi(\psi), \Omega(\omega) : (\text{agg}(Z)\theta t) \wedge g$
to $\quad \text{tl} : \Psi(\psi), \Omega(\omega) : (\exists S(s)(s[n+1]\theta t \wedge f_e(s, \psi))) \wedge g$

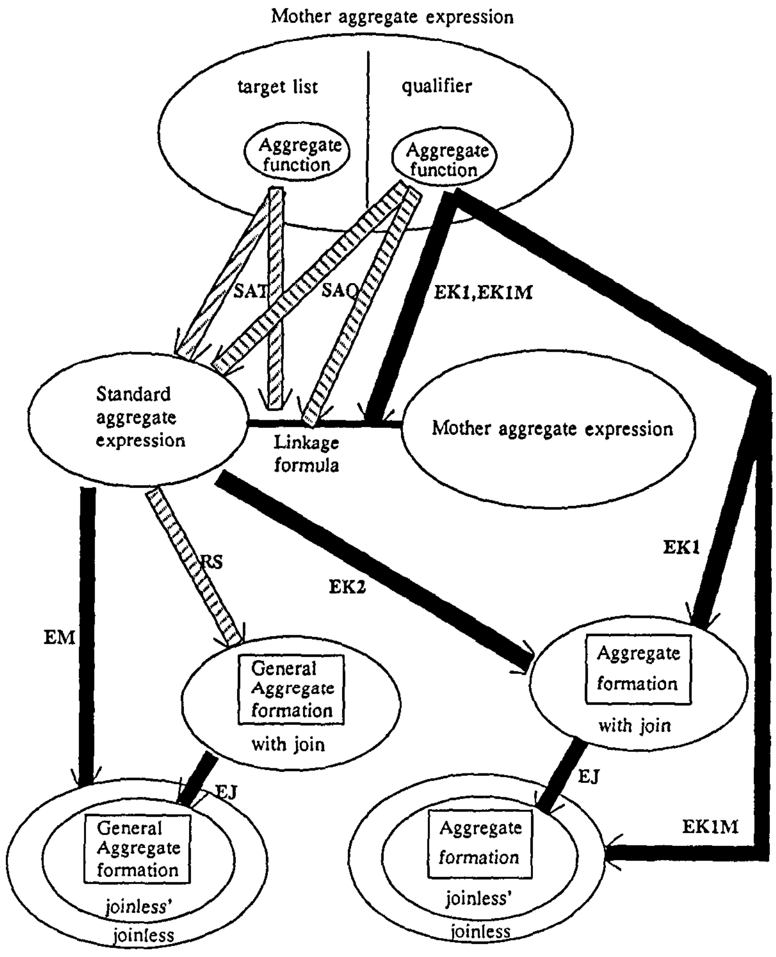

Fig. 4. Detail flow of aggregate function translation.

where $Z = Z(\psi)$, $S = (\psi z, \text{agg}(Z)){:}\Psi(\psi){:}(\,)$, $f_e(s, \psi) = (s[1] = \psi_z[1] \cdots \wedge s[n] = \psi_z[n])$ and $n = \text{len}(\psi_z)$.

PROOF.

$$\begin{aligned}
&(\text{tl}{:}\Psi(\psi), \Omega(\omega){:}(\text{agg}(Z)\theta t) \wedge g)(\text{I}) \\
&= \{(\psi \circ \omega)[\text{TL}] \mid \psi \in \Psi(\text{I}) \wedge \omega \in \Omega(\text{I}) \wedge (\text{agg}(Z(\psi))\theta t) \wedge g\} \\
&= \{(\psi \circ \omega)[\text{TL}] \mid \psi \in \Psi(\text{I}) \wedge \omega \in \Omega(\text{I}) \wedge \psi' \in \Psi(\text{I}) \\
&\quad \wedge \psi_z = \psi_z' \wedge (\text{agg}(Z(\psi'))\theta t) \wedge g\} \\
&= \{(\psi \circ \omega)[\text{TL}] \mid \psi \in \Psi(\text{I}) \wedge \omega \in \Omega(\text{I}) \wedge s \in \text{S}(\text{I}) \qquad (1) \\
&\quad \wedge f_e(s, \psi) \wedge s[n+1]\theta t \wedge g\} \\
&= (\text{tl}{:}\Psi(\psi), \Omega(\omega){:}(\exists \text{S}(s)(s[n+1]\theta t \wedge f_e(s, \psi))) \wedge g)(\text{I}) \quad \Box
\end{aligned}$$

Explanation. (1) We can replace ψ' with s because ψ' works to restrict ψ and ω.

Example 4. Get the name of employee whose salary is higher than the average of his (or her) department.

(u[1]):emp(u):u[2] > avg[2](v:emp(v):v[4] = u[4])

By applying Rule SAQ, we get the following expression:

(u[1]):emp(u):∃S(s) (u[2] > s[2] ∧ s[1] = u[4])

where

S = (u[4], avg[2](v:emp(v):v[4] = u[4])):emp(u):( )

Rule RS. Resolution of Standard Aggregate Expression

from $(\mathrm{tl}_1, \mathrm{agg}(\mathrm{tl}_2\!:\!\Upsilon(\upsilon)\!:\!h))\!:\!\Psi(\psi)\!:\!(\,)$
to $((\mathrm{tl}_1, \mathrm{tl}_2)\!:\!\Psi(\psi), \Upsilon(\upsilon)\!:\!h)[(\#1, \ldots, \#n)/(\mathrm{TL}_1);\ \mathrm{agg}']\Psi$

where $n = \mathrm{len}(\mathrm{tl}_1)$, agg′ is identical to agg except that the attribute number to be aggregated is increased by n and $\psi_h = \mathrm{tl}_1$ (the condition of a standard aggregate expression).

PROOF.

$$((\mathrm{tl}_1, \mathrm{agg}(\mathrm{tl}_2\!:\!\Upsilon(\upsilon)\!:\!h))\!:\!\Psi(\psi)\!:\!(\,))(\mathrm{I})$$
$$= \{\psi[\mathrm{TL}_1] \circ \mathrm{agg}(\{\upsilon[\mathrm{TL}_2] \mid \upsilon \in \Upsilon(\mathrm{I}) \wedge h\}) \mid \psi \in \Psi(\mathrm{I})\}$$
$$= \{\psi[\mathrm{TL}_1] \circ \mathrm{agg}'(\{s \mid s \in \mathrm{S}(\mathrm{I}) \wedge s[1, \ldots, n] = \psi[\mathrm{TL}_1]\}) \mid \psi \in \Psi(\mathrm{I})\} \qquad (1)$$
$$= \{\psi[\mathrm{TL}_1] \circ f \mid \psi \in \Psi(\mathrm{I}) \wedge f = \mathrm{agg}'(\{s \mid s \in \mathrm{S}(\mathrm{I})$$
$$\wedge\ s[1, \ldots, n] = \psi[\mathrm{TL}_1]\})\}$$
$$= \mathrm{S}(\mathrm{I})[(\#1, \ldots, \#n)/(\mathrm{TL}_1);\ \mathrm{agg}']\Psi(\mathrm{I}) \qquad (2)$$
$$= (((\mathrm{tl}_1, \mathrm{tl}_2)\!:\!\Psi(\psi), \Upsilon(\upsilon)\!:\!h)[(\#1, \ldots, \#n)/(\mathrm{TL}_1);\ \mathrm{agg}']\Psi)(\mathrm{I}) \quad \Box$$

Explanation. (1) Since $\psi_h = \mathrm{tl}_1$, we can replace $\Upsilon(\mathrm{I})$ with the following $\mathrm{S}(\mathrm{I})$.

$$\mathrm{S}(\mathrm{I}) \equiv \{\psi[\mathrm{TL}_1] \circ \upsilon[\mathrm{TL}_2] \mid \psi \in \Psi(\mathrm{I}) \wedge \upsilon \in \Upsilon(\mathrm{I}) \wedge h\}$$

(2) Apply the definition of the general aggregate formation.

Example 5. For each floor get the number of departments located on a higher floor.

(u[2], count((v[1]):loc(v):v[2] > u[2])):loc(u):( )

By applying Rule RS we get the following expression, which does not have an aggregate function anymore.

((u[2], v[1]):loc(u), loc(v):v[2] > u[2])[#1/#2; count] loc

Rule CAF. Closed Aggregate Function

from agg(Z)
to Z ⟨agg⟩

where Z is closed.

PROOF. Omitted. □

Now we extend Theorem 1 to include an aggregate function. Klug [17] showed that Theorem 2 holds, but the proof is quite different from the following.

THEOREM 2. *Any calculus expression has an equivalent algebraic expression.*

PROOF. Within the relational calculus kernel the above claim holds, which is shown by Theorem 1. Hence, suffice it to say that every calculus expression having an aggregate function is reducible to a mixed expression that consists of both the general aggregate formation and closed alphas.

First we consider a calculus expression having only one aggregate function. Since an aggregate function is a term, it appears in either a target list or a qualifier. In either case, the calculus expression is reduced to the mother aggregate expression, which has both the standard aggregate expression and the linkage formula, by applying Rule SAT or SAQ. Now an aggregate function appears in a target list of the standard aggregate expression. Applying Rule RS, we have an equivalent mixed expression which consists of both the general aggregate formation and closed alphas. When there is more than one aggregate function, a repeated application of the above procedure will solve the problem. □

5.3 Heuristic Rewriting Rules

Heuristic rewriting rules provide for more succinct rewriting when certain conditions are satisfied. The addition of the following heuristic rules has no effect on the correctness of Theorem 2, because they do not disturb the translation by basic rules.

Klug's correction makes up for dangling groups, which are destined to disappear because they have no tuples to be aggregated (i.e., the alpha to be aggregated is empty). There are two cases for the elimination of Klug's correction. One is when an empty set has no effect, and the other is when an empty set does not occur. The former corresponds to Rule EK1 and the latter to Rule EK2. Both rules make it possible to use an aggregate formation instead of a general one in the second (resolution) phase. Rule EK1 is limited to a special type of count function and stretches over two phases: detection in the first phase and application in the second phase. We use the temporary function name "pcount" to indicate that Rule EK1 is applicable.

When the qualifier of the alpha to be aggregated is a join-type formula between free variables and closed ones, resolution of a standard aggregate expression can be made much simpler; that is, the merge (join) of two alphas is unnecessary. Rule EM corresponds to this case. The join elimination is possible because the general aggregate formation substantially joins two alphas, discarding the dangling tuples that might remain by the elimination of a join.

Similary, when Rule EK1 is applicable and the qualifier of the alpha to be aggregated is a join-type formula between free variables and closed ones, resolution of a standard aggregate expression can be made much simpler; again, the join of two alphas is unnecessary. Rule EK1M corresponds to this case. The rule is more general than a simple combination of Rules EK1 and EJ. The join elimination is possible because the linkage formula f_e discards the dangling tuples that might remain by the elimination of a join.

Rule EK1. Elimination of Klug's Correction Type 1

from $\mathrm{tl}:\Psi(\psi), \Omega(\omega):(\mathrm{count}(Z)\theta c) \wedge g$
to $\mathrm{tl}:\Psi(\psi), \Omega(\omega):(\exists \mathrm{S}(s)(s[n+1]\theta c \wedge f_e(s, \psi))) \wedge g$

where $Z = (\mathrm{tl}_2:\Upsilon(v):h(\psi, v))$, $\mathrm{S} = ((\psi_h, \mathrm{tl}_2):\Psi(\psi), \Upsilon(v):h(\psi, v)) \langle \#1, \ldots, \#n; \mathrm{count}\rangle$, θc satisfies > 0 (ex. >2, $>= 4$, $=3$, $\neq 0$), $f_e(s, \psi) = (s[1] = \psi_h[1] \wedge \ldots \wedge s[n] = \psi_h[n])$ and $n = \mathrm{len}(\psi_h)$.

PROOF.

$$(\mathrm{tl}:\Psi(\psi), \Omega(\omega):(\mathrm{count}(Z)\theta c) \wedge g)(\mathrm{I})$$
$$= (\mathrm{tl}:\Psi(\psi), \Omega(\omega):(\exists \mathrm{S}(s)(s[n+1]\theta c \wedge f_e(s, \psi))) \wedge g)(\mathrm{I}) \qquad (1)$$

where

$$\mathrm{S} = (\psi_h, \mathrm{pcount}(Z)):\Psi(\psi):(\,)$$
$$= (\psi_h, \mathrm{pcount}(\mathrm{tl}_2:\Upsilon(v):h(\psi, v))):\Psi(\psi):(\,)$$
$$= ((\psi_h, \mathrm{tl}_2):\Psi(\psi), \Upsilon(v)):h(\psi, v)) \langle \#1, \ldots, \#n; \mathrm{count}\rangle \quad \square \qquad (2)$$

Explanation. (1) Apply Rule SAQ. (2) Apply Rule RS paying attention to pcount.

Example 6. List items which are sold at more than or equal to three floors.

(u[2]):sales(u):count((v[2]):loc(v):∃sales(w) (w[1] = v[1] ∧ w[2] = u[2])) >= 3

By applying Rule EK1, we get the following expression:

(u[2]):sales(u):∃S(s)(s[2] >= 3 ∧ s[1] = u[2])
S = ((u[2], v[2]):sales(u), loc(v):∃sales(w)(w[1] = v[1] ∧ w[2] = u[2])) ⟨#1; count⟩

Rule EK2. Elimination of Klug's Correction Type 2

from $(\mathrm{tl}_1, \mathrm{agg}(\mathrm{tl}_2:\Upsilon(v):h_j)):\Psi(\psi):(\,)$
to $((\mathrm{tl}_1, \mathrm{tl}_2):\Psi(\psi), \Upsilon(v):h_j) \langle \#1, \ldots, \#n; \mathrm{agg}'\rangle$

where $h_j = h_j(\psi, v) = \ldots \wedge y[i]\theta u[i] \wedge \ldots$, θ is among $\{=, <=, >=\}$, $\Psi = (\Upsilon$ or $\Upsilon[F_s])$, $n = \mathrm{len}(\mathrm{tl}_1)$, agg′ is like agg except that the attribute number to be aggregated is increased by n and $\psi_h = \mathrm{tl}_1$ (the condition of a standard aggregate expression).

PROOF. Both the range condition $\Psi = (\Upsilon$ or $\Upsilon[F_s])$ and formula condition h_j guarantee that any group tl_1 has at least one tuple and Klug's correction is not needed. Hence the general aggregate formation in Rule RS can be replaced with the original aggregate formation. □

Example 7. For each floor get the number of departments located either on that floor or on a higher floor.

(u[2], count((v[1]):loc(v):v[2] >= u[2])):loc(u):( )

Note that this example is slightly different from Example 5. Rule EK2 brings the following:

((u[2], v[1]):loc(u), loc(v):v[2] >= u[2]) ⟨#1; count⟩

Rule EM. Elimination of Merge

from $(\text{tl}_1, \text{agg}(\text{tl}_2{:}\Upsilon(v){:}h_e \wedge g)){:}\Psi(\psi){:}(\,)$
to $((v_h, \text{tl}_2){:}\Upsilon(v){:}g)[(\#1, \ldots, \#n)/(\text{TL}_1); \text{agg}']\Psi$

where $h_e = h_e(\psi, v) = \ldots \wedge y[i] = u[j] \wedge \ldots, g = g(v)$, $n = \text{len}(\text{tl}_1)$, agg′ is like agg except that the attribute number to be aggregated is increased by n and $\psi_h = \text{tl}_1$ (the condition of a standard aggregate expression).

PROOF. Compared with Rule RS, suffice it to say that the following holds under the present condition,

$$((\text{tl}_1, \text{tl}_2){:}\Psi(\psi), \Upsilon(v){:}h_e(\psi, v) \wedge g(v)) = ((v_h, \text{tl}_2){:}\Upsilon(v){:}g(v)). \quad (1)$$

From the condition $\psi_h = \text{tl}_1$ and h_e, we get $\text{tl}_1 = v_h$. Hence the target list becomes dependent only on v as follows:

$$(v_h, \text{tl}_2){:}\Psi(\psi), \Upsilon(v){:}h_e(\psi, v) \wedge g(v).$$

If we eliminate the range Ψ and formula $h_e(\psi, v)$ from the above expression, dangling tuples v that will be detected in $h_e(\psi, v)$ and then discarded may remain; however, they will be neglected in the general aggregate formation. Hence, equation (1) holds under the present condition. □

Example 8. For each item currently supplied, get the number of departments selling the item.

```
(u[3], count((v[1]):sales(v):v[2] = u[3])):supply(u):( )
```

By applying Rule EM, we get the following expression:

```
((v[2], v[1]):sales(v):( ))[#1/#3; count]supply
```

Rule EK1M. Elimination of both Klug's Correction Type 1 and Merge

from $\text{tl}{:}\Psi(\psi), \Omega(\omega){:}(\text{count}(Z)\theta c) \wedge g$
to $\text{tl}{:}\Psi(\psi), \Omega(\omega){:}(\exists \text{S}(s)(s[n+1]\theta c \wedge f_e(s, \psi))) \wedge g$

where $Z = (\text{tl}_2{:}\Upsilon(v){:}h_e \wedge q)$, $h_e = h_e(\psi, v') = \ldots \wedge y[i] = u[j] \wedge \ldots, q = q(v)$, $\text{S} = (v_h, \text{tl}_2){:}\Upsilon(v{:}q)\ \langle \#1, \ldots, \#n;\ \text{count}\rangle$, θc satisfies >0, $f_e(s, \psi) = (s[1] = \psi_h[1] \wedge \ldots \wedge s[n] = \psi_h[n])$, and $n = \text{len}(\psi_h)$.

PROOF. Much the same as above. □

6. EXTENSIONS FOR NULL VALUES

So far null values have not been considered, but we cannot neglect them because real world information is often incomplete and because fun[A](ϕ) generates them, as shown in Section 2.3. Hence the domain of each attribute includes the null value. After introducing the SQL-type null value, this section extends relational calculus, relational algebra, and the translation method. The extension of relational calculus was motivated by Bultzingsloewen [1], but the total extensions are quite different from his work because of the dissimilarity of global translation strategies. Moreover, note that we concern ourselves only with the null value whose interpretation is "value at present unknown."

6.1 SQL-Type Null Values

Although the treatment of null values still remains controversial, at least at present the SQL feature [12] of null values can be considered a reasonable compromise. Hence we adopt SQL-type null values, the treatments of which are summarized below.

(1) *Extended comparison predicate.* The special comparison operator $\equiv$ is added to the set of comparison operators θ. The truth value of a comparison $x \equiv y$ yields true if $x = y$ is true or both null values, and false otherwise. The other comparison operators are evaluated as unknown if at least one of the values compared is a null value.

(2) *Three-valued logic.* When evaluating a search condition in SQL (a formula in relational calculus), we employ the three-valued logic, where ~(unknown) is unknown, false $\wedge$ unknown is false, true $\wedge$ unknown is unknown, unknown $\wedge$ unknown is unknown, true $\vee$ unknown is true, false $\vee$ unknown is unknown, unknown $\vee$ unknown is unknown, and so on.

(3) *Where-clause truncation.* A "WHERE ⟨search condition⟩" in SQL (a qualifier in relational calculus) results in a relation for each tuple of which the final value of the ⟨search condition⟩ is true. A valulation that produces false or unknown as the final value of the condition is discarded here. Such discarding is called a where-clause truncation. A *truncation* $\perp$ can be defined as follows: $\perp$(true) = true, $\perp$(false) = false, and $\perp$(unknown) = false.

(4) *Exclusion in aggregation.* Null values are excluded in the computation of all aggregate functions except count. Hence, the result of count is the cardinality of the set.

(5) *Exceptional treatments.* Although a comparison null = null is unknown, there are exceptional cases where null = null is true.[1] Such exceptions take place only in the contexts of duplicates (DISTINCT), group-by, and ordering (ORDER BY).

Since the above treatments are mostly calculus-oriented, they can be applied to relational calculus in a straightforward manner. For relational algebra, however, the following set-oriented treatments of null values should be added.

(a) *Extended membership condition.* The membership condition ⟨element⟩ $\in$ ⟨set⟩ is the basis of set theory. If we choose the usual definition [6], which states $e \in \{e1, e2, e3\}$ is equivalent to the expression $e = e1 \vee e = e2 \vee e = e3$, we have no reason to admit that a tuple [1, null] is a member of a relation {[1, null], [2, a]} because null = null is unknown. In set-oriented representation we usually denote $t \in R$, which means every tuple t is selected in turn from the relation R whether t has a null value or not. Hence we should extend the definition of the membership condition as follows:

$$e \in \{e_1, \ldots, e_n\} \leftrightarrow e \equiv e_1 \vee \ldots \vee e \equiv e_n.$$

[1] Obviously, this is a contradiction in dealing with a comparison null = null. Confusion in the translation, however, can be avoided if we treat exceptions in the same manner throughout the translation. Apparent contradiction can be solved by replacing = with $\equiv$ in the definition of exceptions. For example, tuples t and t' are said to be duplicates of each other if and only if, for every attribute i, $t[i] \equiv t'[i]$.

This can be regarded as a basic treatment of null values in sets, which includes the duplicates exception.

(b) *Set formation truncation.* A set formation of {⟨list⟩ | ⟨condition⟩} is always used in both the definitions of algebraic operations and proofs of translation rules. It should be made clear that the above set formation results in a set of the ⟨list⟩, which makes the value of the ⟨condition⟩ true. A ⟨list⟩ that produces false or unknown as the value of the ⟨condition⟩ is discarded. This kind of discarding is a set-oriented counterpart of where-clause truncation.

6.2 Extensions of Relational Calculus and Relational Algebra

The above calculus-oriented null value treatments can be applied to relational calculus. Moreover, the set-oriented null value treatments should be added to relational algebra. In addition, most algebraic operations are under the influence of null values and should be extended, as described below. Finally, a new operation is added to select only tuples having nonnull values in the specified attributes.

Selection

$$(e[F_s])(\mathrm{I}) = \{t \mid t \in e(\mathrm{I}) \wedge f_s(t)\}$$

Explanation. Although the formal definition remains unchanged, the formula f_s is evaluated under the three-valued logic, just like a formula in relational calculus.

Join, semijoin

$$(e_1[F_j]e_2)(\mathrm{I}) = \{\mathrm{t} \circ \mathrm{s} \mid t \in e_1(\mathrm{I}) \wedge s \in e_2(\mathrm{I}) \wedge f_j(t, s)\}$$

$$(e_1[\exists;\ F_j]e_2)(\mathrm{I}) = \{t \mid t \in e_1(\mathrm{I}) \wedge s \in e_2(\mathrm{I}) \wedge f_j(t, s)\}$$

$$(e_1[\sim\exists;\ F_j]e_2)(\mathrm{I}) = \{t \mid t \in e_1(\mathrm{I}) \wedge \{s \mid s \in e_2(\mathrm{I}) \wedge f_j(t, s) = \{1, \tfrac{1}{2}\}\} = \phi\}$$

Explanation. The formula f_j is evaluated under the three-valued logic, just like a formula in relational calculus. The formal definitions of join and positive semijoin remain unchanged. Special attention, however, should be paid to negative semijoin, because it has a negation sign, which plays a critical role in the extension caused by null values. An additional feature of this operation is to exclude a tuple t, which makes the formula f_j unknown. The truth value of unknown is denoted by $\frac{1}{2}$.

Division

$$\begin{aligned}(e_1[A_1/A_2]e_2)(\mathrm{I}) = \{t[\sim A_1] \mid\ & t \in e_1(\mathrm{I}) \wedge s \in e_1(\mathrm{I}) \wedge r \in e_2(\mathrm{I}) \\ & \wedge \forall r \exists s(s[\sim A_1] \equiv t[\sim A_1] \wedge s[A_1] = r[A_2])\}\end{aligned}$$

Explanation. First of all, the group comparison in the definition must obey the group-by exception:

$$\begin{aligned}(e_1[A_1/A_2]e_2)(\mathrm{I}) = \{t[\sim A_1] \mid\ & t \in e_1(\mathrm{I}) \wedge \{s[A_1] \mid s \in e_1(\mathrm{I}) \\ & \wedge s[\sim A_1] \equiv t[\sim A_1]\} \supseteq \{r[A_2] \mid r \in e_2(\mathrm{I})\}\}.\end{aligned}$$

This definition uses the inclusion, which is based on the extended membership condition, that is, for any set S_1 and S_2, $S_1 \supseteq S_2 \leftrightarrow \forall e(e \in S_2 \rightarrow e \in S_1)$. Hence the calculus-oriented counterpart of this definition is

$$(e_1[A_1/A_2]e_2)(\mathrm{I}) = \{t[{\sim}A_1] \mid t \in e_1(\mathrm{I}) \wedge s \in e_1(\mathrm{I}) \wedge r \in e_2(\mathrm{I}) \wedge \forall r \exists s(s[{\sim}A_1] \equiv t[{\sim}A_1] \wedge s[A_1) \equiv r[A_2])\}.$$

These two definitions are equvalent to each other even when null values are introduced. However, we adopt the following, since it will be used in the translation more frequently:

$$(e_1[A_1/A_2]e_2)(\mathrm{I}) = \{t[{\sim}A_1] \mid t \in e_1(\mathrm{I}) \wedge s \in e_1(\mathrm{I}) \wedge r \in e_2(\mathrm{I}) \wedge \forall r \exists s(s[{\sim}A_1] \equiv t[{\sim}A_1] \wedge s[A_1] = r[A_2])\}.$$

The difference is only the comparator of the last predicate; hence, if $r[A_2]$ has a null value, the above two definitions may produce different results. For example, let the relation R be $\{[a, 1], [a, \text{null}], [b, 2]\}$ and the relation S be $\{[1], [\text{null}]\}$, and consider $R[\#2/\#1]$S. The former definition produces $\{[a]\}$, but the latter an empty relation.

Union, intersection, difference

$$(e_1[+]e_2(\mathrm{I}) = \{t \mid t \in e_1(\mathrm{I}) \vee t \in e_2(\mathrm{I})\}$$

$$(e_1[*]e_2)(\mathrm{I}) = \{t \mid t \in e_1(\mathrm{I}) \wedge t \in e_2(\mathrm{I})\}$$

$$(e_1[-]e_2)(\mathrm{I}) = \{t \mid t \in e_1(\mathrm{I}) \wedge t \notin e_2(\mathrm{I})\}$$

Explanation. Although the formal definitions remain unchanged, note that the membership condition is extended, as shown in the following examples.

```
{1, null}[+]{2, null} = {1, 2, null}
{1, null}[*][2, null}  = {null}
{1, null}[−]{2, null} = {1}
```

Aggregate formation, general aggregate formation

$$(e\ \langle A;\ \text{agg's}\rangle)(\mathrm{I}) = \{t[A] \circ f's \mid t \in e(\mathrm{I}) \wedge f's = \text{agg's}(\{r \mid r \in e(\mathrm{I}) \wedge r[A] \equiv t[A]\})\}$$

$$(e_1[A_1/A_2;\ \text{agg's}]e_2)(\mathrm{I}) = \{t[A_2] \circ f's \mid t \in e_2(\mathrm{I}) \wedge f's = \text{agg's}(\{r \mid r \in e_1(\mathrm{I}) \wedge r[A_1) \equiv t[A_2]\})\}$$

Explanation. Both aggregate formations must conform to the group-by exception. Next, we must define the treatment of a null value to be aggregated, although it is not explicitly stated in the above definitions. The treatment is defined in conformity to "exclusion in aggregation" in SQL, as shown in Figure 5. The same treatment gives a good perspective on the correctness of the translation.

Nonnull selection

$$(e[{\sim}?;\ A](\mathrm{I}) = \{t \mid t \in e(\mathrm{I}) \wedge {\sim}(t[a_1] \equiv \text{null}) \wedge {\sim}(t[a_2] \equiv \text{null}) \wedge \ldots\} \quad \text{where} \quad A = \{a_1, a_2, \ldots\}$$

R

A	8
B	6
B	4
-null-	9
A	-null-
A	2
-null-	3

S

A	3	2	8	10	5
B	2	4	6	10	5
-null-	2	3	9	12	6

```
S := R<#1;count,min[2],max[2],sum[2],avg[2]>
```

Fig. 5. Aggregate formulation for null values.

Explanation. This operation selects only tuples having a nonnull value in every specified attribute. The operation is regarded as a multiple-attribute version of the IS NOT NULL predicate in SQL. The operation will be used to make up for a fault in join elimination rules.

6.3 Extension of Translation

It is our aim to guarantee that the proposed translation method should be extended so as to generate optimized extended algebraic expressions that are equivalent to any extended calculus expressions. We have already described the translation without null values and now consider the null value impact on the translation.

Our basic idea is to localize to the very limit an area where the three-valued logic should be executed. Within the area the three-valued logic is applied, and outside the area the two-valued logic is applied. We call this kind of area a *three-valued zone*. In an alpha, the three-valued zone is a qualifier, since where-clause truncation occurs at the final evaluation of the qualifier. The heart of our idea is that under a certain condition, we can localize the three-valued zone of a qualifier into each atomic formula forming the qualifier. The certain condition is the nonexistence of a negative sign in a qualifier.

THEOREM 3. *If and only if there is no negation sign in the formula F, the three-valued zone of F can be localized into each atomic formula f_i forming F. That is,*

$$\perp F(\{f_i\}) = F(\{\perp f_i\}).$$

PROOF. We start with the case where F has no alpha nesting. First, we prove the sufficiency of the nonexistence of negation signs in F; that is, if there is no negation sign in F, then $\perp F(\{f_i\}) = F(\{\perp f_i\})$. An existentially or universally quantified formula can be rewritten using $\vee$ or $\wedge$. After the rewriting, the rewritten formula consists of a set of atomic formulas, which are connected by $\vee$ or $\wedge$. Here, we turn to the inductive proof on the number of $\vee$s or $\wedge$s.

Basis: No $\vee$s or $\wedge$s in F. Then F is nothing but an atomic formula, which clearly makes the claim hold.

Induction: Assume that the inductive hypothesis is true for a formula that has not more than n occurrences of $\vee$s or $\wedge$s. Then consider a formula F that comprises $n + 1$ occurrences of $\vee$s or $\wedge$s. The formula F is in either form $F = F_1 \vee F_2$ or $F = F_1 \wedge F_2$. Both F_1 and F_2 consist of not more than n occurrences of $\vee$s or $\wedge$s. Hence $\perp F_1(\{f_j\}) = F_1(\{\perp f_j\})$, and $\perp F_2(\{f_k\}) = F_2(\{\perp f_k\})$. The following complete the induction.

$$\perp F = \perp(F_1 \vee F_2) = (\perp F_1) \vee (\perp F_2) = F_1(\{\perp f_j\}) \vee F_2(\{\perp f_k\}) = F(\{\perp f_i\})$$
$$\perp F = \perp(F_1 \wedge F_2) = (\perp F_1) \wedge (\perp F_2) = F_1(\{\perp f_j\}) \wedge F_2(\{\perp f_k\}) = F(\{\perp f_i\}).$$

To prove the necessity, assume that there is a negation sign in front of a subformula F_3 in F. It is clear that the three-valued zone of F_3 cannot be localized any more, because $\perp(\sim F_3) \neq \sim(\perp F_3)$. Hence $\perp F(\{f_i\}) \neq F(\{\perp f_i\})$.

Next, consider the case where F has an alpha nesting, including an aggregate function. An alpha nesting can be evaluated step-by-step in an inner-first order. This means that at the time when a formula (qualifier) F of an alpha is evaluated, every alpha nesting within the alpha has already been evaluated. That is, the situation is just the same as in no alpha nesting. Hence, the theorem holds. □

Theorem 3 indicates that a negation sign plays a critical role in the extension of the translation. The theorem guarantees that without a negation sign we can localize the three-valued logic into an atomic formula, and can treat other aspects of the translation in two-valued logic. Fortunately, the present translation removes every negation sign at the first stage, as described in Section 4.1. However, note that a negation sign is generated by Rule UQ. The significance of this rewriting is to give the opportunity to apply an efficient rule, NS. Hence we cannot avoid this type of negation: $\sim\exists\Omega(\omega)f$.

There are two rules that rewite a negation sign followed by an existential quantifier: NEQ (negative existential quantifier) and NS (negative semijoin). Both rules should be examined.

Rule NEQ itself remains unchanged, although the proof is slightly modified as shown below. Although the rule generates a rather complex expression, it is a basic one and our expectation is that this type of negation sign will be rewritten mostly by the more efficient rule NS+ (described later). Note also that the negation sign handed over to $\sim f$ can be resolved in the same way described in the previous sections.

Rule NEQ. Negative Existential Quantifier (with proof modified)

from $\quad$ tl:$\Psi(\psi)$:$(\sim\exists\Omega(\omega)f) \wedge g$
to $\quad$ tl:$(((\psi', \omega_f'):\Psi(\psi'), \Omega(\omega'):\sim f)[\omega_f'/\omega_f]\Omega)(\psi)$:$g$

PROOF.

$$\begin{aligned}
&\text{tl}:\Psi(\psi):(\sim\exists\Omega(\omega)f) \wedge g)(\text{I})\\
&= \{\psi[\text{TL}] \mid \psi \in \Psi(\text{I}) \wedge g(\psi) \wedge \sim\exists\omega(\omega \in \Omega(\text{I}) \wedge f(\psi, \omega) = \{1, \tfrac{1}{2}\})\} && (1)\\
&= \{\psi[\text{TL}] \mid \psi \in \Psi(\text{I}) \wedge g(\psi) \wedge \psi\\
&\quad \notin \{\psi'' \mid \psi'' \in \Psi(\text{I}) \wedge \omega \in \Omega(\text{I}) \wedge f(\psi'', \omega) = \{1, \tfrac{1}{2}\}\}\}\\
&= \{\psi[\text{TL}] \mid \psi \in \Psi(\text{I}) \wedge g(\psi) \wedge \psi \in \{\psi' \mid \psi' \in \Psi(\text{I}) \wedge \forall\Omega(\omega)(\sim f(\psi', \omega))\}\} && (2)\\
&= \{\psi[\text{TL}] \mid g(\psi) \wedge \psi \in \{\psi' \mid \psi' \in \Psi(\text{I}) \wedge \forall\Omega(\omega)(\sim f(\psi', \omega))\}\}
\end{aligned}$$

$$= \{\psi[\mathrm{TL}] \mid g(\psi) \wedge \psi$$
$$\in \{\psi' \mid \psi' \in \Psi(\mathrm{I}) \wedge \forall \Omega(\omega) \exists \Omega(\omega')({\sim}f(\psi', \omega') \wedge \omega_f = \omega'_f)\}\} \qquad (3)$$
$$= \{\psi[\mathrm{TL}] \mid g(\psi) \wedge \psi \in (((\psi', \omega'_f){:}\Psi(\psi'), \Omega(\omega'){:}{\sim}f(\psi', \omega'))[\omega'_f/\omega_f]\Omega, (\mathrm{I})\} \qquad (4)$$
$$= (\mathrm{tl}{:}(((\psi', \omega'_f){:}\Psi(\psi'), \Omega(\omega'){:}{\sim}f(\psi', \omega'))[\omega'_f/\omega_f]\Omega)(\psi){:}g)(\mathrm{I}) \quad \square$$

Explanation. (1) A tuple that makes $f(\psi, \omega)$ unknown should also be discarded. (2) Directly derivable from substituting ${\sim}\exists\Omega(\omega)f$ by $\forall\Omega(\omega)({\sim}f)$. (3) Strictly speaking, $\omega_f \equiv \omega'_f$ should be used in place of $\omega_f = \omega'_f$, but $\omega_f = \omega'_f$ will do because a tuple that has a null in ω_f will ultimately be discarded. (4) Apply Rule JD.

Rule NS should be replaced with a slightly heavier version NS+, that is, one difference is added. Compared with Rule NEQ, this rule generates a much simpler expression by making good use of the extended negative semijoin.

Rule NS+. Negative Semijoin (modified)

from $\quad \mathrm{tl}{:}X(x),\ \Omega(\omega){:}({\sim}\exists \mathrm{Y}(y)(f_j(x, y) \wedge g(y)) \wedge h$

to $\quad \mathrm{tl}{:}(X[{\sim}\exists;\ F_j](\mathrm{Y}[-](y{:}\mathrm{Y}(y){:}{\sim}g)))(x),\ \Omega(\omega){:}h$

PROOF.

$$(\mathrm{tl}{:}X(x), \Omega(\omega){:}({\sim}\exists \mathrm{Y}(y)(f_j(x,y) \wedge g(y)) \wedge h)(\mathrm{I})$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in X(\mathrm{I}) \wedge \omega \in \Omega(\mathrm{I}) \wedge {\sim}\exists y(u \in \mathrm{Y}(\mathrm{I})$$
$$\wedge (f_j(x,y) \wedge g(y)) = \{1, \tfrac{1}{2}\}) \wedge h\}$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in X(\mathrm{I}) \wedge x \notin \{x'' \mid x'' \in X(\mathrm{I}) \wedge y \in \mathrm{Y}(\mathrm{I})$$
$$\wedge (f_j(x'',y) \wedge g(y)) = \{1, \tfrac{1}{2}\}\} \wedge \omega \in \Omega(\mathrm{I}) \wedge h\}$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in X(\mathrm{I}) \wedge x \in \{x' \mid x' \in X(\mathrm{I}) \wedge \{y \mid y \in \mathrm{Y}(\mathrm{I})$$
$$\wedge (f_j(x',y) \wedge g(y)) = [1, \tfrac{1}{2}\}\} = \phi\} \wedge \omega \in \Omega(\mathrm{I}) \wedge h\}$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in \{x' \mid x' \in X(\mathrm{I})$$
$$\wedge \{y \mid y \in \mathrm{Y}(\mathrm{I}) \wedge (f_j(x',y) \wedge g(y)) = \{1, \tfrac{1}{2}\}\} = \phi\} \wedge \omega \in \Omega(\mathrm{I}) \wedge h\}$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in \{x' \mid x' \in X(\mathrm{I})$$
$$\wedge \{y \mid y \in \mathrm{Y}(\mathrm{I}) \wedge g(y) = \{1, \tfrac{1}{2}\} \wedge f_j(x',y) = \{1, \tfrac{1}{2}\}\} = \phi\} \wedge \omega \in \Omega(\mathrm{I}) \wedge h\}$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in \{x' \mid x' \in X(\mathrm{I}) \wedge \{u \mid u$$
$$\in \{y \mid y \in \mathrm{Y}(\mathrm{I}) \wedge g(y) = \{1, \tfrac{1}{2}\}\} \wedge f_j(x',u) = \{1, \tfrac{1}{2}\}\} = \phi\} \wedge \omega \in \Omega(\mathrm{I}) \wedge h\}$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in \{x' \mid x' \in X(\mathrm{I}) \wedge \{u \mid u \in \{y \mid y \in \mathrm{Y}(\mathrm{I}) \wedge y$$
$$\notin \{y' \mid y' \in \mathrm{Y}(\mathrm{I}) \wedge g(y') = 0\}\} \wedge f_j(x',u) = \{1, \tfrac{1}{2}\}\} = \phi\} \wedge \omega \in \Omega(\mathrm{I}) \wedge h\}$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in \{x' \mid x' \in X(\mathrm{I}) \wedge \{u \mid u \in (\mathrm{Y}[-](y{:}\mathrm{Y}(y){:}{\sim}g))$$
$$\wedge f_j(x',u) = \{1, \tfrac{1}{2}\}\} = \phi\} \wedge \omega \in \Omega(\mathrm{I}) \wedge h\}$$
$$= \{(x \circ \omega)[\mathrm{TL}] \mid x \in (X[{\sim}\exists; F_j](\mathrm{Y}[-](y{:}\mathrm{Y}(y){:}{\sim}g)))(\mathrm{I}) \wedge \omega \in \Omega(\mathrm{I}) \wedge h\} \qquad (1)$$
$$= (\mathrm{tl}{:}(X[{\sim}\exists; F_j](\mathrm{Y}[-](y{:}\mathrm{Y}(y){:}{\sim}g)))(x), \Omega(\omega){:}h)(\mathrm{I}) \quad \square$$

Explanation. (1) Apply the definition of the extended negative semijoin.

Comment. When we use the inverse truncation $\top$ that maps true or unknown to true, $(\mathrm{Y}[-](y{:}\mathrm{Y}(y){:}{\sim}g))$ can be made simpler.

Now that the problem of a negation sign has been solved, we can focus attention on the truncation of an atomic formula, as is guaranteed by Theorem 3. Basically, any atomic formula in a calculus expression is handed over to the atomic formula embedded in definitions of algebraic operations, as is shown in each proof of the translation. Until the extension of the translation is perfected, however, the following three considerations are indispensable: (1) There are discontinuous points where atomic formulas disappear by join elimination rules. Hence we must reexamine a join elimination rule to determine whether the elimination still holds in the three-valued logic context. (2) On the other hand, some atomic formulas are generated in the middle of the translation, and they also must be reexamined to determine whether they are still correct in the three-valued logic context. (3) Moreover, the influence of the newly introduced comparator $\equiv$ should be examined; that is, rules assuming an eaf or ef type formula must be examined to determine if what they assume is still $=$ or not ($\equiv$, or both).

First of all, we extend the formula classification by adding new types: eaf+ and ef+. The former is a jaf whose comparator is $\equiv$, and the latter is eaf+, or conjunction of eaf+'s where variable pairs are the same. Moreover, previous types eaf and ef are extended to admit an eaf+ as an atomic formula.

(a) *Join elimination rules should be modified.* An elimination of join holds owing to the fact that the join does not work at all. The elimination assumes an eaf or ef without exception. In the three-valued logic context, however, the join of eaf or ef, for example, $x[i] = y[j]$, does work; it removes a tuple having a null value. Hence, if an eaf and ef are replaced with an eaf+ and ef+ respectively, the join elimination holds. Otherwise, in the case of an eaf or ef, nonnull selection should be added to eliminate a tuple having a null in the joining attribute. Modified Rules EJ+, D+, EM+, and EK1M+ are described below. Proofs are omitted since they are obvious.

Rule EJ+. Elimination of Join (modified)

(1) *from* $\mathrm{tl}{:}X(x),\ \mathrm{Y}(y),\ \Omega(\omega){:}f_e(x, y) \wedge g$
to $\mathrm{tl}'{:}\mathrm{Y}[\sim ?;\ y_f'](y),\ \Omega(\omega){:}g'$

(2) *from* $\mathrm{tl}{:}X(x),\ \Omega(\omega){:}(\exists \mathrm{Y}(y)(f_e(x, y) \wedge g_1)) \wedge g_2$
to $\mathrm{tl}'{:}\mathrm{Y}[\sim ?;\ y_f'](y),\ \Omega(\omega){:}g'$

(3) *from* $\mathrm{tl}{:}\ \mathrm{Y}(y),\ \Omega(\omega){:}(\exists X(x)(f_e(x, y) \wedge g_1)) \wedge g_2$
to $\mathrm{tl}'{:}\mathrm{Y}[\sim ?;\ y_f'](y),\ \Omega(\omega){:}g'$,

where y_f' is the attribute(s) found at $y[i] = \cdots$ in f_e, excluding $y[i] \equiv \cdots$. The remaining conditions are the same as in Rule EJ.

Comment. Special cases where f_e is restricted to f_{e+} are not explicitly stated, because they are included in the above rewritings; in such special cases, nonnull selection is no longer necessary and should be omitted.

Rule D+. Division (modified)

from $\mathrm{tl}_x{:}X(x){:}\forall \mathrm{Y}(y)\exists Z(z)(f_e(x, z) \wedge g_e(y, z))$
to $\mathrm{tl}_w{:}(Z[z_f, z_g][z_g/y_g]\mathrm{Y})[\sim ?;\ z_f'](w){:}(\,)$

where z_f' is the attribute(s) found at $z[i] = ..$ in f_e, excluding $z[i] \equiv \cdots$. The remaining conditions are the same as in Rule D.

Rule EM+. Elimination of Merge (modified)

from $(\mathrm{tl}_1,\ \mathrm{agg}(\mathrm{tl}_2{:}\Upsilon(v){:}h_e \wedge g)){:}\Psi(\psi){:}(\,)$
to $((v_h,\ \mathrm{tl}_2){:}\Upsilon(\theta v){:}g)[\sim ?;\ u_h'][(\#1, \ldots, \#n)/\mathrm{TL}_1);\ \mathrm{agg}']\Psi$,

where u'_h is the attribute(s) found at $u[i] = \cdots$ in h_e, excluding $u[i] \equiv \cdots$. The remaining conditions are the same as in Rule EM.

Rule EK1M+. Elimination of both Klug's Correction Type 1 and Merge (modified)

from $\quad$ $\text{tl}:\Psi(\psi),\ \Omega(\omega):(\text{count}(Z)\theta c) \wedge g$
to $\quad$ $\text{tl}:\Psi(\psi),\ \Omega(\omega):(\exists S(s)(s[n+1]\theta c \wedge f_{e+}(s, \psi))) \wedge g,$

where $S = ((v_h, \text{tl}_2):\Upsilon(v):q)[\sim?;\ u'_h]\ \langle \#1, \ldots, \#n;\ \text{count}\rangle$. u'_h is the attribute(s) found at $u[i] = \cdots$ in h_e, excluding $u[i] \equiv \cdots$. The remaining conditions are the same as in Rule EK1M.

Comment. Note that the nonnull selection $[\sim?;\ u'_h]$ should be done before the aggregate formation. Note also that the linkage formula f_e is replaced with f_{e+} (see below).

(b) *A linkage formula should be ef*$^+$. A linkage formula is generated in the first phase of the translation of an aggregate function. Since even null values have to be handed over through the linkage, the linkage formula should be ef$^+$ as shown below:

$$f_{e+}(s, \psi) = (s[1] \equiv \psi z[1] \wedge \cdots \wedge s[n] \equiv \psi z[n]).$$

Rules SAT, SAQ, and EK1 must be modified in this point.

(c) *The other rules need no change.* The other rules which are relevant to an eaf or ef are confined to Rules JD and DS. Corresponding to the definition of the extended division, every comparator of the formula g_e is strictly limited to $=$, causing no change in the rules. Moreover, the predicate $s[x] = t[x]$ in the third line of the proof of Rule JD must be replaced with $s[x] \equiv t[x]$, which will result in the application of the definition of the extended division. Similar replacement must be done in Rule DS. However, these replacements cause no change in the rules.

THEOREM 4. *Even when SQL-type null values are introduced, any calculus expression has an equivalent algebraic expression.*

PROOF. The correctness of the theorem is clear from the comprehensive explanation described in this section. □

7. CONSIDERATIONS

7.1 Translation Examples and Optimization

We programmed the proposed translation method in PROLOG and subjected it to about a hundred example queries. The translation experiment shows that the proposed translation works pretty well and generates optimized algebraic expressions.

The following are the translation results of several examples given in [5] and [17]. Each example consists of an English query, its calculus expression, the generated algebraic expression, and a set of rules used in the translation. A set of rules is represented in a list whose nested structure reflects alpha nestings.

Chang 1. Find the volume of guns sold by the toy department.

rc	(u[3]):sales(u):u[1] = toy ∧ u[2] = gun
ra	sales[(#1 = toy) ∧ (#2 = gun)][#3]
rules	[S, P]

Chang 2. List the names and managers of employees in the shoe department with a salary greater than 10000.

rc	(u[1], u[3]):emp(u):u[2] > 10000 ∧ u[4] = shoe
ra	emp[(#2 > 10000) ∧ (#4 = shoe)][#1, #3]
rules	[S, P]

Chang 3. Find those items sold by departments on the second floor.

rc	(u[2]:sales(u):∃loc(v))(v[1] = u[1] ∧ v[2] = 2)
ra	(sales[∃; #1 = #1](loc[#2 = 2]))[#2]
rules	[[S, EP], PS, P]

Chang 4. Find the salary of Anderson's manager.

rc	(u[2]):emp(u):∃loc(v) (v[3] = u[1] ∧ v[1] = Anderson)
ra	(emp[∃; #1 = #3](emp[#1 = Anderson]))[#2]
rules	[[S, EP], PS, P]

Chang 5. Find the names of employees who make more than their managers.

rc	(u[1]):emp(u):∃emp(v) (v[1] = u[3] ∧ v[2] < u[2])
ra	(emp[∃; #3 = #1, #2 > #2]emp)[#1]
rules	[PS, P]

Chang 6. Find the average salary of employees in the toy or shoe departments.

rc	avg[2](u:emp(u):u[4] = toy ∨ u[4] = shoe)
ra	emp[(#4 = toy) ∨ (#4 = shoe)] ⟨avg[2]⟩
rules	[CAF, [S, EP]]

Chang 7. How many type A items?

rc	count(u:class(u):u[2] = A)
ra	class[#2 = A] ⟨count⟩
rules	[CAF, [S, EP]]

Chang 8. List the name and salary of each manager who manages more than 10 employees.

rc	(u[1], u[2]):emp(u):count(v:emp(v):v[3] = u[1]) > 10
ra	(emp[∃; #1 = #1](emp ⟨#3; count⟩[#2 > 10]))[#1, #2]
rules	[EK1M, [EP], [S, EP], PS, P]

Chang 9. Find the names of those employees who make more than any employee in the shoe department.

rc	(u[1]):emp(u):∀emp(v) (v[4] ⟨ ⟩ shoe ∨ v[2] < u[2])
ra	(emp[~∃; #2 < #2](emp[#4 = shoe]))[#1]
rules	[UQ, [S, EP], NS, P]

Chang 10. Find those companies, each of which supplies every item.

rc	(u[1]):supply(u):∀class(v)∃supply(w) (w[1] = u[1] ∧ w[3] = v[1])
ra	supply[#1, #3][#2/#1]class
rules	[D, EP]

Chang 11. Find the companies, each of which supplies every item of type A to some department on the second floor.

```
rc      (u[1]):supply(u):∀(t:class(t):t[2] = A) (v)
        ∃((s):supply(s):∃loc(r) (r[1] = s[2] ∧ r[2] = 2))(w)
        (w[1] = u[1] ∧ w[3] = v[1])
ra      ((supply[∃; #2 = #1](loc[#2 = 2]))[#1, #3])[#2/#1](class[#2 = A])
rules   [[S, EP], [[S, EP], PS, EP], D, EP]
```

Chang 12. List all companies that supply at least two departments with more than 100 items.

```
rc      (u[1]):supply(u):count(v:((w[1], w[2]):supply(w):
        count(x:supply(x):x[1] = w[1] ∧ x[2] = w[2]) > 100) (v):v[1] = u[1]) >= 2
ra      supply ⟨#1, #2; count⟩[#3 > 100][#1, #2] ⟨#1; count⟩[#2 >= 2][#1]
rules   [EK1M, [[EK1M, [EP], EJ, S, P], EP], EJ, S, P]
```

Chang 13. Among all departments with total salary greater than 1000000, find those departments that sell dresses.

```
rc      (u[1]):(v:sales(v):v[2] = dress)(u):sum[2](w:emp(w):w[4] = u[1]) > 1000000
ra      (emp[#4/#1; sum[2]](sales[#2 = dress]))[#2 > 1000000][#1]
rules   [[S, EP], SAQ, [EM, [EP]], S, EJ, P]
```

Chang 14. Find all items that are sold on more than 2 floors.

```
rc      (u[2]):sales(u):count((v[2]):loc(v):∃sales(w)(w[1] = v[1] ∧ w[2] = u[2])) > 2
ra      (loc[#1 = #1]sales)[#4, #2] < #1; count > [#2 > 2][#1]
rules   [EK1, [EJ, J, P], EJ, S, P]
```

Chang 15. Among all pairs of departments that sell dresses, find every pair that sells at least 3 items in common.

```
rc      (u[1], v[1]):(s:sales(s):s[2] = dress) (u),
        (t:sales(t):t[2] = dress) (v):u[1] ⟨ ⟩ v[1]
        ∧ count((w[2]):sales(w):w[1] = u[1] ∧ ∃sales(x)
        (x[1] = v[1] ∧ x[2] = w[2])) > = 3
ra      ((((sales[#2 = dress])[#1 ⟨ ⟩ #1](sales[#2 = dress]))
        [#1 = #1]sales)[∃; #4 = #1, #8 = #2]
        sales)[#1, #4, #8]⟨#1, #2; count⟩[#3 >= 3][#1, #2]
rules   [[S, EP], [S, EP], J, EK1, [J, PS, [EP], P], S, EJ, P]
```

Chang 16. Find all floors where a department sells all items.

```
rc      (u[2]):loc(u):∀class(v)∃sales(w) (w[1] = u[1] ∧ w[2] = v[1])
ra      (loc[∃; #1 = #1]((sales[#1, #2])[#2/#1]class)[#2]
rules   [DS, P]
```

Chang 17. Find every floor where all departments on the floor sell a type A item.

```
rc      (u[2]):loc(u):∀loc(v)[2]⟨ ⟩u[2]
        ∨ ∃sales(w) (w[1] = v[1] ∧
        ∃class(x) (x[1] = w[2] ∧ x[2] = A)))
ra      (loc[~∃; #2 = #2](loc[~∃; #1 = #1]
        (sales [∃; #2 = #1](class[#2 = A]))))[#2]
rules   [UQ, [[[S, EP], PS, EP], NS, EP], NS, P]
```

The university schema given in [17] is as follows:

```
student(name, yr, gpa)
dept(name, head, college)
faculty(name, dname)
grad(name, majorprof, grandamt).
```

Klug 1. For each student year, find the average of gpa's for students in that year.

```
rc      (u[2], avg[3](v:student(v):v[2] = u[2])):student(u):()
ra      student[#2, #1, #2, #3] ⟨#1; avg[4]⟩
rules   [EK2, [EJ, P]]
```

Klug 2. For each student year, find the average of gpa's for students in that year or in a greater year.

```
rc      (u[2], avg[3](v:student(v):v[2] >= u[2])):student(u):()
ra      (student[#2 <= #2]student)[#2, #4, #5, #6]⟨#1; avg[4]⟩
rules   [EK2, [J, P]]
```

Klug 3. For every department in the LS College, find the average graduate student support.

```
rc      (u[1], avg[2]((v[1], sum[2]((w[1], w[3]):grad(w):w[2] = v[1])):
        faculty(v):v[2] = u[1])):dept(u):u[3] = LS
ra      (faculty[#1 ≡ #1](grad[#2, #1, #3][#1/#1; sum[3]]faculty))[#2, #1, #4]
        [#1/#1; avg[3]](dept[#3 = LS])
rules   [S, EM, [SAT, [EM, [P], [P]], J, P], [P]]
```

The interesting point is that the translation results obtained for the examples shown in [5] are exactly the same as those that the author thought were the optimized algebraic expressions. The results were much the same for the examples presented in [17]; however, the examples Klug 1 and Klug 3 are slightly different. The following in the expression the author derived for Klug 1:

```
ra      student ⟨#2; avg[3]⟩
```

The expression the author derived for Klug 3 is as follows:

```
        (faculty[#1 ≡ #1](grad[#2/#1; sum[3]]faculty))[#2, #1, #4]
        [#1/#1; avg[3]](dept[#3 = LS])
```

These expressions are almost identical to the translation results, except that the translation results include unnecessary projections. If we identify an attribute by the unique name instead of by the number, the superfluous projections can easily be eliminated. Otherwise, a simple postprocessing procedure can also eliminate them.

Thus the above experiment shows that the proposed translation method works well. Note also that any translation, except for a quite simple one, uses at least one heuristic rewriting rule.

7.2 General Aggregate Formation and Outer Join

It is known that Klug's correction can be performed by introducing the (left) outer join [7]. The result of the outer join includes dangling tuples of the left-side relation with null values padded out, along with the result of the regular join. The general aggregate formation also performs Klug's correction; hence, the following two definitions appear the same:

(1) $(e_1[A_1/A_2;\ \mathrm{agg}]e_2)$
(2) $(e_2[+;\ A_2 = A_1]e_1)\ \langle A_2;\ \mathrm{agg}\rangle$,

where $e_1[+;\ Fj]e_2$ denotes the outer join between e_1 and e_2. In fact, the above two are the same except in one case; namely, in the case of count, count(*) in SQL, the results for Klug's correction are different. While the former produces the desired zero, the result of the latter is the cardinality of a group, that is, one. Hence the general aggregate formation more naturally performs Klug's correction.

Moreover, the general aggregate formation can be more efficiently implemented, because it can avoid real tuple concatenations which are carried out by the outer join. In fact, the additional processing load of the general aggregate formation, compared with the original one, is only the scan and hashing of grouping-by attribute values.

8. CONCLUSION

In this paper we have proposed a rule-based translation method with optimization, which has enough rewriting rules, especially heuristic ones, and translates calculus expressions having aggregate functions into optimized algebraic expressions. Human knowledge about constructing optimized expressions is incorporated into the heuristic rewriting rules. Thus, logical optimization is carried out through the translation. Moreover, after SQL-type null values are formally defined, their impact on the present translation methods is thoroughly investigated, resulting in the slight extensions of relational calculus, relational algebra, and the translation rules. A translation experiment using many queries showed that the proposed translation works fairly well. It also showed that heuristic rewriting rules play an essential role in the optimization. Thus translation with optimization has been shown to be quite promising.

The results reported here can form a foundation for more work in the following directions:

(1) We select relational calculus as calculus-type query language, because it has important query language features (nested queries, quantifiers, aggregation, etc.) within simple syntax and strict semantics. Since the expressive powers of SQL and relational calculus are equivalent and the translation between the two languages is rather straightforward, the proposed translation is applicable to a translation from SQL to relational algebra.

(2) The present translation will form the front end of a database machine. Recent database machines, for example MACH [21], developed by the author, can execute relational algebra very quickly. Thus, a new system architecture, in which calculus-type queries are effectively executed through algebraic operations, has become practical and attractive. A comprehensive comparison is required between the new architecture and the traditional nested iterative one.

(3) Although the proposed basic rules are necessary and sufficient to finish the translation, it is not guaranteed that the proposed heuristic rules are sufficient for any poor, but legitimate, calculus-type queries. Experiments in actual application environments may suggest a lack of additional heuristic rules, probably

rather trivial ones. Adding them would finally guarantee the sufficiency of the heuristic rules.

(4) Along with the spread of database applications, various attempts have been made to relax the first normal restriction of the relational model. The approach of extending the present translation to the nested relation contexts will give us a good perspective and requires further research.

ACKNOWLEDGMENTS

The author is grateful to Kunio Murakami, Setsuo Ohsuga, Yahiko Kambayashi, Hirofumi Katsuno, and the anonymous referees for their many helpful comments and suggestions, to Kazumi Saito for pointing out some rewriting rules, and to Haruo Hayami for providing a hint about Theorem 3.

REFERENCES

1. BULTZINGSLOEWEN, G. V. Translating and optimizing SQL queries having aggregates. In *Proceedings of the 13th Conference on Very Large Data Bases* (Brighton, England, Sept. 1987), pp. 235–243.
2. CERI, S., AND GOTTLOB, G. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr. 1985), 324–345.
3. CODD, E. F. Relational completeness of data base sublanguages. In *Data Base Systems*, Vol. 6, Courant Computer Symposia Series. Prentice-Hall, Englewood Cliffs, Nov., 1972, pp. 65–98.
4. CODD, E. F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 397–434.
5. CHANG, C. L. Deduce 2: Further investigations of deduction in relational data bases. In *Logic and Data Bases*. H. Gallaire and J. Minker, Eds., Plenum Press, New York, 1978, pp. 201–236.
6. DATE, C. J. *An Introduction to Database Systems, Volume* 2. Addison-Wesley, Reading, Mass., 1983.
7. DAYAL, U. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the 13th Conference on Very Large Data Bases* (Brighton, England, Sept. 1987). pp. 197–208.
8. DEWITT, D. J., GHANDEHARIZADEH, S., SCHNEIDER, D., JAUHARI, R., MURALIKRISHNA, M., AND SHARMA, A. A single user evaluation of the Gamma database machine. In *Proceedings of the 5th International Workshop on Database Machines* (Karuizawa, Japan, Oct. 1987). pp. 43–59.
9. FREYTAG, J. C. A rule-based view of query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, May 1987). ACM, New York, pp. 173–180.
10. GRAEFE, G., AND DEWITT, D. J. The EXODUS optimizer generator. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, May 1987). ACM, New York, pp. 160–172.
11. HAAS, L. M., FREYTAG, J. C., LOHMAN, G. M., AND PIRAHESH, H. Extensible query processing in Starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Portland, Ore., May–June 1989). ACM, New York, pp. 377–388.
12. ISO/TC97 *International standard*: *Information Processing Systems—Database Language* SQL, ISO 9075:1987(E). Printed in Switzerland, 1987.
13. JARKE, M., AND KOCH, J. Range nesting: A fast method to evaluate quantified queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Jose, Calif., May 1983). ACM, New York, pp. 196–206.
14. JARKE, M., AND KOCH, J. Query optimization in database systems. *ACM Comput. Surv. 16*, 2 (June 1984), 111–152.
15. KIESSLING, W. On semantic reefs and efficient processing of correlation queries with aggregates. In *Proceedings of the 11th Conference on Very Large Data Bases* (Stockholm, Aug. 1985). pp. 241–250.

16. KIM, W. On optimizing an SQL-like nested query. *ACM Trans. Database Syst. 7*, 3 (Sept. 1982), 443–469.
17. KLUG, A. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM 29*, 3 (July 1982), 699–717.
18. LELAND, M. D. P., AND ROOME, W. D. The Silicon database machine: Rationale, design, and results. In *Proceedings of the 5th International Workshop on Database Machines* (Karuizawa, Japan, Oct. 1987). pp. 454–467.
19. LEVIET, C. Translation and compatibility of SQL and QUEL queries. *J. Inf. Process. 8*, 1 (1985), 1–15.
20. MAIER, D. *The Theory of Relational Databases.* Computer Science Press, Rockville, Md., 1983.
21. NAKANO, R., AND KIYAMA, M. MACH: Much faster associative machine. In *Proceedings of the 5th International Workshop on Database Machines* (Karuizawa, Japan, Oct. 1987). pp. 482–495.
22. SMITH, L. M., AND CHANG, P. Y. Optimizing the performane of a relational algebra database interface. *Commun. ACM 18*, 10 (Oct. 1975), 568–579.
23. ULLMAN, J. D. *Principles of Database Systems.* Computer Science Press, Rockville, Md., 1982.
24. YAO, S. B. Optimization of query evaluation algorithm. *ACM Trans. Database Syst. 4*, 2 (June 1979), 133–155.

Received February 1988; revised February 1989 and November1989; accepted December1989