Translating and Optimizing SQL Queries Having Aggregates

Günter von Bültzingsloewen

Forschungszentrum Informatik an der Universität Karlsruhe
Haid-und-Neu-Str. 10-14, D-7500 Karlsruhe, West Germany

Abstract

In this paper, we give a precise definition of the semantics of SQL queries having aggregate functions, identify the problems associated with the optimization of such queries and give some solutions. The semantics of SQL queries is defined by translating them into expressions of an extended relational calculus (extensions are necessary for a correct treatment of aggregate functions and null values). The discussion of the optimization problems is based on a new transformation of a relational calculus expression into relational algebra. By investigating the transformation of aggregate functions we are able to identify two major problems: correct integration of the values of aggregate functions applied to empty relations and unnecessary computation of unneeded function values. To solve these problems we propose to interpret an aggregate function applied to a calculus expression with some free variables as a function on the attributes of these variables that are referenced in the expression. Doing so, we are able to develop several new processing strategies that should be considered by an optimizer.

1 Introduction

The SQL query language has become the standard relational manipulation language, as is reflected by ongoing standardization activities [ANSI85]. An important feature of SQL are aggregate functions like sum, average, minimum etc. However, the processing of queries with aggregate functions is not well understood. A formal definition of the semantics of SQL queries having aggregates is still lacking, as is a unified operator tree model covering this class of queries [Kies 85]. This paper is an attempt to solve these problems.

In order to give a formal definition of the semantics of SQL queries, we have to translate them into a formal language which is at least as powerful as SQL. Two well known relational query languages that have a sound theoretical foundation are relational algebra and relational calculus. As SQL is more closely related to relational calculus, we define the semantics of SQL by translation into calculus. However, relational calculus has to be extended in order to deal with aggregate functions and null values. The optimization of SQL queries is thus reduced to the optimization of relational calculus queries, which makes the following results also applicable to other query languages based on relational calculus (e.g. QUEL).

To develop an operator tree model for SQL and extended relational calculus, an algebraic query representation has to be found; the basic problem here is to identify a suitable set of algebraic operators. A possible choice is relational algebra with aggregate functions developed by [Klug82], with an extension to cover null values. To study the problem whether this is a good choice, we present a translation of relational calculus queries into relational algebra. Investigating the translation of aggregate functions, we are able to identify two major problems of the representation in relational algebra: correct integration of the values of aggregate functions applied to empty relations and duplicated computation of certain function values.

To solve these problems we propose to interpret an aggregate function applied to a calculus expression with some free variables as a function F on the attributes of these variables that are referenced in the expression, and to include selection using such a function into relational algebra. We develop a general processing strategy using this approach and present several special instances of the general strategy that are intended to approximate the minimal representation of F (i.e. each function value is represented only once for a set of arguments).

The major contributions of our work are threefold: first, the extension of relational algebra and relational calculus to cover a significant subset of SQL. Second, the translations from SQL into calculus and from calculus into algebra, showing that the extended versions of relational algebra and relational calculus have the same expressive power. Third, the development of a new general processing strategy for aggregate functions, which may form the foundation of an operator tree model for the class of queries considered.

While this work was primarily motivated by [Kies85] who illustrated the semantic problems associated with efficient processing of queries having aggregates, several other papers are relevant to our discussion. [Kim82,CeGo85, LeVi85] are concerned with the transformation of an SQL query into some kind of normal form which is more suitable for further optimization. [Kim82] describes a translation from SQL into SQL, transform-

ing one complex query into several simpler ones. [LeVi85] proposes a translation from SQL into QUEL and [CeGo85] gives a translation from SQL into relational algebra having aggregate functions. All have in common that they treat only subsets of the SQL query language. Subqueries may contain either no GROUP BY – HAVING clause [Kim82,CeGo85] or no referential variables [LeVi85] (i.e. variables declared in an outer query block). The SQL feature of null values and query evaluation using three-valued logic is generally ignored. Furthermore, arithmetic operators and control of duplicate elimination are omitted.

While we agree that the last restriction is justified for technical simplification and does not constrain the applicability of the transformations, we claim that to ignore three valued logic leads to severe semantic problems. This is the case even if attributes may have only nonull values, as the application of an aggregate function like average, sum, min and max to an empty relation gives null as a result [ANSI85]. Consequently we include the treatment of null values and three-valued logic in our dicussion. Furthermore we pose no restrictions on subqueries. However, for technical simplification we also omit arithmetic operators and control of duplicate elimination.

Besides their limitations, the above-mentioned approaches cannot be used directly for two reasons: First, as has been pointed out by [Kies85], the transformations of [Kim82,LeVi85] are incorrect in general: The application of the aggregate function count to an empty relation is not treated appropriately. Second, [CeGo85] translates an SQL query directly into relational algebra. Therefore the application of optimization techniques developed for relational calculus like [JaKo 83] is impossible and the problems associated with the processing of aggregate functions cannot be identified as easily.

In [Klug82] a precise definition of relational algebra and relational calculus query languages having aggregate functions is given and the expressive power of the two languages is proven to be equivalent. However, the case that an aggregate function may have null as a result is not considered and the proof of equivalence is quite complex, making it difficult to discuss the processing of aggregate functions.

Though not directly applicable, we can make use of some of the techniques proposed earlier. The definition of extended relational algebra and relational calculus given in section 2 is based on [Klug82]. The translation of an SQL query into extended relational calculus (section 3) makes use of the preprocessing step of [CeGo85] and some of the techniques developed in [LeVi85], which have to be extended to deal with three-valued logic. The proof of equivalence of extended relational algebra and relational calculus (section 4) is – as far as we know – new and considers a more powerful class of languages than considered before. The processing techniques for aggregate functions developed in section 5 contain as one special case the technique proposed by [Kim82].

2 Formal Definitions

Before giving definitions of relational algebra and relational calculus expression we first define the relational model in the usual way [Maie83,Klug82]. A *relation scheme* consists of a name R and a degree deg(R)$\in$ **N**. Associated with R is the set attrs(R)=$\{1, \ldots, deg(R)\}$ of attribute names (i.e. attribute names are assumed to be natural numbers). For reasons of simplicity, the domain of each attribute is assumed to be the set $\hat{\mathbf{N}}$=**N**$\cup\{\omega\}$ of natural numbers including the null value ω. *Tuples* and *relations* over a relation scheme are defined in the usual way.

A *schema* is a sequence <$R_1,\ldots,R_N$> of relation schemes. An *instance* I of schema <$R_1,\ldots,R_N$> is a sequence <$r_1,\ldots,r_N$>, where for each i=1,...,N, r_i is a relation over scheme R_i. Throughout this paper, one fixed schema <$R_1,\ldots,R_N$> is assumed.

An aggregate function f$\in$Agg is a function f: $\mathbf{R} \rightarrow \hat{\mathbf{N}}$, where **R** is the set of all relations. To translate SQL queries into relational calculus we will need aggregate functions of the form $f = agg(A_i)$, where $agg \in \{sum, min, max, avg, count\}$ and $A_i \in \mathbf{N}$ is an attribute name. For example, the function $sum(A_i)$ determines the sum of the values of attribute A_i when applied to a relation. Aggregate functions $f = agg(A_i)$ with $agg \neq count$ yield ω as result, if the relation they are applied to is empty or does not have an attribute with name A_i. An aggregate function with $agg = count$ has 0 as result when applied to an empty relation.

2.1 Relational Calculus

The definition of relational calculus given in this section is based on [Klug82]. However, the following extensions are necessary in order to be as expressive as the subset of SQL we consider, i.e. in order to include the treatment of null values and predicate evaluation using three-valued logic:

- First, the set Θ of comparison operators θ has to be extended: $\Theta = \{=, \neq, <, \leq, >, \geq, \equiv\}$. The special comparison operator $\equiv$ yields TRUE if the values compared are identical in the usual sense or both ω, and FALSE otherwise. The other operatòrs are evaluated to UNKNOWN if at least one of the values compared equals ω, TRUE if the comparison is TRUE in the usual sense and FALSE otherwise.

- Second, we need two new logical connectives $\top$ and $\bot$ which map the truth value UNKNOWN into TRUE and FALSE respectively.

Calculus expressions are defined recursively using several classes of objects: variables, terms, formulas, range formulas and alphas. Thereof only the closed alphas (alphas containing no free variables) correspond to calculus expressions. The definition starts with atomic alphas, i.e. relations defined in the schema. Free variables, bound variables and closed objects are defined in the usual way.

Variables: $\mathbf{V}=\{v_1, v_2, v_3, \ldots\}$ is the set of variables.

Terms: Terms correspond to elements of $\hat{\mathbf{N}}$. The set **T** of terms is defined as follows. For any $c \in \hat{\mathbf{N}}$, $c \in \mathbf{T}$ (constants). If $v_i \in \mathbf{V}$ and A is an attribute name, then $v_i[A] \in \mathbf{T}$ (attribute values). If f$\in$Agg and $\alpha \in \mathbf{A}$, then $f(\alpha) \in \mathbf{T}$ (application of an aggregate function to a relation).

Formulas: Formulas correspond to truth values. The set **F** of formulas is defined as follows. If t_1, t_2 are terms and $\theta \in \Theta$ is a comparison operator, then $t_1 \theta t_2 \in \mathbf{F}$. If ψ, ψ_1 and ψ_2 are formulas, then $\neg\psi \in \mathbf{F}$, $\top\psi \in \mathbf{F}$, $\bot\psi \in \mathbf{F}$, $\psi_1 \vee \psi_2 \in \mathbf{F}$, $\psi_1 \wedge \psi_2 \in \mathbf{F}$. For any formula $\psi \in \mathbf{F}$ and range formula $r_i(v_i) \in \mathbf{RF}$ we have $(\exists r_i(v_i))\psi \in \mathbf{F}$ and $(\forall r_i(v_i))\psi \in \mathbf{F}$.

Range formulas: Range formulas bind variables tp relations that are described by closed alphas, i.e. alphas containing no free variables. The set **RF** of range formulas is defined as follows. If $\alpha_{i_1}, \ldots, \alpha_{i_k}$ are closed alphas and $v_i \in \mathbf{V}$, then $r_i(v_i) \in \mathbf{RF}$ with $r_i = \alpha_{i_1} \vee \cdots \vee \alpha_{i_k}$.

Terms **T**:		
$t(I,\beta)$	$\in$	$\hat{\mathbf{N}}$
$c(I,\beta)$	$=$	c
$v_i[A](I,\beta)$	$=$	$\beta(v_i)[A]$
$f(\alpha)(I,\beta)$	$=$	$f(\alpha(I,\beta))$
Formulas **F**:		
$\psi(I,\beta)$	$\in$	{1(TRUE), 0(FALSE), $\frac{1}{2}$(UNKNOWN)}
$(t_1 \theta t_2)(I,\beta)$	$=$	$t_1(I,\beta)\theta t_2(I,\beta)$
$(\psi_1 \vee \psi_2)(I,\beta)$	$=$	$\max\{\psi_1(I,\beta), \psi_2(I,\beta)\}$
$(\psi_1 \wedge \psi_2)(I,\beta)$	$=$	$\min\{\psi_1(I,\beta), \psi_2(I,\beta)\}$
$(\neg\psi)(I,\beta)$	$=$	$1 - \psi(I,\beta)$
$(\bot\psi)(I,\beta)$	$=$	$\lfloor\psi(I,\beta)\rfloor$
$(\top\psi)(I,\beta)$	$=$	$\lceil\psi(I,\beta)\rceil$
$((\exists r_i(v_i))\psi)(I,\beta)$	$=$	$\max\{0, \max_{T \in r_i(I,\beta)} \psi(I, \beta[v_i/T])\}$
$((\forall r_i(v_i))\psi)(I,\beta)$	$=$	$\min\{1, \min_{T \in r_i(I,\beta)} \psi(I, \beta[v_i/T])\}$
Range Formulas **RF**:		
$r_i(v_i)(I,\beta)$	$\in$	{1(TRUE),0(FALSE)}
$(\alpha_{i_1} \vee \cdots \vee \alpha_{i_k})(v_i)(I,\beta)$		
	$=$	1, if $\beta(v_i) \in \alpha_{i_1}(I,\beta) \cup \cdots \cup \alpha_{i_k}(I,\beta)$ 0, otherwise
Alphas **A**:		
$\alpha(I,\beta)$	$\in$	$\mathbf{R}$
$R_i(I,\beta)$	$=$	r_i
$\phi_{<X,f>}(\alpha)(I,\beta)$	$=$	$\{T[X] \circ y : T \in \alpha(I,\beta) \wedge y = f\{T' \in \alpha(I,\beta) : T'[X] \equiv T[X]\}\}$
$(\,(t_1,\ldots,t_n) : r_1(v_1),\ldots,r_m(v_m) : \psi\,)(I,\beta)$		
	$=$	$\{(t_1(I,\beta'),\ldots,t_n(I,\beta')) : \psi(I,\beta') = 1 \wedge \beta' = \beta[v_1/T_1,\ldots,v_m/T_m] \wedge r_i(v_i)(I,\beta') = 1\}$

Table 1: Semantics of Calculus Objects

Alphas: Alphas correspond to relations. The set **A** of alphas is defined as follows. If R_i is in the schema, then $R_i \in \mathbf{A}$ (atomic alphas). If $t_1,\ldots,t_n$ are terms not containing aggregate functions, $r_1(v_1),\ldots r_m(v_m)$ are range formulas and ψ is a formula, then

$$(t_1,\ldots,t_n) : r_1(v_1),\ldots,r_m(v_m) : \psi \in \mathbf{A}.$$

This definition is recursive, based on range formulas that consist of atomic alphas.

If α is a closed alpha, f∈Agg and X⊂attrs(α), then

$$\phi_{<X,f>}(\alpha) \in \mathbf{A} \text{ (aggregate formation).}$$

In contrast to [Klug82], we do not allow aggregate functions in the target list of an alpha. Instead calculus includes the algebraic aggregate formation operator ϕ. The reason for these modifications are that translation of an SQL query into calculus as well as translation of a calculus expression into algebra become easier. However, the expressive power remains the same.

The semantics of calculus objects is defined in table 1. Objects of relational calculus are interpreted with respect to a schema instance I and a mapping β (called a valuation) which associates with each variable $v_i \in \mathbf{V}$ a tuple $\beta(v_i) = T$. $\beta[v_i/T]$ is defined to be the valuation, which yields T when applied to v_i and equals $\beta(v)$ when applied to another variable v.

The aggregate formation operator $\phi_{<X,f>}$ groups its input relation on the attributes X ($T[X]$ denotes the tuple consisting of the X-values of T; $\equiv$ yields TRUE, if the tuples being compared are identical). It applies the function f to each group and returns the X-values and the associated function value for each group ($\circ$ denotes concatenation). It can be generalized such that several functions are applied to each group ($\phi_{<X,(f_1,\ldots,f_n)>}$).

2.2 Relational Algebra

The definition of the set E of relational algebra expressions over our fixed schema and the value e(I) of an expression on an instance I of the schema is given in table 2. The following notations are used in the definition: c denotes an element of $\hat{\mathbf{N}}$. e, e_1 and e_2 denote relational algebra expressions. A and B are attribute names, X and Y sets of attribute names of equal size.

Relational Algebra Expressions: $e(I) \in \mathbf{R}$	
Literal	$\{c\}(I) = \{c\}$
Base Relation	$R_i(I) = r_i$
Projection	$\pi_X(e)(I) = \{T[X] : T \in e(I)\}$
Union	$(e_1 \cup e_2)(I) = e_1(I) \cup e_2(I)$
Difference	$(e_1 \setminus e_2)(I) = e_1(I) \setminus e_2(I)$
Product	$(e_1 \times e_2)(I) = e_1(I) \times e_2(I)$
Restriction	$\sigma_{A\theta B}(e)(I) = \{T \in e(I) : T[A]\theta T[B] = 1\}$
Restriction'	$\sigma'_{A\theta B}(e)(I) = \{T \in e(I) : T[A]\theta T[B] \in \{1, \frac{1}{2}\}\}$
Aggregate formation	$\phi_{<X,f>}(e)(I) = \{T[X] \circ y : T \in e(I) \wedge y = f\{T' \in e(I) : T'[X] \equiv T[X]\}\}$
Intersection	$(e_1 \cap e_2)(I) = e_1(I) \cap e_2(I)$
Selection	$\sigma_{A\theta c}(e)(I) = \{T \in e(I) : T[A]\theta c] = 1\}$
Selection'	$\sigma'_{A\theta c}(e)(I) = \{T \in e(I) : T[A]\theta c \in \{1, \frac{1}{2}\}\}$
Join	$(e_1[A\theta B]e_2)(I) = \sigma_{A\theta B}(e_1 \times e_2)(I)$

Table 2: Relational Algebra Expressions

In order to be at least as expressive as SQL, relational algebra also has to be extended to include the handling of null values. This is accomplished by introducing modified versions σ' of restriction and selection, which retain tuples for which the corresponding predicate yields UNKNOWN. This extension corresponds to the introduction of $\bot$ and $\top$ as logical connectives into relational calculus. In fact, instead of defining new operators we could as well introduce the logical connectives in the restriction/ selection predicate. Selection, intersection and join are or can be defined in terms of the other of the above operators.

3 Translating SQL Queries into Relational Calculus

As we are primarily interested in the translation and optimization of queries having aggregate functions, we consider only a relevant subset of SQL, which is still more powerful than the subsets covered in earlier papers [CeGo85,LeVi85,Kim82]. The only restrictions are that attribute domains are restricted to the set $\hat{\mathbf{N}}$ of natural numbers including ω, terms are restricted to contain no arithmetic operators and duplicates may not be retained in the result of a query or subquery (which actually does not restrict the formulation of subquery predicates). The effect of these restrictions is a technical simplification of the translation into calculus and algebra, the extension to complete SQL being straightforward.

A grammar for the subset of SQL being considered is given in the appendix. We can represent an arbitrary query of our subset in a form according to this grammar if we allow two simple preprocessing steps [CeGo85]:

Sub Query SQ		Predicate	Calculus Formula
SELECT	f	EXISTS SQ:	TRUE
FROM	$r_1(v_1), \ldots, r_n(v_n)$		
WHERE	P_w	$t\,\theta$ SOME SQ:	$t\theta f'(* : r_1(v_1), \ldots, r_n(v_n) : P_w)$
SELECT	t_s $(\neq f)$	EXISTS SQ:	$(\exists r_1(v_1), \ldots, \exists r_n(v_n))(\bot P_w)$
FROM	$r_1(v_1), \ldots, r_n(v_n)$		
WHERE	P_w	$t\,\theta$ SOME SQ:	$(\exists r_1(v_1), \ldots, \exists r_n(v_n))(\bot P_w \wedge t\theta t_s)$
SELECT	t_s	EXISTS SQ:	$(\exists r_1(v_1), \ldots, \exists r_n(v_n))\ (\bot(P_w \wedge P'_h))$
FROM	$r_1(v_1), \ldots, r_n(v_n)$		
WHERE	P_w	$t\,\theta$ SOME SQ	$(\exists r_1(v_1), \ldots, \exists r_n(v_n))\ (\bot(P_w \wedge P'_h) \wedge t\theta t'_s)$
GROUP BY	$v_{i_1}[A_{i_1}], \ldots, v_{i_k}[A_{i_k}]$		
HAVING	P_h	$\alpha_0 = (* : r_1(v'_1), \ldots, r_n(v'_n) : P_w[v_i/v'_i] \wedge v_{i_1}[A_{i_1}] \equiv v'_{i_1}[A_{i_1}] \wedge \cdots \wedge v_{i_k}[A_{i_k}] \equiv v'_{i_k}[A_{i_k}])$	
		$P'_h = P_h[f/f'(\alpha_0),\ f \in P_h],\ t'_s = t_s[f/f'(\alpha_0),\ f \in t_s],$	

Table 3: Translation of Subquery Predicates

SQL Query		Calculus Expression
SELECT	$f_1, \ldots, f_l$	
FROM	$r_1(v_1), \ldots, r_n(v_n)$	$(f'_1, \ldots, f'_l)(* : r_1(v_1), \ldots, r_n(v_n) : P_w)$
WHERE	P_w	
SELECT	$t_1, \ldots, t_l$ $(\neq f)$	
FROM	$r_1(v_1), \ldots, r_n(v_n)$	$(t_1, \ldots, t_l) : r_1(v_1), \ldots, r_n(v_n) : P_w$
WHERE	P_w	
SELECT	$t_1, \ldots, t_l$	$(t'_1, \ldots, t'_l) : \alpha(v) : P'_h$
FROM	$r_1(v_1), \ldots, r_n(v_n)$	
WHERE	P_w	$\alpha = (\phi_{<(A_{i_1}, \ldots, A_{i_k}), (f'_1, \ldots, f'_m)>}(* : r_1(v_1), \ldots, r_n(v_n) : P_w));$
GROUP BY	$v_{i_1}[A_{i_1}], \ldots, v_{i_k}[A_{i_k}]$	
HAVING	P_h	$(t'_1, \ldots, t'_l, P'_h) = (t_1, \ldots, t_l, P_h)[f_i/v[k+i], v_{i_j}[A_{i_j}]/v[j]];$
		$(f_1, \ldots, f_m$ aggregate functions in $t_1, \ldots, t_l,\ P_h)$

Table 4: Query Translation

SQL Query			Calculus Expression
SELECT	$sum(v_1[1]), v_1[3]$		$(v[2], v[1]) : \alpha(v) : v[3] > 0$
FROM	$R_1(v_1)$		
WHERE	P_w		$\alpha = \phi_{<3,(sum(1),max(4))>}(* : R_1(v_1) : P_w)$
GROUP BY	$v_1[3]$		
HAVING	$max(v_1[4]) > 0$		
$P_w = \neg$ ($v_1[2]$ = SOME			$P_w = \neg(\exists R_2(v_2))\ (\bot min(3)(\alpha_0) > v_1[3] \wedge v_1[2] = max(1)(\alpha_0))$
	SELECT	$max(v_2[1])$	
	FROM	$R_2(v_2)$	$\alpha_0 = (* : R_2(v'_2) : v_2[3] > v_1[3] \wedge v_2[2] \equiv v'_2[2])$
	GROUP BY	$v_2[2]$	
	HAVING	$min(v2[3]) > v_1[3]$)	

Example 1: Translation of SQL Queries

- Associate with each relation in a FROM clause a unique variable and extend each attribute reference to contain the variable it refers to.

- Transform each predicate into one of the three basic predicates. For instance, an <in predicate> can be reduced to a <quantified predicate>.

To translate an SQL query into a calculus expression, the syntactical differences between SQL and calculus (subquery predicates, GROUP BY — HAVING clause and position of aggregate functions) must be resolved. This is accomplished with two kinds of translations, i.e. translation of subquery predicates and translation of the query (see Tables 3,4).

In SQL, aggregate functions are represented in the form $f = agg(v_i[A_i])$, where v_i is bound to a relation r_i in the FROM clause of an SQL query or subquery (FROM $r_1(v_1), \ldots, r_n(v_n)$). f is applied to the result of the evaluation of the FROM and WHERE clauses of the query or subquery, which is a subset of $r = r_1 \times \cdots \times r_n$, or to partitions of this subset (in case the query or subquery contains a GROUP BY clause). Hence f is translated into $f' = agg(A'_i)$, where A'_i names the attribute of r that corresponds to the attribute A_i of r_i (In tables 3,4 this is indicated by f and f').

The translation of subquery predicates has the following property: The calculus formula yields the same truth value as the original SQL predicate when applied to the same schema instance I and the same valuation β for the free variables of the calculus formula resp. SQL predicate. In the translation, three types of subqueries are distinguished:

- The first type yields exactly one value as the result of an aggregate function. Hence the <exists predicate> is always TRUE and the <quantified predicate> is evaluated by comparison with the result of the aggregate function.

- The second type is evaluated to a set S of values. The <exists predicate> is TRUE, if S is not empty, and FALSE otherwise. The <quantified predicate> is TRUE, if the comparison is TRUE for at least one value in S. It is FALSE, if S is empty or if the comparison is FALSE for every value in S, and UNKNOWN otherwise. The subtle point in the translation of this subquery type is that we have to make sure that the comparison is evaluated only for values in S, i.e. that values for which P_w is evaluated to UNKNOWN may not be included. This is accomplished by application of the special logical connective $\perp$ to P_w. In example 1, if $\alpha_0 = \emptyset$ for some $T_1 \in r_1$, P_w is evaluated to TRUE and T qualifies. If $\perp$ would be omitted, P_w would be evaluated to UNKNOWN and T would not qualify.

- The third type is evaluated to a set of values, one value for each group of tuples built when processing the GROUP BY — HAVING clause (t_s is either a grouping attribute or an aggregate function). The special problem here is that we have to make sure that each aggregate function contained in t_s or P_w is applied to the correct group of tuples. This is done by means of the alpha α_0, which yields the group corresponding to a list of values of the grouping attributes. Any aggregate function f contained in the SELECT clause or in the HAVING clause has to be applied to α_0, i.e. f has to be substituted by $f'(\alpha_0)$ (In example 1, $min(v_2[3])$ is substituted by $min(3)(\alpha_0)$).

The calculus expression resulting in the translation of an SQL query gives the same result as the query when applied to the same schema instance I. As the structure of a calculus expression directly reflects the structure of the original SQL query, the translation given in Table 4 should be self-explanatory. The alpha α built in the translation of the third query type is the result of evaluating FROM, WHERE and GROUP BY clause of the SQL query to which the aggregate functions contained in the HAVING clause and the SELECT clause are applied. Attribute references and aggregate functions contained in the latter constructs have to be substituted such that they reference the corresponding attributes of α (In example 1, $sum(v_1[1]), v_1[3]$ is substituted by $v[2], v[1]$).

4 Equivalence of Extended Relational Algebra and Relational Calculus

In order to show that our extended relational algebra and relational calculus have the same expressive power, we have to prove the following two propositions:

(1) For every algebraic expression $e \in \mathbf{E}$ there is a closed alpha $\alpha \in \mathbf{A}$ with $e(I) = \alpha(I)$ for all schema instances I.

(2) For every closed alpha $\alpha \in \mathbf{A}$ there is an algebraic expression $e \in \mathbf{E}$ with $\alpha(I) = e(I)$ for all schema instances I.

Proposition (1) can be proven in the usual way (e.g. [Klug82]). Hence we will prove proposition (2) only. It has been proven before by [Klug82] for query evaluation using two-valued logic without null values. His approach is to produce for all terms, formulas and alphas an equivalent algebraic expression. However, his proof is unnecessarily complicated, as it is sufficient to produce an equivalent algebraic expression for all closed alphas.

Our construction of the algebraic expression corresponding to a closed alpha will be similar to Codd's proof of equivalence for queries not having aggregate functions [Codd72]. To solve the special problems associated with the translation of aggregate functions, we introduce an extended selection/restriction operator σ as intermediate representation:

> Let $\psi \in \mathbf{F}$ be a calculus formula, $e \in \mathbf{E}$ an algebraic expression and γ a mapping which maps each term $v_i[A]$, v_i free in ψ, to an attribute name in attrs(e) and leaves constants c unchanged. Then
>
> $$\sigma_{\gamma,\psi}(e)(I) := \{T \in e(I) : \psi(I, \beta_T) = 1\}$$
>
> $$\sigma'_{\gamma,\psi}(e)(I) := \{T \in e(I) : \psi(I, \beta_T) \in \{1, \tfrac{1}{2}\}\}$$
>
> with $\beta_T(v_i)[A] = T[\gamma(v_i[A])]$. The purpose of γ is to bind each free variable in ψ to the corresponding attributes of e. Hence the valuation β_T associates with each free variable in ψ the corresponding attributes of T.

We will first construct for any closed alpha an algebraic expression using the extended selection operator. Afterwards we show how an extended selection can be reduced to a regular algebraic expression.

Construction of the extended algebraic expression:

(0) For a schema relation R_i the corresponding algebraic expression is simply R_i.

$\sigma_{\gamma,\psi_1\vee\psi_2}(e) = \sigma_{\gamma,\psi_1}(e) \cup \sigma_{\gamma,\psi_2}(e)$ $\sigma_{\gamma,\psi_1\wedge\psi_2}(e) = \sigma_{\gamma,\psi_1}(e) \cap \sigma_{\gamma,\psi_2}(e)$ $\sigma_{\gamma,(\neg\psi)}(e) = e \setminus \sigma'_{\gamma,\psi}(e)$ $\sigma_{\gamma,(\bot\psi)}(e) = \sigma_{\gamma,\psi}(e)$ $\sigma_{\gamma,(\top\psi)}(e) = \sigma'_{\gamma,\psi}(e)$ $\sigma_{\gamma,(\forall r(v))\psi}(e) = \sigma_{\gamma,\neg(\exists r(v))(\neg\psi)}(e)$ $\sigma_{\gamma,(\exists r(v))\psi}(e) = \pi_{attrs(e)}\sigma_{\gamma',\psi}(e \times e_r)$ $\sigma_{\gamma,t_1\theta t_2}(e) = \sigma_{\gamma(t_1)\theta\gamma(t_2)}(e)$	$\sigma'_{\gamma,\psi_1\vee\psi_2}(e) = \sigma'_{\gamma,\psi_1}(e) \cup \sigma'_{\gamma,\psi_2}(e)$ $\sigma'_{\gamma,\psi_1\wedge\psi_2}(e) = \sigma'_{\gamma,\psi_1}(e) \cap \sigma'_{\gamma,\psi_2}(e)$ $\sigma'_{\gamma,(\neg\psi)}(e) = e \setminus \sigma_{\gamma,\psi}(e)$ $\sigma'_{\gamma,(\bot\psi)}(e) = \sigma_{\gamma,\psi}(e)$ $\sigma'_{\gamma,(\top\psi)}(e) = \sigma'_{\gamma,\psi}(e)$ $\sigma'_{\gamma,(\forall r(v))\psi}(e) = \sigma'_{\gamma,\neg(\exists r(v))(\neg\psi)}(e)$ $\sigma'_{\gamma,(\exists r(v))\psi}(e) = \pi_{attrs(e)}\sigma'_{\gamma',\psi}(e \times e_r)$ $\sigma'_{\gamma,t_1\theta t_2}(e) = \sigma'_{\gamma(t_1)\theta\gamma(t_2)}(e)$
t_1, t_2 are not of the form $f(\alpha)$; γ' equals γ except that terms of the form $v[A]$ are mapped on the corresponding attribute of e_r	

Table 5: Translation rules for the extended selection operator

Lemma 1: Let $\alpha = (t_1, \ldots, t_n) : r_1(v'_1), \ldots, r_m(v'_m) : \psi$ be an alpha. Then

$\sigma_{\gamma,t\theta f(\alpha)}(e)(I) = \pi_{attrs(e)}\sigma_{\gamma(t)\theta A_f}(e_f)(I)$ and $\sigma'_{\gamma,t\theta f(\alpha)}(e)(I) = \pi_{attrs(e)}\sigma'_{\gamma(t)\theta A_f}(e_f)(I)$

with $e_f = \phi_{<attrs(e),f>}(e \cdot \alpha) \cup (e \setminus \pi_{attrs(e)}(e \cdot \alpha)) \times f(\emptyset)$

and $(e \cdot \alpha) = \pi_{(attrs(e),\gamma'(t_1),\ldots,\gamma'(t_n))}\sigma_{\gamma',\psi}(e \times e_{r_1} \times \cdots \times e_{r_m})$,

where γ' equals γ for the variables free in α and associates with $v'_j[A]$ the corresponding attribute in $e_{r_1} \times \cdots \times e_{r_m}$ and A_f names the attribute of e_f that contains values of the aggregate function.

Proof: For $T \in e(I)$ let β_T be the following valuation: $\beta_T(v_i)[A] := T[\gamma(v_i[A])]$, for each free variable v_i in α. Note that with this definition

$$\alpha(I, \beta_T) = \pi_{(\gamma'(t_1),\ldots,\gamma'(t_n))}\sigma_{\gamma',\psi}(\{T\} \times e_{r_1} \times \cdots \times e_{r_m})(I)$$

and $(e \cdot \alpha)(I) = \pi_{(attrs(e),\gamma'(t_1),\ldots,\gamma'(t_n))}\sigma_{\gamma',\psi}(e \times e_{r_1} \times \cdots \times e_{r_m})(I) = \bigcup_{T\in e(I)}\{T\} \times \alpha(I, \beta_T)$

Now

$$\begin{aligned}
e_f(I) &= \phi_{<attrs(e),f>}(e \cdot \alpha)(I) \cup (e(I) \setminus \pi_{attrs(e)}(e \cdot \alpha)(I)) \times f(\emptyset) \\
&= \phi_{<attrs(e),f>}(\textstyle\bigcup_{T\in e(I)}\{T\} \times \alpha(I, \beta_T)) \cup (e(I) \setminus \pi_{attrs(e)}(\textstyle\bigcup_{T\in e(I)}\{T\} \times \alpha(I, \beta_T))) \times f(\emptyset) \\
&= \{T \circ f(\alpha(I, \beta_T)) : T \in e(I) \wedge \alpha(I, \beta_T) \neq \emptyset\} \cup (e(I) \setminus \{T \in e(I) : \alpha(I, \beta_T) \neq \emptyset\}) \times f(\emptyset) \\
&= \{T \circ f(\alpha(I, \beta_T)) : T \in e(I)\}
\end{aligned}$$

Hence

$$\begin{aligned}
&\pi_{attrs(e)}\sigma_{\gamma(t)\theta f}(e_f)(I) \\
&= \pi_{attrs(e)}\sigma_{\gamma(t)\theta f}(\{T \circ f(\alpha(I, \beta_T)) : T \in e(I)\}) \\
&= \{T \in e(I) : T[\gamma(t)]\theta f(\alpha(I, \beta_T)) = 1\} \\
&= \sigma_{\gamma,t\theta f(\alpha)}(e)(I)
\end{aligned}$$

□

Calculus Query	Relational Algebra Expression
$(v_1[1]) : R_1(v_1) : v_1(2) = f(\alpha)$	$\pi_{(\gamma(v_1[1]))}\sigma_{\gamma,v_1[2]=f(\alpha)}(R_1) = \pi_{(1)}\sigma_{2=4}(e_f)$
$\alpha = v_2[1] : R_2(v_2) : v_1[3]\theta v_2[3]$	$e_f = \phi_{<(1,2,3),f>}\pi_{(1,2,3,\gamma'(v_2[1]))}\sigma_{\gamma',v_1[3]\theta v_2[3]}(R_1 \times R_2) \cup (R_1 \setminus \pi_{(1,2,3)}\sigma_{\gamma',v_1[3]\theta v_2[3]}(R_1 \times R_2)) \times f(\emptyset)$ $= \phi_{<(1,2,3),f>}\pi_{(1,2,3,4)}\sigma_{3\theta 6}(R_1 \times R_2) \cup (R_1 \setminus \pi_{(1,2,3)}\sigma_{3\theta 6}(R_1 \times R_2)) \times f(\emptyset)$
Relations: $R_1(1,2,3)$, $R_2(1,2,3)$;	$\gamma(v_1[i]) = i$, $\gamma'(v_1[i]) = i$ and $\gamma'(v_2[i]) = i + 3$

Example 2: Translation of a calculus query into algebra

For the following steps assume that for any closed alpha α contained in the one currently translated a corresponding algebraic expression e_α has been constructed.

(1) To $\phi_{<X,f>}(\alpha)$ corresponds $\phi_{<X,f>}(e_\alpha)$.

(2) To a range $r_i = \alpha_{i_1} \vee \cdots \vee \alpha_{i_k}$ corresponds $e_{r_i} = e_{\alpha_{i_1}} \cup \cdots \cup e_{\alpha_{i_k}}$.

(3) To $\alpha = ((t_1, \ldots, t_n) : r_1(v_1), \ldots, r_m(v_m) : \psi)$ corresponds

$e_\alpha = \pi_{(\gamma(t_1),\ldots,\gamma(t_n))}\sigma_{\gamma,\psi}(e_{r_1} \times \cdots \times e_{r_m})$,

where γ associates with each term $v_i[A]$ the corresponding attribute of $e_{r_1} \times \cdots \times e_{r_m}$.

In order to translate an extended selection operation $\sigma_{\gamma,\psi}(e)$ into a regular algebraic expression, we have to break down the operation into parts, thereby reducing the complexity of the selection predicate ψ. Most of the translation rules necessary to achieve this reduction are quite simple. These rules are given in table 5. (As $(\top)$ maps UNKNOWN to TRUE, a selection with a predicate $(\top\psi)$ yields exactly the tuples where ψ is evaluated to TRUE or UNKNOWN. A similar rule holds for $(\bot)$.)

Note that the translation of a negation $(\neg)$ transforms σ into σ' and vice versa. This has been a source of inconsistency in earlier approaches that did not include σ'. For example, the translation of an all subquery in [CeGo85] is performed by means of a negation. Therefore it is incorrect, if the predicate in the WHERE clause of the subquery contains an aggregate function that may return the null value as a result.

The only nontrivial rule is concerned with the translation of an extended selection where the selection predicate contains an aggregate function (i.e. an expression $\sigma_{\gamma,t\theta f(\alpha)}(e)$). The idea behind this rule is as follows: We construct an algebraic expression e_f representing $\{T \circ f(\alpha(I,\beta_T)) : T \in e(I)\}$, i.e. to each tuple $T \in e(I)$ the corresponding function value is attached, and obtain the result by applying selection and projection to e_f appropriately.

Basically, e_f is constructed by applying an aggregate formation operator to an expression $(e \cdot \alpha)$ representing the union $\bigcup_{T \in e(I)}\{T\} \times \alpha(I,\beta_T)$, i.e. each tuple $T \in e(I)$ is concatenated with the tuples of $\alpha(I,\beta_T)$. However, this represents only $\{T \circ f(\alpha(I,\beta_T)) : T \in e(I) \wedge \alpha(I,\beta_T) \neq \emptyset\}$, as for empty relations $\alpha(I,\beta_T)$ no tuple is contained in the union. Hence, e_f will contain a second part representing $\{T \circ f(\emptyset) : T \in e(I) \wedge \alpha(I,\beta_T) = \emptyset\}$. Formally the rule is presented in lemma 1. The translation of a calculus query containing an aggregate function into algebra is illustrated in example 2.

5 Processing Strategies for Aggregate Functions

The purpose of this section is to develop a general strategy for the processing of aggregate functions, i.e. of expressions of the form $\sigma_{\gamma,t\theta f(\alpha)}(e)(I)$ (we adopt the notation used in lemma 1) and to present several special strategies following the general approach. Two obvious processing strategies are the following:

- Nested Iteration: Determine $\alpha(I,\beta_T)$ for each $T \in e(I)$, apply the aggregate function and test the selection predicate.
- Algebraic Processing: Determine $\{T \circ f(\alpha(I,\beta_T)) : T \in e(I)\}$ as in lemma 1 and apply selection and projection.

Nested iteration is in general inefficient, as has been shown by [Kim 82]. Algebraic processing has the disadvantage that computation of e_f leads to duplication of work if $|\{f(\alpha(I,\beta_T)) : T \in e(I)\}| < |e(I)|$, as then several function values are computed more than once. Specially, to compute $\{T \circ f(\emptyset) : T \in e(I) \wedge \alpha(I,\beta_T) = \emptyset\}$ is wasted, as f is constant. It is however necessary to compute this set if we want to stay inside relational algebra. What we really would like to do in this case is to find some kind of minimal representation of the aggregate function combined with an efficient method to obtain the value corresponding to a tuple $T \in e(I)$. To state this problem more formally, we define for an arbitrary but fixed expression $\sigma_{\gamma,t\theta f(\alpha)}(e)(I)$ the following function F, which computes $f(\alpha(I,\beta_T))$ for $T \in \hat{\mathbf{N}}^k$, where k is the number of attributes of $e(I)$:

$$F : \hat{\mathbf{N}}^k \to \hat{\mathbf{N}}, \quad F(T) = f(\alpha(I,\beta_T)).$$

In the minimal representation of F each function value would be represented exactly once. The basic idea is to approximate the minimal representation by grouping tuples with identical function values together and representing the function value only once for every group. Following this approach we can present the following general processing strategy for aggregate functions:

General Processing Strategy:

(1) Define an equivalence relation $\cong$ on a set

$$R \supset \{T \in e(I) : \alpha(I,\beta_T) \neq \emptyset\}$$

such that

$$T \cong T' \implies F(T) = F(T')$$

The corresponding equivalence classes are

$$\tilde{T} = \{T' \in R : T \cong T'\}, \; T \in R.$$

(2) Represent $M_F = \{\tilde{T} \circ F(T) : T \in R\}$

in the form of a relation m_F with a set $A_{\tilde{R}}$ of attributes corresponding to $\tilde{T}$, and an attribute A_F corresponding to $F(T)$.

(3) Use m_f to compute $\sigma_{\gamma,t\theta f(\alpha)}(e)(I)$ in one of the following ways:

(3.1) Generally, we can compute F using m_F and integrate selection using the function F into relational algebra:

$$F(T) = \begin{cases} \pi_{A_F}\sigma_{A_{\tilde{R}}=\tilde{T}}(m_F) & \text{if } \tilde{T} \in \pi_{A_{\tilde{R}}}(m_F) \\ f(\emptyset) & \text{otherwise} \end{cases}$$

$$\sigma_{\gamma,t\theta f(\alpha)}(e)(I) = \sigma_{\gamma(t)\theta F(attrs(e))}(e)(I).$$

That is, we compute the result by repeatedly looking up the value of F in m_F.

(3.2) If $f(\emptyset)$ = UNKNOWN, i.e. if $f \neq$COUNT, we can use m_F directly to compute the desired result, as all necessary function values are represented in m_F:

$$\sigma_{\gamma,t\theta f(\alpha)}(e)(I) = \pi_{attrs(e)}\sigma_{\gamma(t)\theta A_F}\sigma_{\widetilde{attrs(e)}=A_{\tilde{R}}}(e(I) \times m_F)$$

According to this general strategy, relational algebra has to be extended to allow equivalence classes and selection using a function, where the mechanism to identify a certain equivalence class depends upon the special equivalence relation chosen.

We are now able to define special processing strategies by specifying the choices made in steps (1) and (2). In the remainder of this section, we will present several strategies with the corresponding choices.

Strategy 1: Minimal representation of F

In the minimal representation of F each function value is represented exactly once:

(1) $R = \{T \in e(I) : \alpha(I, \beta_t) \neq \emptyset\}$,
$T \cong T' \iff F(T) = F(T')$.

(2) It is an open problem to determine the minimal representation m_F corresponding to the equivalence relation defined in (1). Therefore the following strategies intend to approximate the minimal representation.

(Ex.) For the query of example 2 we have

$R = \pi_{(1,2,3)}\sigma_{3\theta 6}(R_1 \times R_2)$
$T \cong T' \iff f(\pi_{(2)}\sigma_{3\theta 6}(\{T\} \times R_2)) = f(\pi_{(2)}\sigma_{3\theta 6}(\{T'\} \times R_2))$

Strategy 2: Optimised algebraic processing

We obtain an optimised version of the algebraic processing strategy (lemma 1) if we represent only the values of F corresponding to attributes of e that are referenced in α: Let $\text{free}(\alpha) = \gamma(v_{i_1}[A_{i_1}], \ldots, v_{i_k}[A_{i_k}])$ be those attributes ($v_{i_1}, \ldots, v_{i_k}$ are the free variables in α).

(1) $R = \{T \in e(I) : \alpha(I, \beta_T) \neq \emptyset\}$,
$T \cong T' \iff T[\text{free}(\alpha)] \equiv T'[\text{free}(\alpha)]$,
$\tilde{T}$ is represented by $T[\text{free}(\alpha)]$.

(2) $m_F = \phi_{<(1,\ldots,k),f>\pi_{(\text{free}(\alpha),\gamma'(t_1),\ldots,\gamma'(t_n))}\sigma_{\gamma',\psi}(e \times e_{r_1} \times \cdots \times e_{r_m})(I)}$

γ' is defined as in lemma 1. The difference between m_f and e_f of lemma 1 is, that function values $f(\emptyset)$ are not represented and a projection of the attributes $\text{free}(\alpha)$ is performed as early as possible.

(Ex.) For the query of example 2 we have

$R = \pi_{(1,2,3)}\sigma_{3\theta 6}(R_1 \times R_2)$
$T \cong T' \iff T[3] \equiv T'[3]$
$m_f = \phi_{<1,f>\pi_{(3,4)}\sigma_{3\theta 6}}(R_1 \times R_2)$

If $f(\emptyset) \neq$ UNKNOWN, we have

$$F(T) = \begin{cases} \pi_{(2)}\sigma_{1 \equiv T[3]}(m_F) & \text{if } T[3] \in \pi_{(1)}(m_F) \\ f(\emptyset) & \text{otherwise} \end{cases}$$

and $\sigma_{\gamma, v_1[2]=f(\alpha)}(R_1) = \sigma_{2=F((1,2,3))}(R_1)$.

If $f(\emptyset) =$ UNKNOWN, we can use m_f directly:

$\sigma_{\gamma, v_1[2]=f(\alpha)}(R_1) = \pi_{(1,2,3)}\sigma_{2=5}\sigma_{3 \equiv 4}(R_1 \times m_f)$.

Strategy 3: Conjunctive equality predicate

If the predicate ψ in α is $\psi = \psi' \wedge v_{i_1}[A_{i_1}] = t_{i_1} \wedge \ldots \wedge v_{i_k}[A_{i_k}] = t_{i_k}$, where $\psi', t_{i_1}, \ldots, t_{i_k}$ do not contain any variables free in α, we can formulate the following strategy (here γ' associates with the variables $v'_1, \ldots, v'_m$ of α the corresponding attributes of $(e_{r_1} \times \ldots \times e_{r_m})$):

(1) $R = \{T \in \mathbf{N}^k : T[\text{free}(\alpha)] \in \pi_{(\gamma'(t_{i_1}),\ldots,\gamma'(t_{i_k}))}\sigma_{\gamma',\psi'}(e_{r_1} \times \ldots \times e_{r_m})(I)\}$,
$T \cong T' \iff T[\text{free}(\alpha)] = T'[\text{free}(\alpha)]$.

(2) $m_F = \phi_{<(1,\ldots,k),f>\pi_{(\gamma'(t_{i_1}),\ldots,\gamma'(t_{i_k}),\gamma'(t_1),\ldots,\gamma'(t_n))}\sigma_{\gamma',\psi'}(e_{r_1} \times \ldots \times e_{r_m})(I)}$.

Note that R and m_F are defined without using e. Consequently m_F can be determined independent of e. The correctness of this strategy follows from the fact, that $R \supset \{T \in e(I) : \alpha(I, \beta_T) \neq \emptyset\}$ because $\alpha(I, \beta_T) \neq \emptyset$ only for tuples $T[\text{free}(\alpha)]$ matched by some $(t_{i_1}, \ldots, t_{i_k})$. For the case $f(\emptyset) =$ UNKNOWN, strategy 3 has been presented by [Kim82].

(Ex.) If 'θ' equals '$=$', we can compute the query of example 2 in the following way:

$R = \{T \in \mathbf{N}^3 : T[3] \in \pi_{(3)}(R_2)\}$,
$T \cong T' \iff T[3] = T'[3]$,
$m_f = \phi_{<1,f>\pi_{(3,1)}}(R_2)$.

The integration of m_f is accomplished as in strategy 2.

Strategy 4: Range predicate

If the predicate ψ in α is $\psi = \psi' \wedge v_0[A_0]\theta t_0$ where ψ' does not contain any variables free in α, we can formulate a strategy, where the equivalence classes are basically intervalls in $\mathbf{N}$. The endpoints of the intervalls will be taken from the set

$GP = \pi_{\gamma'(t_0)}\sigma_{\gamma',\psi'}(e_{r_1} \times \cdots \times e_{r_m})(I) \cup \{-\infty, \infty\}$
$= \{gp_i : 1 \leq i \leq |\, GP \,|, gp_i < gp_{i+1}\}$.

(1) $R = e(I)$,
$T \cong T' \iff \alpha(I, \beta_T) = \alpha(I, \beta'_T) \iff \{gp \in GP : T[\gamma(v_0[A_0])]\theta gp\} = \{gp \in GP : T'[\gamma(v_0[A_0])]\theta gp\}$.

If $\theta \in \{<, \geq\}$, this means
$T \cong T' \iff T[\gamma(v_0[A_0])], T'[\gamma(v_0[A_0])] \in [gp_i, gp_{i+1})$
for some $gp_i, gp_{i+1} \in GP$. $\tilde{T}$ can be represented by gp_i.

If $\theta \in \{>, \leq\}$, this means
$T \cong T' \iff T[\gamma(v_0[A_0])], T'[\gamma(v_0[A_0])] \in (gp_i, gp_{i+1}]$
for some $gp_i, gp_{i+1} \in GP$. $\tilde{T}$ can be represented by gp_{i+1}.

(2) $m_f = \phi_{<gp,f>\pi_{(gp,\gamma'(t_1),\ldots,\gamma'(t_n))}\sigma_{\gamma',\psi' \wedge gp\theta\gamma'(t_0)}(e_{r_1}(I) \times \cdots \times e_{r_m}(I) \times GP)}$

(Ex.) If 'θ' equals '$<$', we can compute the query of example 2 in the following way:

$GP = \pi_{(3)}(R_2) \cup \{-\infty, \infty\}$,
$R = \mathbf{N}^3$,
$T \cong T' \iff T[3], T'[3] \in [gp_i, gp_{i+1}),\ gp_i, gp_{i+1} \in GP$,
$m_f = \phi_{<1,f>\pi(4,1)\sigma_{4<3}}(R_2 \times GP)$.

To compute $F(T)$, we have to determine the gp_i corresponding to $T[3]$ and look up the corresponding function value in m_f. This can be accomplished by sorting m_f on gp_i-values and performing a binary search.

Strategy 4 can be generalised to deal with arbitrary conjunctions and disjunction of range predicates $v_{i_j}[A_{i_j}]\theta_j t_{i_j}$. In order to accomplish this, GP has to be defined as the cross product of sets GP_j, determined as above for each individual range predicate.

Strategies 2-4 should be considered by an optimiser to process queries having aggregates more efficiently. Whether one of the strategies 3,4 is superior to strategy 2 depends on the size

of m_F (the number of function values that are represented) and on the complexity of the computation that is necessary to determine m_F. The optimizer should contain corresponding decision procedures.

6 Summary and Future Work

To recapitulate, we have extended relational algebra and relational calculus to cover a significant subset of SQL. We gave a translation from SQL into relational calculus, thus obtaining a formal definition of the semantics of SQL. We presented a translation from relational calculus into relational algebra, which was a good foundation to study the processing of aggregate functions. Finally, we developed a new processing strategy for aggregate functions.

Our results can be applied in the following areas, thus forming a foundation for more work in these directions:

- In [ANSI85], the semantics of an SQL query is defined by nested iteration evaluation: A subquery is completely evaluated for each tuple of the outer query block. This has been directly implemented in System R for example [Seli79]. However, the direct implementation of this feature is in general inefficient [Kim 82]. Therefore defining the semantics of an SQL query by translation into relational algebra and relational calculus opens up new optimization opportunities:
  - The optimizer can investigate the whole query and is no longer constrained to look at one subquery at a time.
  - The optimizer can use the broad body of knowledge developed for the optimization of relational calculus and relational algebra queries (see [JaKo85] for a survey and further literature).
- The representation of an SQL query in a formal language is useful for proving the equivalence of two queries. This can be applied to detect common subexpressions in one query or in a set of different queries [Jarke85].
- The new transformation from extended relational calculus into relational algebra has a simpler structure than previously known. Therefore an easy comparison of relational algebra and relational calculus query optimization becomes possible.
- The optimization strategies developed for aggregate functions can be integrated into existing optimizers to process queries with aggregates more efficiently.
- Utilizing relational calculus or relational algebra as internal system language of a database management system makes it easier to support a multiple language environment, which is interesting for example in a distributed environment with a backend database machine.

Our future work will focus on the development of an optimizer for SQL queries in a database machine environment that is based on the approach presented here.

References

[ANSI85] ANSC X3H2, (draft proposed) American National Standard Database Language SQL. Washington, Feb. 1985

[CeGo85] S. Ceri, G. Gottlob: Translating SQL into Relational Algebra: Optimization, Semantics and Equivalence of SQL Queries. IEEE Trans. S.E., April 1985, pp. 324-345

[Codd72] E.F. Codd: Relational completeness of data base sublanguages. In Data Base System, R. Rustin, Ed., Prentice Hall, Englewood Cliffs, N.J., 1972

[JaKo83] M. Jarke, J.Koch: Range Nesting: A Fast Method to Evaluate Quantified Queries. Proc. ACM SIGMOD 1983, San Jose, May 1983, pp. 196-206

[JaKo84] M. Jarke, J. Koch: Query Optimization in Database Systems. ACM Computing Surveys, June 1984, pp. 111-152

[Jarke85] M. Jarke: Common Subexpression Isolation in Multiple Query Optimization. In Query Processing in Database Systems, W. Kim, D. Reiner, D. Batory, Eds. Springer, New York 1985, pp. 191-205

[Kies85] W. Kiessling: On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates. Proc. VLDB 1985, pp. 241-250

[Kim 82] W. Kim: On Optimizing an SQL-like Nested Query. ACM TODS, Sept. 1982, pp. 443-469

[Klug 82] A. Klug: Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. Journal of the ACM, Vol. 29, No.3, July 1982, pp. 699-717

[LeVi85] C. Le Viet: Translation and Compatibility of SQL and QUEL Queries. Journ. Inf. Proc., Vol. 8, No. 1, 1985, pp. 1-15

[Maie83] D. Maier: The Theory of Relational Databases. Pitman, London, 1983

[Seli79] P.G Selinger et. al.: Access Path Selection in a Relational Database System. Proc. ACM SIGMOD 1979, Boston, May 1979, pp 23-34

Grammar for the SQL Subset

```
< query>          ::= SELECT   <select list>
                      FROM     <range list>
                      WHERE    <predicate>
                    [ GROUP BY <attribute list>
                    [ HAVING   <predicate> ]    ]
<select list>     ::= <term> [, <term> ...]
<range list>      ::= <range formula>
                      [, <range formula> ...]
<range formula>   ::= <relation name> ( <variable name> )
<attribute list>  ::= <attr_spec> [, <attr_spec> ...]
<predicate>       ::= [NOT] ( <predicate> )
                    | <predicate> {AND|OR} <predicate>
                    | <comparison predicate>
                    | <quantified predicate>
                    | <exists predicate>
<comparison predicate> ::= <term> <comp op> <term>
<quantified predicate> ::= <term> <comp op> SOME <query>
<exists predicate>     ::= EXISTS <query>
<term>            ::= <literal> | <attr_spec> | <aggr_fun>
<attr_spec>       ::= <variable name>.<attribute name>
<aggr_fun>        ::= {AVG|MAX|MIN|SUM|COUNT}(<attr_spec>)
```

<relation name>, <variable name>, <literal> and <comp op>(Θ) are defined as in relational calculus.