International Journal of Foundations of Computer Science
© World Scientific Publishing Company

ON GROUPING IN RELATIONAL ALGEBRA

KIM S. LARSEN
Department of Mathematics and Computer Science
University of Southern Denmark, main campus: Odense University
Campusvej 55, DK-5230 Odense M, Denmark

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

The concept of grouping in relational algebra is well-known from its connection to aggregation. In this paper we generalize the grouping notion by defining a simultaneous grouping of more than one relation, and we discuss the application of operations on grouping elements other than just arithmetic aggregation. Finally we show that this grouping mechanism can be added to relational algebra without increasing its computational power.

1. Introduction

The concept of grouping in relational algebra is well-known from its connection to aggregation, and grouping constructs such as **group_by** [3,4] have been defined in order to incorporate the ideas into relational languages.

However, there is no reason why the use of grouping should be limited to a simple addition, counting, or maximization of a collection of domain values. It might be natural to perform relational computations on groups of a relation before, or even without, applying aggregation. We extend the grouping mechanism to more than one relational argument, and discuss grouping as a general mechanism for posing queries.

A query language [7,8] which offers this more general grouping mechanism has been used for a number of years for teaching and minor administrational tasks at some Danish universities.

In this paper we discuss the essence of such a query language, focusing on the grouping mechanism and the extra possibilities it offers as an addition to relational algebra. Since we extend relational algebra, we also show that the computational power is unchanged. It is of great interest to extend relational algebra in the direction of adding more computational power, but this is a separate issue; it should not be a side-effect of the decisions concerning the issues under consideration here.

We show that the computational power is unchanged by giving a translation of our grouping mechanism into relational calculus. Since the nesting operator [9,5] also involves grouping, a translation for unary grouping can be derived from [10].

2. Examples

In this section we give some examples. All concepts and notation appearing in this section will be defined formally in the following sections.

First we discuss the concept of *grouping of relations.* Let two relations, r_1 and r_2 with schemas R_1 and R_2, be as listed below.

r_1:

A	B
a_1	b_1
a_2	b_2
a_3	b_3

r_2:

B	C	D
b_2	c_1	d_1
b_2	c_2	d_2
b_3	c_3	d_3
b_4	c_4	d_4

First we note that $R_1 \cap R_2 = \{B\}$, and that the (tuple) values in the B-columns are $\pi_B(r_1) \cup \pi_B(r_2) = \{[b_1], [b_2], [b_3], [b_4]\}$. We shall write r_1 and r_2 as a combination of the B-tuples. Clearly, r_1 can be written as

$$\{[b_1]\} \times \{[a_1]\} \;\cup\; \{[b_2]\} \times \{[a_2]\} \;\cup\; \{[b_3]\} \times \{[a_3]\} \;\cup\; \{[b_4]\} \times \{\}$$

and r_2 can be written as

$$\{[b_1]\} \times \{\} \;\cup\; \{[b_2]\} \times \{[c_1, d_1], [c_2, d_2]\} \;\cup\; \{[b_3]\} \times \{[c_3, d_3]\} \;\cup\; \{[b_4]\} \times \{[c_4, d_4]\}$$

We refer to this as the *grouping* of r_1 and r_2 by B.

We define a class of *environments* from a grouping. As usual an environment is simply an association of values with identifiers. As an example, consider the environment

$$\eta_{[b_2]}: \quad \begin{array}{lll} \# & \mapsto & [b_2] \\ @(1) & \mapsto & \{[a_2]\} \\ @(2) & \mapsto & \{[c_1, d_1], [c_2, d_2]\} \end{array}$$

In this example, we have three identifiers, #, @(1), and @(2), each of which has an associated value. This environment consists of the tuple $[b_2]$ together with the relations consisting of the tuples from r_1 and r_2 which contain $[b_2]$; except that in @(1) and @(2), the $[b_2]$-parts of the tuples have been removed. Similarly, we have

$$\eta_{[b_4]}: \quad \begin{array}{lll} \# & \mapsto & [b_4] \\ @(1) & \mapsto & \{\} \\ @(2) & \mapsto & \{[c_4, d_4]\} \end{array}$$

We can evaluate expressions in different environments. As an example, $@(1) \times @(2)$ evaluated in $\eta_{[b_2]}$ (this is denoted $[\![@(1) \times @(2)]\!]\eta_{[b_2]}$) is $\{[a_2, c_1, d_1], [a_2, c_2, d_2]\}$. Similarly, $[\![@(1) \times @(2)]\!]\eta_{[b_4]} = \{\}$. Because of the way the four environments $\eta_{[b_1]}$, $\eta_{[b_2]}$, $\eta_{[b_3]}$, and $\eta_{[b_4]}$ are defined, $[\![@(1) \times @(2)]\!]\eta$ will have the same schema, no matter which of the four environments we use for η. Thus, the union of all four (partial) results is well-defined. The expression

$$\textbf{group } r_1, r_2 \textbf{ by } B \textbf{ do } @(1) \times @(2)$$

denotes exactly this value, $\{[a_2, c_1, d_1], [a_2, c_2, d_2], [a_3, c_3, d_3]\}$. In fact, by this example, we have given an informal semantics of the grouping operator. In standard relational algebra, the relation just computed could have been expressed by $\pi_X(r_1 \bowtie r_2)$, where $X = (R_1 \setminus R_2) \cup (R_2 \setminus R_1)$.

We move on to an example using aggregation. We use the notation $\mathrm{SUM}_A(r)$, where r is a relation which has an integer attribute A. This expression evaluates to the sum of the integers in column A of r.

Consider the relation Sales with schema

$$\{\mathrm{SalesPerson}, \mathrm{District}, \mathrm{Customer}, \mathrm{Amount}\},$$

which contains information about sales by different salespersons, i.e., which salesperson, which district, to whom, and the total value of the sale. In the following, we use the initial letters: S, D, C, and A. Also for brevity, let Jones denote $\pi_{DCA}(\sigma_{S=\mathrm{Jones}}(\mathrm{Sales}))$, and let Miller denote $\pi_{DCA}(\sigma_{S=\mathrm{Miller}}(\mathrm{Sales}))$. We use the following example data:

Jones:

D	C	A
2	Smith	17
7	Lee	20
7	O'Neil	12
7	Brown	2
8	Chang	7

Miller:

D	C	A
7	Clark	25
8	Morrison	9
8	Kent	9
13	Hansen	12

Now for each of the districts numbered from 5 through 20, we want to find the difference of their sales. This can be expressed by

$$\begin{array}{l}\textbf{group } \mathrm{Jones,Miller} \textbf{ by } D \textbf{ do} \\ \quad (5 \leq D) \wedge (D \leq 20)?\, \{\#\,[V\colon\ \mathrm{SUM}_A(@(1)) - \mathrm{SUM}_A(@(2))]\}\end{array}$$

A typical environment looks like this:

$$\begin{array}{lll} \eta\colon & \# & \mapsto & [D\colon\ 7] \\ & @(1) & \mapsto & \{[C\colon\ \text{Lee},\ A\colon\ 20], [C\colon\ \text{O'Neil},\ A\colon\ 12], [C\colon\ \text{Brown},\ A\colon\ 2]\} \\ & @(2) & \mapsto & \{[C\colon\ \text{Clark},\ A\colon\ 25]\} \end{array}$$

For this environment, the boolean expression $(5 \leq D) \wedge (D \leq 20)$ evaluates to true. This means that we proceed to evaluate the expression following the question mark. Had it evaluated to false, the result of the whole evaluation (for this particular environment) would be the empty relation (with schema $\{D, V\}$). As it is, we obtain a relation consisting of one tuple, which is the concatenation of $[D\colon\ 7]$ (the value of $\#$) and $[V\colon\ 9]$ (the value of $[V\colon\ \mathrm{SUM}_A(@(1)) - \mathrm{SUM}_A(@(2))]$). In total, we obtain the following:

D	V
7	9
8	-11
13	-12

For each district, we can find the number of customers that Jones have dealt with, but Miller has not as follows:

$$\textbf{group}\ \text{Jones}, \text{Miller}\ \textbf{by}\ D\ \textbf{do}\ \{\#\,[V\colon\ \text{COUNT}_A(\pi_C(@(1)) - \pi_C(@(2)))]\}$$

Of course, the minus here is relation difference.

New possibilities arise when nesting is allowed. Consider division, usually defined as:

$$r_1/r_2 = \{t \mid \{t\} \times r_2 \subseteq r_1\}$$

We can write

$$\textbf{group}\ r_1, r_2\ \textbf{by}\ R_2\ \textbf{do group}\ @(1)\ \textbf{by}\ R_1 \setminus R_2\ \textbf{do}\ \{\#\} \times r_2 \subseteq r_1?\,\{\#\}$$

Here grouping is with respect to more than one attribute name (R_2 and $R_1 \setminus R_2$ should of course really be written out as a sequence of attribute names). The first grouping provides us with @(1)'s consisting of tuples from r_1 with schema $R_1 \setminus R_2$. The second **group** runs through these tuples one by one and includes them in the result if they pass the test. The second grouping is basically a **forall**-construction, since we group on the entire schema of the (one) argument relation.

3. Notation

We use standard relational algebra and calculus as defined in the original paper [1] or in textbooks [11]. In this section we specify which variants we are using.

We fix a domain $\mathcal{D}$ of all constants that can appear in relations and expressions, and an infinite set $\mathcal{A}$ of attribute names. In this paper there is no reason to distinguish between different types, so schemas are simply finite subsets of $\mathcal{A}$.

We fix the letters we use as names for various entities:

- $a, a_1, a_2, \ldots$ ranges over $\mathcal{D}$ and expressions of that type.
- $A, A_1, A_2, \ldots, B, C$ range over $\mathcal{A}$.
- X, Y range over finite subsets of $\mathcal{A}$.
- $t, t_1, t_2, \ldots, u, x, x_1, x_2, \ldots$ range over tuples.
- $r, r_1, r_2, \ldots, e, e_1, e_2, \ldots$ range over relation names and expressions.
- $R, R_1, R_2, \ldots, E, E_1, E_2, \ldots$ range over schemas of relations and expressions.
- c, c_1, c_2 range over Boolean expressions (conditionals).

A tuple is a total function from a schema into $\mathcal{D}$. We write $t.A$ for t's value on the attribute name A. Constant tuples are listed using square brackets, so $[A_1\colon\ a_1, \ldots A_k\colon\ a_k]$ is the tuple such that $t.A_i = a_i$, $1 \le i \le k$. The empty tuple (the function defined on the empty schema), is written $[\,]$.

A relation is a schema R and a finite set of tuples r. The schema of every tuple $t \in r$ must be R. The notation $R(r)$ is used to indicate that relation r has schema R. As usual, we sometimes overload notation and let r refer to the relation $R(r)$. We need a notation for empty relations with different schemas: if Z is a set of attribute names (a schema), then we let $Z(\emptyset)$ denote the relation with schema Z, but no tuples. Thus, $\emptyset(\emptyset)$ is the empty relation with the empty schema. There are exactly two different relations with the empty schema, namely $\emptyset(\emptyset)$ and $\emptyset(\{[\,]\})$ (the latter is the neutral element with respect to Cartesian product and natural join).

If e is a relational *expression*, then we always let the corresponding capital letter, E, denote the schema of the relation, the value of which the expression e denotes. If t is a tuple, $\{t\}$ denotes the corresponding singleton relation.

We use standard symbols for the relational operators: $\cup$, $-$, $\times$, σ, π, and δ for union, difference, Cartesian product, selection, projection, and renaming, respectively. Select conditions consist of a single equality or inequality between two attribute names or an attribute name and a constant. If $t = [A_1\colon\ a_1, \ldots A_k\colon\ a_k]$ and $\{A_1, \ldots, A_k\} \subseteq R$, then we use $\sigma_t(r)$ as short for $\sigma_{A_k=a_k}(\cdots\sigma_{A_2=a_2}(\sigma_{A_1=a_1}(r))\cdots)$. This could equivalently be written $\{t\} \bowtie r$. We also use the derived operator natural join, $\bowtie$.

Finally, it will be convenient for us to allow projection on an empty set of attribute names. The obvious generalization is that $\pi_\emptyset(r)$ is $\emptyset(\emptyset)$, if r is empty, and $\emptyset(\{[\,]\})$, otherwise.

For relational calculus, we use a tuple-based variant.

The atomic formulas are $t \in r$, where r is a relation, as well as $t.A\,\theta\,t.B$ and $t.A\,\theta\,a$, where A and B are attribute names, a is a constant, and θ is one of $<$, $>$, $=$, $\neq$, $\le$, and $\ge$.

A formula is an atomic formula, or one of $\neg p$, $p \wedge q$, $p \vee q$, $\exists t\colon\ p$ or $\forall t\colon\ p$, where p and q are formulas.

If X is a subset of the schema of some tuple t, then $t[X]$ denotes the tuple t restricted to X, i.e., its schema is X, and for every $A \in X:\ \ t[X].A = t.A$. As a special case, $t[\emptyset]$ is the empty tuple.

A tuple relational query is an expression of the form $\{t \mid p\}$, where p is a formula, and t is the only free variable in p. We use the convention that t is defined on any attribute mentioned in connection with t in the query, e.g., if $t.A$ and $t[X]$ appears in p, then t is defined on $\{A\} \cup X$.

4. Relational Algebra with Grouping

To make the presentation clearer, we initially omit nested groupings and computations on domain values, including aggregation. In the last sections, we discuss how these restrictions are removed. We define the extension of relational algebra which is specified in Fig. 1 using BNF-notation.

$$
\begin{aligned}
\texttt{<atom>} &::= a \mid \#.A \\
\texttt{<tup>} &::= [\,] \mid [A\colon \texttt{<atom>}] \mid \# \\
\texttt{<rel>} &::= r \mid Z(\emptyset) \mid \{\texttt{<tup>}\} \mid \texttt{<cond>}?\,\texttt{<rel>} \mid @(1) \mid @(2) \mid \ldots \mid \\
&\quad\ \texttt{<rel>} \cup \texttt{<rel>} \mid \texttt{<rel>} - \texttt{<rel>} \mid \texttt{<rel>} \times \texttt{<rel>} \mid \\
&\quad\ \sigma_{A\theta C}(\texttt{<rel>}) \mid \pi_{A_1\ldots A_k}(\texttt{<rel>}) \mid \delta_{A\to B}(\texttt{<rel>}) \\
\texttt{<cond>} &::= \texttt{<atom>} \leq \texttt{<atom>} \mid \texttt{<rel>} \subseteq \texttt{<rel>} \mid \\
&\quad\ \texttt{<cond>} \wedge \texttt{<cond>} \mid \texttt{<cond>} \vee \texttt{<cond>} \mid \neg\texttt{<cond>}
\end{aligned}
$$

Figure 1: Grammar for the extended language.

`<atom>`, `<tup>`, `<rel>`, and `<cond>` stand for atomic expression, tuple expression, relational expression, and conditional expression, respectively.

An atomic expression can be a constant or a tuple selection.

A tuple expression can be the empty tuple, a constant tuple, or a grouping tuple.

A relational expression can be a relation name, the empty relation with schema Z, a singleton relation containing only the one specified tuple, a gate expression (which evaluates to its relational argument if the condition evaluates to true), grouping relations, or a standard relational operator applied to relational expressions.

There is the additional syntactical restriction that # must appear in the scope (in the **do**-part) of a grouping expression, and $@(i)$ must appear in the scope of a grouping expression with at least i arguments.

A conditional expression can be a comparison between domain values or relational values, or it can be a Boolean formula over conditional expression.

In order to focus on what is essential, we have left out unnecessary constructions. For instance, all the usual six comparison operators can be expressed using $\leq$ and the Boolean connectives. Similarly for containment of relations, $\subseteq$.

Also, tuple concatenation has been used in the examples, but since the expression in the grouping construction must be relational, tuples can be converted into singleton relations sooner, and Cartesian product can be used instead. For instance, $\{\#\,[R\colon\ \mathrm{SUM}_A(@(1)) - \mathrm{SUM}_A(@(2))]\}$ from an earlier example can be written $\{\#\} \times \{[R\colon\ \mathrm{SUM}_A(@(1)) - \mathrm{SUM}_A(@(2))]\}$.

Finally, in the examples, an attribute name A has denoted the value of the current grouping tuple on A. In the grammar, we require that the explicit form $\#.A$ is used instead.

For now, until we allowed nested applications of grouping, we discuss the language where we have only one grouping construct with a relational expression, defined by the BNF-grammar, in the **do**-part:

$$\textbf{group}\ r_1,\ldots,r_n\ \textbf{by}\ A_1,\ldots,A_k\ \textbf{do}\ \texttt{<rel>}$$

We require that every attribute A_j in the **by**-part appears in every schema R_i. In other words, $\{A_1,\ldots,A_k\} \subseteq \bigcap_i R_i$.

We now formally define the meaning of such an expression. We do this in terms of the standard relational operators, but notice that this does not define a standard relational algebra expression, since, among other things, we use a data dependent number of unions. Compare with the first example in Section 2.

Definition 1 *Let* $\mathcal{F} = \textbf{group } r_1, \ldots, r_n \textbf{ by } A_1, \ldots, A_k \textbf{ do } e$, *where*

$$X = \{A_1, \ldots, A_k\} \subseteq \bigcap_i R_i$$

The semantics of $\mathcal{F}$ *is denoted* $[\![\mathcal{F}]\!]$ *and defined by*

$$[\![\mathcal{F}]\!] = \bigcup_{t \in b_\mathcal{F}} [\![e]\!]\eta_t$$

where $b_\mathcal{F} = \bigcup_i \pi_X(r_i)$, *and for each* $t \in b_\mathcal{F}$ *and* $i \in \{1, \ldots, n\}$,

$$r_\mathcal{F}^{i,t} = \pi_{R_i \setminus X}(\sigma_t(r_i))$$

and, finally, for each $t \in b_\mathcal{F}$, *we define the environment* η_t *by*

$$\eta_t\colon \begin{array}{ccc} \# & \mapsto & t \\ @(1) & \mapsto & r_\mathcal{F}^{1,t} \\ \vdots & \vdots & \vdots \\ @(n) & \mapsto & r_\mathcal{F}^{n,t} \end{array}$$

Note that # is always bound to a tuple value with schema X, and $@(i)$ is a relation with schema $R_i \setminus X$.

5. Computational Power Is Preserved

We show how a grouping expression can be translated into relational calculus. Since relational algebra and calculus are equivalent [2], this shows that grouping expressions can also be translated into relational algebra. Since the grouping expressions contain a complete [2] version of relational algebra, grouping expressions are equivalent to relational algebra.

An expression **group** $r_1, \ldots, r_n$ **by** $A_1, \ldots, A_k$ **do** e is translated into

$$\{x \mid \exists t_\#\colon \bigvee_i (\exists t_i\colon\ t_i \in r_i \wedge t_i[X] = t_\#) \wedge \mathcal{T}(e)\}$$

where the translation of e, $\mathcal{T}(e)$, is defined in Fig. 2.

In that table, we use $t_\#$ for values corresponding to #, and t_i for tuples belonging to the relation r_i. Additionally, we use X for the set $A_1, \ldots, A_k$, i.e., the schema of $t_\#$ (and #).

With regards to scope of variables, we assume that in each step of a recursive translation, parenthesis are used around the entire formula in the right column of the table. The scope of universal and existential quantification extends as far to the right as it is meaningful, i.e., to the first unmatched right parenthesis.

Finally, we use the following convention in order to make the table readable. The translation is compositional, so the translation of a composite expression is given in terms of the translations of the components. Expressions are translated into formulas with exactly one free variable. When a composite expression is to be translated, we assume that the first expression is translated into a formula, where the only free variable is x_1, and the second expression is translated into a formula, where the only free variable is x_2. For the only free variable in the result, we use the variable x. When translating a formula recursively, this means that after a subexpression has been translated, the free variable, x, must be renamed to x_1 (other occurrences of x_1 should first be renamed to some new variable).

exp	$\mathcal{T}$(exp)
Domain expressions	
a	a
$\#.A$	$t_{\#}.A$
Tuple expressions	
$[\,]$	$x = [\,]$
$[A\colon\ a]$	$x.A = \mathcal{T}(a)$
$\#$	$x = t_{\#}$
Relation expressions	
r_i	$x \in r_i$
$Z(\emptyset)$	*false*
$\{u\}$	$\mathcal{T}(u) \wedge x = x_1$
$@(i)$	$\exists t_i\colon\ t_i \in r_i \wedge t_{\#} = t_i[X] \wedge x = t_i[R_i \setminus X]$
$c?\, e_1$	$\mathcal{T}(c) \wedge \exists x_1\colon\ \mathcal{T}(e_1) \wedge x = x_1$
$e_1 \cup e_2$	$(\exists x_1\colon\ \mathcal{T}(e_1) \wedge x = x_1) \vee (\exists x_2\colon\ \mathcal{T}(e_2) \wedge x = x_2)$
$e_1 - e_2$	$(\exists x_1\colon\ \mathcal{T}(e_1) \wedge x = x_1) \wedge \neg(\exists x_2\colon\ \mathcal{T}(e_2) \wedge x = x_2)$
$e_1 \times e_2$	$\exists x_1\colon\ \exists x_2\colon\ \mathcal{T}(e_1) \wedge \mathcal{T}(e_2) \wedge x[E_1] = x_1 \wedge x[E_2] = x_2$
$\sigma_{A\,\theta\,a}(e_1)$	$\exists x_1\colon\ \mathcal{T}(e_1) \wedge x = x_1 \wedge x.A\,\theta\,a$
$\sigma_{A\theta B}(e_1)$	$\exists x_1\colon\ \mathcal{T}(e_1) \wedge x = x_1 \wedge x.A\,\theta\,x.B$
$\pi_Y(e_1)$	$\exists x_1\colon\ \mathcal{T}(e_1) \wedge x = x_1[Y]$
$\delta_{A\to B}(e_1)$	$\exists x_1\colon\ \mathcal{T}(e_1) \wedge x = x_1[E_1 \setminus \{A\}] \wedge x.B = x_1.A$
Conditional expressions	
$a_1 \leq a_2$	$\mathcal{T}(a_1) \leq \mathcal{T}(a_2)$
$e_1 \subseteq e_2$	$\forall x_1\colon\ \mathcal{T}(e_1) \Rightarrow \exists x_2\colon\ \mathcal{T}(e_2) \wedge x_1 = x_2$
$c_1 \wedge c_2$	$\mathcal{T}(c_1) \wedge \mathcal{T}(c_2)$
$c_1 \vee c_2$	$\mathcal{T}(c_1) \vee \mathcal{T}(c_2)$
$\neg c_1$	$\neg\mathcal{T}(c_1)$

Figure 2: Translation into relational calculus.

Example 1 *Consider the following expression:*

$$\textbf{group}\ r_1, r_2\ \textbf{by}\ R_1 \cap R_2\ \textbf{do}\ @(1) \times @(2)$$

In relational algebra, this would be expressed as:

$$\pi_{(R_1 \cup R_2)\setminus(R_1 \cap R_2)}(r_1 \bowtie r_2)$$

In both cases, the attributes on which to group or project would be given explicitly, though. Let X be the set of attributes $R_1 \cap R_2$.

Changing variable names whenever necessary, the translation defined above results in:

$$\begin{aligned}\{x \mid \exists t_\#:\ & \textstyle\bigvee_i(\exists t_i:\ t_i \in r_i \wedge t_i[X] = t_\#) \wedge \\ & \exists x_1:\ \exists x_2:\quad \exists t_1:\ t_1 \in r_1 \wedge t_\# = t_1[X] \wedge x_1 = t_1[R_1 \setminus X] \\ & \qquad\qquad\quad \wedge\ \exists t_2:\ t_2 \in r_2 \wedge t_\# = t_2[X] \wedge x_2 = t_2[R_2 \setminus X] \\ & \qquad\qquad\quad \wedge\ x[R_1 \setminus R_2] = x_1 \wedge x[R_2 \setminus R_1] = x_2\}\end{aligned}$$

We state the following result.

Theorem 1 *For any sequence of relations, $r_1, \ldots, r_n$, any set of attribute names $X \subseteq \bigcap_i R_i$, and any relation expression e,*

$$[\![\textbf{group } r_1, \ldots, r_n \textbf{ by } X \textbf{ do } e]\!] = \{x \mid \exists t_\#:\ \bigvee_i (\exists t_i:\ t_i \in r_i \wedge t_i[X] = t_\#) \wedge \mathcal{T}(e)\}$$

Proof. Note that $t_\# \in \bigcup_i \pi_X(r_i)$ if and only if $\bigvee_i(\exists t_i:\ t_i \in r_i \wedge t_i[X] = t_\#)$ is true. Thus, the result can be established by showing that for all environments, $\eta_{t'}$, where $t' \in \bigcup_i \pi_X(r_i)$, we have that $[\![e]\!]\eta_{t'} = \{x \mid t_\# = t' \wedge \mathcal{T}(e)\}$.

This equality can be shown by an induction argument, by going through the translations defined in Fig. 2. The proof is simple, but lengthy, and has been omitted. □

6. Nested Groupings

In this section we briefly consider the necessary and desired additions when nested groupings are allowed, and we give the extensions to the semantics and to the translation into relational calculus.

First the BNF-grammar from Fig. 2 is extended with the following:

$$\begin{aligned}\texttt{<rel>} ::= \ & \textbf{group } \texttt{<rel>}, \ldots, \texttt{<rel>} \textbf{ by } A_1, \ldots, A_k \textbf{ do } \texttt{<rel>} \mid \\ & \backslash@(1) \mid \backslash@(2) \mid \cdots \mid \backslash\backslash@(1) \mid \backslash\backslash@(2) \mid \cdots \\ \texttt{<tup>} ::= \ & \backslash\# \mid \backslash\backslash\# \mid \cdots\end{aligned}$$

So now groupings can appear anywhere a relational expression is expected. When a grouping is used in the **do**-part of another grouping, two sets of grouping tuples and grouping relations are active at a time. The intention is that # and the @(i)'s refer to the nearest grouping and \# and the \@(i)'s refer to the grouping one level further out. So we must require that if, for instance, \@(i) is used, then the grouping one level out has at least i arguments. In general, the number of \'s specifies the level relatively to the current.

As an example, consider the query

$$\textbf{group } r_1, r_2 \textbf{ by } Y \textbf{ do group } @(2), r_3 \textbf{ by } Z \textbf{ do } @(2) \times \backslash@(1)$$

where Y and Z are some sets of attribute names. Here the first @(2) comes from r_2, the second @(2) comes from r_3, and \@(1) comes from r_1.

To be precise about the meaning of a nested query, we extend the definition of the semantics of grouping. First, if η is an environment, then we let $\backslash\eta$ denote the environment where all names have been equipped with an extra \, e.g., if η was defined on the set of names $\{\#, @(1), \backslash\#, \backslash@(1), \backslash@(2)\}$, then $\backslash\eta$ is defined on $\{\backslash\#, \backslash@(1), \backslash\backslash\#, \backslash\backslash@(1), \backslash\backslash@(2)\}$. Assuming that η' and η'' are defined on disjoint sets of names, we use $\eta = \eta' \oplus \eta''$ to denote the combination of the two environments, where η is defined on the union of the two sets of names, and the value on a given name is the value from either η' or η''.

When extending Definition 1, the crucial change is that grouping can now appear in the expression e. Thus, we must be able to evaluate a grouping construction in an environment. Using the notation just defined, the semantics of $[\![\textbf{group } r_1, \ldots, r_n \textbf{ by } X \textbf{ do } e]\!]\eta$ should be $\bigcup_{t \in b_{\mathcal{F}}} [\![e]\!](\backslash\eta \oplus \eta_t)$.

When defining the translation of nested queries into relational calculus, we must introduce a variable $t'_{\#}$ for each nested grouping. This should be a new variable name each time since we want it to be in the scope of the $t_{\#}$'s defined at levels further out. Translation of #'s preceded by a number of \'s, is then simply a matter of using the correct $t'_{\#}$. Grouping relations are handled similarly.

7. Aggregation and Computation on Domain Values

We extend the BNF-grammar from Fig. 1 with the following:

$$\texttt{<atom>} ::= \mathrm{AGGR}_A(\texttt{<rel>}) \mid \texttt{<atom>}\ \mathrm{op}\ \texttt{<atom>}$$

Thus, an atomic expression can now also be an aggregation of a given attribute over a given relation. Here, AGGR could be any of the usual aggregate functions, i.e., SUM, COUNT, MAXIMUM, etc. Furthermore, an atomic expression could be some operation applied to other atomic expressions, e.g., one of the usual arithmetic operations addition, subtraction, multiplication, or division.

Since we have seen how we can translate any relational expression into relational calculus, we can also express the result of an aggregation, assuming that relational calculus is extended with some appropriate construction for this [6]. We could use a construction such as $\exists t_{\mathrm{SUM_A}} :\ p$, where t is the only free variable in p. The value of this expression is the sum of the $t.A$'s for all t which fulfill p.

The translation does not, in principle, become any harder. However, since relation expressions can now appear inside tuple expressions (via domain expressions), we cannot just use a compositional translation as in Fig. 2. Instead, we have to define a separate analysis and translation of tuples. This is simple, but cumbersome, and has been omitted.

Acknowledgments

The author would like to thank an anonymous referee for suggesting that the translation of the extended relational algebra be made into relational calculus instead of into relational algebra.

References

1. E. F. Codd, "A Relational Model of Data for Large Shared Data Bases", *Communications of the ACM*, **13**(6) (1970) 377–387.

2. E. F. Codd, “Relational Completeness of Data Base Sublanguages”, In: Data Base Systems, Randall Rustin (ed.), 65–98 (Prentice-Hall, 1972).
3. Peter M. D. Gray, “The GROUP_BY Operation in Relational Algebra”, In: Databases, S. M. Deen, P. Hammersley (eds.), 84–98 (Pentech Press Limited, 1981).
4. Peter M. D. Gray, “Logic, Algebra and Databases”, (Ellis Horwood Limited, 1984).
5. G. Jaeschke, H. J. Schek, “Remarks on the Algebra on Non First Normal Form Relations”, In: Proceedings of the 1st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 124–138 (ACM Press, 1982).
6. Anthony Klug, “Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions”, Journal of the ACM, **29**(3) (1982) 699–717.
7. Kim S. Larsen, “RASMUS User’s Manual”, MD-60, Computer Science Department, Aarhus University, 1992.
8. Kim S. Larsen, Michael I. Schwartzbach, Erik M. Schmidt, “A New Formalism for Relational Algebra”, *Information Processing Letters*, **41**(3) (1992) 163–168.
9. Akifumi Makinouchi, “A Consideration on Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model”, In: Proceedings of the Third International Conference on Very Large Data Bases, 447–453 (IEEE Computer Society Press, 1977).
10. Jan Paredaens, Dirk van Gucht, “Converting Nested Algebra Expressions into Flat Algebra Expressions”, *ACM Transactions on Database Systems*, **17**(1) (1992) 65–93.
11. J. D. Ullman, “Principles of Database and Knowledge-Base Systems, Vol. 1”, (Computer Science Press, 1988).