Efficient Evaluation of Aggregates on Bulk Types

Sophie Cluet
INRIA
BP 105
Domaine de Voluceau
78153 Le Chesnay Cedex
France
Sophie.Cluet@inria.fr

Guido Moerkotte
Lehrstuhl für Informatik III
RWTH-Aachen
Ahornstr. 55
52074 Aachen
Germany
moer@gom.informatik.rwth-aachen.de

October 6, 1995

Abstract

A new method for efficiently evaluating queries with aggregate functions is presented. More specifically, we introduce a class of aggregate queries where traditional query evaluation strategies in general require $O(n^2)$ time and space in the size of the (at most two) input relations. For this class of aggregate queries our approach needs at most $O(n \log n)$ time and linear space. Further, our approach deals not only with relations but with general bulk types like sets, bags, and lists.

1 Introduction

Many queries involve the application of functions like *count*, *sum*, *avg* [15]. Among these queries, two different classes can be detected: *scalar aggregates* and *aggregate functions* [10]. Scalar aggregates return a single number from an input relation. Examples thereof are *count*, *sum*, *avg* etc. Aggregate functions, on the other hand, return a set of values for a given relation. Aggregate functions typically involve grouping. Then, a scalar aggregate is applied to each group. As an example consider the query retrieving the number of employees for each department. In this paper we are concerned with the efficient evaluation of aggregate functions.

At the source level of an SQL-like query language, aggregate functions are often expressed as nested queries (Type JA of Kim's classification [14]). The original proposal to evaluate these queries is to perform a nested loop [1]. In order to eliminate the inherent inefficiency of this approach, Kim suggests a more efficient method [14]: perform a join between the relations of the outer and inner block, group the result, and then apply the scalar aggregate to each group. Clearly, if the join is an equi-join, this can be efficiently implemented. Subsequently, several bugs in this approach have been detected and corrected [9, 11, 13]. The main point is to use an outer join instead of a regular join. Nevertheless, the main idea — join, group, aggregate — still remains the same and is even today the only known improvement over nested loops [7, 16, 17]. Let us call this approach A-G-J. Lately, it was proposed to exchange grouping and aggregate operations with joins [19, 18, 20, 5, 6]. This work is orthogonal to our's in that our method can be used to implement the grouping and aggregate operations used there.

Consider the performance of this approach if the join is a non-equi join, e.g., a $\leq$-join, or $\neq$-join. Then, if the two relations involved contain n and m tuples each, the size of the output in general is $O(n * m)$. Hence, the approach needs $O(n * m)$ time and space. If both relations are the same, which is frequently the case, the A-G-J approach needs $O(n^2)$ time and space.

The goal of the paper is to improve this to $O(n \log n + m \log n)$ and $O(n \log n)$. At the core of our approach is a special parameterized abstract datatype which we call *θ-table* for a comparison operator θ (e.g. $\theta \in \{=, \leq, \neq\}$). A θ-table allows a linear space representation of all stages of intermediate results of aggregate functions. Thereby, it avoids the explicit representation of the (intermediate) result of the join which is the crucial point of inefficiency of the A-G-J approach[1]. Further, we show that θ-tables can be implemented and that the result of the aggregate function can be extracted efficiently. This efficiency is achieved by exploiting a simple observation on aggregates: they are decomposable and reversible (cf. Sec. 2).

The paper is outlined as follows. Section 2 gives some basic notations and the definitions for *decomposable* and *reversible* aggregates. Section 3 introduces the abstract datatype θ-table. It contains a short motivating introduction, the definition of θ-tables together with an outline of their efficient implementation, and some example applications. Section 4 concludes the paper.

2 Preliminaries

Since we do not want to restrict our approach to the relational case, we consider aggregates on bulk type instances. Typical bulk types are sets, lists, and bags, denoted by *set*, *list*, and *bag*, respectively. They all contain a number of elements of some type. To abstract from this type, we introduce type variables, denoted by τ. To indicate that the elements of a bulk type are restricted to type τ, we write $bulk(\tau)$. We use the upper case letters X, Y, and Z, to denote bulk type instances. The elements thereof are denoted by their lower case equivalents x, y, and z. A singleton bulk type instance containing solely the element x is denoted by $bulk(x)$.

For any given bulk type *bulk*, we assume the existence of $\emptyset$ and $+$. For the above bulk types *set*, *list*, and *bag*, these symbols denote

- the empty set and disjoint set union, if $bulk = set$,
- the emtpy list and the append operation, if $bulk = list$, and
- the empty bag and bag union, if $bulk = bag$.

[1] In fact, it is possible to implicitly represent the result of any θ-join with a θ-table in linear space. Hence, this opens the road for further applications like the evaluation of queries involving a sequence of non-equi joins (cf. Sec. 3.4). Nevertheless, this point is beyond the scope of the paper.

Operations corresponding to $+$ can be defined for all bulk types (see, e.g., [2]: their bulk constructor C can be used to define $+$). For all bulk types, we denote membership by the symbol $\in$. Further, we assume the existence of an iterator *foreach* which iterates over the elements of a bulk instance where duplicates are iterated over as many times as they are present and bulk type orders are adhered to. This roughly corresponds to the *apply-to-all* operator [2] but *foreach* will be used at a lower level of abstraction as a means for expressing query evaluation plans. At the algebraic level, we use the χ operator instead. For a function $f : \tau_1 \rightarrow \tau_2$, the signature of χ_f is $\chi_f : \mathit{bulk}(\tau_1) \rightarrow \mathit{bulk}(\tau_2)$, and its semantics is defined with $\chi_f(\emptyset) = \emptyset$, $\chi_f(\mathit{bulk}(x)) = \mathit{bulk}(f(x))$ and $\chi_f(x + y) = \chi_f(x) + \chi_f(y)$. χ is sometimes called *map* operator. For convenience, we introduce two abbreviations concerning χ. If τ is a tuple type and X is an instance of a bulk type $\mathit{bulk}(\tau)$, we abbreviate $\chi_{\lambda x.x \circ [a:f]}(X)$ by $\chi_{a:f}(X)$ to denote the extension of a tuple by an additional attribute a containing the application of f to the original tuple. As usual, $\circ$ denotes tuple concatenation. We further abbreviate $\chi_{\lambda x.[x:x]}(X)$ by $X[x]$. This expression builds a bulk of tuples of a single attribute x whose values are the original element x of a bulk type X. Hence, any $\mathit{bulk}(\tau)$ instance is turned into a $\mathit{bulk}([x : \tau])$ instance. The main motivation behind these definitions is that it is quite convenient to deal with bulks of tuples only [7, 12]. The last operator needed on bulk types is selection. For a function $p : \tau \rightarrow \mathit{Bool}$, it is denoted by σ_p and has the signature $\sigma_p : \mathit{bulk}(\tau) \rightarrow \mathit{bulk}(\tau)$. The definition is given by $\sigma_p(\emptyset) = \emptyset$, $\sigma_p(x+y) = \sigma_p(x) + \sigma_p(y)$, and $\sigma_p(\mathit{bulk}(x)) = \mathit{bulk}(x)$ for a singleton $\mathit{bulk}(x)$, if $p(x)$ and $= \emptyset$ otherwise.

After these preliminaries, we can now concentrate on aggregates. In order to keep the definition of aggregates as general as possible, we use the special type symbol $\mathcal{N}$ to denote the codomain of an aggregate. Typically, $\mathcal{N}$ is a numeral data type such as *integer* or *float*. Within the paper, we further need $\mathcal{N}'$ being tuples of numeral data types. That is, $\mathcal{N}'$ has a signature of kind $[a_1 : \tau_1, \ldots, a_n : \tau_n]$ where each of the τ_n is a numeral.

A *scalar* aggregate f is a function

$$f : \mathit{bulk}(\tau) \rightarrow \mathcal{N}.$$

A scalar aggregate $f : \mathit{bulk}(\tau) \rightarrow \mathcal{N}$ is called *decomposable*, if there exist functions

$$\begin{aligned} \alpha : \mathit{bulk}(\tau) &\rightarrow \mathcal{N}' \\ \beta : \mathcal{N}', \mathcal{N}' &\rightarrow \mathcal{N}' \\ \gamma : \mathcal{N}' &\rightarrow \mathcal{N} \end{aligned}$$

with

$$f(Z) = \gamma(\beta(\alpha(X), \alpha(Y)))$$

for all X, Y, and Z with $Z = X + Y$. This condition assures that $f(Z)$ can be computed on arbitrary subsets (-lists, -bags) of Z independently and the (partial) results can be joined to yield the correct (total) result.

A decomposable scalar aggregate $f : \mathit{bulk}(\tau) \rightarrow \mathcal{N}$ is called *reversible* if for β there exists a function $\beta^{-1} : \mathcal{N}', \mathcal{N}' \rightarrow \mathcal{N}'$ with

$$f(X) = \gamma(\beta^{-1}(\alpha(Z), \alpha(Y)))$$

for all X, Y, and Z with $Z = X + Y$. This condition assures that we can compute $f(X)$ for a subset (-list, -bag) X of Z by "subtracting" its aggregated complement Y from the "total" $\beta(\alpha(Z))$ by using β^{-1}.

The fact that scalar aggregates can be decomposable and reversible is the basic observation uppon which the efficient evaluation of aggregate functions builds.

As an example consider the scalar aggregate $avg : bag([a : float]) \rightarrow float$ averaging the values of the attributes a of a bag of tuples with a single attribute a. It is reversible with

$$\begin{aligned}
\alpha : \{[a : float]\} &\rightarrow [sum : float, count : float] \\
\beta : [sum : float, count : float], [sum : float, count : float] &\rightarrow [sum : float, count : float] \\
\beta^{-1} : [sum : float, count : float], [sum : float, count : float] &\rightarrow [sum : float, count : float] \\
\gamma : [sum : float, count : float] &\rightarrow float
\end{aligned}$$

where

$$\begin{aligned}
\alpha(X) &= [sum : sum(X.a), count : |X|] \\
\beta([sum : s_1, count : c_1], [sum : s_2, count : c_2]) &= [sum : s_1 + s_2, count : c_1 + c_2] \\
\beta^{-1}([sum : s_1, count : c_1], [sum : s_2, count : c_2]) &= [sum : s_1 - s_2, count : c_1 - c_2] \\
\gamma([sum : s, count : c]) &= s/c
\end{aligned}$$

$sum(X.a)$ denotes the sum of all values of attribute a of the tuples in X, and $|X|$ denotes the cardinality of X. Note that $\alpha(\emptyset) = [sum : 0, count : 0]$, and $\gamma([sum : 0, count : 0])$ is undefined as is $avg(\emptyset)$.

3 The Abstract Data Type θ-Table

3.1 Motivating Introduction

Our goal is to apply our technique to queries whose translation into the algebra contains subexpressions of the form

$$\chi_{a:f(\sigma_p(\sigma_{p_y}(\sigma_{p_x}(Y[y]))))}(\sigma_{q_x}(X[x]))$$

where

1. p is a predicate involving x and y,
2. p_x is a predicate involving x only,
3. p_y is a predicate involving y only,
4. q_x is a predicate involving x only,
5. f is a decomposable/reversible scalar aggregate, and
6. a is a new attribute unequal to x and y.

Let us call the class of these queries Θ.

Since the treatment of the additional selections σ_{p_x}, σ_{p_y} and σ_{q_x} is rather trivial and we do not care about the actual construction of the tuples from the elements in X and Y, we further restrict our discussion to expressions of the form

$$(*) \qquad \chi_{a:f(\sigma_p(Y))}(X).$$

where X and Y are bulk type instances containing tuples. Nevertheless, the extensions of the algorithms below to the general case are straight forward.

Before we proceed, let us consider two example queries contained in the class Θ. The first query retrieves *all managers together with the number of employees earning more than the manager*. The translation into the algebra yields

$$\chi_{a:count(\sigma_{m.salary \leq e.salary}([e:Emp]))}(Mgr[m]).$$

This directly matches the form (*).

The second query demonstrates the special case $X = Y$ and — at a first sight — looks more complex than the first one. The query retrieves *all students together with the percentage of students better than the student*. We measure *better* by a higher *gpa*. The translation of the second query into the algebra yields

$$\chi_{s:s,b:b/Stud.card}(\chi_{b:count(\sigma_{b.gpa>s.gpa}(Stud[b]))}(Stud[s]))$$

where *Stud.card* denotes the cardinality of all students. If not materialized, it can be computed during the processing of the expression

$$\chi_{b:count(\sigma_{b.gpa>s.gpa}(Stud[b]))}(Stud[s])$$

which is of the form (*).

In order to further simplify the subsequent discussion and not to overwhelm the reader with the technical subtleties of the approach, we further restrict the predicate p of (*) to the form

$$x.a\theta y.b.$$

for $\theta \in \{\leq, \neq\}$. Other comparison operators can be treated similarly. More complex predicates, involving (implicit) conjunctions like $x.a - c_1 \leq y.b \leq x.a + c_2$ for constants c_i, can easily be treated by expanding the $\leq$ case discussed below by exploring reversibility as exemplified below for $\neq$.

The introduction of θ-tables is easier to understand when their usage is clear already. Hence, we illustrate their usage without giving their definition but simply explain verbally the functionality and complexity of the evaluation with and without θ-tables. For this, let us consider yet another simple example query: *Retrieve all employees together with the number of employees earning less* (to cheer them up). If all employees are tuples and are contained in a set X, and s denotes the salary of each employee, then translation into the algebra yields

$$\chi_{a:count(\sigma_{y.s \leq x.s}(X[y]))}(X[x]).$$

The additional information — the result of count — will be contained in the additional attribute a which is added during the evaluation of χ.

Let us first evaluate the costs imposed by the A-G-J approach. If the number of employees is n, and their salaries are equally distributed, then the size of the $\leq$-join to be computed for the A-G-J evaluation is $n * n/2$. Hence, the time and space complexity of this approach is $O(n^2)$.

Opposedly, we will apply the following general procedure to evaluate algebraic expressions of the form (*) in the case $X = Y$:

```
(1)   tt := new ≤-table;
(2)  foreach (x in X)
(3)       tt.insadd(x);
(4)  tt.sort();
(5)  tt.eval();
```

(1) creates a new θ-table *tt* which takes constant time. (2) and (3) insert all elements (x) into *tt*. Each insertion takes constant time. (4) performs some kind of sorting and processing of duplicates. This is the most expensive step taking $O(n \log n)$ time. (5) then performs the last step of evaluating the aggregate. Whereas α and β of, e.g., *count* have been applied during *insadd* already, γ is applied during *eval*. This step takes $O(n)$. Since the space consumed by the $\leq$-table is linear in n, the analysis of this evaluation yields the time complexity $O(n \log n)$ and space requirements of $O(n)$ which clearly outperforms the corresponding A-G-J approach.

In case $X \neq Y$, the general evaluation procedure of (*) is

```
(1)  tt := new θ-table;
(2) foreach (x in X)
(3)      tt.ins(x);
(4) tt.sort();
(5) foreach (y in Y)
(6)      tt.add(y);
(7) tt.eval();
```

The only difference is that the information of Y is added to the θ-table in a distinct step (4) and (5). The complexity of this algorithm will be $O(n)$ for inserting the n elements of X, $O(n \log n)$ for sorting, $O(m \log n)$ for adding the m elements of Y, and $O(n)$ for *eval*. Hence, we reach the promised $O(n \log n + m \log n)$.

3.2 $\leq$-Tables

Let us start by giving the signature of a θ-table which depends on several parameters:

1. the comparison operator θ,
2. the types $bulk_x(\tau_x)$ and $bulk_y(\tau_y)$ of X and Y, respectively, and
3. the signature of the component β of the aggregate f to be computed.

X
a
10
3
7
3

Y
b
2
3
4

Figure 1: The Extensions of two bags of tuples with a single attribute

[n:1, x:[a: 10], f: 0]	[n:2, x:[a: 3], f: 0]	[n:2, x:[a: 3], f: 2]	[n:2, x:[a: 3], f: 2]
[n:1, x:[a: 3], f: 0]	[n:1, x:[a: 7], f: 0]	[n:1, x:[a: 7], f: 1]	[n:1, x:[a: 7], f: 3]
[n:1, x:[a: 7], f: 0]	[n:1, x:[a:10], f: 0]	[n:1, x:[a:10], f: 0]	[n:1, x:[a:10], f: 3]
[n:1, x:[a: 3], f: 0]			

Figure 2: Different States of a θ-Table

The $\leq$-table has the signature

$$list[n : int, x : \tau_x, f : \mathcal{N}']$$

if $\beta : \mathcal{N}', \mathcal{N}' \rightarrow \mathcal{N}'$. The attribute n counts the number of occurrences of the element x in X. Special care has to be taken not to destroy the original order of $bulk_x$ if there is one. Nevertheless, we omit this technical issue in the present extended abstract. Hence, we just assume that the elements x of X are directly contained in the attribute x of the θ-table. The third attribute, f, contains the result of applying α and β to some subset of X. This subset is determined by the predicate $p = y.b \leq x.a$. More specifically, the value of f in the entry containing $x \in X$ is the same as applying α to the set of all y with $y.b \leq x.a$.

Since a θ-table will always be of type *list*, it is necessary, to convert the θ-table during *eval* into the bulk type $bulk_y$ but we will skip this trivial step since it also depends on operators occurring outside (*), e.g., projections, other conversions like sort, unique etc.

We implement θ-tables as arrays. Hence, linear space will suffice and sorting can be done in place. Further, insertion via binary search takes $O(\log n)$ if the θ-table contains n entries. Typical extensions of a $\leq$-table are shown in Fig 2. An alternative is to implement a θ-table as a balanced binary tree or $B - tree$ but note that this consumes $O(n \log n)$ space. Also, note that although these data structures useful for implementing θ-tables have been known, the way we use these data structures to evaluate aggregate queries more efficiently is new.

insert builds a trivial entry from the element x to be inserted and appends it to the θ-table. The entry appended to the θ-table is

$$list[n : 1, x : x, f : \alpha(\emptyset)].$$

sort After inserting all elements, the method *sort* is applied. This method does two things. First, it sorts the elements in the θ-table according to the value of $x.a$. Then, it eliminates duplicate entries and remembers the number of occurrences in the additional attribute n of the entries of a θ-table. Thereby, it accumulates the f entries using β.

add For adding an element y to the $\leq$-table *add* performs the following. Using binary search, *add* searches for the element $[n : n, x : x, f : f]$ within the $\leq$-table with the smallest value $x.a$ such that $y.b \leq x.a$. This element is then replaced by

$$[n : n, x : x, f : \beta(f, \alpha(\mathit{bulky}))].$$

insadd For insadd(x), we append

$$[n : 1, x : x, f : \alpha(\mathit{bulk}(x))]$$

to the θ-table.

eval is defined as follows:

eval() {
    $\mathcal{N}'$ $a = \alpha(\emptyset)$;
    foreach (t **in** **self**)
        replace $t = [n : n, x : x, f : f]$ by $[n : n, x : x, f : a = \beta(a, f)]$;
    return **self**;
}

The local variable a accumulates all of the f values for each entry in the θ-table up to the current element. Then, each f value is set to this accumulated value.

Let us demonstrate the methods by giving their results when applied to the two bags X and Y of tuples with a single attribute a and b, respectively. X and Y are shown in Fig 1. The algebraic expression is

$$\chi_{a:count(\sigma_{y.a \leq x.a}(Y))}(X)$$

For the scalar aggregate *count*, we have α the regular scalar aggregate *count*, β is $+$ and γ is identity. Fig. 2 shows the results after *insert*, *sort*, *add*, and *eval* in a left to right order.

3.3 $\neq$-Tables

For a scalar aggregate f with decomposition α, β, and γ where $\beta : \mathcal{N}', \mathcal{N}' \rightarrow \mathcal{N}'$, a $\neq$-table has the following signature:

$$\mathit{list}([n : int, x : x, f : \mathcal{N}']).$$

Hence, it is the same as for $\leq$-tables. The only difference will be the value of f. Let $x.a \neq y.b$ be the predicate of (*). Then, f will collect the aggregate of all the y values with $y.b = x.a$. The computation of the correct aggregate value for a given x can then easily be computed by "subtracting" the value of f from the total aggregate of all $y \in Y$. The total aggregate will be computed during the *add* or *insadd* phase depending on $X \neq Y$ or $X = Y$. For holding the total aggregate, we assume the $\neq$-table to have an additional local variable a. Now, we are ready to specify the methods of the $\neq$-table:

create during the creation, the local variable a is initialized with $\alpha(\emptyset)$.

insert(x) appends the tuple $[n : 1, x : x, f : \alpha(\emptyset)]$ to the θ-table.

sort proceeds as before.

add(y) looks for an entry $[n : n, x : x, f : f]$ with $y.b = x.a$ and replaces it by $[n : n, x : x, f : \beta(f, \alpha(bulk(y)))]$. Further, it performs $a := \beta(a, \alpha(bulk(y)))$.

addins(x) appends the tuple $[n : 1, x : x, f : \alpha(bulk(x))]$ and assigns $a := \beta(a, \alpha(bulk(x)))$.

eval is defined as follows:

eval() {
  foreach (t **in self**)
    replace $t = [n : n, x : x, f : f]$ by $[n : n, x : x, f : \gamma(\beta^{-1}(a, f))]$;
  return **self**;
}

Eval does the following. For a given x, it "subtracts" the value of the aggregate of those tuples of Y fulfilling $y.b = x.a$ from the total aggregate of all values of Y. Thus, the resulting aggregate value is exactly the one for all $y.b \neq x.a$.

As an example let us compute

$$\chi_{a:avg(\sigma_{y.b \neq x.a}(Y))}(X)$$

for X and Y of Fig. 1. (For the definition of *avg* see page 2.) The expression computes for each $x \in X$, the average value of $y.b$ of all $y \in Y$ with $y.b \neq x.a$. The resulting states of the $\neq$-table after *insert* and *sort* are given in the upper row of Fig. 3, those after *add* and *eval* in the bottom row.

3.4 Remarks on θ-Tables

We would like to make some remarks.

First, note that θ-tables can be used to implicitly represent the result of any θ-join in linear space. This can be achieved by chosing appropriate α, β, and γ operators with a non-numeral in their signature, i.e., $\mathcal{N}'$ is some bulk type. More specifically, if α and γ are chosen to be identity and β is chosen to be $+$, θ-tables implicitly represent the result of a θ-join in linear space. Hence, they can also be used to optimize queries involving non-equi joins.

$a = [sum : 0, count : 0]$

[n:1, x:[a:10], f: [sum: 0, count: 0]]
[n:1, x:[a: 3], f: [sum: 0, count: 0]]
[n:1, x:[a: 7], f: [sum: 0, count: 0]]
[n:1, x:[a: 3], f: [sum: 0, count: 0]]

$a = [sum : 0, count : 0]$

[n:2, x:[a: 3], f: [sum: 0, count: 0]]
[n:1, x:[a: 7], f: [sum: 0, count: 0]]
[n:1, x:[a:10], f: [sum: 0, count: 0]]

$a = [sum : 9, count : 3]$

[n:2, x:[a: 3], f: [sum: 3, count: 1]]
[n:1, x:[a: 7], f: [sum: 0, count: 0]]
[n:1, x:[a:10], f: [sum: 0, count: 0]]

$a = [sum : 9, count : 3]$

[n:2, x:[a: 3], f: γ([sum: 6, count: 2])=3]
[n:1, x:[a: 7], f: γ([sum: 9, count: 3])=3]
[n:1, x:[a:10], f: γ([sum: 9, count: 3])=3]

Figure 3: States of a $\neq$-Table

Second, note that the methods *add* and *eval* give a possible implementation of the generalized aggregate function G-aggr [9]. Remember that G-aggr is a combination of (unary) grouping and the computation of a scalar aggregate.

Third, θ-tables can be used to efficiently implement the binary grouping operation introduced in [8] in order to enable the unnesting of queries which cannot be unnested otherwise.

Fourth, the technique introduced here can be seen as an extension of two techniques proposed in Section 5.3 of [4] and [3]. In the context of distributed relational databases, the techniques proposed there are the following: (1) Grouping followed by aggregation is distributed over union, if each group is contained in one fragment. (2) For queries containing a scalar aggregate but no grouping, the answer to the query is computed by distributing the aggregate over the partitions of the queried relation, if the scalar aggregate to be computed is decomposable.

4 Conclusion

We introduced the abstract datatype θ-table which allows to evaluate queries of class Θ involving aggregates in $O(n \log n + m \log n)$ or $O(n \log n)$ time and linear space, where the traditional approach needs $O(n*m)$ or $O(n^2)$ time and space. A syntactic characterization of the class Θ based on the result of translating the query into the algebra was given. This seems to be insufficient since the range of applications of θ-tables is not limited to this class. Hence, further research should investigate the total range of applicability of θ-tables and come up with a useful characterization of the corresponding queries. By useful we mean that an optimization can easily detect the applicability of θ-tables.

A first step in investigating the applicability is implied by the following observation. The queries capturable by the algebraic expressions treated in this paper are typically those requiring one level of nesting within some SQL-like query language. Future research will explore θ-tables for more levels of nesting.

Last not least, since θ-tables are implicit representations of θ-joins, their usefulness for efficiently evaluating successive non-equi joins seems worth to be investigated.

Acknowledgement: We thank the anonymous referees for their detailed comments. We also thank S. Ceri for pointing out to us Chapter 5 of [4].

References

[1] M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–588, 1975.

[2] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 72–88, 1990.

[3] S. Ceri and G. Pelagatti. Correctness of query execution strategies in distributed databases. *ACM Trans. on Database Systems*, 8(4):?–?, 1983.

[4] S. Ceri and G. Pelagatti. *Distributed Databases.* Computer Science Series. McGraw-Hill, 1984.

[5] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 354–366, Santiago, Chile, Sept. 1994.

[6] S. Chaudhuri and K. Shim. The promise of early aggregation. Technical report, HP Lab, 1994.

[7] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, 1993.

[8] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[9] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[10] R. Epstein. Techniques for processing of aggregates in relational database systems. ERL/UCB Memo M79/8, University of California, Berkeley, 1979.

[11] R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–33, 1987.

[12] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In *Proc. Dagstuhl Workshop on Query Optimization (J.-C. Freytag, D. Maier und G. Vossen (eds.))*. Morgan-Kaufman, 1993.

[13] W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.

[14] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. on Database Systems*, 7(3):443–469, Sep 82.

[15] A. Klug. Access paths in the "ABE" statistical query facility. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 161–173, Orlando, Fla., June 1982.

[16] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 91–102, 1992.

[17] R. Nakano. Translation with optimization from relational calculus to relational algebra having aggregate funktions. *ACM Trans. on Database Systems*, 15(4):518–557, 1990.

[18] W. Yan and P.-A. Larson. Performing group-by before join. Technical Report CS 93-46, Dept. of Computer Science, University of Waterloo, Canada, 1993.

[19] W. Yan and P.-A. Larson. Performing group-by before join. In *Proc. IEEE Conference on Data Engineering*, pages 89–100, Houston, TX, Feb. 1994.

[20] W. Yan and P.-A. Larson. Interchanging the order of grouping and join. Technical Report CS 95-09, Dept. of Computer Science, University of Waterloo, Canada, 1995.