Querying Multiple Features of Groups in Relational Databases

Damianos Chatziantoniou Kenneth A. Ross*
Department of Computer Science, Columbia University
damianos,kar@cs.columbia.edu

Abstract

Some aggregate and grouping queries are conceptually simple, but difficult to express in SQL. This difficulty causes both conceptual and implementation problems for the SQL-based database system. Complicated queries and views are hard to understand and maintain. Further, the code produced is sometimes unnecessarily inefficient, as we demonstrate experimentally using a commercial database system. In this paper, we examine a class of queries involving (potentially repeated) selection, grouping and aggregation over the same groups, and propose an extension of SQL syntax that allows the succinct representation of these queries. We propose a new relational algebra operation that represents several levels of aggregation over the same groups in an operand relation. We demonstrate that the extended relational operator can be evaluated using efficient algorithms. We describe a translation from the extended SQL language into our algebraic language. We have implemented a preprocessor that evaluates our extended language on top of a commercial database system. We demonstrate that on a variety of examples, our implementation improves performance over standard SQL representations of the same examples by orders of magnitude.

* This research was supported by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, by NSF grants IRI-9209029, CDA-90-24735, and by an NSF Young Investigator award.

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

1 Introduction

Aggregation is an important component of a query language because it is very commonly and intensively used in a wide range of applications. Examples of such applications include decision support systems, business databases, statistical databases and scientific databases. In order to support such applications, it is crucial for a database query language to be able to *succinctly express* common aggregate queries, and for the query processing system to be able to *efficiently evaluate* such queries.

Unfortunately, standard SQL (SQL92) does not allow the concise expression of a number of conceptually simple queries involving (potentially repeated) grouping and aggregation over the same groups. In order to express such queries in SQL, one has to perform additional joins, or to construct additional subqueries, while a kind of sequential processing of the grouped relation would be sufficient. The SQL versions of many such queries show a high degree of redundancy. There are two important consequences of such redundancy. First, it is very difficult to write, understand, and maintain such queries. Second, it is very difficult for a query optimizer to identify efficient algorithms for such queries, as we illustrate experimentally.

In this paper we define an extension to SQL syntax that allows the succinct representation of various aggregate queries. We also propose a new relational algebra operation Φ that concisely represents several levels of aggregation over the same groups in an operand relation. We show how Φ can be implemented using efficient algorithms. We then show how queries can be translated from our SQL extension into our extended relational algebra. Our queries are significantly shorter and simpler than their standard SQL counterparts, and are much more easily optimized. The syntax that we propose is significantly more general than previous proposals, and enables the concise solution of a number of well-known query representation problems. While Φ does not give additional expressive power to the relational algebra, it allows an important subset of queries to be succinctly expressed and efficiently evaluated.

We implemented a version of our extended SQL language on top of a commercial relational database system. We compare the performance of the extended system on examples in our exended language to that of the underlying commercial system on standard SQL representations of the same examples. Our experimental results demonstrate orders of magnitude improvements in performance. We argue that extending SQL in a fashion similar to ours is essential in order for the query optimizer to be able to identify and utilize efficient evaluation techniques.

The rest of the paper is organized as follows. Section 2 motivates the paper, presenting examples of queries, potential target implementation algorithms, and explaining why optimizing standard SQL versions of the example queries is hard. In Section 3 we describe our extension of SQL and we take the relational algebra with aggregation and extend it to enable the concise representation of our aggregate queries using a new operator Φ. Further, we show how our version of SQL can be translated into the extended relational algebra. Section 4 presents an algorithm to evaluate Φ and compares performance results using standard SQL of a commercial system and our implementation. Section 5 compares our work with related work, and we conclude in Section 6.

2 Motivation

In Section 2.1 we describe several motivating examples, and show why they are *conceptually* more complex than necessary. In Section 2.2 we describe an efficient target *implementation* for these queries. Section 2.3 explains why relational database systems might have difficulty identifying such implementations given the specifications in Section 2.1. In the examples we shall use the following relation, taken from [LM96].

```
CALLS(FromAC,FromTel,ToAC,ToTel,Date,Length)
```

The `CALLS` relation stores the calls placed on a telephone network over the period of one year. It includes the From number (area code and telephone number), the To number (area code and telphone number), the date and the length of the call. Each From number corresponds to a customer.

2.1 Motivating Examples

Example 2.1: Consider the following queries to the relation described above:

Q1. For each customer, find the longest call and the area code to which it was made.

Q2. For each customer, show the average-length of calls made to area codes 201 and 301 (in the same output record).

Q3. For each customer, show the number of calls made during the first 6 months that exceeded the average-length of all calls made during the year, and the number of calls made during the second 6 months that exceeded the same average length.

Q4. Suppose we are interested in those customers for whom the total length of their calls during summer (June to August) exceeds one-third of the total length of their calls during the entire year. For these customers, we would like to find the longest call made during the summer period, and the area code to which it was made. □

Each of these examples is a query in which all aggregation is grouped by the same grouping attributes. These are natural queries for a marketing application or for a decision support system. Hence we have an important practical class of queries. Additional examples may be found in textbooks [DD92], business-related articles [KS95], and benchmarks [Cou95].

All of the examples are cumbersome to express in SQL. In every case, they need to be expressed using joins or subqueries. Query Q1 might be expressed in standard SQL as

```
create view Q1View as
    select FromAC, FromTel, maxL=max(Length)
    from CALLS
    group by FromAC, FromTel

select V.FromAC, V.FromTel, ToAC, Length
from CALLS C, Q1View V
where  Length=maxL AND C.FromAC = V.FromAC
       AND C.FromTel=V.FromTel
```

Conceptually, the query is simple. However, the SQL solution is unnatural in that it requires the user to think in terms of *two* passes through the `CALLS` relation rather than one. Furthermore, the resulting implementation may be sub-optimal. (We shall address some of the resulting implementation difficulties in the next section.) Query Q2 is an example (a case of "Value-to-Attribute" conflict) of the well-known schematic discrepancies problem in database interoperability [KS91, KLK91, SCG93]. Data in one relation corresponds to metadata in another, in this case the output result. Again, SQL needs a join of the CALLS relation with itself to express this query. A possible expression of Query Q2 in standard SQL is

```
create view Q2View1 as
    select FromAC, FromTel, avgL=avg(Length)
    from CALLS
    where ToAC = "201"
    group by FromAC, FromTel

create view Q2View2 as
    select FromAC, FromTel, avgL=avg(Length)
    from CALLS
    where ToAC = "301"
    group by FromAC, FromTel

select  Q2View1.FromAC, Q2View1.FromTel,
        Q2View1.avgL, Q2View2.avgL
from Q2View1, Q2View2
where  Q2View1.FromAC=Q2View2.FromAC AND
       Q2View1.FromTel=Q2View2.FromTel
```

Query Q3 illustrates the use of two levels of aggregation using the same grouping attributes. A first aggregation (computing the average-length call) is used along with a condition on Date, to restrict tuples to be aggregated (count) at the second level. Queries with three or more levels of aggregation are also possible. In SQL, each aggregation has to be embodied in a separate subquery or view, even if the aggregates are over the same groups. Additionally, each such view needs to perform a join with the original relation. A standard SQL version of Query Q3 is shown below:

```
create view Q3View as
    select FromAC, FromTel, avgL=avg(Length)
    from CALLS
    group by FromAC, FromTel

create view Q3View1 as
    select C.FromAC,C.FromTel,cnt=count(*)
    from CALLS C, Q3View V
    where  C.FromAC = V.FromAC AND
           C.FromTel = V.FromTel AND
           Length > avgL AND Date<"96/07/01"
    group by CALLS.FromAC, CALLS.FromTel

create view Q3View2 as
    select C.FromAC,C.FromTel,cnt=count(*)
    from CALLS C, Q3View V
    where  C.FromAC = V.FromAC AND
           C.FromTel = V.FromTel AND
           Length > avgL AND Date>"96/06/30"
    group by CALLS.FromAC, CALLS.FromTel

select  Q3View1.FromAC, Q3View1.FromTel,
        Q3View1.cnt, Q3View2.cnt
from Q3View1, Q3View2
where  Q3View1.FromAC=Q3View2.FromAC AND
       Q3View1.FromTel=Q3View2.FromTel
```

Query Q4 requires many of the features described above. It aggregates over one subset of each group and compares the resulting value with an aggregate over the whole group. Then a selection of a particular *row* of that subset (the one with the maximum length) has to be made. An SQL formulation of Query Q4 is given below. Notice that it is important to define Q4View2 as a separate view and *not* incorporate it as a where clause within Q4View3. We need the area codes (ToAC) in the final result and we cannot select them in Q4View2 because standard SQL insists that selected attributes be grouping attributes.

```
create view Q4View1 as
    select FromAC, FromTel, sumL=sum(Length)
    from CALLS
    group by FromAC, FromTel

create view Q4View2 as
    select *
    from CALLS
    where Date>"96/05/31" AND Date<"96/09/01"

create view Q4View3 as
    select  FromAC, FromTel,
            sumL=sum(Length),maxL=max(Length)
    from Q4View2
    group by FromAC, FromTel

select  V1.FromAC, V1.FromTel,
        V2.ToAC, V2.Length
from Q4View1 V1, Q4View2 V2, Q4View3 V3
where V1.FromAC = V2.FromAC AND
      V1.FromAC = V3.FromAC AND
      V1.FromTel = V2.FromTel AND
      V1.FromTel = V3.FromTel AND
      V3.sumL * 3 > V1.sumL AND
      V2.Length = V3.maxL
```

It is not easy to derive (and maintain) such a query! Further, it is not easy for a query optimizer to implement this query efficiently, as we shall see shortly.

We have criticized standard SQL for its complexity in specifying queries involving multiple features of the same group. In this paper we present SQL extensions that reduce such complexity. While we shall delay the presentation of our SQL language to Section 3, we present below our version of Query Q3.

```
select FromAC,FromTel,count(X.*),count(Y.*)
from CALLS
group by FromAC, FromTel :  X, Y
suchthat  (X.Date < "96/07/01" AND
           X.Length > avg(Length)) AND
           (Y.Date > "96/06/30" AND
           Y.Length > avg(Length))
```

The X and Y variables of the group-by clause denote tuple variables that range over the tuples in each group. We refer to X and Y as *grouping variables.* (If there are multiple relations in the `from` clause, then there is a grouping variable tuple for every combination of underlying relation tuples that satisfy the `where` clause.) The newly introduced `suchthat` clause is used to define these grouping variables. The order in which they are declared in the group-by clause implies the order of evaluation. For example, the Y grouping variable could be defined in terms of aggregates over variable X. The `having` clause is still present, with a wider functionality as we shall see in Section 3.

2.2 Target Implementation

While conceptual complexity is an important issue, it is not the only reason for extending SQL with special features for grouping. There are also significant implementation issues. When performing aggregations and selections *repeatedly over the same groups*, there are specialized algorithms that can operate much more efficiently than standard joins and aggregations. The algorithms presented here are not particularly novel. However, they do illustrate the kinds of efficient implementation techniques that are available.

We assume that we aggregate over one relation, R. If R is the result of a join, we first perform the join to materialize R.

Algorithm 2.1: General algorithm for aggregates over the same groups.

Input: A relation R and a set V of grouping attributes.

Output: Aggregates (grouped by V) and selected records from R.

Method: Partition R according to the grouping attributes into buckets. Partitioning could be done by hashing or range-partitioning the grouping attributes. Read each bucket in turn, sort by the grouping attributes, and process the groups one by one. For each group make as many passes as necessary (aggregating or selecting records, using computed aggregates in subsequent passes) to answer the query on that group. ∎

Algorithm 2.1 is a general implementation that applies to any aggregate query all of whose aggregates use the same grouping attributes. In particular, all of queries Q1 though Q4 are of this form. If the main memory is big enough to hold each bucket[1] then the total amount of I/O is one pass of R for the partitioning, and one pass of the partitioned version of R to answer the query. Further, the total amount of CPU time required is linear in the number of aggregations and record selections performed in the query.

For special cases, more efficient algorithms may be possible. We give one example below.

Algorithm 2.2: Algorithm for `min` and `max` aggregates over the same groups.

Input: A relation R and a set V of grouping attributes.

Output: Single `min` and/or `max` aggregates (grouped by V) and (optionally) records from R with minimal or maximal attribute values.

Method: Make a single sequential scan of R. For each value v of the grouping attributes V keep the running minimum and maximum, and the record(s) from R with the minimum and/or maximum attribute value. ∎

Algorithm 2.2 applies to special kinds of queries in which a `min` or `max` is used, and the aggregated attribute is equated with the attribute itself. It could be used for Query Q1. If the main memory is large enough to hold one (or possibly several) records for each group, then the total I/O cost is one pass through R and a negligible CPU cost. When memory is limited, Algorithms 2.1 and 2.2 can be tuned for the amount of available memory; such issues are beyond the scope of this paper.

We want to avoid *naive* implementations of the queries using joins and subqueries. Unfortunately, as we shall see in Section 2.3, it is very difficult to find good implementations when the queries are expressed in standard SQL. What we need is a syntax to express these queries succinctly, without any redundancy. The query can then be optimized appropriately based on its special characteristics, or translated into an algebraic operator that can be implemented more efficiently than general joins.

2.3 Optimizing the Standard SQL Formulations is Hard

The most noticeable feature of the SQL solutions to the queries presented above is redundancy. Information is repeated in several places. Consider the SQL solution to Query Q4.

[1] We choose the partitioning algorithm to generate sufficiently small buckets if possible.

- The relation `CALLS` and the view `Q4View2` are mentioned more than once.
- Grouping by `FromAC`, `FromTel` is mentioned more than once.
- Extra conditions equating the `FromAC`, `FromTel` attributes in the various views are necessary.
- The keywords `select`, `from`, and `where` appear multiple times.

Why is removing redundancy so important? As we have already seen, redundancy leads to longer queries that are more difficult to understand and maintain. Further, redundancy makes the job of a query optimizer very hard. Consider a query optimizer trying to identify Algorithm 2.1 as a candidate strategy for the SQL specification of Query Q4. The optimizer has to identify aspects of the query that indicate multiple aggregations and selections over the same groups. In order to do that, it has to notice that

- Both `Q4View1` and `Q4View3` are grouped by the same attributes.
- View `Q4View3` includes the grouping attributes FromAC, FromTel.
- The top-level query equates grouping attributes, compares aggregate values with attributes and other attribute values, but *does not* compare non-grouping attributes of the views.

In addition, the optimizer has to be smart enough to ignore small changes that do not affect the applicability of the algorithm, but that might change the internal representation of the query.

In general, such observations are *global* properties of the query, not local properties. The search space for observing these properties is prohibitively large, and so optimizers traditionally focus on sequences of local optimizations (with some global optimization to determine join orders). Thus, it is unreasonable to expect an optimizer to take the SQL version of Query Q4 and find Algorithm 2.1.

The implementation of our extended SQL language, described in Section 4, shows dramatically improved performance because it is relatively easy (as we shall see) to identify good implementation techniques.

3 Extending SQL with grouping variables

In this section we present our extension of SQL, show how it can be used to express the motivating examples, and provide a translation from the extended SQL to the relational algebra including a new operator Φ.

3.1 Syntax

We propose the following SQL syntax extensions:

From clause. There is no change. Suppose that the relations in the from clause are $T_1, \ldots, T_p$.

Where clause. There is no change. Suppose that the conditions in the where clause are $D_1, \ldots, D_q$.

Group By clause. The group by clause is the same as in standard SQL, with the following addition: after specifying the grouping attributes it may contain *grouping variables*. These will range over the tuples within each group. For example, we may write

```
group by FromAC, FromTel :  R, S
```

In the following description we shall assume that there are n grouping variables named $r_1, r_2, ..., r_n$. Grouping variables range over tuples in the join of $T_1, \ldots, T_p$ according to $D_1, \ldots, D_q$.

Suchthat clause. This clause *defines* the range of the grouping variables mentioned in the group by clause. (The `suchthat` clause is like a `where` clause for grouping variables.) It has the following form:

$$C_1 \text{ and } C_2 \text{ and ... and } C_n$$

Any C_i may be omitted. Each C_i is a (potentially complex) condition involving the attributes of r_i, constants, aggregate functions of ordinary attributes and aggregate functions of attributes of $r_1, \ldots, r_{i-1}$.

Having clause. The having clause is a condition G of the form G_1 and ... and G_m. Each G_i can be:

- an expression where each subexpression involves *only* ordinary aggregates, aggregates of $r_1, ..., r_n$ and constants, such as "count(∗) > 10 or sum(R.Length) < sum(Length)", or
- an expression where each subexpression involves an attribute of r_j (for some fixed j) and some ordinary aggregate, an aggregate of $r_1, ..., r_n$ or a constant, such as "R.Length = max(Length) or R.Length > 20".

Select clause. The select clause is the same as in standard SQL, with the following addition: attributes of the grouping variables, and aggregates of attributes of the grouping variables can also appear in the select clause.

3.2 Semantics

With the extended SQL syntax, we are able to define (using the $C_1, \ldots, C_n$ conditions in the `suchthat` clause) n "areas" within a group, with each of $r_1, \ldots, r_n$ ranging over one area. We can perform aggregation on each such area. The G condition can select tuples from these areas, or impose a condition on the group as a whole. Observe that the constraints on $C_1, \ldots, C_n$ in the having clause prevent cyclic definitions of the grouping variables' areas. Each group variable is restricted only by comparison with constants and aggregates of previous grouping variables. In other words, the C conditions (in the suchthat clause) are *defining* conditions for the grouping variables, while the G conditions (in the having clause) are *selection* conditions. Furthermore, G is constrained to allow an efficient implementation as we will see later. Extended SQL queries for Example 2.1 are:

(Q1)
```
select    FromAC, FromTel,
          R.ToAC, R.Length
from CALLS
group by FromAC, FromTel :  R
suchthat R.Length=max(Length)
```

(Q2)
```
select    FromAC, FromTel,
          avg(R.Length), avg(S.Length)
from CALLS
group by FromAC, FromTel :  R, S
suchthat  R.ToAC = "201" AND
            S.ToAC = "301"
```

(Q3)
```
select    FromAC, FromTel,
          count(X.*), count(Y.*)
from CALLS
group by FromAC, FromTel :  X, Y
suchthat  (X.Date<"96/07/01" AND
            X.Length>avg(Length)) AND
            (Y.Date>"96/06/30" AND
            Y.Length>avg(Length))
```

(Q4)
```
select    FromAC, FromTel,
          R.ToAC, R.Length
from CALLS
group by FromAC, FromTel :  R
suchthat  R.Date > "96/05/31" AND
            R.Date < "96/09/01"
having    sum(R.Length)*3>sum(Length)
          AND R.Length=max(R.Length)
```

Note that non-aggregate attributes of the grouping variables can appear in the select clause, as in queries Q1 and Q4. These attributes do not necessarily have the same value for all rows in the group. That means that a join between the areas designated by the grouping variables may be necessary. This will become apparent in the following section, where we define our new aggregation operator in terms of relational algebra.

If there is ambiguity about which relation an attribute of a group variable came from, such as when there are several relations in the `from` clause, it is explicitly specified as in `R.CALLS.ToAC`. An interesting aspect of our solution is the possible use of grouping *without* aggregation (i.e., if we are interested in features of groups that do not require aggregation). Notice the conceptual simplicity of the specifications of these examples, and the absence of redundancy in the specifications. In particular, compare these examples with the SQL versions of section 2.1.

3.3 Relational Algebra

In this section we present the standard relational algebra with aggregation, and extend it with a new operator Φ. Φ expresses multiple levels of selection and aggregation over the same groups. We show some examples of the use of Φ. We argue that, even though Φ does not yield additional expressive power compared with the standard relational algebra with aggregation, it does allow an important class of queries to be expressed more succinctly.

We assume that the reader is familiar with the standard relational algebra operations selection (σ), projection (π), join ($\bowtie$), union ($\cup$), and difference ($-$). An additional renaming operator ρ is defined as follows: $\rho_i(R)$ simply prepends the label "i." to every attribute name of R. There are several proposals for a grouping operator in the literature [EN89, Mum91]. We use a syntax similar to the syntax in [EN89]: an aggregate function operation $\mathcal{F}$ is defined as

$$_{\text{<grouping\_attributes>}}\mathcal{F}[\text{<function\_list>}](R)$$

where <grouping_attributes> is the list of grouping attributes of the relation R, and <function_list> is a list of the aggregate functions (min, max, count, average, sum), accompanied by an appropriate attribute of the relation specified in R. (These are the standard SQL aggregate operators; the techniques described below would also be applicable to other aggregate operators.) The name of the aggregated attribute is obtained by prepending the aggregate function name before the attribute name, for example "avg_Length." So, for example

$$_{\text{FromAC,FromTel}}\mathcal{F}[\text{sum Length, max Length}](\text{CALLS})$$

computes a relation with four attributes, namely FromAC, FromTel, sum_Length, and max_Length. For each (FromAC,FromTel), there is a tuple in the result with the corresponding total and maximum length. (We may override the naming convention by placing an attribute name in the function list, as in "total=sum Length".) An *instance* of $\mathcal{F}$ is given by the values of the grouping attributes and the function list.

In what follows, the grouping arguments to functions $\mathcal{F}$ will be understood from the context; we may choose to omit the grouping attributes in $\mathcal{F}$ for notational convenience. When no aggregate functions are listed, $\mathcal{F}$ returns the grouping attributes only: this aggregate operator is denoted by "_". The selection function that selects *all* tuples is also denoted by "_".

Example 3.1: Consider how Query Q4 would be expressed in relational algebra. First we have to define $Q4View2$, which consists of calls that were made during the summer period.

$$Q4View2 = \sigma_{\text{Date}>96/05/31 \text{ and Date}<96/09/01} \text{ (CALLS)}$$

The final result is then given by:

$$\pi_S\sigma_\Theta(\ {}_G\mathcal{F}[\text{total=sum Length}](CALLS) \bowtie {}_G\mathcal{F}[\text{sum Length, max Length}](Q4View2) \bowtie Q4View2)$$

where all joins are natural joins on the FromAC and FromTel attributes, and

$$\begin{array}{lcl} S & = & \{\text{FromAC, FromTel, ToAC, Length}\} \\ \Theta & = & \text{"sum\_Length*3} > \text{total and} \\ & & \text{Length} = \text{max\_Length"} \\ G & = & \text{FromAC, FromTel} \end{array}$$ □

Note that relational algebra suffers from the same problems observed for SQL: there are many joins and repeated subexpressions.

We now define a new algebraic relational operator called Φ that is more general than $\mathcal{F}$. The purpose of such an operator is twofold. First, it allows succinct representation of complex queries. Second, it can be implemented using specialized algorithms.

Like $\mathcal{F}$, Φ has some grouping attributes. However in addition Φ has a vector of selection conditions and a vector of aggregate functions that are applied to Φ's argument.

As can be observed in Example 3.1, there is a necessary *order of evaluation* of the aggregate functions and the selection conditions. Certain aggregates are applied to tuples satisfying a previous set of selections, and tuples can be selected based on previous aggregate values. Our Φ operator expresses multiple sequential selections and aggregations in a single operator. Within each group we initially aggregate over the whole group; later selection conditions may restrict tuples to a subset of the group. Φ returns a relation combining all of the computed aggregates and all of the copies of the argument relation that have been selected in various ways.

Definition 3.1: Let R be a relation, and let V be a set of grouping attributes from R. Let $n \geq 0$, and let $F_0, \ldots, F_n$ be instances of $\mathcal{F}$ that have grouping attributes V, and that are well-defined on R. Let $\sigma_1, \ldots, \sigma_n$ be selection conditions. Each σ_i can mention attributes in R and attributes in the result of any of $F_0, \ldots, F_{i-1}$. Let $\vec{F}$ denote the tuple $(F_0, \ldots, F_n)$. and let $\vec{\sigma}$ denote the tuple of selection conditions $(\sigma_1, \ldots, \sigma_n)$. We now inductively define two sets $R_1, \ldots, R_n$ and $A_0, \ldots, A_n$. Given $A_0, \cdots, A_{i-1}$, with $1 \leq i \leq n$, we define

$$R_i = \pi_R\sigma_i(R \bowtie A_0 \bowtie \cdots \bowtie A_{i-1})$$

where each join above is an equijoin on the grouping attributes, and π_R denotes a projection onto the attributes of R. We define $A_0 = F_0(R)$ and for $1 \leq i \leq n$, we define $A_i = \rho_i F_i(R_i)$. Finally, we define the operator $\Phi^n[V, \vec{F}, \vec{\sigma}]$ via

$$\Phi^n[V, \vec{F}, \vec{\sigma}](R) = A_0 \bowtie (\bowtie_{i=1}^n (\rho_i R_i \bowtie A_i))$$

where each join equates the grouping attributes of the corresponding arguments. □

The result of Φ will have attributes of R prepended with label 1 (from R_1), attributes of R with label 2 (from R_2) and so on. Similarly, the result of Φ will have as attributes aggregates over R (from A_0), aggregates over R_1 prepended with label 1 (from A_1), and so on. Intuitively, we can perform any number of aggregates, to any fixed nesting level, as long as the grouping attributes remain the same throughout. Note that there is an implicit order in the way that the aggregates are nested. Let us consider some examples of the use of Φ.

Example 3.2: Query Q1 could be expressed as:

$$\pi_S\Phi^1[\{\text{FromAC,FromTel}\}, (\ \mathcal{F}[\text{max Length}], \_), (1.Length = max\_Length)](\text{CALLS})$$

where S = "FromAC, FromTel, 1.ToAC, 1.Length". Query Q4 could be expressed as:

$$\pi_S\sigma_\Theta\Phi^1[\{\text{FromAC,FromTel}\}, (\ \mathcal{F}[\text{sum Length}],\ \mathcal{F}[\text{sum Length, max Length}]), (\text{Date}>96/05/31 \text{ and Date}<96/09/01)](\text{CALLS})$$

where Θ = "1.sum_Length$*3 >$ sum_Length and 1.Length = 1.max_Length" and S is as above. For Query Q4 the Φ subexpression is computed using three relations, A_0, R_1 and A_1. A_0 contains the grouping attributes FromAC, FromTel and the total Length; R_1 contains the call records having Date within the summer period; A_1 contains the total and maximum Length for tuples in R_1, for each customer □

3.4 Translating Extended SQL to Relational Algebra with Φ

We now show how to systematically translate our extended SQL into relational algebra. This is important for two reasons. Firstly, we will later show how to implement Φ, and so the translation will give a way to implement our extended SQL. Secondly, many query optimizers work on algebraic representations of queries, and try to optimize the order of operations to minimize the cost while still computing an algebraically equivalent query. By compiling into an algebraic language, we facilitate query optimization.

Let V denote the grouping attributes mentioned in the group by clause. Let R denote the join of all relations in the from clause, according to the conditions in the where clause. Suppose $r_1, \ldots, r_n$ are the grouping variables, and that $C_1, \ldots, C_n$ and G are the conditions as given in Section 3.1. In each C_i (but not in G) we drop the grouping variable prefix on all nonaggregated attributes mentioned in conditions. We convert all remaining grouping variables r_i into numbers i, so that r_2.`ToAC` becomes 2.`ToAC`, for example. Aggregates are also modified, so that `sum(1.Length)` is converted to sum_1_Length, for example. The attributes and aggregates now appearing in the select statement are denoted by S. Then a query in our extended SQL syntax is translated into

$$\pi_S(\sigma_G(\Phi^n[V, \vec{F}, \vec{\sigma}]))(R)$$

where $\vec{\sigma} = (C_1, \cdots, C_n)$, $\vec{F} = (F_0, \cdots, F_n)$, and F_i has aggregate functions that return aggregates that are used in S, G, or some C_j with $j > i$. The constraints on the C_i's ensure that the domains of the selections and aggregations match the requirements of Definition 3.1.

Example 3.3: Let S be the attribute list "FromAC, FromTel, avg_1_Length, avg_2_Length". Then Query Q2 from Section 3.2 would be translated to:

$$\begin{array}{l}\pi_S \Phi^2[\{\text{FromAC,FromTel}\}, \\ \quad (\_,\ \mathcal{F}[\text{avg Length}],\ \mathcal{F}[\text{avg Length}]), \\ \quad (\text{ToAC} = \text{"201"}, \text{ToAC} = \text{"301"})](\text{CALLS}) \qquad \Box\end{array}$$

3.5 Null Values and Duplicate Values

Consider Query Q2 executed on a database in which some customer has not made any calls to area code "201". Both our algebra and our SQL extension would give no tuple for that customer in the result even if s/he has made calls to area code "301" (and therefore an average for that area code exists). A better answer to this query might be the specification of an answer tuple that gave the average length to area code "301", together with a `NULL` value for the average length to area code "201".

Our algebraic operator Φ can be modified to give this behavior by changing some of the joins to outer joins. Similarly, our SQL language could be extended to put a keyword `mandatory` before each grouping variable that is required to take a `NULL` value if no matching tuples are found. The details are omitted due to lack of space.

A related problem concerns duplicate values. If we take a multiset semantics for the algebraic operations, as would normally be the case for the target of an SQL-translation, we may find that our answers contain many more tuples than we expect. For example, Query Q3 from Example 3.3 would return k copies of the (FromAC, FromTel, count_1_*, count_2_*) answer, where k is the product of the number of calls having over-the-average length in the first 6 months with the number of calls having over-the-average length in the second 6 months. The problem here is that even though the attributes of CALLS_1 and CALLS_2 in Φ are projected out, these two relations still participate in the join and hence contribute to the multiplicity of the result.

Our algebraic operator Φ can be modified to omit a set of specified grouping variable relations from the join in order to prevent this behavior. In order to decide which of the grouping-variable relations should appear in the join of the definition of Φ and which should not, we have to understand the nature of our class of queries. We usually want to define an "area" of the group in order to get certain aggregate values of this area. Then, these aggregates are used to select tuples from this or another area, or to define a new area. In most of the cases, we are only interested in the aggregates and not in the area being aggregated. In these cases, the grouping-variable relation should not participate in the join, unless we explicitly specify it. The term $(\rho_i R_i \bowtie A_i)$ in Definition 3.1 is replaced by A_i. Notice that with that replacement, we avoid the duplicate values problem, because A_i contains one tuple per group. For example, in query Q3 we want to output *only* the aggregates of the two areas defined in that query. As a result, CALLS_1 and CALLS_2 should not participate in the join. Φ^2 is given by the join of A_0, A_1 and A_2. CALLS_1 and CALLS_2 are defined as before, simply do not participate to the join.

In order to determine which $(\rho_i R_i \bowtie A_i)$ terms should be replaced by their aggregates A_i, we use the `select` and the `having` clauses from the original SQL specification. An R_i must have a *non-omit* status (i.e., its relation must participate in the join) if some attribute of R_i is either in the `select` clause or the `having` clause, where the multiplicity would be significant. Otherwise, all grouping variables have an *omit* status, which means that their relations do not participate in the join. The rationale for such a choice is the following. If we do not output any attribute of a grouping variable R_i, or use it in the having clause, then its multiplicity is probably not significant and we omit it from the join. We can allow the user to force a grouping variable into the join, by writing the keyword `use` before the grouping variable in the `group by` clause.

4 Implementation

In this section we present an algorithm and an implementation of the Φ operator. We compare performance results on the queries of Example 2.1, using standard SQL and our implementation.

4.1 An Algorithm for Evaluating Φ

We have a translation from our extended SQL into our extended algebra (with Φ). We now show how to identify efficient candidate algorithms for Φ, and consequently we will be able to use those algorithms to implement queries in our extended SQL language. We will not actually implement Φ directly since it may be a lot larger than we need. For example, in Query Q4 (Example 3.2) the result of Φ would contain *all* the summer-period calls of the customers specified in that query. However, when combined with a selection and projection, the result is much smaller. Thus we shall identify implementations for operations of the form $\pi_S \sigma_\Theta \Phi(R)$. We show how to use Algorithm 2.1 to implement $\pi_S(\sigma_G(\Phi^n[V, \vec{F}, \vec{\sigma}]))(R)$ where G is of the form "G_1 and $\cdots$ and G_m" described in Section 3.1, $\vec{\sigma} = (\sigma_1, \cdots, \sigma_n)$, and $\vec{F} = (F_0, \cdots, F_n)$. If R is a complex expression, then we first materialize R.

Step 1 Partition R according to the grouping attributes into buckets. Read in a bucket and process each group. If a group has size g, then create n bitmaps $b_1, \ldots, b_n$ of length g. The bitmaps encode the underlying R_i relations in the definition of Φ^n (on the current group). We process each group according to the following steps.

Step 2 Make $(n+1)$ passes over the group. On the first pass ($i = 0$), the aggregates F_0 of the full group are computed. On subsequent passes $1 \leq i \leq n$, the selections σ_i on r_i and the aggregates F_i of r_i are simultaneously computed. The selection result is stored in b_i, with a bit set if the corresponding tuple (together with previously computed aggregate values) satisfies σ_i. If any selection result is empty, we can immediately move to the next group.

Step 3 Because of the form of G, each G_j can be checked either (a) on the aggregate tables alone, or (b) on a single R_i relation together with the aggregate tables. For a group, there is a single tuple in each aggregate table. Hence selection conditions on the aggregate tables either include or exclude the group as a whole; if such a condition is violated, we simply move to the next group. Conditions that mention an attribute from R_i are processed by making an additional pass, further restricting the bitmap b_i. Again, if the resulting bitmap is all zeroes we immediately move to the next group. We make at most m additional passes over the group.

Step 4 We now compose the join (according to the grouping attributes) of all of the R_i relations as indicated by their bitmaps, together with the aggregate values for the R_is. We project onto the attributes in S as we compute the join.

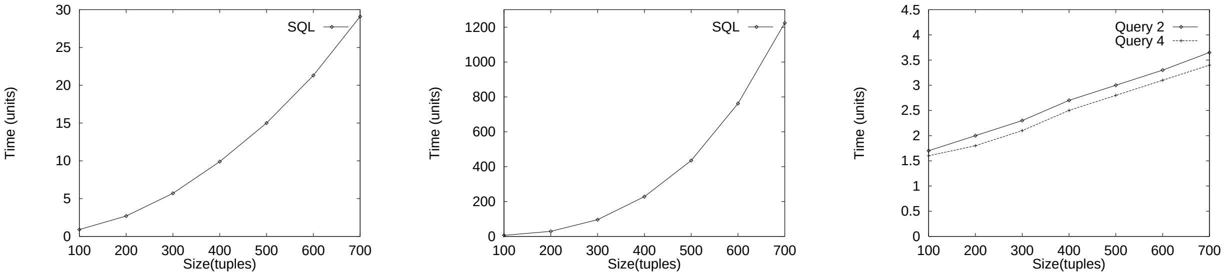
In Step 3 above, we can process each condition in G in parallel. We make a single pass through the current group in the actual relation R, and process all of the corresponding bitmaps simultaneously. An important aspect of Step 4 of the algorithm is the computation of the join even for group variables that are not mentioned in S. As discussed in Section 3.5, the join is necessary if we are using a multiset semantics for the underlying relational operations. If relation R_i is to be omitted from the result, as outlined in Section 3.5, then the join with grouping variable i can be omitted. In practice, the omission of some of the R_is will happen relatively often, limiting significantly the joins of step 4. Furthermore, the R_i's that participate in the final result often have just a few tuples, making the cost of step 4 very small.

While $n+1$ passes is specified above, some queries employing Φ^n can be answered in fewer passes. An example is Query Q2. *One* pass over each group would be sufficient to define both R_1 and R_2 simultaneously, along with their aggregates. Some analysis of the arguments of Φ can be performed in order to determine which R_i's can be evaluated on the current pass, and which need to wait for an aggregate value to be computed.

For special cases, more efficient algorithms can be identified. Algorithm 2.2 could be identified for Query Q1 (Example 3.2) on the basis of the form of the aggregation (a maximum) and the subsequent selection (an equality with the previously calculated maximum).

4.2 Experimental Platform and Results

We used a state-of-the-art commercial relational database system, running on a Sparcserver 630MP with SunOS5.4. The time measurements are elapsed (wall-clock) times. We measure the performance in "units," without giving the conversion to seconds, deliberately. We are not aiming to show how well (or badly) our commercial database system performs in absolute terms. Rather, we aim to show the speedups



(a) Standard SQL for Q2 (b) Standard SQL for Q4 (c) Extended SQL for Q2 and Q4

Figure 1: Elapsed time versus size for different query syntax (for queries Q2 and Q4)

that are potentially possible with an extended syntax and suitable evaluation techniques. Each measurement is the average of many runs, usually more than ten and repeated at different times of the day. During the experiments, we were the only users of the database system. The Sparcserver was very lightly loaded.

In all experiments we do not write the query result back to disk.

For the standard SQL measurements we used the standard SQL representations of Queries Q1 to Q4 given in Section 2.

Our extended language is implemented as follows. A compilation phase reads in queries expressed in a version of our SQL extended language and generates a C program that interacts with the commercial database system. The C program follows the steps of Algorithm 2.1 given in Section 4.1, except for Step 1. Instead of Step 1, the C program poses a subquery to the database system in which the `group by` attributes are also mentioned in an `orderby` clause. The results of this subquery are returned in a buffered fashion to the C program. The `orderby` clause ensures that the groups can be further processed one at a time.

The behavior of our implementation with respect to null and duplicate values (as was described in Section 3.5) is as follows. All grouping variables have a *non-mandatory* status; all grouping variables mentioned in the select clause have a *non-omit* status; the remaining grouping variables have an *omit* status.

For the extended SQL measurements we measured the total elapsed time for execution. The queries used were the extended representations of Queries Q1 to Q4 given in Section 3.2.

The performance of queries Q2 and Q4 is shown in Figure 1. We have similar graphs for queries Q1 and Q3. The first two graphs correspond to standard SQL and the third to our extended SQL. In all cases, the size of each group is twenty tuples and there are no indices. The main cost is CPU time. In all cases, standard SQL performance is non-linear and much worse than our implementation. The optimizer was not able to identify that joins were unnecessary and that a kind of sequential processing would be sufficient. Even if we had indices or other join optimization techniques to speed up the process, we would still need to perform joins, a major cost. Our implementation avoids joins altogether.

We have the same results on relations of bigger size, where the cost of standard SQL for queries like Query Q4 is in the range of hours.

5 Related Work

Aggregation queries are essential in decision support systems. This importance has been recognized quite recently and led to a number of papers on optimization of such queries. However, none of these addresses the optimization of our particular class of queries. These authors mainly examine which of a group by and a join should be executed first. As we have seen, joins in our class of queries could be avoided completely.

Yan and Larson in [YL94, YL95] describe a class of transformations that allow the query optimizer to push a group-by past a join (eager aggregation) or pulls a group-by above a join (lazy aggregation). In a similar direction, Chaudhuri and Shim in [CS94, CS96] present a similar class of pull-up and push-down transformations. Furthermore, they incorporate these transformations in optimizers and propose a cost-based optimization algorithm to pick a plan. In [GHQ95], Gupta, Harinarayan and Quass try to unify these transformations, viewing aggregation as an extension of duplicate-eliminating projection.

A different type of optimization is found in [LMS94, LM96]. While the above mentioned authors give criteria on when to apply the pull-up and push-down tranformations(given a set of predicates), Levy and Mumick consider an orthogonal problem. They present a method to infer predicates either for the attributes of the view from the predicates on the attributes of the relations defining the view, or the other way around.

Gray, Bosworth, Layman and Pirahesh propose a relational aggregation operator, called datacube, which is useful in data analysis applications [GB$^+$96]. Each of the aggregation attributes is a dimension in a n-dimensional space. They propose an extension of SQL, using an argument similar to ours, i.e., the accomodation of this class of queries is important. The main contribution is a conceptualization of the the aggregation accross many dimensions. There is an overlapping of the class of queries that this operator expresses with our class. However, our techniques allow for significantly more complex aggregate queries.

Several authors have pointed out that SQL cannot simply express a number of queries involving grouping and aggregation [DD92, KS95]. Kimball and Strehlo in [KS95] argued that SQL should be extended in order to be more powerful (in both syntax and performance) for queries related to grouping. They present some examples where standard SQL syntax is cumbersome and performance is bad, while the queries are conceptually easy. They also propose a new keyword, called ALTERNATE, which is associated with a constraint. This constraint replaces all constraints on the same table in the surrounding query. While the change in SQL syntax is minimal, the expressivity of that keyword is limited. For example, they can have only one level of aggregation.

Sybase allows some flexibility in the syntax of the `select` and `having` statements [Cor94]. In particular, `select` and `having` statements can include *any* attribute, not just those mentioned in the group by clause. Sybase's extended SQL can be used to express some group queries that would have to be phrased as a join in SQL92. In particular, Query Q1, can be expressed in a syntax similar to ours. The main idea is similar to ours: enabling succinct, optimizable aggregate queries. To a certain extent they were successful. However, their techniques are significantly less general than ours; their syntax corresponds roughly with allowing just one grouping variable. Consequently, their extended syntax cannot succinctly express Queries Q2, Q3, or Q4.

Rao, Badia and Van Gucht address the issue of supporting quantified subqueries [RBV96]. Their work, which was done independently from ours, is motivated by similar concerns that SQL's syntax is cumbersome for expressing and optimizing a natural class of queries.

Our syntax can succinctly solve (for a known number of columns) the so-called *Value-to-Attribute* conflict, a case of schematic discrepancies in interoperable databases. This conflict occurs when the same information is expressed as values in one table and as attributes in another [KS91, KLK91, SCG93, Roz94]. In [SCG93] the complexity of the SQL solution to this problem is criticized and a new operator is proposed. Rozenshtein emphasizes that the Value-to-Attribute conflict (which he calls Table-Pivoting) can be solved in just one pass over the grouped relation, and proposes a solution using the notion of *characteristic functions* [Roz94]. Again, our syntax expresses a significantly more general class of queries.

6 Conclusions

We have identified redundancy in both relational algebra and SQL in the way these languages specify a number of conceptually simple aggregate queries. This redundancy leads to queries that are difficult to write, maintain, understand, and optimize. We have extended both relational algebra and SQL to enable the succinct representation of these queries. We provided a translation of our SQL language into our algebra, and demonstrated that efficient algorithms for queries expressed in our algebra exist and are easy for an optimizer to identify. We experimentally verified that orders of magnitude savings in processing time can be achieved by using our extended syntax.

The impact of this work is twofold. First, using our syntax allows conceptually simple aggregate queries to be expressed simply, reducing the effort required for writing and maintaining such queries. Second, it allows such queries to be optimized and executed efficiently; naive plans resulting from an inability to recognize the special form of the query may be orders of magnitude slower.

References

[Cor94] Sybase Corporation. *Sybase SQL Server, Reference manual, Vol. 1.* Sybase, Inc, 1994.

[Cou95] Transaction Processing Performance Council. TPC-D benchmark description. (available from `http://www.tpc.org`), April 1995.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB Conference*, pages 354–366, 1994.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimizing queries with aggregate views. In *Extending Database Technology*, pages 167–182, 1996.

[DD92] C. J. Date and H. Darwen. *Relational Database Writings 1989-1991.* Addison-Wesley, 1992.

[EN89] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, 1989.

[GB$^+$96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Datacube : A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *IEEE International Conference on Data Engineering*, pages 152–159, 1996.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *VLDB Conference*, pages 358–369, 1995.

[KLK91] Ravi Krishnamurthy, Witold Litwin, and William Kent. Language features for interoperability of databases with schematic discrepancies. In *ACM SIGMOD, Conference on Management of Data*, pages 40–49, 1991.

[KS91] Won Kim and Jungyum Seo. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, 24(12):12–18, 1991.

[KS95] Ralph Kimball and Kevin Strehlo. Why decision support fails and how to fix it. *SIGMOD RECORD*, 24(3):92–97, 1995.

[LM96] Alon Levy and Inderpal Singh Mumick. Reasoning with aggregation constraints. In *Extending Database Technology*, pages 514–534, 1996.

[LMS94] Alon Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate movearound. In *VLDB Conference*, pages 96–107, 1994.

[Mum91] I. S. Mumick. *Query Optimization in Deductive and Relational Databases.* PhD thesis, Department of Computer Science, Stanford University, 1991.

[RBV96] Sudhir Rao, Antonio Badia, and Dirk Van Gucht. Providing better support for a class of decision support queries. In *ACM SIGMOD, Conference on Management of Data*, pages 217–227, 1996.

[Roz94] David Rozenshtein. Linguistic optimization: A new approach to writing efficient SQL queries. In *Ventures in Research, Long Island University*, 1994.

[SCG93] F. Saltor, M.G. Castelanos, and M. Garcia-Solaco. Overcoming schematic discrepancies in interoperable databases. In *Interoperable Database Systems*, pages 191–205. Elsevier Science Pub N.V. (North-Holland), 1993.

[YL94] Weipeng P. Yan and Per-Ake Larson. Performing Group-By before Join. In *IEEE International Conference on Data Engineering*, pages 89–100, 1994.

[YL95] Weipeng P. Yan and Per-Ake Larson. Eager aggregation and lazy aggregation. In *VLDB Conference*, pages 345–357, 1995.