Extending Complex Ad-Hoc OLAP

Theodore Johnson
Database Research Dept.
AT&T Labs - Research
johnsont@research.att.com

Damianos Chatziantoniou
Dept. of CS
Stevens Institute of Technology
damianos@cs.stevens-tech.edu

Abstract

Large scale data analysis and mining activities require sophisticated information extraction queries. Many queries require complex aggregation, and many of these aggregates are non-distributive. Conventional solutions to this problem involve defining User Defined Aggregate Functions (UDAFs). However, the use of UDAFs entails several problems. Defining a new UDAF can be a significant burden for the user, and optimizing queries involving UDAFs is difficult because of the "black box" nature of the UDAF.

In this paper, we present a method for expressing *nested* aggregates in a declarative way. A nested aggregate, which is a rollup of another aggregated value, expresses a wide range of useful non-distributive aggregation. For example, *most frequent* type aggregation can be naturally expressed using nested aggregation, e.g. "For each product, report its total sales during the month with the largest total sales of the product". By expressing complex aggregates declaratively, we relieve the user of the burden of defining UDAFs, and allow the evaluation of the complex aggregates to be optimized.

We use the *Extended Multi-Feature* (EMF) syntax as the basis for expressing nested aggregation. An advantage of this approach is that EMF SQL can already express a wide range of complex aggregation in a succinct way, and EMF SQL is easily optimized into efficient query plans. We show that nested aggregation queries can be evaluated efficiently by using a small extension to the EMF SQL query evaluation algorithm. A side effect of this extension is to extend EMF SQL to permit complex aggregation of data from multiple sources.

1 Introduction

The growing use of data warehouses has pointed out the need for query languages and tools that are more sophisticated than standard SQL on relational databases. Recent research has developed some approaches for these languages and tools. Gray, Bosworth, Layman, and Pirahesh [14] have proposed the Cube operator. Sarawagi, Thomas, and Agrawal [25] show how frequent sets can be computed by generating SQL queries. Gingras and Lakshmanan [12] propose an algebra which allows complex data restructuring and aggregation at multiple granularities. Microsoft Corporation is developing MDX (MultiDimensional expressions), a language for making multiple OLAP queries [10].

On-line analytical processing has attracted a lot of attention recently because large enterprises want to analyze the warehouses of their collected data. Although there exists literature on modeling and conceptualizing OLAP [15, 2, 22], research has been mainly focused on expression, evaluation, maintenance, and usage of datacubes [14, 1]. The processing and optimization of complex *ad-hoc* OLAP queries has been given little attention [31, 8], although certain query processing techniques may be applicable [9, 26]. Better evaluation of these queries is the motivation of several SQL syntactic extensions proposed in the past [20, 24, 7].

A number of commercial vendors, such as COGNOS, Business Objects, Hyperion (Essbase), and Oracle Express, provide multidimensional data analysis and OLAP tools. Most of these systems have their own query language, not based on SQL. Some require data to be stored in their proprietary storage format (e.g. ESSBASE), while others either utilize standard database systems (e.g. Business Objects) or first extract the data to flat file format and then process them. While these products provide a suite of multidimensional data analysis tools (rollup, drill down, slicing, dicing, etc.), they do not provide facilities for the complex ad-hoc OLAP queries that are the subject of this paper.

Conventional aggregate functions (e.g., min, max, count, sum, etc.) have the property of being *distributive* – that is, it is easy to combine subtotals into grand totals. While this property of the aggregate functions enables their efficient computation [13, 27], often the user desires a more complex aggregate. Examples include percent-of-total, moving-average, most-frequent, and median [14, 17]. However, these aggregates may be difficult to compute. For example, most-frequent and median are holistic (summing parts into a grand total requires unbounded temporary storage) [14].

The conventional solution to providing new complex aggregates for a query is to define a User Defined Aggregate Function (UDAF) in an object-relational DBMS [17, 3, 16]. The user supplies modules that initialize an aggregate, add a tuple to it, and compute a final value. For optimization [23, 29], the user registers information about the aggregate that can be used by the optimizer. To allow parallelism, the user must supply additional modules and registration information [17, 23].

Our recent work on the Extended Multi-Featured syntax

CIKM '99 11/99 Kansas City, MO, USA

(EMF SQL) is motivated by the desire to express a wide range of complex aggregates in a succinct and declarative manner. We had found that users of data warehouses (at AT&T, and at a medical school [18]) were frequently unable to express their complex aggregation queries in SQL. Answering these queries required the help of a database administrator to write the complex, slow (and frequently buggy) SQL or procedural code. As a result, these queries are usually answered late, or not at all.

The EMF syntax allows a very precise control over the range of tuples used to compute an aggregate. As a result, complex aggregates including percent-of-total, moving-average, or median can be easily expressed, modified, and generalized without resorting to UDAFs. We present some examples in Section 2 (examples 2.1 and 2.2 appear in [18]). EMF SQL has an efficient evaluation algorithm, and many automatic optimizations, including parallelization, are possible.

The major focus of this paper is to introduce an extension to the EMF SQL syntax that allows the expression of complex non-distributive aggregates in a declarative manner using standard aggregates built into SQL (plus one additional aggregate). The idea is to use *nested aggregation*, i.e. aggregation over aggregated quantities. This extension allows the expression of a subset of holistic aggregates, e.g. those that can be expressed as rollups of other aggregates. One example of this type of aggregate is the most-frequent aggregate, which is the max of a count. Nested aggregation allows the user to express generalizations of this type of query, for example "For each product, what is the maximum of the average monthly sales, during 1997". A wide range of other queries can be expressed as well, and we provide some examples.

Evaluating nested aggregates requires access to multiple data sources. We expand on this idea to allow EMF queries access to multiple fact tables. The modifications to the EMF syntax and evaluation algorithm are minor, but they permit the expression of aggregation queries that range over multiple data sources. We give several examples of this type of query.

We show that the evaluation of nested EMF queries can be performed by a small extension to the evaluation algorithm for EMF queries over multiple fact tables. Because the efficient EMF query evaluation algorithm can be applied with only a few modifications, the query optimizations that we developed in previous work (including scalable evaluation and parallelization) can be applied. We also present some optimizations that can be made for the new extensions.

In Section 2, we present the EMF syntax and evaluation algorithm for background. In Section 3, we show how to extend the EMF syntax to express queries over multiple fact tables. In Section 4, we discuss nested aggregation. In Section 5, we discuss query plan optimizations. Finally, in Section 7 we conclude.

2 Extended Multi-Feature Syntax

In previous works [4, 6] we have developed the Extended Multi-Feature syntax (EMF SQL). Because the material in this paper is phrased in terms of EMF SQL, we review it in this section.

We have found that ad-hoc decision support queries present two key features that can be exploited either in developing query optimization techniques, or in equipping a language with appropriate syntactic extensions:

- First, OLAP queries group the relation(s) on a set of attributes and perform some complex (or simple) operation *within* each group. Although SQL handles simple operations within each group well (e.g. compute `avg(salary)`), it requires a high degree of redundancy (joins, correlated subqueries) to express more complex operations within each group. This has been addressed in [7, 8].
- Second, OLAP queries may correlate results of groupings on *different* sets of attributes. These sets are usually related somehow (e.g. the one is subset of the other.)

The extended multi-feature syntax, a minor syntactic extension of SQL, addresses both of these issues and is a generalization of the multi-feature syntax discussed in [7]. A brief discussion of EMF SQL follows.

2.1 Extended Multi-Feature Queries

The idea behind the extended multifeature syntax [4] is simple. For each group, the user defines one or more *grouping variables*. Each grouping variable represents a subset of the entire relation, whose range is constrained by the *such that* clause. The defining condition of a grouping variable may contain comparisons between ordinary attributes and constants, aggregates of the group, and aggregates of *previously defined* grouping variables. As a result, one may define a series of selections and aggregations over the same grouping attributes. The group itself can be considered as one grouping variable, denoted as X_0. Aggregates of the grouping variables can appear in the select clause. For a more formal definition of EMF-SQL, see [5].

This small extension to SQL allows the user to express a large class of decision support queries in a simple and declarative fashion. This is mainly achieved because the group by clause acts as an implicit iterator over groups, the same way the `from` clause acts for the tuples of a relation. At the same time, grouping variables define the processing to be done for each value of the grouping attributes. This syntax also contributes to the second characteristic of OLAP queries, identified at the begin of Section 2.

2.1.1 Examples

Example 2.1: Suppose that We want to compute for each customer the average sale amount in "NY", in "NJ" and in "CT" (the tri-state area). This query *pivots* a portion of the `state` column and creates columns. Its expression in EMF SQL is:

```
Select cust, avg(x.amount),avg(y.amount),
  avg(z.amount)
From Sales
Where year=1997
Group By cust ; x,y,z
Such That  x.cust=cust and x.state='NY',
           y.cust=cust and y.state='CT',
           z.cust=cust and z.state='NJ'
```

Example 2.2: Suppose that one wants to determine, for each product, by which month had half of the 1998 yearly sales occurred. Here we need to compute a generalized median. This query can be expressed using the extended multi-feature syntax as [18]:

```
select Product, Month
from Sales
Where Year = 1998
Group By Product, Month ; X, Y, Z
Such That   (X.Product=Product and X.Month=Month),
            (Y.Product=Product and Y.Month<Month),
            (Z.Product=Product)
Having     sum(Y.amount) < sum(Z.amount)/2 AND
   sum(Y.amount)+sum(X.amount) >= sum(Z.amount)/2
```

The Sales relation is grouped by Product, Month. The grouping variable X denotes sales of the Product during the Month, Y denotes sales of the Product during previous Months, and Z represents sales during the entire year. The having clause selects for output only those months whose sales straddle the half-way mark of the yearly sales. Note that grouping variables Y and Z contain tuples that are not in the group.

This query shows how many other complex aggregates can be computed. If we were interested in the number of sales, as opposed to their dollar quantity, we would use count aggregates instead of sum aggregates. The quantity $sum(X.sales)/sum(Z.sales)$ is a percent-of-total aggregates. The grouping variable Y represents a running sum. With an additional constraint (e.g., $Y.Month > Month - 3$), it represents a moving window (e.g., for a moving average). □

Example 2.3: Assume that we are interested to find for each product the average quantity sold before and after each month of 1997 (a generalization of a moving-average aggregate).

```
Select Product, Month, avg(X.Quantity),
  avg(Y.Quantity)
From Sales
Where Year='1997'
Group By Product, Month ; X, Y
Such That   (X.Product=Product and X.Month < Month),
            (Y.Product=Product and Y.Month > Month)
```

For each product and month of "1997" we define two sets, X and Y. X contains all the sales of the current group's product *before* the current group's month (X.Month < Month) and Y the sales of that product after that month (Y.Month > Month.) □

2.1.2 Performance

As is reported in [4, 5], we wrote a translator that generates C or PL/SQL code from an EMF query. We wrote SQL and EMF queries for Examples 2.1 and 2.3, and generated PL/SQL and C programs from the EMF queries. We used an Oracle 7 database to execute the SQL and PL/SQL queries. In spite of the high overhead required to execute a PL/SQL program, these versions of the queries are substantially faster than the SQL versions. When implemented in C, the queries execute two orders of magnitude faster.

We also ran experiments with Oracle 8. We found that the first time we executed a query, the query execution time is similar to that achieved by Oracle 7. However on subsequent executions, the query was evaluated two to three times faster. Clearly Oracle 8 is performing a sophisticated processing to speed up these types of queries. However the C implementation remains orders of magnitude faster, reflecting the better evaluation algorithms.

2.2 Evaluation and Optimization of EMF Queries

In this section we present a direct implementation of extended multi-feature queries and optimizations of that implementation. All aggregate functions are presumed to be algebraic[1]. We start with two definitions.

Definition 2.1: A grouping variable Y *depends* on grouping variable X, if some aggregate value of X appears in the defining condition of Y. This is denoted as $Y \rightarrow X$. If the defining condition of a grouping variable Y contains aggregates of the group or grouping attributes, then the group is denoted as a grouping variable X_0 and we write $Y \rightarrow X_0$. The directed acyclic graph that is formed from the grouping variables' interdependencies is called *emf-dependency* graph. □

Definition 2.2: The *output* of a grouping variable X, denoted as *outp(X)* is the set of the aggregates of X that appear in either the such that clause, the having clause, or the select clause. □

2.2.1 Evaluation

Let H be a special table, called the *mf-structure* of an extended multi-feature query, with the following structure. Each row of H, called *entry*, corresponds to a group. The columns consist of the value of the grouping attributes, the aggregates of the group, and the aggregates of the grouping variables. Let $X_1, \ldots, X_n$ be the grouping variables of the query, ordered by a reverse topological sort of the emf-dependency graph.

The algorithm [5] performs $n + 1$ scans of the base relation. On scan i it computes the aggregates of X_i grouping variable (X_0 denotes the group.) As a result, if X_j depends on X_i (i.e. if aggregates of X_i appear in the defining condition of X_j), this algorithm makes sure that X_i's aggregates will have been calculated before the jth scan. Note also that given a tuple t on scan i, *all* entries of table H must be examined, since t may belong to grouping variable X_i of several groups, as in Example 2.2: a tuple t affects several groups with respect to grouping variable Z, namely those that agree on Product with t's Product.

This algorithm represents an efficient, self-join free direct implementation of the extended multi-feature syntax. This is the main contribution of the extended multi-feature syntax: complex decision support queries not only can be expressed intuitively and declaratively, but also there exists a *direct* mapping between the representation (syntax) and an evaluation algorithm that guarantees the answer in few scans of the base data.

2.2.2 Optimization

The EMF query evaluation algorithm in Section 2.2.1 is a naive implementation – many optimizations can be made (as is presented in length in [4, 5]) that greatly improve performance. For example, the evaluation can become very expensive if the mf-structure has a large number of entries, since on each scan, for every tuple, all H's entries are examined, resulting in an implicit nested-loop join. However, this is not necessary since one can identify, given a tuple t and a grouping variable X_i, a small number of entries (i.e. groups) that this tuple "affects". Consider Example 2.2 and grouping variable X. Given a tuple t, we know that only *one* entry of the mf-structure will be affected during the evaluation of

[1] Holistic aggregates require additional memory management considerations.

grouping variable X (the one that agrees on t's Product, Month values) because X denotes the group. As a result, a tuple t can belong only to X of *one* group. Therefore, only one entry is updated. The same holds for all grouping variables of Example 2.1. A syntatic analysis shows which indices on H should be built to accelerate the inner loop.

We note that while the evaluation algorithm is described works in main-memory, extensions to out-of-core are possible. In [4] we describe a multi-pass algorithm based on an automatic partitioning of the mf-structure. This algorithm, and other optimizations, are implemented in the EMF SQL prototype available at www.panakea.com.

3 Multiple Fact Tables

The expression and evalaution of nested aggregation requires access to data from multiple sources. We find that extending EMF SQL to access data from multiple sources to be of independent interest. In many situations, the data to be queried exists in several fact tables. Normally, querying these tables requires one or more joins. However, most of these queries express a type of looping that is similar to what is discussed in the previous section:

For each value v of a set of attributes S from one *base table, select a subset of tuples from* another *table (perhaps the same table). Then, using computed aggregates if needed, select another subset of tuples from another table and so on. Output aggregates of the selected subsets.*

The EMF syntax for multiple fact tables has only one small change. If a grouping variable ranges over a fact table other than the first one specified in the from clause, then the *range* of each grouping variable (i.e., the fact table that the grouping variable scans) must also be specified, in parentheses. If no explicit range is given, then the range of the grouping variable is the fact table in the from clause.

The evaluation of multi-fact table EMF queries is almost the same as for single fact table queries. A pass is made through a source fact table to build the mf-structure. Then one pass is made through the a fact table for each grouping variable. Let X_i be the grouping variable for the ith scan. If $range(X_i) = R_k$, then scan i is made over fact table R_k. The optimizations previously developed [4] can be applied with little change (see Section 5).

Lets consider an example query. Most of the remaining examples in this paper will use the following tables:

```
Purchases(account,prod-cat,day,month,year,amount)
    Calls(account,frmAC,frmTel,toAC,toTel,
          day,month,year,length)
WebLog(account,webSite,type,day,month,year,ts,length)
```

Example 3.1: For 1997, for each customer, find the months with average spending amount greater than the yearly average, and output the type of web-sites of each month that had average length greater than the customer's length of that month.

```
Select account, month, type
From WebLog
Group By account,month,type ; X(Purchases),
  Y(Purchases),Z,Q
Such That  X.account=account and X.month=month,
           Y.account=account,
           Z.account=account and Z.month=month
           and Z.type=type ,
           Q.account=account and Q.month=month
Having  avg(X.amount)>avg(Y.amount) and
        avg(Z.length)>avg(Q.length)
```

Expressing this query in SQL would require building four temporary tables (corresponding to Q, X, Y, and Z) and a four-way join to construct the output table. The query would be difficult to express and understand, and slow to evaluate.

Let us consider another example, which integrates data from two sources.

Example 3.2: Show the total amount of purchases, average length of calls, and average length of weblog for every month and for every customer whose total length of weblog is greater than or equal to their total length of calls.

```
Select account, month, sum(X.amount),
  avg(Y.length), avg(Z.length)
From Purchases
Group By account,month ; X,Y(Calls),Z(WebLog),
  Q(Calls), R(WebLog)
Such That  X.account=account AND X.month=month,
           Y.account=account AND Y.month = month,
           Z.account=account AND Z.month = month,
           Q.account=account,
           R.account=account
Having sum(Q.length)≥sum(R.length)
```

A restriction of our syntax is that all group-by attributes must be derived from a single table, eliminating the need to perform a join in the initial step of building the mf-structure. As the examples show, a large class of interesting queries can be expressed with this restriction. For some queries, the group-by attributes might need to be derived from more than one table. Example 3.2 illustrates the point. It might be the case that during some month, a customer makes no purchases, but makes several web accesses.

Rather than making the EMF semantics more complex, we assume that the user will pre-compute a table with the group-by entries needed for a query. We feel that this procedure does not substantially add to the complexity of the query, because the logic for computing the group-by entries and the logic for computing the aggregates are separate. Furthermore, the group-by table is likely to be computed in many queries, and is a good candidate for materialization anyway.

Evaluating multi-table EMF queries is as efficient as evaluating single-table EMF queries. In Example 3.2, six passes are needed to compute the query using the basic algorithm. However, this is only two passes through the entire data set. Pass-reduction optimizations (i.e., none of the grouping variable definitions depend on each other, see [4, 6]) show that only 1 pass through each fact table is required.

4 Multi-level Aggregation

Many data analyses require the use of complex, non-distributive aggregation. In many cases, the aggregate is a rollup aggregate (i.e., a summary of aggregates grouped by an additional dimension) One common example is the most-frequent aggregate, which is a max of a count. Another closely related query is, "For 1997, what are the total sales of each product during the month with the maximum total sales of the product?".

These aggregates can be phrased as *nested aggregation*, which reflects their rollup nature. For the example query we must first find the total sales of each product during each month of 1997. Then for each product, we must find the month with the maximum total sales. Notice that computing

nested aggregates requires that we specify nested levels of group-by attributes. In this example, the per-month groups are nested within the product groups.

We introduce some syntax to express nested aggregation EMF queries. Informally, the extensions to the EMF syntax are:

1. Group-by nesting is indicated by matching square braces.
2. The nested group-by attributes are indicated by a group-by statement at the end of the nested aggregate statement.
3. Grouping variables are listed after the group-by statement of their associated nesting block.
4. Nested aggregates are referred to by multiple levels of aggregation.

The definition of the group-by variables can make use of the "visible" group-by attribute values of the group, and the "visible" aggregates of previously defined grouping variables. We defer the definition of "visible", and present examples to make the meaning intuitively clear.

Example 4.1: The motivating example can be expressed as follows.

```
Select prod-cat, max(sum(X.amount))
From Purchases
Group By prod-cat
Such That  [(X.prod-cat=prod-cat and X.month=month)
           Group By month ; X]
```

The operation of this query can be visualized by the example shown in Figure 1. We evaluate the query by building an mf-structure table with group-by attribute prod-cat. Each mf-structure entry contains space for the aggregate max(sum(X.amount)), and also a nested mf-structure with group-by attribute month. and aggregate sum(X.amount). We make a scan through the Purchases table to compute the sum(X.amount), and then for every mf-structure entry, make a scan through the nested mf-structure to compute max(sum(X.amount)). We note the relationship between the nested aggregation table and nested relations [11] (which motivates the name nested aggregation).

This model of a nested EMF-query is useful for gaining an intuition about the semantics of query evaluation. When we compute an aggregate of a grouping variable, we use EMF semantics, but when we compute an aggregate of an aggregate (a *nested aggregate*), we use MF semantics (i.e., each mf-structure entry is isolated from the others). An mf-structure entry can have more than one nested mf-structure, and the nested mf-structure entries can also have nested mf-structure entries. A nested mf-structure entry can "see" fields in its parent (ancestor) mf-structure entry, but is isolated from other nested mf-structure entries.

To simplify the presentation of the nested EMF query evaluation algorithm, we keep the semantics that mf-structure entries are single valued. We choose to use multiple mf-structure to represent the nested mf-structure. However other approaches are possible. One uses a single flat mf-structure. We also note that it is possible, and perhaps desirable, to develop an evaluation algorithm on the nested EMF-table model.

In the multiple mf-structure approach, the fundamental extension to the regular EMF query processing algorithm is

prod-cat	max(sum(X.sales))	month	sum(X.sales)
shoes	22	1	22
		2	17
		3	12
socks	19	2	15
		4	19
coats	8	5	8
hats	4	6	4

Figure 1: Nested aggregation with nested mf-structures.

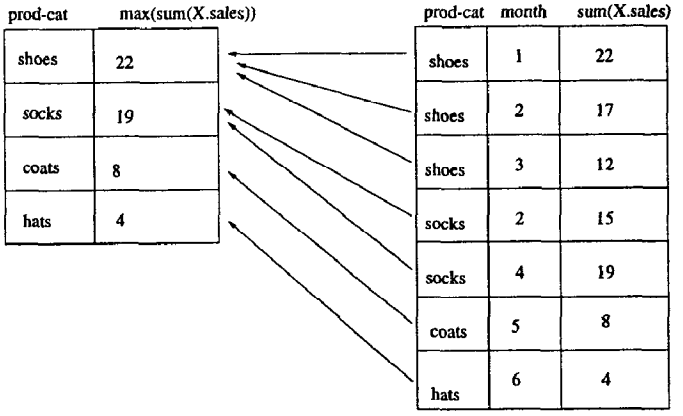

Figure 2: Evaluating a nested group-by query.

to use one table per nested group-by block, and a restricted type of join between them. The mf-structure used in evaluation of the example query is shown in Figure 2. The first table (which contains the final answer) is grouped by prod-cat and contains max(sum(X.amount)) summaries, while the second table is grouped by prod-cat and month and contains (sum(X.amount)) summaries. The second table is built using the usual EMF algorithm, and then the first table is built from summaries on the second.

The aggregates in a nested aggregation query can be any commutative aggregates, including more complex ones such as count_of_min(). More generally, we can use *linked* aggregates. Let X and Y be two values defined at a group-by nesting level. Then, *linked_aggr*(X,*aggr*(Y)) returns the value of x in one of the entries of the nested mf-structure in which Y has the value *aggr*(Y). The aggregate *aggr*(Y) must return a value that Y actually takes (e.g., min, max, median, etc.). Because there might be several entries in which Y = *aggr*(Y), the linked aggregate (e.g., any, first, last) defines which of the entries supplies the value of Y. The quantity X can be an aggregate value defined at the nested mf-structure, or it can be one of the group-by attributes of the nested mf-structure. For example, any(month, max(sum(X.amount))) returns the month with the maximum amount. We note that the quantity returned by a linked aggregate can be

computed without using a linked aggregate, but at the cost of duplicating the nested aggregation block.

Example 4.2: For each credit card customer, find the amount spent in the maximum spending month, and that month.

```
Select account, year, max(sum(X.amount)),
  any( month, max(sum(X.amount)) )
From Purchases
Group By account, year
Such That  [(X.account=account and X.year=year and
           X.month = month) Group By month ; X]
```

This query is evaluated in a manner similar to that of the previous example. When the table grouped by (account, year) is built from the table grouped by (account, year, month), the any aggregate is computed. The current value of the max(sum(X.amount)) is stored, and whenever it is updated, the month attribute is recorded.

The nested group-by EMF syntax can use aggregates of previously defined grouping variables to define the range of new grouping variables. The grouping variable can be defined at a different level of aggregation. The following example also uses a having clause at the lower level of aggregation. The having clause selects the bottom level aggregates that can be further aggregated in the top-level result (i.e., MF semantics).

Example 4.3: For each credit card customer, find the months of 1997 that have average spending amount greater than the 1997 yearly average, and output the minimum one for each customer.

```
Select account, min(avg(Y.amount)),
  any( month, min(avg(Y.amount)) )
From Purchases
Group By account ; X
Such That ( X.account=account and X.year=1997 ),
   [ ( Y.account = account and Y.year = 1997 and
   Y.month=month)
   Having avg(Y.amount) > avg(X.amount)
   Group By month ; Y ]
```

Figure 3 illustrates the query evaluation, which starts by building two mf-structures, the first grouped by account, and the second grouped by account and month. A pass is made over the Purchases table to build the mf-structure entries. Then a second pass is made over the Purchases table to compute avg(X.amount), grouped by account. Because this value is used in the nested group-by table, it is *reflected* to the second table (the quantity avg(X.amount) is defined at the parent mf-structure, and therefore is visible at the child mf-structure). The reflection can be accomplished in several different ways, but we will use the convention that a tuple t from the Purchases table is applied to all entries in both mf-structures that have a matching value of account. A third pass is made over the Purchases table to compute the avg(Y.amount). A fourth pass is made over the second mf-structure to compute min(avg(Y.amount)) and any(month, min(avg(Y.amount))) in the first mf-structure. Only those entries in the second mf-structure that satisfy the having clause are selected for use in the scan.

The above example illustrates two key ideas. First, all computation is kept completely local. Because the avg(X.amount) aggregate is reflected to the nested mf-structure, entries in the second mf-structure can be independently

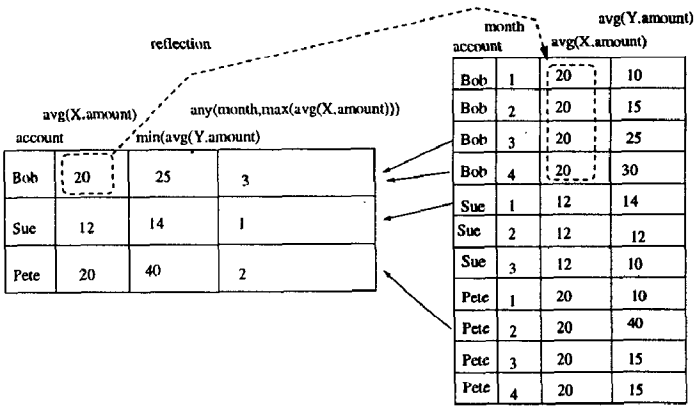

Figure 3: Evaluating a nested group-by query with a having clause.

evaluated for inclusion in the scan. Second, a having clause at nested group-by becomes a such that clause when the nested aggregate is computed. In general, nested aggregates define implicit grouping variables that range over a mf-structure. The having clause in a nested group-by block lets us introduce multi-feature (though not extended multi-feature) semantics. The group-by nesting can be extended to multiple levels, as the following example shows.

Example 4.4: For each credit card customer, find the amount spent in the minimum of the maximum-spending month in any year.

```
Select account, min(max(sum(X.amount))),
 first( year, max(sum(X.amount)) ),
 first( any( month, max(sum(X.amount)) ),
 min( max(sum(X.amount))) )
From Purchases
Group By account
Such That [[ X.account=account and X.year=year
   and X.month = month; Group By month ; X ]
   Group By year ]
```

This query is evaluated in the usual way, by using three mf-structures. Finally, we show an example with multiple hierarchies.

Example 4.5: For each customer, count the number of months during which the customer makes a number of purchases larger than the number of times that the customer has purchased their most commonly purchased item.

```
From Purchases P
Select account, count(count(Y.account))
Group By account
Such That
   [ (X.account=account and X.prod-cat=prod-cat)
   Group By prod-cat ; X ]
   [ (Y.account=account and Y.month=month)
   Group By month ; Y
   Having count(Y.account)>max(count(X.account)) ]
```

In this query, count(X.account) counts the number of times a customer has purchased a particular product. Therefore,

max(count(X.account)) is the number of times the customer has purchased their most favorite product. count(Y.account) is the number of purchases a customer has made in a particular month. We can determine if the customer has made a sufficiently large number of purchases during a month by testing count(Y.account) > max(count(X.account)). To count the qualifying months for a customer, we evaluate count(count(Y.account)).

The query is evaluated in a manner similar to the other examples. Three mf-structures are used, the first grouped by account, the second grouped by account, prod-cat, and the third grouped by account, month. The max(count(X.account)) aggregate is reflected from the first mf-structure to the third

In order to precisely define the syntax and evaluation of nested EMF queries, we need to make a number of definitions. Unless otherwise stated, the syntax of nested EMF is the same as EMF.

1. A *group-by block* is indicated by a matching set of square braces (except for the top level group-by block, which is not enclosed by braces). Each group-by block defines one of more grouping attributes, and zero or more explicit grouping variables, and optionally contains a having clause. The grouping variables defined in the group-by block must be listed after the group-by clause, using the usual EMF syntax.

2. Let $\mathcal{N} = \{N_i,\ i = 0,\ldots,n\}$ to be the set of $n+1$ different group-by blocks in a query. N_0 is the root group-by block. The children of a group-by block N, $children(N)$, is the set of all group-by blocks directly nested within N. We similarly define $parent(N)$, $ancestors(N)$, and $descendents(N)$.

3. The *level* of group-by block N, $level(N)$, is the number of matching braces enclosing N. For example, $level(N_0) = 0$.

4. The group by attributes of a group-by block N, $G(N)$ is the set of attributes listed in the group-by clause of N.

5. The group-by *tag* associated with a group-by block N, $tag(N)$ is defined by

$$tag(N) = G(N) \cup \left(\cup_{N' \in ancestor(N)} G(N')\right)$$

6. Each block N has an mf-strucutre, $mf(N)$. The group-by attributes of $mf(N)$ are $tag(N)$.

7. A *nested aggregate* is an expression $agg_n(agg_{n-1}($ $\ldots agg_0(X)\ldots)$, where each agg_i is an aggregate function and $n \geq 1$. A nested aggregate has *subaggregates*, $agg_{n-1}(\ldots agg_0(X)\ldots),\ldots,agg_0(X)$. The child of nested aggregate $agg_n(\ldots agg_0(X)\ldots),\ldots,agg_1(X)$ is $agg_{n-1}(\ldots agg_0(X)\ldots),\ldots,agg_1(X)$, and the child of $agg_0(X)$ is X.

   In the case of linked aggregates, the nested aggregates form a tree in with branches at each linked aggregate. A linked aggregate will have two children, corresponding to both parameters. Note that the syntactic sugar of linked aggregates requires that we strip off one level of aggregation from the second parameter. One or both of these children might be group-by attributes.

8. An *explicit* grouping variable is a named grouping variable. The *block* of an explicit grouping variable V, $block(V)$, is the nested group-by block in which V is defined. For shorthand, we define $tag(V) = tag(block(V))$ and $mf(V) = mf(block(V))$. Also, we define $mf($ $aggr(V)) = mf(V)$ and $block(aggr(V)) = block(V)$ for each aggregate of V. The *range* of the explicit grouping variable is a fact table.

9. The *implicit* grouping variables are defined implicitly by subaggregates of nested aggregates. One implicit grouping variable is defined for each unique subaggregate of each nested aggregate. Let the block of the explicit grouping variable X be N. The implicit grouping variable associated with $agg_0(X)$, $V(agg_0(X))$, is defined at $parent(N)$, and ranges over $mf(N)$ (i.e., $range(V(agg_0(X)) = mf(N))$. If the implicit grouping variable $V(agg_{i-1}(\ldots agg_0(X)))$ is defined at node N then $V(agg_i(agg_{i-1}(\ldots agg_0(X)))$ is defined at $parent(N)$, and ranges over $mf(N)$.

   In the case of linked aggregates, $V(agg_i(agg', agg''))$ is defined at $parent(agg') = parent(agg'')$, and ranges over $mf(agg') = mf(agg'')$ (note that the syntactic sugar of linked aggregates requires that we strip off one level of aggregation from the second parameter).

   As is the case with explicit grouping variables, $mf(aggr(V)) = mf(V)$ and $block(aggr(V)) = block(V)$.

10. An explicit grouping variable depends on explicit grouping variables in its definition, and on the implicit grouping variables of the nested aggregates in its definition. An implicit grouping variable depends on the grouping variable(s) of its child nested aggregate(s), and on the explicit and implicit grouping variables in the having clause of its block. The dependence graph must be acyclic.

11. A value of a group-by attribute a is *visible* at block N if $a \in tag(N)$. An aggregate value $aggr$ is visible at block N if $block(aggr) \in N \bigcup ancestors(N)$. All group-by attributes and aggregate values used as constants in the such that or having clause in block N must be visible at block N.

12. If an aggregate $agg(V)$ of a grouping variable V defined at block N is used in the definition of a such that or having clause in a block $N' \in descendent(N)$, then $agg(V)$ is *reflected* to N'.

13. The aggregates defined at block N, $AGG(N)$ is the set of all aggregates referenced in the query of the implicit and explicit grouping variables defined at N, together with the aggregates reflected to N.

Lets consider an example to illustrate these definitions.

Example 4.6: Let us recall example query 4.5. A diagram of this query is shown in Figure 4. This query has three group-by blocks: the top-level block N_0, the block in which X is defined N_1, and block in which Y is defined N_2. We see that $children(N_0) = descendent(N_0) = \{N_1, N_2\}$, and that $parent(N_1) = ancestor(N_1) = parent(N_2) = ancestor(N_2)$ $= \{N_0\}$. The level of N_0 is 0, while the level of N_1 and N_2 is 1. The group-by attributes of the blocks are $G(N_0) =$ {account}, $G(N_1) =$ {prod − cat}, and $G(N_2) =$ {month}. The tags associated with the blocks (and therefore with their mf-structures) are $tags(N_0) =$ {account}, $tags(N_1) =$ {account, prod − cat}, and $tags(N_2) =$ {account, *month*}.

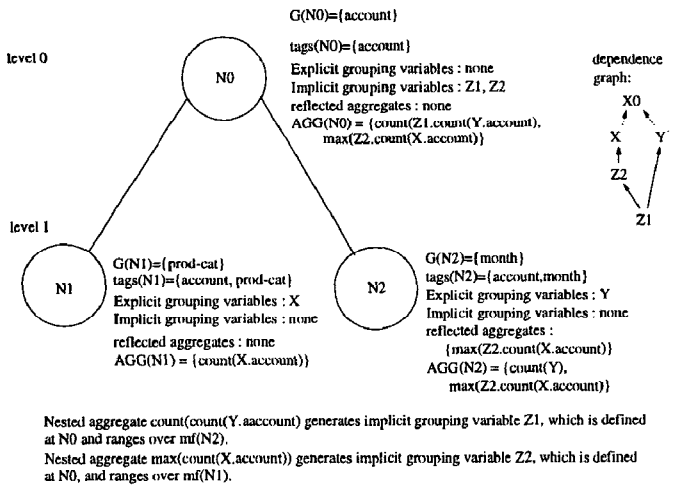

Figure 4: Analysis of nested aggregate query example 4.5.

There are two nested aggregates in the query. The nested aggregate count(count(Y.account)) contains one level of nesting, and therefore contains one subaggregate count(Y.account). This subaggregate generates an implicit random variable Z_1 which is defined at N_0 and which ranges over N_2. The nested aggregate max(count(X.account)) also contains one level of nesting, and therefore contains one subaggregate count(X.account). This subaggregate generates an implicit random variable Z_2 which is defined at N_0 and which ranges over N_1. The explicit grouping variables have no dependencies (we show a dependence on the initial pass, X_0 that builds the mf-structures, but actually X and Y are pass-reducible to X_0). The implicit grouping variable Z_2 depends on X, and the implicit grouping variable Z_1 depends on Y and on Z_2. Note that the aggregate max(count(X.account)) is defined at N_0, but is reflected at N_2. Therefore, $AGG(N_0) =$ {count(count(Y.account)), max(count(X.account))}, $AGG(N_1) =$ {count(X.account)}, and $AGG(N_2) =$ {count(Y.account), max(count(X.account))}. □

The extension to the evaluation algorithm required to handle nested group-by aggregation is to handle reflected aggregates. This requires that during a scan we apply tuples to all of the relevant mf-structures.

Algorithm 4.1: Evaluation of nested EMF queries:

1) Enumerate the grouping variables by a topological sort.
2) Stage 0 makes a pass through the fact table to build all of the mf-structures. Let there be k blocks, $N_0, \ldots, N_{k-1}$. For every tuple t encountered
   a) for $i = 0$ through $k-1$
      i) if $t[tags(N_i)]$ does not exist in $mf(N_i)$, create an entry in $mf(N_i)$ with key $t[tags(N_i)]$, and initialize the aggregates $AGG(N_i)$.
3) Let there be k group-by variables $X_1 \ldots, X_k$, enumerated in the order of a topological sort on the dependences.
4) for each group-by variable X_i execute stage i.
   a) Make a scan through the mf-structure $range(X_i)$. For every tuple t, apply t to every mf-structure $mf(N)$ such that there is an aggregate $agg(X_i) \in AGG(N)$.

Example 4.7: Let us examine how example query 4.5 would be evaluated. A first pass is made through the Purchases table to construct the mf-structure entries for each of $mf(N_0)$, $mf(N_1)$, and $mf(N_2)$. Let us suppose that the topological sort of the implicit and explicit grouping variables on their dependences is $(X, Y, Z2, Z1)$. A second pass is made through the Purchases table to evaluate aggregates of X. Since aggregates of X only occur in $AGG(N_1)$, only $mf(N_1)$ is updated. A third pass is made through the Purchases table for grouping variable Y, and only $mf(N_1)$ is updated. A pass is made through $mf(N_1)$ to evaluate aggregates of $Z2$, which is defined at N_0 but has an aggregate reflected to N_2. So, both $mf(N_0)$ and $mf(N_2)$ are updated. Finally, a pass is made through $mf(N_2)$ to evaluate aggregates of Z_1, and only $mf(N_0)$ is updated.

An alternative query evaluation algorithm that uses a single table can be developed. The idea is to "flatten" the k mf-structures into a single table. The group-by key for the flattened mf-structure is $\cup_N G(N)$. A group-by block N' is represented by the mf-structure entries with valid values in $tag(N')$ and null values in $\cup_N G(N) - tag(N')$. Every implicit and explicit grouping variable is defined on the mf-structure, and the range of every implicit grouping variable is the mf-structure. One nice aspect of this approach is that reflection occurs automatically.

An extension to the nested EMF syntax allows nested group-by blocks to be named, and explicit grouping variables to range over the named group-by blocks. This extension can eliminate the need for linked aggregates, and can permit certain queries to be expressed more concisely. Space constraints prevent an in-depth discussion. We note that the analysis and evaluation of the extended syntax is almost the same as the basic nested EMF syntax that we have presented.

5 Optimizations

Because the query evaluation algorithm for the nested and multi-table EMF syntax is almost identical to the regular EMF query evaluation algorithm, the optimizations that have been developed for EMF queries can be carried over to this new setting with few or no modifications. In this section, we discuss previously proposed optimizations, and also several new issues.

Relative Entries With EMF semantics, a tuple from the fact table can be applied to several mf-structure entries (which is why EMF is so expressive). The basic EMF query evaluation algorithm must test every mf-structure entry for every fact table tuple on every pass to determine if the tuple is a member of a grouping variable for that mf-structure entry. However, the such that clause usually places significant restrictions on the range of the grouping variables. These restrictions are formalized as *relative sets* in [4], and can be used to define indices on the mf-structure.

The definitions of relative sets of the explicit grouping variables are unaffected by the extensions to EMF proposed in this paper. The implicit grouping variables have MF semantics, so a tuple referenced by an implicit grouping variable is applied to only one parent mf-structure entry (which can be found by hashing).

Dependency Analysis The basic EMF query evaluation algorithm makes a pass through the fact table for each grouping variable. The number of passes can be reduced by computing aggregates of several grouping variables at the same

time. The grouping variables that can be processed together can be determined by an analysis of the dependency graph. If the grouping variables preceding X and Y have already been processed, then on the next scan aggregates of both X and Y can be computed.

For the extensions presented in this paper, we need to make the additional requirement that $range(X) = range(Y)$ in order to apply the optimization. If $range(X) \neq range(Y)$, then aggregates of X and Y can be computed in parallel. This optimization is particularly effective if X and Y affect non-overlapping mf-structures. We note that our analysis of nested EMF queries are likely to lead to a large number of implicit grouping variables. By using this optimization, they will all be processed in a few passes (perhaps one) over the nested mf-structure.

Parallel Search Since each mf-structure entry is processed in isolation, the mf-structure can be partitioned and distributed to multiple processors, and a pass over a fact table can be performed in parallel. Because nested mf-structures are "contained" in mf-structure entries, the root mf-structure partition also defines partitions of the nested mf-structures. Passes over the nested mf-structures can be performed in parallel with no communication between processors.

Out-of-core Processing If the mf-structure does not fit into memory, it can be partitioned (as discussed above) and processed in a partition-wise manner, with only the current partition being main-memory resident.

With nested EMF queries, we can make another optimization that trades memory for additional table scans. We observe that a nested mf-structure $MF(N)$ can be deleted once the last grouping variable whose range is N has been processed. Further, $MF(N')$ does not need to be constructed until the first grouping variable defined at N' is processed. If all grouping variables whose range is N can processed before all grouping variables defined at N', then after processing the last grouping variable whose range is N, we can delete N and reuse its space to construct N'. For an example, this optimization would work on Example 4.5. Another possible optimization is to bring into active memory only the mf-structures being referenced by the current grouping variable.

6 Previous Work

Several papers discuss query optimization techniques in the context of standard SQL that are particularly applicable in the processing of complex OLAP queries. Graefe surveys various principles and techniques [13]. The issues of aggregation and join have been studied separately until quite recently, when a number of papers on optimization of *both* aggregation and join have appeared [30, 9]. Yan and Larson in [30] describe a class of transformations that allow the query optimizer to push a group-by past a join (eager aggregation) or pull a group-by above a join (lazy aggregation). In a similar direction, Chaudhuri and Shim in [9] 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.

Although these techniques may speed up complex aggregate processing, they fail to recognize the redundancy incorporated in the SQL expressions of complex OLAP queries: frequently these queries are expressed as multiple joins and aggregations, but can be evaluated much simpler. Chatziantoniou and Ross in [8] observed that many common decision support queries, called group queries, can be processed group-wise, i.e. the base relations can be partitioned and processed in a group-by-group fashion, although their expression in SQL involves several joins.

However, there exist OLAP queries that are not group queries. Extended syntaxes, aiming to succinctness, try to deal with inadequate optimization techniques. Succinctness means that the users can write and understand queries easier, and the system can optimize them better since there is a tight coupling between representation and evaluation.

Several authors have pointed out that SQL cannot simply express a number of queries involving grouping and aggregation [14, 20] Gray, Bosworth, Layman and Pirahesh propose a relational aggregation operator, called datacube, which is useful in data analysis applications [14]. QUEL [28] allows queries in which the range of tuples over which an aggregate is computed can be specified. Kimball and Strehlo in [20] argued that SQL should be extended in order to be more powerful (in both syntax and performance) for queries related to grouping. They 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.

Our syntax can neatly solve (for a known number of columns) the so-called *value-to-attribute* conflict or *pivoting*, 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 [19, 21].

7 Conclusions

Complex data analysis on very large data warehouses require an advanced querying tool. One particular limitation of existing OLAP databases is their ability to express complex aggregates. In object-relational databases, the user can introduce UDAFs to compute arbitrary aggregates. However, defining UDAFs places a significant programming burden on the user. In addition, UDAFs are difficult to optimize.

In previous papers, we have introduced the Extended Multi-Feature syntax, which is a small extension to SQL. EMF SQL can express many complex aggregation queries in a simple and succinct manner. Examples include percent-of-total, moving-average, or median aggregates, and pivoting queries. In addition, the queries can be evaluated using an efficient query processing algorithm that is closely tied to the EMF syntax.

In this paper, we demonstrate that two major extensions to EMF-SQL can be accomplished with minor adjustments to the EMF-SQL syntax and evaluation algorithm. The primary extension is to allow the declarative expression of complex non-distributive aggregates through *nested aggregation*, or aggregation over an aggregated table. An example of a nested aggregate is a most-frequent aggregate, i.e. max of count. Very complex nested aggregation queries can be expressed simply and succinctly, and evaluated efficiently. The second extension is to allow the grouping variables to range over multiple fact tables. This extension lets us apply the power of EMF to the problem of integrating data from multiple sources.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *22nd VLDB Conference*, pages 505–521, 1996.

[2] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *IEEE International Conference on Data Engineering*, 1997.

[3] D. Chamberlin. *Using the New DB2.* Morgan Kaufman, 1996.

[4] D. Chatziantoniou. Ad-Hoc OLAP : Expression and Evaluation. In *International Conference on Data Engineering (ICDE) - to appear*, March 1999. See also www.cs.stevens-tech.edu/ damianos.

[5] D. Chatziantoniou. Evaluation of ad-hoc OLAP: In-place computation. In *Proc. 11th Intl. Conf. Scientific and Statistical Database Management*, pages 34–43, 1999.

[6] D. Chatziantoniou and T. Johnson. Decision support queries on a tape-resident data warehouse. *IEEE Computer (to appear).*

[7] D. Chatziantoniou and K. Ross. Querying multiple features of groups in relational databases. In *22nd VLDB Conference*, pages 295–306, 1996.

[8] D. Chatziantoniou and K. Ross. Groupwise processing of relational queries. In *23rd VLDB Conference*, 1997.

[9] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB Conference*, pages 354–366, 1994.

[10] M. Corp. OLE DB for OLAP design specifications – beta 2. http://www.microsoft.com/fata/oledb/olap/prodinfo.html.

[11] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, 1994.

[12] F. Gingras and L. Lakshmanan. nD-SQL: A multi-dimensional language for interoperperability and OLAP. In *Proc. 24th Intl. Conf. on Very Large Data Bases*, pages 135–145, 1998.

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

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

[15] M. Gyssens and L. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of the 23rd VLDB Conference*, pages 106–115, 1997.

[16] Illustra Information Technologies. *Illustra User's Guide.* 1995.

[17] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relationsl dbms. In *Proc. ACM SIGMOD Conference*, pages 379–389, 1998.

[18] S. Johnson and D. Chatziantoniou. Extended sql for manipulating clinical warehouse data. In *American Medical Informatics Association*, 1999.

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

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

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

[22] C. Li and S. W. Wang. A data model for supporting on-line analytical processing. In *to appear in International Conference on Information and Knowledge Management*, pages 81–88, 1996.

[23] J. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellman, J. Kupsch, and D. DeWitt. Building a scalable geospatial database system: Technology, implementation, and evaluation. In *ACM SIGMOD*, pages 336–347, 1997.

[24] C. Red Brick Systems, Los Gatos. *RISQL Reference Guide, Red Brick Warehouse VPT Version 3.* 1994.

[25] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proc. of the ACM SIGMOD Conf.*, pages 343–354, 1998.

[26] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex query decorrelation. In *International Conference of Data Engineering*, pages 450–458, 1996.

[27] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD, Conference on Management of Data*, pages 104–114, 1995.

[28] M. Stonbraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Trans. on Database Systems*, 1(3):189–222, 1975.

[29] M. Stonebraker and D. Moore. *Object-Relational DBMSs – The Next Great Wave.* Morgan Kaufman, 1996.

[30] W. P. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *VLDB Conference*, pages 345–357, 1995.

[31] Y. Zhao, P. Deshpande, J. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *ACM SIGMOD, Conference on Management of Data (to appear)*, 1998.