Evaluation of Ad Hoc OLAP : In-Place Computation

Damianos Chatziantoniou
Department of Computer Science,
Stevens Institute of Technology
damianos@cs.stevens-tech.edu

Abstract

Large scale data analysis and mining activities, such as identifying interesting trends, making unusual patterns to stand out and verifying hypotheses, require sophisticated information extraction queries. Being able to express these data mining queries concisely is of major importance not only from the user's, but also from the system's point of view. Recent research in OLAP has focused on datacubes and their applications; however, expression and processing of ad hoc decision support queries has been given very little attention. In this paper we present an appropriate framework for these queries and introduce a syntactic construct to support it. This SQL extension allows most OLAP queries, such as complex intra- and inter-group comparisons, trends and hierarchical comparisons, to be expressed in a compact, intuitive and simple manner. However, this syntactic extension is not the focus of this paper. This succinct representation of a complex OLAP query translates immediately *to a novel, simple and efficient evaluation algorithm. We show how to optimize, analyze and parallelize this algorithm and discuss issues such as multiple query analysis and scaling. This algorithm constitutes the main contribution of this paper. Finally we introduce our implementation on top of a commercial system and present several experimental results of real-life queries that show orders of magnitude of performance improvement in certain cases. We argue that this tight coupling between representation and algorithm is essential to efficient processing of ad hoc OLAP queries.*

1. Introduction

Collecting data is easier today than ever before. Performing complex analysis on these data to identify interesting aspects is vital to many companies. Users may ask complex data analysis queries in order to identify irregularities or unusual behavior, or to verify hypotheses. It is important to study these queries and try to "understand" them. That leads to better execution plans and, consequently, performance. In our framework, the user is able to define several interesting features of groups, compare them, and have unusual aspects of data to stand out. As a result, these OLAP queries can be also considered as data mining queries [1].

1.1. Ad hoc OLAP Queries

Although significant research has been conducted on datacubes [12], both in terms of modeling [2] and evaluation [1, 17], little has been done on query optimization of complex *ad hoc* decision support queries, despite of their importance. To express such queries in SQL, a high degree of redundancy is required: multiple self-joins, correlated subqueries and repeated group-bys. This leads to complicated queries, difficult to understand and optimize. Standard query processing techniques [8, 22] help somehow. The problem is that a traditional SQL optimizer will not consider the "big picture", but will try to optimize a series of joins and aggregations, which is *not* always the best approach. In [7] we have addressed this issue and provided techniques to combine joins and aggregations into a more general operation. Similar concerns are discussed in [23]. However, there are many important queries that do not fall in this framework. Consider some typical OLAP queries over the following relation:

```
Sales(customer, product, day, month, year,
                quantity)
```

Q1. Assume that we are interested to check the total sales of the products during January, February and March (the winter months) of 1997: *"For 1997, show for each product the total of January, the total of February and the total of March sales (in three columns)."* (intra-group comparisons, pivoting)

Q2. One may want to identify those months of 1997 that were "significant" for the sales of a product: *"For each*

[1] To a similar direction, some researchers [15, 19] have proposed SQL extensions and query processing techniques to accomodate the discovery of association rules in SQL.

product and sales of 1997, show the product's average sale before and after each month of 1997." (trends)

Q3. Similarly, one may be interested to find changes in a product's sales on a monthly basis, by comparing the previous and the following month's sales to the current month's sales: *"For each product, count for each month of 1997 how many sales of the previous and how many sales of the following month had quantity greater than that month's average sale."* (trends)

Q4. Suppose that we want to identify "good" and "bad" months of a product's sales, as well as interesting irregularities: *"For each product show each month's total sales as percentage of this product's year-long total sales."* (hierarchies)

Q5. Similarly, we may want to identify "good" and "bad" months of a product's sales, utilizing information for other products: *"For each product, find for each month the total of the sales with quantity greater than the all-product year-long average sale, as percentage of the product's year-long total sales."* (hierarchies)

Q6. Finally, one can discover customers who have the potential to increase their purchases in one or more products with the following query: *"For each customer, show for each product the customer's average sale, and the other customers' average sale (for the same product)."* (inter-group comparisons)

Standard SQL representations of these queries require several views joined together or correlated subqueries. These complex representations lack succinctness, a necessary aspect for efficient optimization. For example, Query Q1 could be expressed in standard SQL with a 3-way self-join[2] (views could be used as well) :

```
select x.product, sum(x.quantity),
       sum(y.quantity), sum(z.quantity)
from Sales x, Sales y, Sales z
where x.product=y.product and x.month=1 and
      x.year=1997 and y.product=z.product
      and y.month=2 and y.year=1997 and
      z.month=3 and z.year=1997
group by x.product
```

If the optimizer of the system does not understand the particular structure of this query and executes it as a 3-way self-join (and this is usually the case) evaluation will be very expensive, even for small relations.

[2]To have a row for every product in the answer, joins must be replaced by outer-joins since some products may have NULL values for January, February, or March.

1.2. Syntax : Looping is Important

SQL is the standard query language for relational databases. The goal is to extend it minimally to express in a simple and declarative way most of the ad hoc decision support queries. In order to achieve this goal, we must understand the *nature* of ad hoc OLAP queries. The following observation helps in formulating the appropriate syntactic extension.

Consider Query Q2. The main idea is the following: *for each* value (p, m) of `(product, month)` attributes we want to define two subsets of `Sales`, $X_{(p,m)}$ and $Y_{(p,m)}$, where the first subset $X_{(p,m)}$ contains the sales of product p prior to month m, and the second subset $Y_{(p,m)}$ contains the sales of product p following month m [3]. Then we want to compute the average quantity of $X_{(p,m)}$ and $Y_{(p,m)}$. This process can be expressed by the following programming code :

```
for each (p,m) in (product, month) {
  avg_x = avg(t), where tuples t have
                  t.product=p and t.month<m;
  avg_y = avg(t), where tuples t have
                  t.product=p and t.month>m;
  output(p,m,avg_x,avg_y);
}
```

We believe that the ability to iterate over the values of one or more attributes domain, coupled with the ability to define for each such value one or more "interesting" subsets of the relation constitutes the gist of complex data analysis. The challenge is to provide the user with this "looping" ability without sacrificing the declarativeness of SQL. We argue that this can be achieved with our proposed syntax. The `group by` clause acts as an implicit iterator over the values of the grouping columns, the same way the `from` clause acts as an implicit iterator over the tuples of a relation. We also show that this syntax translates to an efficient, optimizable implementation.

1.3. Evaluation : In-Place Computation

Traditionally, an SQL query translates to a relational algebra expression and the query processor optimizes this expression. A complex query may involve several join and aggregation operators. The job of the optimizer is to commute operators in an equivalent and optimal way, and choose appropriate algorithms for each operator [9]. Consider once again Query Q2. One possible SQL formulation is presented below.

[3]Note that $X_{(p,m)}$ and $Y_{(p,m)}$ are *not* subsets of the group corresponding to the (p, m) value.

```
create view B1(product,month,avg_x) as
select x.product, x.month, avg(y.quantity)
from Sales x, Sales y
where  x.product=y.product and
       x.month > y.month
group by x.product, x.month

create view B2(product,month,avg_y) as
select x.product, x.month, avg(y.quantity)
from Sales x, Sales y
where  x.product=y.product and
       x.month < y.month
group by x.product, x.month

select B1.product, B1.month, avg_x, avg_y
from B1, B2
where  B1.product=B2.product and
       B1.month=B2.month
```

Traditional systems would try to formulate for each (p, m) value the $X_{(p,m)}$ and $Y_{(p,m)}$ sets and aggregate over these. An interesting order could be identified on `product` attribute [20] and hash-based or sort-based techniques [11] could be employed. However, a key observation could lead to a better implementation : we are *only* interested in *distributive* (or *algebraic*) aggregates over $X_{(p,m)}$ and $Y_{(p,m)}$ and not the sets per se. One could create a hash table on `product` attribute; each hash entry may contain up to twelve subentries corresponding to the twelve months of 1997. Each subentry corresponds to one row of the resulting table. Then, one scan of the `Sales` relation would be sufficient to compute the answer. Each scanned tuple t maps to a particular hash entry and one or more subentries are updated, according to the `month` value of t.

Our implementation takes advantage of the particular structure of our extended SQL queries. The algorithm scans one or more times the base relation and for each scanned tuple t, the rows of the resulting table affected by t are identified and updated appropriately. There are no intermediate tables. Our optimization techniques try to (i) reduce the number of scans, (ii) represent the resulting table optimally by an appropriate data structure, and (iii) parallelize the evaluation.

1.4. Contributions

Syntax A syntactic extension of SQL that can express most ad hoc decision support queries concisely and succinctly is introduced. This allows the user to write, understand and maintain complex OLAP queries significantly easier (although SQL's expressive power is not extended.)

Evaluation Evaluation of queries expressed with our syntactic construct maps directly to an efficient evaluation algorithm, easily optimizable. Furthermore, the techniques presented are highly scalable and well-suited for large data warehouses. We believe that this is the main contribution of this paper.

1.5. Outline

Section 2 describes the proposed syntax and its semantics. We give the details of the evaluation algorithm in Section 3 and present several optimizations in Section 4. Our implementation and performance results are discussed in Section 5. We conclude in Section 6 presenting related and future work.

2. Proposed Syntax and Semantics

In [6] we have introduced the concept of *multi-feature queries* which has proven useful for certain OLAP and datacube queries[18]. In this section we extend slightly this syntax, covering however a significantly larger class of data analysis queries. These queries are called *extended multi-feature* queries (EMF queries.)

2.1. Multi-Feature Syntax

Query Q1 constitutes a multi-feature query and can be expressed with the following extended SQL syntax.

```
select  product, sum(X.quantity),
        sum(Y.quantity), sum(Z.quantity)
from Sales
where year=''1997''
group by product :  X, Y, Z
such that  X.month = 1,
           Y.month = 2,
           Z.month = 3
```

The novelty of this extension was the introduction of the concept of grouping variables (e.g. `X, Y, Z` above.) Grouping variables are tuple variables that range over the tuples of the currently processed group. The newly introduced `such that` clause acts as a `where` clause for these variables: there exists one condition for each grouping variable. These defining conditions may involve constants, aggregates of the group, or aggregates of previously defined grouping variables. In the above mentioned example, we define *for each* product's sales, X to be January sales, Y to be February sales and Z to be March sales. At the end we compute the total quantity of the selected tuples for X, Y and Z.

2.2. Extended Multi-Feature Queries

We now slightly extend the previous syntax. Once again the user can define *for each* group several grouping vari-

ables. These are declared in the `group by` clause as before, separated by the grouping attributes with a *semicolon*. However, the scope of these grouping variables is not any longer the group, but the *entire relation*. The assumption that grouping variables are subsets of the group is dropped. As a result, the defining conditions of the grouping variables may involve one or more of the grouping attributes, which can be considered as constants for the currently processed group [4]. Formally, the syntactic extensions are:

- **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*. For example, we may write:

  `group by product ;` $X_1, X_2, \ldots, X_n$

- **Such that clause.** This clause *defines* the range of the grouping variables mentioned in the group by clause. (The `such that` clause is similar to a `where` clause for the grouping variables.) It has the following form:

  `such that` C_1 , C_2 , ... , C_n

  Each C_i is a (potentially complex) condition that is used to define X_i grouping variable, $i = 1, 2, \ldots, n$. It may involve (i) attributes of X_i, (ii) constants, (iii) grouping attributes, (iv) aggregates of the group and (v) aggregates of the $X_1, \ldots, X_{i-1}$ grouping variables. Part (v) means that aggregates of previously defined grouping variables can be used to define subsequent grouping variables.

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

- **Having clause.** The having clause is extended to contain aggregates of the grouping variables.

The group itself can be considered as another grouping variable, denoted as X_0. Aggregates of the group are considered as aggregates of the X_0 grouping variable.

Definition 2.1: The $output$ of a grouping variable X, denoted as *outp(X)*, is the set of the aggregates of X that appear either in the `such that` clause, the `select` clause, or the `having` clause. □

[4]The grouping attributes have a *unique* value within a group. As a result, grouping attributes can be considered as constants for the `such that` clause, similar to the aggregates of the group. This is why we can extend multi-feature syntax. Otherwise the such that conditions would not be able to translate to simple selections over the relation. Rather, they would express joins.

Our syntax provides SQL with an implicit iterator over the values of a set of attributes. As a result many complex data mining queries can be expressed in a succinct way. The following examples show the usability of the proposed syntax.

Example 2.1: Being able to express data mining queries involving trends *declaratively* is an important aspect of a query language. Consider Query Q2. This can be expressed via the EMF syntax as:

```
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 p and *for each* month m we define two sets of tuples, one that contains the sales of that product p prior to month m and another one that contains the sales of product p following month m. Then we find the average quantity of each set. In the above syntax, X and Y represent these two sets.

Query Q3 exhibits two levels of aggregation, i.e. aggregated values at a first level are used at a second level of aggregation, something common in data mining queries. This can be expressed easily with the EMF syntax, due to the succesive declaration of grouping variables.

```
select product,month,count(X.*),count(Y.*)
from Sales
where year="1997" and month>1 and month<12
group by product, month; X , Y
such that
     X.product=product and X.month=month−1
     and X.quantity>avg(quantity),
     Y.product=product and Y.month=month+1
     and Y.quantity>avg(quantity)□
```

Example 2.2: Hierarchical aggregation is a frequent operation in data mining queries. Extended multi-feature syntax expresses this concept very naturally. Consider Query Q4. Its extended syntax is given below[5].

```
select  product, month, year,
        sum(X.quantity)/sum(Y.quantity)
from Sales
group by product,month,year ; X,Y
such that X.product=product and X.month=month
          and X.year=year,
          Y.product=product and Y.year=year
```

[5]This query can be expressed using just Y grouping variable. Since X represents the entire group, sum(X.quantity) could be replaced by sum(quantity).

Query Q5 is similar to this query. One would define a third grouping variable Z to denote the all-product year-long sales (`Z.year = year`) and s/he would use the `avg(Z.quantity)` appropriately in the defining condition of X to restrict X's tuples further. □

2.3. Semantics

Assume that S is the set of the grouping attributes and D is their domain. The idea is simple: for each $x \in D$ we define n relations $X_1^x, \ldots, X_n^x$ corresponding to the grouping variables $X_1, \ldots, X_n$ and $n+1$ singleton (single row) relations $F_0^x, \ldots, F_n^x$ corresponding to the aggregates of the group and the grouping variables. Then we take the projection over the join of these $2n+1$ relations according to the attributes in the `select` clause. Formally, the semantics in relational algebra are defined as follows.

Definition 2.2: Assume that S is the set of the grouping attributes, D is the domain of the grouping attributes, X_i, $i = 1, \ldots, n$ are the grouping variables, C_i, $i = 1, \ldots, n$ are the defining conditions in the `such that` clause and R is the base relation.

1. For each value $x \in D$, define:
   - $G^x = \sigma_{S=x}(R)$, (i.e. the current group)
   - $F_0^x = \mathcal{F}_l(G^x)$ where $\mathcal{F}$ is some aggregate operator (e.g. [10]) and l the list of aggregates to be computed for X_0 (i.e. $outp(X_0)$),
   - $X_i^x = \sigma_{C_i}(R \bowtie F_0^x \bowtie \ldots \bowtie F_{i-1}^x)$, $i = 1, \ldots, n$, where F_i^x denotes the aggregates of X_i^x to be computed,
   - $F_i^x = \mathcal{F}_l(X_i^x)$ where $\mathcal{F}$ is some aggregate operator and l the list of aggregates to be computed for X_i (i.e. $outp(X_i)$), $i = 1, \ldots, n$.
2. The result is given by:

$$\bigcup_{x \in D} \pi_s(F_0^x \bowtie \ldots \bowtie F_n^x \bowtie X_1^x \bowtie \ldots \bowtie X_n^x),$$ where s is the attributes list mentioned in the `select` clause. □

Although the relational algebra semantics are complicated, there is a direct mapping to a simple evaluation algorithm as it is shown in Section 3.

Note that the result may have many duplicate rows due to the join in the last step of Definition 2.2. This topic is discussed in length in [6]. One solution to this problem is to get the join only of the grouping variables (X_i^x's) and/or the aggregates (F_i^x's) mentioned in the `select` clause.

Another issue concerns empty grouping variables. According to the previous definition, if a grouping variable of a group is empty, there is no row(s) in the output for that group. In most of the cases however, users prefer a NULL value in some column rather than a non-existent row (e.g. pivoting, Query Q1). The solution to this problem is to replace joins in Step 2 of Definition 2.2 by outer joins. This issue is also discussed in [6].

3. Evaluation Algorithm

Many complex data analysis queries can be expressed in extended multi-feature syntax significantly easier than in standard SQL. However, the main reason to extend SQL is optimizability. The structure of an extended multi-feature query is such that its evaluation can be mapped directly to an efficient implementation, easy to optimize. A compact data structure is used for this purpose, corresponding most of the times to the output of the EMF query. This is called the *mf-structure* or *mf-table* of the EMF query. The basic evaluation algorithm uses only scans of the base relation in order to compute the answer. This implementation is very attractive for very large data sets, as discussed in [4].

We assume that (i) the `select` clause does not mention grouping variables' attributes (e.g. `X.product`) outside of an aggregation operator, and (ii) all aggregates mentioned in the `select` or the `such that` clause are either distributive (e.g. min, max, sum, count) or algebraic (e.g. avg). Most common EMF queries satisfy these two assumptions. Furthermore, the first one can be easily dropped [6] .

Example 3.1: Consider Example 2.2 and Query Q4. The mf-structure H of this query, corresponding roughly to its output schema, is given in Figure 1(a). Initially, the algorithm scans the `Sales` relation to complete the `product, month, year` columns, i.e. to find the distinct values of the grouping attributes (Figure 1(b).) During the second scan of `Sales`, the algorithm computes the aggregate `sum(X.quantity)` of the first grouping variable X, by identifying for each tuple t of `Sales` relation the rows of H that satisfy X's defining condition in the `such that` clause with respect to t, and update them appropriately. Figure 1(c) shows a currently scanned tuple t, the row of H that satisfies X's defining condition w.r.t. t, and the update that takes place. Finally, during the third scan, Y's aggregate `sum(Y.quantity)` is computed similarly to X. Figure 1(d) shows a currently scanned tuple t, the rows of H that satisfy Y's defining condition w.r.t. t, and the updates that take place. □

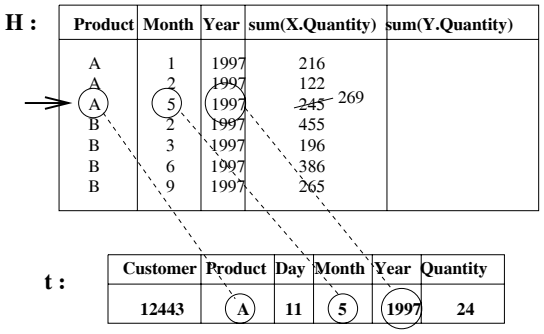
[6] The first assumption guarantees one row output per group and is used mainly for memory management purposes. Our implementation, presented later, allows grouping variables' attributes in the `select` clause. However, it keeps only one value (the last one satisfying the `such that` clause) in the final result, forcing in essence the one row per group requirement. This is sufficient in most cases.

Product	Month	Year	sum(X.Quantity)	sum(Y.Quantity)

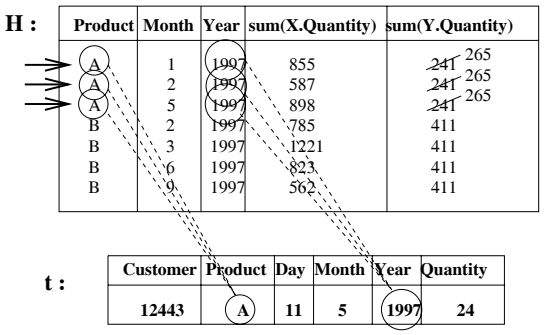
(a) mf-structure of Query Q4

Product	Month	Year	sum(X.Quantity)	sum(Y.Quantity)
A	1	1997		
A	2	1997		
A	5	1997		
B	2	1997		
B	3	1997		
B	6	1997		
B	9	1997		

(b) end of first scan



(c) during second scan



(d) during third scan

Figure 1. Several phases of evaluation of Query Q4

Definition 3.1: Let Q be an extended multi-feature query and H be a table with columns the grouping attributes and the output of each grouping variable ($outp(X_i)$, $i = 0, \ldots, n$). Then H is called the *mf-structure* (or *mf-table*) of Q. The rows of H are called *entries*. □

Each row of H corresponds to a distinct value of the grouping attributes (i.e. a group). In the following section we show that the mf-structure of a query is not implemented necessarily as a table. It can be a hash structure, or a B$^+$-tree. In the general case, it is a collection of entries. The fields of each entry consist of the grouping attributes and the output of the grouping variables.
The following algorithm computes correctly the mf-structure H of an extended multi-feature query.

Algorithm 3.1: Evaluation of extended multi-feature queries:

```
for scan sc=0 to n {
    for each tuple t on scan sc {
        for all entries of H, check if the
        defining condition of grouping var
        X_sc is satisfied.  If yes, update
        X_sc's aggregates of the entry appro-
        priately. X_0 denotes the group (the
        defining condition of X_0 is X_0.S = S,
        S denotes the grouping attributes.)
    }
}□
```

This algorithm performs $n+1$ scans on 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), the algorithm assures that X_i's aggregates will have been calculated before the j^{th} scan. Note also that given a tuple t on scan i, all entries of table H are examined, since t may belong to grouping variable X_i of several groups, as in Query Q4 : a tuple t affects several groups with respect to grouping variable Y, namely those that agree on `product,year` with t's `product, year`.

The only access method used in our algorithm is scanning (or indexed scanning). As a result, the evaluation of multiple EMF queries can overlap since the query-specific computation takes place in the individual mf-structure.

This algorithm represents an efficient, self-join free 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 concisely, but also there exists a *direct mapping*, a *tight coupling* between the representation (syntax) and an algorithm which evaluates the answer in few scans of the base data. Having used SQL, complex views and/or correlated subqueries would have to be decorrelated to conclude to such a representation/implementation, a task usually hard.

4. Optimizations

In this section we describe several immediate optimizations that improve the performance of the algorithm. The main idea is to understand and analyze the defining conditions of the grouping variables.

4.1. Relative Entries

Algorithm 3.1 can become very expensive if the mf-structure has a large number of entries since, for every scanned tuple all H's entries are examined, resulting in an implicit nested-loop join. However, this is not always necessary since, given a tuple t, one can identify a small number of mf-structure's entries that may be updated w.r.t t during the evaluation of a grouping variable X.

Example 4.1: Consider Query Q4. During the evaluation of grouping variable X (scan 1) and a tuple t, only *one* entry of Q4's mf-structure H is updated, the one that agrees on `product, month` and `year` with t. This is not the case during the evaluation of grouping variable Y (scan 2). For a scanned tuple t there is a *set* of H's entries that are updated, namely those that agree on `product, year` with t.

Definition 4.1: Assume that Q is an extended multi-feature query, H denotes its mf-structure and X is a grouping variable. The set of entries of H that are updated during the evaluation of X given a tuple t (i.e. the set of entries updated in the innermost loop body of Algorithm 3.1) is called the *relative set* of Q with respect to X and t, denoted as $Rel(X,t)$. □

In Query Q4 we know that $Rel(X,t)$ contains just one entry, the one that has `product=t.product`, `month=t.month` and `year=t.year`. The $Rel(Y,t)$ contains all the mf-structure's entries that have `product=t.product` and year=t.year.

There are cases however that we may not know a-priori the contents of a relative set. Assume that a grouping variable X is defined with the following `such that` clause : `X.product=product and X.quantity > avg(Z.quantity)`. The contents of $Rel(X,t)$ in this case will depend on the particular relation instance and the previously defined grouping variable Z.

Although we may not know precisely the contents of $Rel(X,t)$ w.r.t. a grouping variable X and a tuple t, we may be able to determine what entries will *not* be included in *Rel(X,t)*. This limits the search for relative entries within a small subset of the mf-structure, reducing the overall cost significantly. By recognizing this fact, we can keep H as a special data structure that makes searching fast (e.g. a hash table or a B^+-tree), in order to locate the relative entries fast. For example, the mf-structure of Query Q4 can be kept as a hash table on `product, year` using a hash function h. Given a tuple t one searches only the entries stored in bucket $h(t.product, t.year)$. Our prototype implementation (described in Section 5) assumes conjunctive `such that` clauses and uses a simple algorithm that checks syntactically the `such that` clauses to determine if the mf-structure of a query can be kept as a hash table on some subset of the grouping attributes.

4.2. Dependency Analysis

A scan of the base relation can be very expensive, especially if the data set is large. It is therefore essential to reduce the number of scans as much as possible. Algorithm 3.1 requires one pass over the base data for each grouping variable, for a total of $n + 1$ scans. This is not always necessary.

Example 4.2: Query Q1 can be evaluated with just one scan. During scan 0, for each scanned tuple t, the relative set w.r.t. X_0 and t consists of only one mf-structure's entry, let us denote it e, having `product = t.product`. Furthermore, note that (i) $Rel(X,t) = Rel(Y,t) = Rel(Z,t) = e$ and (ii) X_0, X, Y, Z are "independent" (no grouping variable uses aggregates of other grouping variables in its `such that` clause). As a result, *in one scan*, one can locate for each scanned tuple t the entry e (if already exists in the mf-structure, otherwise create it) and update at that point X's, Y's and Z's total `quantity`. The number of scans required for this query has been reduced from four to one. □

Definition 4.2: If a grouping variable Y contains in its `such that` clause one or more aggregates of another grouping variable X, then we say that Y *depends on* X and we write $Y \rightarrow X$. If Y's `such that` clause mentions one or more of the grouping attributes, then we say that Y *depends on* X_0 and we write $Y \rightarrow X_0$. The directed acyclic graph formed by the grouping variables' interdependencies of an EMF query is called the *emf-dependency* graph of the query. □

One can analyze *syntactically* the `such that` clauses of a query and create its emf-dependency graph G. We can find the minimal number of required scans and the grouping variables computed on each scan by sorting topologically graph G. There are cases however that this minimal number can be further reduced by evaluating two or more *dependent* grouping variables in the same scan in the event of some additional knowledge.

Example 4.3: Assume that `Sales` is sorted on `month` attribute and an extended multi-feature query Q groups by `month` and declares two grouping variables X and

Y. X is defined by "`X.month = month`" and Y is defined by "`Y.month = month+1 and Y.quantity > avg(X.quantity)`". Although Y depends on X, we *know* that X's average quantity for a group will be ready `before` the first Y tuple is processed for this group, since `Sales` is sorted on `month`. As a result, both X and Y can be computed in the same scan. □

A set of similar optimizations in the context of tape-resident data warehouses and temporal EMF queries appear in [4]. We are currently investigating additional cases of dependent grouping variables that can be evaluated in the same scan.

One can think of the optimizations of this section as a form of multiple query analysis [21]. In standard SQL, grouping variables would correspond to individual query blocks (if views were used) and a standard query processor would evaluate them independently, one after the other. We could say that multiple query analysis in extended multi-feature queries reduces to dependency analysis.

4.3. Parallel Search of the mf-structure

Besides the previous optimization methods, there are other ways to reduce the mf-structure's searching cost. One can parallelize this search by physically partitioning the mf-structure to m independent processing nodes $N_1, N_2, \ldots, N_m$, each equipped with sufficient memory to keep one partition of the mf-structure H. Node N_i keeps partition H_i, $i = 1, \ldots, n$. Algorithm 3.1 proceeds as before but upon scanning of a tuple t, this tuple is distributed to *all* m nodes and the search is carried out *locally* on each node, i.e. Algorithm 3.1 executes within each node. When all scans have been completed, the answer of the query is represented by the union of the individual outputs. Note that the partitioning of the mf-structure is not necessarily hash-based. Figure 2 shows the architecture described in this section.

Reducing the cost of the mf-structure's search is only one of the benefits of this optimization. It may be the case that the mf-structure of a query is too large to fit in one system's memory. With this technique the mf-structure can be partitioned to fit to several systems' memory in a clean, transparent way with no communication costs among the nodes.

4.4. More Scans, Less Memory

So far, we have assumed that the mf-structure fits in main memory. This is not always the case, however. We have EMF queries that their mf-structures grow too large to fit in memory. One solution is to partition the mf-structure to several processing nodes, as discussed in the previous section. Another solution is to compute the answer in m "rounds" instead of one, using only one node. During each "round", only part of the mf-structure is computed by Algorithm 3.1.

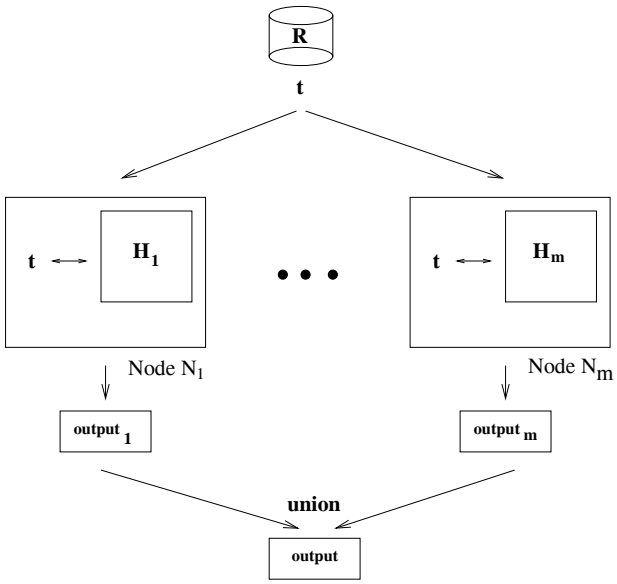


Figure 2. Parallel search of the mf-structure

This technique is conceptually the same to the method of the previous section, but it does not assume the presence of m machines and physical partitioning of the mf-structure. The answer is computed by r applications of Algorithm 3.1 (rounds). During scan 0 (in which the mf-structure is created) of round i, partition i of the mf-structure is built into memory (e.g. according to some hash function on the grouping attributes, or some existing index). The remaining scans proceed as before, considering *only the current* partition in memory. Upon completion of round i, the answer for partition i has been computed, $i = 1, \ldots, m$.

4.5. Decorrelation

In several EMF queries, the output of certain grouping variables is the same in many entries of the mf-structure. For example, consider Query Q4 and grouping variable Y. The relative set of Q4 w.r.t. Y and a tuple t consists of up to twelve mf-structure's entries (one for each month), those that agree on t's product and year. However, the output of Y on all these entries will be the same, the total quantity of t's product on t's year. It seems that we repeat the computation of a product's yearly sum twelve times. One could decorrelate the computation of this grouping variable by creating a separate mf-structure.

However, the benefits of this optimization have to be assessed carefully for several reasons. The main idea is not as simple as before and Algorithm 3.1 has to change in

order to incorporate joins of mf-structures at a later stage, resembling traditional query processing. Certain optimizations, such as those described in Sections 4.3 and 4.4, are not applicable any longer. In general, the intuitiveness of Algorithm 3.1 is lost. We are currently investigating decorrelation methods and query patterns and try to understand how valuable such an optimization would be.

5. Implementation

We have implemented extended multi-feature queries on top of a leading commercial datbase system, running on SGI platform. One can enter an EMF query through a simple text interface. Our system generates automatically a program written in the procedural language of the DBMS, implementing Algorithm 3.1 and incorporating two basic optimizations: (i) simple relative row analysis to determine if the mf-structure can be represented as a hash table on any subset of the grouping attributes, and (ii) simple dependency analysis on grouping variables to determine the minimal number of scans required to compute the answer. These two optimizations are reflected in the generated program.

Below we compare the performance of the SQL component of the commercial database system versus our system's performance on Queries Q1 and Q2. Similar results hold for Queries Q3, to Q6 [7]. During the experiments we were the only users of the system. Time measurements are elapsed times and each measurement is the average of several runs. The goal is not to prove that our system is better than some commerical DBMS(s), but to emphasize that traditional optimizers do not consider the special structure of extended multi-feature queries.

The performance of Query Q1 is shown in figure 3. The size of each group is 6 tuples. Our implementation generates a program that computes the answer in one scan. The mf-structure is represented as a hash table on `product`. "SQL-A" and "SQL-B" correspond to two different standard SQL representations (views, self-joins) of Query Q1, evaluated by the SQL component of the DBMS. The performance of our system is marked as "Implementation". Our method uses the procedural language of the DBMS and this incurs a huge overhead in terms of scanning and processing. The line marked "Flat Files" shows what the performance of our method would be if Algorithm 3.1 has been implemented *in* the server, instead of on top of it. In order to get this measurement we used a different version of our implementation, operating on flat files and generating C programs incorporating Algorithm 3.1. Description and usage of this implementation appears in [3].

The performance of Query Q2 is shown in figure 4. The size of each group is 12 tuples. The generated program computes the answer in two scans and the mf-structure is represented as a hash table on `product`. "SQL-A", "Implementation" and "Flat Files" represent measurements similar to those described above.

[7] We also experimented with a different commercial DBMS and its performance was comparable to the results presented here.

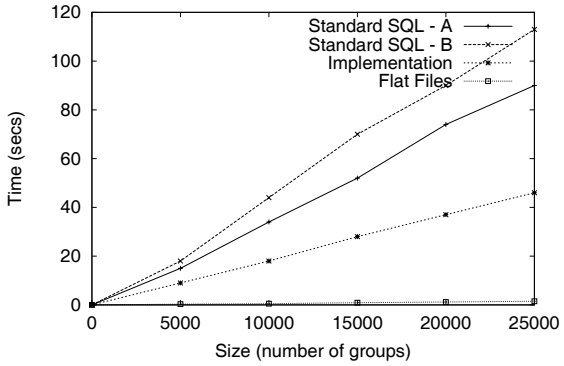


Figure 3. Performance of Query Q1 using standard SQL vs EMF

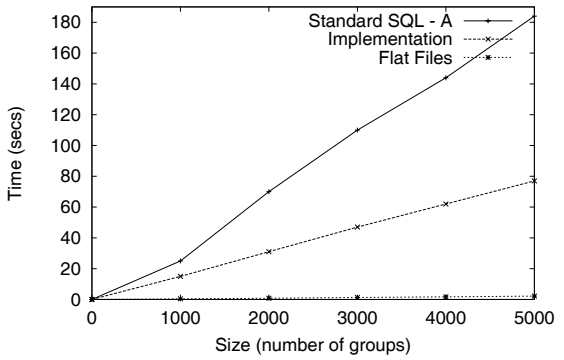


Figure 4. Performance of Query Q2 using standard SQL vs EMF

6. Conclusions

Many authors have argued in the past that SQL has to be extended in order to handle complex decision support queries adequately [14]. Few commercial systems offer SQL extensions to express complex aggregate queries easily[16]. Furthermore, query optimization of ad hoc data analysis has been given very little attention in the database research literature. We try to address all these issues.

In this paper we discuss a class of data analysis queries called extended multi-feature queries. This class contains many useful, practical ad hoc OLAP queries not easily

expressed with standard SQL. We propose a simple and intuitive syntactic extension of SQL in order to express EMF queries. It equips SQL with a looping construct in a declarative way. We argue that such an extension is imperative to express and evaluate ad hoc OLAP queries. A simple evaluation algorithm and several optimizations are presented, showing the flexibility of our implementation.

Current work involves implementation and further research of the optimization techniques presented in Section 4. We also examine optimization techniques of temporal EMF queries in the context of tape-resident data warehouses [4]. We investigate natural additions to the EMF syntax in order to handle (i) multiple base relations (i.e. the user can select "features" from several base tables), and (ii) nested aggregation (e.g. for each customer, find the total sales for each month and output the maximum of these totals). It seems that both cases can be evaluated by an algorithm very similar to 3.1 and similar optimizations apply [13]. Finally, we study a new relational operator called MD-join (multi-dimensional join) along with a set of algebraic transformations in order to integrate the optimization techniques proposed here in a standard relational system [5].

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. Chatziantoniou. Introducing the PanQ Tool for Complex Data Management. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD) (to appear)*, 1999.

[4] D. Chatziantoniou and T. Johnson. Decision Support Queries on a Tape-Resident Data Warehouse. *IEEE Computer (to appear)*.

[5] D. Chatziantoniou, T. Johnson, and S. Kim. On Modeling and Processing Decision Support Queries. Submitted for publication, February 1999.

[6] D. Chatziantoniou and K. Ross. Querying Multiple Features of Groups in Relational Databases. In *22nd VLDB Conference*, pages 295–306, 1996.

[7] D. Chatziantoniou and K. Ross. Groupwise Processing of Relational Queries. In *23rd VLDB Conference*, 1997.

[8] S. Chaudhuri and K. Shim. Including Group-by in Query Optimization. In *VLDB Conference*, pages 354–366, 1994.

[9] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *TKDE*, 2(1):44–62, 1990.

[10] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, 1989.

[11] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[12] 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.

[13] T. Johnson and D. Chatziantoniou. Extending Complex Ad Hoc OLAP. Submitted for publication, February 1999.

[14] R. Kimball and K. Strehlo. Why Decision Support Fails and How to Fix it. *SIGMOD RECORD*, 24(3):92–97, 1995.

[15] R. Meo, G. Psaila, and S. Ceri. A Tightly-Coupled Architecture for Data Mining. In *IEEE International Conference on Data Engineering*, pages 316–323, 1998.

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

[17] K. Ross and D. Srivastava. Fast Computation of Sparse Datacubes. In *23nd VLDB Conference*, pages 116–125, 1996.

[18] K. Ross, D. Srivastava, and D. Chatziantoniou. Complex Aggregation at Multiple Granularities. In *Extending Database Technology (EDBT), Valencia*, pages 263–277, 1998.

[19] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating Mining with Relational Database Systems: Alternatives and Implications. In *ACM SIGMOD, Conference on Management of Data*, pages 343–354, 1998.

[20] P. Selinger, D. Astrahan, R. Chamberlain, R. Lorie, and P. T. Access Path Selection in a Relational Database Management System. In *ACM SIGMOD, Conference on Management of Data*, pages 23–34, 1979.

[21] T. Sellis. Multiple-query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[22] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex Query Decorrelation. In *International Conference on Data Engineering*, pages 450–458, 1996.

[23] S. Subramanian and S. Venkataraman. Cost-based Optimization of Decision Support Queries using Transient Views. In *ACM SIGMOD, Conference on Management of Data*, pages 319–330, 1998.