The MD-join : An Operator for Complex OLAP

Damianos Chatziantoniou
Panakea Software Inc.
damianos@panakea.com

Michael Akinde
Dept. of Computer Science*
Aalborg University
strategy@cs.auc.dk

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

Samuel Kim
Dept. of Computer Science
Stevens Institute of Tech.
skim@cs.stevens-tech.edu

Abstract

OLAP queries (i.e. group-by or cube-by queries with aggregation) have proven to be valuable for data analysis and exploration. Many decision support applications need very complex OLAP queries, requiring a fine degree of control over both the group definition and the aggregates that are computed. For example, suppose that the user has access to a data cube whose measure attribute is Sum(Sales)*. Then the user might wish to compute the sum of sales in New York and the sum of sales in California for those data cube entries in which* Sum(Sales) $>$ \$1,000,000.

This type of complex OLAP query is often difficult to express and difficult to optimize using standard relational operators (including standard aggregation operators). In this paper we propose the MD-join *operator for complex OLAP queries. The MD-join provides a clean separation between group definition and aggregate computation, allowing great flexibility in the expression of OLAP queries. In addition, the MD-join has a simple and easily optimizable implementation, while the equivalent relational algebra expression is often complex and difficult to optimize. We present several algebraic transformations that allow relational algebra queries that include MD-joins to be optimized.*

1. Introduction

Decision support systems (DSS), on-line analytical processing (OLAP) and multi-dimensional analysis have been the focus of intense research and commercial activity the past few years. Large private and public organizations use data warehouses to store and organize information collected during normal business processes. To be of use, the data must be analyzed and mined, leading to the development of many new data analysis and mining tools.

A common OLAP query computes the aggregate of measure attributes from a fact table, grouped by one or more dimension attributes (i.e., Select $\cdots$ From R Group By $\cdots$). However, researchers have observed that providing a greater degree of control over the query greatly extends the types of analyses that can readily be performed.

One type of extension is to allow alternative definitions of the groups. Gray et al. in [GBLP96] have proposed the `Cube By` keyword and appropriate SQL extensions. Graefe, Fayyad, and Chaudhuri [GFC98] have proposed the `unpivot` operator, which allows the easy extraction of marginal distributions from a database for use as input to decision tree algorithms. The SQL99 standard [Cha96, SQL99] defines *grouping sets*, which compute a user-controlled collection of rollups (instead of all rollups, as in the Cube By keyword). The EMF-SQL language [Cha99] allows the user to specify aggregation independently of the group specification.

Another type of extension is to allow the user to specify aggregate functions more complex than the standard set of count, sum, average, min and max (for example, most frequent, median, moving average, and so on). The SQL99 standard [SQL99] provides a large collection of new keywords for expressing new aggregate functions. Chatziantoniou and Ross in [CR96] and Chatziantoniou in [Cha99] proposed extensions to SQL that allow the user succinctly express customized aggregation conditions. Another approach is to allow the user to write User Defined Aggregate Functions (UDAFs) [JM98, Cha96, Ill95, WZ00a, WZ00b]. A user defined aggregate function requires the user to specify the resources that must be allocated for the aggregate, and callbacks that initialize the aggregate, add a value to the aggregate and report the aggregate's value. Wang and Zaniolo [WZ00a] also support an *early return* callback. The AXL system [WZ00b] uses relational tables instead of scratchpads. Johnson and Chatziantoniou [JC99] allow the user to specify a restricted scratchpad, but declaratively instead of through a UDAF. Queries involving UDAFs can be difficult to optimize because the UDAF's behavior is unknown. The user can describe additional properties and callbacks of the UDAF to allow cost-based optimization and parallelization [JM98, SM96].

In this paper, we show how complex group specification can be extended and unified with complex aggregate specification. We propose a new operator, the *MD-join* that aggregates a relation R to a base values table B. The base values table B as well as the relation R can be the result of a relational algebra expression (which can include MD-join operations), allowing tremendous flexibility in expressing decision support queries. We show several algebraic transformations of relational expressions involving MD-joins. These transformations allow for a wide variety of optimizations, permitting automatic parallelization and ensuring scalability. For example, one algebraic transformation can express the cube computation optimization algorithms of [AAD$^+$96, RS96], and generalize their application within a cost-based optimizer.

The paper proceeds as follows: In Section 2 we examine several decision support examples and discuss briefly how all these seemingly different queries can be seen as instances of the same framework. In Section 3 we discuss how this framework is set up. We argue that decision support queries have to be seen as a two-phase process and give one relational operator (coupled with a simple generic algorithm) to handle the aggregation phase. In Section 4 we describe generic optimization techniques for the proposed operator and briefly show that in fact, many of the known algorithms are simply subcases of our algorithm. A general discussion on syntactic issues, performance, related and future work follows in Section 5. The following table is used by the examples in the rest of the paper:

```
Sales(cust,prod,day,month,year,state,sale)
```

2 Motivation

We have observed that decision support and data mining applications often need a fine-grained control over the *base values* (i.e., groups) used to define the aggregation, as well as a fine-grained control over what aggregated values are computed. Let us consider a series of examples to illustrate this idea.

prod	month	state	sum(sale)
12	3	NY	1292
82	8	CA	2198
ALL	2	NJ	22199
22	ALL	IL	12911
44	3	ALL	81922
ALL	ALL	NY	213911
ALL	4	ALL	651202
13	ALL	ALL	981221
15	ALL	ALL	856765
ALL	ALL	ALL	9102992

Figure 1. Output tables of Example 2.1.

Example 2.1: One may be interested to compute the total sales broken down by all possible combinations of attributes `prod, month, state` (i.e. eight group bys). Gray et al. proposed in [GBLP96] the cube by operator and extended appropriately SQL. The query is then formulated as:

```
    Select prod, month, state, sum(sale)
    From Sales
    Cube By prod, month, state
```

A subset of the output table is shown in Figure 1. A thick line partitions the output table in two parts, the cube by attribute values and the computed total. We merge the eight separate group-by tables into a single table by using 'ALL' values. We defer the definition of 'ALL' until later, but its meaning mirrors the modeling of subcubes of a base cube (as in [GBLP96]). That is, (`prod,ALL,state`) is a rollup of the `prod, month, state` by dimension `month`. In 1(a), the row with cube-by attributes (`44, 3, ALL`) represents all sales of product 44 during month 3, over all states.

The user might not be interested in all the entire cube, but rather a restricted collection of group-bys. For example, Graefe, Chaudhuri and Fayyad in [GFC98] discuss the "unpivot" operator, for use in the context of data mining. These authors observed that most decision tree computation algorithms use as input data the marginal densities of the input data rather than the entire data set. The Grouping Sets keyword [Cha96] allows the following query to request the marginals:

```
Select prod, month, state, sum(sale)
From Sales
Group By Grouping Sets ((prod), (month),
                        (state))  █
```

Example 2.2: Consider now a *pivoting* example, which does not classify as a data cube or multi-dimensional query. Suppose that we want to compute for each customer the average sale in "NY", in "NJ" and in "CT" (the tri-state area). Figure 1 shows part of the output table. Note once again the thick line that separates the grouping attribute values and the aggregated values.

This type of query is cumbersome to express in SQL because the definition of aggregation is tied to the definition of the groups. Three subqueries are required to compute per-customer sales in NY, NJ, and CT, respectively. Our intended collection of groups is the list of all customers, whether or not they made any purchases in NY, NJ, or CT. Therefore a fourth subquery is required to select all unique customers, followed by four outer joins to attach the sales to the customer in NY, NJ, and CT. ▮

Example 2.3: Suppose that we want to count how many sales were above the average sale, when table `Sales` is broken down by all possible combinations of `prod, month, state`. The last sentence implies a data cube structure over `prod, month` and `year` attribute; however, we want to compute something more complex than a simple aggregate. Even if the `cube by` syntax is available in our version of SQL, it can not be used. Rather, the user has to define eight group bys, join each one with the `Sales` table and perform eight new group bys.

In [RSC98], Ross, Srivastava and Chatziantoniou argued that the multi-featured aggregation and the cube by syntax should be merged for succinctness and performance reasons. However, we would like to generalize this idea by separating the definition of the groups and the definition of the (multi-pass) aggregation. Instead of proposing special purpose evaluation algorithms, we would like to use query transformations and a cost-based optimizer to find an optimal evaluation plan. ▮

Example 2.4: Suppose that we are interested in computing the total sale at certain points of a data cube, given to us in a precomputed datafile or table. For example, a data mining algorithm may be interested in the total sale only at these crucial/representative points. A generic decision support framework should be able to handle such queries in a clean and succinct way. This can be achieved by separating the specification method of the left and the right part of the output table. ▮

Example 2.5: Suppose that for each product you want to count for each month of 1997 the sales that were between previous month's average sale and following month's average sale.

Computing the answer to this query requires that for each product and month, we compute aggregates from tuples outside the group (the next and previous month's average sales). After these values are computed, we have enough information to compute the output aggregate. That is, multiple pass aggregation is required. ▮

The list of examples can continue (e.g., using computed values in the base values, for example to aggregate by quarter instead of month), but we stop here to analyze the similarities and differences between the example queries. Although each of the queries seems to have a different flavor and a different evaluation strategy, they all involve the same approach: defining a set of base values which define the rows of the output table, associating subsets of a relation with each row, and computing aggregates of the subset. Both the definition of the base values and the computation of the associated aggregates can involve complex computations. This observation argues that the definition of the base values and the computation of the aggregated values should be decoupled. By decoupling these two definitions, we can achieve not only a greater flexibility in defining the queries, but also a greater succinctness and simplicity in expressing them.

Example 2.1 is a classical data cube example. Several algorithms have been proposed for fast data cube computations [AAD$^+$96, RS96]. In these formulations aggregation is considered part of the data cube structure. As a result, if one computes the total sale for some data cube and later wants to compute the average sale, s/he must re-execute one of the proposed algorithms. Example 2.3 shows that although the base values structure is a data cube, the `cube by` clause can not be used, due to the fact that the semantics of this operator are tied to the computation. Although the proposed solution in [RSC98] solves nicely such cases, it still does not distinguish between base values and aggregation. Note that modeling data cubes as in [AGS97] does not help much, since we must join the data cube with the detail data, an operation not supported in [AGS97].

Chatziantoniou and Ross introduced the concept of grouping variable [CR96, Cha99], a useful idea for complex and ad hoc computations (Examples 2.2 and 2.5 can be expresed simply using grouping variables). Still, this idea is restricted to group by queries.

In this paper we propose a clean cut between the base values set-up phase and the aggregation phase. We need only one operator to carry out the aggregation, no matter what are the base values. Furthermore, we want to be able to combine several of these operators to represent complex ad hoc computations. We show that this formulation leads to better performance, optimization and expressibility of decision support queries. Other useful OLAP relational operators or SQL extensions are proposed in [KS95, GL97, CT97]. Gyssens and Lakshmanan in particular [GL97] recognize the need to separate group definition and aggregation.

3 The MD-Join Operator

In this section, we define the MD-join operator, which cleanly separates the definition of the groups from the definition of the aggregation. The MD-join does not group tuples for aggregation in the way that conventional aggregation does. Instead, the collection of output tuples is defined by a *base values relation,* which contains a collection of "group" keys. We will generally refer to base values instead of groups from now on.

Definition 3.1: Let $B(\mathbf{B})$ and $R(\mathbf{R})$ be relations, θ a condition involving attributes in $\mathbf{B}$ and $\mathbf{R}$, and l a list of aggregate functions $(f_1, f_2, \ldots, f_n)$ over attributes $c_1, c_2, \ldots, c_n$ in $\mathbf{R}$. Then the MD-join, $MD((, R, B, ,), l, \theta)$, is a relation with schema $\mathbf{B}, f_1\_R\_c_1, ..., f_n\_R\_c_n$,[1] whose instance is determined as follows. Each tuple $b \in B$ contributes to an output tuple $\mathbf{b}$, such that:

- $\mathbf{b}[A] = b[A]$, for every attribute $A \in \mathbf{B}$
- for each tuple b of relation B let $RNG(b, R, \theta) = \{r \in R | \theta(b, r) \text{ is true } \}$. Then, the value of attribute $f_i\_R\_c_i$ of tuple $\mathbf{b}$ is given by $\mathbf{b}[f_i\_R\_c_i] = f_i\{\{t[c_i] \mid t \in RNG(b, R, \theta)\}\}$, where $\{\{\cdots\}\}$ denotes multiset.

B is called the *base-values relation* and R is called the *detail relation*. ▮

Following Gray et al [GBLP96], we use the value "ALL" in the base-values table To model multiple granularity aggregates within a single table (as discussed in Example 2.1).

Note that if every row in B is distinct, then the above definition corresponds to the relational algebra expression:

$$\begin{gathered} {}_b\mathcal{F}_{f_1(R.c_1),\ldots,f_n(R.c_n)}(\mathrm{B} \bowtie_\theta \mathrm{R}) \\ \cup ((B - \pi_b(B \bowtie_\theta R)) \times \mathrm{N}_l) \end{gathered}$$

where $\mathcal{F}$ is an aggregation operator (e.g. [EN89]), b denotes all the attributes of B and N is a one tuple relation whose fields are the initial values for the aggregates in l.

The definition of the MD-join operator allows the user a tremendous amount of flexibility in defining an aggregation query, as B and R can be arbitrary relational expressions and θ can be an arbitrary join predicate. For example, to compute in query of Example 2.5 for each customer and month the average of sales of this customer in the previous month, one defines θ as: `Sales.cust=cust and Sales.month = month+1`.

Note that the row count of the result of the MD-join is the same as the row count of B (i.e., the MD-join performs an outer join). This semantics more accurately captures the user's intentions than the standard aggregation does (consider Example 2.2).

Note also that the MD-join operator can be considered as a shortcut for a somewhat more complex expression. However, the expression that the MD-join represents often occurs in OLAP queries. By understanding the properties of the operator, we can easily obtain many query transformations leading to efficient evaluation plans, as will be shown in Section 4.

We give below a simple algorithm to implement MD-join[2]. Note that although definition 3.1 states that for each row b of B we identify a set of tuples of R, in this algorithm we follow the opposite direction: we scan the detail table R and loop over all tuples of B to identify matches based on condition θ. If a match is detected, we update the aggregate columns appropriately.

Algorithm 3.1: Evaluation of the MD-join operator:

```
scan R, and for all tuples $t$ in $R${
   for all rows $r$ of $B$, check if condition
   $\theta$ is satisfied with respect to $r$ and $t$.
   If yes, update $r$'s aggregate columns
   appropriately.
} ▮
```

The conventional group-wise aggregation algorithm cannot in general be applied, because the tuples in B do not necessarily represent groups in R. Neither can a conventional hash aggregation algorithm be applied, as a tuple from R might join with many tuples from B. However, by using the appropriate transformations and optimizations one can develop efficient evaluation plans the generalize the group-wise and the hash aggregation algorithms.

3.1 Complex Ad Hoc Computations

Many decision support queries ask for something more complex than a simple average or total. Examples 2.2, 2.3 and 2.5 are instances of queries requiring complex ad-hoc computations. More examples can be found in [Cha99]. One of the primary motivations for separating grouping and computation is the ability to define complex ad hoc computations without needing to define a new operator for each query.

The semantics of the MD-join operator has been defined in such a way that several MD-joins can be combined in a sequence to carry out most ad hoc complex computations.

Example 3.1: Example 2.2 can be expressed in relational algebra using MD-joins as [3]:

$$\begin{gathered} MD(MD(MD(B, Sales, avg(sale), \theta_1), \\ Sales, avg(sale), \theta_2), Sales, avg(sale), \theta_3) \end{gathered}$$

[1]Attributes are appropriately renamed if there are any duplicate names generated this way.

[2]This algorithm works only for distributive and algebraic aggregates; holistic aggregates can be processed by a similar algorithm that handles memory allocation issues (e.g., see [WZ00b]). However, some holistic aggregates can be made algebraic by using approximation, e.g. approximate medians[MRL98].

[3]We omit for simplicity several relational algebra details in this formulation. Each application of the MD-join should be preceded by renaming of the `Sales` table. The same holds for the remaining examples.

where θ_1 : Sales.cust=cust and Sales.state="NY",
θ_2 : Sales.cust=cust and Sales.state="CT",
θ_3 : Sales.cust=cust and Sales.state="NJ",
and B is the resulting table of a `select distinct cust from Sales`.

The advantage of this formulation is that we do not have to define a new class of queries. It is still a group by (to construct the base-values table), followed by a complex computation. If the `cube by` keyword in this query is replaced by, e.g. `unpivot`, the algebraic expression is unchanged (except for the definition of B). ▮

Example 3.2: Example 2.3 can be expressed in relational algebra using MD-joins as:

$$MD(MD(B, Sales, avg(sale), \theta_1)\ , Sales, count(*), \theta_2)$$

where

θ_1 : Sales.prod=prod and Sales.month=month and Sales.state=state,
θ_2 : Sales.prod=prod and Sales.month=month and Sales.state=state andSales.sale>avg_Sales_sale,

and B is the data cube of `prod, month` and `state` attributes.

Once again this is a data cube query with some complex computation. It is not necessary to define a whole new class of queries. ▮

Example 3.3: A complex operation may involve *different* detail tables. Let us assume that there is another table, called `Payments` with schema `(cust,day,month,year,amount)` and a user wants to know the total sales and payments for each customer and month. This query can be expressed as:

$$MD(MD(B, Sales, sum(sale), \theta_1)$$
$$Payments, sum(amount), \theta_2))$$

where
θ_1 : Sales.cust=cust and Sales.month=month,
θ_2 : Payments.cust=cust and Payments.month=month,
and B is the resulting table of a `select distinct cust,month from Sales`.

Such a formulation can be optimized significantly better than a traditional relational algebra expression, due to its conciseness. ▮

4 Optimizations

In this section we discuss how the MD-join operator interacts with other relational operators and present several algebraic transformations useful for generating optimized query plans. Thus we show that the MD-join contributes to the efficient evaluation of decision support queries as well as to their succinct expression. The MD-join can be incorporated immediately into present cost- and algebraic-based query optimizers. We also show that the implementation of the MD-join is efficient and offers orders of magnitude speedups as compared to current commercial systems. We briefly discuss performance in Section 5.

4.1 In-Memory Computation and Parallelism

In this section, we present how the MD-join and union operators interact. Specifically, we show the base-values table partitioning transformation, and show its uses in developing query plans for large-scale computations and for parallel query evaluation. We also show how MD-join can employ intra-operator parallelism.

Theorem 4.1: If B and R are relations, $B_1, B_2, \ldots, B_m$ a partition of B, l a list of aggregate functions over columns of R and θ a condition involving attributes of B and R, then:

$$MD(B, R, l, \theta) = MD(B_1, R, l, \theta) \bigcup \ldots$$
$$\ldots \bigcup MD(B_m, R, l, \theta) \square$$

4.1.1 In-Memory Computation

One interpretation of Theorem 4.1 is that the base-values table B can be partitioned in any arbitrary way and the MD-join can be computed in m scans of the detail table R instead of one. As a result, we can always devise a query evaluation plan in which Algorithm 3.1 operates on memory-resident data. Regardless of the indices constructed on B for use in the MD-join (discussed in Section 4.5), in-memory evaluation will usually be significantly faster than an out-of-core evaluation. The cost is a well-defined increase in the number of scans of R.

4.1.2 Parallelism

A different application of Theorem 4.1 is some form of intra-operator parallelism. That is, the partitions B_1 through B_m are distributed to m processors and each MD-join fragment evaluated locally. Each evaluation requires a scan through R, but we will show how the work requried to perform the evaluations can be reduced.

4.2 MD-join and Selections

The MD-join operator interacts with selections in a similar fashion to joins.

Theorem 4.2: Let B and R be relations, l be a list of aggregate functions over columns of R and θ be a condition involving attributes of B and R. If $\theta = \theta_1 \; and \; \theta_2$ and θ_2 involves only attributes of R, then:

$$MD(B, R, l, \theta_1 \; and \; \theta_2) = MD(B, \sigma_{\theta_2}(R), l, \theta_1)$$

Proof: Tuples of R not satisfying θ_2 will not be considered by the MD-join and as a result this selection can be pushed to R. ▮

This theorem says that the MD-join operator can be implemented via an indexed instead of a full scan of R. This is a very important optimization, especially if R is very large (e.g. internet/web/call logs).

Example 4.1: One may be interested to compare for each product the total sales of the period 1994 to 1996 versus the total sales of 1999. This can be expressed in relational algebra using MD-joins as:

$$MD(MD(B, Sales, sum(sale), \theta_1), Sales, sum(sale), \theta_2)$$

where

θ_1 : Sales.prod=prod and Sales.year$\geq$1994 and Sales.year$\leq$1996,
θ_2 : Sales.prod=prod and Sales.year=1999

and B is the resulting table of a `select distinct prod from Sales`.

If there is a clustered index on the date `(day, month, year)` set of attributes, there is no need to perform two full scans of the `Sales` relation. Instead, the first scan will involve sales of 1994 to 1996 and the second scan will involve sales of 1999. ▮

We note that Theorem 4.2 can work in conjunction with Theorem 4.1 to limit the range of R that must be MD-joined with B.

Observation 4.1: Let us consider an MD-join $MD(B, R, l, \theta)$ in which B does not fit in memory and is partitioned into $B_1 \bigcup \ldots \bigcup B_m$, where $B_i = \sigma_i(B)$, σ_i is a range selection on a set S of attributes of B, $i = 1, 2, \ldots, m$. Further assume that the condition θ can be expressed as a conjunction of an equality test on attributes of S between B and R and another condition θ', i.e. θ : `B.S=R.S' and` θ'. Then,

$$MD(\sigma_i(B), R, l, \theta) = MD(\sigma_i(B), \sigma'_i(R), l, \theta)$$

where σ'_i is σ_i with references to attributes S of B replaced by the corresponding references to attributes S' of R. ▮

Observation 4.1 states that a range selection on the base-value relation B can be pushed to the detail relation R when the theta condition of the MD-join is appropriate. This optimization is very similar to optimization techniques of hash-based aggregation methods of [Gra93]. However, the framework is more general since the theta condition is not necessarily an equijoin.

One application of Observation 4.1 is to reduce amount of data scanned when applying the in-memory optimization in Section 4.1.1. For an example, consider the query in Section 4.1.2. If we range partition the base-value relation on `month` (e.g. 1–3, 4–8, 9–12), we can push these range selections to the detail relation. If there is a clustering index on `month` we can reduce the work required to evaluate aggregates of x and y to scanning only appropriate partitions of the `Sales` relation (i.e., do group-wise processing). A similar optimization can be made when performing the parallelization optimization discussed in Section 4.1.2, except that the effect is to reduce the number of processors that receive each tuple.

4.3 Series of MD-joins

Another set of important algebraic transformations applies when there is a series of MD-join operators, a very common case in decision support queries (almost all of our examples involve series of MD-join operators).

A scan of a 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. If there is a series of k consecutive MD-joins, Algorithm 3.1 requires one pass over the detail relation(s) for each operator, for a total of k scans. However, this is not always necessary.

Consider Example 2.2. Although this query is expressed as a series of three MD-joins, obviously it can be evaluated with just one scan. The key observation in this example is that the theta conditions of the MD-joins are “independent” of each other. More precisely, there are no augmented columns generated by an MD-join used in a theta condition of a subsequent MD-join.

Dependency analysis between correlated aggregates has been treated in [Cha99] and many ideas are common. However it is valuable to formulate this concept as an algebraic transformation. One approach is to generalize the MD-join operator to incorporate a *vector* of theta conditions and a vector of lists of aggregate functions (and possibly a vector of names to rename aggregate functions). This is denoted as:

$$MD(B, R, (l_1, l_2, \ldots, l_k), (\theta_1, \theta_2, \ldots, \theta_k))$$

with the obvious semantics' extensions (for each tuple of B we define k subsets of R instead of one). Using this new generalized MD-join, two consecutive MD-join operators can be combined in one, if (i) the condition of the second MD-join does not mention any column generated by the first and (ii) the detail relation is the same for both MD-joins.

A possible impediment to using the series MD-join operator is that we might be provided with the algebraic expression in an inappropriate order. Fortunately, MD-joins commute under suitable conditions.

Theorem 4.3: If B,R_1,R_2 are relations, l_1 and l_2 are list of aggregate functions and θ_1, θ_2 conditions, then:

$$MD(MD(B, R_1, l_1, \theta_1), R_2, l_2, \theta_2) = MD(MD(B, R_2, l_2, \theta_2), R_1, l_1, \theta_1)$$

if θ_1 involves only attributes of B and R_1 and θ_2 involves only attributes of B and R_2. ▮

The last theorem gives us the ability to assign a series of k MD-joins to a minimal number of generalized MD-joins in O(k^2) worst-case time (by topologically sorting the MD-joins).

In addition to combining a sequence of MD-joins into a single operation, it is possible to express one as a join of two separate MD-joins.

Theorem 4.4: Assume that B, R_1, and R_2 are relations, l_1 and l_2 are lists of aggregate functions over columns of R_1 and R_2 respectively, and θ_1 and θ_2 are conditions involving attributes of B and respectively R_1 and R_2.

$$MD(MD(B, R_1, l_1, \theta_1), R_2, l_2, \theta_2) = MD(B, R_1, l_1, \theta_1) \bowtie_B MD(B, R_2, l_2, \theta_2) \Box$$

Because the MD-join does not change the rows of B, it is possible under suitable conditions to perform the equijoin very efficiently. An implication of the theorem is that one can move the computation of the MD-join to the source of a relation R, and in fact perform several such joins in parallel. Consider Example 2.2. Suppose that the `Sales` table is a distributed relation, and data for New Jersey is stored in Trenton, data for New York in Albany, and so on. It is likely to be more efficient to move the base-value relation to the three data stores, perform local MD-joins, then equijoin the results to obtain the answer. Note that we make use of Theorem 4.2 here.

4.4 Projections and MD-Joins

In this section we show how projections and MD-joins can be used to express the *roll-up* property of data cubes – that is, sub-cubes can be constructed from their drill-down cubes – as an algebraic transformation, and discuss how this allows us to algebraically express and generalize more specialized cube computation algorithms.

Theorem 4.5: Let S and R be two relations, X and Y be attribute sets of S. Let $\pi_{X,ALL_i}(B)$ be consist of the unique tuples of S projected to attributes in X, and with i copies of 'ALL' attached. Let l be a list of distributive aggregates, and let θ be a predicate that is a conjunction of predicates that test the equality of an attribute of X or Y with an attribute in R, for all attributes in $X \bigcup Y$. Then,

$$MD(\pi_{X,ALL_{|Y|}}(S), R, l, \theta) = MD(\pi_{X,ALL_{|Y|}}(S), MD(\pi_{X,Y}(S), R, l, \theta), l', \theta)$$

where l' is the set of distributive aggregates in l modified appropriately (e.g. a *count* in l becomes a *sum* in l'). ▮

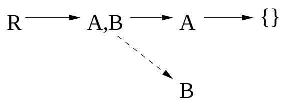

Figure 2. Pipelined paths of the PIPESORT algorithm

Theorem 4.5 simply states that a coarser granularity cuboid can be computed by a finer granularity cuboid. The efficient cube computation algorithms of [AAD$^+$96] are based largely on appropriate partitions of the data cube, on pushing selections on the base-values table partitions to the detail table partitions, and the *roll up* property of data cubes. Using Theorem 4.5 (in conjuction with the preceding theorems), we can express the algorithms of [AAD$^+$96] algebraically, and thus generalize their application.

For example, the PIPESORT algorithm of [AAD$^+$96] proceeds level-by-level in the search lattice of a data cube starting from the root, building pipelined paths according to certain cost criteria and converting the search lattice into a tree. Theorem 4.5 can be used to build pipelined paths. The cost criteria can be incorporated into the optimizer. As a result we can generalize these algorithms and apply them in new settings.

Example 4.2: Assume that R is a relation, A, B, M are attributes of R (M is the measure) and we want to compute the total of M over the datacube of attributes A and B. Suppose that by running PIPESORT on this we get the paths of Figure 2, where the dashed line means resorting.

If D denotes the datacube of A and B, then we want to compute $MD(D, R, l, \theta)$, where $l = sum(M)$ and θ: $R.A = D.A$ and $R.B = D.B$. This expands to:

$$MD(\pi_{A,B}(R) \bigcup \pi_{A,ALL}(R) \bigcup \pi_{ALL,B}(R) \bigcup \pi_{ALL,ALL}(R), R, l, \theta)$$

Applying Theorems 4.1 and 4.5 - more than once in some cases- we have:

$$\begin{array}{l} MD(\pi_{A,B}(R), R, l, \theta) \bigcup \\ MD(\pi_{A,ALL}, MD(\pi_{A,B}(R), R, l, \theta), l', \theta) \bigcup \\ MD(\pi_{ALL,B}(R), MD(\pi_{A,B}(R), R, l, \theta), l', \theta) \bigcup \\ MD(\pi_{ALL,ALL}(R), MD(\pi_{A,ALL}(R), \\ MD(\pi_{A,B}(R), R, l, \theta), l', \theta), l'', \theta) \end{array}$$

This algebraic expression can be annotated appropriately to reflect the pipelining and resorting steps corresponding to the PIPESORT output of Figure 2. Usually optimizers perform common subexpressions elimination. We note that the MD-join operator can be implemented differently than described by Algorithm 3.1. In this case, a more efficient algorithm is possible because the detail relation is provided in sorted order.

Note that using an annotated relational algebra expression to represent the cube computation, one can divert from the classical evaluation methods and build hybrid techniques. For example, the dashed line in Figure 2 could denote either resorting, or hash evaluation [Gra93]. ▮

Ross and Srivastava in [RS96] proposed a divide-and-conquer strategy for the computation of a data cube over $D_1, D_2, \ldots, D_n$ dimensions. If the detail relation R fits in main memory, then their algorithm performs multiple in-memory sorts computing the cuboids, using the idea of pipelined paths of the PIPESORT algorithm [AAD$^+$96], executing however an optimal number of sorts. If the detail relation R does not fit in memory, then a dimension D_i is picked (chosen carefully) and R is partitioned based on the values of D_i. Then, the subcube of $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ is computed in memory for each value of D_i using the previous algorithm - if the partition does not fit in memory there is further partitioning recursively. Finally the subcube of $D_1, \ldots, D_{i-1}, ALL, D_{i+1}, \ldots, D_n$ is computed.

Their proposed in-memory data cube computation is similar in spirit with [AAD$^+$96] and therefore Theorem 4.5 can be used to construct the appropriate pipelined paths for cuboids computation, based on their path selection methods. For the partitioning phase of their algorithm, Theorem 4.1 can be used: $B_1, B_2, \ldots, B_m$ corresponds to the subcubes $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ for all values of D_i attribute (including the ALL value). Specifically, the algebraic transformations are (B denotes the data cube over $D_1, D_2, \ldots, D_n$) :

$$\begin{array}{lll} & MD(B, R, l, \theta) & \\ = & MD(\bigcup_{x \in D_i} \sigma_{D_i = x}(B), R, l, \theta) & \\ = & \bigcup_{x \in D_i} MD(\sigma_{D_i = x}, R, l, \theta) & \text{Thm. } 4.1 \\ = & \bigcup_{x \in D_i} MD(\sigma_{D_i = x}, \sigma_{R.D_i = x}(R), l, \theta) & \text{Observ. } 4.1 \end{array}$$

The second operand of the last MD-join corresponds to the partitions mentioned in Ross and Srivastava's algorithm.

We can generalize the efficient cube computation to new applications using the algebraic framework. For one example, we can generalize efficient data cube construction to cubes over multiple fact tables. For another example, we can apply the optimizations to finding efficient techniques for computing a selected set of subcubes.

4.5 Indexing

Algorithm 3.1 can become very expensive if the base-values relation B has a large number of rows (even though B is memory resident), since for every scanned tuple t of the detail table R all B's rows are examined, resulting in a nested-loop join. However, this is not always necessary since, given a tuple t, one can identify a small number of B's rows that may be updated with respect to t during the evaluation of the MD-join operator.

For instance, consider Examples 2.2 and 3.1. During the evaluation of the first MD-join, given a tuple t of `Sales` relation, there is only one row of B that may be updated, the one that has the same value in the `cust` attribute with t's `cust` value). If B is indexed on the `cust` attribute, searching of B becomes very fast. Now consider Example 2.5. This query can be expressed by three MD-joins with theta conditions as described in that example. During the evaluation of the first MD-join, given a tuple t of `Sales` relation, there is one row of B that may be updated, the one that has the same value in the `cust` attribute with t's `cust` value and its `month` value equals t's `month` value + 1. An index on `(prod, month)` on B would reduce searching of B significantly (an index on `prod` would be sufficient.)

Definition 4.1: The set of rows of B that are updated during the evaluation of an MD-join given a tuple t (i.e. the set of rows updated in the innermost loop body of Algorithm 3.1) is called the *relative set* of B with respect to t, denoted as $Rel(t)$. ▮

Sometimes it is possible to index B once and use this indexing for the evaluation of all MD-joins in a query, as in the examples described above. In other cases we may have to create an additional index (for example, we may have to index on a column generated by a previous MD-join). However, since B can be made memory-resident (using Algorithm 3.1), it is not very expensive to build an index from scratch.

5 Discussion

Query Language The MD-join algebra provides a way to express complex OLAP queries in a succinct relational algebra, a necessary condition for efficient optimization. However users do not express queries in algebraic form, they use a query language such as SQL. Since SQL translates to relational algebra and MD-join is not part of it, this means that either there must be a mapping phase from standard relational algebra to relational algebra with MD-join (a difficult task), or SQL must be extended. We believe (having tried to express decision support queries in SQL) that the succinctness of the MD-join operator must be relayed to SQL. Several SQL extensions have been proposed in the past by many authors [KS95, GBLP96, CR96, GL98, Cha99, JC99], arguing that SQL is cumbersome and difficult to use for decision support. In this section we present some ideas on how to decouple the description of the base-values table and the description of the computation.

In [Cha99], Chatziantoniou presented EMF-SQL, a language for complex aggregation. An EMF-SQL query uses *grouping variables* to constrain the tuples to be aggregated over for each group. The grouping variable constraint may specify that tuples outside of the group should be aggregated. That is, a grouping variable is processed using an MD-join. For an example, the following EMF-SQL query expresses Example 2.5:

```
select prod, month, count(Z.*)
from Sales
where year=1997
group by prod,month ; X,Y,Z
such that X.prod=prod and X.month=month-1,
  Y.prod=prod and Y.month=month+1,
  Z.prod=prod and Z.month=month and
  Z.sale>avg(X.sale) and Z.sale<avg(Y.sale)
```

However, the EMF-SQL language cannot easily express the full range of queries that are succinctly expressed with MD-joins (that is, without extensive use of multiblock queries and views). In a previous work [JC99], we present a query language that allows the expression of many holistic aggregates by aggregating over the result of an MD-join. However, the base table is still defined as a selection from a table.

We propose to replace the `group by` or `cube by` clause by a more general clause that incorporates grouping or cubing as special cases. The format of this clause is:

`analyze by` *grouping_operation or table* `(` *list_of_attributes* `)`

The first argument *grouping_operation or table* provides the base-values table B. It can be a known operation (e.g. `group by`, `cube by`, `unpivot`, `roll-up`, `grouping sets`) or a table or view. In general, it can be any expression returning a table. The second argument is just a list of attributes.

Example 5.1: Example 2.1 could be expressed as:

```
select prod, month, state, sum(sale)
from Sales
analyze by cube(prod, month, state)
```

The unpivot query in Example 2.1 could be expressed as:

```
select prod, month, sum(sale)
from Sales
analyze by unpivot(prod, month, state)
```

Suppose that that table T contains the data points of Example 2.4. Then, this query could be expressed as:

```
select prod, month, state, sum(sale)
from Sales
analyze by T(prod, month, state)
```
∎

Performance It is natural to wonder whether Algorithm 3.1 has an efficient implementation, even in the presence of the optimizations described in Section 4. In [Cha99], Chatziantoniou presents the EMF-SQL language for expressing complex OLAP queries and also a performance study using a prototype EMF-SQL tool. The query in Example 2.5 was evaluated using both a commercially available DBMS and the prototype, and the prototype was an order of magnitude faster.

6 Conclusions

We have observed that different decision support queries require aggregation over different sets of base values. Traditional examples are queries involving group-bys and cube-bys, which define the base values differently. Other examples that aggregate over different sets of base values are: computing new aggregates for select rows of an existing data cube, the unpivot operator proposed for data mining, and materializing an optimal set of subcubes of a data cube. Other queries can involve aggregation over base values resulting from ad-hoc SQL queries. Often the aggregation over the base values is complex also (e.g., the number of purchases larger than the average purchase for the group). To unify all of these types of queries in a single relational algebraic framework, we decouple the definition of the base value set from the definition of the aggregates.

We propose a new relational operator, the *MD-join* which cleanly provides this decoupling. The operands of the MD-join include the set of base values and the relation to be aggregated; these operands can be the result of arbitrary relational expressions. We show that the MD-join operator provides tremendous flexibility in expressing decision support queries.

In addition, we show that expressing decision support queries in terms of the MD-join leads to efficient and optimized query plans. We show an efficient algorithm for implementing an MD-join. We next show a variety of algebraic transformations of expressions involving an MD-join that permits parallel execution and partitioned in-memory computation. By use of a roll-up transformation we can express the efficient data cube computation algorithms as transformed MD-joins expressions. As a result, we can generalize the cube computation algorithms and apply them to new settings.

The introduction of the MD-join operator opens a range of research questions. While we have demonstrated algebraic transformations that lead to a wide variety of optimizations, much more is possible. We also note that the MD-join operator permits the concise expression of a rich collection of decision support queries, which have not yet been fully explored.

References

[AAD$^+$96] Sarneet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the Computation of Multidimensional Aggregates. In *22nd VLDB Conference*, pages 505–521, 1996.

[AGS97] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *IEEE International Conference on Data Engineering*, 1997.

[Cha96] D. Chamberlin. *Using the New DB2*. Morgan Kaufman, 1996.

[Cha99] Damianos Chatziantoniou. Ad Hoc OLAP : Expression and Evaluation. In *IEEE International Conference on Data Engineering*, 1999.

[CR96] Damianos Chatziantoniou and Kenneth Ross. Querying Multiple Features of Groups in Relational Databases. In *22nd VLDB Conference*, pages 295–306, 1996.

[CT97] Luca Cabibbo and Riccardo Torlone. Querying Multidimensional Databases. In *International Workshop on Database Programming Languages*, pages 319–335, 1997.

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

[GBLP96] 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.

[GFC98] Goetz Graefe, Usama Fayyad, and Surajit Chaudhuri. On the Efficient Gathering of Sufficient Statistics for Classification from Large SQL Databases. In *International Conference on Knowledge Discovery and Data Mining*, pages 204–208, 1998.

[GL97] Marc Gyssens and Laks Lakshmanan. A Foundation for Multi-Dimensional Databases. In *Proceedings of the 23rd VLDB Conference*, pages 106–115, 1997.

[GL98] Frederic Gingras and Laks Lakshmanan. nD-SQL: A Multi-Dimensional Language for Interoperability and OLAP. In *VLDB Conference*, pages 134–145, 1998.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[Ill95] Illustra Information Technologies. *Illustra User's Guide*. 1995.

[JC99] T. Johnson and D. Chatziatoniou. Extending complex ad-hoc OLAP. In *Conference on Information and Knowledge Management*, pages 170–179, 1999.

[JM98] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational DBMS. In *Proc. ACM SIGMOD Conf.*, pages 379–389, 1998.

[KS95] Ralph Kimball and Kevin Strehlo. Why Decision Support Fails and How to Fix it. *SIGMOD RECORD*, 24(3):92–97, 1995.

[MRL98] Gurmeet Manku, Sridhar Rajagopalan, and Bruce Lindsay. Approximate Medians and other Quantiles in One Pass and with Limited Memory. In *ACM SIGMOD, Conference on Management of Data*, pages 426–435, 1998.

[RS96] Kenneth Ross and Divesh Srivastava. Fast Computation of Sparse Datacubes. In *23nd VLDB Conference*, pages 116–125, 1996.

[RSC98] Kenneth Ross, Divesh Srivastava, and Damianos Chatziantoniou. Complex Aggregation at Multiple Granularities. In *Extending Database Technology (EDBT), Valencia*, pages 263–277, 1998.

[SM96] M. Stonebraker and D. Moore. *Object-Relational DBMSs – The Next Great Wave*. Morgan Kaufman, 1996.

[SQL99] SQL99. *SQL99*. SQL99, 1999.

[WZ00a] H. Wang and C. Zaniolo. User defined aggregates in object-relational systems. In *Proc. 16th Intl. Conf. Data Engineering*, 2000.

[WZ00b] H. Wang and C. Zaniolo. Using SQL to build new aggregates and extenders for object-relational systems. In *Proc. Intl. Conf. Very Large Data Bases*, 2000.