Research Report

Modeling Multidimensional Databases

Rakesh Agrawal   *Ashish Gupta*   *Sunita Sarawagi*

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Research Division
Yorktown Heights, New York • San Jose, California • Zurich, Switzerland

Modeling Multidimensional Databases

Rakesh Agrawal *Ashish Gupta*[*] *Sunita Sarawagi*[†]

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

ABSTRACT: Multidimensional databases have recently gained widespread acceptance in the commercial world for supporting on-line analytical processing (OLAP) applications. We propose a hypercube-based data model and a few algebraic operations that provide semantic foundation to multidimensional databases and extend their current functionality. The distinguishing feature of the proposed model is the symmetric treatment not only of all dimensions but also measures. The model also is very flexible in that it provides support for multiple hierarchies along each dimension and support for adhoc aggregates. The proposed operators are composable, reorderable, and closed in application. These operators are also minimal in the sense that none can be expressed in terms of others nor can any one be dropped without sacrificing functionality. They make possible the declarative specification and optimization of multidimensional database queries that are currently specified operationally. The operators have been designed to be translated to SQL and can be implemented either on top of a relational database system or within a special purpose multidimensional database engine. In effect, they provide an algebraic application programming interface (API) that allows the separation of the frontend from the backend. Finally, the proposed model provides a framework in which to study multidimensional databases and opens several new research problems.

[*] Current Address: Oracle Corporation, Redwood City, California.
[†] Current Address: University of California, Berkeley, California.

1. Introduction

Multidimensional database systems have gathered tremendous market momentum as the platform for building new decision-support applications. Codd coined the phrase On-Line Analytical Processing (OLAP) in 1993 in [CCS93] to characterize the requirements for summarizing, consolidating, viewing, applying formulae to, and synthesizing data according to multiple dimensions. OLAP software enables analysts, managers, and executives to gain insight into the performance of an enterprise through fast access to a wide variety of views of data organized to reflect the multidimensional nature of the enterprise data [Col95]. It has been said that current relational database systems have been designed and tuned for On-Line Transaction Processing applications (OLTP) and are inadequate for OLAP applications [Cod93] [Fin95] [KS94] [Sta93]. In response, several multidimensional database products have appeared on the market. Examples include Arbor Software's Essbase, Comshare's Commander, Dimensional Insight's CrossTarget, Holistic Systems' Holos, Information Advantage's Axsys, Kenan Technologies' Accummate ES, MicroStrategy's DSS/Server, Oracle (IRI)'s Express, Pilot Software's LightShip Server, Planning Sciences' Gentium, Redbrick Systems' Redbrick Warehouse, Sniper's TM/1, and Stanford Technology Group's MetaCube [Rad95].

The database research community, however, has so far not played a major role in this market phenomenon. Gray et al. [GBLP95] recently proposed an extension to SQL with a *Data Cube* operator that generalizes the groupby construct to support some of the multidimensional analysis. Techniques for efficiently computing the data cube operator have attracted considerable research interest [HRU96] [JS96] [SR96] [SAG96]. The research in multidimensional indexing structures (see, for example, [Gut94] for an overview) is relevant as well. Lastly, research in statistical databases (see, for example, [Sho82] for an overview) also addressed some of the same concerns in the early eighties.

This paper presents a framework for research in multidimensional databases. We first review multidimensional database concepts and terminologies in vogue in current multidimensional database products in Section 2. We also point out some of the deficiencies in the current products. We then propose in Section 3 a data model to provide semantic backing to the techniques used by current multidimensional database products. The salient features of our model are:

- Our data model is a hypercube with a set of basic operations designed to unify the divergent styles in use today and to extend the current functionality.
- The proposed model provides symmetric treatment to not only all dimensions but also to measures. The model also is very flexible in providing support for multiple hierarchies along each dimension and support for adhoc aggregates.
- Each of our operators are defined on the cube and produce as output a new cube. Thus the operators are closed and can be freely reordered. This free composition allows a user to form larger queries, thereby replacing the relatively inefficient one-operation-at-a-time approach of many existing products. The algebraic nature of the cube also provides an opportunity for optimizing multidimensional queries.
- The proposed operators are minimal. None can be expressed in terms of others nor can any one be dropped without sacrificing functionality.
- Our modeling framework provides the logical separation of the frontend GUI used by a business analyst from the backend storage system used by the corporation. The operators thus

provide an algebraic application programming interface (API) that allows the interchange of frontends and backends.

We discuss in Section 4 some of our design choices and show how the current popular multidimensional database operations can be expressed in terms of the proposed operators. These operators have been designed to be translated into SQL, albeit with some minor extensions to SQL. We give these translations in the Appendix. Thus, our data model can be implemented on either a general-purpose relational system or a specialized engine. We conclude with a summary in Section 5.

2. Current State of the Art

We begin with a brief overview of the current state of art in multidimensional databases. Readers familiar with the OLAP literature and current products may skip this review.

Example 2.1. Consider a database that contains point of sale data about the sales price of products, the date of sale, and the supplier who made the sale. The *sales* value is functionally determined by the other three attributes. Intuitively, each of the other three attributes can "vary" and accordingly determine the sales value. Figure 1 illustrates this "hypercube" view of the world.

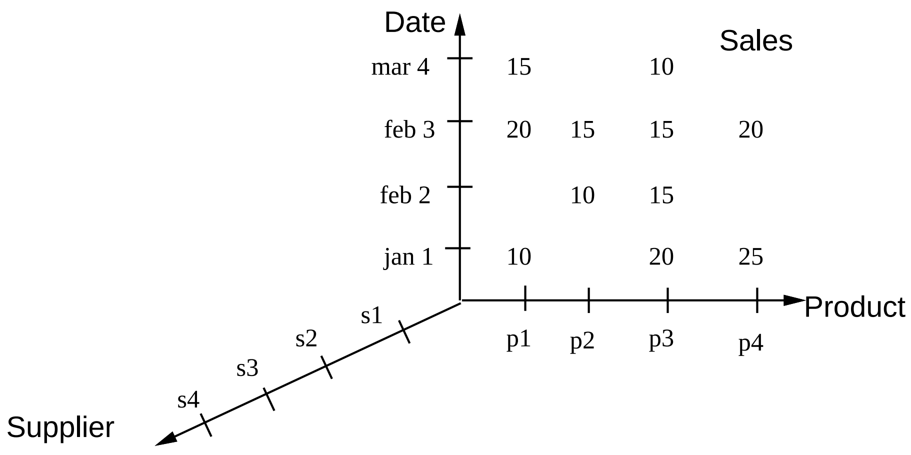

Figure 1: Example data cube

□

2.1. Terminology

Determining attributes like *product*, *date*, *supplier* are referred to as *dimensions* while the determined attributes like *sales* are referred to as *measures*[1]. There is no formal way of deciding which attributes should be made dimensions and which attributes should be made measures. It is left as a database design decision.

[1]Dimensions are called categorical attributes and measures are called numerical or summary attributes in the statistical database literature [Sho82].

Dimensions usually have associated with them *hierarchies* that specify aggregation levels and hence granularity of viewing data. Thus, *day* $\rightarrow$ *month* $\rightarrow$ *quarter* $\rightarrow$ *year* is a hierarchy on *date* that specifies various aggregation levels. Similarly, *product name* $\rightarrow$ *type* $\rightarrow$ *category* is a hierarchy on the *product* dimension. For instance, "ivory" and "irish spring" both are of type "soap." Furthermore, "soap" and "shampoo" both are in category "personal hygiene" products.

An analyst might want to see only a subset of the data and thus might view some attributes and within each selected attribute might restrict the values of interest. In multidimensional database parlance, the operations are called *pivoting* (rotate the cube to show a particular face) and *slicing-dicing* (select some subset of the cube). The multidimensional view allows hierarchies associated with each dimension also to be viewed in a logical manner. Aggregating the product dimension from product name to product type is expressed as a *roll-up* operation in a multidimensional database. The converse of roll-up is *drill-down* that displays detail information for each aggregated point. Thus, drilling-down the product dimension from product category to product type gets the sales for each product type corresponding to each product category. Further drill down will get sales for individual products. Drill-down is essential because often users want to see only aggregated data first and selectively see more detailed data.

Example 2.2. We give below some queries to give a flavor of multidimensional queries. These queries use the database from Example 2.1 and other necessary hierarchies on product and time dimensions.

- Give the total sales for each product in each quarter of 1995. (Note that quarter is a function of date).
- For supplier "Ace" and for each product, give the fractional increase in the sales in January 1995 relative to the sales in January 1994.
- For each product give its market share in its category today minus its market share in its category in October 1994.
- Select top 5 suppliers for each product category for last year, based on total sales.
- For each product category, select total sales this month of the product that had highest sales in that category last month.
- Select suppliers that currently sell the highest selling product of last month.
- Select suppliers for which the total sale of every product increased in each of last 5 years.
- Select suppliers for which the total sale of every product category increased in each of last 5 years.

$\square$

2.2. Implementation Architectures

There are two main approaches currently used to build multidimensional databases. One approach maintains the data as a k-dimensional cube based on a non-relational specialized storage structure for storing k-dimensional data. The database designer specifies all the aggregations they consider useful. While building the storage structure these aggregations associated with all possible roll-ups are precomputed and stored. Thus, roll-ups and drill-downs are answered in interactive time. Many products have adopted this approach – for instance, Arbor Essbase [Arb] and IRI Express [IRI].

Another approach uses a relational backend wherein operations on the data cube are translated to relational queries (posed in a possibly enhanced dialect of SQL). Indexes built on materialized views are heavily used in such systems. This approach also manifests itself in many products – for instance, Redbrick [Eri95] and Microstrategy [Mic].

2.3. Additional Desired Functionality

We believe that multidimensional database systems must provide the following additional functionality, which is either missing or poorly supported in current products:

- **Symmetric treatment not only of all dimensions but also of measures**. That is, selections and aggregations should be allowed on all dimensions and measures. For example, consider a query that finds the total sales for each product for ranges of sales price like 0-999, 1000-9999 and so on. Here the sales price of a product, besides being treated as a measure, is also the grouping attribute. Such queries that require categorizing on a "measure" are quite frequent. Non-uniform treatment of dimensions and measures makes such queries very hard in current products. Codd also emphasized the need for symmetric treatment of dimensions and measures in his 12 rules of OLAP [CCS93].
- **Support for multiple hierarchies along each dimension**. For instance, Example 2.1 illustrates the type-category hierarchy on products (of interest to a consumer analyst). An alternative hierarchy is one based on which company manufactures the product and who owns that company, namely, $product \rightarrow manufacturer \rightarrow parent\ company$ (of interest to a stock market analyst). Roll-ups/drill-downs can be on either of the hierarchies.
- **Support for computing ad-hoc aggregates**. That is, aggregates other than those originally prespecified should be computable. For instance, for each product both the total sales and the average sales are interesting numbers.
- **Support for a query model in place of one-operation-at-a-time computation model**. Currently, a user operates on a cube once and obtains the resulting cube. Then the user makes the next operation. However, not all the intermediate cubes are of interest to the user. A set of basic operators that have well defined semantics enable this computation to be replaced by a query model. Thus, having tools to compose operators allows complex multidimensional queries to be built and executed faster than having the user specify each step. This approach is also more declarative and less operational.

2.4. Related Research Work

Data models developed in the context of temporal, spatial and statistical databases also incorporate dimensionality and hence have similarities with our work.

In temporal databases [TCG$^+$93], rows and columns of a relational table are viewed as two dimensions and "time" adds a third dimension forming what is called the "time cube". This cube is different from a cube in our model where dimensions correspond to arbitrary attributes and all dimensions are treated uniformly without attaching any fixed connotation with any one of them.

The modeling efforts in spatial databases [Gut94] mostly concentrate on representing arbitrary geometric objects (points, lines, polygons, regions *etc.*) in multidimensional space. By viewing OLAP data as points in the multidimensional space of attributes, we could draw analogies between the two models. But the operations central to spatial databases ("overlap", "containment", etc.) are very different from the common OLAP operations that we are trying to capture ("roll-up", "drill-down", "joins" *etc.*). However, the multi-dimensional indexing structures developed for spatial databases (see [Gut94]) are likely to figure prominently in developing efficient implementations of OLAP databases.

Statistical databases also address some of the same concerns as OLAP databases. However, models in the statistical database literature [Mic92] [CL94] have been primarily concerned with extending existing data models (mostly relational) for representing summaries and supporting operations for statistical processing. In contrast, our objective has been to develop a model and a basic set of operations that closely abstracts the analysts view of enterprise data. In statistical databases, category (dimensions) and summaries (measures) are treated quite differently, whereas we have strived to treat dimensions and measures uniformly. However, OLAP databases will benefit from implementation techniques developed in statistical databases, particularly related to aggregation views (see [Sho82] [STL89]).

3. Data Model

We now outline our proposed multidimensional data model and operations that capture the functionality currently provided by multidimensional database products and the additional desired functionality listed above. Our design was driven by the following key considerations:

- Treat dimensions and measures symmetrically.
- Strive for flexibility (multiple hierarchies, adhoc aggregates).
- Keep the number of operators small.
- Stay as close to relational algebra as possible. Make operators translatable to SQL.

In our logical model, data is organized in one or more hypercubes. A cube has the following components:

- k dimensions, and for each dimension a name D_i, a domain dom_i from which values are taken.
- Elements defined as a mapping $E(C)$ from $dom_1 \times \ldots \times dom_k$ to either an n-tuple, 0, or 1. Thus, $E(C)(d_1, \ldots, d_k)$ refers to the element at "position" $d_1, \ldots, d_k$ of cube C. Note, the d_is refer to values not positions per se.
- Part of the metadata is an n-tuple of names where each of the names describes one of the members of a n-tuple element of the cube. Note, if the cube has no n-tuple elements, then this description is an empty tuple.

The elements of a cube can be either 0, 1, or an n-tuple $< X_1, \ldots, X_n >$. If the element corresponding to $E(C)(d_1, \ldots, d_k)$ is 0 then that combination of dimension values does not exist in the database. A 1 indicates the existence of that particular combination. Finally, an n-tuple indicates that additional information is available for that combination of dimension values. If any of the elements of a cube is a 1 then none of the elements can be a n-tuple and vice-versa. We represent only those values along a dimension of a cube for which at least one of the elements of the cube is not 0. If all the elements of a cube are 0 then the cube is *empty*. Additionally, if domain dom_i of dimension D_i has no values then too the cube is considered to be empty.

In our model, no distinction is made between measures and dimensions. Thus, in Example 2.1, **sales** is just another dimension (unlike existing multidimensional database systems that will treat **sales** as a "measure" or "element"). Note that this is a logical model and does not force any storage mechanism. Thus, a cube in our data model may have more logical dimensions than the number of dimensions used to physically store the cube in a multidimensional storage system.

3.1. Operators

We now discuss our multidimensional operators. We illustrate the operators using a 2-D subset of the cube introduced in Example 2.1. We omit the **supplier** dimension and display in Figure 2 only the **product**, **date**, and **sales** dimensions. Note, **sales** is not a measure but another dimension, albeit only logical, in the model.

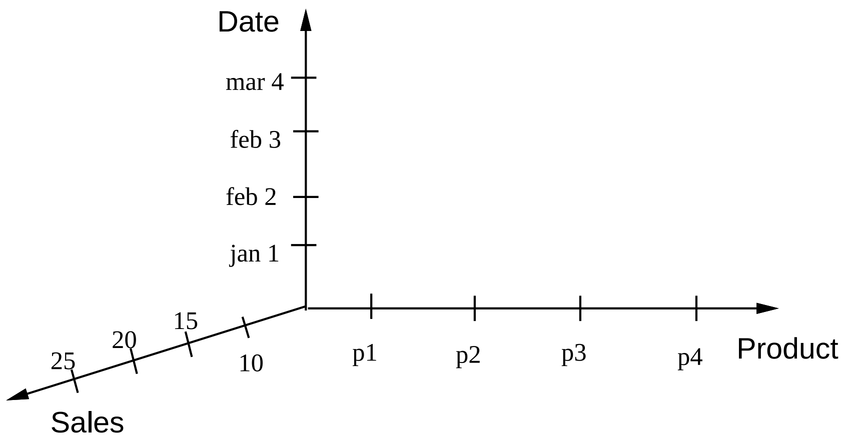

Figure 2: Logical cube wherein **sales** is a dimension (omitting the 1/0's)

To operate on the logical cube, the **sales** dimension may have to be folded into the cube such that sales values seem determined by the **product** and **date** dimensions. We describe later how this is achieved. For now, we will use the cube with **sales** values as the sole member of the elements of the cube. Thus, the value $<15>$ for "$date = mar$ 4" and "$product = p1$" in Figure 3 indicates that in the logical cube of Figure 2 the element corresponding to "$date = mar$ 4", "$product = p1$", and "$sales = 15$" is "1". We show the metadata description of the elements as an annotation in the cube. Thus, $<$**sales**$>$ in Figure 3 indicates that each element in the cube is a sales value.

Notation. We define the operators using a cube C with k dimensions. We refer to the dimensions as $D_1, \ldots, D_k$. We use D_i to refer also to the domain of dimension D_i if the context makes the usage clear; otherwise we refer to the domain of dimension D_i as $dom_i(C)$. We use lower case letters like a, b, c to refer to constants.

Dimension values in our data model functionally determine elements of the cube. As a result of an application of an operation, more than one element value may be mapped to the same element (*i.e.* the same combination of values of dimension attributes) of the answer cube. These element values are combined into one value to maintain functional dependency by specifying what we call an *element combining function*, denoted as f_{elem}.

We also sometime merge values along a dimension. We call functions used for this purpose *dimension merging functions*, denoted as f_{merge}.

Push. The push operation (see Figure 3) is used to convert dimensions into elements that can then be manipulated using function f_{elem}. This operator is needed to allow dimensions and measures to be treated uniformly.

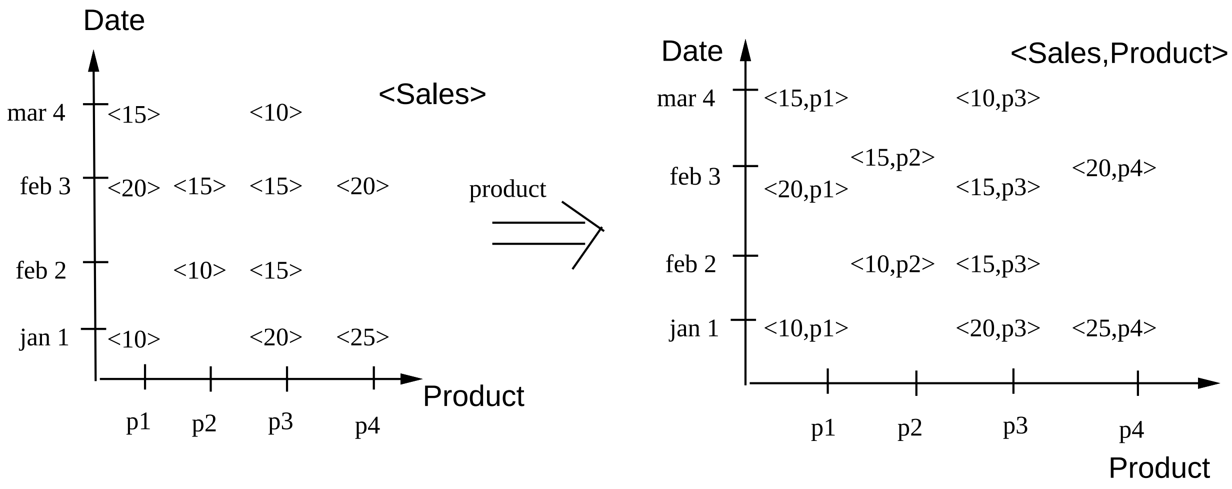

Figure 3: The push operation on dimension **product**

- Input: C, D_i.
- Output: C with each non-0 element extended by an additional member, the value of dimension D_i for that element. If an element is not a n-tuple but a "1", then it is converted to a 1-tuple that contains the corresponding value of the dimension.
- Mathematically: $\text{push}(C, D_i) = C_{ans}$.
  $E(C_{ans})(d_1, \ldots, d_k) = g \oplus < d_i >$ where $g = E(C)(d_1, \ldots, d_k)$. The operator $\oplus$ is defined to be 0 if $g = 0$, it is $< d_i >$ if $g = 1$, and in all other cases it concatenates the two tuples.

Pull. This operation is the converse of the push operator. Pull creates a new dimension for a specified member of each element. The operator is useful to convert an element into a dimension so that the element can be used for merging or joining. This operator too is needed for the symmetric treatment of dimensions and measures.

- Input: C, new dimension name D, integer i.
- Output: C_{ans} with an additional dimension D that is obtained by pulling out the i^{th} element of each element of the matrix.
- Constraint: all non-0 elements of C are n-tuples because each non-0 element need at least one member to enable the creation of a new dimension.

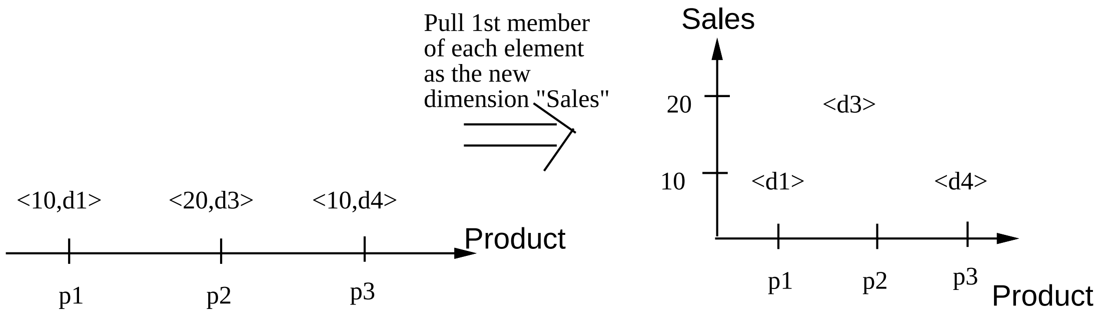

Figure 4: Pull first member of each element as dimension **sales**

- Mathematically: pull$(C, D, i) = C_{ans}$, $1 \leq i \leq n$.
  D becomes the $k+1^{st}$ dimension of the cube.
  $dom_{k+1}(C_{ans}) = \{e | e \textit{ is the } i^{th} \textit{ member of some element } E(C)(d_1, \ldots, d_k)\}$.
  $E(C_{ans})(d_1, \ldots, d_k, e_i) = < e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n >$ if $E(C)(d_1, \ldots, d_k) = < e_1, \ldots, e_i, \ldots, e_n >$, otherwise $E(C_{ans})(d_1, \ldots, d_k, e_i) = 0$. Note, if the resulting element has no members then it is replaced by 1

Destroy Dimension. Often the dimensionality of a cube needs to be reduced. This operator removes a dimension D that has in its domain a single value. The presence of a single value implies that for the remaining $k-1$ dimensions, there is a unique $k-1$ dimensional cube. Thus, if dimension D is eliminated then the resulting $k-1$ dimensional space is occupied by this unique cube.

- Input: C, dimension name D_i.
- Output: C_{ans} with dimension D_i absent.
- Constraint: D_i has only one value.
- Mathematically: destroy(C, D_i)
  C_{ans} has $k-1$ dimensions.

A dimension that has multiple values cannot be directly destroyed because then elements would no longer be functionally determined by dimension values. A multi-valued dimension is destroyed by first applying a merge operation (described later) and then applying the above operation.

Restriction. The restrict operator operates on a dimension of a cube and removes the cube values of the dimension that do not satisfy a stated condition. Figure 5 illustrates an application of restriction. Note that this operator realizes slicing/dicing of a cube in multidimensional database terminology.

- Input: Cube C and predicate P defined on D_i.
- Output: New cube C_{ans} obtained by removing from C those values of dimension D_i that do not satisfy the predicate P. Note, P is evaluated on a set of values and not on just a single value. Thus, P takes as input the entire domain D_i and could output a set of values like the "top 5 values". If no element of dimension D_i satisfies P then C_{ans} is an empty cube.

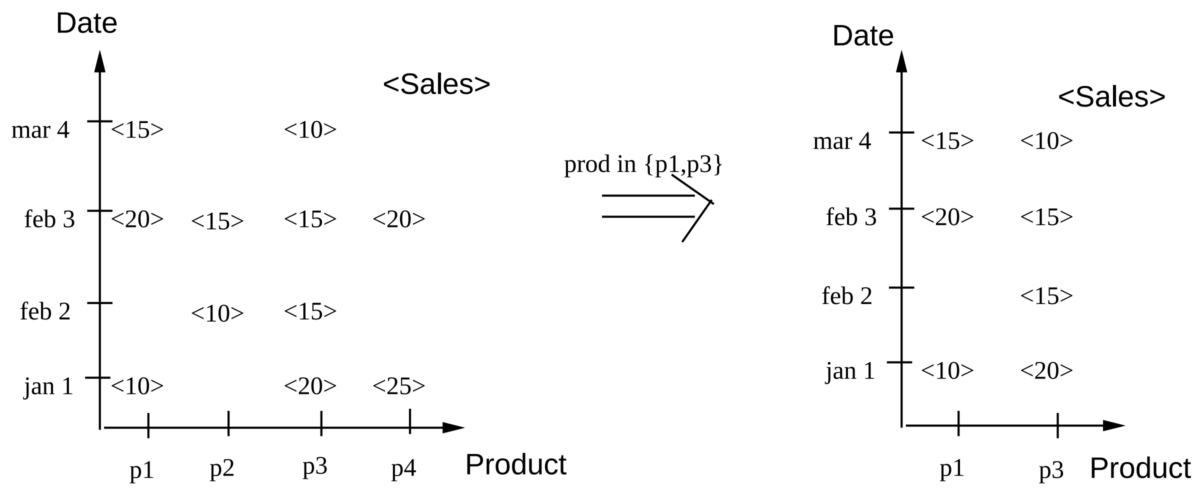

Figure 5: The restriction operation

- Mathematically: restrict$(C, D_i, P) = C_{ans}$.
  $dom_j(C_{ans}) = dom_j(C)$ if $1 \leq j \leq k$ & $j \neq i$ else $dom_j(C_{ans}) = P(dom_j(C))$.
  $E(C_{ans})(d_1, \ldots, d_k) = E(C)(d_1, \ldots, d_k)$.

Join. The join operation is used to relate information in two cubes. The result of joining a m-dimensional cube C with an n-dimensional cube $C1$ on k dimensions, called *joining dimensions*, is cube C_{ans} with $m + n - k$ dimensions. Each joining dimension D_i of C combines with exactly one dimension D_{x_i} of $C1$; the corresponding result dimension will have values that are union of the values on D_i and D_{x_i}. Transformations can optionally be applied to the values of dimensions D_i and D_{x_i} before they are mapped to the result dimension. The elements of the resulting cube C_{ans} are obtained by combining via a function f_{elem} all elements of C and $C1$ that get mapped to the same element of C_{ans}.

Figure 6 illustrates cube C joining with cube $C1$ on dimension D_1 (no transformation is used). Dimension D_1 of the resulting cube has only two values. The function f_{elem} divides the element value from cube C by the element value from $C1$; if either element is 0 then the resulting element is also 0. Values of result dimension that have only 0 elements corresponding to them are eliminated from C_{ans} (like values 0 and 3 for dimension D_1). This elimination is a consequence of representing in the cube only those values along a dimension that have at least one non-0 element.

- Input: C with dimensions $D_1 \ldots D_m$ and $C1$ with dimensions $D_{m-k} \ldots D_n$. Without loss of generality, dimensions $D_{m-k}, \ldots, D_m$ are the join dimensions. $2k$ mapping functions, $f_{m-k}, \ldots, f_m$ defined over values of dimensions $D_{m-k}, \ldots, D_m$ of C and $f'_{m-k}, \ldots, f'_m$ defined over dimensions $D_{m-k}, \ldots, D_m$ of $C1$. Mapping f_i applied to value $v \in dom_i(C)$ produces values for dimension D_i in C_{ans}. Similarly f'_i applied to $v' \in dom_i(C1)$ produces values for dimension D_i in C_{ans}. Also needed is a function f_{elem} that combines sets of elements from C and $C1$ to output elements of C_{ans}.

- Output: C_{ans} with n dimensions, $D_1 \ldots D_n$. Multiple elements in C and $C1$ could get mapped to the same element in C_{ans}. All elements of C and $C1$ that get mapped to the same point in C_{ans} are combined by function f_{elem} to produce the output element of C_{ans}. If for some

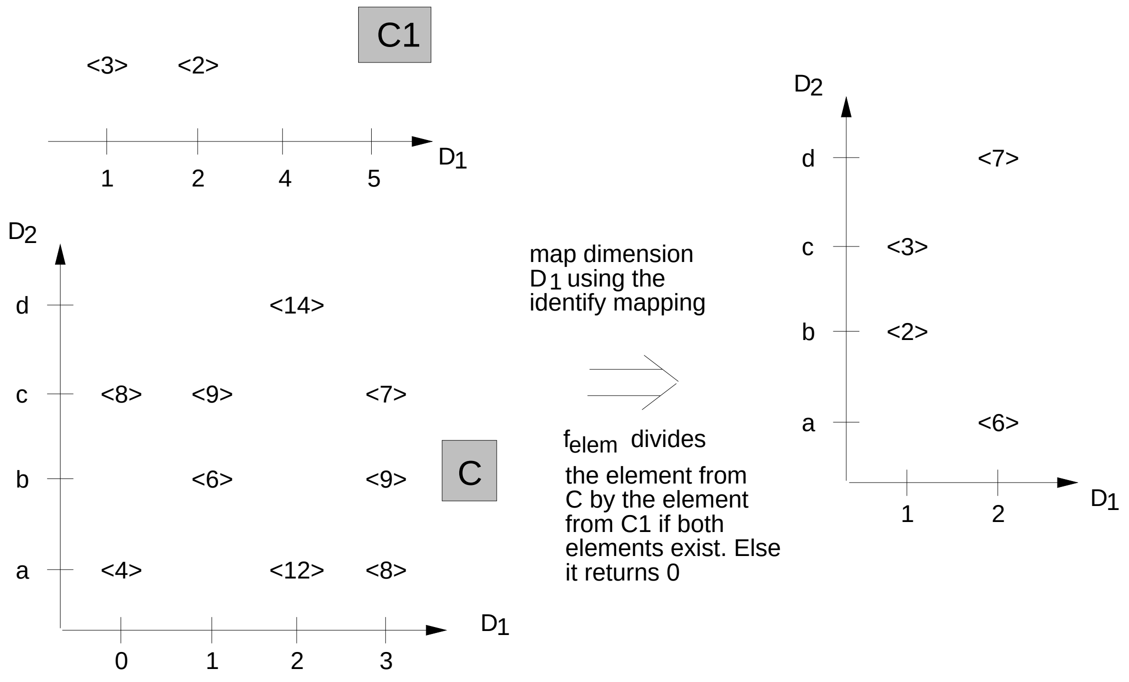
C1
<3>
<2>
D1
1
2
4
5
D2
d
c
b
a
<14>
<8>
<9>
<7>
<6>
<9>
<4>
<12>
<8>
C
D1
0
1
2
3
map dimension D1 using the identify mapping
felem divides the element from C by the element from C1 if both elements exist. Else it returns 0
D2
d
<7>
c
<3>
b
<2>
a
<6>
1
2
D1

Figure 6: Joining two cubes

value v of dimension D_i, the elements $E(C_{ans})(x_1, \ldots, v, x_{i+1}, \ldots, x_n)$ is 0 for all values of the other dimensions, then v is not included in dimension D_i of C_{ans}.

- Mathematically: join$(C, C1, [f_{m-k}, \ldots, f_m, f'_{m-k}, \ldots, f'_m], f_{elem}) = C_{ans}$.
  $dom_i(C_{ans}) = dom_i(C)$ if $1 \leq i \leq m-k-1$.
  $dom_i(C_{ans}) = dom_i(C1)$ if $m \leq i \leq n$.
  $dom_i(C_{ans}) = \{d^a | d^a \in f_i(d), d \in dom_i(C)$ OR $d^a \in f'_i(d'), d' \in dom_i(C1)\}$.
  $E(C_{ans})(d_1, \ldots, d^a_{m-k}, \ldots, d^a_m, \ldots, d_n) = f_{elem}(\{t1\}, \{t2\})$ such that
  $t1 = E(C)(d_1, \ldots, d_{m-k}, \ldots, d_m)$, $t2 = E(C1)(d'_{m-k}, \ldots, d'_m, d_{m+1}, \ldots, d_n)$, and
  $d^a_i \in f_i(d_i)$ OR $d^a_i \in f'_i(d'_i)$ for $m-k \leq i \leq m$.

The join operator has two notable special cases: **cartesian product** and **associate**. In the case of cartesian product, the two cubes have no common joining dimension.

Associate is especially useful in OLAP applications for computations like "express each month's sale as a percentage of the quarterly sale." Associate is asymmetric and requires that each dimension of $C1$ be joined with some dimension of C. Figure 7 illustrates associating cube $C1$ with C where *month* dimension of $C1$ and *date* dimension of C are joined by mapping them to the *date* dimension of C_{ans}. Similarly, *category* and *product* are joined by mapping them to *product* of C_{ans}. For dimension *month*, each month is mapped to all the dates in that month. For dimension *category*, value *cat*1 is mapped to *products* $p1$ and $p2$, and *cat*2 is mapped to $p3$ and $p4$. For dimensions *date* and *product* the identity mapping is used. Function f_{elem} divides the element value from cube C by the element value from $C1$; if either element is 0 then the resulting element is also 0. Note, value *mar4* is eliminated from C_{ans} because all its corresponding elements are 0.

Merge. The merge operation is an aggregation operation. We illustrate it in Figure 8. The figure shows how hierarchies in a multidimensional database are implemented using the merge operator.

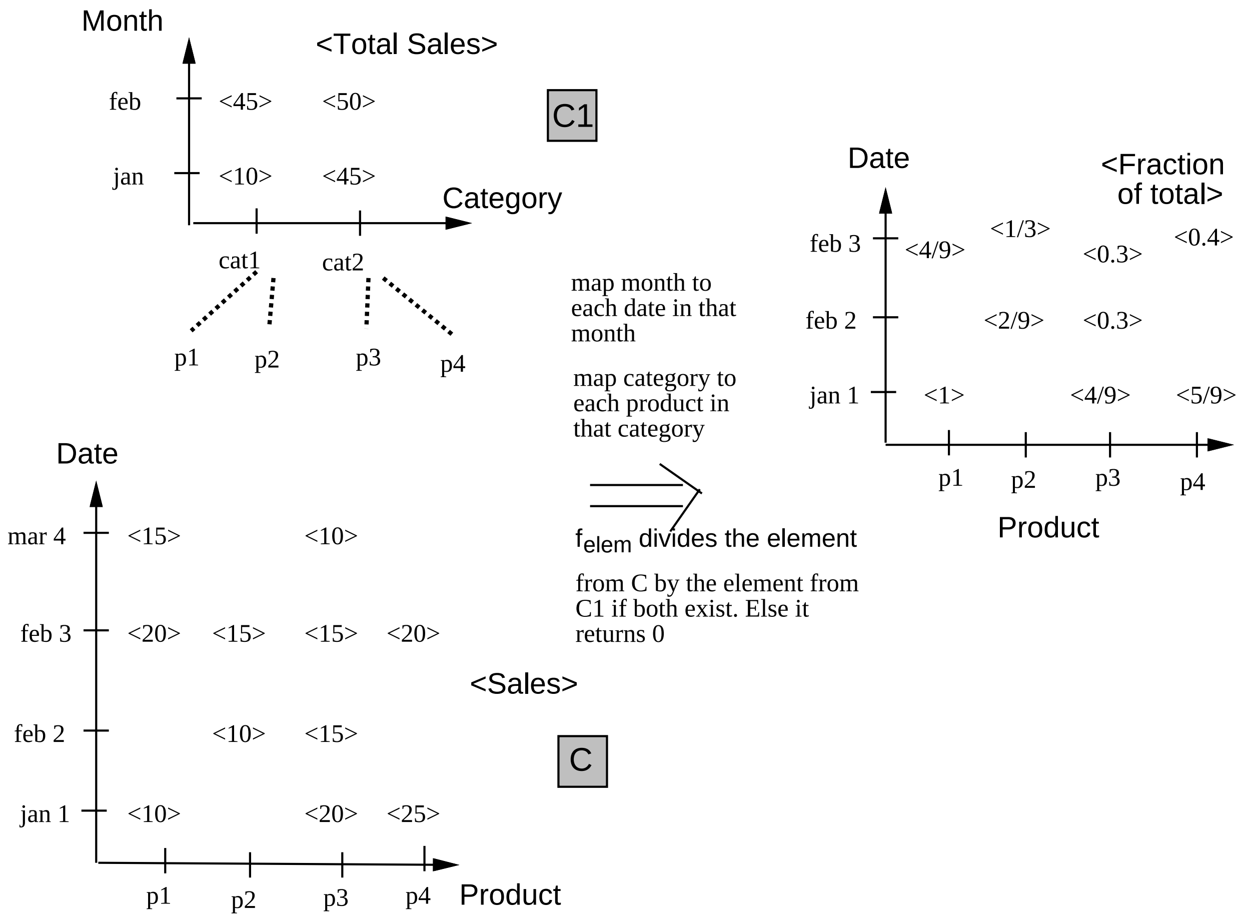

Figure 7: Associating two cubes

Intuitively, a dimension merging function is used to map multiple product names into one or more categories and another function is used to map individual dates into their corresponding month. Thus, multiple elements on each dimension are merged to produce a dimension with a possibly smaller domain. As a result of merging each dimension, multiple elements in the original cube get mapped to the same element in the new cube. An element combining function is used to specify how these multiple elements are aggregated into one. In the example in Figure 8, the aggregation function f_{elem} is **sum**.

In general, the dimension merging function might be a one-to-many mapping that takes an element in the lower level into *multiple* elements in the higher level of hierarchies. For instance, a $1 \rightarrow n$ mapping can be used to merge a product belonging to n categories to handle multiple hierarchies.

- Input: C, function f_{elem} for merging elements and m (dimension, function) pairs. Without loss of generality, assume that the m pairs are $[D_1, f_{merge_1}], \ldots, [D_m, f_{merge_m}]$
- Output: Cube C_{ans} of same dimensionality as C. Dimension D_i is merged as per function f_{merge_i}. An element corresponding to the merged elements is aggregated as per f_{elem}.
- Mathematically: merge$(C, \{[D_1, f_{merge_1}], \ldots, [D_m, f_{merge_m}]\}, f_{elem}) = C_{ans}$.
  $dom_i(C_{ans}) = \{f_{merge_i}(e) | e \in dom_i(C)\}$ if $1 \leq i \leq m$, else $dom_i(C_{ans}) = dom_i(C)$.

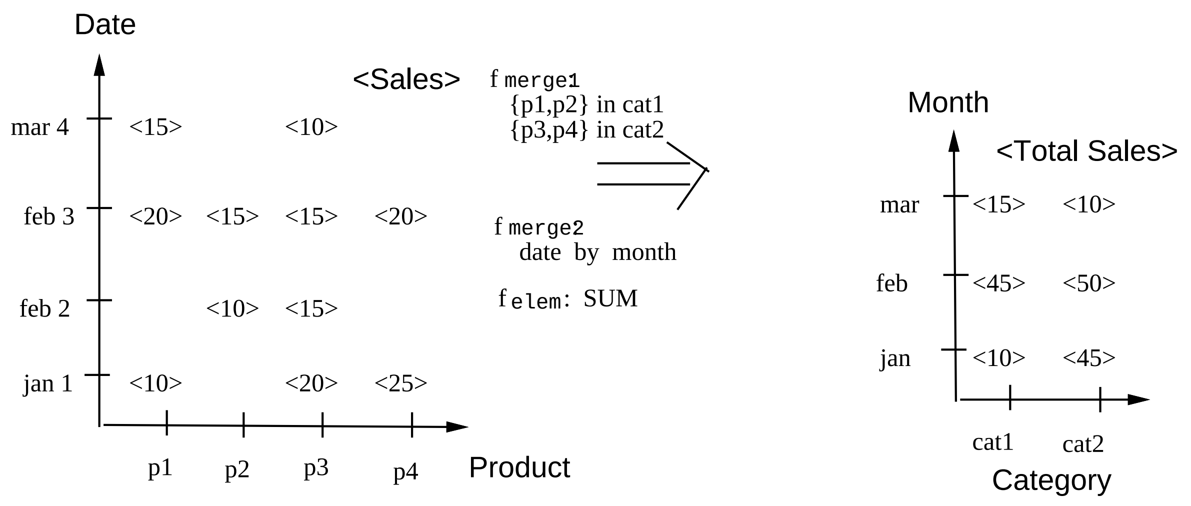

Figure 8: Merging dimensions **date** and **product** using $f_{elem} = sum$

$E(C_{ans})(d_1, \ldots, d_k) = f_{elem}(\{t | t = E(C)(d'_1, \ldots, d'_k)$ where $f_{merge_i}(d'_i) = d_i$ if $1 \leq i \leq m$ else $d'_i = d_i\})$.

A special case of the merge operator is when all the merging functions are identity. In this case, the merge operator can be used to *apply* a function f_{elem} to each element of a cube.

Remark. The merge operator is strictly not part of our basic set of operators. It can be expressed as a special case of the self-join of a cube using f_{merge} transformation functions on dimensions being merged and identity transformation functions for other dimensions. We chose to separately define merge because it is a unary operator unlike the binary join and also for performance reasons.

4. Discussion

The reader may have noticed similarities in the operators proposed and relational algebra [Cod70]. It is by design. One of our goals was to explore how much of the functionality of current multidimensional products can be abstracted in terms of relational algebra. By developing operators that can be translated into SQL (see Appendix), our hope was to create a fast path for providing OLAP capability on top of relational database systems. We must hasten to add that we are not arguing against specialized OLAP engines—we believe the design and prototyping of such engines is a fruitful research direction. We are also not suggesting that simply translating these operators into SQL would be sufficient for providing OLAP capabilities in relational database systems. However, it does point to directions in which optimization techniques and storage structures in relational database systems need to evolve.

The goal of treating dimensions and measures symmetrically had a permeating influence in our design. It is a functionality either not present or poorly supported in current multidimensional database products. Its absence causes expensive schema redesign when an unanticipated need arises for aggregation along an attribute that was initially thought to be a measure. In hindsight, the push and pull operations may appear trivial. However, their introduction was the key that made the symmetric treatment of dimensions and measures possible.

The reader may argue with the way we have chosen to incorporate order-based information into our algebra. We rely on functions for this purpose, which implies that the system may not be able to use this information in optimizing queries. We debated about allowing a native order to be specified with each dimension and providing ordering operators. We decided against it because of the large number of such operators and because the semantics gets quite complex when there are multiple hierarchies along a dimension. In a practical implementation of our model, it will be worthwhile to allow a default order to be specified with each dimension and make system aware of some built-in ordering functions such as "first n". The same holds for providing the knowledge of some built-in aggregate functions.

The reader may also notice the absence of direct analogs of relational projection, union, intersection, and difference. These operations can be expressed in terms of our proposed operators as follows:

Projection. The projection of a cube is computed by merging each dimension not included in the projection and then destroying the dimension. A f_{elem} function specifying how multiple elements are combined is needed as part of the specification of the projection.

Union. Two cubes are union-compatible if (i) they have the same number of dimensions; and (ii) for each dimension D_i in C, dimension D_i in $C1$ is of the same domain type. Union is computed by joining the two cubes using the identity transformation functions for each dimension of each cube and by choosing a function f_{elem} that produces a non-empty element for element e in C_{ans} whenever an element from either of the two cubes is mapped into e. Dimension D_i in the resulting cube has as its values the union of the values in $dom_i(C)$ and in $dom_i(C1)$.

Intersect. The intersection of two union-compatible cubes is computed by joining the cubes through the identity mapping that effectively retains only those dimension values that are present in both cubes. Thus, function f_{elem} makes non-0 an element for point p in C_{ans} only if elements from both cubes are mapped into p.

Difference. The difference of two union-compatible cubes $C1$ and $C2$ is expressed as an intersection of $C1$ and $C2$ followed by a union of the result with $C1$. The f_{elem} function for combining two elements for the intersection steps discards the value of the element for $C1$ and retains $C2$'s element. The f_{elem} function for combining two elements for the union step saves the value of $C1$'s element if the two elements are different and makes the result 0 if they are identical[2].

4.1. Expressive Power

It is easy to see that our algebra is at least as powerful as relational algebra [Cod70]. A precise circumscription of the expressive power of the proposed data model is an open problem. A related interesting open question pertains to defining a formal notion of completeness for multidimensional database queries and evaluating how complete our algebra is with respect to that metric. Formalisms developed in the context of relational databases, for example, in [AU79] [CH82] may provide starting point for pursuing this direction.

[2]This implementation corresponds to the following semantics for $C1 - C2$: $E(C_{ans})(d_1, \ldots, d_k)$ equals 0 if $E(C2)(d_1, \ldots, d_k) = E(C1)(d_1, \ldots, d_k)$; it is $E(C1)(d_1, \ldots, d_k)$ otherwise. Another alternative semantics could be that $E(C_{ans})(d_1, \ldots, d_k)$ equals 0 if $E(C2)(d_1, \ldots, d_k) \neq 0$, and $E(C1)(d_1, \ldots, d_k)$ otherwise. This semantics can be implemented by a small change in the f_{elem} function used in the union step.

We take an empirical approach and discuss below how the current high-level multidimensional operations can be built using our proposed operators.

Roll-up. Roll-up is a merge operation that needs one dimension merging function and one element combining function. If a hierarchy is specified on a dimension then the dimension merging function is defined implicitly by the hierarchy. The elements corresponding to merged values on the dimension are combined using the user-specified element combining function like SUM.

Drill-down. This operator is actually a binary operation even though most current multidimensional database products make it seem like a unary operation. Consider computing the sum X of 10 values. Drill-down from X to the underlying 10 values is possible in infinite ways. Thus, the underlying 10 values have to be known. That is, the aggregate cube has to be joined (actually associated) with the cube that has detailed information. Continuing with our analogy, to drill down from X to its constituents the database has to keep track of how X was obtained and then associate X with these values. Thus, if users merge cubes along stored paths and there are unique path down the merging tree, then drill down is uniquely specified. By storing hierarchy information and by restricting single element merging functions to be used along each hierarchy, drill-down can be provided as a high-level operation on top of associate.

Star Join. In a star join [Eri95], a large detail "mother" table M is joined with several small "daughter" tables that describe join keys in the mother table. A star join denormalizes the mother table by joining it with its daughter tables after applying selection conditions to their descriptive attributes. We describe how our operators capture a star join when M has one daughter table F_1 that describes the join key field D of M. Table F_1 can be viewed as a one-dimensional cube, C_1 with the join key field D as the dimension and all the description fields pulled in as elements. A restriction on a description attribute A of table F_1 corresponds to a function application to the elements of C_1. Restrictions on the join key attribute translate to restrictions on dimension D of C_1. The join between M and F_1 is achieved by associating the mother cube with the daughter cube on the key dimension D using the identity mapping function. The description of each key value is pulled in from the daughter cube into the mother cube via the f_{elem} function.

Expressing a dimension as a function of other dimensions. This functionality is basic in spread sheets. We can create a new dimension D expressed as a function, f of another dimension D' by first pushing D' into the cube elements, then modifying the cube elements by applying function f and finally pulling out the corresponding member of the cube element as a new dimension D.

4.2. Example queries

This section illustrates how to express some of the queries of Example 2.2 using our operators. Assume we have a cube C with dimensions product, month, supplier and element sales.

- For supplier "Ace" and for each product, give the fractional increase in the sales in January 1995 relative to the sales in January 1994.

  Restrict supplier to "Ace" and dates to "January 1994 or January 1995". **Merge** date dimension using an f_{elem} that combines sales as $(B - A)/A$ where A is the sale in Jan 1994 and B is the sale in Jan 1995.

- For each product give its market share in its category this month minus its market share in its category in October 1994.

  Restrict date to "October 1994 or current month". **Merge** supplier to a single point using sum of sales as the f_{elem} function to get $C1$. **Merge** product dimension to category using sum as the f_{elem} function to get in $C2$ the total sale for the two months of interest. **Associate** $C1$ and $C2$, mapping a category in $C2$ to each of its products in $C1$. The identity mapping is used for the Month dimension. Function f_{elem} divides the element from $C1$ by the element from $C2$ to get the market share. For the resulting cube, **Merge** dimension month to a single point using a f_{elem} function $(A - B)$ where A is the market share for "this" month and B is the market share in October 1994.

- For each product category, select total sales this month of the product that had highest sales in that category last month.

  Restrict dimension month to "last" month. **Merge** supplier to a single point using sum of sales as the f_{elem} function. **Push** product dimension resulting in 2-tuple elements with <Sale and product>. **Merge** product to category using f_{elem} function that retains an element if it has the "maximum" sales. **Pull** product into the category dimension (over-riding the category dimension, this can be easily done using our basic operators). Let the resulting cube be $C1$. This cube has the highest sales value for each element for "last" month. **Restrict** C on dimension date to "this" month, **Merge** supplier to a single point using sum of sales as the f_{elem} function and **associate** it with $C1$ on the product dimension using f_{elem} function that only outputs the element of C when it is the same as the corresponding elements from $C1$ (otherwise returns 0).

- Select suppliers for which the total sale of every product increased in each of last 5 years.

  Restrict to months of last 6 years. **Merge** month to year. **Merge** years to a single point using a f_{elem} function that maps the six sales values to "1" if sales values are increasing, "0" otherwise. **Merge** product to a point where f_{elem} function is "1" if and only if all its arguments are "1".

5. Conclusions and Future Work

This paper introduced a data model and a set of algebraic operations that unify and extend the functionality provided by current multidimensional database products. As illustrated in Section 4.1 the proposed operators can be composed to build OLAP operators like roll-up, drill-down, star-join and many others. In addition, the model provides symmetric treatment to dimensions and measures. The model also provides support for multiple hierarchies along each dimension and support for adhoc aggregates. Absence of these features in current products results in expensive schema redesign when an unanticipated need arises for a new aggregation or aggregation along an attribute that was initially thought to be a measure.

The proposed operators have several desirable properties. They have well-defined semantics. They are minimal in the sense that none can be expressed in terms of others nor can any one be dropped without sacrificing functionality. Every operator is defined on cubes and produces as output a cube. That is, the operators are closed and can be freely composed and reordered. This allows the inefficient one-operation-at-a-time approach currently in vogue to be replaced by a query model and makes multidimensional queries amenable to optimization.

The proposed operators enable the logical separation of the frontend user interface from the backend that stores and executes queries. They thus provide an algebraic API that allows the interchange of frontends and backends. The operators are designed to be translated into SQL. Thus, they can be implemented on either a relational system or a specialized engine.

For future, on the modeling side, work is needed to incorporate duplicates and NULL values in our model. We believe that the duplicates can be handled by treating elements of the cube as pairs consisting of an arity and a tuple of values. The arity gives the number of occurrences of the corresponding combination of dimensional values. NULLs can be represented by allowing for a NULL value for each dimension. Details of these extensions and other possible alternatives require further investigation.

On the implementation side, there are interesting research problems for implementing our model on top of a relational system as well as within a specialized engine. Although each of the proposed operators can be translated into a SQL query, simply executing this translated SQL on a relational engine is likely to be quite inefficient (see the translation of the join operation in Appendix). Corresponding to a multidimensional query composed of several of these operators, we will get a sequence of SQL queries that offers opportunity for multi-query optimization. It needs to be investigated whether the known techniques (e.g. [SG90]) will suffice or do we need to develop new techniques. Similarly, there is opportunity for research in storage and access structures and materialized views. We believe that multidimensional databases offer interesting technical challenges and at the same time have large commercial importance.

Acknowledgments. We wish to thank Mike Carey, Piyush Goel, Bala Iyer, and Eugene Shekita for stimulating discussions and probing questions.

References.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[Arb] Arbor Software Corporation, Sunnyvale, CA. *Multidimensional Analysis: Converting Corporate Data into Strategic Information.*

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 110–120, San Antonio, Texas, January 1979.

[CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Beyond decision support. *Computerworld*, 27(30), July 1993.

[CH82] A. K. Chandra and D. Harel. Horn clauses and the fixpoint query hierarchy. In *Proceedings of the 1st Symposium on Principles of Database Systems (PODS)*, pages 158–163, 1982.

[Cha96] Don Chamberlin. *Using the New DB2: IBM's Object-Relational Database System.* Morgan Kaufmann, 1996.

[CL94] R. Cicchetti and L. Lakhal. Matrix relation for statistical database management. In *Proc. of the Fourth Int'l Conference on Extending Database Technology (EDBT)*, March 1994.

[Cod70] E.F. Codd. A relational model for large shared data banks. *Comm. ACM*, 13(6):377–387, 1970.

[Cod93] E. F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E. F. Codd and Associates, 1993.

[Col95] George Colliat. OLAP, relational, and multidimensional database systems. Technical report, Arbor Software Corporation, Sunnyvale, CA, 1995.

[Eri95] Christopher G. Erickson. Multidimensionalism and the data warehouse. In *The Data Warehousing Conference*, Orlando, Florida, February 1995.

[Fin95] Richard Finkelstein. MDD: Database reaches the next dimension. *Database Programming and Design*, pages 27–38, April 1995.

[Fri95] David Friend. An introduction to OLAP: An explanation of multidimensional database terminology and technology. In *The OLAP Forum*, Orlando, Florida, February 1995.

[GBLP95] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. Technical Report MSR-TR-95-22, Microsoft Research, Advance Technology Division, Microsoft Corporation, Redmond, Washington, November 1995.

[Gut94] R. H. Guting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1996.

[IRI] IRI Software, Information Resources Inc., Waltham, MA. *OLAP: Turning Corporate Data into Business Intelligence.*

[JS96] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design, 1996. Working Paper.

[KS94] R. Kimball and K. Strehlo. What's wrong with SQL. *Datamation*, June 1994.

[Mic] Microstrategy Inc., Vienna, VA 22182. *True Relational OLAP.*

[Mic92] Z. Michalewicz. *Statistical and Scientific Databases.* Ellis Horwood, 1992.

[Mon94] Montage. *Montage User's Guide*, March 1994.

[Rad95] Neil Raden. Data, data everywhere. *Information Week*, pages 60–65, October 30 1995.

[SAG96] Sunita Sarawagi, Rakesh Agrawal, and Ashish Gupta. On computing the data cube, 1996. Working Paper.

[SG90] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

[Sho82] A. Shoshani. Statistical databases: Characteristics, problems and some solutions. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB)*, pages 208–213, Mexico City, Mexico, September 1982.

[SR96] B. Salzberg and A. Reuter. Indexing for aggregation, 1996. Working Paper.

[Sta93] Jeffery P. Stamen. Structuring databases for analysis. *IEEE Spectrum*, pages 55–58, October 1993.

[STL89] J. Srivastava, J.S.E. Tan, and V.Y. Lum. Tbsam: An access method for efficient processing of statistical queries. *IEEE Transactions on Knowledge and Data Engineering*, 1(4), 1989.

[Szy94] Andre Szykier. Fractal compression of data structures. Technical report, Cross/Z International Inc., Alameda, CA, 1994.

[TCG$^+$93] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

A. Translation to SQL

We describe in this appendix[3] how to translate the proposed operators to corresponding SQL queries, if the cube is represented as a table R. Note, a k-dimensional logical cube C that has 1/0 as its elements can be represented as a table that has k attributes and has $(d_1, \ldots, d_k)$ as a tuple if $E(C)(d_1, \ldots, d_k) = 1$. If the elements of a cube are n-tuples, then the relation has n extra attributes; with one attribute for each member of the element. Information about which attribute in R corresponds to a member of an element in cube C is kept as meta-data.

To enable this translation, we need to extend SQL to allow functions in the groupby clause and user-defined aggregate functions that could return sets in the select clause. The details of the proposed extensions are given in Section A.2. It is also shown there how functions in the groupby clause can be emulated, albeit inefficiently, in SQL with user-defined functions (e.g. DB2/CS [Cha96]).

A.1. Translations

Push. Causes another attribute to be added to the relation. The new attribute is a copy of some other attribute in the original relation.

Pull. The element-member attribute that is pulled into a dimension, namely the i^{th} member, is renamed to be a dimension name. Note, this operation is an update to the meta-data associated with the relation.

Destroy Dimension. If dimension D_i is destroyed, then the corresponding relational operation involves removing the attribute in R corresponding to dimension D_i. This operation does not introduce any duplicates in R because D_i is restricted to have only one value.

Restriction. First consider the translation of an efficient special case of the restriction operator. If predicate P is evaluable on individual values of dimension D_i, for instance $X > 20$, then restriction translates to a simple **select** clause on relation R. Namely:

select $*$ **from** R **where** $P(D_i)$.

In the more general case, P is evaluated on the set of values of dimension D_i. SQL does not support such queries. Thus, we need to extend SQL to allow user-defined aggregate functions in the **select** clause where the function may return a set of values. Thus, the following query returns all the values of dimension D_i that satisfy predicate P:

select $*$ **from** R **where** D_i **in** (**select** $P(D_i)$ **from** R) .

Function P is an aggregate function like "max", "top-5". The function P is applied to the set of values for dimension D_i and the satisfying values are picked. Note, we abuse notation and use the term P to refer to a predicate and also to an aggregate function.

[3]The paper is self-contained without this Appendix. Should space become a constraint, the conference version of the paper will not include the Appendix, but will refer to an IBM Research Report.

Join. Intuitively cube join is defined as a combination of relational join and groupby. First we define two views that capture each cube after applying the mappings to the relevant dimentions.

Define view V_r as
select $D_1, \ldots, D_{m-k-1}, f_1(D_{m-k}), \ldots, f_m(D_m), A_1, \ldots, A_r$
from R .

Define view V_s as
select $f'_1(D_{m-k}), \ldots, f'_k(D_m), D_{m+1}, \ldots, D_n, B_1, \ldots, B_p$
from S.

In both the above views the f_i and f'_i are mappings and the result of the select is a cross product of all the values for every attribute. Also, the attributes of view V_r are named the same as the attributes of R, namely $D_1, \ldots, D_m, A_1, \ldots, A_k$. Similarly the attributes of V_s are named $D_{m-k}, \ldots, D_n, B_1, \ldots, B_p$.
We then do the following relational join:

select $R.D_1, \ldots, R.D_m, S.D_{m+1}, \ldots S.D_n, f_{elem}(R.A_1, \ldots, R.A_r, S.B_1, \ldots, S.B_p)$
from V_r R, V_s S
where $(R.D_{m-k} = S.D_{m-k}$
$\ldots$
$R.D_m = S.D_m)$
groupby $R.D_1, \ldots, R.D_m, S.D_{m+1}, \ldots S.D_n$.

The above result does not include the tuples of V_r that do not match with any tuple of V_s and vice-versa. To include those values we first take a difference of the two views $(V_r - V_s)$ based on the join attributes $D_{m-k} \ldots D_m$; call this difference U_r. We now union the result of the above relational join with the following to get those tuples of V_r that do not match with any tuple of V_s:

select $R.D_1, \ldots, R.D_m, S.D_{m+1}, \ldots S.D_n, f_{elem}(R.A_1, \ldots, R.A_r, NULL, \ldots, NULL)$
from U_r R, V_s S
groupby $R.D_1, \ldots, R.D_m, S.D_{m+1}, \ldots S.D_n$.

Similarly, we union the above result include tuples of V_s not matching any tuple of V_r. Finally, we select only those tuples from the result where the function f_{elem} is not zero.

Merge. To translate merge into SQL, we need to extend SQL to include user-defined aggregate functions (e.g. [Mon94]). The translation process also needs to know the arity of the output of function f_{elem} so that the correct number of attributes can be created in the resulting tuple. This information is needed to correctly store the output of a merge operation even if the underlying store is multidimensional. Thus, the form of the output of f_{elem} is required as a part of the the function's specification. Merge is translated as follows (assuming f_{elem} outputs a p-tuple):

select $f_{merge_1}(D_1), \ldots, f_{merge_m}(D_m), D_{m+1}, \ldots, D_k, f_{elem}(A_1, \ldots, A_n)$
from R
where $f_{elem}(A_1, \ldots, A_n) \neq NULL$
groupby $f_{merge_1}(D_1), \ldots, f_{merge_m}(D_m)$.

where attributes $A_1, \ldots, A_n$ comprise the element of the cube. To conform to SQL we can rewrite the select clause to be of the form B_1 *as first_element_of(*$f_{elem}(A_1, \ldots, A_n)$*)*, B_2 *as second_element_of(*$f_{elem}(A_1, \ldots, A_n)$*)*, *etc.*, whereby attributes $B_1, B_2, \ldots$ are the new members of the elements.

A.2. Proposed Extensions to SQL

We propose an extension to the groupby construct of SQL to allow translation of the proposed multidimensional operators to SQL. Currently grouping in relational systems is restricted to be attribute based. That is, groups are formed on the basis of the values of a (set of) attribute and then an aggregate is computed using the tuples in each group.

Example A.1. Consider a database with the following tables:

$sales(S, P, A, D)$	supplier S supplied product P on date D for amount A.
$region(S, R)$	supplier S is in region R.
$category(P, C)$	product P is in product category C.

Consider a query that computes for each product the total sales in each region. This query is expressed as:

select $R, sum(A)$ **from** $sales, region$ **where** $sales.S = region.S$ **groupby** $region.R$.

However, note that the relation region represents a function that maps each supplier name to a unique region. We propose the following more intuitive rewrite of the above query:

select $region(S), sum(A)$ **from** $sales$ **groupby** $region(S)$.

where *region* is a function that maps each supplier name to the corresponding region. Strictly speaking, there is no reason why *region* needs to be a function. It could well map the same supplier to multiple regions (thereby no longer being a function but a 1-n mapping). We abuse the term function to actually refer to a mapping.

Consider another query that is not easily expressible in SQL. The query computes for each product the total sales in each quarter. This query can be represented in extended SQL as:

select $quarter(D), sum(A)$ **from** $sales$ **groupby** $quarter(D)$.

where function *quarter* maps each date into one of four quarters. There is no straightforward way of relationally expressing the above query. □

The above query represents a "roll-up" of the supplier attribute (or dimension) to the region attribute (or dimension). Gray et al. [GBLP95] also propose introducing functions in the groupby clause for implementing the "data-cube" operator. We think it appropriate to go even one step further and allow a 1-n mapping in the groupby clause because then the relational model can incorporate the equivalent of multiple levels of hierarchies. We refer to 1-n mappings as multi-valued functions. The use of multi-valued functions is illustrated below:

Example A.2. For the schema of Example A.1, consider another query that computes the running average of the sales made by each supplier for a particular year. In the extended notation using user defined functions, we can easily write this query as follows:

select $S, f(D), avg(A)$ **from** *sales* **groupby** $f(D)$.

Function (mapping) $f(D)$ maps each date into one or more groups as required by the running average intervals. For instance, if the running average was over 3 months, taken every month, then every month's sales figures would be mapped into three groups. Note, the query cannot currently be computed using SQL. □

The main point being made in the above examples is that grouping needs to be based on multi-valued functions of attributes and not just on single (or more) attributes.

Semantics. Unlike conventional grouping, a groupby clause now can increase the size of a relation if the function is a $1 \rightarrow n$ mapping and not a $1 \rightarrow 1$ function. The following example illustrates the kind of grouping that could arise:

Example A.3. Consider a relation R defined on three attributes A, B, C. Let mapping f be defined on A, and mapping g be defined on B. Let us see the groups to which tuple $t(a, b, c)$ might contribute if the query of interest is:

select $sum(C)$ **from** R **groupby** $f(A), g(B)$.

Let $f(a) = \{1, 2\}$ and $g(b) = \{\alpha, \beta\}$. Then tuple t contributes to the groups $\{(1, \alpha), (1, \beta), (2, \alpha), (2, \beta)\}$. For each of these groups, c contributes to the sum. □

Thus, the semantics are that a tuple t contributes to as many groups as the number of elements in the cross product of the result of applying the grouping functions to the attributes of t. Thus, even for multi-valued functions the semantics are well defined. In general, the groupby clause may use user-defined functions that may be multi-valued and that may be defined over an arbitrary number of attributes.

Function based grouping can be incorporated easily in hash based implementations of grouping.

Expressing the SQL Extensions in Current Systems. Some of the above suggested extensions of SQL can be simulated in current systems in somewhat round-about ways. We illustrate this for DB2/CS that allows user-defined functions in the select clause [Cha96].

Example A.4. Consider the query:

select $S, f(D), avg(A)$ **from** *sales* **groupby** $f(D)$.

We can replace the function $f(D)$ in the **groupby** clause by an occurrence of $f(D)$ in the **select** clause (of a different but equivalent) query as follows:

define view mapping as **select distinct** $D, f(D)$ **from** *sales*.

The above view contains the mapping from D to $f(D)$. This view can be joined with relation sales to obtain the original query as follows:

select $S, FD, avg(A)$ **from** *sales*, *mapping*(D, FD) **where** *supp.D* = *mapping.D* **groupby** *FD*.
□

The basic idea is that the function computation can be captured as a table via a separate select query. The resulting table can be joined with the original query to generate the necessary grouping attribute.

The above method of implementing functions in the grouping clause is not attractive because it introduces an extra join and also is not intuitive. Also, the workaround does not allow us to emulate mappings (multi-valued "functions") in the grouping clause or aggregate functions in the select clause.