An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees

Yannis Kotidis
Department of Computer Science
University of Maryland
kotidis@cs.umd.edu

Nick Roussopoulos
Department of Computer Science
Institute of Advanced Computer Studies
University of Maryland
nick@cs.umd.edu

Abstract

The Relational On-Line Analytical Processing (ROLAP) is emerging as the dominant approach in data warehousing with decision support applications. In order to enhance query performance, the ROLAP approach relies on selecting and materializing in summary tables appropriate subsets of aggregate views which are then engaged in speeding up OLAP queries. However, a straight forward relational storage implementation of materialized ROLAP views is immensely wasteful on storage and incredibly inadequate on query performance and incremental update speed. In this paper we propose the use of *Cubetrees*, a collection of packed and compressed R-trees, as an alternative storage and index organization for ROLAP views and provide an efficient algorithm for mapping an arbitrary set of OLAP views to a collection of Cubetrees that achieve excellent performance. Compared to a conventional (relational) storage organization of materialized OLAP views, Cubetrees offer at least a 2-1 storage reduction, a 10-1 better OLAP query performance, and a 100-1 faster updates. We compare the two alternative approaches with data generated from the TPC-D benchmark and stored in the Informix Universal Server (IUS). The straight forward implementation materializes the ROLAP views using IUS tables and conventional B-tree indexing. The Cubetree implementation materializes the same ROLAP views using a *Cubetree Datablade* developed for IUS. The experiments demonstrate that the Cubetree storage organization is superior in storage, query performance and update speed.

1 Introduction

Decision support applications often require fast response time to a wide variety of On-Line Analytical Processing (OLAP) queries over vast amounts of data. These queries project the data onto multidimensional planes (slices) of it and aggregate some other aspects of it. The "multi-dimensional modeling" can be realized by a Multidimensional indexing (MOLAP) typically implemented by an external to the relational system engine. The Relational OLAP approach starts off with the premise that OLAP queries can generate the multidimensional projections on the fly without having to store and maintain them in foreign storage. This approach is exemplified by a "star schema" [Kim96] linking the "dimensions" with a "fact table" storing the data. Join and bit-map indices [Val87, OQ97, OG95] are used for speeding up the joins between the dimension and the fact tables. Since data is generated on the fly, the maintenance cost of the MOLAP structures is avoided at the cost of index maintenance. This is very important because it determines the "down-time" window for an "incremental update" (refresh) of the warehouse.

However, in large data warehouses, indexing alone is often not sufficient to achieve high performance for some queries. For example, computing the sum of all sales from a fact table grouped by their region would require (no less than) scanning the whole fact table. On the other hand, some of these aggregate views can be precomputed in the ROLAP approach and stored in "summary tables". In this case, the ROLAP approach relies on selecting and materializing in summary tables the "right" subsets of aggregate views along with their secondary indexing that improves overall aggregate query processing [Rou82, BPT97, GHRU97, Gup97]. Like the MOLAP case, controlled redundancy is introduced to improve performance. A discriminating and fundamental difference remains however. The ROLAP redundancy is supported and controlled by tools that are within the relational paradigm, typically through views in SQL contrary to the arbitrary MOLAP import programs which are external to the underlying relational DBMS.

Having selected the best subset of aggregate ROLAP views, we then look how to implement these views and their indices. A somewhat surprising discovery was that a straight forward relational storage implementation of materialized ROLAP views and B-tree indexing on them is immensely wasteful on storage and incredibly inadequate on query performance and incremental update speed. We will demonstrate that conventional relational storage techniques suffer from the separation of the indexing dimensions residing in B-trees and data residing in tables. Most of the waste stems from the fact that multidimensional B-trees are independent of each other even when they share some dimensions. This causes the keys to be replicated several times in addition to the values stored in the relational storage too. Another major drawback is that in the relational storage data is typically stored unsorted which prohibits efficient merge operations during the updates.

In [RKR97] we introduced *Cubetrees*, a collection of packed R-trees [Gut84, RL85], as a "multidimensional" indexing scheme for the Data Cube. Cubetrees best features include their efficiency during incremental bulk update and their high query throughput. The bulk incremental update relies on their internal organization which is maintained sorted at all times and permits both an efficient mergepack algorithm and sequential writes on the disk. An expanded experiment [KR97] showed that Cubetrees can easily achieve a packing rate of 6GB/hour on an 2100A/275MHz Alphaserver with a single CPU and a single disk, a fairly low-end hardware platform compared with todays warehousing standards.

In this paper, we propose the use of Cubetrees as an alternative storage organization for ROLAP views and provide an efficient algorithm for mapping an arbitrary set of OLAP views to a collection of Cubetrees that achieve excellent performance. The Cubetree organization combines both storage and indexing in a *single* data structure and still within the relational paradigm. We will show that when compared to a conventional (relational) storage organization of materialized OLAP views, Cubetrees offer at least a 2-1 storage reduction, a 10-1 better OLAP query performance, and a 100-1 faster updates. We compare the two alternative approaches with data generated from the TPC-D benchmark and stored in the Informix Universal Server (IUS). The straight forward implementation materializes the ROLAP views using IUS tables which are then indexed with B-trees. The Cubetree implementation materializes the same ROLAP views using a *Cubetree Datablade* [ACT97] developed for IUS. The experiments demonstrate that the Cubetree storage organization is superior in storage, query performance and update speed.

Section 2 defines the Cubetree storage organization for materializing ROLAP aggregate views and the mapping of SQL queries to the underlying Datablade supporting the Cubetrees. Section 3 defines a TPC-D experiment and compares the straight forward relational storage implementation with the Cubetree one. The comparisons are made on all accounts, storage overhead, query performance and update speed. The conclusions are in section 4.

2 Cubetrees and Aggregate ROLAP Views

2.1 A Data Warehouse model

Consider the architecture of a typical warehouse shown in Figure 1, where data is organized through a centralized fact table F, linking several dimension tables. Each dimension table contains information specific to the dimension itself. The *fact table* correlates all dimensions through a set of foreign keys. A typical OLAP query might involve aggregation among different dimensions. The Data Cube [GBLP96] represents the computation of interesting aggregate functions over all combinations of dimension tables. Thus, the size of the cube itself is exponential in the number of dimensions in the warehouse.

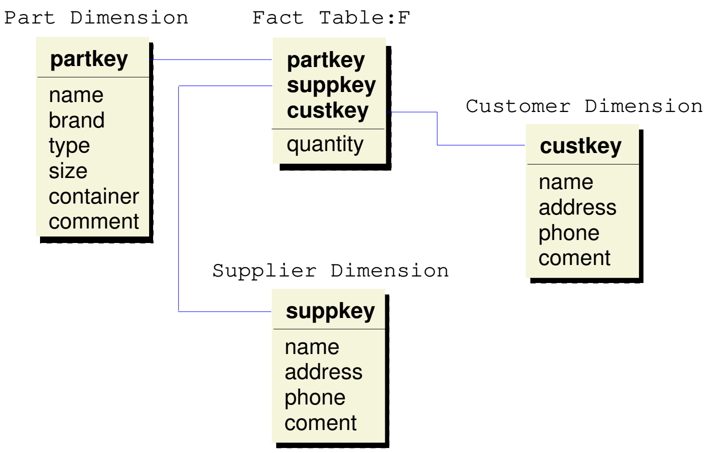

Figure 1: A simple data warehouse

Moreover, some databases contain hierarchies among each dimension attributes, such as there are along the time dimension: day → month → year. Hierarchies are very useful since they provide the means to examine the data in different levels of detail through the *drill-down* and *roll-up* operators. By drilling-down on the aggregate data the user is getting a more detailed view of the information. For example starting from the total sales per year, the user may ask for a more detailed view of the sales for the last year, grouped by month and then examine the daily volume of sales for an interesting month. Roll-up is the opposite operation where the warehouse is examined at progressively lower granularity.

A typical warehouse may contain 10 or more dimension tables, with up to 20 attributes each. The computation and materialization of all possible aggregate views, over all interesting attributes, with respect to the given hierarchies, is often unrealistic both because of the mere size of the data and of the incredibly high update cost when new data is shipped to the warehouse. Several techniques [Rou82, BPT97, GHRU97, Gup97] have been proposed to select appropriate subsets of aggregate views of the Data Cube to materialize through *summary tables*.[1] Because these views are typically very large, indexing, which adds to their redundancy, is necessary to speed up queries on then.

In the rest of this section we present an unified framework for organizing these views. Even though the examples that we use refer to the star scheme in Figure 1 there is nothing that restricts us from applying exactly the same framework to other data warehouse organizations.

2.2 Cubetrees as place holders for Multidimensional Aggregates

In figure 1 we can see an abstraction of a data-warehouse organization whose fact table correlates data from the following dimensions: part, supplier and customer. Each entry in the fact table F consists of a triplet of foreign-keys partkey, suppkey, custkey from the dimension tables and a *measure attribute* quantity. Lets assume that for query performance reasons, we decided to materialize the following views:

V_1: select partkey,suppkey,sum(quantity)
from F
group by partkey,suppkey

V_2: select part.type,sum(quantity)
from F, part
where F.partkey = part.partkey
group by part.type

V_3: select suppkey,partkey,custkey,sum(quantity)
from F
group by suppkey,partkey,custkey

By having these views materialized in the data warehouse and indexed, we are able to give quick answers to a variety of different queries without having to perform costly scans or joins over the fact table. For example view V_1 can be used to answer the query

Q_1: Give me the total sales of every part bought from a given supplier S.

Similarly V_3 can be used for answering the query:

Q_2: Find the total sales per part and supplier to a given customer C.

[1] In the remaining of this paper we will use the term views to refer to these summary tables

Notice that Q_1 can also be answered using view V_3. Even though view V_1 seems more appropriate, other parameters like the existence of an index on V_3 should be taken into account if we are aiming for the best performance.

View V_2 is an example where the grouping is done by an attribute different than the key-attribute of a dimension. Given the hierarchy part-type $\rightarrow$ part, if view V_2 were not materialized, queries would normally require a join between the fact table F and the part table. Special purpose indices [Val87, OQ97, OG95] can be used to compute these joins faster. However, such indices add to the redundancy of the warehouse and, in most cases, a materialized view, accompanied with a conventional B-tree index will perform better.

We will now proceed to show how the above set of views can be materialized using a single Cubetree. Assume that a three dimensional R-tree $R_{\{x,y,z\}}$ is used. Consider for example the tuples of view V_3. We may map suppkey to the x coordinate, partkey to y and custkey to z. In this way, every tuple t_3 of view V_3 is mapped to a point $(t_{3_x}, t_{3_y}, t_{3_z})$ on the three dimensional space of $R_{\{x,y,z\}}$. The value of the sum function is stored as the content of such point. Assuming that each coordinate is a positive (greater than zero) value Figure 2 gives a graphical representation of $R_{\{x,y,z\}}$.

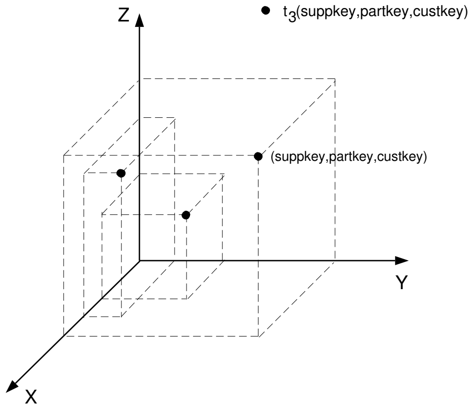

Figure 2: Mapping of view V_3

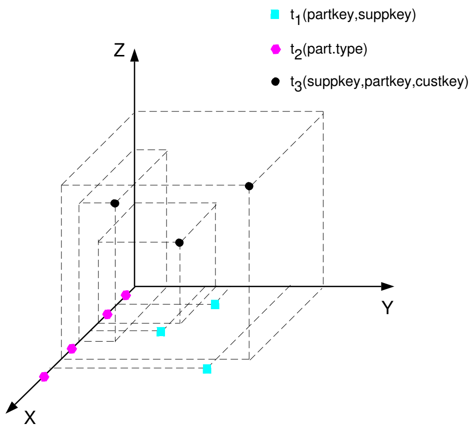

Figure 3: Cubetree organization

By considering view V_1 as a multidimensional dataset, one can also map every tuple of V_1 to a point in $R_{\{x,y,z\}}$ though the following transformation : partkey $\rightarrow$ x, suppkey $\rightarrow$ y, and using zero as the z coordinate. This transformation maps view V_1 to plane (xy) on the index. Finally the tuples of view V_2 can be mapped to the x-axis, where the part.type integer value is used as the corresponding x coordinate. This illustrates how the whole set of views fits in a single Cubetree, while every one of V_1, V_2, V_3 occupies a distinct area in the tree-dimensional index space, see Figure 3. Thus, we may use a single R-tree interface when querying any one of these views. For example Q_1 can be handled though view V_1, by searching $R_{\{x,y,z\}}$ using the slice $(x_{min}, S, 0, x_{max}, S, 0)$ as shown in Figure 4. In the same Figure the shaded plane $(x_{min}, y_{min}, C, x_{max}, y_{max}, C)$ corresponds to query Q_2.

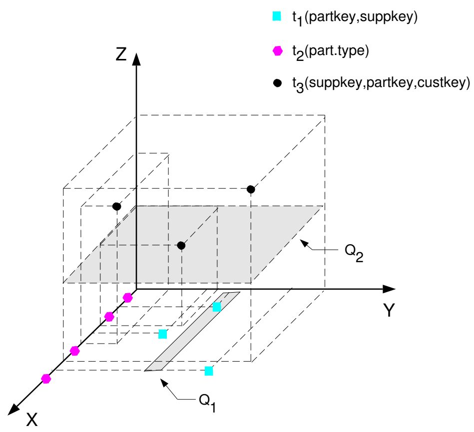

Figure 4: Queries on the views

Even though the above example is a simplified one, there is a point that is worth highlighting. In Figure 3 one can see that the same index is used for storing aggregate data for different attributes that are not necessary correlated. For example the x axis is considered to have suppkey values for view V_3 while the same axis is "named" part.type when querying V_2. This implies that the semantics on the indexing space are defined dynamically depending on which view we focus on.[2] The reason for combining multiple views on the same index is to reduce space requirements and increase the buffer hit ratio as discussed in subsection 2.4. Taking this case to the extreme, one may visualize an index containing arbitrary aggregate data, originating even from different fact tables. Hence our framework is not only applicable to the star-scheme architecture, but can be suited to a much more general data warehouse organization.

2.3 A fast algorithm for placing the ROLAP Views

Given a set $\mathcal{V}$=$\{V_1, V_2, \ldots, V_n\}$ of views one should be able to find an efficient way to map these views to a collection of Cubetrees $\mathcal{R}$ = $\{R_1, R_2, \ldots, R_m\}$. Each Cubetree $R_i = R_{\{x_1,x_2,\ldots,x_{max_i}\}}$ is *packed* having its points sorted first by x_{max_i} coordinate then by x_{max_i-1} and so on. For example $R_{\{x,y\}}$ will have its points first sorted in y, x order. This sorting is done on an external file which is then used to *bulk load* the R-tree and fill its leaf-nodes to capacity. For the purpose of this paper we propose the use of a fast algorithm that runs in linear time with the size of $\mathcal{V}$.

For each view V we will use the term *projection list* of V to refer to the list of attributes from the fact and the dimension tables that are projected by the view. For instance the projection list of view V_1 in the example of the previous section is {partkey,suppkey}. When

[2] The assumption to this scheme is that each coordinate of the index should hold attributes of the same data type.

applicable, we will use the notation $V_{\{projection\ list\}}$ to refer to a view, i.e. $V_1 \equiv V_{\{partkey, suppkey\}}$. The *arity* of view V is defined as the number of attributes in the projection list and is denoted by $|V|$, e.g $|V_{\{partkey, suppkey\}}| = 2$.

A *valid mapping* of view $V = V_{\{a_1, a_2, \ldots, a_k\}}$ to $R_{\{x,y,z,\ldots\}}$ is defined as the transformation where we store each tuple of V as a point in $R_{\{x,y,z,\ldots\}}$ by using attribute a_1 as the x coordinate, attribute a_2 as the y coordinate and so on. If the dimensionality of $R_{\{x,y,z,\ldots\}}$ is higher than the arity k of V then the unspecified coordinates of the tuple are set to zero when stored in the Cubetree. Given these definitions, the SelectMapping algorithm in Figure 5 is used to materialize $\mathcal{V}$ through a forest of Cubetrees.

SelectMapping($\mathcal{V}$=$\{V_1, V_2, \ldots, V_n\}$)
begin
  Let $maxArity = max_{V_i \in \mathcal{V}}(|V_i|)$;
  Initialize sets $S_i, i = 1, \ldots, maxArity$
  /* *Group views according to their arity*
  i.e put all views of arity 1 to S_1 e.t.c */
  for each $V_i \in \mathcal{V}$ **do**
    Let $S_{|V_i|} = S_{|V_i|} \cup \{V_i\}$;
  while $(\cup_{i=1,maxArity} S_i \neq \emptyset)$ **do**
    begin
      /* *calculate the maximum arity of*
      the views that have not been mapped so far */
      Let $arity = max_{S_i \neq \emptyset}(i)$;
      Create new R-tree $R_{\{x_1, x_2, \ldots, x_{arity}\}}$;
      /* *Pick a view from each of the sets S_j*
      and map it to R */
      for j = 1 **to** $arity$ **do**
        if $(S_j \neq \emptyset)$ **then**
          begin
            extract a view $V \in S_j$ from S_j;
            map V to R;
          end
    end /* while */
end

Figure 5: The SelectMapping algorithm

Intuitively the algorithm maps the views in such a way, that no Cubetree contains two views of the same arity. In general, one may choose to map each view to a different Cubetree or, in the other extreme, to put as much information as possible to each Cubetree. In [KR97] we present comparisons using views from a Data Warehouse with 10 dimension tables. These experiments indicate that the SelectMapping algorithm achieves the best compromise with respect to the following criteria:

- Degree of clustering inside each Cubetree
- Storage requirements
- Efficiency of bulk-loading the Cubetrees
- Query performance.

In the following subsection we present an example of using the SelectMapping algorithm and discuss the above issues.

2.4 A more complete example

Consider an example where the set of views shown in Figure 6 is chosen for materialization. This example refers to the Data Warehouse shown in Figure 1 with the addition of a fourth dimension table time. Figure 7 shows the grouping of views by the algorithm according to their arity and how they are mapped to three different Cubetrees, namely $R^1_{\{x,y,z,w\}}$, $R^2_{\{x,y,z,w\}}$, $R^3_{\{x,y\}}$.

Let us now concentrate on views V_8 and V_9 which will be stored in $R^3_{\{x,y\}}$. Table 1 and 3 show the data for these views while Tables 2 and 4 the corresponding 2-dimensional points when stored in $R^3_{\{x,y\}}$. By definition when packing $R^3_{\{x,y\}}$ we will first sort points in order y, x as in Tables 2 and 4. Figure 8 depicts the resulting Cubetree, assuming that the fan-out of the index is 3. The leaf nodes of the index contain these 2-dimensional points along with their aggregate values.[3]

partkey	sum(quantity)
4	15
2	84
3	67
1	102
6	42
5	24

Table 1: Data for view V_8

point	content
{1,0}	102
{2,0}	84
{3,0}	67
{4,0}	15
{5,0}	24
{6,0}	42

Table 2: Sorted points for view V_8

suppkey	custkey	sum(quantity)
3	1	2
1	1	24
1	3	11
3	3	17
2	1	6

Table 3: Data for view V_9

An interesting characteristic of the trees that are generated by using the SelectMapping algorithm is that the points of different views are clearly separated in the leaves of the index. For example in R^3 we can see that the index can be virtually cut in two parts, the left one used for view V_8 and the right part for view V_9. Thus, there is no interleaving between the points of different views. This is true because of the sorting and is one of the reasons for considering only sorts based on lowY, lowX and not space filling curves [FR89] when packing the trees. Clearly the same sort order is used for computing the views at creation time and during updates, as will be shown in the experiments section. One can prove that the SelectMapping algorithm picks a *minimal* set $\mathcal{R}$ of Cubetrees with such organization to store $\mathcal{V}$. The set $\mathcal{R}$ is minimal in the sense that it uses the

[3] This scheme can be extended to support multiple aggregation functions for each point.

V_1: select part.brand,count(*) from F,part where part.partkey = F.partkey group by part.brand	V_2: select suppkey,partkey,sum(quantity) from F group by suppkey,partkey
V_3: select brand,suppkey,custkey,month, sum(quantity) from F,time,part where F.timekey = time.timekey and F.partkey = part.partkey group by brand,suppkey,custkey,month	V_4: select partkey,suppkey,custkey,year, sum(quantity) from F,time where F.timekey = time.timekey group by partkey,suppkey,custkey,year
V_5: select partkey,custkey,year,sum(quantity) from F,time where F.timekey = time.timekey group by partkey,custkey,year	V_6: select custkey,avg(quantity) from F group by custkey
V_7: select custkey,partkey,avg(quantity) from F group by custkey,partkey	V_8: select partkey,sum(quantity) from F group by partkey
V_9: select suppkey,custkey,sum(quantity) from F group by suppkey,custkey	

Figure 6: Selected set of views

point	content
{1,1}	24
{2,1}	6
{3,1}	2
{1,3}	11
{3,3}	17

Table 4: Sorted points (y, x) for view V_9

least number of indices to materialize the views, while it guarantees that each and every one of them occupies a distinct continuous string of leaf-nodes in the corresponding index. By minimizing the number of Cubetrees used, we also minimize the space overhead that their non-leaf nodes add to the storage requirements. In addition, the buffer hit ratio, i.e. the probability of having the top-level pages of the trees in memory, is also increased, leading to higher performance during search.

This organization achieves excellent clustering for the tuples of every view. Moreover, there is no actual need to store the zero coordinates on the leaves. Considering the Cubetree in Figure 8, we can mark that the first two leaf nodes "belong" to view V_8 and compress the tuples by storing only the useful x-coordinate of these points on the leaves. In this way we can dramatically compress the space requirements of the Cubetrees. Our experiments indicate that due to the packing, about 90% of the pages of every index correspond to compressed leaf nodes. Since zero coordinates appear only on the few non-leaf nodes, the resulting compressed and packed Cubetrees occupy less space than an unindexed corresponding relational representation of the same views. This explains the reason why the combined indexing and materializing storage organization of the Cubetrees is more economical by a factor of more than 2-1.

3 Experiments

In order to validate our performance expectations, we used a Cubetree Datablade [ACT97] that implements the Cubetrees, the SelectMapping algorithm and the supporting routines on the Informix Universal Server. This Datablade defines a *Cubetree access method* as an alternative primary storage organization for materialized views and provides the end-user with a clean and transparent SQL interface. For all Cubetree experiments we used this interface to make fair comparisons of the Cubetree performance with a commercial industrial strength system. All experiments in this section were ran on a single processor Ultra Sparc I, with 32MB main memory, running SunOS 5.5. The experiment data was generated using the DBGEN data generation utility available in the TPC-D Benchmark.

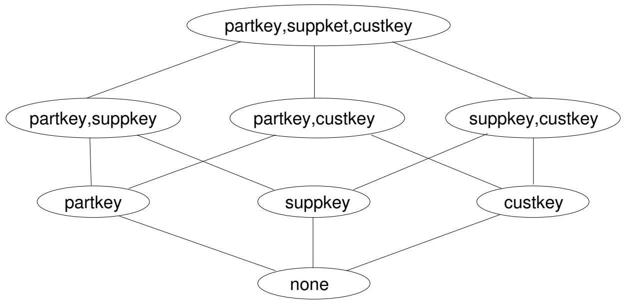

Figure 9: The Data Cube lattice

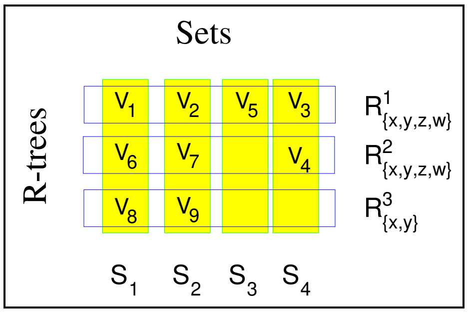

Figure 7: Cubetree selection

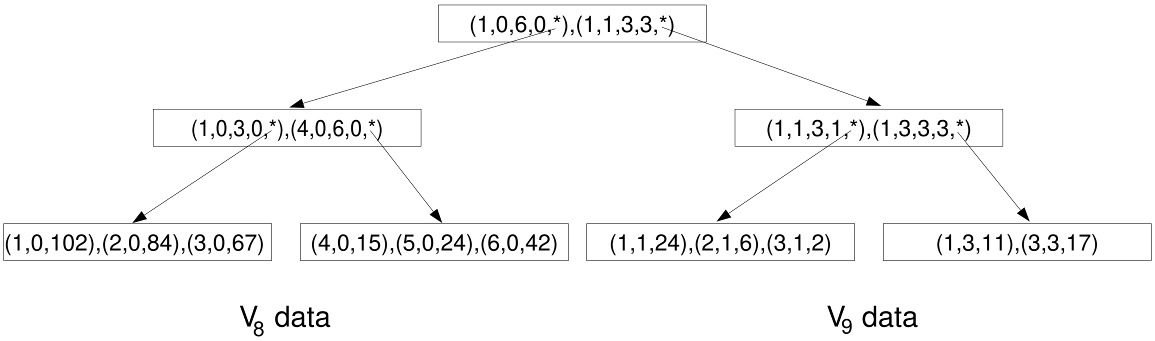

Figure 8: Content of Cubetree R^3

TPC-D models a business warehouse where the business is buying parts from a supplier and sells it to a customer. For our experiments, we will consecrate on a subset of the database that contains these three dimensions only. The measure attribute in every case is the quantity of the parts that is involved in each transaction. Figure 1 shows a simplified model of the warehouse and Figure 9 the Data Cube operator as a lattice [HRU96] in the {partkey, suppkey, custkey} dimensions. In SQL terms, each node in the lattice represents a view that aggregates data by the denoted attributes. For example the element partkey,suppkey corresponds to the view :

```
select partkey,suppkey,sum4 (quantity)
from F
group by partkey,suppkey
```

The none element in figure 9 is the *super aggregate* over all entries on the fact table. This aggregate is a scalar value and in the Cubetrees framework is mapped to the origin point (0,0,..) in one of the Cubetrees.

Since the computation of any of the views of the lattice on-the-fly is extremely time-consuming, data warehouses pre-compute a subset of them to guarantee a satisfactory query response time. The trade-off in this selection is between speed up of the queries and time to reorganize these views. Several techniques have been proposed to deal with this problem. For the purposes of our work we have used the 1-greedy algorithm presented in [GHRU97], for the view selection. This algorithm computes the cost of answering a query q, as the total number of tuples that have to be accessed on every table and index that is used to answer q. At every step the algorithm picks a view or an index that gives the greatest benefit in terms of the number of tuples that need to be processed for answering a given set of queries.

The main reason for using this algorithm is that it selects both materialized views and indices to accelerate the execution of queries. Clearly view selection without additional indexing has no value because these views would be extremely slow. Given the lattice on Figure 9, the algorithm computes the following sets of views $\mathcal{V}$ and indices $\mathcal{I}$ in a decreasing order of benefit:[5]

$$\mathcal{V} = \{V_{\{partkey,suppkey,custkey\}}, V_{\{partkey,suppkey\}}, V_{\{custkey\}}, V_{\{suppkey\}}, V_{\{partkey\}}, V_{\{none\}}\}$$
$$\mathcal{I} = \{I_{custkey,suppkey,partkey}, I_{partkey,custkey,suppkey}, I_{suppkey,partkey,custkey}\}$$

For our tests we used two configurations for storing the TPC-D data. In the first we materialized the set $\mathcal{V}$ using traditional relational tables and also created the selected set of B-trees $\mathcal{I}$. In the second, we materialized the same set $\mathcal{V}$ through a forest of Cubetrees using the SelectMapping algorithm, but no additional indexing. The packing algorithm that is implemented by the Cubetree Datablade provides a data replication scheme, where selected views are stored in multiple sort-orders, to further enhance the performance. In order to compensate for the additional indices that were used by the conventional relational scheme, we used this replication feature for the top view $V_{\{partkey,suppkey,custkey\}}$. The additional replicas that we materialized for that view were $V_{\{suppkey,custkey,partkey\}}$ and $V_{\{custkey,partkey,suppkey\}}$. In the rest of this section we make direct comparisons on query performance, space requirements and update cost, of the two storage alternatives.

[4] We have selected the sum function as a common representative of a typical aggregate operator.

[5] The notation $I_{a,b,c}$ refers to an index on view $V_{\{a,b,c\}}$ that uses as the search key the concatenation of a,b and c attributes.

3.1 Queries Description

The query model that is used by the TPC-D benchmark, involves *slice queries* [GHRU97] on the lattice hyper-space. This type of queries consist of a list of simple selection predicates between a dimension attribute and a constant value, while aggregating the measure quantity among another disjoint set of group-by attributes. For our experiments, we only considered selection predicates that use the *equal* operator. This is because the attributes are foreign keys (see Figure 9), and a generic range query, doesn't seem applicable. Consider for example the partkey, custkey element in Figure 9, the following types of queries can be requested on our model:

- Give me the total sales per part and customer
- Give me the total sales per part for a given customer C
- Give me the total sales per customer for a given part P
- Give me the total sales of a given part P to a given customer C

Notice that for the Cubetrees this kind of queries with "open" dimensions are the most expensive ones in terms of I/O overhead. This is because R-trees in general behave faster in bounded range queries [Sar97, KR97]. Thus, in a more general experiment where arbitrary range queries are allowed we expect that the Cubetrees would be even faster.

For any view V, there are $2^{|V|}$ different types of slice queries. If we sum up for all possible views, the total number of slice queries is 27. More complex queries can be constructed based on the above framework, if we allow join operators between the fact and the dimension tables. For example, by considering the hierarchy part-type → part on the part dimension, one can roll-up and examine the sales to a customer for a particular category of parts. However, the computation of such queries adds the same extra overhead for both implementations and therefore is not included in the experiment.

3.2 Initial load of the TPC-D dataset

Using the DBGEN utility of the TPC-D benchmark, we first created an instance of a 1-GB database. This dataset was then used for loading the set of views with the appropriate tuples. The total number of rows in the generated fact table of Figure 1 was 6,001,215.

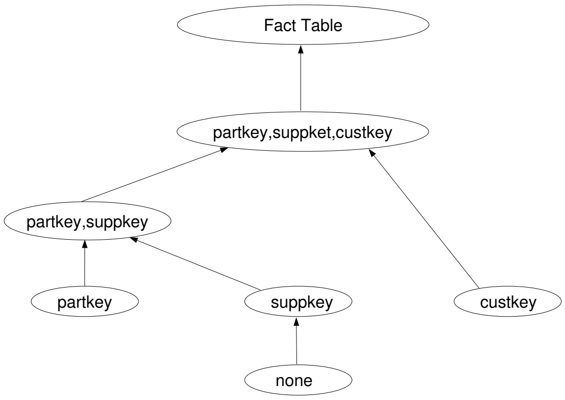

Figure 10: The dependency graph for $\mathcal{V}$

We used the lattice framework to define a *derives-from* relation [MQM97, GHRU97] between the views shown in Figure 9. For example view $V_{\{partkey\}}$ can be derived from view $V_{\{partkey,suppkey\}}$ and also from view $V_{\{partkey,suppkey,custkey\}}$. On the other hand, view $V_{\{partkey,suppkey,custkey\}}$ can be derived only from the fact table. The materialization of set $\mathcal{V}$ through typical relational tables was done by computing each view from the smallest "parent" [AAD+96], as shown in Figure 10. For speeding up the computation we issued transactions that requested exclusive locks on the tables, since concurrency is not an issue when loading the warehouse. After all the views in $\mathcal{V}$ were materialized, we created the set of indices $\mathcal{I}$ to enhance query performance. The total number of tuples in all views was 7,110,464.

Cubetree	View
$R^1_{\{x,y,z\}}$	$V_{\{partkey,suppkey,custkey\}}$
$R^1_{\{x,y,z\}}$	$V_{\{partkey,suppkey\}}$
$R^1_{\{x,y,z\}}$	$V_{\{custkey\}}$
$R^1_{\{x,y,z\}}$	$V_{\{none\}}$
$R^2_{\{x\}}$	$V_{\{suppkey\}}$
$R^3_{\{x\}}$	$V_{\{partkey\}}$

Table 5: View allocation for the TPC-D dataset

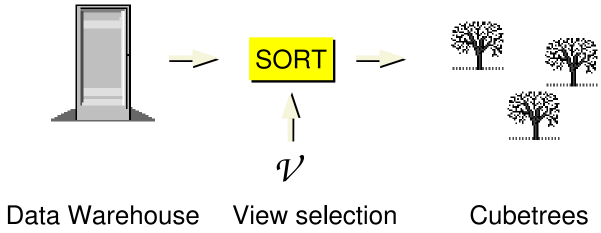

Figure 11: Loading the Cubetrees

For the Cubetree Datablade the creation of the views is slightly different. First the SelectMapping algorithm is used[6] to allocate a forest of Cubetrees to store $\mathcal{V}$. Table 5 depicts the selection of this algorithm. We then bulk-loaded the Cubetrees as shown in Figure 11. Notice that loading the Cubetrees involves a preprocessing step for sorting the tuples. However this step can be hardly considered as an overhead, since sorting is at the same time used for computing the views in $\mathcal{V}$. In our implementation we used a variation of the sort-based algorithms that are discussed in [AAD+96] for computing the lattice. The general idea of the algorithm is to minimize the processing requirements by computing an element of the cube-lattice from one of its parents as we already saw in Figure 10. Our implementation of the Cubetrees considers such optimizations, however the details of the algorithms used are beyond the scope of this paper.

Configuration	Views	Indices	Total Time
Conventional	10h 58m 23s	51m 05s	11h 49m 28s
Cubetrees	45m 04s	-	45m 04s

Table 6: Loading the databases with the TPC-D data

Table 6 shows the total time taken for loading the two configurations. The Cubetree implementation is impressively faster than the conventional approach. Cubetrees require only $1/16^{th}$ of the

[6] This step is executed transparently by the system when the user specifies the materialized set $\mathcal{V}$.

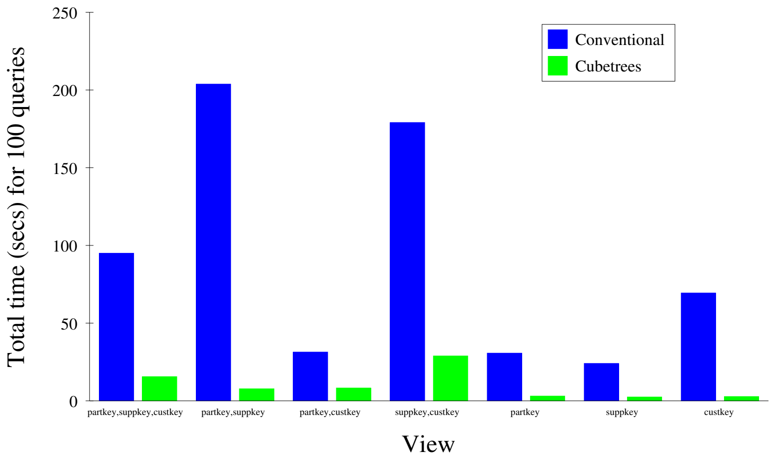

Figure 12: Querying the views

time it took the Informix Universal Server to build the conventional tables and indices. A main contributor to this differential is the internal organization of the Cubetree storage that permits sequential writes to disk during the bulk-load operation as opposed to random I/O. The relational representation of the views and their indices consumes 602MB of disk space in Informix, whereas the total disk space for the Cubetrees in this experiment was only 293MB, 51% less. This advantage is mostly due to the fact that Cubetrees are loaded in a bulk operation and packed and compressed to capacity.

3.3 Queries

For testing the performance, we used a random query generator, coded to provide a uniform selection of slice queries on the views shown in Figure 9. For each one of the 27 possible types of queries, we did a preliminary set of experiments to validate the best way that each query should be written in SQL to achieve the maximum performance under the selected set of views and indices. Consider for example query Q_1 shown in page 6. Using the materialized set $\mathcal{V}$, one may answer this query by scanning either view $V_{\{partkey,suppkey\}}$ or view $V_{\{partkeys,suppkey,custkey\}}$. In SQL terminology, we can use either of the following expressions:

```
select suppkey,sum_quantity        select suppkey,sum(sum_quantity)
from V_partkey_suppkey        or   from V_partkey_suppkey_custkey
where partkey = P                  group by partkey,suppkey
                                   having partkey = P
```

Even though view $V_{\{partkey,suppkey\}}$ seems more applicable our experiments showed that view $V_{\{partkey,suppkey,custkey\}}$ that requires an additional aggregate step for answering Q_1, is indeed faster due to the index $I_{partkey,suppkey,custkey}$.

We used the random query generator to create a set of 100 queries for each one of the views in the lattice. We assumed equal probability for all types of queries, with the exception of queries with no selection predicate, like "Give me the total quantity for all products and customers". These queries generate a very large output, which dilutes the actual retrieval cost. All queries where executed in batch through a script and their output was printed on the screen. In Figure 12, we show the total execution time of the queries along all views, for both configurations. It is interesting to notice that most of the queries ran in sub-second levels. This result validates the assumption, that view materialization significantly enhances the performance of OLAP queries. However, comparing the two approaches,

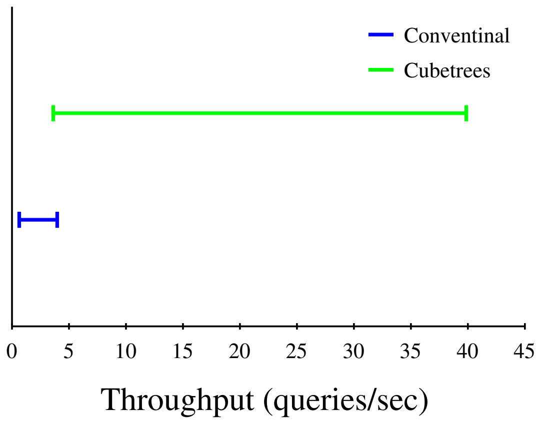

Figure 13: System throughput

we can see that the Cubetrees outperformed the conventional organization in all cases. Figure 13 depicts the minimum and maximum system throughput measured in queries/sec for both configurations. Its shows that the peak performance of the conventional approach, barely matches the system low for the Cubetrees implementation. The average query throughput was 1.1 queries/sec for the conventional views and 10.1 queries/sec for the Cubetrees, almost 10 times faster.

Figure 14 depicts how the Cubetrees scale up with larger input. This time we tested the Cubetree Datablade using a 2-GB dataset of TPC-D data. The Figure shows that query performance is practically unaffected by the larger input. The small differences are caused by the variation on the output size, which for the 2-GB case is larger.

3.4 Updating the Data Warehouse

Perhaps the most critical issue in data warehouse environments is the time to generate and/or refresh the derived data from the raw data. The mere size of them does not permit frequent re-computation. Having a set of views materialized in the warehouse adds an extra overhead to the update phase. Many optimization techniques [AAD$^+$96, HRU96, ZDN97] deal only with the initial computation of the aggregate data, while others [GMS93, GL95, JMS95, MQM97] focus on incrementally updating the materialized views.

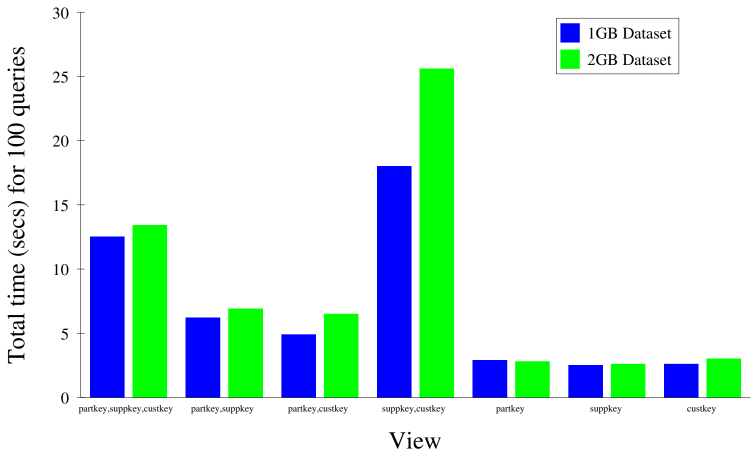

Figure 14: Scalability test (Cubetrees only)

However traditional database systems, will most probably expose their limitations when dealing with updates rates of several MBs or GBs per time unit (hour, date etc) of incoming data, in the context of a data warehouse. A typical approach that most commercial data warehouse environments follow is to rebuild most structures from scratch during an update down-time window. However by creating a new instance of the derived data every time an update increment is obtained is not only wasteful but, it may require a down-time window that leaves no time for OLAP! Thus, the only viable solution is to consider *bulk incremental operations*, where changes are deferred and applied to the warehouse in large batches. The Cubetree organization nicely fits into this framework. Figure 15 shows a bulk incremental architecture for keeping the set $\mathcal{V}$ up-to-date. At first the increment of the warehouse is obtained. Then, using the same operations that were used for the initial creation, the *delta* increment of $\mathcal{V}$ is calculated. During a final phase, this increment is *merged* with the existing Cubetrees, to create an instance of new packed Cubetrees.

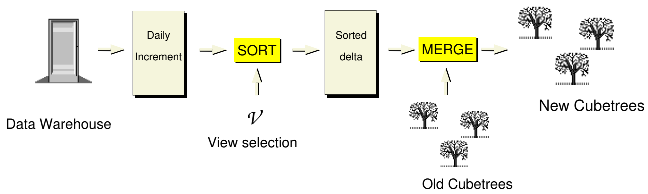

Figure 15: Bulk incremental updates of the views

To demonstrate the performance of the Cubetrees during incremental updates, we generated a 10% increment (598,964 rows) of the 1-GB TPC-D dataset, using again the DBGEN utility. For the conventional representation of the materialized views, we tested both updating the views incrementally and computing them from scratch. For all experiments we assumed a daily update and a drop-dead deadline of 24 hours to incorporate the changes to the data warehouse.

Table 7 shows the time taken to complete the update of the data warehouse in all tree cases. For all experiments we computed the time taken by the server to update the database, after the daily increment was loaded into the system. Updating the views incrementally for the conventional database did not succeed in completing the task within the one day window. This was due to the fact that updating/or inserting tuples one-at-a-time for all views in $\mathcal{V}$ adds too much of an overhead to the relational system. For every tuple in the deferred update-set, we have to perform a *look-up* operation[7] in each one of the views, to check if a corresponding aggregate does already exist and update its value, or if not insert a new row. Thus, the problem with materializing the views as relational tables is that these structures are immensely inadequate for incremental updates. Cubetrees on the other hand, maintain the stored tuples sorted at all times. This permits merge-packing of the sorted deferred set with the old values. This operation requires linear time in the total number of tuples. Furthermore, the packing algorithm that we use does only sequential writes to the disk. Thus, Cubetree organization achieves the smallest down-time by a factor of 100-1.

Method	Total Time
Incremental updates of materialized views	>24hours
Re-computation of materialized views	12h 59m 11s
Incremental updates of Cubetrees	8m 24s

Table 7: Updates on the TPC-D dataset

4 Conclusions

In this paper we proposed the use of Cubetrees as an alternative storage and indexing organization for ROLAP views. We argued and showed by experiments that the relational storage organization and indexing of such views is inadequate for the type of operations needed in the context of a Data Warehouse. Cubetrees, on the other hand, are much more economical in storage and very efficient in query execution and updates.

We have used a *Cubetree Datablade* developed for the Informix Universal Server and presented experimental results using the TPC-D benchmark for populating the same set of views using both the relational and the Cubetrees alternative framework. Our experiments first, validate the need for materializing OLAP views and second, prove that Cubetrees offer 10-1 better query performance and a 100-1 faster update speed over the relational representation. At the same time, Cubetrees provide 51% storage savings due to packing and compression.

[7] For the reported figures we used additional indexing on the conventional implementation of the views to speed up this phase.

Acknowledgments

We would like to thank ACT Inc. for providing the Cubetree Datablade code and its software libraries on which we built our experiments. We would also like to thank Informix Software, Inc. for making available the Informix Universal Server through the University Grant Program.

References

[AAD+96] S. Agrawal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *Proc. of VLDB*, pages 506–521, Bombay, India, August 1996.

[ACT97] ACT Inc. The Cubetree Datablade. August 1997.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. In *Proc. of the 23th International Conference on VLDB*, pages 156–165, Athens, Greece, August 1997.

[FR89] C. Faloutsos and S. Roseman. Fractals for Secondary Key Retrieval. *Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 247–252, March 1989.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Piramish. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proc. of the 12th Int. Conference on Data Engineering*, pages 152–159, New Orleans, February 1996. IEEE.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proceedings of the Intl. Conf. on Data Engineering*, pages 208–219, Burmingham, UK, April 1997.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 328–339, San Jose, CA, May 1995.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, Washington, D.C., May 1993.

[Gup97] H. Gupta. Selections of Views to Materialize in a Data Warehouse. In *Proceedings of ICDT*, pages 98–112, Delphi, January 1997.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. of ACM SIGMOD*, pages 205–216, Montreal, Canada, June 1996.

[JMS95] H. Jagadish, I. Mumick, and A. Silberschatz. View Maintenance Issues in the Chronicle Data Model. In *Proceedings of PODS*, pages 113–124, San Jose, CA, 1995.

[Kim96] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996.

[KR97] Y. Kotidis and N. Roussopoulos. A Generalized Framework for Indexing OLAP Aggregates. Technical Report CS-TR-3841, University of Maryland, Oct 1997.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 100–111, Tucson, Arizona, May 1997.

[OG95] P. O'Neil and G. Graefe. Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, 24(3):8–11, Sept 1995.

[OQ97] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 38–49, Tucson, Arizona, May 1997.

[RKR97] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 89–99, Tucson, Arizona, May 1997.

[RL85] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *Procs. of 1985 ACM SIGMOD Intl. Conf. on Management of Data*, Austin, 1985.

[Rou82] N. Roussopoulos. View Indexing in Relational Databases. *ACM TODS*, 7(2), June 1982.

[Sar97] S. Sarawagi. Indexing OLAP Data. *IEEE Bulletin on Data Engineering*, 20(1):36–43, March 1997.

[Val87] P. Valduriez. Join indices. *ACM TODS*, 12(2):218–246, 1987.

[ZDN97] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–170, Tucson, Arizona, May 1997.