CubiST: A New Algorithm for Improving the Performance of Ad-hoc OLAP Queries

By: Lixin Fu and Joachim Hammer

Lixin Fu, Joachim Hammer, **CubiST: A New Algorithm for Improving the Performance of Ad-hoc OLAP Queries**, ACM Third International Workshop on Data Warehousing and OLAP (DOLAP'00), Washington, D.C, USA, November, 2000, pages 72-79.

Made available courtesy of ASSOCIATION FOR COMPUTING MACHINERY (ACM): http://www.acm.org/

*****Note: Figures may be missing from this format of the document**

Abstract:

Being able to efficiently answer arbitrary OLAP queries that aggregate along any combination of dimensions over numerical and categorical attributes has been a continued, major concern in data warehousing. In this paper, we introduce a new data structure, called Statistics Tree (ST), together with an efficient algorithm called CubiST, for evaluating ad-hoc OLAP queries on top of a relational data warehouse. We are focusing on a class of queries called cube queries, which generalize the data cube operator. CubiST represents a drastic departure from existing relational (ROLAP) and multi-dimensional (MOLAP) approaches in that it does not use the familiar view lattice to compute and materialize new views from existing views in some heuristic fashion. CubiST is the first OLAP algorithm that needs only one scan over the detailed data set and can efficiently answer any cube query without additional I/O when the ST fits into memory. We have implemented CubiST and our experiments have demonstrated significant improvements in performance and scalability over existing ROLAP/MOLAP approaches.

General Terms

Algorithms, Performance, Design, Experimentation, Languages, Theory.

Keywords

Data cube, data warehouse, index structure, OLAP, query processing, query optimization.

Article:

1. INTRODUCTION

Data warehouses and related OLAP (On-line Analytical Processing) technologies [6, 7] continue to receive strong interest from the research community as well as from industry. OLAP tools present their users with a multi-dimensional perspective of the data in the warehouse and facilitate the writing of reports involving aggregations along the various dimensions of the data set [8].

There are also efforts under way to use the data warehouse and OLAP engines to perform data mining [15]. Since the underlying queries are often complex and the data warehouse database is often very large, being able to process the data quickly is an important prerequisite for building efficient decision support systems (DSS).

1.1. OLAP Queries

Users of data warehouses frequently like to visualize the data as a multidimensional "data cube" [12] to facilitate OLAP. This so- called dimensional modeling allows the data to be structured around natural business concepts, namely measures and dimensions. Two approaches to implementing data cubes have emerged: the relational approach (ROLAP), which uses the familiar "row-and-column view," and the multi-dimensional approach (MOLAP), which uses proprietary data structures to store the data cube.

OLAP queries select data that is represented in various 2-D, 3-D, or even higher-dimensional regions of the data cube, called subcubes. Slicing, dicing, rolling-up, drilling-down, and pivoting are special OLAP operators that facilitate the selection of desired subcubes. The data cube operator, which was proposed in [12], generalizes these operators using aggregation, subtotaling, cross tabulation, and grouping. In this paper, we further generalize the cube operator so that each selected dimension set in the query can be a value, a range, or an arbitrary subset of domains. We term this new operation *cube query* (a.k.a. cube operation or cubing) and provide a formalism for this class of queries in Sec. 4.1. For example, a relational warehouse containing information on car sales may have a measure called "sales" and five dimensions called "manufacturer," "color," "style," "time," and "location." Based on this example, a possible cube query involving three of the dimensions is: *"How many Toyotas have been sold in Florida and Georgia between January and March of this year?"* This cube query computes the aggregation (COUNT) over the subcube defined by three selected dimension sets (manufacturer, location, time), which represent a value (in the case of manufacturer), a partial set (in the case of location) and a range (in the case of time). The evaluation of this class of queries can be a very hard task because a large number of records may satisfy the WHERE clause conditions. It is worth pointing out that there is a subtle difference between cube queries and the more general class of OLAP queries. Cube queries return only *aggregated information* while the latter may also return *the detailed records* that satisfy the query conditions.

1.2. Proposed Solution

The focus of this paper is on describing a new, efficient approach to evaluating cube queries. We introduce a new algorithm called *CubiST* (Cubing with Statistics Trees) to evaluate cube queries more efficiently than currently possible. Unlike MOLAP, CubiST does not require the user to pre-define one or more multidimensional arrays to store the data. Unlike ROLAP, CubiST eliminates the complex process of computing new views from existing ones. Another difference is that CubiST is more efficient in evaluating queries that have range or partial set restrictions in the WHERE clause.

CubiST uses a new data structure called *Statistics Tree* (ST) to answer data cube queries. Simply speaking, a statistics tree is a multi-way tree in which internal nodes contain references to child nodes, and are used to direct the query evaluation. Each root-toleaf path in a statistics tree represents a particular subcube against the underlying data set. Leave nodes hold the statistics or histograms for the data (e.g., SUM, COUNT, MIN, MAX values) and are linked together to facilitate scanning, as is the case in the B/B^+-Tree data structure [9].

1.3. Outline of the Paper

The remainder of the paper is organized as follows. Section 2 reviews current and past research activities related to the work presented here, focusing chiefly on OLAP query processing,

indexing and view materialization in data warehouses. In section 3, we discuss the ST data structure including setup and incremental maintenance. Section 4 begins with a formal definition of cube queries, which is a subclass of OLAP queries and can be answered using CubiST. The main part of the section is devoted to explaining CubiST and its capabilities. In section 5, we outline a proposal for how CubiST can be integrated into the current data warehousing architecture. A description of our experimental CubiST system and the results of our evaluation are presented in section 6. Section 7 contains a summary and concluding remarks.

2. RELATED RESEARCH

Research related to this work falls into three broad categories: OLAP servers including ROLAP and MOLAP, indexing, and view materialization in data warehousing.

2.1. OLAP Servers

ROLAP servers store the data in relational tables using a star or snowflake schema design [7]. However, expressing OLAP queries using standard SQL can be very inefficient and many commercial ROLAP servers extend SQL to support important OLAP operations directly (e.g., RISQL from Redbrick Warehouse [30], the cube operator in Microsoft SQL Server [23]). MicroStrategy [24], Redbrick [29], Informix's Metacube [18] and Information Advantage [17] are examples of ROLAP servers.

MOLAP servers use multi-dimensional arrays as the underlying data structure for storage of the warehouse data. When the number of dimensions and their domain sizes increase, the data becomes very sparse resulting in many empty cells in the array structure (especially cells containing high dimensional data). Storing sparse data in an array in this fashion is inefficient. A popular technique to deal with the sparse data is chunking [33]. Zhao et. al. describe a single pass, multi-way algorithm that overlaps group-bys in different computations to reduce memory requirements. However, if there is not enough memory to hold the cubes in the minimum memory spanning tree, several passes over the input data are needed. To address the scalability problem of MOLAP, Goil and Choudhary proposed a parallel MOLAP infrastructure called PARSIMONY [10, 11]. Their algorithm incorporates chunking, data compression, view optimization using a lattice framework, as well as data partitioning and parallelism. Like other MOLAP implementations, the algorithm still suffers from the high I/O cost during aggregation because of frequent paging operations that are necessary to access the underlying data. Arbor Software's Essbase [3], Oracle Express [27] and Pilot LightShip [28] are based on MOLAP technology[1].

2.2. Work on Indexing

Specialized index structures are another way to improve the performance of OLAP queries. The use of complex index structures is made possible by the fact that the data warehouse is a "read-mostly" environment in which updates are performed in large batch processes. This allows time for reorganizing the data and indexes to a new optimal clustered form.

[1] The latest trend is to combine ROLAP and MOLAP. For example, in PARSIMONY, some of the operations within sparse chunks are relational while operations between chunks are multidimensional.

When the domain sizes are small, a bitmap index structure [26] can be used to help speed up OLAP queries. Simple bitmap indexes are not efficient for large-cardinality domains and large range queries. In order to overcome this deficiency, an encoded bitmap scheme has been proposed to reduce the size of the bitmaps [5]. A well-defined encoding can also reduce the complexity of the retrieve function thus optimizing the computation. However, designing well-defined encoding algorithms remains an open problem. A good alternative to encoded bitmaps for large domain sizes is the B-Tree index structure [9]. For an excellent overview of index structures in the context of OLAP refer to O'Neil and Quass [25].

2.3. View Materialization

View materialization in decision support systems refers to the pre- computing of partial query results, which may be used to derive the answers to frequently asked queries which are based on the partial results. Since it is impractical to materialize all possible views in the underlying view lattice, selecting the best views for materialization is an important research problem. For example, [16] introduced a greedy algorithm for choosing a near-optimal subset of views from a view materialization lattice based on user- specified criteria such as available space, number of views, etc. The computation of the materialized views, some of which depend on previously materialized views in the lattice, can be expensive when the views are stored on disk. More recently [14, 19, 20], for example, developed various algorithms for view selection in data warehouse environments.

Another optimization to processing OLAP queries using view materialization is to pipeline and overlap the computation of group-by operations to amortize the disk reads, as proposed by [1]. When the data set is large, external sorting or hashing on some nodes on the minimal cost processing tree is required which often needs multiple passes and is very expensive. Other related research in this area has focused on indexing pre-computed aggregates [31] and how to maintain them incrementally [22]. Also relevant is the work on maintenance of materialized views (see [21] for a summary of excellent papers) and processing of aggregation queries [13, 32].

3. THE STATISTICS TREE

A *Statistics Tree* (ST) is a multi-way tree structure, which is designed to hold aggregate information (e.g., SUM, AVG, MIN, MAX) for one or more attributes over a set of records (e.g., tuples in a relational table). In Sec. 4.2 we show how we use statistics trees to evaluate OLAP queries over large data sets.

3.1. Data Structure and Setup

Assume R is a relational table with attributes $A_1,A_2,...,A_k$ with cardinalities $d_1,d_2,...,d_k$ respectively. In keeping with standard OLAP terminology, we refer to the attributes of R as its dimensions. The structure of a k-dimensional statistics tree for R is determined as follows:

- The height of the tree is $k+1$, its root is at level 1.
- Each level in the tree (except the leaf level) corresponds to an attribute in R.
- The fan-out (degree) of a branch node at level j is d_j+1, where $j = 1,2,...,k$. The first d_j pointers point to the subtrees which store information for the j^{th} column value of the input data. The $(d_j + 1)^{th}$ pointer is called *star pointer* which leads to a region in the tree where this domain has been "collapsed," meaning it contains all of the domain values for this dimension. This

"collapsed" domain is related to the definition of super-aggregate (a.k.a "ALL") presented in Gray et al. [12].

- The s*tar child* of a node is the child node that is referenced by its star pointer.
- The leaf nodes at level $k+1$ contain the aggregate information and form a linked list.

Figure 1 depicts a sample statistics tree for a relational table with three dimensions A_1, A_2, A_3 with cardinalities $d_1=2$, $d_2=3$, and $d_3=4$ respectively. The letter 'V' in the leaf nods indicates the presence of an aggregate value (as opposed to the interior nodes which contain only pointer information). As we can see in this figure, the node at the root, for example, which represents dimension A1, has room for two dimension pointers (since d1=2) plus the star pointer. A node at level 2 has room for three dimension pointers ($d_2=3$) plus the star pointer and so forth. A star child at level *j* represents all data records which have an "ALL" value in the $(j-1)^{th}$ dimension. We will illustrate how ST represents the data from our motivating example in the next section.

Although this data structure is similar to multi-dimensional arrays and B-trees, there are significant differences. For example, a multi-dimensional array does not have a star-pointer although the extension "ALL" to the domain of a dimension attribute has been used in the query model and summary table described in [12]. The implementation of fully materialized cubes using multidimensional arrays is awkward at best since all combinations of the dimensions must be separately declared and initialized as multidimensional arrays. When the number of dimensions is not known in advance (i.e. provided as an input parameter instead of a fixed integer), the set-up of the arrays is even harder. ST, on the other hand, represents all related cubes as a single data structure and can easily deal with the dimensionality using parameter. Hence one can regard the Statistics Tree as a generalization of the multi-dimensional array data structure.

Looking at the B^+-Tree, it also stores data only at the leaf nodes which are linked together to form a list. However, a B-tree is an index structure for one attribute (dimension) only, as opposed to our ST, which can contain aggregates for multiple dimensions. In addition, in the B-Tree, the degree of the internal nodes is restricted. The Statistics Tree on the other hand is naturally balanced (it is always a full tree) and its height is based on the number of dimensions but independent of the number of records or the input order of the data set.

3.2. Populating and Maintaining ST

In order to explain how Statistics Trees are updated, we use the COUNT aggregate to demonstrate how to populate a Statistics Tree to help answer OLAP queries involving COUNT aggregations. STs for other aggregate operations such as SUM, MIN, MAX, etc. can be updated in a similar fashion. In those cases, the ST structure will be the same except that the contents of the leaves reflect the different aggregate operators. Based on the number of dimensions and their cardinalities in the input data, the statistics tree is set up as described in the previous section by creating the nodes and pointers that form the entire tree structure. Initially, the count values in the leaf nodes are zero.

Next, we scan the relational data set record by record, using the attribute values to update the aggregates in the statistics tree with the recursive procedure *update_count()* shown in pseudo-code in Figure 2. Please note that in order to accommodate other aggregate operators, the update

routine has to be modified slightly (lines 3 and 4 of Figure 2). For each record in the input set, the update procedure descends into the tree as follows: Starting at the root (level 1), for each component *xi* of the input record $x=(x_1,x_2,...,x_k)$, where *i* indicates the current level in the tree, follow the xi^{th} pointer as well as the star pointer to the two nodes at the next-lower level. When reaching the nodes at level *k*, increment the count values of the two leaves following the x_k^{th} pointer and the star pointer. Repeat these steps for each record in the input until all records have been processed in this fashion.

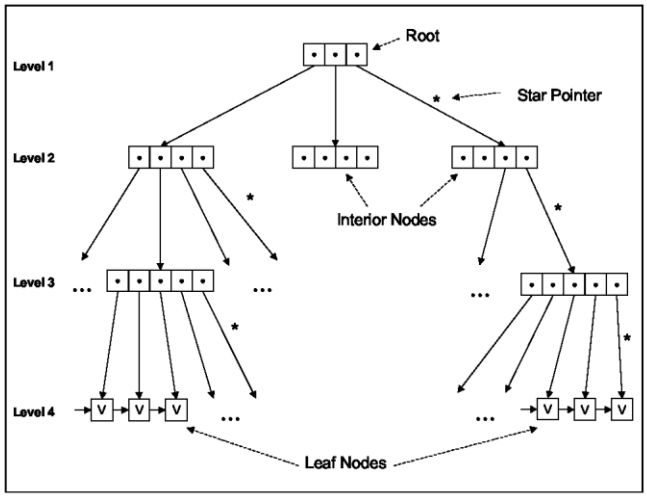


Figure 1: Statistics tree for data set with 3 dimensions.

In Figure 2, line 1 is the signature of the recursive function *update _count()*. Node *n* is a pointer to the current node in the tree. The one-dimensional array *x* contains the current input record from the table. The third parameter, *level*, indicates the level of node *n*. Lines 2 to 5 handle the base case which occurs at level *k*. Lines 12 to 14 scan the input data set record by record to update the aggregation information in the leaves using procedure *update _count()*. Note that lines 4 and 8 handle the case of the "ALL" values.

```
1   update_count(Node n, record x, int level) {
2    IF level == k THEN
3      increase count field for x_k^th child of Node n;
4      increase count field for star child;
5      return;
6    level := level + 1;
7    update_count(x_level-1^th child of n, x, level);
8    update_count(star child of n, x, level);
9   }
10  // end update_count
11
12  WHILE ( more records ) DO
13  read next record x;
14  update_count(root,x,1);
```

Figure 2: Algorithm for updating aggregates in a ST.

In order to illustrate the insert procedure, let us continue with our example and assume that the first record in the input set is (1,2,4). An interpretation of this tuple in terms of our car sales example could be the sale of a Toyota car (value 1 dimension 1) in February (value 2 dimension 2) in Florida (value 4 dimension 3). In a sense, we have devised a mapping from the underlying domain values (which may be strings or real numbers) to integers representing them.

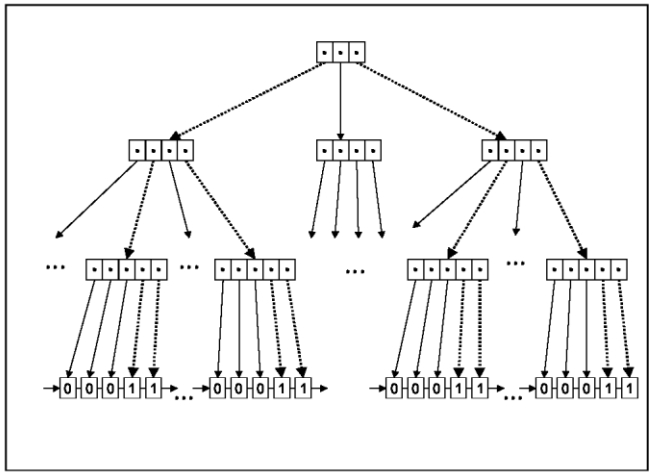


Figure 3: Statistics tree after processing input record (1,2,4).

Figure 3 depicts the contents of the statistics tree after processing the input record. The update paths relating to this record are indicated as dashed lines. Pointers and nodes, which do not play a role in this update, are omitted from the picture to improve readability. Since the first component value of x is 1 (x_1=1), we follow the first pointer as well as the star pointer to the first and third nodes in the second level. From each of these nodes, we follow the second (x_2=2) and star pointers to the third level in the tree. From here we follow the fourth (x_3=4) and star pointers to the leaf nodes where we increment the count values by one. Note, for example, that the star child of the root (i.e., the rightmost node at level 2) represents a node along the search path aggregating all the sales involving ALL car manufacturers. Once all records have been processed, the Statistics Tree is ready to answer cube queries.

4. QUERY ANSWERING ALGORITHM

Before we can discuss our CubiST query-answering algorithm, we present a formal representation for cube queries, which are the focus of this investigation.

4.1. Formal Representation for Cube Queries

The conceptual model, which is underlying the theory in this paper, is the multi-dimensional database [2] also known as the data cube model, first described in [12]. The cube can be viewed as a k-dimensional array, where k represents the number of dimensional attributes that have been selected (together with the measures) from the underlying data warehouse to track the metrics of interest. The cells of the cube contain the values of the measure attributes for the corresponding combination of dimensions.

Definition: A *cell* of a *k*-dimensional data cube with attributes A_1 , A_2 ,..., A_k with cardinalities d_1, d 2 ,..., d_k respectively, is the smallest full-dimensional cube[2] seated at a point $P = (x_1, x_2, ..., x_k),\ x_j \in [1..d_i]$. The total number of cells in a *k*-dimensional data cube is $\prod_{i=1}^{k} d_i$

Definition: A *Cube Query q* is an aggregate operation that is performed on the cells of a *k*-dimensional data cube. The query *q* is a tuple of the form: $_q = (si_1 , si_2 ,..., si_r)$, where $\{i_1, i_2, ..., i_r\} \subseteq \{1,2,...,k\}$ and *r* is the number of dimensions specified in the query. Each selection S_{ij} can be one of the following (in decreasing order of generality). Let $w = i_j$:

1. A *partial* selection, $\{t_1, t_2, ..., t_r\},\ 2 \le r < d_w,\ t_i \in \{1,2,...,d_w\}$, specifying any subset of domain values for dimension i_j.
2. A *range* selection [a,b], specifying a contiguous range in the domains of some of the attributes, $a, b \in [1..d_w],\ [a,b] \neq [1, d_w]$.
3. A *singleton* value a, $a \in [1..d_w]$.

When $r \neq k$, some of the dimensions may be collapsed. If $r = k$, we say the query is in *normal form*. A query can be transformed into its normal form by adding collapsed dimensions using "ALL" values.

Definition: A query is *partial* iff $\exists j$ such that s_{ij} is a partial selection. By convention, the integer values in the partial set are not contiguous. Consequently, a query is a *range query* iff $\exists j$ such that s_{ij} is a range selection. Finally, a query is a *singleton query* iff $\forall j\ s_{i_j}$ is a singleton value including the "ALL" value.

It is worth noting that a singleton query represents a data cube.

Definition: The *degree r* of a query *q* is the number of dimensions specified in the query (not including "ALL" values). When the degree of a query is small (≤4), the query is of *low dimensionality*. Continuing our car sales example, assume that the domain values for the different measures are as follows: {Ford, GM, Honda, Toyota} for manufacturer, {blue, red, white, black} for color, {sports car, sedan, SUV} for type, {Jan, Feb, ..., Dec} for time, and {Florida, Alabama, Georgia, South Carolina, Tennessee} for location. A formal representation of the cube query "How many Toyota cars have been sold in Florida and Georgia between January and March of this year?" is $q = (s_1, s_4, s_5) = (4, [1,3], \{1,3\})$. In this example, *q* is a partial query of degree r = 3 and its normal form is (4, ALL, ALL, [1 3], {1,3}).

4.2. A Recursive Cube Query Algorithm

The Statistics Tree is capable of answering complex cube queries. Figure 4 shows a sketch of our recursive query-answering algorithm, called CubiST. CubiST assumes that the query is in its normal form. Suppose the input query *q* has the

[2] A cube containing no "ALL" values. Otherwise, the cube is called *supercube*.

form: $q = \{z_{i,j} \in [1..d_i] \mid i = 1,2,...,k, j = 1,2,...,d_i + 1\}$. Here z_{ij} refers to the i^{th} selected value of the $_it''$ dimension. To compute the aggregate for a subtree rooted at node *n*, follow the pointers corresponding to the selected values in the current level to the nodes on the lower level and sum up their returned values (lines 6, 7). As base case (level k), return the summation of the aggregates in the selected leaves (lines 2-4). Line 12 invokes CUBIST on the root to return the final result. If $j = d_i + 1$ or $j = 1$ (singleton), we only need to follow one pointer to the node on the next lower level.

```
1 INT cubist(Node n, query q, int level) {
2   IF level == k, THEN
3     count := count summation of z_{k,1}^th , z_{k,2}^th ,...child of n;
4     return count;
5  level := level+1;
6  count := cubist(Z_{level-1,1}^th child of n, q, level)+
7        cubist(Z_{level-1,2}^th child of n, q, level)+ ...
8
9   }
10   end cubist
11
12 Total counts in the query regions := cubist(root,q,1);
```

Figure 4: Recursive algorithm for evaluating cube queries.

CubiST is particularly useful in data marts where the dimensionality of the data is lower, yet the number of tuples is large enough to warrant the use of special query processing algorithm such as CubiST. CubiST is also well suited for large data warehouses (where we have many dimensions with large domain sizes) to provide a high-level "analytical view" over the data in order to decide on which subcube to focus (i.e., start drilling down). With the continued increase in available memory size, CubiST and ST have become a superior alternative to existing methodologies. As far as the run-time of CubiST is concerned, for a singleton query it is linear to the height of the ST (which is constant when the number of dimensions is constant). In the case of an arbitrary cube query it may involve traversing most of the ST in the worst case. The additional space cost of STs (for interior nodes) is a fraction of the cost for storing the leaves.

5. A DATA WAREHOUSE ARCHITECTURE FOR CUBIST

So far, we have presented our algorithm and data structure to evaluate cube queries. However, in many cases, having only one ST to answer cube queries is not feasible, especially when the queries involve many dimensions with large domain sizes. It is noteworthy that we do *not* assume using a single in-memory ST to answer all possible cube queries; however, we do assume that a particular ST corresponding to a group of attributes is kept in memory. In this section, we briefly describe how CubiST can be integrated into a data warehousing architecture to support efficient cube query processing over large data sets in real world applications using *multiple* STs. This is shown in Figure 5. In this figure, the familiar back-end, consisting of source data extractors, warehouse integrator, and warehouse is responsible for accessing, cleaning, integrating, and storing the potentially heterogeneous data from the underlying data sources. Data in the warehouse is typically stored in the form of a star or snowflake schema with materialized views and indexes as optional support structures.

5.1. System Architecture

In order to make use CubiST, the data on which to perform the cube queries has to be transformed into a single table. In such a "single-table schema," attributes from the same source (i.e. the same dimension table in a star schema) are grouped together, since queries are more likely to request aggregates over attributes from the same group. In order to produce the data, records from the existing star schema are joined across the foreign keys in the fact table. Each joined record is only used once during the creation of the ST and does not have to be stored.

STs are managed and stored in the *ST Repository*, which is shown in the upper-left hand corner of Figure 5. Using the *ST Manager*, shown directly to the right, STs are selected from the repository to match the incoming cube queries. If there is no matching ST, a new ST is generated using the algorithms in Sec. 3.

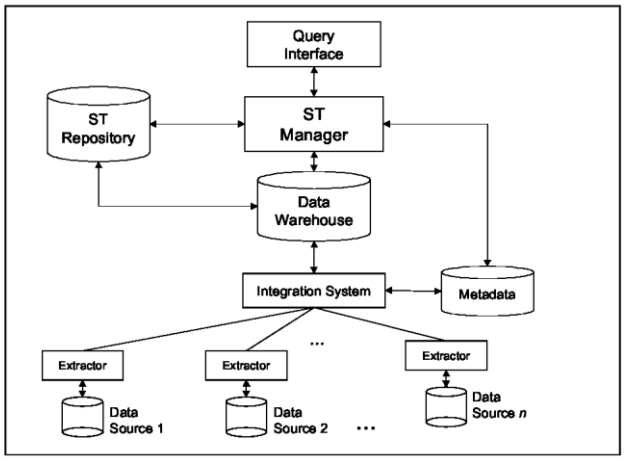


Figure 5: Warehouse architecture with CubiST.

5.2. The ST Manager

Once the single table schema has been created, one can set up an initial ST for each attribute group. The STs are placed in the ST Repository and the attribute information in the STs is stored in a metadata repository. The choice as to which ST to build depends on the domain sizes and the "closeness" among the attributes: The closer the attributes the higher the likelihood that they will appear together in a query. Domain knowledge can provide hints regarding the "closeness" of attributes.

When a query arrives, the ST Manager checks if there is an ST that matches the query. If there is, the query is evaluated using this ST. Otherwise, a new ST is set up and placed into the ST repository. The characteristics of the ST (i.e., number of dimensions, their cardinalities, underlying aggregate operators) are stored in the metadata repository. This metadata also includes information about the usage statistics of the STs. In order to reduce the cost of paging STs in and out of memory, we envision an optimized management algorithm which only transmits the leaves of the ST trees to and from the ST Repository. The internal nodes can be generated in memory without extra I/O operations. In this way, the I/O complexity of STs is the same as that of multidimensional arrays.

6. EVALUATION OF CUBIST

We have implemented the ST and CubiST and conducted a series of experiments to validate the superior performance of our algorithm. The testbed consists of a SUN ULTRA 10 workstation running Sun OS 5.6 with 90MB of available main memory. The ST and CubiST were implemented using JDK 1.2.

6.1. Synthetic Data Sets

Our experiments consist of a simulated warehouse that is used to provide the data for the incoming cube queries. Specifically, we assume that our warehouse consists of the single table schema outlined in Sec. 5 and is defined by the following parameters: r is the number of records and k is the number of dimensions with cardinalities d_1, d_2, ..., d_k respectively. We use a random number generator to generate a uniformly distributed array with r rows and k columns. The elements in the array are in the range of the domain sizes of the corresponding columns. After the array is generated, it is stored on disk and serves as input to CubiST.

Table 1: Characteristics of the data sets used in the simulation

set	size	r	domain sizes
D1			
	20 K	1 K	10, 10, 10, 10, 10
	200 K	10 K	10, 10, 10, 10, 10
	2 M	100 K	10, 10, 10, 10, 10
D2			
	1.6 M	100 K	10, 10, 20, 81
	2 M	100 K	10, 10, 20, 9, 9
	2.4 M	100 K	10, 10, 5, 4, 9, 9
	2.8 M	100 K	10, 10, 5, 4, 9, 3, 3

We have generated synthetic data sets D1 and D2 as shown in Table 1. The data sets vary in the number of records and number of dimensions respectively. In order to evaluate CubiST we are comparing its setup and response times with those of two other frequently used techniques: A simple query evaluation technique, henceforth referred to as *scanning*, and a bitmap-based query evaluation algorithm, henceforth referred to as *bitmap*. The scanning algorithm computes the aggregates for each cube query by scanning the entire data set each time. The bitmap-based query evaluation algorithm uses a bitmap index structure to speed up the processing of cube queries without touching the original data set (assuming the bit vectors have already been created).

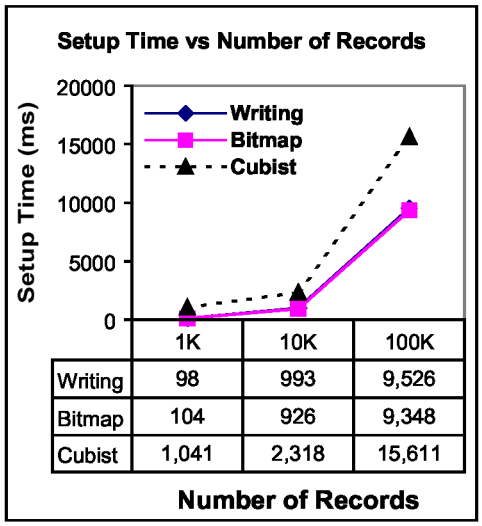


Figure 6: Setup time vs. number of records.

6.2. Varying Number of Records

In this first series of experiments which uses data set D1, the number of dimensions and corresponding domain sizes are fixed while we increase the size of the input set. The goal is to validate our hypothesis that CubiST has excellent scalability in terms of the number of records.

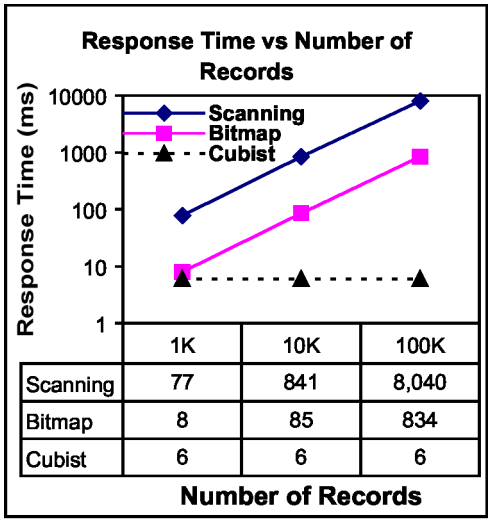
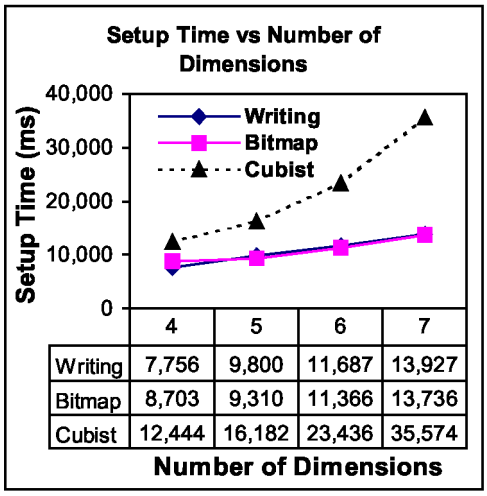


Figure 7: Response time vs number of records.

Figures 6 and 7 show the performance of the three algorithms for the cube query *Q*1: (s_1,s_2,s_3,s_4,s_5) = ([0,5],[0,5],[0,5],[0,5],[0,5]). Since there is no setup time for the scanning algorithm, we simply measured how long it would take to write the data out to disk as a reference value. We call this set of measurements "writing" in the first row in Figure 6. Although the setup time for CubiST is larger than that of both bitmap and scanning (due to the insertion operation for each record), its response time is much faster and *almost independent* of the number of records.

6.3. Varying Number of Dimensions

In this set of experiments, we used data set D2 to investigate the behavior of CubiST with respect to varying numbers of dimensions. Specifically, we varied the number of dimensions from 4 to 7, but maintained the same data density (100,000/(10x10x20x81) = 61.73%) and fixed the number of records (100,000). The input query $Q2$ is defined as follows: $Q2 = (s_1,s_2,s_3,s_4) = ([0,8], [1,8], [1,4], [1,3])$.



	4	5	6	7
Writing	7,756	9,800	11,687	13,927
Bitmap	8,703	9,310	11,366	13,736
Cubist	12,444	16,182	23,436	35,574

Figure 8: Setup Time vs number of dimensions.

As expected, the time for writing the data to disk increases as the size of the data set increases. As for the bitmap approach, the setup time for bitmaps is dominated by the read time since in D2 all the bitmaps fit into memory. As for CubiST, since number of leaves of the ST increases from 208,362 (11x11x21x82) to 363,000 (11x11x6x5x10x10), the ST actually increases with number of dimensions. However, the number of full dimensional cells remains the same. As a result, the setup time of CubiST increases as shown in Figure 8. However, when the size of the data set increases to the point when the I/O operations begin to dominate, the gap in setup time between Bitmap and CubiST starts to decrease.

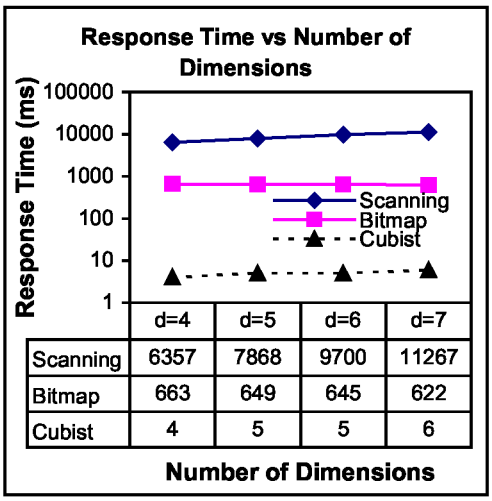
When the number of dimensions increases, the resulting ST becomes deeper. As a result, the response time of CubiST increases marginally (Figure 9). For data sets with large cardinalities, a bitmap is not space efficient and the performance will degrade quickly. Although the bitmaps fit into memory, we observe that the response time decreases as the cardinalities decrease (since the number of dimensions increases).

In addition, we conducted experiments to validate its behavior under varying query complexity and varying the domain size. The results are similar in that they prove the superiority of CubiST over traditional ROLAP approaches. However, due to space limitations, we omit a detailed description of the experiments.

7. CONCLUSION

We have presented the Statistics Tree data structure and a new algorithm, called CubiST, for efficiently answering cube queries, which are a generalization of the cube operator. In CubiST, records are aggregated into the leaves of a Statistics Tree without storing detailed input data. During the scan of the input, all views are materialized in the form of STs, eliminating the need

to compute and materialize new views from existing views in some heuristic fashion based on the lattice structure.



	d=4	d=5	d=6	d=7
Scanning	6357	7868	9700	11267
Bitmap	663	649	645	622
Cubist	4	5	5	6

Figure 9: Response Time vs number of records.

As we have shown in this paper, the advantages of CubiST are numerous and significant:

1. Fast: The initialization of the ST needs only one read pass over the data set; all subsequent computations can be done internally on the ST without referring to the original data.
2. Space Efficient: Keeping summary information in an ST requires a fraction of the space that would otherwise be needed to store the detailed data set.
3. Incremental: When updates are made to the original data set, the aggregation information in an ST can be maintained incrementally without rebuilding the entire ST from scratch.
4. Versatile: CubiST supports generalized ad-hoc cube queries on any combination of dimensions (numerical or categorical); queries may involve equality, ranges, or subsets of the domains.
5. Scalable: The number of records in the underlying data sets has almost no effect on the performance of CubiST. As long as an ST fits into memory, the response times for cube queries over a data set containing 1 million or 1 billion records are almost the same.

We have implemented CubiST and validated its superior performance and scalability through experiments. In order to make STs and CubiST usable in real-world applications, we have also proposed a new data warehousing architecture that incorporates our technology. In the future, we plan to explore ways of partitioning large STs across multiple processors and parallelize query evaluation. We expect that such a parallel version will improve query processing even further. We are also pursuing the development of new data mining algorithms on top of the data warehouse that can benefit from the abilities of CubiST.

REFERENCES

1. S. Agarwal, R. Agrawal, P. Deshpande, J. Naughton, S. Sarawagi, and R. Ramakrishnan, "On The Computation of Multidimensional Aggregates," in *Proceedings of the International Conference on Very Large Databases*, Mumbai (Bomabi), India, 1996.

2. R. Agrawal, A. Gupta, and S. Sarawagi, “Modeling Multidimensional Databases,” in *Proceedings of the Thirteenth International Conference on Database Engineering*,
3. Birmingham, U.K., 1997.
4. Arbor Systems, “Large-Scale Data Warehousing Using Hyperion Essbase OLAP Technology,” Arbor Systems, White Paper, www.hyperion.com/whitepapers.cfm.
5. S. Berchtold and D. A. Keim, “High-dimensional index structures database support for next decade's applications (tutorial),” in *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, Seattle, WA, pp. 501, 1998.
6. C. Y. Chan and Y. E. Ioannidis, “Bitmap Index Design and Evaluation,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, WA, pp. 355-366, 1998.
7. S. Chaudhuri and U. Dayal, “Data Warehousing and OLAP for Decision Support,” *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **26**:2, pp. 507-508, 1997.
8. S. Chaudhuri and U. Dayal, “An Overview of Data Warehousing and OLAP Technology,” *SIGMOD Record*, **26**:1, pp. 65-74, 1997.
9. E. F. Codd, S. B. Codd, and C. T. Salley, “Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate,” Technical Report, www.arborsoft.com/OLAP.html.
10. D. Comer, “The Ubiquitous Btree,” *ACM Computing Surveys*, **11**:2, pp. 121-137, 1979.
11. S. Goil and A. Choudhary, “High Performance OLAP and Data Mining on Parallel Computers,,” *Journal of Data Mining and Knowledge Discovery*, **1**:4, pp. 391-417, 1997.
12. S. Goil and A. Choudhary, “PARSIMONY: An Infrastructure for Parallel Multidimensional Analysis and Data Mining,,” *Journal of Parallel and Distributed Computing*, to appear.
13. J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross- Tab, and Sub-Totals,” *Data Mining and Knowledge Discovery*, **1**:1, pp. 29-53, 1997.
14. A. Gupta, V. Harinarayan, and D. Quass, “Aggregate-query Processing in Data Warehousing Environments,” in *Proceedings of the Eighth International Conference on Very Large*
15. *Databases*, Zurich, Switzerland, pp. 358-369, 1995.
16. H. Gupta and I. Mumick, “Selection of Views to Materialize Under a Maintenance Cost Constraint,” in *Proceedings of the International Conference on Management of Data*, Jerusalem, Israel, pp. 453-470, 1999.
17. J. Han, “Towards On-Line Analytical Mining in Large Databases,” *SIGMOD Record*, **27**:1, pp. 97-107, 1998.
18. V. Harinarayan, A. Rajaraman, and J. D. Ullman, “Implementing data cubes efficiently,” *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **25**:2, pp. 205-216, 1996.
19. Information Advantage, “Business Intelligence,” White Paper, 1998, www.sterling.com/eureka/.

20. Informix Inc., “Informix MetaCube 4.2, Delivering the Most Flexible Business-Critical Decision Support Environments,” Informix, Menlo Park, CA, White Paper, www.informix.com/informix/products/tools/me tacube/datasheet.htm.
21. W. Labio, D. Quass, and B. Adelberg, “Physical Database Design for Data Warehouses,” in *Proceedings of the International Conference on Database Engineering*, Birmingham, England, pp. 277-288, 1997.
22. M. Lee and J. Hammer, “Speeding Up Warehouse Physical Design Using A Randomized Algorithm,” in *Proceedings of the International Workshop on Design and Management of data Warehouses (DMDW '99)*, Heidelberg, Germany, 1999,
23. D. Lomet, *Bulletin of the Technical Committee on Data Engineering*, **18**, IEEEE Computer Society, 1995.
24. Z. Michalewicz, *Statistical and Scientific Databases*, Ellis Horwood, 1992.
25. Microsoft Corp., “Microsoft SQL Server,” Microsoft, Seattle, WA, White Paper, www.microsoft.com/federal/sql7/white.htm.
26. MicroStrategy Inc., “The Case For Relational OLAP,” MicroStrategy, White Paper, www.microstrategy.com/publications/whitepaper s/Case4Rolap/execsumm.htm.
27. P. O'Neil and D. Quass, “Improved Query Performance with Variant Indexes,” *SIGMOD Record (A CM Special Interest Group on Management of Data)*, **26**:2, pp. 38-49, 1997.
28. P. E. O'Neil, “Model 204 Architecture and Performance,” in *Proc. of the 2nd International Workshop on High Performance Transaction Systems*, Asilomar, CA, pp. 40-59, 1987.
29. Oracle Corp., “Oracle Express OLAP Technology,” www.oracle.com/olap/index.html.
30. Pilot Software Inc., “An Introduction to OLAP Multidimensional Terminology and Technology,” Pilot Software, Cambridge, MA, White Paper, www.pilotsw.com/olap/olap.htm.
31. Redbrick Systems, “Aggregate Computation and Management,” Redbrick, Los Gatos, CA, White Paper, www.informix.com/informix/solutions/dw/redb rick/wpapers/redbrickvistawhitepaper.html.
32. Redbrick Systems, “Decision-Makers, Business Data and RISQL,” Informix, Los Gatos, CA, White Paper, 1997.
33. 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, pp. 414- 423, 1989.
34. W. P. Yan and P. Larson, “Eager Aggregation and Lazy Aggregation,” in *Proceedings of the Eighth International Conference on Very Large Databases*, Zurich, Switzerland, pp. 345-357, 1995.
35. [33] Y. Zhao, P. M. Deshpande, and J. F. Naughton, “An Array- Based Algorithm for Simultaneous Multidimensional Aggregates,” *SIGMOD Record (A CM Special Interest Group on Management of Data)*, **26**:2, pp. 159-170, 1997.