CUBIST: A New Approach to Speeding Up OLAP Queries in Data Cubes

Lixin Fu and Joachim Hammer

Department of Computer & Information Sciences and Engineering
University of Florida
Gainesville, Florida 32611-6120
{lfu,jhammer}@cise.ufl.edu

Abstract

We report on a new, efficient encoding for the data cube, which results in a drastic speed-up of OLAP queries that aggregate along any combination of dimensions over numerical and categorical attributes. Specifically, we introduce a new data structure, called *Statistics Tree* (ST), together with an algorithm, called *CubiST (Cubing with Statistics Trees),* for evaluating OLAP queries on top of a relational data warehouse. We are focusing on a class of queries called cube queries, which return aggregated values rather than sets of tuples. CubiST represents a drastic departure from existing relational (ROLAP) and multi-dimensional (MOLAP) approaches in that it does not use the view lattice to compute and materialize new views from existing views in some heuristic fashion. Instead, CubiST encodes all possible aggregate views in the leaves of the underlying ST during a one-time scan of the detailed data. 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 initial set of experiments has demonstrated significant improvements in performance and scalability over existing ROLAP approaches.

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. A warehouse contains data from a number of independent sources, integrated and cleansed to support clients who wish to analyze the data for trends and anomalies. The decision support is provided by OLAP tools which 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]. Because all of these queries are often complex and the data warehouse database is often very large, processing the queries quickly is an important prerequisite for building efficient decision support systems (DSS). Before we introduce our new approach to answering complex OLAP queries in data warehousing environments to satisfy current and future decision support demands, we first review some important background information.

1.1. OLAP Queries

Users of data warehouses frequently like to "visualize" the data as a multi-dimensional "data cube" to facilitate OLAP. This so-called dimensional modeling allows the data to be structured around natural business concepts, namely measures and dimensions. Measures are numerical data being tracked (e.g. sales). Dimensions are the natural business parameters that define the individual transactions (e.g. time, location, product). Some dimensions may have hierarchies. For example, time may have a "day→month→year" hierarchy. To build a data cube, certain dimensions and measures of interest are selected from the underlying data warehouse. 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 typical operators found in OLAP queries. The data cube operator, which was proposed in [12], contains these operators and generalizes aggregates, subtotals, cross tabulations, and group-bys. In our 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." Using the warehouse, a possible cube query is: "How many Toyotas have been sold in Florida and Georgia from January to March this year?"

The focus of this report is on describing a new, efficient representation for aggregates in a data cube and a corresponding algorithm for evaluating cube queries using this representation. It is worth pointing out that there is a subtle difference between cube queries and OLAP queries. Cube queries return only *aggregate information* while the latter may also return *the detailed records* that satisfy the query conditions. Using the sample car sales example from above, an OLAP query may also return the individual sales records that contributed to the result rather than only the aggregated total sales value.

In the data warehouse environment, implementing cube queries over large databases poses great challenges. The database size often grows to hundreds of GBs or TBs with millions or even billions of records with high dimensionality and large domain sizes. There are many algorithms for evaluating OLAP and cube queries, but no existing indexing and query optimization performs sufficiently well for high dimensional data [4]. Sampling techniques are sometimes used with large data sets. However, the accuracy of the result may not be acceptable to many applications. Instead, many systems resort to parallel computing to achieve fast query response times, relying on additional hardware and software support.

1.2. Proposed Solution

In this report we introduce a new approach called *CubiST* (Cubing with Statistics Trees) for representing data cubes and for evaluating cube queries more efficiently than currently possible[1]. CubiST can be considered a MOLAP approach in spite of the fact that CubiST does not use multi-dimensional arrays directly. Instead, it uses a new data structure called *Statistics Tree* (ST) to store the information that is needed to answer data cube queries. Simply speaking, a statistics tree is a multi-way tree in which internal nodes contain references to next-level nodes and are used to direct the query evaluation. Leave nodes hold the statistics or histograms for the data (e.g., SUM, COUNT, MIN, MAX values) and are linked together to facilitate scanning, similarly to the B/B^+-Tree data structure [9]. Each root-to-leaf path in a statistics tree represents a particular subcube of underlying data set. In order to use an ST to answer cube queries over a particular data set, one must first pre-compute the aggregations on all subcubes by scanning the detailed data set. Statistics trees can also be called aggregation trees since they only store the aggregate information instead of the record details.

The usefulness of CubiST is based on the observation that in decision support queries such as "How many data points make up a certain region of the data cube?" (i.e., what we term cube queries) are much more important to the analysts than queries that retrieve the details of the individual records belonging to a certain region; this is due to the fact that given the size of the data cubes, often more than thousands of records satisfy a particular query condition.

[1] The data structures and algorithms that make up CubiST are protected by US Provisional Patent, Serial No. 60/203,5776, which was granted on May 11, 2000. A conversion into a fill-enabled utility patent is in progress.

Some other important advantages of CubisST are:

1. **Fast:** The initialization of the ST needs only *one* reading pass over the data set; all subsequent computations can be done internally on the ST without touching 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. Given the size of a data warehouse, this is an important feature.
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 the ST fits into memory, the paper will show that the response times for cube queries over a data set containing 1 million or 1 billion records are almost the same.

1.3. Outline of the Report

The remainder of the report 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 particular class of OLAP queries which can be answered using CubiST. The main part of the section is devoted to describing 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 initial results of our on-going evaluation of CubiST are presented in section 6. A summary and concluding remarks are presented in section 7.

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. ROLAP Servers

ROLAP servers store the data in relational tables using a star or snowflake schema design [7]. In the star schema, there is a fact table plus one or more dimension tables. The snowflake schema is a generalization of the star schema where the core dimensions have aggregation levels of different granularities. In the ROLAP approach, cube queries are translated into relational queries against the underlying star or snowflake schema using the standard relational operators such as selection, projection, relational join, group-by, etc. However, directly executing translated SQL can be very inefficient and as a result, many commercial ROLAP servers extend SQL to support important OLAP operations directly (e.g., RISQL from Redbrick Warehouse [30], cube operator in Microsoft SQL Server [23]).

A simple algorithm 2^N-algorithm for evaluating the cube operator is proposed in [12]. In this algorithm, where N is the number of dimensions, a handle is allocated for each for each cell of the data cube. For each new record $(x_1,x_2,\ldots,x_N,v)$ the handle function is called 2^N times – once for each handle of each cell of the cube matching this value. Here, x_i are the dimension values and v is the measure. When all input values have been processed, the final aggregate for each of the nodes in the cube is computed. Due to the large number of handles, this algorithm does not scale well for large N.

To speed up the group-by's, indices and materialized views are widely used. As far as we know, there is no internal ROLAP algorithm in the literature for evaluating cube queries efficiently. MicroStrategy [24], Redbrick [29], Informix's Metacube [18] and Information Advantage [17] are examples of ROLAP servers.

2.2. MOLAP Servers

MOLAP servers use multi-dimensional arrays as the underlying data structure. MOLAP is often several orders faster than the ROLAP alternative when the dimensionality and domain size are relatively small compared to the available memory. However, 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]. The full cube (array) is chunked into small pieces called cuboids. For a non-empty cell, a `(OffsetInChunk,data)` pair is stored. Zhao et. al. describe a single pass, multi-way algorithm that overlaps the different group-by computations to minimize the memory requirement. The authors also give a lower-bound for the memory which is required by the minimal memory spanning tree (MMST) of the optimal dimension order (which increases with the domain sizes of these dimensions). Their performance evaluations show that a MOLAP server using an appropriate chunk-offset compression algorithm is much faster than most ROLAP servers. However, if there is not enough memory to hold the MMST, several passes over the input data are needed. In the first read-write pass, data is partitioned. In the second read-write pass, the partitions are clustered further into chunks. Additional passes may be needed to compute all aggregates in the MMST execution plan. In this case, the initialization time may be prohibitively large. In addition, since the materialized views reside on disk, answering OLAP queries may require multiple disk I/Os.

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. The chunks can be stored as multi-dimensional arrays or `(OffsetInChunk,data)` pairs depending on whether they are dense or sparse. The OffsetInChunk is bit-encoded (BESS). However, like other MOLAP implementations, the algorithm still suffers from high I/O costs during aggregation because of frequent paging operations that are necessary to access the underlying data.

In general, ROALP is more scalable in terms of the data size, while MOLAP has better performance when the number of dimensions is small. However, the decision of whether to use ROLAP or MOLAP does not only depend on the original data size but also the data volume which is defined as the product of the cardinalities. To illustrate our point, consider a database with 50 billion records and three dimensions each having 100 values. Suppose that each row needs 4*4 bytes (assuming an integer uses 4 bytes and the table has one measure), then the data size is $16*50*10^9$ = 800GB but the data volume is 100*100*100 = 1M. In this case, MOLAP would a better choice.

Arbor Systems's Essbase [3], Oracle Express [27] and Pilot LightShip [28] are based on MOLAP technology. The latest trend is to combine ROLAP and MOLAP in order to take advantage of the best of both worlds. For example, in PARSIMONY, some of the operations within sparse chunks are relational while operations between chunks are multi-dimensional.

2.3. 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.

When the domain sizes are small, a bitmap index structure [26] can be used to help speed up OLAP queries. A bitmap index for a dimension with *m* values generates *m* bitmaps (bit vectors) of length *N*, where *N* is the number of records in the underlying table. To initialize a bitmap index on a particular attribute (dimension) of a table, we set the bits in each bitmap as follows: for each record we indicate the occurrence of a particular value with a 1 in the same row of the bitmap that represents the value; the bits in all other bitmaps for this row will be set to 0.

Bitmap indexes use bit-wise logical AND, OR, NOT operations to speed up the computation of the where-clause predicates in queries. However, 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 [5]. Suppose a dimension has 1,024 values. Instead of using 1,024 bit vectors most rows of which are zero, log 1024 = 10 bit vectors are used plus a mapping table, and a Boolean retrieve function. A well-defined encoding can reduce the complexity of the retrieve function thus optimizing the computation. However, designing well-defined encoding algorithms remains an open problem.

Bitmap schemes are a powerful means to evaluate complex OLAP queries when the number of records is small enough so that the entire bitmap fits into main memory. If not all of the bitmaps fit into memory (e.g., when the number of records is large), query processing will require many I/O operations. Even when all the bitmaps fit into memory, the runtime of an algorithm using bitmap indexes is proportional to the number of records in the table. Later in the report, we show that the runtime of a bitmap-based algorithm is much larger than the runtime of CubiST which has a worst-case runtime proportional to the number of dimensions of the data cube.

A good alternative to encoded bitmaps for large domain sizes is the B-Tree index structure [9]. O'Neil and Quass [25] provide an excellent overview of and detailed analyses for index structures which can be used to speed up OLAP queries.

2.4. View Materialization

View materialization in decision support systems refers to the pre-computing of partial query results which may be used to derive the answer for frequently asked queries. Since it is impractical to materialize all possible views, view selection 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. In their approach, the next view to be materialized is chosen such that its benefit is maximal among all the non-materialized views. 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]. To illustrate, assume a table has four dimensions A, B, C, and D. The computation of the following group-by operations can be pipelined: ABCD $\rightarrow$ ABC $\rightarrow$ AB $\rightarrow$A (i.e., from view ABCD we can compute view ABC form which view AB can be computed and so on). When a data partition is computed, the memory can be reused for the next partition. Interleaved with the pipelined aggregations are sorting or hashing operations on some nodes on the minimal cost processing tree. When the data set is large, external sorting 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 incrementally maintaining them [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].

However, in order to be able to support true ad-hoc OLAP queries, indexing and pre-computation of results alone will not produce good results. For example, building an index for each attribute of the warehouse or pre-computing every sub-cube requires too much space and results in unacceptable maintenance costs. On the other hand, if we index only some of the dimensions or pre-compute few views, queries for which no indexes or views are available will be slow. We believe that the approach described here which is based on a new encoding for aggregates and a new query processing algorithm represents a major step towards supporting ad-hoc OLAP queries efficiently.

3. A New Data Structure for OLAP: Statistics Tree

A *Statistics Tree* (ST) is a multi-way tree structure, which holds 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 to use statistics trees to evaluate OLAP queries over large data sets. The structure of a Statistics Tree is similar to that of the B-Tree where information is only stored in the leaf nodes of the tree; internal nodes are used as branch nodes containing pointers to subtrees.

3.1. Data Structure and Setup

Let us assume R is a relational table with attributes $A_1,A_2,\ldots,A_k$ with cardinalities $d_1,d_2,\ldots,d_k$ 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, $j = 1,2,\ldots,k$. 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* and leads to a region in the tree where this domain has been "collapsed," meaning it contains all 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 leaf nodes at level k+1 contain the aggregation information and form a linked list.

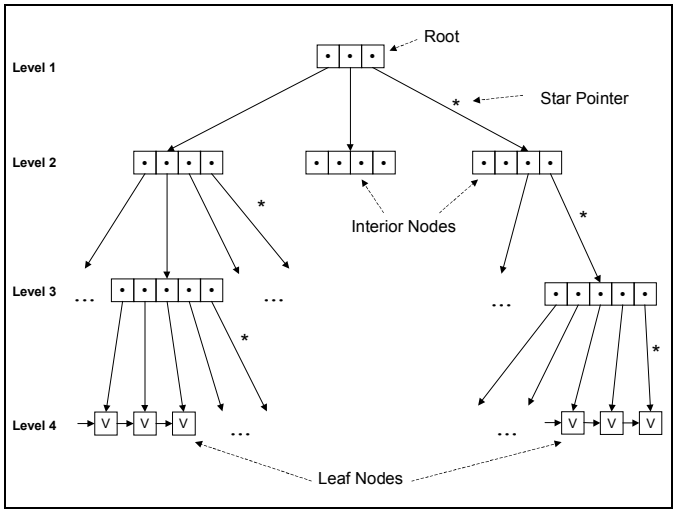


Figure 1: Example of a statistics tree for a data set with 3 dimensions.

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).

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]. Hence one can regard the Statistics Tree as a generalization of the multi-dimensional array data structure. Looking at the B^+-Tree, for example, it too, stores data only at the leaf nodes (or pointers to pages containing data) which linked together to form a list. However, a B-tree is an index structure for *one* attribute (dimension) only, as opposed to a Statistics Tree which can contain aggregates for multiple dimensions as shown in Figure 1. 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 in the input data set. Next, we describe an algorithm for updating the aggregate information that is stored in the leaf nodes of a statistics tree.

```
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 child following star pointer;
5              return;
6          level := level + 1;
7          update_count( x_(level-1)^th child of n, x, level);
8          update_count(child of n following star pointer, 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: Recursive algorithm for updating aggregates in a statistics tree.

3.2. Populating and Maintaining Statistics Trees

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 operator. 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()*. Please note that 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 x_i of the input record $x=(x_1,x_2,\ldots,x_k)$, where i indicates the current level in the tree, follow the x_i^{th} pointer as well as the star pointer to the two nodes at the next-lower level. When reaching the nodes at level k (after repeated calls to *update_count()*), increment the count values of the two leaves following the x_k^{th} pointer and the star pointer. Repeat for each record until

all input records have been processed in this fashion. A pseudo-code description of the algorithm is shown 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 when reaching 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 are used for "ALL" values.

In order to illustrate the insert procedure, let us continue with our example from Figure 1 and assume that the first record in the input set is (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.

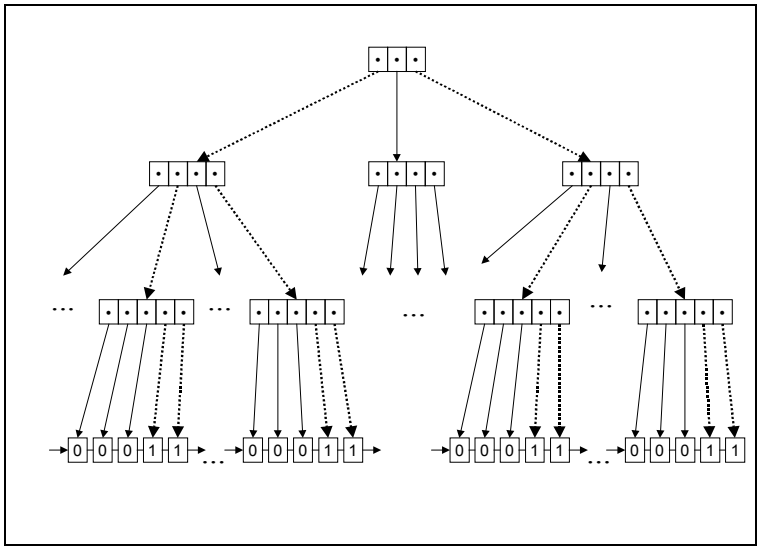


Figure 3: Statistics tree after processing input record (1,2,4).

Note, in case the elements of the database records are real numbers or integers not in [1..d_i], i=1,...,k, we can devise a mapping $F: t \rightarrow t' = (t'_1, t'_2, ..., t'_k)$, such that $t'_i \in [1..d_i]$. Once all records have been processed, the Statistics Tree is ready to answer cube queries.

4. The CUBIST 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. A 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 functional 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 4.1: 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 $P = (x_1, x_2, ..., x_k)$, $x_i \in [1..d_i]$. The total number of cells in a *k*-dimensional data cube is $\prod_{i=1}^{k} d_i$.

Definition 4.2: 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: $q = (s_{i_1}, s_{i_2}, ..., s_{i_r})$, 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_{i_j} can be one of the following (in decreasing order of generality). Let $w = i_j$:

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

Note, when $r \ne k$, 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 in the query using "ALL" values.

Definition 4.3: A query is *partial* iff $\exists j$ such that s_{i_j} 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_{i_j} 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 subcube.

Definition 4.4: 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 (usually less than 4), we say the query is of *low dimensionality*.

Continuing our car sales example from the beginning, assume that the domain of manufacturer is {Ford, GM, Honda, Toyota}, of color is {blue, red, white, black}, of style is {sports car, sedan, SUV}, of time is {Jan, Feb, …, Dec} and of location is {Florida, Alabama, Georgia, South Carolina, Tennessee}. Then a formal representation of the sample cube query "How many Toyotas have been sold from January to March this year in Florida and Georgia?" 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 has the form: $q = \{z_{i,j} \in [1..d_i] \mid i = 1,2,...,k, j = 1,2,...,d_i + 1\}$. Here $z_{i,j}$ refers to the j^{th} selected value of the i^{th} dimension. `Cubist()` is a recursive function that returns the aggregate for a cube query *q*. To compute the aggregate for a subtree rooted at Node *n*, follow the

[2] A cube containing no "ALL" values.

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 value), we only need to follow one pointer to node on the lower level.

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

Figure 4: Recursive algorithm for evaluating cube queries.

Theorem 4.1: The amount of memory that is need to store a *k*-dimensional Statistics Tree is bounded by: $c\prod_{i=1}^{k}(d_i+1) < M$, where M is the amount of memory needed to store the tree, the constant value *c* accounts for the space that is needed to store the internal nodes of the tree. Please note that c is typically around 2 or 3 because the number of internal nodes is no larger than twice the number of leaves.

Theorem 4.2: The update time of ST for each record is O(2^k). The runtime of answering a singleton query is O(k), where k is the number of dimensions. The worst case runtime of answering an arbitrary query is O(*size of ST*).

In order to update the aggregate information in the ST for each input record, the number of nodes touched is $1+2+2^2+\ldots+2^{k-1}$. To answer a singleton query, only a search path from root to leaf (k nodes are touched) is needed. However, in order to answer an arbitrary cube query, potentially most nodes of the tree may be visited.

CubiST is particularly useful when the number of records is large and the aggregated data volume does not exceed the memory size M (as calculated in Theorem 1). For example, this is the case in data marts where the dimensionality of the data is low, 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 to provide a high-level "view" over the data in order to decide where to start drilling down: in this scenario there are many dimensions with large domain sizes. Finally, with the continued increase in available memory size, more and more cube queries can be answered using ST. We will introduce a new data warehouse system architecture in the next section that achieves superior performance for cube queries when combined with CubiST.

5. A Data Warehouse Architecture Supporting CubiST

So far we have presented our algorithm and data structure to evaluate cube queries. However, in many cases, having one ST to answer cube queries is not sufficient, especially when the queries involve

many dimensions with large domain sizes. 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 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.

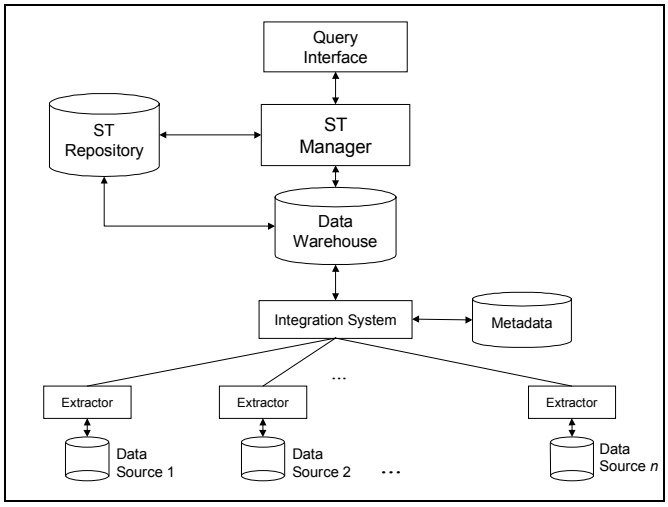


Figure 5: Data warehouse architecture with CubiST processor.

5.1. System Architecture

In order to make use of the capabilities of CubiST, the data on which to perform the cube queries has to be transformed into a single table. This single table serves as the basis from which all STs are constructed. STs are stored in the *ST Repository* which is shown in the upper-left hand corner of Fig. 5. Using the *ST Manager*, shown directly above the data warehouse, STs are selected from the repository to match the incoming cube queries: the dimensions and aggregate operators specified in the query must match the dimensions on which the tree is constructed and aggregate operators used to generate the contents of the leaf nodes. If there is no matching ST, a new ST is generated using the algorithms in Sec. 3.

5.2. Single Table Schema

In the single-table schema, attributes from the same source (i.e. the same dimension table in star schema) are grouped together, since queries are more likely to request aggregates over attributes from the same group. In order to exhibit reasonable access performance, the single table is vertically partitioned to facilitate the selective scanning. Thus, when initializing an ST, only related partitions are scanned, thus improving setup performance.

Finding a common join key for merging the individual tables in a star schema can be challenging and requires understanding of the source schemas and user requirements. For example, in the car sales example, the manufacturer ID of each car can serve as a common join key. Using a hash index on the common join key in the tables, one can perform a star join to efficiently merge the different tables. Tables can also be combined along the foreign keys in the fact table. Yet another alternative to generating the single table is to join star schema tables in pairs, gradually expanding the result until all

tables are included. This approach is particularly useful when no "common key" exists or is difficult to find. Table 1 displays a single table schema for the car sales example.

Make			Properties			Customer			Sale			Measures	
m-name	m-loc	m-date	color	style	#cyl	c-name	age	income	s-name	s-loc	s-date	sales	invoice

Table 1: Sample 1-table schema for car sales.

5.3. 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 of which STs to build depends on the "closeness" among the attributes: The closer the attributes the more likely it is 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, aggregate operators used to compute the contents of the leaves) are stored in the metadata repository. At the same time, the usage statistics of the STs are recorded. To improve the hit ratio of queries against existing STs, small, frequently matched STs are periodically merged to form one larger ST, memory size permitting. The rationale behind merging is that the attributes from frequently queried STs are more likely be queried together later.

6. Evaluation of CUBIST

We have implemented the ST and CubiST and conducted a series of experiments to validate our performance analysis of the algorithm. Our testbed consists of a 450 MHZ Pentium III running Windows 98 with 96MB of main memory as well as a SUN ULTRA 10 workstation running Sun OS 5.6 with 90MB of available main memory. The ST and its operators as well as CubiST were implemented in JAVA using JDK 1.2.

6.1. Overview of the Experiments and 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 written to disk as a text file and serves as input to our ST and CubiST algorithms.

We have generated five synthetic data sets D1 through D5 as shown in Table 2. The data sets are meant to reflect different characteristics in the data, varying in the number of records, number of dimensions, query complexity, and domain size. The data sets represent a trade-off between having sufficient data to arrive at meaningful conclusions versus limiting the time spend in conducting the experiments.

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 simple query evaluation 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 answer cube queries without touching the original data set (assuming the bit vectors are already setup). CubiST is implemented as described in Sections 3 and 4. We now describe the three series of experiments.

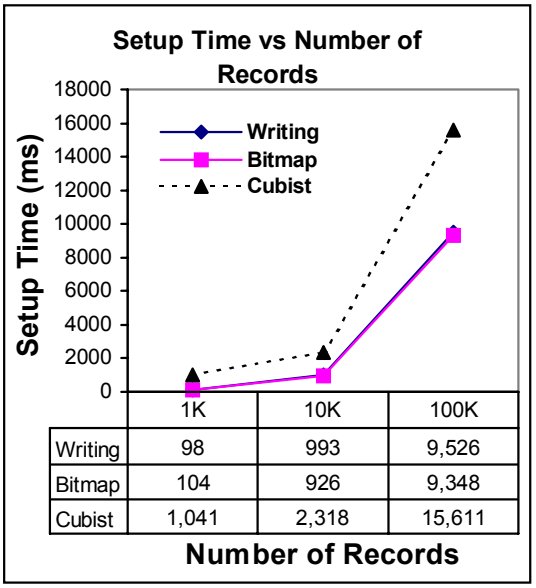
data set	dataset size	# records	domain sizes
D1			
	20 K	1 K	10, 10, 10, 10, 10
	200 K	10 K	10, 10, 10, 10, 10
	2 M	100 K	10, 10, 10, 10, 10
	20 M	1 M	10, 10, 10, 10, 10
D2			
	2 M	100 K	15, 15, 15, 15, 15
	20 M	1 M	15, 15, 15, 15, 15
	100 M	5 M	15, 15, 15, 15, 15
D3			
	1.6 M	100 K	10, 10, 20, 81
	2 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
D4			
	2 M	100 K	15, 15, 15, 15, 15
D5			
	1.2 M	100 K	20, 20, 20
	1.2 M	100 K	40, 40, 40
	1.2 M	100 K	60, 60, 60
	1.2 M	100 K	80, 80, 80

Table 2 : Five data sets used in the simulations.

6.2. Varying Number of Records

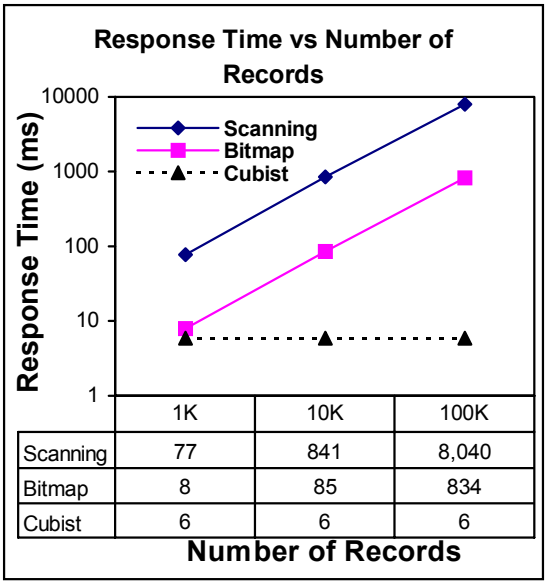
In this first series of experiments which uses data sets D1 and D2, the number of dimensions and corresponding domain sizes are fixed while we increase the size of the input set. The goal is to validate our claim that CUBIST has excellent scalability in terms of the number of records.

Figures 6 and 7 show the performance of the three algorithms on the first data set D1. The sample cube query on which the experiments are based is as follows: $q1 = (s_1,s_2,s_3,s_4,s_5) = ([0,5],[0,5],[0,5],[0,5],[0,5])$. Query $q1$ is an example of a range query which computes the total number of records that lie in the region where all dimension values are in the range of [0,5]. Although the setup time for CubiST is larger than that used by Bitmap because of the insertion operation for each record, its response time is much faster and almost independent of the number of records. 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.



	1K	10K	100K
Writing	98	993	9,526
Bitmap	104	926	9,348
Cubist	1,041	2,318	15,611

Figure 6: Setup time vs number of records.



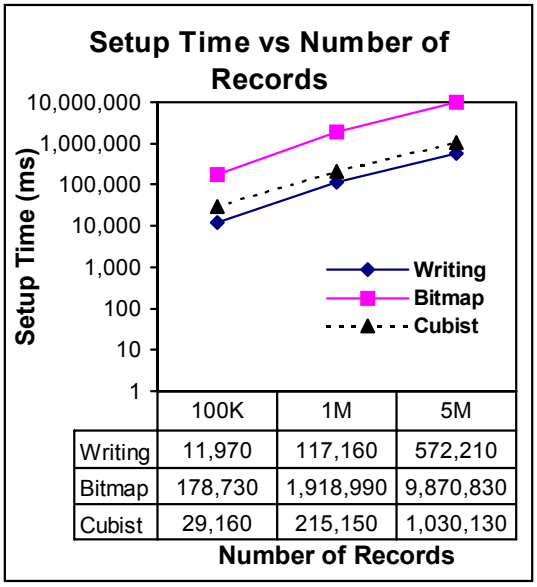
	1K	10K	100K
Scanning	77	841	8,040
Bitmap	8	85	834
Cubist	6	6	6

Figure 7: Response time vs number of records.

We have repeated this series of experiments using data set D2 and query $q2 = (s_1,s_2,s_3,s_4) = ([1,10],[1,10],[1,10],[1,10])$. This time the size of the data set increased from 2MB to 100MB. The domain size for each of the five dimensions is 15. The experiments for D2 were run on the PC, which has a larger available local disk than the SUN workstation, to accommodate the increased data volume. The measurements for $q2$ are shown in Figures 8, 9. Our measurements clearly show the linear time characteristics of the scanning and bitmap algorithms as well as the superiority of CUBIST. The setup and response times of the bitmap algorithm are considerably worse than those of the scanning algorithm: when the number of records increases beyond 1M, the bitmaps cannot fit into memory which requires the algorithm to write to and subsequently read the bitmaps back from disk.

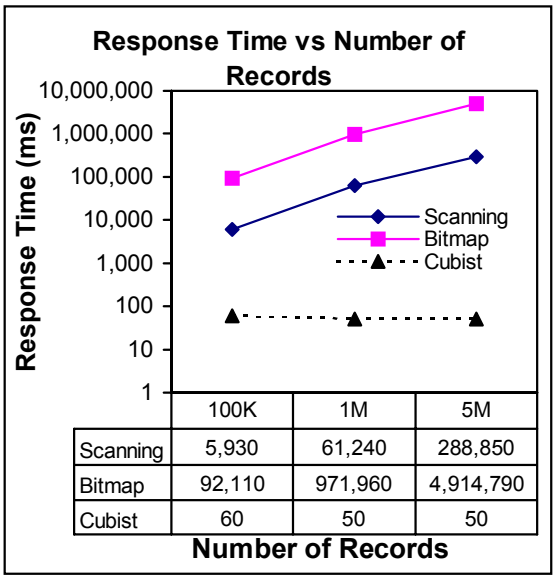
In the last experiment of this series (data set D2), the size of each bitmap is 75MB (5M×15Bytes, 1 byte for a Boolean variable in JAVA, domain size 15). The five bitmaps, one for each dimension, are partitioned horizontally into segments so that each segment fits into memory. During setup, bitmaps are written to disk while at the same time reading data from the input file. In order to evaluate cube queries, bitmaps are read into memory segment by segment. In this experiment, the total size of bitmaps (75MB×5=375MB) is much larger than the original data set size (100MB). The additional writing and reading of the large bitmaps becomes the major bottle-neck, which is a good reason to avoid this approach when the cardinalities and number of records are large.

From the results of this series of experiments, we also see that the response time of CubiST is almost independent of the number of records, once the ST has been setup and fits into memory. Furthermore, the gap between the setup time of ST and the writing time of input data decreases as the data set increases.



	100K	1M	5M
Writing	11,970	117,160	572,210
Bitmap	178,730	1,918,990	9,870,830
Cubist	29,160	215,150	1,030,130

Figure 8: Setup time vs number of records.

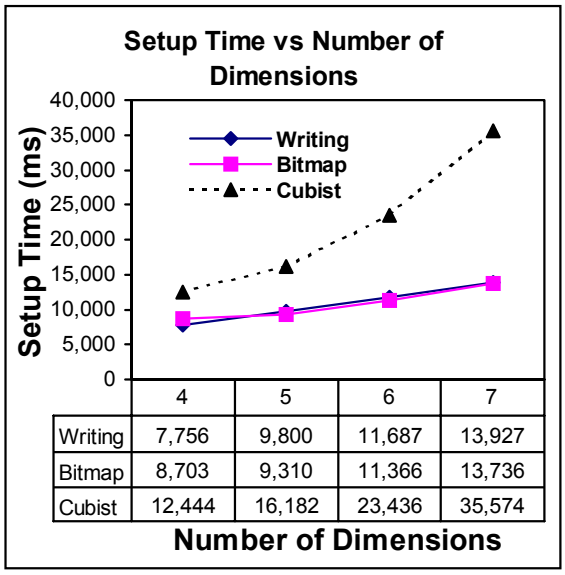


	100K	1M	5M
Scanning	5,930	61,240	288,850
Bitmap	92,110	971,960	4,914,790
Cubist	60	50	50

Figure 9: Response time vs number of records.

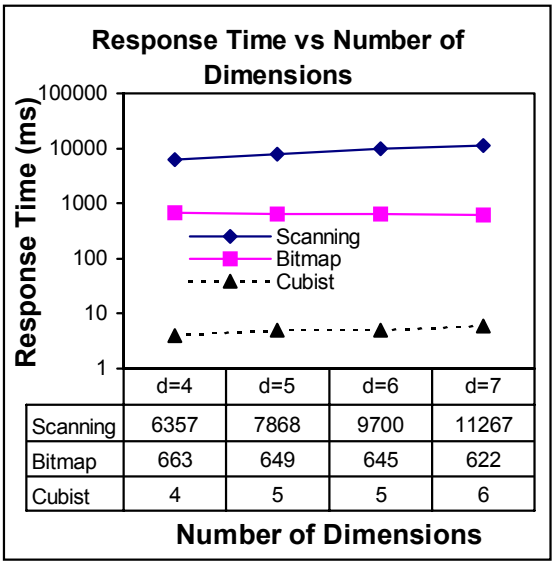
6.3. Varying Number of Dimensions

In this set of experiments, we used data set D3 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/(10×10×20×81) = 61.73%) and fixed the number of records (100,000). The input query $q3 = (s_1, s_2, s_3, s_4) = ([0,8], [1,8], [1,4], [1,3])$.



	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

Fig. 10: Setup Time vs number of dimensions.



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

Fig. 11: Response 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 all the bitmaps fit into memory (using D3). As for CubiST, since number of leaves of the ST increases from 208,362

(11×11×21×82) to 363,000 (11×11×6×5×10×10), 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 (Figure 10). However, when the size of the data set increases to the point when I/O operations 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 11). For data sets with large cardinalities, bitmap is not space efficient and the performance of will degrade quickly. Although using the bitmaps fit into memory this time, we can observe that the response time decreases as the cardinalities decrease (since number of dimensions increases).

6.4. Response Time vs Domain Size and Query Complexity

In this final series of experiments, which is based on data sets D4 and D5, we compare response times to domain size and query complexity (measured in terms of selection components). In data set D4, the number of records and the domain sizes are fixed but the degree of the input queries increases as follows:

$q4.1 : (s_1) = ([1,10]);$

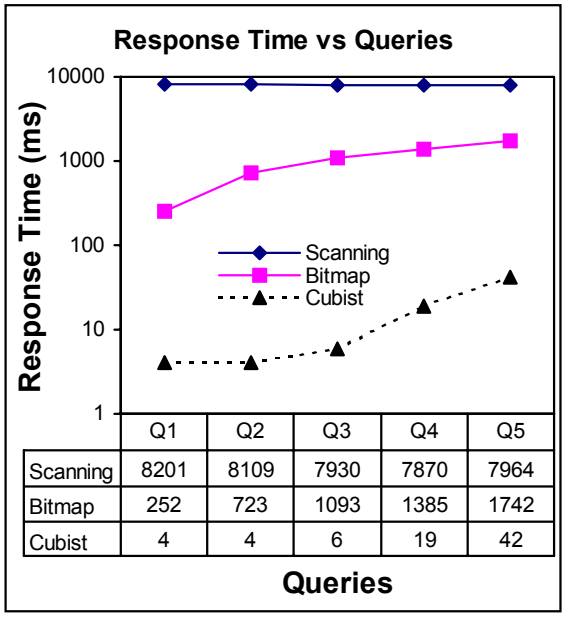
$q4.2 : (s_1, s_2) = ([1,10],[1,10]);$

$q4.3 : (s_1, s_2, s_3) = ([1,10],[1,10],[1,10]);$

$q4.4 : (s_1, s_2, s_3, s_4) = ([1,10],[1,10],[1,10],[1,10]);$

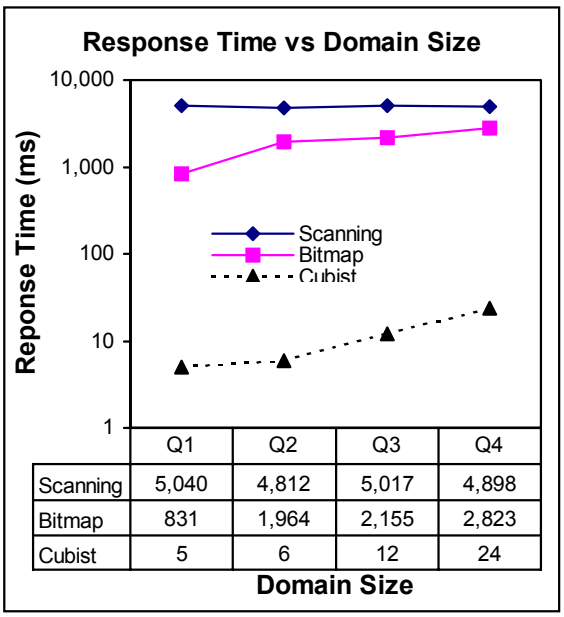
$q4.5 : (s_1, s_2, s_3, s_4, s_5) = ([1,10],[1,10],[1,10],[1,10],[1,10]);$

Although the performance of the scanning approach is less sensitive to the varying degree of a query when compared with bitmap and CUBIST, its response times were nonetheless several orders larger (Figure 12). As before, CUBIST is considerably faster than the bitmap approach. The large increase of the number of the selected cells in the query, an increase from 10 to 100,000, accounts for the increase in response time of CUBIST.



	Q1	Q2	Q3	Q4	Q5
Scanning	8201	8109	7930	7870	7964
Bitmap	252	723	1093	1385	1742
Cubist	4	4	6	19	42

Figure 12: Response time vs query complexity.



	Q1	Q2	Q3	Q4
Scanning	5,040	4,812	5,017	4,898
Bitmap	831	1,964	2,155	2,823
Cubist	5	6	12	24

Figure 13: Response time vs domain sizes.

We repeated the series of experiments using data set D5. This time we fixed the number of records and dimensions. Instead, we increased the domain sizes of each dimension from 20 to 80 and used the following sets of queries:

$$q5.1 : (s_1, s_2, s_3) = ([0,10],[0,10],[0,10]);$$
$$q5.2 : (s_1, s_2, s_3) = ([0,20],[0,20],[0,20]);$$
$$q5.3 : (s_1, s_2, s_3) = ([0,30],[0,30],[0,30]);$$
$$q5.4 : (s_1, s_2, s_3) = ([0,40],[0,40],[0,40]);$$

From Figure 13, we can see the drastic increase in response times for both CubiST and bitmap as the number of selected cells increases. In summary, the above experiments showed that CubiST demonstrates significant improvement in performance and scalability over both the scanning algorithm as well as ROLAP implementations based on bitmap indexing techniques.

7. Conclusion

In this report, we have presented an approach to answering cube queries called CubiST which stands for Cubing with Statistics Trees. Our approach is based on a new efficient representation for data cubes called Statistics Tree, the concept of cube queries, and a corresponding query processing algorithm. Cube queries are a generalization of the cube operator and compute aggregate values rather then return set of tuples. Our optimal one-pass initialization and internal query evaluation algorithms ensure fast set-up and near instant responses to the complex cube queries. Using CubiST, the records are aggregated into the leaves of the statistics tree without storing the detail input data. During the scan of the input data, 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. Our initial simulation results demonstrate the performance benefits and superior scalability. CubiST is incremental and highly adaptive and thus an ideal tool to evaluate ad-hoc cube queries.

When the number of dimensions and the domain sizes are large, a single ST over the entire data set may not fit in memory. In order to offer CubiST as a viable solution in decision support systems, we propse a new data warehousing architecture that incorporates the techniques of CubiST. The idea is to set up and manage STs for those dimensions, which appear frequently in queries, in a separate ST repository. Pre-computing STs in this way will allow for more efficient processing of all queries on these dimensions. We are now exploring the idea of using multiple derived STs to better support cube queries requesting data at higher levels of abstractions (rolled-up dimensions) which may be better served by a smaller ST involving only the desired dimensions at the proper level of detail. This significantly reduces I/O time and improves in-memory performance over the current, single-tree version of Cubist. In the long-term future, we will explore ways of partitioning large STs across multiple processors and parallelize query evaluation. It is expected that a parallel version of CubiST will improve query processing even further. We are also actively pursuing the development of new data mining algorithms that can take advantage of the query processing capabilities 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*, Birmingham, U.K., 1997.

[3] Arbor Systems, "Large-Scale Data Warehousing Using Hyperion Essbase OLAP Technology," Arbor Systems, White Paper, http://www.hyperion.com/whitepapers.cfm.

[4] S. Berchtold and D. A. Keim, "High-dimensional index structures database support for next decade's applications (tutorial)," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, WA, pp. 501, 1998.

[5] 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.

[6] 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.

[7] S. Chaudhuri and U. Dayal, "An Overview of Data Warehousing and OLAP Technology," *SIGMOD Record*, **26**:1, pp. 65-74, 1997.

[8] E. F. Codd, S. B. Codd, and C. T. Salley, "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate," Technical Report, http://www.arborsoft.com/OLAP.html.

[9] D. Comer, "The Ubiquitous Btree," *ACM Computing Surveys*, **11**:2, pp. 121-137, 1979.

[10] 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.

[11] S. Goil and A. Choudhary, "PARSIMONY: An Infrastructure for Parallel Multidimensi onal Analysis and Data Mining,," *Journal of Parallel and Distributed Computing*, to appear.

[12] 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.

[13] A. Gupta, V. Harinarayan, and D. Quass, "Aggregate-query Processing in Data Warehousing Environments," in *Proceedings of the Eighth International Conference on Very Large Databases*, Zurich, Switzerland, pp. 358-369, 1995.

[14] H. Gupta and I. Mumick, "Selection of Views to Materialize Under a Maintenance Cost Constraint," Stanford University, Technical Report.

[15] J. Han, "Towards On-Line Analytical Mining in Large Databases," *SIGMOD Record*, **27**:1, pp. 97-107, 1998.

[16] 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.

[17] Information Advantage, "Business Intelligence," White Paper, 1998, http://www.sterling.com/eureka/.

[18] Informix Inc., "Informix MetaCube 4.2, Delivering the Most Flexible Business-Critical Decision Support Environments," Informix, Menlo Park, CA, White Paper, http://www.informix.com/informix/products/tools/metacube/datasheet.htm.

[19] 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.

[20] 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,

[21] D. Lomet, *Bulletin of the Technical Committee on Data Engineering*, vol. **18**, IEEEE Computer Society, 1995.

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

[23] Microsoft Corp., "Microsoft SQL Server," Microsoft, Seattle, WA, White Paper, `http://www.microsoft.com/federal/sql7/white.htm`.

[24] MicroStrategy Inc., "The Case For Relational OLAP," MicroStrategy, White Paper, `http://www.microstrategy.com/publications/whitepapers/Case4Rolap`.

[25] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **26**:2, pp. 38-49, 1997.

[26] P. E. O'Neil, "Model 204 Architecture and Performance," in *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, Asilomar, CA, pp. 40-59, 1987.

[27] Oracle Corp., "Oracle Express OLAP Technology", Web site, `http://www.oracle.com/olap/index.html`.

[28] Pilot Software Inc., "An Introduction to OLAP Multidimensional Terminology and Technology," Pilot Software, Cambridge, MA, White Paper, `http://www.pilotsw.com/olap/olap.htm`.

[29] Redbrick Systems, "Aggregate Computation and Management," Redbrick, Los Gatos, CA, White Paper, `http://www.informix.com/informix/solutions/dw/redbrick/`.

[30] Redbrick Systems, "Decision-Makers, Business Data and RISQL," Informix, Los Gatos, CA, White Paper, 1997, `http://www.informix.com/informix/solutions/dw/redbrick/`.

[31] 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.

[32] 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.

[33] Y. Zhao, P. M. Deshpande, and J. F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **26**:2, pp. 159-170, 1997.