



CubiST⁺⁺: Evaluating Ad-Hoc CUBE Queries Using Statistics Trees

JOACHIM HAMMER*

jhammer@cise.ufl.edu

Computer & Information Science & Eng., University of Florida, Gainesville, FL 32611-6120, USA

LIXIN FU

lfu@uncg.edu

Division of Computer Science, University of North Carolina, Greensboro, Greensboro, NC 27402-6170, USA

Recommended by: Abdelsalam Helal

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. We are focusing on a class of queries called *cube queries*, which return aggregated values rather than sets of tuples. Our approach, termed CubiST⁺⁺ (Cubing with Statistics Trees Plus Families), 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 a new data structure called *statistics tree* (ST) during a one-time scan of the detailed data. In order to optimize the queries involving constraints on hierarchy levels of the underlying dimensions, we select and materialize a family of candidate trees, which represent superviews over the different hierarchical levels of the dimensions. Given a query, our query evaluation algorithm selects the smallest tree in the family, which can provide the answer. Extensive evaluations of our prototype implementation have demonstrated its superior run-time performance and scalability when compared with existing MOLAP and ROLAP systems.

Keywords: data cube, data warehouse, multi-dimensional OLAP, query processing, statistics tree

1. OLAP queries and data cubes

Given the importance of decision support, data warehouse and related on-line analytical processing (OLAP) technologies [5, 6] continue to receive strong interest from the research community and from industry. Incidentally, the term OLAP is credited to Codd, who proposed the creation of large data warehouses for multidimensional data analysis [8]. The decision support is provided by OLAP tools (e.g., Brio Technology's Brio ONE, Comshare's Decision), 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 [9]. Since analytical queries are complex and the data warehouse is often very large, processing queries against the warehouse quickly is an important prerequisite for building efficient decision support systems.

*To whom correspondence should be addressed.

Users of data warehouses frequently like to “visualize” the data as a multidimensional “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 OLAP approach (ROLAP), which uses the familiar “row-and-column view,” and the multidimensional OLAP (MOLAP) approach, which uses proprietary data structures to store the data cube.

OLAP queries select data that is represented in various n -dimensional regions of the data cube, called subcubes. Slicing, dicing, rolling-up, drilling-down, pivoting, etc. are typical operators found in OLAP queries. The *data cube* operator, which was proposed in [15] contains these operators and generalizes aggregates, subtotals, cross tabulations, and group-bys. In this work, we further generalize the data cube operator so that each selected dimension set in the query can be a value, a range, or an arbitrary subset of the domain. Furthermore, the selected values of the dimensions that constitute the subcubes can be at any hierarchical level of the underlying dimensions. We term this new operation *cube query* and provide a formalism for this class of queries in Section 4. For example, in a relational warehouse containing information on car sales with a measure called “Sales” and five dimensions called “Manufacturer,” “Color,” “Style,” “Time,” and “Location,” a possible cube query is: “How many Toyotas have been sold in Florida and Georgia between January and March of this year?” Evaluating such cube queries efficiently is henceforth referred to as the “*cubing* problem.” In this paper, we describe a new approach to solving the cubing problem efficiently and without the limitations of existing methods, most importantly the limited support for ad-hoc querying. There is a subtle difference between cube queries and general 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 value.

1.1. Overview of the approach and contributions to the state of the art

We introduce a new algorithm called CubiST⁺⁺ (Cubing with Statistics Trees Plus Families) to evaluate cube queries more efficiently than currently possible. CubiST⁺⁺ does not use multidimensional arrays directly. Instead, it uses a new data structure called *Statistics Tree* (ST) to encode those facts about the data that are 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. Leaf nodes hold the statistics or histograms for the data (e.g., SUM, COUNT, MIN, MAX) and are linked to facilitate scanning, similarly to the B-tree data structure [10].

Each root-to-leaf path in a statistics tree represents a particular subcube of the underlying data set. In order to use an ST to answer cube queries against a particular data set, one must

first pre-compute the aggregations on all subcubes by scanning the detailed data set. For each record, the aggregate values in corresponding leaves are updated. As we show later, each record is responsible for multiple aggregates in the ST. Once the ST (called *base tree*) is populated, one can then derive a family of smaller trees from it to further improve the performance of queries that involve only dimension values at higher levels of abstraction. These *derived trees* which contain the same dimensions as the base tree, represent various types of aggregates over the base data set. We have developed a greedy algorithm to select the family members and compute the derived trees. After the initial setup and derivation steps, the family is written to disk and can be used to answer cube queries that match the dimensions and level of abstraction encoded in the STs. In order to reduce the number of I/O operations, our new query-matching algorithm selects the smallest statistics trees from the families that can provide the answers to the input cube queries. Queries are evaluated against the selected trees using our query-answering algorithm.

In summary, our main contributions are three-fold. We have developed:

1. A new representation for data cubes, called statistics tree (ST), which supports efficient encoding of multidimensional aggregates.
2. Algorithms for selecting and computing a family of statistics trees, which can be used to evaluate all possible cube queries that aggregate over a given set of dimensions each with its own hierarchy of domain values.
3. A cube query language (CQL) and a query evaluation algorithm to select from the family of STs the smallest tree that can answer a given cube query in the most efficient fashion.

1.2. Outline of the paper

The remainder of the paper is organized as follows. In Section 2, we review related research activities. Section 3 introduces the ST data structure and its maintenance, our representation of dimension hierarchies as well as the generation of families. Our new language for representing cube queries is described in Section 4. Section 5 presents the query evaluation algorithms using families of statistics trees. A description of our CubiST⁺⁺ prototype system and the results of our experimental analysis are summarized in Section 6. Section 7 concludes the paper.

2. Related research

Research related to CubiST⁺⁺ falls into three categories: OLAP servers including relational OLAP (ROLAP) and multidimensional OLAP (MOLAP), indexing, and view materialization.

2.1. ROLAP

ROLAP servers store the data in relational tables using a star or snowflake schema design [6]. 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 standard relational operators such as *selection*, *projection*, *relational join*, *group-by*, etc. However, executing these SQL conversions can be very inefficient and as a result, many commercial ROLAP servers extend SQL to support important OLAP operations natively (e.g., RISQL from Redbrick Warehouse [31], the cube operator in Microsoft SQL Server [24]). For example, Redbrick Intelligent SQL (RISQL) extends SQL with analysis functions such as running total (CUME), moving average (MOVINGAVG), quantiling (NTILE), selection (RANK), and fractions (RATIOTOREPORT). In order to speed up grouping, indexes and materialized views are widely used.

As far as we know, there is no in-memory ROLAP algorithm for evaluating cube queries efficiently. A simple 2^N -algorithm (N is the number of dimensions in the cube) for evaluating the cube operator is proposed by Gray et al. [15]. However, the algorithm does not scale well for large N . MicroStrategy [25], Redbrick [31], Informix's Metacube [19] and Information Advantage [18] are examples of ROLAP servers.

2.2. MOLAP

MOLAP servers use multidimensional 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 become very sparse resulting in many empty cells in the array structure. Storing sparse data in an array in this fashion is inefficient.

A popular technique to deal with the sparse data is *chunking*. 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. [39] 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 significantly 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 are 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.

To address the scalability problem of MOLAP, Goil and Choudhary proposed a parallel MOLAP infrastructure called PARSIMONY [13, 14]. 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 (*Offset-InChunk*, *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.

Beyer and Ramakrishnan have proposed an algorithm called *bottom up cube* (BUC) that solves a special subclass of the cubing problem called *Iceberg CUBE* [4]. An iceberg CUBE only computes the cubes that have aggregates above some threshold instead of computing all the cubes. BUC computes the cubes in a bottom-up fashion with pruning (i.e. computing one dimensional and two dimensional dense cubes first). Similar to the *Apriori* strategy used in the mining of association rules [34], BUC avoids computing the cubes with aggregation values below a user-specified minimum support. In this way, BUC is efficient for evaluating sparse cubes. However, like the apriori approach, BUC prunes from the bottom up while the low-dimensional cubes are usually above the threshold and are not pruned until the algorithm proceeds to high dimensional sparse cubes. BUC may require multiple scans of the input data sets if the intermediate data do not fit into memory, which is expensive in large warehouses.

Roussopoulos et al. [32] have proposed a data model called *extended datacube model* (EDM) for visualizing data cubes. In their model, each tuple of a relation R is projected into all subspaces of the full cube and the group bys are regarded as multidimensional objects. Queries are mapped into multidimensional range queries. The tree-like structure of the EDM is referred as the *cubetree* of R which is implemented using a packed R -tree [33] and also reduced the problem of setting up and maintaining the cube as sorting and bulk incremental merge-packing of cubetrees. However, their approach does not address how to efficiently handle dimension hierarchies and only optimizes range queries. The sort-pack order of the dimensions is sensitive. In addition, external sorting is time consuming for large data sets.

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 multidimensional. Hyperion's Essbase [3], Oracle Express [28] and Pilot LightShip [30] are based on MOLAP technology.

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 applied in batch processes allowing time to reorganize data and indexes to an optimally clustered form.

When the domain sizes are small, a bitmap index structure [27] 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. The occurrence of a particular value in a dimension is indicated with a 1 in the same row of the bitmap that represents the value; the bits in all other bitmaps for this row are 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 range queries returning a large result set. In order to overcome this deficiency, an encoded bitmap scheme has been proposed [7]. Suppose a dimension has 1,024 values. Instead of using 1,024 bit vectors most rows of which contain zeros, $\log 1024 = 10$ 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. Otherwise, query processing requires 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. For comparison, the runtime of CubiST⁺⁺ is proportional to the number of dimensions.

A good alternative to encoded bitmaps for large domain sizes is the *B*-tree index structure [10]. O'Neil and Quass [26] provide an excellent overview of and detailed analyses for index structures which can be used to speed up OLAP queries. In decision support systems, a hybrid index combining bitmap and B-tree is often used [19].

Joins, especially those among multiple tables, are very common operations in data warehousing. How to efficiently perform joins is a critical issue. One can use general-purpose join techniques such as nested loop join, merge join, hybrid join, hash join, etc. A join index provides a means whereby a database system can efficiently translate restrictions on columns of one table to restrictions on another table. For a multi-table join, the STARindex proposed by Redbrick [19] is effective. The STARindex translates in advance restrictions on multiple dimension tables to restrictions on the fact table. Like composite indexes, the STARindex is a multi-column index. However, the order of the dimensions in the STARindex is important because computing queries that only constrain on non-leading dimensions is much less efficient.

Another index structure that is relevant to our work is the *cube forest* concept described by Johnson and Shasha in [20]. The basic structure is a *cube tree*, in which each node represents an index on one or more attributes of the data set and contains the aggregate over the values of their children. Different instantiations of a cube tree template, which contains the order in which attributes are indexed, lead to multiple cube trees, which make up a cube forest. In order to efficiently answer queries involving dimension hierarchies, the authors have developed a *hierarchically split cube forest* which contains the attributes for a subset of the dimensions (as opposed to full cube forest which contains all dimensions). As their paper shows, the cube forest index structure is superior to two-level indexes and bit vectors. However, there is significant overhead in keeping track of the cube tree templates, which is made worse by the fact that the cube trees are attached to each of the co-dimensional attributes (called "splitting a dimension"). On the contrary, in our approach, we combine these separate trees into one compact data structure, which we call the ST. Having a single data structure eliminates the overhead of managing the different templates. In addition, we store aggregate values only in the leaves thereby simplifying query processing and allowing for a more compact representation of STs. Similarly to the cube forest approach, we also materialize additional derived trees containing only a subset of the dimensions and different levels of aggregation. However, an essential component of our approach is a greedy algorithm for deciding which derived ST to materialize, so that during query processing, we can use the smallest possible ST to answer a given query.

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, Harinarayan et al. [17] introduced a greedy algorithm for choosing a near-optimal subset of views from a view materialization lattice based on 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, various algorithms in [16, 21, 22], for example, were developed 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 Agrawal et al. [1]. Other related research in this area has focused on warehouse physical design [21], view materialization and maintenance (e.g., refer to [40] or [23] for a summary of excellent papers on this topic), answering queries using incomplete data cubes [11], and processing of aggregation queries [16, 35, 38].

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 subcube requires a lot of space and results in high maintenance cost. 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.

3. The 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. The structure of a statistics tree is similar to that of the B-tree where information is only stored in the leaf nodes; internal nodes are used as branch nodes containing pointers to subtrees.

3.1. Structure and maintenance of statistics trees

Let us assume R is a data set with attributes A_1, A_2, \dots, A_k with cardinalities d_1, d_2, \dots, 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 a dimension in R .
- The fan-out (degree) of a branch node at level j is $d_j + 1$, where $j = 1, 2, \dots, k$. The first d_j pointers point to the subtrees, which store information for the j th dimension 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 dimension has been “collapsed,” meaning it contains all of the domain values for this dimension. This “collapsed” dimension is related to the definition of super-aggregate (a.k.a “ALL”) presented in Gray et al. [15].

- The leaf nodes at level $k + 1$ contain the aggregates and form a linked list.

Figure 1 depicts a sample statistics tree for a data set with three dimensions A_1, A_2, A_3 with cardinalities $d_1 = 2, d_2 = 3,$ and $d_3 = 4$ respectively. Since the tree is still empty, the letter ‘V’ in the leaf nodes is a placeholder for the aggregate values that will eventually be stored there. Notice that unlike the internal nodes in the tree, the leaves have only one entry—the aggregate value.

Although the ST data structure is similar to multidimensional arrays and B-trees, there are significant differences. For example, a multidimensional array does not have a star pointer although the addition of the “ALL” value to the domain of a dimension attribute has been used in the query model and summary table described in Gray et al. [15]. More importantly, unlike a multidimensional array, whose dimensionality must be declared at compile-time and cannot easily be changed, the statistics tree data structure is more dynamic, meaning its shape (e.g., levels, fan-out) can be adjusted at load time depending of the characteristics of the data set. Looking at the B-tree, for example, it too, stores data only at the leaf nodes (or pointers to pages containing data), which are 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. In addition, in a B-tree, the degree of internal nodes is restricted. The statistics tree on the other hand is naturally balanced (it is always a full tree); its height is based on the number of dimensions but independent of the number of records in the input data set.

The amount of memory that is needed to store a k -dimensional ST is bounded by $cs \prod_{i=1}^k (d_i + 1) < M$, where M is the amount of memory allocated for CubiST⁺⁺, s

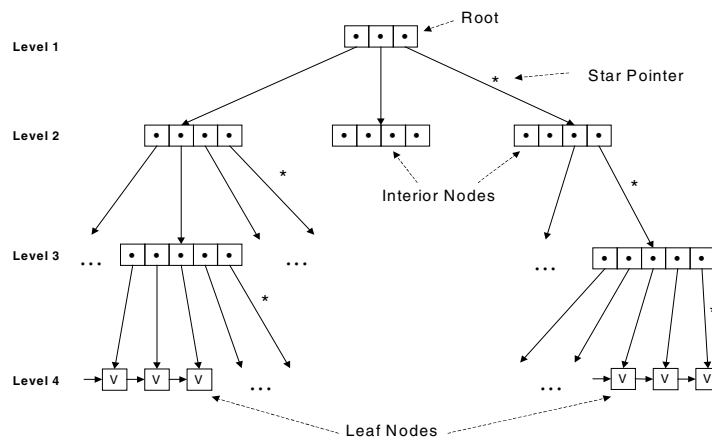


Figure 1. Sample statistics tree for a data set with three dimensions.

is the node size in bytes, and the constant c accounts for the space that is needed to store the internal nodes of the tree. Note that c is around 2 or 3 since the number of internal nodes is no larger than twice the number of leaves. It is clear that the space requirement for storing an ST tree could exceed the available memory when the data set has many large-cardinality domains. Fortunately, in decision support, most users are interested in so-called “big picture queries” which are highly aggregated. These types of queries can be answered by STs whose domain values are aggregated and hence require STs that are much smaller than those based on un-aggregated data. We have more to say on aggregated STs and how our algorithm takes advantage of their smaller size in Section 3.3.

We demonstrate how to initialize a statistics tree to help answer OLAP queries involving COUNT aggregations. STs for other aggregate operations such as SUM, MIN, MAX, etc. can be populated 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 used. For the following, we assume an empty statistics tree has been setup and the count values in the leaf nodes are zero.

Algorithm 1 depicts the pseudo-code for the initialization procedure *update_count*(). Node n is a pointer to the current node in the tree. The one-dimensional array x contains the current input record. The third parameter, *level*, indicates the level of node n . The recursive algorithm scans the input data set record by record, using the attribute values of each record to update the aggregates in the statistics tree. Note, in order to accommodate other aggregate operators, the code has to be modified only slightly. For example, to implement *update_sum*(), we increase the aggregated values in the leaves by the new measure value for each record (lines 3 and 4).

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 $\vec{x} = (x_1, x_2, \dots, x_k)$, 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

```

1  update_count(Node n, record x, int level) {
2      IF level == k THEN
3          increase count field for  $x_i$ th child of Node n;
4          increase count field for child following star pointer;
5          return;
6          level := level + 1;
7          update_count( $x_{level}$ 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);

```

Algorithm 1. Recursive algorithm for updating leaf nodes in a statistics tree using count aggregation.

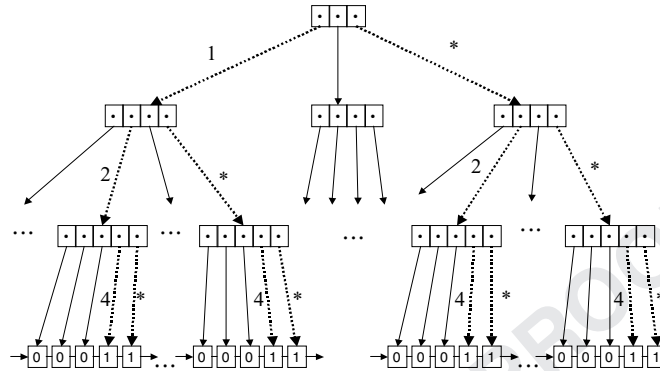


Figure 2. Statistics tree after inserting input record (1, 2, 4).

repeated calls to *update_count*(), increment the count values of the two leaves following the x_k th pointer and the star pointer. The update procedure is repeated for each record until all input records have been processed in this fashion. In order to update the aggregate information in an ST for a given input record, the number of nodes that have to be touched is $1 + 2 + 2^2 + \dots + 2^{k-1}$, where k is the number of dimensions. Hence, the update time for a single record in an ST is $O(2^k)$.

To illustrate the updating of leaf nodes, let us continue with the empty ST shown in figure 1 and assume that the first record in the input set is $x = (1, 2, 4)$. Note, the values “1”, “2”, and “4” are integer representations for the domain values of the three dimensions A_1, A_2, A_3 . For example, if the first dimension represents Manufacturer, “1” may represent the value “Acura”, etc.¹ The mappings for the other two dimensions work similarly. In general, whenever elements x_i of the input records are not integers in $[1..d_i], i = 1, \dots, k$, we can devise a mapping $F : \vec{x} \rightarrow \vec{x}' = (x'_1, x'_2, \dots, x'_k)$, such that $x'_i \in [1 \dots d_i]$. For example, the sample mapping above can be represented and stored as a trie or dictionary for the string to integer encoding and as an array for the integer to string decoding.

Figure 2 below depicts the contents of the statistics tree after processing the input record $x = (1, 2, 4)$. The update paths relating to this record are shown as dashed lines. Pointers and nodes, which do not play a role in this update, are omitted to improve readability. The ST is updated as follows. Since the first component value of x is 1 ($x_1 = 1$), we follow the first pointer as well as the star (ALL) pointer to the first and third nodes in the second level. From each of these nodes, we recursively follow the second ($x_2 = 2$) and star pointers to the third level in the tree. Finally, we follow the fourth ($x_3 = 4$) and star pointers of the third level nodes to the leaf nodes where we increment the count aggregates of all eight affected nodes by one.

3.2. Hierarchies

Often dimensions can be naturally abstracted into different hierarchy levels representing varying degrees of granularities. For example, a “day-month-year” hierarchy for Time

reflects different granularities of time. Analysts often submit queries with constraints on different levels of a hierarchy in addition to selecting different combinations of dimensions. This adds to the complexity of queries and requires an efficient representation for such hierarchies.

In relational systems, hierarchical attributes (e.g., “day,” “month” and “year” of the Time dimension) are represented in the same way as other non-hierarchical attributes. In our new framework, we *explicitly* represent the hierarchical inclusion relationships among the domain values as integer mappings (similarly to the domain value encodings described earlier). In particular, we partition lower level domain values into groups whose labels constitute the values for the next higher level of the dimension hierarchy. The partitioning process is repeated until the highest level has been reached. This explicit representation of hierarchies through domain-value mapping and hierarchical partitioning is useful for answering queries about the dimensional structure (e.g., metadata) of the cube such as which cities lie in a certain state. It is important to point out that by using mappings to express the inclusion relationships the encoding scheme can also be applied to dimensions whose hierarchy level domains do not have an inherent sort order (e.g. product or location dimension).

We can formalize the partitioning process as follows:

Definition 3.1. For a set of data values V , if $\exists V_1, V_2, \dots, V_k$, where $V_i \subset V, i = 1, 2, \dots, k$, such that the two conditions $V_i \cap V_j = \emptyset, i, j = 1, 2, \dots, k, i \neq j$ and $\cup V_i = V, i = 1, 2, \dots, k$ hold, then V_1, V_2, \dots, V_k is called a *partition* of V .

Let us illustrate this approach using the Time dimension and its commonly used hierarchy “year-quarter-month.” We first sort the values in each hierarchy level and map them into integers using the sort order. Note that we must qualify quarters and months with the respective year to eliminate ambiguities. For instance, we map 1999/JAN to 1, 1999/FEB to 2, 2000/JAN to 13, 1999/Q1 to 1, 1999 to 1 and so on. Next, starting from the bottom of the hierarchy, we partition the domain values of the month attribute into quarters and record the mappings in an inclusion relationship. We repeat this process for the other hierarchical attributes and arrive at the conceptual representation shown in figure 3. Notice, besides the domain value mappings and the inclusion relations, we only need to store the attribute values representing the finest level of granularity (e.g. months in the previous example); all other attribute values can be inferred from the existing metadata. This may result in a potentially *large compression of dimensional data*.

3.3. Families of statistics trees

Using a single ST to represent the desired aggregates for the relevant dimensions at the lowest level of granularity is inefficient in terms of space and query processing time: to answer a query using a single ST, one must first transform the conditions of the query, which could be formulated using different levels of granularity for each dimension hierarchy, into equivalent conditions involving only the lowest levels (those that are stored in the tree). Rewriting an original query in this way may produce one or more range or partial queries.

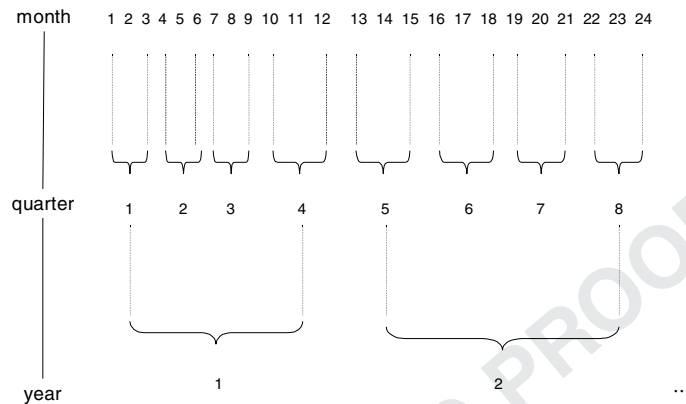


Figure 3. Representation for the month-quarter-year hierarchy of the time dimension.

This simple approach to answering cube queries has been described in [12]. However, this is obviously inefficient. Consider, for example, an ST representing summarized car sales along time, location, and product dimensions, all of which represented at the finest level of granularity: day for the time dimension, city for the location dimension and model for the product dimension. Answering the query “How many Toyotas were sold in the southeast in 2000?” would be equivalent to answering $366 * 150 * 1 = 54,900$ singleton queries on the base tree if we assume that the southeast region is made up of 150 cities and that the year 2000 has 366 days. In addition, since the ST may be too large to fit into memory, these queries will result in a large number of I/Os.

One way to reduce the I/O cost in this example is to transmit only the leaves of the ST. The internal nodes can be generated in memory without additional I/O using only the metadata about the ST such as number of dimensions, their cardinalities, existing hierarchies, etc. In this way, the I/O complexity for STs is comparable to multidimensional arrays. A better approach is to materialize *multiple, smaller* STs representing different dimensions at different levels of granularity. For instance, in the car sales example, if we materialize an ST representing only year and region data, the sample cube query above becomes a singleton query and the I/O cost for answering this query against the smaller ST is negligible.

We select, compute and materialize multiple smaller trees containing subsets of the data cube as follows. Starting from the ST representing the selected dimensions at the lowest level of granularity, we generate a set of smaller STs, each representing a certain combination of the base dimensions and hierarchy levels.² We term this single ST *base tree* and the smaller trees *derived trees*. In most cases, cube queries can be answered using one of the derived trees rather than the base tree. A base tree and its derived trees form a *family of statistics trees* with respect to the base tree. To generate a family, we first choose from the set of all possible candidate trees (i.e., all combinations of represented dimensions and corresponding hierarchy levels) those derived trees that eventually make up the family. For a k -dimensional base tree with L_1, L_2, \dots, L_k , levels respectively the

total number of candidate trees is $\prod_{i=1}^k L_i$, where L_i represents the number of hierarchy levels for the i th dimension. Please note that throughout the paper, we are assuming that each dimension has a single hierarchy. However, our approach and calculations can be extended to those cases where dimensions have multiple hierarchies by representing such dimensions as using multiple instances of the same dimension with a single hierarchy. We now show how we can compute and materialize the derived trees from the base tree.

3.3.1. A greedy algorithm for selecting family members. For a given set of dimensions, any tree representing a certain combination of hierarchy levels can be selected as a family member. However, we need to consider space limitations and maintenance overhead. We introduce a greedy algorithm to choose the members in a step-by-step fashion as follows. Starting with a base tree, we “roll-up” the dimension with the largest cardinality to its next higher level in the hierarchy, keeping other dimensions the same. This newly formed ST, whose size represents only a small fraction of the base tree, becomes a new member of the family. We continue this derivation process until each dimension of the base tree is rolled-up to its highest level or until the combined size of all trees in the family exceeds the maximum available space.

The pseudo-code for selecting a family of statistics trees is shown in Algorithm 2. In line 1, the metadata for the base tree is retrieved from the metadata repository. The rollup procedure in line 5 is outlined in the next subsection. Line 6 stores the leaf nodes of the newly derived tree on disk since the internal nodes can be re-created on the fly. The ST metadata (e.g. dimensions, hierarchy levels), which is stored in the repository (line 7), is used by the cube query processor to match incoming queries against STs. In addition to the stop conditions mentioned earlier, other possible stop conditions in line 9 are: total amount of space used up by the family and the query cost for certain important (frequently asked) queries, to name a few. Without space limitation, this greedy algorithm will generate $\sum_{i=1}^k (L_i - 1)$ candidate trees. All rollup operations are performed in memory. The dominant variable of the algorithm is the time to write each derived tree to disk.

Let us illustrate Algorithm 2 with an example. Suppose we want to represent a data set with three dimensions d_1 , d_2 , and d_3 whose cardinalities are 100 each. Each dimension

```

1 Retrieve metadata for base tree T0;
2 T = T0;
3 Repeat {
4   Find dimension with largest cardinality in T;
5   Rollup dimension to generate a new derived Tree T;
6   Save T;
7   Save T's metadata to Metadata Repository;
8 }
9 Until (stop conditions);

```

Algorithm 2. Sketch of greedy algorithm for selecting a family of statistics trees.

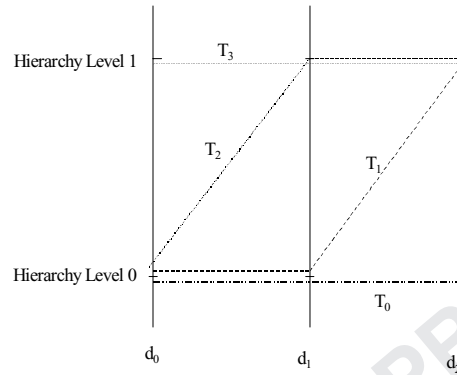


Figure 4. A family of STs consisting of $\{T_0, T_1, T_2, T_3\}$.

contains a two-level hierarchy, which organizes the 100 elements into ten groups of ten values each. The base ST T_0 for this data set has three levels; the degrees of its *internal* nodes are 101 (100 values plus 1 star). The selected derived trees T_1 , T_2 , and T_3 that make up a possible family together with T_0 are as follows: T_1 is the result of rolling up d_2 in T_0 (since all three dimensions are of the same size, we pick one at random). T_1 has degrees 101, 101 and 11 for the first, the second, and the third level nodes. T_2 is the result of rolling up d_1 in T_1 . Its degrees are 101, 11, and 11. T_3 is the result of rolling up d_0 in T_2 . Its degrees are 11, 11, and 11. The generated family is shown in figure 4.

3.3.2. Computing derived trees. Each derived tree can be materialized from the base tree without having to use the detailed data set. Before introducing our tree derivation algorithm, we first define a “merge” operator called “ \oplus ” on STs as follows:

Definition 3.2. Two STs are *isomorphic* if they have the same structure except for the values of their leaves. *Merging* two isomorphic ST’s S_1, S_2 results in a new ST S that has the same structure as S_1 or S_2 but its leaf values contain the *aggregated* value of the corresponding leaf values of S_1 and S_2 . This relationship is denoted by $S = S_1 \oplus S_2$.

In order to derive a new tree, the derivation algorithm proceeds from the root to the level (i.e., the dimension) that is being rolled up. Here, we reorganize and merge the sub-trees for all the nodes in that level to form new subtrees. For each node, we adjust the degree and update its pointers to reference the newly created subtrees.

Suppose that during the derivation, we need to rollup a dimension that has nine values and forms a two-level hierarchy consisting of three groups with three values each. A sample node N of this dimension with children S_1, \dots, S_9, S^* is shown on the left of figure 5. Note we use the superscripted hash mark to indicate values corresponding to the second hierarchy level. For example, $1^\#$ represents values 1 through 3, $2^\#$ represents values 4 through 6 etc. To roll-up node N , we merge its nine subtrees into three new subtrees $S_1^\#, S_2^\#$ and $S_3^\#$ (the

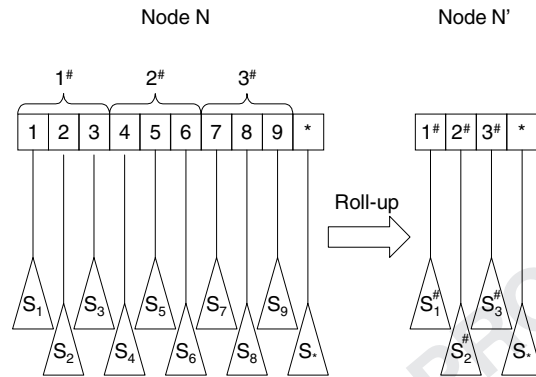


Figure 5. Rolling-up a dimension of a statistics tree.

subtree representing the star pointer remains the same). The new node N' with degree four is shown on the right side of figure 5. Using the definition from above, we can see that the following relationships hold among the subtrees of nodes N and N' :

$$S_{1\#} = S_1 \oplus S_2 \oplus S_3, S_{2\#} = S_4 \oplus S_5 \oplus S_6, \text{ and } S_{3\#} = S_7 \oplus S_8 \oplus S_9.$$

The pseudocode for the rollup algorithm is shown in Algorithm 3. Line 1 is the signature of the recursive function `rollUp()`, which proceeds from a node `nd` at level `level` to the rollup dimension `rollDim`. The variable `factor` indicates the grouping factor, i.e., it indicates how many low-level values make up a high-level value during the roll-up process. Hence, the new hierarchy is formed by grouping the lower level values into groups of size `factor`. Lines 2 to 9 correspond to the roll-up process described in figure 5. Line 5 performs the actual merge operation according to the merge operator “ \oplus ” using the parameterized function `merge_op` (e.g. COUNT). For example, in the case of count, the leaves of a new subtree represent the summation of the corresponding leaves of the merged subtrees. Line 13 invokes `rollUp()` on the root of the statistics tree whose dimensions are to be rolled up.

```

1 rollUp(Node nd, int level, int rollDim, int factor, Function merge_op)
  {
2   IF level == rollDim THEN
3     group children of nd such that each group has
      factor subtrees except possibly the last group and star node;
5   merge the subtrees in each group using the merge function;
6   make new node nd' whose children represent merged subtrees incl. star node;
7   adjust domain of node nd';
8   link nd' to nd's original parent IF level not equal to 1;
9   return;
10  level++;
11  FOR each child Ci of nd, rollUp(Ci, level, rollDim, factor, merge_op);
12  }
13 rollUp(root, 1, rollDim, factor, merge_op);

```

Algorithm 3. Rolling-up a dimension.

4. The CUBE query language CQL

In ROLAP systems, queries against the data cube are transferred into SQL statements against the underlying relational warehouse schema (most often represented as a star schema) and are then evaluated by the relational engine. To illustrate, assume that we have a relational warehouse containing car sales data organized as shown in the star schema in figure 6. CarSales is the fact table connecting to four dimension tables Manufacturer, Product, Time, and Location. Primary keys are underlined; the remaining attributes except Price in the fact table CarSales serve as foreign keys into the dimension tables.

Using this sample schema, our sample query “How many Toyotas have been sold in Florida and Georgia between January and March of this year?” can be expressed in SQL using the count aggregate function, assuming that each record represents the sale of only one car:

```
SELECT    COUNT(*)
FROM      Car_Sales, Manufacturer, Time, Location
WHERE     Car_Sales.ManufactID = Manufacturer.ManufactID AND
          Car_Sales.TimeID = Time.TimeID AND
          Car_Sales.LocationID = Location.LocationID AND
          Name='Toyota' AND Month IN {'JAN', 'FEB', 'MAR'} AND
          State IN {'Florida', 'Georgia'};
```

In contrast, we represent the same query in the following simpler format:

```
COUNT(Name:Toyota; month:[JAN,MAR]; state:{Florida,Georgia})
```

Here, count represents the aggregate function which is applied to the tuples in the fact table satisfying the conditions inside the parenthesis: a colon separates the attribute name to its left from the constraint to its right, a semicolon means “AND” and two brackets represent “BETWEEN... AND”. If we apply a domain value mapping as described in Section 3 and assume that the dimensions are ordered from 0 to 3 starting with Manufacturer, followed

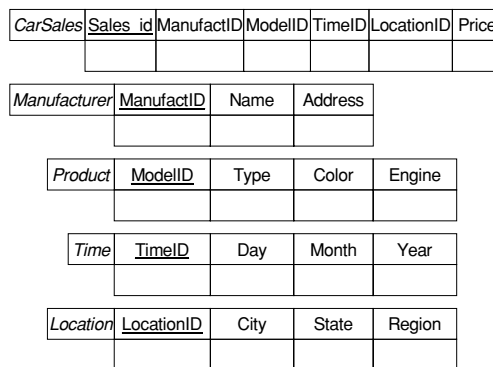


Figure 6. Sample car sales star schema.

by `Product`, `Time` and `Location`, we can represent the above query even more compactly as:

```
COUNT(0:4; 2:[1,3]; 3:{1,3}).
```

Here, `0:4` means select the fourth value of dimension zero (i.e., Toyota from the `Manufacturer` dimension), and so forth. This relatively simple example is chosen to illustrate the basic idea behind our cube query representation. In Section 4.2 we provide a more complex example, which demonstrates the capabilities of our representation when embedded in a host language.

4.1. A formal representation for cube queries

Recall that the conceptual data model underlying the theory described in this report is the multidimensional data model or data cube model. A 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 set to track aggregated metrics of interest. The cells of the cube contain the values of the measure attributes for the selected of dimensions.

Definition 4.1. A cell of a k -dimensional data cube with attributes A_1, A_2, \dots, A_k with cardinalities d_1, d_2, \dots, d_k respectively, is the smallest full-dimensional cube at a point $P = (x_1, x_2, \dots, x_k)$, $x_i \in [1..d_i]$ (i.e., each dimension is constrained by a value in its domain). We note that the total number of cells in a k -dimensional data cube is $\prod_{i=1}^k d_i$.

Definition 4.2. A cube query is an aggregate operation that is performed on the cells of a k -dimensional data cube. The cells (or region) on which aggregation is performed are defined by a set of constraints in the query. The selected cells on which to aggregate are subsets of the full data cube. Formally, the constraints are in the form of r -tuples $(s_{i_1}, s_{i_2}, \dots, s_{i_r})$, where $\{i_1, i_2, \dots, i_r\} \subseteq \{1, 2, \dots, k\}$ and r is the number of dimensions specified in the query. Each constraint s_{i_j} selects a subset of the domain underlying the w th dimension (let $w = i_j$).

A constraint can be a

1. *partial selection*, $\{t_1, t_2, \dots, t_r\}$, $2 \leq r < d_w$, $t_i \in \{1, 2, \dots, d_w\}$, specifying any subset of the domain values for dimension i_j ,
2. *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. *single value* a , $a \in [1..d_w]$ or the special `*` value selecting the entire domain.

Notice that if a partial selection contains all domain values or a range spans all values, their representation can be simplified by a `*`. Further note, when $r < k$, some of the dimensions are collapsed, meaning the entire domain is selected. 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 ‘*’.

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. Analogously, a query is a *range query* iff $\exists j$ such that s_{i_j} is a range selection and does not include a partial selection. Finally, a query is a *singleton query* iff $\forall j s_{i_j}$ is a singleton value including the ‘*’ value. It is worth noting that a singleton query represents a single cell or a result of a “slice” through the cube.

Definition 4.4. The *degree* r of a query q is the number of dimensions specified in the query not including ‘*’ values.

We have termed our new formalism *Cube Query Language (CQL)*. Using CQL, a cube query is represented as:

AGGR-OPERATOR^{MEASURE} (D-INDEX, H-LEVEL) : SELECTED-VALUES; . . .)

Here AGGR-OPERATOR represents the aggregate operation type and MEASURE the aggregation attribute (usually numeric). The information about the measure can be dropped in conjunction with the count operator since count computes the total number of facts in the fact table independent of the measure. The constraints, which are separated by semicolons, appear in the body of the query between the parentheses. Each constraint contains the index of the constrained dimension (D-INDEX) as well as its hierarchy level (H-LEVEL). The selected values can be specified as a single value, a range, or a partial selection. If there is no constraint (null constraint) on a dimension, it can be omitted or represented by a “*”. The hierarchy level can be omitted if the constrained dimension is non-hierarchical or the selected values are at the lowest level of the dimension. In this case, the constraint is simplified as D-INDEX: SELECTED VALUES. The BNF for CQL is shown in the Appendix.

Continuing our car sales example, assume that for the Time dimension, the attributes Day, Month and Year make up hierarchy levels 0, 1 and 2 respectively, and for the Location dimension, attributes City, State and Region form hierarchy levels 0, 1 and 2 respectively. The CQL expressions shown in Table 1 are syntactically correct. For clarity, we are using actual schema names and values rather than their integer encodings. Note, the last query is a range query of degree $r = 2$. It is in its *normal form* since all dimension constraints are present; the Product and Manufacturer dimensions, which have no constraints, are represented by “*”.

4.2. Characteristics of CQL

In comparison with query languages in existing ROLAP and MOLAP systems, CQL has several important advantages. First, CQL is *intuitive* and *concise*. CQL represents the essential elements of a cube query in the simplest possible format: information about the measure, aggregate operator, and selected regions of the cube are defined by constraints on the dimensions and their hierarchies. Our representation of the selected domain values is

Table 1. Sample CQL queries.

<code>q = COUNT()</code>	Find the total number of cars sold in all states for all years
<code>q = COUNT(; ;)</code>	Same as above
<code>q = SUM^{Price}(City:Orlando)</code>	Find the total sales price for all cars ever sold in Orlando
<code>q = COUNT(Color:red; sedan; (Location,region):SE)</code>	Find the total number of red sedans ever sold in the southeast region
<code>q = SUM^{Price}(*; *;</code> <code>(Time,Month): [MAR/2000,OCT/2000];</code> <code>(Location,City):NYC)</code>	Find the total sales price for all cars sold in NYC from March through October 2000

flexible, allowing a single value, a range of values, or any subset of values. In contrast, in many ROLAP systems, cube queries are often represented by SQL statements, which are significantly more complex than the equivalent CQL expressions. We have seen a comparison between a simple CQL query and its SQL counterpart at the beginning of Section 4.1. Additional comparisons with languages used by MOLAP systems later in this subsection support this point.

Second, CQL is *expressive*. As we showed in the sample queries on the previous page, CQL can easily represent aggregations of measures over any subset or region of the data cube using different levels of granularities. Third, CQL can be *easily integrated into a programming environment*. For example, to find interesting patterns in decision support systems, analysts' requests are usually unpredictable and can be very complicated [2]. For example, in the car sales warehouse, an analyst might want to know which manufacturer is growing more rapidly in terms of market share change by car types as in the following query: "For each of type of car, list the market share change in terms of car sales for each manufacturer between January 1999 and January 2000." This query cannot be represented in one CQL expression. However, we can embed CQL expressions in a host language (e.g., C, C++, JAVA) and solve the problem in a procedural fashion using the power of the host language. For example, in CubiST⁺⁺, a preprocessor transforms the CQL expressions into primitive calls of the host language (C++).

The pseudocode to compute the request above is shown in figure 7. The embedded CQL statements (lines 2, 3, 5, and 6) represent functions returning aggregates. For comparison, an equivalent procedure using Oracle EXPRESS [29] is given in figure 8. Lines 2–5 declare local variables; lines 7–8 specify the ranges of the loop variables (i.e. the type and manufacturer dimensions) of the two nested FOR loops below since one must limit each dimension to the desired values before executing the FOR command. Notice that we can implement the SQL group-by statement in a similar fashion.

Lastly, CQL is *easy to implement*. Unlike SQL or MOLAP languages such as Oracle EXPRESS, which contain many constructs, CQL's syntax is sparse. Based on the implementation of a CQL query processor for our prototype, we believe that implementing a fully

```

1 FOR t = 1 TO num_carTypes DO {
2   S1 = SUM price (Type:t; Month:1999/JAN);
3   S2 = SUM price (Type:t; Month:2000/JAN);
4   FOR m = 1 TO num_makers DO {
5     A=SUM price (Name:m;
6               Type:t;
7               Month:1999/JAN);
8     B=SUM price (Name:m;
9               Type:t;
10              Month:2000/JAN);
11    OUTPUT m, t, B/S2 - A/S1;
12  }
13 }

```

Figure 7. Using embedded CQL to compute an advanced cube query.

```

1  DEFINE MARKET_SHARE_CHANGE PROGRAM
2  Variable S1 integer
3  Variable S2 integer
4  Variable A integer
5  Variable B integer
6  PROGRAM
7  Limit type to all
8  Limit manufacturer to all
9  For type
10   do
11     limit month to '1999/JAN'
12     S1 = total(price)
13     limit month to '2000/JAN'
14     S2 = total(price)
15     heading under '=' valonly type
16     for manufacturer
17     do
18       limit month to '1999/JAN'
19       A = total(price)
20       limit month to '2000/JAN'
21       B = total(price)
22       row manufacturer, B/S2-A/S1
23     doend
24   doend
25 END

```

Figure 8. Advanced cube query written in Oracle's Express language.

functional CQL query processor is less difficult (e.g., compared to EXPRESS) due to the simplicity of the syntax as well as the simpler query optimization, which is described in Section 5.

5. Answering CUBE queries using families of trees

Given a CQL query, we first obtain its measure, type of aggregate operation, and the underlying dimensions. This metadata information is matched against the corresponding information of the available base trees since a particular CubiST⁺⁺ installation may contain multiple families. Our query matching algorithm selects the family and within a family, the smallest ST that can provide the answer to the query. This ST is called *optimal derived tree* and is defined formally in the next subsection. Finally, we evaluate the submitted query on the selected ST using our query-answering algorithm. This may require a rewrite of the query in case there is no ST that exactly matches the query (partial match).

5.1. Choosing the optimal derived tree

We select the optimal derived tree using the matching scheme described below. Our selection algorithm uses a tree matrix, which represents the characteristics for a family of STs.

Definition 5.1. A tree matrix (TM) for a given family is a matrix in which rows represent dimensions, columns represent the hierarchy levels, and the row/column intersections contain the labels of the corresponding ST's in the family. Intuitively, a TM describes what STs have been materialized in the hierarchies.

Recall the sample family of STs from the example in Section 3.3, which represents a 3-dimensional data set with a 2-level hierarchy for each dimension. In this family, T_1 is a derived tree, which represents dimensions d_1 and d_2 at their lowest level (level 0) and dimension d_3 at the rolled-up level (level 1). Hence, T_1 is placed in rows 1 and 2 of column 1 and in row 3 of column 2 of the corresponding TM as shown in figure 9. Similarly, T_3 is placed in column 2 since all its three dimensions are at level 1.

Level of Abstraction Dimension	Level 0	Level 1
1	T_0, T_1, T_2	T_3
2	T_0, T_1	T_2, T_3
3	T_0	T_1, T_2, T_3

Figure 9. Sample tree matrix for the family of STs introduced in Section 3.3.

T_3	(1, 1, 1)
T_2	(0, 1, 1)
T_1	(0, 0, 1)
T_0	(0, 0, 0)

Figure 10. Stack representation of the TM in figure 9.

We have implemented the TM as a stack of vectors, each representing the abstraction level for each dimension by tree. For example, T_3 represents an ST in which all dimensions are rolled-up to level 1. Popping off a layer from the stack is equivalent to removing the right-most ST from the TM. As we will show in the next section, it may be necessary to remove a tree from the TM during query evaluation, if it does not provide enough detail for the given query. The stack corresponding to the TM above is shown in figure 10.

We now introduce a new way to represent hierarchies in cube queries to facilitate matching.

Definition 5.2. A query hierarchy vector (QHV) is a vector (l_1, l_2, \dots, l_k) , where l_i is the hierarchical level value of the i th dimension in the query. If dimension j does not have a constraint, then $l_j = L_j - 1$.

Definition 5.2 implies that we choose the smallest tree (i.e. the highest level for dimension j) to answer the query.

Definition 5.3. A QHV (l_1, l_2, \dots, l_k) matches view T in the TM iff $l_i \geq$ column index of the i th row entry of T in TM, $\forall i, i = 1, 2, \dots, k$. In other words, T 's column index vector, which is composed of T 's column indices, is less than QHV.

Definition 5.4. An optimal matching tree with respect to a query is the tree in TM that has the largest index number and matches the query's QHV.

Consider the sample cube query $q = \text{count}((0,1):3; 1:2; (2,1):5)$, which computes the total number of records that contain the 3rd and 5th domain value of the first-level hierarchy for dimensions 0 and 2, and the 2nd domain value for the bottom-level hierarchy from dimension 1. Its QHV is $(1, 0, 1)$. Since the column vector for T_3 is $(1, 1, 1)$ which does not match $(1, 0, 1)$, T_3 is removed. In the same way, T_2 is also removed. On the other hand, the column index vector of T_1 is $(0, 0, 1)$ is less than $(1, 0, 1)$. Hence T_1 matches QHV and T_1 constitutes the optimal matching tree and will be used to answer q . The resulting TM is shown in figure 11.

Level of Abstraction Dimension	Level 0	Level 1
1	T_0, T_1	—
2	T_0, T_1	—
3	T_0	T_1

Figure 11. TM after removing T_2 and T_3 .

```

1 match(String queryCQL) {
2   Find the base tree of the family that matches
   the measure, constraint dimensions, and aggregator of the query;
3   Find the TM stack; Compute QHV for the query;
4   IF TM.top <= QHV, return the ST whose column index vector is top.
5   REPEAT
6     TM.pop();
7   UNTIL (TM.top <= QHV);
8   RETURN the ST whose column index vector is the top of TM;
9 }

```

Algorithm 4. Finding the smallest matching ST for a given query.

The pseudo-code for finding the optimal matching tree is shown in Algorithm 4. From the CQL expression, we extract the level values as QHV. The measure, aggregation operator, and constrained dimensions are matched with the metadata generated during the family generation phase (line 7 in Algorithm 2). The tree matrix TM in line 3 was also generated at that time.

5.2. Processing CQL queries

Our recursive query evaluation algorithm assumes that the input query is given in its internal form using the matrix representation $q = Z[i, j]$ where each $z_{i,j} \in [1..d_i] \mid i = 1, 2, \dots, k, j = 1, 2, \dots, d_i$. Here $z_{i,j}$ refers to the j th selected value of the i th dimension, d_i is the cardinality of dimension i and k is the total number of dimensions. This matrix representation covers all the aforementioned three types of queries, namely partial, range, and single value queries.

The core of our query evaluation algorithm is a recursive function $\text{Cubist}()$ that returns the aggregated value of the leaves of a subtree rooted at node n of the selected ST. The pseudocode for $\text{Cubist}()$ evaluating a count query is shown as Algorithm 5. We can design $\text{Cubist}()$ for other aggregation functions similarly by modifying Lines 3 and 6 accordingly. Line 1 is the signature of $\text{Cubist}()$, which returns the sum of all the related leaves of the subtree rooted at node n at level “level” for query q . Invoking $\text{Cubist}()$ in line 8 on the root of the ST produces the query result. Lines 2–4 represent the base case.

```

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( $z_{level-1,1}^{th}$  child of n,q,level)+cubist( $z_{level-1,2}^{th}$  child of n,q,level)+ ...
7 } // end cubist
8 Query_result := cubist(root,q,1);

```

Algorithm 5. Recursive algorithm for evaluating cube queries.

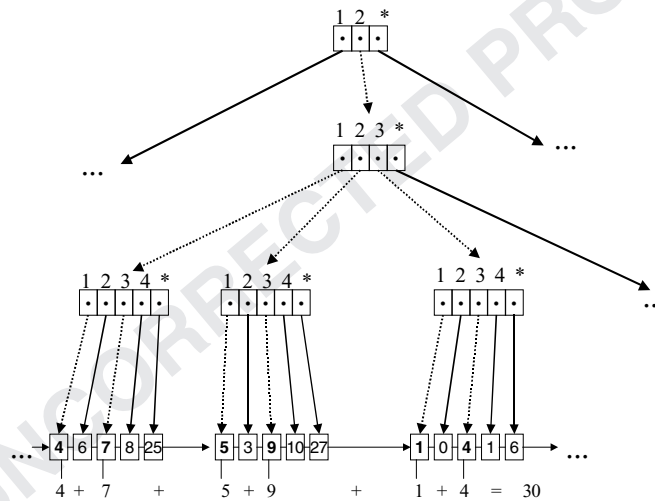


Figure 12. Answering query $q = \text{COUNT}(0:2;1:[1,3];2:\{1,3\})$.

Lines 3 and 6, sum up the related leaves of the subtrees bounded by the selected domain values in the query.

To answer a singleton query, $\text{Cubist}()$ needs to follow a path from root to the single leaf by touching k nodes. However, in order to answer an arbitrary cube query, most of nodes of the tree have to be visited. The time to answer a singleton query is $O(k)$ (k is number of dimensions); the worst case running time for answering an arbitrary query is $O(\text{size of ST})$.

Let us illustrate our query-processing algorithm with an example. Suppose figure 12 shows the ST from figure 2 after the data set has been loaded. To answer the query:

$$q = \text{COUNT}(0:2; 1:[1, 3]; 2:\{1, 3\}),$$

we follow the second pointer of the root to the node of level 2 and from there the first three pointers to the nodes of level 3. Finally, from each of the nodes at level 3, we follow the first and the third pointers to the leaves that contain the pre-computed aggregates shown in

bold. The summation of the count values in these leaves ($4 + 7 + 5 + 9 + 1 + 4 = 30$) is the final answer to the query. The paths taken by Cubist () are denoted as dashed lines.

6. Experimental prototype and evaluation

We have performed extensive experiments to validate the performance claims about CubiST⁺⁺ made earlier in this report. The following sections describe the results of our experiments including the CubiST⁺⁺ prototype system that was used as our testbed.

6.1. Description of the prototype

The conceptual architecture of the Cubist⁺⁺ prototype system is shown in figure 13. The architecture can be divided into built-time and run-time components, which correspond roughly to the left side and right side of the figure respectively. Activities that fall under the built-time are creating and loading a base ST and deriving a family of ST's corresponding to a base ST. The only run-time activity supported by the prototype is the execution of CQL statements.

Starting with the built-time, The *Base Tree Initializer & Loader* module lets the administrative user define a data cube (i.e., a base ST) given its measure, dimension info (incl. hierarchies), and underlying aggregate function(s). The language of defining data cubes, dimensions, hierarchies, etc. is called *Cube Definition Language (CDL)*. This metadata is stored in the *ST Repository* shown at the bottom left of figure 13. The data that is used to load the tree is generated by the *DBGEN Data Generator* in the upper left-hand corner. The majority of our experiments so far have been conducted with synthetic data sets based on the TPC-H benchmark [37]. The TPC-H data sets, which consist of one fact file and one

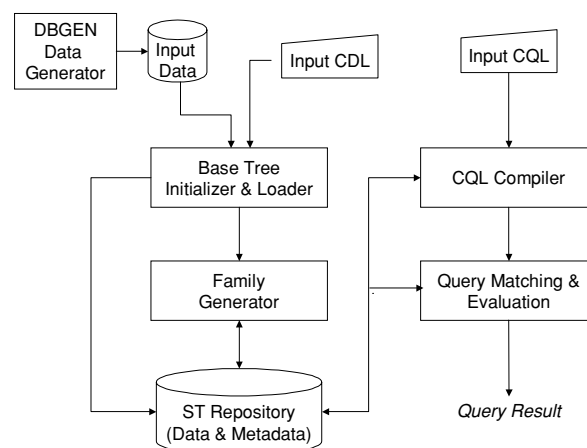


Figure 13. Overview of the conceptual architecture of the Cubist⁺⁺ prototype system.

or more dimension files, are stored on disk and serve as input to the base tree initialization routine. Loading is a step-wise process, which starts with the scanning of the dimension files and their hierarchies (if any). During this time, the loader routine produces the domain value mappings and, in case hierarchies are present, the inclusion relationships. The mappings and relationships are needed during the loading of the fact records, which are used to update the aggregate value(s) in the leaf nodes. Using the base tree, the *Family Generator* module derives the remaining STs that make up the family using the selection and derivation algorithms for STs described in Section 3.

During run-time, an incoming CQL query is parsed and validated against the metadata of the available data cubes by the *CQL Compiler*. If the input statement is valid, the query is converted into its internal matrix representation which is used by the *Query Matching & Processing* module to retrieve an optimally matching ST. This is done using the query matching algorithm introduced in Section 5: specifically, the dimensions and aggregate operators specified in the query must match the dimensions on which the tree is constructed as well as the aggregate operators used to generate the contents of the leaf nodes. The query is processed using Algorithm 5 and the result is returned to the user. All modules with the exception of the DBGEN have been implemented using Java. Input files are stored in ASCII format on disk. The ST Repository, which is implemented as a collection of operating system files, stores STs (the leaf nodes only) and their related metadata.

6.2. Testbed and data sets

Our infrastructure consists of a Dell Dimension XPS 450 Pentium II with 256 MB of RAM running Windows NT Server 4.0 and a SUN Enterprise E450 workstation with two 248 MHZ UltraSparc processors and 640 MB of RAM running SunOS 5.8. The two separate platforms were necessary due to the hardware requirements of our two benchmark applications: the experiments involving a commercial MOLAP server required Windows NT (described in Section 6.6), all other experiments including the comparison against a commercial ROLAP system were performed on the Sun workstation. Note, since our experiments involving the commercial products are only unofficial, i.e., we did not contact the manufacturer to obtain feedback, support or even permission, we will not reveal the product and manufacturer names.

The data sets for the experiments were generated using a random number generator and TPC's DBGEN using the TPC-H benchmark [36]. Table 2 summarizes the characteristics of our ten data sets: *data set size* specifies the total amount of data in kilobytes, r denotes the number of records in thousands in the fact table of the TPC-H benchmark database, k the number of dimensions, and d_i the cardinality for each dimension i , $1 \leq i \leq k$. The values in data sets D1.x and D2.x are uniformly distributed random numbers. Data sets D3.1 through D3.3 are variations of the "lineitem" table from the TPC-H benchmark with scale factor 1 (6,001,215 records): We projected out the four columns "l_returnflag", "l_linestatus", "l_shipdate", and "l_commitdate" and restricted the total number of records to 100 K, 500 K, and 1,000 K resulting in data sets D3.1, D3.2, and D3.3 respectively.

The experiments are setup as follows: In Section 6.3 we validate the advantage of using family of trees rather than a single base tree (CubiST .() [11]) to answer a variety of

Table 2. Statistics for the synthetic data sets used in the experiments.

Data set ID	Data set size (KB)	r (K)	k	d_i
D1.1	300	100	3	60, 60, 60
D1.2	300	100	3	20, 30, 30
D1.3	300	100	3	15, 15, 20
D2.1	500	100	5	10, 10, 10, 10, 10
D2.2	2,500	500	5	10, 10, 10, 10, 10
D2.3	5,000	1000	5	10, 10, 10, 10, 10
D2.4	10,000	2000	5	10, 10, 10, 10, 10
D3.1	1,330	100	4	3, 2, 2557, 2537
D3.2	7,020	500	4	3, 2, 2557, 2537
D3.3	13,300	1000	4	3, 2, 2557, 2537

randomly selected cube queries. In Section 6.4, we compare the setup and response times of CubiST⁺⁺ with those of a bitmap-based query evaluation algorithm, henceforth referred to as *BITMAP*. In Sections 6.5 and 6.6, we compare CubiST⁺⁺ with commercial ROLAP and MOLAP servers respectively.

6.3. CubiST⁺⁺ vs. cubist

This set of experiments demonstrates the effectiveness of materializing and using families of the statistics trees to answer a variety of cube queries. In data sets D1, each of the three dimensions has a two-level hierarchy. To simplify the hierarchical inclusion relationships, we assume that each higher-level value includes an equal number of lower level values; this number is referred to as *factor*. In general, the partitions are not even and the runtimes of queries can vary slightly depending on the constraints in the query. This does not limit the generality of the results measured in this experiment.

Throughout this set of experiments, we have queried the same set of cells represented by the following three queries for the factors 1, 2, 3 on d_3 respectively:

$$q_1 = \text{count}((2, 0) : [0, 5]), \quad q_2 = \text{count}((2, 1) : [0, 2]), \quad \text{and} \\ q_3 = \text{count}((2, 1) : [0, 1]).$$

We have varied the size of the factors. For our evaluation, we compare the sizes of the STs in terms of the number of leaves, the I/O time spent on loading the ST from the ST repository and the total answering time. Looking at Table 3, we can see that the I/O time and the total answer time decrease dramatically as the factor size increases. The more low-level values are included in a high level value (i.e., the larger the factor size), the smaller derived trees that are based on the higher-level hierarchies and the faster to answer a query.

The total space consumed by the derived trees is no more than that of the base tree. However, by materializing additional STs with higher aggregations, we can speedup those queries whose aggregation levels correspond to one of the derived trees. If a query does not

Table 3. Query processing times for different STs using data set D1.

Data set ID	D1.1	D1.2	D1.3
Factors	1, 1, 1	3, 2, 2	4, 4, 3
d_1, d_2, d_3	60, 60, 60	20, 30, 30	15, 15, 20
ST size (# leaf nodes)	226, 981	20, 181	5, 376
I/O (ms)	197	15	4
Running time(ms)	5,087	55	21

match one of the available derived STs, we choose the next-best (more detailed) available family member to answer it. Answering these unmatched queries using one of the members of the family is still fast since the derived trees are chosen in a greedy fashion, allowing CubiST⁺⁺ to trade-off the space needed to store the derived trees and time to process incoming cube queries.

6.4. CubiST⁺⁺ vs. BITMAP

In this set of experiments, we compare the setup and response times of CubiST⁺⁺ against a bitmap-based algorithm (*BITMAP*). The data organization of *BITMAP*-based algorithms has been briefly described in Section 2.3. For small cardinality domains, *BITMAP* is a flexible and very efficient algorithm for computing cube queries. It is faster than other row-level alternatives. *BITMAP* is widely incorporated in many ROLAP systems. Its query evaluation algorithm uses a bitmap index structure and bit-wise logical operations and can compute cube queries without touching the original data set (assuming the bit vectors are already setup).

In the experiments, which use data sets D2.1 through D2.4, the number of dimensions and corresponding domain sizes are fixed while the number of fact records increases. The goal is to validate the scalability of CubiST⁺⁺ in terms of the number of records. Figure 14 shows the performance of the two algorithms using cube query: $q = \text{count}(0: [1, 6]; 1: [1, 6]; 2: [1, 6]; 3: [1, 6]; 4: [1, 6])$. This query 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 [1, 6]. Although the setup time for CubiST⁺⁺ is larger than that of *BITMAP* (mainly due to the time it takes to setup the ST), its response time is faster and independent of the number of input records.

Our response time measurements clearly show the linear time characteristics of the bitmap-based algorithm compared to the almost constant behavior of CubiST⁺⁺. In order to illustrate the storage requirements of the bitmap-based approach, consider the following. Suppose we use one byte to represent a Boolean bit. The required space for storing the bitmaps of 500,000 to 2,000,000 records in data sets D2.1 through D2.4 increases from 25 MB ($500,000 * 5 * 10$) to 100 MB. When the number of records increases beyond a certain threshold (i.e., when the bitmaps cannot fit into memory), *BITMAP* must first write to and subsequently read the bitmaps back from disk. This significantly degrades the performance of this approach to the point where we have to avoid bitmaps when the

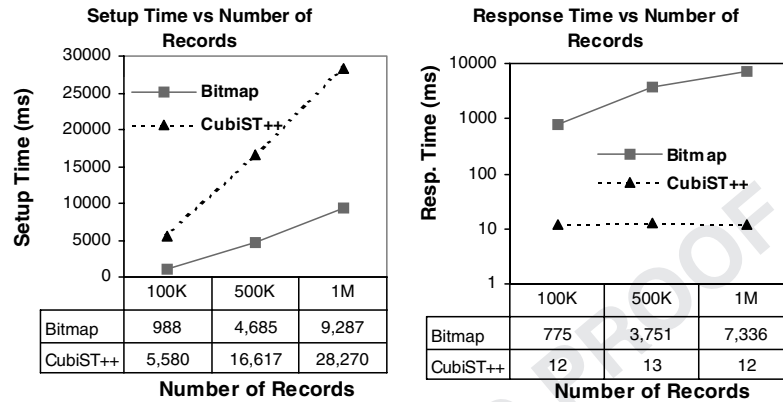


Figure 14. Setup and response times vs. number of input records for CubiST⁺⁺ and BITMAP.

cardinalities and number of records are large (e.g. more than 32 domain values). From the results of this experiment, we can also see that the response time of CubiST⁺⁺ is independent of the number of records, assuming the ST that is used to answer the query fits into memory.

6.5. CubiST⁺⁺ vs. commercial ROLAP server

In this subsection, we use data sets D3.x, which represent the restricted fact table of the TPC-H benchmark database as described in Section 6.2. The pricing summary query Q1 from the TPC-H benchmark (see [36]) reports the amount of business that was billed, shipped, and returned. To simplify the experiment, we only use the COUNT operator and limit the attributes in the table “lineitem” to “l_returnflag”, “l_linestatus”, “l_shipdate”, and “l_commitdate.” The SUM and AVG operators on measures such as “l_quantity”, “l_extendedprice”, etc. can be computed similarly as in our initialization and execution algorithms described earlier. The SQL syntax for the modified query Q1 is as follows

```
SELECT  l_returnflag,l_linestatus, count(*) as count_order
FROM    lineitem
WHERE   l_returnflag='A' AND l_linestatus='F' AND
        l_shipdate<=date '1998-12-01' - interval '90' day;
```

In this experiment, which is conducted on the SUN workstation described in Section 6.2, we compare the setup and response times of CubiST⁺⁺ with our benchmark ROLAP server. In order to apply CubiST⁺⁺ we first generate the integer encodings as follows: For l_returnflag, we map ‘A’ to 0, ‘N’ to 1, ‘R’ to 2, for l_linestatus we map ‘F’ to 0, ‘O’ to 1, and for l_shipdate and l_commitdate we map the dates 1992-01-01 to 0, 1992-01-02 to 1, and so on. Using these encodings, the CQL expression corresponding to the above SQL query is $q = \text{count}(0:0;1:0;2:[0,2436])$.

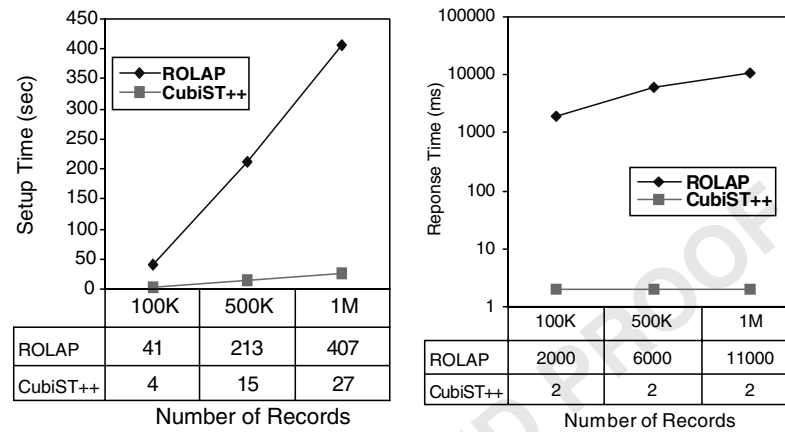


Figure 15. Setup and response times for CubiST⁺⁺ and commercial ROLAP server.

In the case of the ROLAP system, we use the SQL loader to load the table containing the datasets D3.x and evaluate the query using its SQL engine. Note that we spent considerable effort in fine-tuning the configuration of the ROLAP server and database including views and indexes in order to make the comparison as fair as possible. However, a description of the setup would exceed the scope of this paper. Instead, details can be obtained upon request by contacting the authors. Looking at figure 15, we can see that the query performance of CubiST⁺⁺ is faster than that exhibited by the ROLAP server by at least three orders of magnitude; the load time of CubiST⁺⁺ is at least ten times faster than that of the SQL loader.

CubiST⁺⁺ is also more space efficient since ROLAP systems store the original records (instead of just aggregates). Furthermore, materializing the group-by views and their indexes requires even more space.

6.6. CubiST⁺⁺ vs. commercial MOLAP server

In this series of experiments, we compare CubiST⁺⁺ against a MOLAP server using the same data sets (D3.x) and query as in Section 6.5. The experiments are run on the Windows NT PC. Figure 16 shows the superior setup and query performance of CubiST⁺⁺ over the MOLAP server. For the original 6,001,215-row data set, the setup time for the MOLAP server is 10,549 seconds while the setup time of CubiST⁺⁺ is 109 seconds. The response time of the MOLAP server for this data set is 437,550 ms while the response time of CubiST⁺⁺ for the same data set and query is still one millisecond.³ What causes the big difference is that the MOLAP server database increases from 6.6 MB to 317.6 MB when the number of records increases from 100,000 to 6,001,215. However, the size of the ST in CubiST⁺⁺ does not change. In the MOLAP server, when the database size is larger than the memory size, thrashing occurs. As a result, the majority of the processing time is spent on I/O.

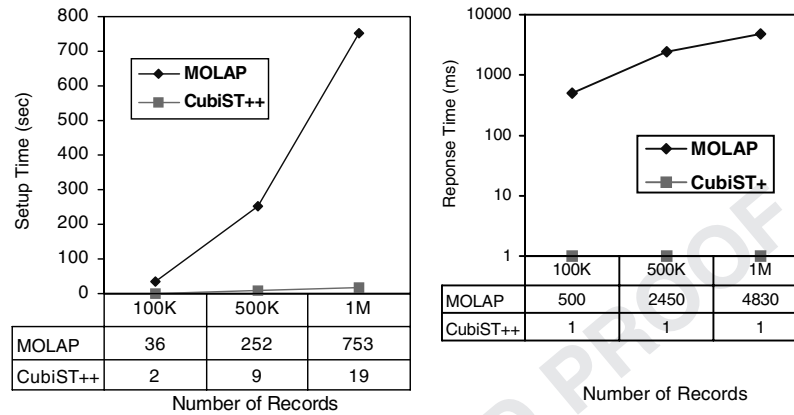


Figure 16. Setup and response times of CubiST⁺⁺ and the MOLAP server.

7. Conclusion

In this paper, we have presented the new Statistics Tree (ST) data structure and a new set of algorithms, called CubiST⁺⁺, for efficiently answering cube queries. Cube queries are a generalization of the cube operator and we have provided a formal representation called Cube Query Language (CQL) for this important class of queries. CQL is concise, powerful and easy to implement. Embedded in a host language, it is capable of representing very complex OLAP queries for which a translation into SQL for evaluation using a ROLAP system becomes impractical. Our one-pass initialization, query matching and evaluation algorithms ensure fast set-up and near immediate responses to complex cube queries. In CubiST⁺⁺, the records are aggregated into the leaves of the statistics tree without storing the details of 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. From the chosen family that can provide the answer, CubiST⁺⁺ selects the optimal matching tree to answer the query based on coarse granularity levels of the dimensions.

Cube queries compute aggregation information rather than retrieving the records that satisfy the query conditions. Queries such as “How many data points make up a certain region of the data cube?” are more relevant than queries such as “What are the specific individual records that lie in this region?” due to the often large number of records that make up a region. Moreover, for many types of trend analysis, being able to compute aggregate information quickly is more helpful than having access to the detailed data. This is a critical observation since CubiST⁺⁺ only stores the aggregates for the subcubes. Duplicate records are aggregated into the same leaves. In this way, many operations performed on records such as sorting, hashing, etc. are eliminated.

Other advantages of CubiST⁺⁺ are:

1. *Fast*: The initialization of the STs needs only *one* reading pass over the data set; all subsequent computations can be done on the STs without touching the original data.

2. *Space Efficient*: Keeping summary information in STs 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) and their hierarchical levels; 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 response times for cube queries over a data set containing 1 million or 1 billion records are almost the same.

Our simulations on various synthetic and TPC-H benchmark data sets clearly show that our new algorithms outperform prior algorithms in terms of setup and response times. We are currently enhancing the functionality of our prototype CubiST⁺⁺ system. Specifically, we are implementing a new ST Repository based on a database storage manager and are improving the functionality of the remaining components including loader, CQL compiler, and family generator. We will report on our findings in future workshops and conferences. Other future work will focus on potential applications of CubiST⁺⁺ to data mining problems.

Appendix—CQL syntax

```

CQL ::= <Aggregate><Measure> ( <Constraints> )
<Aggregate> ::= COUNT | SUM | MIN | MAX | AVG
<Measure> ::= null | <IntId>
<IntId> ::= integer | identifier
<Constraints> ::= <Constraints> ; <Constraint> | <Constraint>
<Constraint> ::= null | <Dimension> : <SelectedValues> |
<SelectedValues>
<Dimension> ::= ( <Dim>, <Level> ) | <Dim>
<Dim> ::= <IntId>
<Level> ::= <IntId>
<SelectedValues> ::= null | <IntId> | <Range> | {<Set>}

<Range> ::= [ <IntId>, <IntId> ]
<Set> ::= <Set> , <Term> | <Term>
<Term> ::= <IntId> | <Range>

```

Acknowledgments

The authors thank Profs. Stanley Su, Abdelsalam (Sumi) Helal, and Daniel Conway for the many fruitful discussions on the ST data structure and related algorithms, and for their insightful comments, which helped us improve the presentation of our results. We also thank

the members of the University of Florida Database Center for participating in our design meetings and for suggesting many useful improvements in both design and implementation of our prototype. Finally, we thank the anonymous reviewers for correcting any remaining mistakes and for their valuable input and suggestions.

Notes

1. By storing integers instead of strings like “Acura” or “Daimler-Chrysler,” we need to set aside only one byte if there are no more than 256 different data manufacturers in the domain.
2. Computing these STs is similar to slicing and dicing the data cube and rolling up its values along the different dimension hierarchies.
3. Due to the significant difference in these two running times, their data values are not depicted in figure 5.

References

1. S. Agarwal, R. Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and S. Sarawagi, “On the computation of multidimensional aggregates,” in 22th International Conference on Very Large Data Bases, Mumbai (Bombay), India, 1996.
2. R. Agrawal, A. Gupta, and S. Sarawagi, “Modeling multidimensional databases,” in Thirteenth International Conference on Database Engineering, Birmingham, UK, 1997.
3. Arbor Systems, “Large-scale data warehousing using hyperion esbase OLAP technology,” Arbor Systems, White Paper 1997.
4. K. Beyer and R. Ramakrishnan, “Bottom-up computation of sparse and iceberg CUBEs,” in ACM SIGMOD International Conference on Management of Data, Philadelphia, PA, 1999.
5. S. Chaudhuri and U. Dayal, “Data warehousing and OLAP for decision support,” SIGMOD Record (ACM Special Interest Group on Management of Data), vol. 26, pp. 507–508, 1997.
6. S. Chaudhuri and U. Dayal, “An overview of data warehousing and OLAP technology,” SIGMOD Record, vol. 26, pp. 65–74, 1997.
7. C.Y. Chan and Y.E. Ioannidis, “Bitmap index design and evaluation,” in 1998 ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, 1998.
8. E.F. Codd, S.B. Codd, and C.T. Salley, “Beyond decision support,” in Computer World, vol. 27, 1993, www.arborsoft.com/OLAP.html.
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 1993.
10. D. Comer, “The ubiquitous Btree,” ACM Computing Surveys, vol. 11, pp. 121–137, 1979.
11. C.E. Dyreson, “Information retrieval from an incomplete data cube,” in Twenty-Second International Conference on Very Large Data Bases, Mumbai (Bombay), India, 1996.
12. L. Fu and J. Hammer, “CubiST: A new algorithm for improving the performance of ad-hoc OLAP queries,” ACM Third International Workshop on Data Warehousing and OLAP (DOLAP), Washington, DC, 2000.
13. S. Goil and A. Choudhary, “High performance OLAP and data mining on parallel computers,” Journal of Data Mining and Knowledge Discovery, vol. 1, pp. 391–417, 1997.
14. S. Goil and A. Choudhary, “PARSIMONY: An infrastructure for parallel multidimensional analysis and data mining,” Journal of Parallel and Distributed Computing, vol. 61, pp. 285–321, 2001.
15. 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, vol. 1, pp. 29–53, 1997.
16. H. Gupta and I. Mumick, “Selection of views to materialize under a maintenance cost constraint,” Stanford University, Technical Report, 1997.
17. V. Harinarayan, A. Rajaraman, and J.D. Ullman, “Implementing data cubes efficiently,” SIGMOD Record (ACM Special Interest Group on Management of Data), vol. 25, pp. 205–216, 1996.

18. Information Advantage, "Business intelligence," White Paper, 1998.
19. Informix Corp., "Informix red brick decision server," 2001, <http://www.informix.com/redbrick/>.
20. T. Johnson and D. Shasha, "Some approaches to index design for cube forests," *Bulletin of the Technical Committee on Data Engineering*, IEEE Computer Society, vol. 20, pp. 27–35, 1997.
21. W. Labio, D. Quass, and B. Adelberg, "Physical database design for data warehouses," in *International Conference on Database Engineering*, Birmingham, England, 1997.
22. M. Lee and J. Hammer, "Speeding up warehouse physical design using a randomized algorithm," *International Journal of Cooperative Information Systems (IJCIS)—Special Issue on Design and Management of Data Warehouses*, vol. 10, pp. 327–354, 2001.
23. D. Lomet, "Bulletin of the technical committee on data engineering," in *Special Issue on Materialized Views and Data Warehousing*, J. Widom (Ed.), IEEE Computer Society, 1995, vol. 18.
24. Microsoft Corp., "Microsoft SQL server," Microsoft, Seattle, WA, White Paper.
25. MicroStrategy Inc., "The case for relational OLAP," MicroStrategy, White Paper.
26. P. O'Neil and D. Quass, "Improved query performance with variant indexes," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 26, pp. 38–49, 1997.
27. P.E. O'Neil, "Model 204 architecture and performance," *2nd International Workshop on High Performance Transaction Systems*, Asilomar, CA, 1987.
28. Oracle Corp., "Oracle express OLAP technology," <http://www.oracle.com/olap/index.html>.
29. Oracle Corp., "Oracle express server documentation," Oracle Corporation, Redwood Shores, CA, Documentation, August 2001.
30. Pilot Software Inc., "An introduction to OLAP multidimensional terminology and technology," Pilot Software, Cambridge, MA, White Paper.
31. Redbrick Systems, "Decision-makers, business data and RSQL," Informix, Los Gatos, CA, White Paper, 1997.
32. N. Roussopoulos, Y. Kotidis, and M. Roussopoulos, "Cubetree: Organization of and bulk updates on the data cube," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 26, pp. 89–99, 1997.
33. N. Roussopoulos and D. Leifker, "Direct spatial search on pictorial databases using packed R-trees," in *1985 ACM SIGMOD International Conference on Management of Data*, Austin, Texas, 1985.
34. R. Srikant and R. Agrawal, "Mining quantitative association rules in large relational tables," in *1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Quebec, Canada, 1996.
35. 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*, vol. 1, pp. 414–423, 1989.
36. Transaction Processing Performance Council, "The TPC BenchmarkTMH," Transaction Processing Council, 2001, <http://www.tpc.org/tpch/>.
37. Transaction Processing Performance Council, "Transaction processing performance council," 2001, <http://www.tpc.org/>.
38. W.P. Yan and P. Larson, "Eager aggregation and lazy aggregation," in *21th International Conference on Very Large Data Bases*, Zurich, Switzerland, 1995.
39. 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)*, vol. 26, pp. 159–170, 1997.
40. Y. Zhuge, H. Garcia-Molina, and J.L. Wiener, "Consistency algorithms for multi-source warehouse view maintenance," *Distributed and Parallel Databases*, vol. 6, pp. 7–40, 1998.