# ROLAP Implementations of the Data Cube

KONSTANTINOS MORFONIOS

*University of Athens*

STRATIS KONAKAS

*University of Athens*

YANNIS IOANNIDIS

*University of Athens*

and

NIKOLAOS KOTSIS

*Capgemini UK*



Implementation of the data cube is an important and scientifically interesting issue in On-Line Analytical Processing (OLAP) and has been the subject of a plethora of related publications. Naive implementation methods that compute each node separately and store the result are impractical, since they have exponential time and space complexity with respect to the cube dimensionality. To overcome this drawback, a wide range of methods that provide efficient cube implementation (with respect to both computation and storage) have been proposed, which make use of relational, multidimensional, or graph-based data structures. Furthermore, there are several other methods that compute and store approximate descriptions of data cubes, sacrificing accuracy for condensation. In this article, we focus on Relational-OLAP (ROLAP), following the majority of the efforts so far. We review existing ROLAP methods that implement the data cube and identify six orthogonal parameters/dimensions that characterize them. We place the existing techniques at the appropriate points within the problem space defined by these parameters and identify several clusters that the techniques form with various interesting properties. A careful study of these properties leads to the identification of particularly effective values for the space parameters and indicates the potential for devising new algorithms with better overall performance.

Categories and Subject Descriptors: H.4.2 [**Information Systems Applications**]: Types of Systems; H.2.8 [**Database Management**]: Database Applications

General Terms: Algorithms

Additional Key Words and Phrases: Data cube, data warehouse, OLAP

This project is co-financed within Op. Education by the ESF (European Social Fund) and National Resources.

Author's email: kmorfo@di.uoa.gr.

**ACM Reference Format:**
Morfonios, K., Konakas, S., Ioannidis, Y., and Kotsis, N. 2007. ROLAP implementations of the data cube. *ACM Comput. Surv. 39*, 4, Article 12 (October 2007), 53 pages DOI = 10.1145/1287620.1287623. http://doi.acm.org/10.1145/1287620.1287623.

## 1. INTRODUCTION

Implementation of the data cube is one of the most important, albeit "expensive," processes in On-Line Analytical Processing (OLAP). It involves the computation and storage of the results of aggregate queries grouping on all possible dimension-attribute combinations over a fact table in a data warehouse. Such precomputation and materialization of (parts of) the cube is critical for improving the response time of OLAP queries and of operators such as roll-up, drill-down, slice-and-dice, and pivot, which use aggregation extensively [Chaudhuri and Dayal 1997; Gray et al. 1996]. Materializing the entire cube is ideal for fast access to aggregated data but may pose considerable costs both in computation time and in storage space. To balance the tradeoff between query response times and cube resource requirements, several efficient methods have been proposed, whose study is the main purpose of this article.

As a running example, consider a fact table R consisting of three dimensions (A, B, C), and one measure M (Figure 1a). Figure 1b presents the corresponding cube. Each view that belongs to the data cube (also called *cube node* hereafter) materializes a specific group-by query as illustrated in Figure 1b. Clearly, if D is the number of dimensions of a fact table, the number of all possible group-by queries is $2^D$, which implies that the data cube size is exponentially larger with respect to D than the size of the original data (in the worst case). In typical applications, this is in the order of gigabytes, so development of efficient data-cube implementation algorithms is extremely critical.

The data-cube implementation algorithms that have been proposed in the literature can be partitioned into four main categories, depending on the format they use in order to compute and store a data cube: Relational-OLAP (ROLAP) methods use traditional materialized views; Multidimensional-OLAP (MOLAP) methods use multidimensional arrays; Graph-Based methods take advantage of specialized graphs that usually take the form of tree-like data structures; finally, approximation methods exploit various in-memory representations (like histograms), borrowed mainly from statistics. Our focus in this article is on algorithms for ROLAP environments, due to several reasons: (a) Most existing publications share this focus; (b) ROLAP methods can be easily incorporated into existing relational servers, turning them into powerful OLAP tools with little effort; by contrast, MOLAP and Graph-Based methods construct and store specialized data structures, making them incompatible, in any direct sense, with conventional database engines; (c) ROLAP methods generate and store precise results, which are much easier to manage at run time compared to approximations.

Our main contributions can be summarized as follows:

—**Comprehensive Review:** We present a comprehensive review of existing cubing methods that belong to the ROLAP framework, highlighting interesting concepts from various publications and studying their advantages and disadvantages. In this sense, we target both OLAP experts who can use this article as a quick reference, and researchers who are new in this area and need an informative tutorial. To the best of our knowledge, this is the first overview of its kind.

—**Problem Parameterization:** We carefully analyze existing ROLAP cubing methods and identify six orthogonal parameters/dimensions that affect their performance. The resulting 6-dimensional parameter space essentially models the data cube implementation problem.

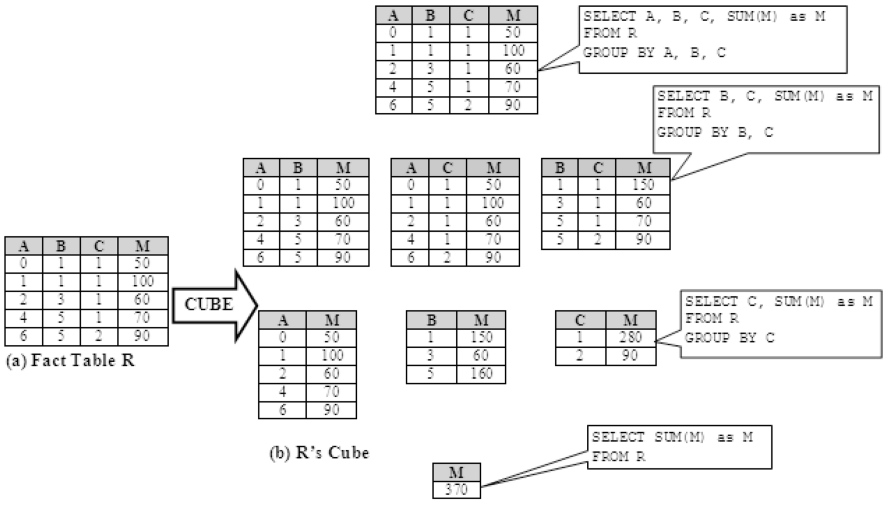


**Fig. 1**. Fact table R and its data cube.

—**Method Classification:** We place existing techniques at the appropriate points within this parameter space and identify several clusters that these form. These clusters have various interesting properties, whose study leads to the identification of particularly effective values for the space parameters and indicates the potential for devising new, better-performing algorithms overall.

The rest of this article is organized as follows: In Section 2, we present a brief overview of non-ROLAP data cube implementation methods, which are outside our focus. In Section 3, we define the data cube implementation problem and, in Sections 4, 5, and 6, we review existing algorithms for efficient data cube implementation in a ROLAP environment. In Section 7, we analyze the algorithms' comparisons that have been published so far and, in Section 8, we define a 6-dimensional parameter space that models the cube implementation problem, and place existing techniques at the appropriate points within it. Finally, we conclude in Section 9.

## 2. RELATED WORK

Implementation of the data cube consists of two subproblems: one concerning the actual computation of the cube and one concerning the particulars of storing parts of the results of that computation. The set of algorithms that are applicable to each subproblem is intimately dependent on the particular approach that has been chosen: ROLAP, MOLAP, Graph-Based, or Approximate. Specifically for ROLAP, which is the focus of this article, the two subproblems take on the following specialized forms:

—*Data cube computation* is the problem of scanning the original data, applying the required aggregate function to all groupings, and generating relational views with the corresponding cube contents.

—*Data cube selection* is the problem of determining the subset of the data cube views that will actually be stored. Selection methods avoid storing some data cube pieces according to certain criteria, so that what is finally materialized balances the tradeoff between query response time and cube resource requirements.

Both of these problems have been studied in the past only in a fragmented fashion. Our work fills this gap and presents the first systematic analysis of all relevant solutions. As a matter of completeness, in the rest of this section, we briefly highlight earlier work on MOLAP, Graph-Based, and Approximate methods, which are outside the scope of this article and are not analyzed any further.

MOLAP stores a data cube as a multidimensional array. In this approach, there is no need for storing dimension values in each separate cell, since they are determined by the position of the cell itself. Its main disadvantage comes from the fact that, in practice, cubes are sparse, with a large number of empty cells, making MOLAP techniques inefficient in storage space. Sparsity is the reason why the so-called chunk-based methods have been proposed [Zhao et al. 1997]. Such methods discard the majority of empty cells, storing only nonempty subarrays called chunks. ArrayCube [Zhao et al. 1997] is the most widely accepted algorithm in this category. It has also served as an inspiration to algorithm MM-Cubing [Shao et al. 2004], which applies similar techniques to just the dense areas of the cube, taking into account the distribution of data in a way that avoids chunking. Finally, Cube-File [Karayannidis et al. 2004] is a multidimensional file structure that can be used in conjunction with any MOLAP computation algorithm and achieves hierarchical clustering of the fact table itself for better query performance.

Graph-Based methods use some kind of graph or sometimes tree-like data structures for fast cube computation, storage-space reduction, high OLAP-query performance, and fast updates. This category includes Cubetrees [Kotidis and Roussopoulos 1998; Roussopoulos et al. 1997], which are based on R-trees; Cube Forests [Johnson and Shasha 1997], which can be implemented using B-trees; Statistic Trees [Fu and Hammer 2000], which can be used for answering cube queries efficiently; Dwarf [Sismanis et al. 2002; Sismanis and Roussopoulos 2004], which is a highly compressed data structure that eliminates prefix and suffix redundancies; and QC-Trees [Lakshmanan et al. 2003], which is a summary structure that also preserves semantic roll-up/drill-down links inherent in the data cube.

Similarly, algorithms H-Cubing [Han et al. 2001], Star-Cubing [Xin et al. 2003], and Range-Cube [Feng et al. 2004] exploit sophisticated data structures for accelerating the data cube computation. They continue to store the cube in relational views but construct it with the help of specialized in-memory data structures that represent the fact table compactly and are traversed in efficient, nonrelational ways. Hence, we consider them as not purely ROLAP and study them no further.

Finally, approximate methods are based on the assumption that decision support does not need very detailed or absolutely accurate results and store an approximate description of the data cube, sacrificing accuracy for storage-space reduction. These methods use various techniques, such as wavelet transformations [Vitter and Wang 1999; Vitter et al. 1998], multivariate polynomials [Barbará and Sullivan 1997, 1998], sampling [Gibbons and Matias 1998; Acharya et al. 2000], data probability density distributions [Shanmugasundaram et al. 1999], histograms [Poosala and Ganti 1999], and others [Gunopulos et al. 2000].

## 3. PROBLEM DESCRIPTION

Let "grouping attributes" be the fields of a table that participate in the group-by part of a query. All group-bys computed for the data cube construction can be partially ordered in a lattice structure [Harinarayan et al. 1996]. This is a directed acyclic graph (DAG), where each node represents a group-by query $q$ on the fact table and is connected via a directed edge with every other node whose corresponding group-by part is missing one of the grouping attributes of $q$. The source of an edge is called its *parent*

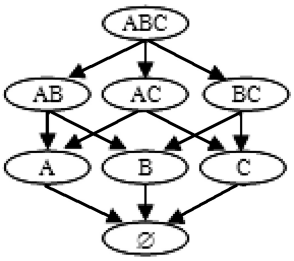


**Fig. 2**. Example of a cube lattice.

node and the destination of the edge is called its *child* node. The node whose grouping attributes consist of all dimensions (highest) is called the *root* of the lattice. The node whose grouping attribute set is empty (lowest) is called the *ALL node*. For example, Figure 2 presents the lattice that corresponds to the fact table R (Figure 1a). Nodes in higher lattice levels contain more grouping attributes and are thus more *detailed* (hold data of finer granularity) than nodes in lower levels, which are more *specialized* (hold data of coarser granularity). Exploiting the "1-1" mapping between group-by queries and lattice nodes, we call *node size* the size of the result of the corresponding query.

In most cases, computation of the data cube is an extremely time consuming process due to the size of the original fact table and the exponential number of cube nodes with respect to the number of dimensions. Hence, it is common in practice to select a proper subset of the data cube, precompute it off line, and store it for further use. We call this process "data cube implementation". Since 1996 there has been intense interest in efficient data cube implementation and a variety of methods has been proposed for this task.

As mentioned earlier, for ROLAP approaches, data cube implementation consists of data cube computation and data cube selection. These, however, are not necessarily two separate procedures applied sequentially. A priori knowledge of the strategy used for selection may affect the computation procedure and vice versa. Hence, several methods have been proposed that combine fast computation and efficient selection in one step; we call them *integrated methods*.

In the rest of this section, we elaborate on the nature of computation, selection, and integrated methods. Furthermore, we introduce the notion of redundancy in cube data, which plays a significant role in a large number of efforts related to data cubes. In the subsequent three sections, we focus on computation, selection, and integrated methods, respectively, and review the most influential data cube implementation algorithms that have been proposed in the literature so far and belong to the ROLAP framework.

### 3.1. Computation Methods

As mentioned in Section 2, computation involves scanning the original data, applying the required aggregate function on all groupings, and generating the cube contents. The main objective of computation algorithms is to place tuples that aggregate together (i.e., tuples with the same values in the grouping attributes) in adjacent in-memory positions, so that all group-bys can be computed with as few data scans as possible. The two main alternative approaches that can be used to achieve such a tuple placement are sorting and hashing. Furthermore, as the lattice structure indicates (Figure 2), there is much commonality between a parent node and its children. Taking advantage of this

commonality may lead to particularly efficient algorithms. For example, if the initial data in R (Figure 1a) is sorted according to attributes ABC, then it is also sorted according to both AB and A. Thus, sorting cost can be shared and nodes ABC→AB→A→Ø can be computed in a pipelined fashion. Five techniques that exploit such node relationships have been proposed in the literature [Agarwal et al. 1996] in order to increase the efficiency of computation algorithms: smallest-parent, cache-results, amortize-scans, share-shorts, and share-partitions.

Interestingly, sorting and hashing operations used for aggregation can be performed more efficiently when the data processed fits in memory; otherwise, external memory algorithms must be used, which incur greater I/O costs (in general increased by a factor of 2 or 3). To overcome this drawback, most computation methods partition data into disjoint fragments that do fit in memory, called *partitions*. Tuples are not placed into partitions randomly, but according to the policy that tuples that aggregate together must belong to the same partition. Hence, tuples from different partitions do not merge for aggregation and a partition becomes the unit of computation, since each partition can be processed independently of the others.

### 3.2. Selection Methods

Recall that selection methods determine the subset of the data cube that will actually be stored. To perform selection, such methods use several optimality criteria under a variety of constraints. Their objective is to balance the tradeoff between the amount of resources consumed by the data cube (such resources usually include storage space and time for incremental maintenance) and query response time. The problem of selecting the optimum subset of a cube is NP-complete, even under some rather loose constraints [Harinarayan et al. 1996]. Hence, all methods search for near-optimum solutions using appropriate heuristics. As a side effect, effective selection methods in general accelerate computation as well, since partial cube storage implies smaller cost of writing the result to the disk and also probably fewer aggregation operations.

### 3.3. Integrated Methods

From the previous description, it is evident that data cube computation algorithms address the question of "**how**" the cube should be computed, whereas selection algorithms address the question of "**what**" part of the cube should be stored. The current trend is to address both problems together through integrated algorithms that compute the data cube, condensing the result of the computation at the same time. Hence, fast computation is combined with efficient storage, leading to better results.

### 3.4. Cube Redundancy

First Kotsis and McGregor [2000] and then several other researchers realized that a great portion of the data in a cube is redundant [Feng et al. 2004; Lakshmanan et al. 2002, 2003; Morfonios and Ioannidis 2006a, 2006b; Sismanis et al. 2002; Wang et al. 2002]. Removing redundant data results in a condensed yet fully materialized cube. Existing methods that avoid redundancy achieve impressive reductions in the size of the cube stored, which benefit computation performance as well, because output costs are significantly lower and, in some cases, because early identification of redundant tuples enables pruning of parts of the computation. Additionally, at query time, neither aggregation nor decompression takes place, but only a simple redirection of retrieval to other nodes in the lattice.

In this paper, we distinguish two notions of redundancy: (a) A tuple in a node is **partially redundant** when it can be produced by simple projection of a single tuple

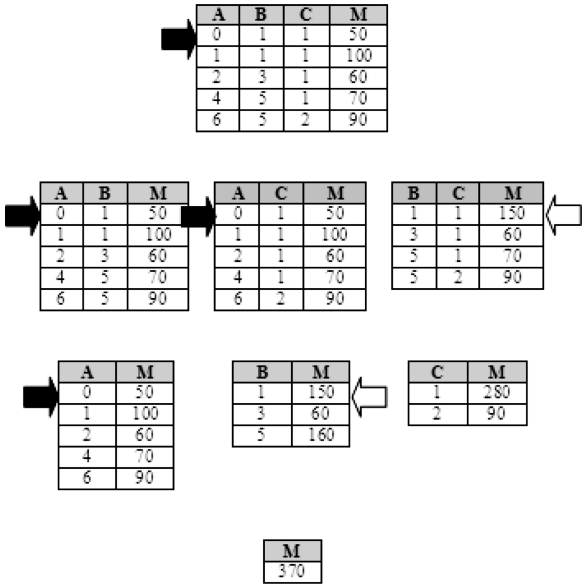


**Fig. 3**. Example of redundant tuples.

from its **parent node** in the cube lattice without aggregation. Partially redundant tuples are generated by the same set of tuples in the original fact table and have, therefore, the same aggregate values. Such redundant tuples are taken into account by several existing algorithms [Feng et al. 2004; Lakshmanan et al. 2002; Lakshmanan et al. 2003; Morfonios and Ioannidis 2006b; Sismanis et al. 2002; Kotsis 2000]. (b) A partially redundant tuple in a node is further characterized as **totally redundant** when it can be produced by projection of a single tuple from **any ancestor node** all the way up to the **root of the lattice** without aggregation. This concept has also been identified by earlier efforts as well [Morfonios and Ioannidis 2006a, 2006b; Wang et al. 2002; Kotsis and McGregor 2000]. Both notions of redundancy are better explained with the example below.

**Example:** Consider again the fact table R in Figure 1a. Figure 3 presents the corresponding cube (as in Figure 1b but note that this time some cube tuples are pointed by arrows). Tuple $\langle 0, 50\rangle$ in cube node A is totally redundant, since it can be produced by a simple projection of tuple $\langle 0, 1, 1, 50\rangle$ in R. In fact, the black arrows in Figure 3 point to several such tuples that can generate tuple $\langle 0, 50\rangle$; these are all the tuples in an entire subcube repeating the same information with respect to A. Essentially, $A = 0$ "functionally determines" the values in all other fields, thus the corresponding tuple is redundant in all nodes containing dimension A. Similarly, tuple $\langle 1, 150\rangle$ in B can be evaluated from a simple projection of tuple $\langle 1, 1, 150\rangle$ in its parent node BC (white arrows in Figure 3). However, $B = 1$ does not "functionally determine" the values in all the other fields, hence, tuple $\langle 1, 150\rangle$ is partially but not totally redundant.

Note that the notion of a partially-redundant tuple denotes a relationship between the tuple's node and its parent only. No conclusions can be drawn about other ancestor nodes in higher lattice levels. On the contrary, the definition of a totally redundant tuple implies by induction, that it is totally redundant in all of its ancestors up to the lattice root, and is also found in the original fact table.

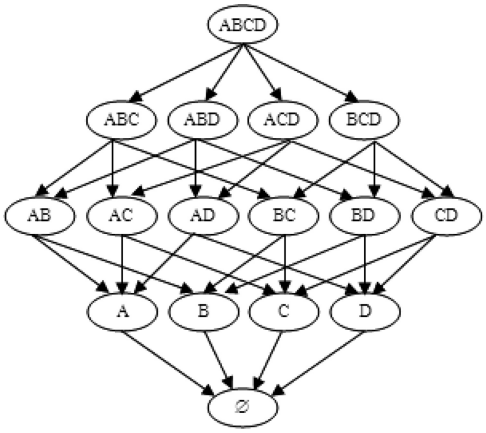


**Fig. 4**. The lattice of a 4-dimensional cube.

## 4. COMPUTATION METHODS

There are essentially seven computation algorithms that have been discussed in the past and that we present here: $2^D$, GBLP, PipeSort, Overlap, PipeHash, PartitionedCube, and BottomUpCube (BUC).

### 4.1. Algorithm $2^D$

Algorithm $2^D$ [Gray et al. 1996] is the simplest algorithm for data cube computation. It computes each group-by directly from the original fact table and then takes the union of all partial results for the data cube. Algorithm $2^D$ is not efficient, since it has exponential complexity with respect to the number of dimensions and takes no advantage of the commonalities among the interim results of different group-bys. Hence, any data structures, tuple sortings, or hash tables constructed for the computation of one group-by are never reused but are recreated from scratch whenever necessary. Such behavior renders it impractical. Nevertheless, note that Algorithm $2^D$ has been proposed as part of the introduction of the data cube structure. Hence, we present it here just for completeness.

To overcome the main problem of Algorithm $2^D$, all subsequent algorithms identify a spanning tree T of the cube lattice (T will be hereafter called *execution tree*) whose edges indicate which node (parent) will be used for the computation of each node (child). A key difference among these algorithms is the particular execution tree they use.

### 4.2. Algorithm GBLP

Algorithm GBLP [Gray et al. 1996] improves the efficiency of $2^D$ by computing each node in the lattice using its smallest parent and not the original fact table. The group-by operation may be either sort-based or hash-based, depending on the implementation.

Moving towards lower levels in the lattice, node sizes tend to decrease, since groups of tuples get aggregated and are replaced with summary (less detailed) tuples. Thus, selecting the smallest parent leads to faster computation of a node. For example, assume a 4-dimensional fact table whose lattice is shown in Figure 4. Let the result size that corresponds to each node of that lattice be as shown in Figure 5. Then, the execution plan of the GBLP algorithm is the tree shown in the figure, where each edge indicates

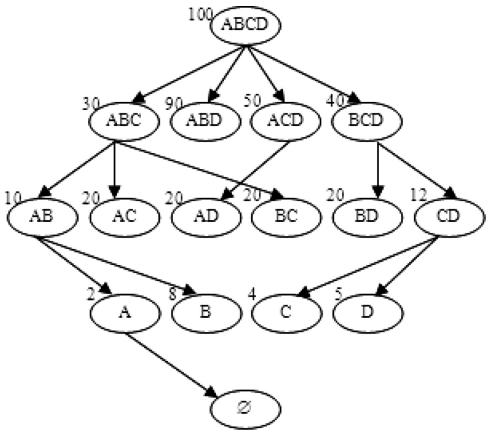


**Fig. 5**. Execution tree of the lattice according to the smallest-parent method.

the parent used for the computation of the child. In reality, GBLP (as well as other algorithms presented below) obtains estimates of the size for each lattice node based on a variety of statistical methods. Such methods have been widely studied in the literature [Haas et al. 1995; Ross and Srivastava 1997; Shukla et al. 1996], but their presentation exceeds the scope of this article.

The main disadvantage of the GBLP algorithm is that it only prunes the lattice into a tree, without suggesting a particular way to traverse it. Lack of an efficient disk-access plan makes GBLP impractical when dealing with large datasets that do not fit in main memory. Nevertheless, the corresponding paper [Gray et al. 1996] is seminal, as it introduced the concept of the data cube itself.

Furthermore, in the same paper [Gray et al. 1996], there is a very rough comparison between the sizes of a fact table and the corresponding data cube. The size C of the fact table (in terms of number of tuples) is approximated by $C = C_1 \times C_2 \times \cdots \times C_D$, where $C_i$ denotes the cardinality (domain size) of the i-th dimension ($i = 1 \ldots D$). The data cube adds only one additional value (ALL) to each dimension, so its size, again in terms of number of tuples, is approximated by $C_{Cube} = (C_1 + 1) \times (C_2 + 1) \times \cdots \times (C_D + 1)$, which implies that it is slightly larger than the size of the original fact table. Hence, the argument is made that whenever the original fact table fits in memory, it is highly likely that the data cube fits in main memory as well.

The strength of this argument is weakened [Harinarayan et al. 1996] by the fact that C and $C_{Cube}$, as defined in the original paper, are actually very rough estimations of the sizes of the fact table and the data cube, respectively; in most real-world applications, the corresponding formulas do not hold. In general, the original fact table is sparse: its actual size is a small fraction of the size of the Cartesian product of the domains of its attributes. Furthermore, the sparser a fact table is, the higher the quotient of the cube size over the fact-table size is. In other words, in real-world applications, the size of the data cube is expected to be much larger than the size of the fact table in terms of number of tuples. This situation makes the need for an efficient disk access plan even more critical.

To overcome these memory size issues, all the following algorithms partition their execution tree into an appropriate set of subtrees, each one of which has a small enough number of nodes so that they can all be constructed concurrently, reducing memory requirements overall. The problem of finding the optimum set of subtrees is NP-complete [Agarwal et al. 1996], hence, each approach uses some heuristics. In presenting these

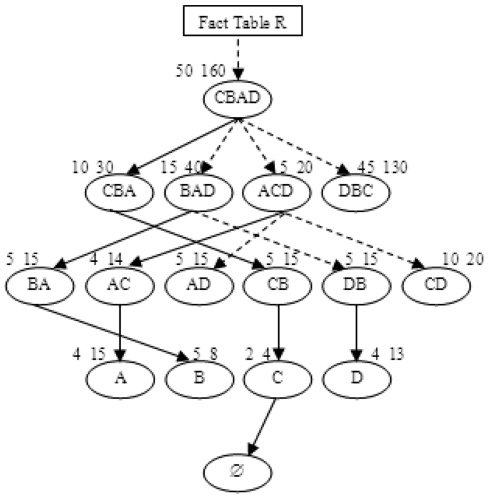


**Fig. 6**. The execution tree created by PipeSort.

approaches in the subsequent subsections, we use the lattice of the 4-dimensional cube shown in Figure 4 as an example.

### 4.3. Algorithm PipeSort

Algorithm PipeSort [Agarwal et al. 1996] was the first attempt to improve algorithm GBLP, by trying to minimize the total computation cost of a data cube. It uses cost estimations for the different means of computing each lattice node $N_y$ in order to decide which node $N_x$ among its parents will be finally used.

In the beginning, PipeSort labels each edge $N_{xy}$ ($N_x \rightarrow N_y$) of the lattice with two costs: $S_{xy}$ is the cost of computing $N_y$ using node $N_x$ if it is not appropriately sorted, and $A_{xy}$ is the cost of computing $N_y$ using node $N_x$ if it is appropriately sorted. Afterwards, PipeSort processes the lattice level-by-level starting from the root, determining the order of the attributes of each node in which it will be sorted during computation. If the sort order of node $N_y$ is a prefix of the sort order of its parent $N_x$, then the former can be computed from the latter without any extra sorting. In this case, edge $N_{xy}$ is marked A and incurs cost $A_{xy}$. Otherwise, $N_x$ has to be sorted first according to the sort order of $N_y$. In this case, edge $N_{xy}$ is marked S and incurs cost $S_{xy}$. Clearly, at most one edge that starts from $N_x$ can be marked A. PipeSort uses a local optimization technique based on weighted bipartite matching in order to identify the edges minimizing the sum of edge costs at each level of the lattice and to transform it into an efficient execution tree.

Assuming for the sake of simplicity that, for a given node $N_x$, costs $A_{xy}$ and $S_{xy}$ are constant (they depend only on the parent node $N_x$ regardless of its particular child node $N_y$), the tree that arises after pruning the lattice in Figure 4 is shown in Figure 6. Since we have removed the dependence on $N_y$, in this figure, we show the costs $A_x$ and $S_x$ ($A_x$ is always smaller) above the corresponding parent node $N_x$ and not next to every edge $N_{xy}$. Dashed arrows denote the edges marked with label S, for which a sorting operation must be applied before aggregation. Note that, in this particular example, the choice of which edge to mark with label A in the first level (between the root and its children) is trivial due to the fact that all edges are associated with the same costs (50 and 160). The algorithm breaks the tie by choosing the first node CBA. In the following steps, the costs differ and the choice is based on weighted bipartite matching.

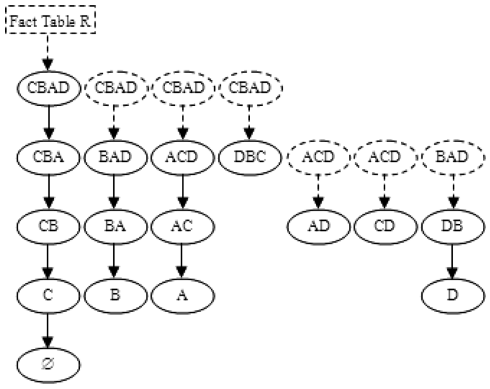


**Fig. 7**. The pipelined paths that are evaluated.

After creating an execution tree, PipeSort adds a node that corresponds to the original fact table R and an edge marked S that connects R to the root of the tree. Then, it transforms the tree into a set of paths (Figure 7), so that each edge appears in exactly one path and all edges apart from the first in each path are marked A. In other words, only the first node of each path needs to be sorted. During cube computation, each path is processed separately. For a path rooted at $N_x$ (or R), the tuples of $N_x$ (or R) are sorted with respect to the attribute sort order of its child; then all nodes can be computed in a pipelined fashion. The pipeline needs only one tuple buffer in main memory for each node of the path except the root, which makes the memory usage of the algorithm particularly efficient.

The main disadvantage of PipeSort is that it does not scale well as the number of dimensions increases. This disadvantage is attributed to the fact that PipeSort performs a sorting operation for the pipelined computation of each path. If D denotes the number of dimensions, a lower bound on the number of sortings that PipeSort performs is given by the expression $\binom{D}{\lceil D/2 \rceil}$, which is exponential in D. The proof is based on the fact that no pair of nodes whose sets of grouping attributes is each of size $\lceil D/2 \rceil$ can be on the same path. In the example of Figure 7, seven sorting operations are performed, while the estimated lower bound is six. Sorting is in general an expensive operation, whose cost increases when the original fact table is sparse, because node sizes decrease slowly from level to level, since fewer aggregations occur. Hence, for a large original fact table, there are also several other nodes that do not fit in main memory, leading to a great number of external sorts.

### 4.4. Algorithm Overlap

Algorithm Overlap [Agarwal et al. 1996] has been developed in parallel with PipeSort, and its objective is minimizing disk accesses through overlapping of the computation of multiple data cube nodes. In order to reduce I/O costs, Overlap uses the notion of partially-matching sort orders and decreases the number of necessary sort operations. For instance, given the sort order ABC, the sort orders AB, AC, BC, A, B and C are partially matching with the initial one, since they contain a subset of its attributes in the same order. On the contrary, sort orders BA, CA and CB are not partially matching with ABC.

Initially, algorithm Overlap chooses a sort order that contains all cube attributes. This is used for sorting the root of the lattice and for ordering the grouping attributes of all its descendants (sorting of descendant nodes themselves occurs only in later

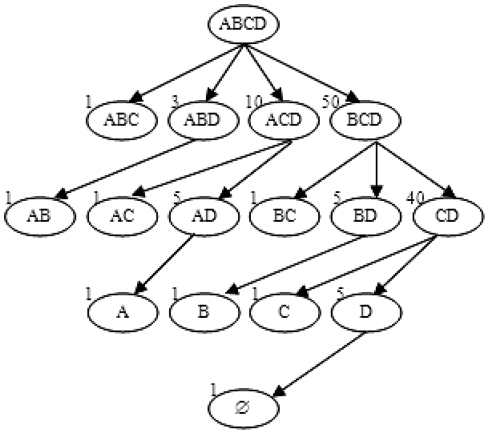


**Fig. 8**. The execution tree created by Overlap.

stages). Selection of this sort order can be based on heuristics [Agarwal et al. 1996] and, although it does not affect the way Overlap functions, it can play an important role in the efficiency of the algorithm.

The next step of Overlap prunes the lattice into a tree, using the following procedure: for every node N in the lattice, its parent with whom it shares the longest prefix in the initially chosen sort order becomes its parent in the execution tree. After creation of the tree, Overlap estimates the memory size necessary for computing each node from its parent, assuming uniform data distribution and that the parent's tuples are sorted according to the initially chosen sort order. Figure 8 shows the tree created by Overlap starting from the lattice in Figure 4. The initially chosen sort order in our example is ABCD. Next to each node appears the estimated memory size (in pages) that is necessary for its computation. Note that among the children of a particular node, memory size increases from left to right. Such increase is attributed to the fact that nodes to the left share a longer common prefix with their parent, hence resorting tuples that propagate from parent to child according to the new sort order requires less resources. For example, memory size for computing ABC from ABCD is 1, since tuples are propagated in the correct order and no resorting is necessary, whereas memory size for BCD is the maximum (equal to the estimated size of node BCD itself).

After creation of the tree, Overlap functions iteratively. In each step, it selects a set of nodes that have not yet been processed and can be computed together, taking into consideration memory constraints. The selected nodes, whose estimated memory requirements can be satisfied, are placed into the so-called "Partition" class. The rest are placed into the "SortRun" class and each of them is given one memory page. Selection of nodes is based on the following heuristic for tree traversal: Nodes are selected in a top-down/breadth-first fashion, giving priority to nodes with greater numbers of grouping attributes and smaller estimated memory requirements.

During data cube computation, each selected node set is evaluated separately. Nodes in the "Partition" class are computed in a pipelined fashion, since their memory requirements are fully satisfied and resorting tuples propagating from their parents can be performed in memory. Additionally, the tuples of nodes in the "SortRun" class are partially sorted, using the individual pages mentioned earlier. The resulting sorted runs are stored on the disk and are merged in subsequent iterations, actually pipelined with aggregation; hence, use of "SortRun" nodes saves on scans of expensive parent nodes, doing as much as possible with minimum memory requirements. Figure 9 shows the

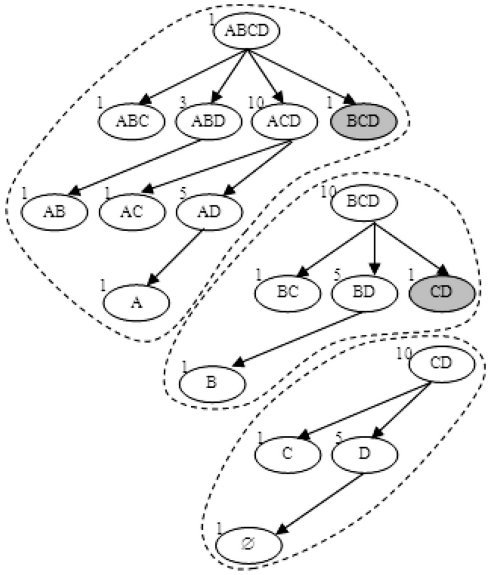


**Fig. 9**. Subtrees of in-memory computation.

three node sets that will be chosen by Overlap, assuming a memory size of 25 pages. Nodes in the "SortRun" class have been shaded. Note that a single page is allocated to nodes in the "SortRun" class and to nodes in the "Partitioned" class whose ordered attributes are a prefix of their parents' ordered attributes.

In conclusion, Overlap follows a strategy similar to PipeSort, using a sort-based algorithm for computing separate group-bys. Their main differences are in the way they prune the lattice, the manner in which they traverse the resulting tree, and the fact that Overlap uses the notion of partially matching sort orders to further optimize the sorting process.

### 4.5. Algorithm PipeHash

Algorithm PipeHash [Agarwal et al. 1996] has been proposed as an alternative to PipeSort and is based on hashing. Initially, PipeHash prunes the lattice into an execution tree like GBLP; among all parents of each node, it selects the one with the minimum estimated size. The resulting execution tree XT that corresponds to the lattice of Figure 4 is shown in Figure 10, assuming the sizes are those next to each node.

PipeHash uses a hash table for every simultaneously computed node in order to place tuples that aggregate together in adjacent in-memory positions. If only some of the hash tables fit in memory simultaneously, PipeHash partitions XT into subtrees of nodes whose hash tables do fit in memory all together. Hence, it works iteratively using a set W of all the subtrees that have not been processed yet, which is initialized with XT. In each iteration, a subtree $XT_i$ is removed from W. If memory requirements for the construction of $XT_i$ can be satisfied, then the corresponding nodes are computed and stored. Otherwise, PipeHash selects a subset S of the attributes of the root of $XT_i$, which defines a subtree $XT_s$ that has the same root as $XT_i$ and contains all nodes that contain S in their attributes. Selection of S is based on maximizing the size of $XT_s$, under the constraint that memory requirements for the construction of all of the nodes in $XT_s$ do not exceed main memory. The root of $XT_s$ is then partitioned on S and, for each

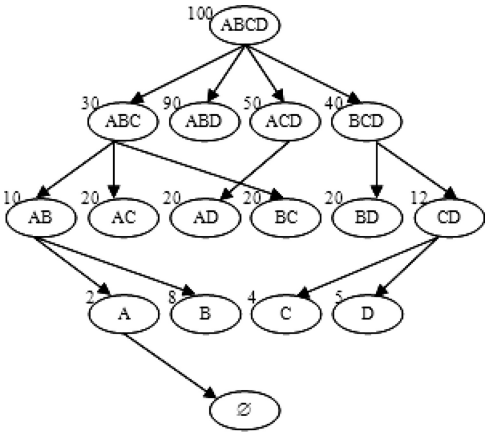


**Fig. 10**. The execution tree created by PipeHash.

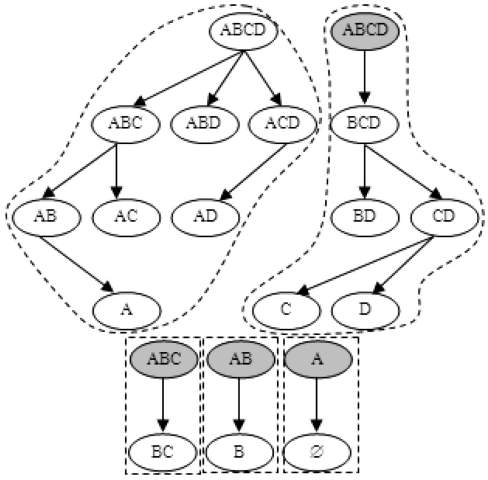


**Fig. 11**. The subtrees created by PipeHash.

partition, all nodes in $XT_s$ are computed and stored. The remaining subtrees, those in $XT_i$-$XT_s$, are inserted into W for processing in subsequent iterations.

Figure 11 presents the subtrees that are processed in each iteration of PipeHash. Assuming that XT cannot be computed in main memory, PipeHash selects $S = \{A\}$. The original fact table is partitioned on A and all nodes that contain A in their grouping attributes are computed. As illustrated in the figure, four subtrees remain, which are inserted into W. Assuming that each of them can be computed in main memory, no further partitioning is necessary.

### 4.6. Algorithm Partitioned-Cube

Data in real applications tends to be sparse due to the following two reasons [Ross and Srivastava 1997]:

—The number of dimensions of the original fact table is large.

—Some dimensions of the original fact table have large domains.

Sparsity results in fewer aggregation operations, since in sparse datasets, the number of tuples with the same dimension values decreases. Consequently, the relative sizes of intermediate nodes in the cube lattice tend to increase with sparsity. Large sizes of intermediate nodes introduce considerable I/O costs to the computation methods presented here, since these methods sort (or hash) and scan a potentially large number of such nodes during their execution (see for example Figure 7). Therefore, all previous methods are considered to be ineffective when applied over sparse datasets, especially when these datasets are much larger than the available memory size, since they do not scale well. The objective of Algorithm Partitioned-Cube [Ross and Srivastava 1997] is to combine the advantages of all previous algorithms with a more effective way of partitioning data, in order to reduce I/O and achieve faster computation even when the original fact table is sparse.

Partitioned-Cube is recursive and follows a divide-and-conquer strategy based on a fundamental idea that has been successfully used for performing complex operations (such as sorting and join): it partitions the large tables into fragments that fit in memory and performs the complex operation on each memory-sized fragment independently. Partitioning into smaller fragments starts from the original fact table and proceeds recursively until all partitions fit in main memory. For each memory-sized partition Algorithm Memory-Cube is called, which computes entire subcubes inside memory.

At each stage, Partitioned-Cube proceeds as follows:

—It partitions the input table R into n fragments based on the value of some attribute d and recursively calls itself n times passing as input of every recursive call one partition at a time. The union of the n results is the subcube $SC_1$ that consists of all nodes that contain d in their grouping attributes.

—It then makes one more recursive call to itself giving as input the more detailed node of $SC_1$ and fixing d to the value ALL. In this way, it computes the subcube $SC_2$ that consists of all nodes that do not contain d in their grouping attributes.

Note that half of the nodes of the cube of R belong to $SC_1$, while the remaining half belong to $SC_2$.

An interesting feature of Partitioned-Cube is that the computation of the data cube of R is broken up into $n + 1$ smaller subcube computations, n of which are likely to be much smaller than the original data cube, if the domain of the partitioning attribute d is sufficiently large. In the absence of significant skew, we expect roughly $|R|/n$ tuples in each of the n partitions. Thus, it is relatively likely that, even for a fact table significantly larger than main memory, each of these n subcubes can be computed in memory using Algorithm Memory-Cube. The $(n + l)$-th subcube has one fewer grouping attribute and at most $|R|$ tuples. Hence, the I/O cost of Partitioned-Cube is typically proportional to the number of dimensions D, and not exponential (as in PipeSort) or quadratic (as in Overlap). In the aforementioned cost estimations, we have omitted the factor $|R|$, because it is common in all of them.

Figure 12 presents the recursive calls of Partitioned-Cube on the lattice of Figure 4. As before, the dashed edges in Figure 12 denote a sorting operation. Assuming that the original fact table R does not fit in main memory, it is partitioned on attribute A. In our example, all n partitions fit in memory and the nodes that contain A in their grouping attributes are computed by Memory-Cube. The input table in the $(n + 1)$-th recursive call is node ABCD, which is the most detailed node of the previous stage. Assuming that ABCD does not fit in memory either, there is a new partitioning on attribute B (attribute A is projected out), which is the final one, since we assume that it generates partitions, for which Memory-Cube is enough.

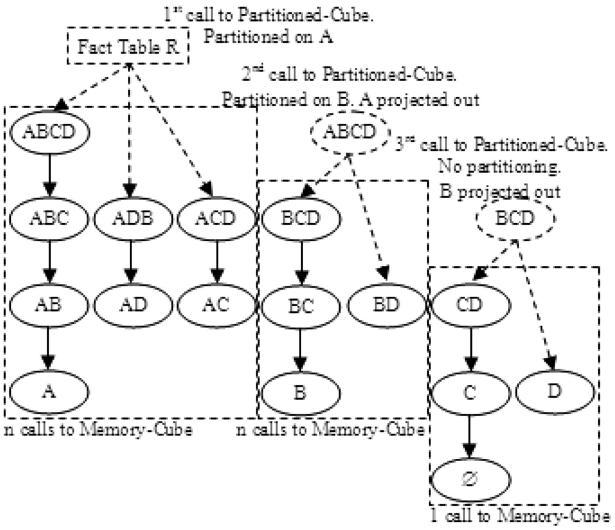


**Fig. 12**. Recursive calls of Partitioned-Cube.

Using a dynamic partitioning scheme, Partitioned-Cube is adaptable even when data is not uniformly distributed. Its adaptability is attributed to the fact that, if a single partition does not fit in main memory, then it is further treated recursively and independently from all other partitions, until it fits. This feature differentiates Partitioned-Cube from all previous algorithms, which do not scale well when the original fact table is sparse and data is skewed.

Finally, with respect to the in-memory algorithm Memory-Cube, it is a variation of PipeSort with the following fundamental differences:

—The number of sorting operations that Memory-Cube performs is equal to the number of paths that are formed traversing the lattice. Thus, minimizing the number of paths implies fewer time-consuming operations. Memory-Cube guarantees the creation of an access plan that contains the minimum number of paths, which is $\binom{D}{\lceil D/2 \rceil}$, where D is the number of dimensions. On the contrary, PipeSort does not provide similar guarantees and thus in general incurs greater sorting costs.

—Memory-Cube takes advantage of the dependencies between consecutive sorting operations, sharing a considerable amount of computational costs. PipeSort does not act like this, but performs each sorting operation from scratch.

### 4.7. Algorithm BUC

Algorithm BUC [Beyer and Ramakrishnan 1999] was primarily introduced for the computation of the Iceberg-Cube, which computes only those group-by tuples with an aggregate value (e.g. count) above some prespecified minimum support threshold (minsup). In other words, the algorithm takes into further consideration only sets of tuples that aggregate together, and for which the aggregate function returns a value greater than minsup. The Iceberg-Cube can be used for answering SQL queries that contain clauses like HAVING COUNT(*) $\geq$ X, for mining multidimensional association rules, and for extending selection methods (like PBS, which will be presented later).

The name of the algorithm (BottomUpCube) indicates the way it traverses the cube lattice (Figure 13). It builds the cube in a bottom-up fashion, starting with the

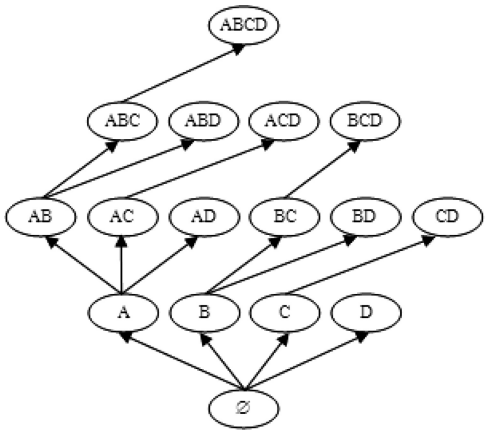


**Fig. 13**. The execution tree created by BUC.

computation of the ALL node, which contains only one tuple, moving towards more detailed nodes with more grouping attributes, and finishing at the root of the lattice, which contains all dimensions as its grouping attributes. This feature differentiates BUC from all previous methods, which compute the cube moving top-down, and gives it the advantage of early pruning of tuple sets that do not satisfy minimum support criteria, expressed through the parameter minsup. Such pruning is valid since the sets of tuples that do not satisfy minimum support at a node cannot satisfy it at any of its ancestors either, and can therefore be pruned. The parameter minsup is analogous to the minimum support that appears in the Apriori algorithm [Agrawal et al. 1996], which is used for the identification of association rules. Note that unlike all other algorithms, to have this pruning ability, BUC sacrifices any exploitation of parent/child commonalities in the cube lattice. Its pruning ability makes it particularly effective in the computation of the Iceberg-Cube. This is especially true when the original fact table is sparse since it generates fewer aggregations, making the satisfaction of minimum support more difficult, and thus leading to more pruning.

In more detail, BUC proceeds as follows: In the first step, it aggregates the entire fact table into a single value, essentially computing the ALL node. It then iterates over all dimensions and, for each dimension $d \in [1, D]$, it partitions the original fact table so that all tuples within a partition share the same value on d. Each partition that satisfies the minimum support condition is aggregated and becomes the input of a recursive call to BUC, where it is further partitioned into smaller pieces based on all the remaining dimensions. Figure 14 illustrates how the input is partitioned during the first four calls to BUC (assuming minsup = 1). First, BUC produces the ALL node. Next, it partitions the fact table R on dimension A, produces partitions $a_1$ to $a_3$, and then continues recursively on partition $a_1$. Partition $a_1$ is first aggregated to produce a single tuple for node A and then partitioned on dimension B. BUC continues recursively on the $\langle a_1, b_1 \rangle$ partition, which results in a $\langle a_1, b_1 \rangle$ tuple for node AB. Similarly for $\langle a_1, b_1, c_1 \rangle$, and $\langle a_1, b_1, c_1, d_1 \rangle$. At this point BUC returns only to continue recursively again on the $\langle a_1, b_1, c_1, d_2 \rangle$ partition. It then returns twice and continues with the $\langle a_1, b_1, c_2 \rangle$ partition. When this is complete, it partitions the $\langle a_1, b_1 \rangle$ partition on D to produce the $\langle a_1, b_1, D \rangle$ aggregates. Once the $\langle a_1, b_1 \rangle$ partition is processed, BUC proceeds with $\langle a_1, b_2 \rangle$, and so on.

Based on the above, it is clear that BUC traverses the lattice in a bottom-up/depth-first fashion (Figure 13).

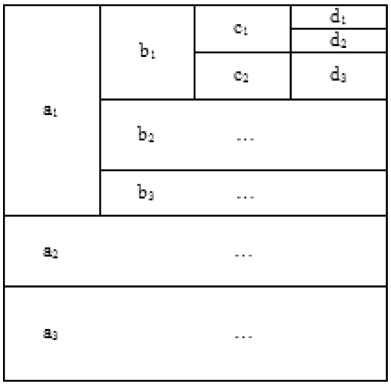


**Fig. 14**. BUC partitioning.

## 5. SELECTION METHODS

As mentioned earlier, besides computation, which has been studied in the previous section, the second essential aspect of data-cube implementation is selection. We next present the most influential selection algorithms that have been discussed in the past, namely Greedy, Benefit-Per-Unit-Space (BPUS), Greedy-Interchange, Inner-Level Greedy, r-Greedy, MDred-lattice, Pick-By-Size (PBS), Inverted-Tree Greedy, Polynomial-Greedy-Algorithm (PGA), Key, DynaMat, and some randomized algorithms. Note that the general view selection problem in a data warehouse is broader than the problem we study in this article. The objective of the algorithms that we present here is to select a subset of the views of a data cube for computation and storage, not to select general views that may include operations like selections, projections, and joins. Clearly, the study of algorithms of the latter category [Theodoratos et al. 2001; Theodoratos and Sellis 1997] is beyond the scope of this article.

### 5.1. Algorithm Greedy

Algorithm Greedy [Harinarayan et al. 1996] is the first systematic attempt to find a solution to the problem of selecting an appropriate subset of the cube for computation and storage. Its name indicates that it is based on a "greedy" heuristic in order to avoid an exhaustive search in the space of all possible solutions.

Greedy provides a solution to the selection problem with the goal of minimizing the average time taken to evaluate a lattice node on the fly, under the constraint that the number of nodes that are maintained in materialized form is bounded. Note that the space used by materialized nodes plays no role in this approach. Even in this simple setting, the aforementioned optimization problem is NP-complete [Harinarayan et al. 1996] (there is a straightforward reduction from Set-Cover).

The algorithm assumes that the storage cost (size) of each node is known a-priori. Let C(u) be the size of node u, measured in terms of the number of tuples. Suppose also that there is a limit k in the number of nodes that can be stored along with the root of the lattice. The root is always materialized, since it contains the most detailed information and cannot be computed from its descendants. Let S be the set of nodes that the algorithm selects. Set S is initialized to the root of the lattice. The algorithm is iterative and terminates after k iterations. In each iteration, one more node is added to S, building the final result in steps. The choice of a node u in the i-th iteration depends on the quantity B(u, S), which is called the *benefit* of u relative to S and is calculated

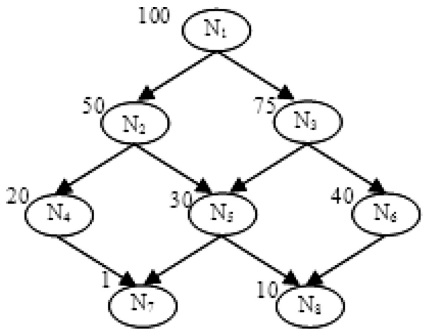


**Fig. 15**. Example of a lattice with known costs.

| | 1st Step | 2nd Step | 3rd Step |
|---|---|---|---|
| $N_2$ | **50 × 5 = 250** | | |
| $N_3$ | 25 × 5 = 125 | 25 × 2 = 50 | 25 × 1 = 25 |
| $N_4$ | 80 × 2 = 160 | 30 × 2 = 60 | **30 × 2 = 60** |
| $N_5$ | 70 × 3 = 210 | 20 × 3 = 60 | 20 + 20 + 10 = 50 |
| $N_6$ | 60 × 2 = 120 | **60 + 10 = 70** | |
| $N_7$ | 99 × 1 = 99 | 49 × 1 = 49 | 49 × 1 = 49 |
| $N_8$ | 90 × 1 = 90 | 40 × 1 = 40 | 30 × 1 = 30 |

**Fig. 16**. The three steps of the algorithm.

as follows:

(1) For each node w $\sqsubseteq$ u, where w $\sqsubseteq$ u denotes that w is either u or one of its descendants, let v be the node of least cost in S such that w $\sqsubseteq$ v. (Note that such a node always exists and in the worst case it coincides with the root.) Then, define the quantity $B_w$ as $B_w = \max\{C(v) - C(u), 0\}$.

(2) B(u, S) is defined as $B(u, S) = \sum_{w \sqsubseteq u} B_w$.

In other words, the benefit of u relative to S depends on the improvement that its materialization (addition to S) offers in the cost of computing itself and all of its descendants. For each w$\sqsubseteq$u, the cost of computing w using u is compared with the cheapest cost of computing w using some node v that already belongs to S. If u helps, which means that C(u) < C(v), then the difference C(v) - C(u) contributes to the total benefit, which is the sum of all such differences. In the i-th iteration, the Greedy algorithm inserts to S the node u that does not belong to it and has the maximum benefit B(u, S).

For example, suppose that we want to materialize a subset of the cube whose lattice is shown in Figure 15. Note the estimated storage cost shown next to each node. The constraint is that we can store only the root $N_1$ and k = 3 more nodes. Greedy will terminate after k = 3 iterations, where in each iteration it calculates the benefit of the remaining nodes and selects for materialization the one with the maximum benefit. Figure 16 presents all steps in detail, highlighting with bold font the maximum benefit achieved in every step, based on which the corresponding node is selected. The outcome is that, in addition to root $N_1$, nodes $N_2$, $N_6$, and $N_4$ are also selected for storage.

It is clear that the complexity of Greedy is $O(k \times n^2)$, where n is the number of lattice nodes. Furthermore, the creators of Greedy have proven that, regardless of the given lattice, the total benefit of Greedy is at least 0.63 times the total benefit of the optimal algorithm. Although this does not necessarily imply near-optimal performance, nevertheless, it is an indication of certain guarantees in the effectiveness of the algorithm. The total benefit $B_T$ of a selection algorithm is defined as follows: Let $u_1, u_2, \ldots, u_k$ be the k nodes selected by the algorithm in their selection order. Let $B(u_i, S_{i-1})$ be the

benefit achieved by the selection of $u_i$, with respect to the set $S_{i-1}$ consisting of the root node plus $u_1, u_2, \ldots, u_{i-1}$. Then, define $B_T$ as $B_T = \sum_{i=1}^{k} B(u_i, S_{i-1})$.

### 5.2. Algorithm BPUS

BPUS [Harinarayan et al. 1996] is a variation of Greedy. The difference is in the constraint that must be satisfied: instead of having a limit on the number of nodes that can be materialized, there is an upper bound on the total storage space that the precomputed nodes can occupy. Again, this upper bound does not include the space occupied by the lattice root, which is always stored. Taking into account the storage capabilities of an OLAP server generates a more realistic criterion, since in practice, the problem often appears in this form.

BPUS functions in the same way as Greedy, proceeding iteratively and adding nodes in a set S, while the upper bound of disk space occupied by the nodes in S has not been exceeded. The only difference is in the way a node u is selected in the i-th iteration. The choice is not based on the absolute benefit B(u, S), but on the benefit per space-unit that u occupies—BPUS selects the node that does not belong to S and maximizes the fraction B(u, S)/C(u).

A problem arises when there is a very small node s with great benefit per space-unit and a much larger node b with similar benefit per space-unit. Selecting s may exclude b from further selection, since it might no longer fit. The absolute benefit of b, however, is much greater than the benefit of s. This fact may move the result away from any near-optimum solution. Nevertheless, BPUS also provides performance guarantees, similar to Greedy: If no lattice node occupies space larger than some fraction f of the totally allowed storage space, then the total benefit of the solution of BPUS will be at least (0.63 - f) times the total benefit of the optimal algorithm.

### 5.3. Algorithms Greedy-Interchange and Inner-Level Greedy

Gupta [1997] has studied the general problem of choosing a set of views for materialization in a data-warehouse environment, given some constraint on the total space that can be used, in order to optimize query response time. His proposed algorithms are based on a priori knowledge of queries that will be executed, their frequencies, and the cost of updating the materialized views, in order to estimate the total benefit of a solution, a parameter similar to that defined in Section 5.1. Note that the problem he states is more general than the problem solved by the algorithms we have previously described, since they focus on the selection of nodes of the cube lattice, not on the selection of any view that can be materialized.

Gupta presents general solutions, which are then adapted to the cube case. In particular, he proposes two new algorithms, namely Greedy-Interchange and Inner-Level Greedy.[1]

Algorithm Greedy-Interchange starts from the solution generated by BPUS and tries to improve it by exchanging nodes that have not been selected with selected nodes. Since the nodes have different sizes, it is possible to exchange one or more nodes with one or more other nodes. The algorithm iterates until there is no possible exchange that improves the total benefit.

Unfortunately, nothing has been proven about the effectiveness of this algorithm, except for the obvious fact that it constructs a solution at least as good as BPUS. Moreover, a great disadvantage of Greedy-Interchange is that there are no bounds

[1]The author also proposes an algorithm named "Greedy" which is, nevertheless, identical to the algorithm presented as BPUS in Section 5.2.

on its execution time. The paper introducing the algorithm mentions that in a great number of experiments, the algorithm usually terminates after some time that is at most a factor of 1.5 greater than the execution time of BPUS. However, no strict bound has been theoretically proven.

On the other hand, Inner-Level Greedy takes into account the existence of indexes on the selected nodes, which can also be stored if they positively affect the total benefit. Thus, the algorithm selects not only nodes but indexes as well. Inner-Level Greedy is also iterative. In each iteration, it selects a subset C that may consist of either one node and some of its indexes selected in a greedy manner or a single index of a node that has been already selected in some previous iteration. In particular, one can think of each iteration as consisting of two steps:

(1) In the first step, for each node $v_i$, a set $IG_i$ is constructed and is initialized to $\{v_i\}$. Then, the indexes of $v_i$ are added one by one to $IG_i$ in order of their incremental benefits, until the benefit per unit space of $IG_i$ with respect to S is maximized. Here, S denotes the set of nodes and indexes already selected. The algorithm chooses as C the $IG_i$ that has the maximum benefit per unit space with respect to S.

(2) In the second step, the algorithm selects an index of a node that already belongs to S. The selected index is the one whose benefit per unit space is the maximum with respect to S.

The benefit per unit space of C from the first step is compared with that of the index selected in the second step. The better one is added to S.

The complexity of the algorithm is $O(k^2 \times n^2)$, where k denotes the maximum number of nodes and indexes that fit in the space given according to the initial constraint, and n is the total number of nodes and indexes among which the algorithm selects. Furthermore, it has been proven that the total benefit of Inner-Level Greedy is at least 0.467 times the total benefit of the optimal algorithm, so, it never performs too poorly.

### 5.4. Algorithm r-Greedy

In the same spirit of Inner-Level Greedy, Gupta et al. [1997] proposed a variation called r-Greedy (as explained below, r is a numeric parameter used by the algorithm). It also takes into account the existence of indexes on the selected nodes, which are then stored, if they positively affect the total benefit. The only difference is that, in step 1 above, indexes are not selected one-by-one in a greedy manner (in order of their incremental benefits); instead, there is an upper bound of r-1 indexes that can be inserted in $IG_i$. The algorithm tries all possible combinations of up to r-1 indexes and selects the one that maximizes the benefit per unit space.

It is obvious that as r increases, the complexity of r-greedy increases, too. It is proven that an upper bound on the running time of the algorithm is $O(k \times n^r)$, where k denotes the number of nodes and indexes selected, and n is the total number of nodes and indexes among which the algorithm selects. Moreover, it has been shown that the total benefit of the solution produced by r-greedy is at least within a factor of $(1 - 1/e^{(r-1)/r})$ of the optimal total benefit achievable using the same space. This factor is equal to 0.39 for 2-Greedy, 0.49 for 3-Greedy, 0.53 for 4-Greedy and approaches 0.63 asymptotically. Nevertheless, it seems that in practice, using algorithm r-Greedy is not worth the additional complexity for $r>4$. Note also that Inner-Level Greedy is preferable to 2-Greedy, since the total benefit of the former is closer to the optimum than the total benefit of the latter for approximately the same running time.

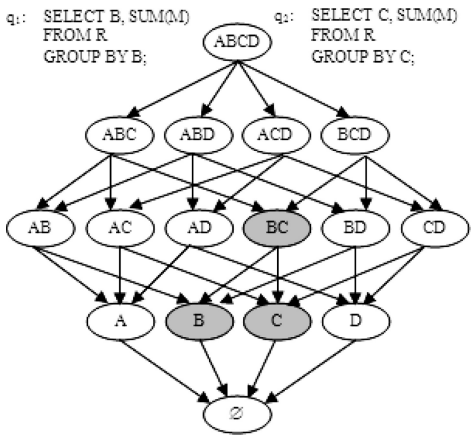


**Fig. 17**. An example of candidate views.

### 5.5. Algorithm MDred-lattice

Algorithm MDred-lattice [Baralis et al. 1997] takes a different approach from the previous algorithms: its objective is to select an appropriate subset of lattice nodes so that the system responds efficiently to certain known queries, balancing at the same time the space that this subset occupies on disk. Hence, the main difference is that it optimizes the average response time of queries in a particular workload and is not concerned with the overall average query. Given a set of queries SQ as the workload of interest, the algorithm defines the set of the so-called *candidate views*, which contains the lattice nodes that will be beneficial if saved. A lattice node $v_i$ belongs to the candidate-view set, if it satisfies one of the following two conditions:

(1) Node $v_i$ is associated to some query $q_i \in SQ$, in particular, the one that contains in its group-by clause the same grouping attributes as $v_i$.
(2) There are two candidate views $v_j$ and $v_k$, which have $v_i$ as the most specialized common ancestor.

MDred-lattice functions iteratively and returns the set L of all candidate views. L is initialized to the set of nodes associated with the queries in SQ. In each step, L is extended with the nodes that are the most specialized common ancestors of node pairs that already belong to L. The algorithm stops when all candidate views are found.

For example, suppose that the given lattice is the one shown in Figure 4 and that SQ consists of two queries $q_1$ and $q_2$. The candidate views are shaded in Figure 17. Nodes B and C are candidate views, since they are associated to queries $q_1$ and $q_2$, respectively. Node BC is a candidate view because it satisfies the second condition.

Since the number of candidate views is too large in practice, the authors of the paper propose a variation of the algorithm that takes into account the (estimated) size of each candidate view as well. Depending on the sizes of candidate views, it is decided whether or not a level of the lattice is too specialized so materialization of its nodes does not offer great benefit for answering the queries of interest. In this case, the nodes of the selected level are substituted by some of their ancestors in higher levels.

### 5.6. Algorithm PBS

PBS [Shukla et al. 1998] is in the spirit of BPUS, but uses a simpler heuristic for solving the problem of selecting a cube subset for storage. As in BPUS, an upper bound

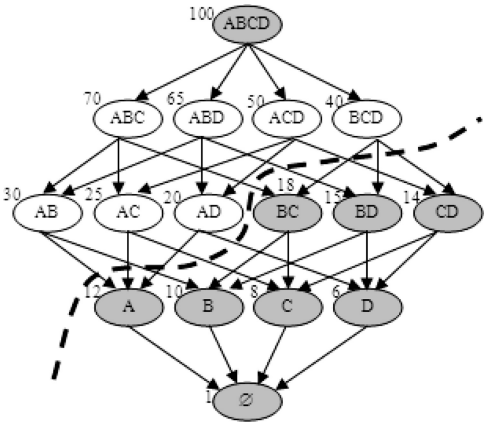


**Fig. 18**. The nodes selected by PBS.

on the total storage space that precomputed nodes can occupy must be satisfied. PBS initializes some set S to the root of the lattice and then iterates, each time selecting and adding into S the smallest remaining node, until it exceeds the space limit posed by the initial constraint.

Interestingly, PBS proceeds in a bottom-up fashion, traversing the lattice from smaller and more specialized nodes towards larger and more detailed ones. Proceeding this way, PBS creates a frontier that separates the selected from the nonselected nodes. Recall that the lower a node is in the lattice, the smaller it is, since the number of aggregations it has been subject to increases from level to level.

The algorithm is rather simple, since it calculates no cost and does not take into account a benefit quantity like Greedy and BPUS. It only needs an estimation of the size of each node. Hence, its complexity is O(n $\times$ logn), where n is the number of lattice nodes. The factor n $\times$ logn comes from ordering the nodes by their size. Figure 18 shows the lattice of a 4-dimensional cube and the estimated size of each node in blocks. Let the available storage space be 200 blocks. The nodes that are selected for storage have been shaded. The dashed line denotes the frontier that separates the selected from the nodes not selected. Note that 16 blocks remain unexploited.

The main disadvantage of the algorithm is that it sacrifices the quality of its result in order to accelerate the process. In particular, as for BPUS, the total benefit of the PBS solution is at least within a factor of (0.63 - f) of the optimal algorithm total benefit (where f is the maximum fraction of the size occupied by a node with respect to the total available space). For PBS, however, this property holds only under strict conditions. More precisely, such performance guarantees can be given only for the subclass of Size Restricted hypercube lattices (or SR-hypercube lattices), which have the following special ordering between the sizes of parent and children nodes: for each pair of nodes u, w, where w is the parent of u and different from the root, if k denotes the number of children of w, then $|u|/|w| \leq 1/(1+k)$. In other words, in SR-hypercube lattices, the size of a node is equally distributed to its children, which have to be at least $(1+k)$ times smaller than their parent.

### 5.7. Algorithm Inverted-Tree Greedy

Most of the algorithms presented up to this point provide solutions to the view selection problem under a space constraint. Nevertheless, since the price/capacity ratio of secondary storage media keeps decreasing, the limiting factor during the view selection

process is not always the space that is available for the storage of the selected materialized views; it can also be the time necessary for their incremental maintenance, a process typical in data warehouses.

The view selection problem under a maintenance-cost constraint seems similar to the view selection problem under a space constraint except for one fundamental difference [Gupta and Mumick 1999] that makes it harder to solve: the space occupied by a set of views monotonically increases when more views are added to the set, whereas the maintenance cost does not. In other words, it is possible that the maintenance cost of a set of views decreases after the insertion of one more view in the set, if for example the latter is an ancestor of some views in the original set and its updated data can be used as intermediate results to accelerate the update process of its descendants. Because of the nonmonotonic property of the update cost, a straightforward greedy algorithm, in the spirit of BPUS, may produce a result of arbitrarily poor quality compared to the optimal solution [Gupta and Mumick 1999]. Inverted-Tree Greedy [Gupta and Mumick 1999] is an algorithm that overcomes this problem.

Inverted-Tree Greedy uses a measure called *benefit per unit of effective maintenance-cost* (hereafter simply called benefit), which is similar to the benefit per space-unit used by BPUS, but which takes into account maintenance costs instead of storage costs [Gupta and Mumick 1999]. As mentioned above, incrementally building the final solution by greedily choosing the lattice node with the largest benefit with respect to the set S of nodes already selected for materialization has been found to produce results of poor quality. Alternatively, Inverted-Tree Greedy iteratively selects the most beneficial *inverted-tree set* with respect to S, consisting of a set of nodes P of the cube lattice that do not belong to S. A set of nodes P is defined to be an inverted-tree set in a directed graph G (in our case, the cube lattice) if there is a not necessarily induced subgraph $T_P$ in the transitive closure of G such that the set of vertices of $T_P$ is P and the inverse graph[2] of $T_P$ is a tree.

In other words, Inverted-Tree Greedy works iteratively while the maintenance cost constraint is satisfied. In each step, it considers all possible inverted-tree sets of the cube-lattice nodes that do not belong to S. Among them it selects the one with the largest benefit with respect to S and inserts its nodes into S. Note that, unlike the greedy methods presented above, in each step, Inverted-Tree Greedy selects a set of nodes instead of a single one.

It has been proven [Gupta and Mumick 1999] that under certain conditions, which usually hold in practice, Inverted-Tree Greedy returns a solution S whose total benefit is at least 0.63 times the total benefit of the optimal solution that incurs at most the same maintenance cost. Unfortunately, in the worst case, the total number of inverted-tree sets in a directed graph is exponential in the size of the graph, which increases the computational costs of the method, raising some concerns about its applicability in real-world applications.

Finally note that, unlike Inverted-Tree Greedy, which selects sets of nodes in every iteration, there are also several other greedy algorithms [Liang et al. 2001; Uchiyama et al. 1999] that provide solutions to the view selection problem under a maintenance-cost constraint by selecting one node at a time. These algorithms overlook the hint of Gupta and Mumick [1999] that such behaviour may result in solutions of arbitrarily low quality. As expected, they provide no guarantees for the quality of the final result; hence, we do not study them further in this article.

[2]The inverse of a directed graph is the graph with its edges reversed.

### 5.8. Algorithm PGA

Most of the greedy algorithms presented here have been shown to have polynomial complexity with respect to the number of nodes n in the cube lattice. Nevertheless, this result is not as promising as it initially appears; actually, it may be misleading, since the number of nodes n in the cube lattice grows exponentially with the number of dimensions D of the fact table. For example, under this perspective, the complexity $O(k \times n^2)$ of BPUS is equal to $O(k \times 2^{2D})$. Algorithm PGA [Nadeau and Teorey 2002] overcomes this problem by using a heuristic that is polynomial in the number of dimensions D.

PGA is iterative, with every iteration divided into two phases, nomination and selection. During nomination, PGA selects a path of D promising nodes from the root to the bottom of the lattice. The nodes in this path are the candidates considered for materialization in the second phase. The nomination phase begins with the root as the parent node and estimates the sizes of all its children nodes. The smallest child that has not yet been selected during a previous iteration is nominated as a candidate for materialization. The nomination process then moves down one level. The nominated child is considered as a parent, and the smallest child of that node that has not yet been selected during a previous iteration is nominated as a candidate view. The process continues until the bottom of the lattice is reached. When the nomination phase completes, PGA moves to the selection phase, during which it selects for materialization the candidate view with the largest estimated benefit. Note that the benefit of materializing a candidate view is not calculated by recomputing the query response time of all its descendants, as performed for example by BPUS. On the contrary, it is estimated based on some metadata specifically kept for this purpose.

Based on this description, it should be evident that PGA avoids the two sources of exponential complexity with respect to D of all previous greedy algorithms: First, during its iterations it does not consider all remaining nodes on the entire lattice, which are in the order of $2^D$, but only a path of D candidate nodes. Second, it does not calculate the benefit of a candidate node by visiting all its descendants, which are again in the order of $2^D$ in the worst case, but it estimates this benefit in O(1) time based on specialized metadata. Hence, the creators of PGA [Nadeau and Teorey 2002] have shown that its time complexity is $O(k^2 \times D^2)$, where k is the number of nodes finally selected.

Note that, unlike previous greedy algorithms, which are exponential in D, PGA offers no theoretical guarantees about the quality of the solution it provides. Its creators have shown in a rather limited set of experiments that the total benefit of the solution generated by PGA is not much worse than that of BPUS.

### 5.9. Algorithm Key

Algorithm Key [Kotsis and McGregor 2000] differs from all previous methods on even the criterion it uses to select the subset of lattice nodes for materialization. It ignores all nodes that contain only totally-redundant tuples. As defined in Section 3.4, these are the tuples that can be constructed by a simple projection (without duplicate elimination) of the fact table, without any additional computation.

The authors of the corresponding paper were the first to recognize that a data cube contains redundancy and proposed the theory of g-equivalent tuples: Tuple $t_{R'}$ of a relation R' is defined to be g-equivalent to tuple $t_R$ of relation R if the set of grouping attributes of R' is a subset of the set of grouping attributes of R and the value of the measure of interest is equal in both tuples. Furthermore, a relation R' is defined to be g-equivalent to relation R if, for every tuple in R', there is a g-equivalent tuple in R and both relations have the same number of tuples. Cube nodes (relations that belong to the data cube) that are g-equivalent to other nodes are redundant and need not be stored.

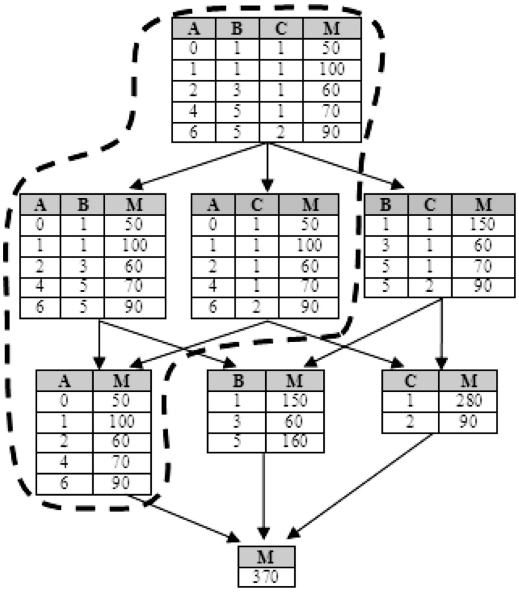


**Fig. 19**. The totally redundant nodes of the cube of R.

Algorithm Key traverses the lattice nodes in a bottom-up/breadth-first fashion, searching for *observational keys* among their grouping attributes. An observational key is a minimal set of attributes whose values uniquely identify each tuple of the original fact table. Observational keys are similar to candidate keys [Codd 1970] with the fundamental difference that they are a property of the specific data instance and not of the data schema. Clearly, any node whose attribute set is a superset of an observational key contains the same number of tuples as the fact table and can actually be constructed on the fly by a simple projection of the fact table, since none of its tuples can be the result of aggregation at some higher level. Hence, such nodes are considered as totally redundant and do not need to be materialized. It is straightforward to show that, if a node is totally redundant, the same property holds for all of its ancestor nodes in the cube lattice as well. Thus, after identification of an observational key, Algorithm Key can immediately remove an entire tree of nodes. This property is the reason why bottom-up traversal is ideal for Key: discovering observational keys with fewer attributes leads to pruning larger trees of nodes and to condensing the final result at greater levels.

An important property of the algorithm is that, to construct a nonprecomputed, totally-redundant node on the fly, no aggregation is necessary but only a simple scan of the fact table. Hence, the cube may be considered as fully materialized, even if some nodes are not stored. Key was the first algorithm proposed with this property, which leads to low average query-response times. For instance, consider the lattice shown in Figure 19. Note that attribute sets A, AB, AC, and ABC are supersets of the observational key A, whose values uniquely identify each tuple of the original fact table R (Figure 1a). Hence, the corresponding nodes are totally redundant and need not be stored.

Algorithm Key has a complexity of $O(T \times n)$ in the worst case, where T is the number of tuples in the original fact table and n the number of nodes in the cube lattice. Nevertheless, the authors of the corresponding paper [Kotsis and McGregor 2000] claim

that in practice the algorithm performs much better. This argument is based on the following properties:

(1) A set of attributes that is not an observational key can be identified relatively quickly, after locating the first duplicate, without having to scan the entire fact table. This property reduces factor T.
(2) Identification of a totally-redundant node leads to ignoring the entire tree of its ancestors. This property reduces factor n.

Moreover, note that according to the creators of the algorithm, further savings in storage can be achieved by applying the Key algorithm recursively to lattice nodes other than the fact table, for example, to the root's children. An obvious implementation of this variation treats every such child node as an input of the Key algorithm and identifies all totally redundant nodes with respect to that child node. Experiments have illustrated that storage savings of up to 85% overall are achieved when redundancy is eliminated recursively.

### 5.10. Algorithm DynaMat

All selection algorithms previously described are static in the sense that, given some restriction regarding the available space and/or update time and sometimes some description of the expected workload, they statically suggest a set of views to materialize for better performance. Nevertheless, the nature of decision support is dynamic, since both data and trends in it keep changing; hence, a set of views statically selected might quickly become outdated. Keeping the set of materialized views up to date requires that some of the aforementioned view-selection algorithms may be rerun periodically, which is a rather tedious and time-consuming task. DynaMat[3] [Kotidis and Roussopoulos 1999] overcomes this problem by constantly monitoring incoming queries and materializing a promising set of views subject to space and update time constraints.

Another strong point of DynaMat is that it does not materialize entire views but only segments of them that correspond to results of aggregate queries with certain types of range selections on some dimensions; hence, DynaMat performs selection at a finer level of granularity. The segments are stored in a so-called view pool, which is initially empty.

DynaMat distinguishes two operational phases: on-line and update. During the on-line phase the system answers queries. When a new query arrives, DynaMat checks if it can answer it efficiently using a segment already stored in the pool. Otherwise, DynaMat accesses the source data stored in the fact table (and potentially some relevant indexes) to answer the query. In both cases, after computing the result, DynaMat invokes an admission control module to decide whether or not it is beneficial to store the result in the pool. If the result fits in the pool, the admission control module always permits its storage for future use. Otherwise, if the pool size has reached its space limit, the admission control module uses a measure of quality (e.g., the time when a segment was last accessed) to locate candidate segments for replacement. A segment already stored in the pool is considered for replacement only if its quality is lower than the quality of the new result. If no appropriate replacement candidates are found, the admission control module rejects the request for storage of the new result in the pool. Otherwise, it applies some replacement policy (e.g., LRU) to replace some old segments with the new one. Thus, DynaMat guarantees that the size of the pool never exceeds its limits.

[3]DynaMat is actually more than just a view-selection algorithm. It is a comprehensive view management system for data warehouses. In this article, for simplicity, we use DynaMat to refer to just the selection algorithm used in that system.

During the update phase, the system receives updates from the data sources and refreshes the segments materialized in the pool. In order to guarantee that the update process will not exceed the available update-time window, DynaMat estimates the time necessary for updating the materialized segments. If this exceeds the system constraints, DynaMat again uses a policy based on the chosen measure of quality to discard some segments from the pool and decrease the total time necessary for the update process. Initially, it discards all segments whose maintenance cost is larger than the update window. Then, it uses a greedy algorithm that iteratively discards segments of low quality until the constraint is satisfied. (As already mentioned, discarding some materialized data does not necessarily imply a decrease in the overall update time. DynaMat takes this into account by not considering for replacement, segments that contribute to the acceleration of the update time of other segments.) The remaining segments are updated and the system returns to the on-line phase.

### 5.11. Algorithms of Randomized Search

All selection algorithms presented so far use systematic approaches to identify parts of the cube whose materialization benefits the total query response time under some given constraints. As we have already seen, these algorithms usually provide acceptable guarantees on the quality of the final solution compared to the quality of the optimal solution achievable under the same constraints. Nevertheless, systematic/exhaustive search may be computationally expensive in real world applications that involve large multidimensional datasets. For these cases, as with other hard combinatorial optimization problems, randomized search is a potential solution [Kalnis et al. 2002]. Although it does not always provide guarantees for the quality of its results, it has been proven very efficient for some classes of optimization problems (e.g., optimization of large join queries [Ioannidis and Kang 1990]).

Based on these observations, Kalnis et al. [2002] proposed the use of well known algorithms for randomized search in order to solve the view-selection problem under a space constraint, under a maintenance-cost constraint, or under both constraints simultaneously. To make such algorithms applicable, they modelled the view-selection problem as a connected state graph. Each node/state corresponds to a set of views that satisfy the space/time constraints and is labelled with a cost representing the expected query response time if the particular set of views is materialized. Each edge represents a transition from one state to another after applying a simple transformation. A transition is called downhill if the cost of the destination state is lower than the cost of the source state; otherwise, it is called uphill. A state that is the source of only uphill transitions is called a local minimum; the local minimum with the minimum cost overall is called global minimum. For the view-selection problem under a space constraint, Kalnis et al. [2002] defined the following types of transition:

(1) Pick an unselected view and add it to the existing selection. If the space constraint is violated, randomly remove selected views until a valid state is reached.
(2) Remove a selected view and "fill" the rest of the space with views picked randomly.

Furthermore, for the view-selection problem under a maintenance cost constraint, they proposed the following transitions (these transitions differ from the previous ones due to the nonmonotonic nature of the maintenance cost, which has been already analyzed above):

(1) Pick an unselected view and add it to the existing solution.
(2) Replace a selected view with an unselected one.
(3) Remove a selected view.

The particular randomized algorithms that they have used include Random Sampling (RS), Iterative Improvement (II, which only follows downhill transitions), Simulated Annealing (SA, which follows downhill transitions but allows some uphill transitions as well, with a probability decreasing with time and uphill cost difference), and Two-Phase Optimization (2PO, which combines II and SA). These algorithms are well known and their detailed description exceeds our purpose [Kalnis et al. 2002].

As Kalnis et al. [2002] note, an advantage of randomized search algorithms is that their parameters can be tuned to achieve a tradeoff between execution cost and solution quality. In their implementation, they have determined their final parameter values through experimentation. Given these particular values and a cube lattice with n nodes, they have calculated the worst-case execution cost of the randomized algorithms as follows: the cost of II is $O(h \times n \times \log n)$, where h is the depth of a local minimum, whereas the cost of SA and 2PO is bounded by $O(n^2 \times \log n)$. The experimental evaluation has shown that in practice 2PO is the most promising randomized search algorithm, converging faster, to a solution of better quality, than the rest. Interestingly, the total benefit of 2PO has been found usually very close to, and sometimes even better than, the total benefit of BPUS (recall that BPUS is a systematic algorithm of greater complexity, since it is expected that $k > \log n$, where k is the number of nodes selected by PBUS).

In a similar spirit, there are also other randomized algorithms for view selection [Lawrence 2006; Lee and Hammer 1999; Yu et al. 2003], mainly based on genetic algorithms. Reviewing all such algorithms exceeds the purpose of this article.

## 6. INTEGRATED METHODS

In Sections 4 and 5, we studied algorithms that treat the problems of data cube computation and selection as separate. In this section, we follow the most recent tendency and study algorithms BUC (with minsup $> 1$), Vector-Aggregator (V-Aggregator), MinCube, Bottom-Up-Base-Single-Tuple (BU-BST), Reordered-BU-BST (RBU-BST), Quotient-Cube-Depth-First-Search (QC-DFS), and Cubing-Using-a-ROLAP-Engine (CURE), which are integrated.

### 6.1. Algorithm BUC with minsup>1

BUC has already been presented in Section 4.7 as an efficient method for data cube computation. During its description, we mentioned that it uses the parameter minsup in order to prune partitions that do not satisfy some minimum support constraint. If minsup is set to 1 (and the aggregate function is count), the algorithm computes and stores an entire cube. However, if minsup>1, the algorithm applies some additional form of selection, to condense the result. The condensed cube that is stored is appropriate for answering queries that include a HAVING clause that poses similar support thresholds. So, as mentioned in Section 4.7, when minsup>1, BUC solves the Iceberg-Cube problem.

Although we have introduced BUC as a computation method, when combined with some form of selection processing it can be classified as an integrated method as well.

### 6.2. Algorithm V-Aggregator

Algorithm V-Aggregator [Kotsis 2000] is in the spirit of Algorithm Key, which was presented in Section 5.9. It constructs the data cube while searching for redundant data at the same time. Since redundancy is removed, the final result is condensed.

Unlike Algorithm Key, which identifies totally redundant nodes, nodes that can be constructed with a simple projection of the fact table, V-Aggregator searches for partially redundant tuples. As also defined in Section 3.4, a tuple of a node is considered partially redundant when it has not been produced by some aggregation on the parent

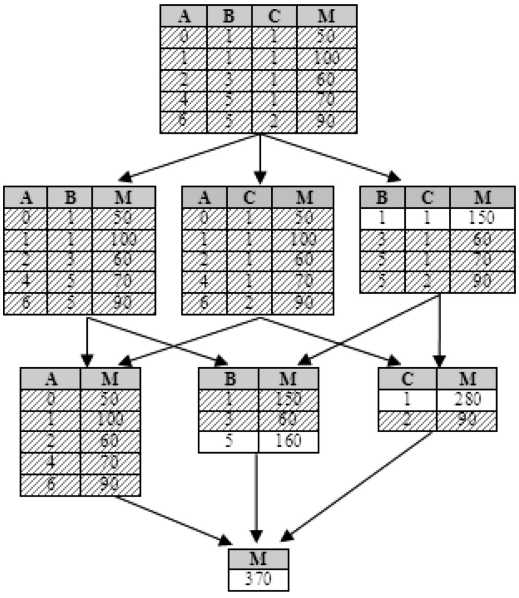


**Fig. 20**. The partially redundant tuples of the cube.

node, but by some simple projection of a single tuple on that node. A node that contains partially redundant tuples is called partially redundant. For example, in the cube of Figure 20 we have shaded the partially redundant tuples. (We consider the tuples of the lattice root as partially redundant, because we assume that, in some sense, the fact table is its parent.)

Note that all tuples of a totally redundant node are partially redundant as well. Note also that the notion of a partially redundant node only denotes some relationship between this node and its parent. No conclusions can be drawn about other ancestor nodes in higher lattice levels. On the contrary, recall that by definition, all ancestors of a totally redundant node are totally redundant as well.

V-Aggregator traverses the lattice in a top-down fashion, computing one node at a time from one of its parents. Many heuristics can be applied for selecting the proper parent. This selection is important, since it affects the level of redundancy that can be detected. In Figure 20, if we select node AB as the parent of B, then the tuple $\langle 1, 150 \rangle$ cannot be detected as redundant, whereas, if we use BC, it can. Detection of a parent that maximizes the number of identifiable redundant tuples is an open issue.

For the evaluation of aggregations at a node, the algorithm uses hashing. It assumes that the parent of the node currently being computed, and all necessary data structures, fit in main memory, even if the parent node is the root of the lattice. Hence, it applies no partitioning scheme to transfer the data from the disk. While aggregating, V-Aggregator stores information in appropriate bitmap vectors that allow fast detection of redundant tuples and an overall computation of a node with only one scan of its parent.

An important property of the stored result of V-Aggregator is that, to generate any of the redundant, nonstored tuples of a node, no aggregation is necessary, but simply recursive access to the corresponding tuples stored in its ancestors. Hence, the cube may be considered to be fully materialized, even if not all tuples are stored. This remark is important, because it explains why the V-Aggregator algorithm exhibits low average query response time.

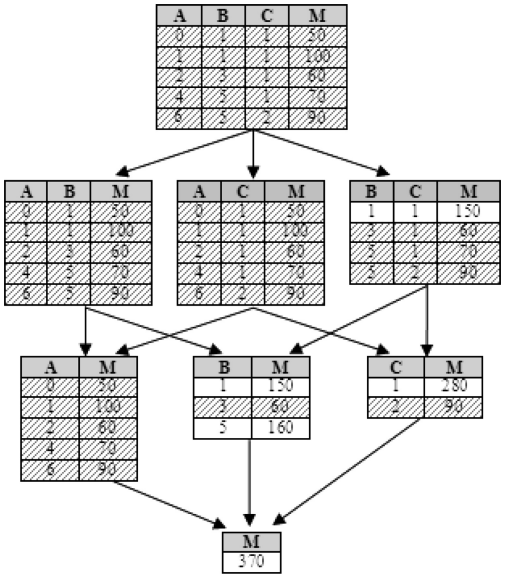


**Fig. 21**. The totally redundant tuples of the cube of R.

### 6.3. Algorithm MinCube

Algorithm MinCube [Wang et al. 2002] follows a philosophy similar to that of algorithms Key and V-Aggregator, which remove redundancy in order to decrease the size of the final result. Like Key and V-Aggregator, MinCube computes and stores a fully precomputed condensed cube that needs neither decompression nor extra aggregations at query time. Hence, like these other algorithms, size reduction contributes not only to saving storage resources, but also to fast query answering. The difference lies in the specific method used for finding redundant tuples, which is explained below.

In the corresponding paper, the notion of a *Base Single Tuple* (BST) is defined: Let SD be an attribute set and assume that the tuples of the original (base) fact table are partitioned on SD. If some partition consists of a single tuple r, then r is called a BST on SD. Furthermore, let SDSET be a set of attribute sets. If a tuple r is a BST on all sets of SDSET, it is characterized as BST on SDSET. In other words, using the terminology of Section 3.4, a BST on SDSET can be characterized as a totally redundant tuple.

Based on the preceding definitions, a *BST-Condensed Cube* is a cube in which some[4] BSTs on a set of attribute sets SDSET have not been stored. Moreover, a BST-Condensed Cube that stores no totally redundant tuples is called a *Minimal BST-Condensed Cube*. Algorithm MinCube computes a Minimal BST-Condensed Cube; hence it avoids storing any totally-redundant tuple, in the same sense that Key and V-Aggregator avoid storing totally-redundant nodes, and partially-redundant tuples, respectively.

For instance, in Figure 21, the totally-redundant tuples of R (Figure 1a) appear shaded. By definition, the tuples in the lattice root are BST on the set of its grouping attributes (assuming that the fact table contains no duplicates). Note also that all tuples of a totally redundant node N are BST on the set of the grouping attributes of N, so are totally redundant, and therefore, partially redundant as well (Section 3.4).

[4]The above definition deliberately refers to some and not to all possible BSTs, because the corresponding paper [Wang et al. 2002] also contains algorithms that do not remove all BSTs (e.g. BU-BST, which appears in Section 6.4).

| A | B | C | M | SDSET |
|---|---|---|---|---|
| 0 | 1 | 1 | 50 | {A, AB, AC, ABC} |
| 1 | 1 | 1 | 100 | {A, AB, AC, ABC} |
| 2 | 3 | 1 | 60 | {A, B, AB, AC, BC, ABC} |
| 4 | 5 | 1 | 70 | {A, AB, AC, BC, ABC} |
| 6 | 5 | 2 | 90 | {A, C, AB, AC, BC, ABC} |
| * | 1 | * | 150 | {} |
| * | 5 | * | 160 | {} |
| * | * | 1 | 280 | {} |
| * | 1 | 1 | 150 | {} |
| * | * | * | 370 | {} |

**Fig. 22**. The Minimal BST-Condensed Cube of R.

Comparing Figure 21 with Figure 20, we see that their only difference is tuple ⟨1, 150⟩ in node B, which is partially, but not totally, redundant.

An interesting property is that if a tuple is BST on an attribute set SD, then it is also BST on every superset of SD. For example in Figure 21, since tuple ⟨0, 1, 1, 50⟩ is BST on SD = {A}, it is also BST on all its supersets: {A, B}, {A, C}, and {A, B, C}. This is similar to what holds for totally redundant nodes, where if a node is totally redundant, then its ancestors are totally redundant as well.

Figure 22 illustrates the Minimal BST-Condensed Cube that corresponds to fact table R (Figure 1a). To compute such a Minimal BST-Condensed Cube, MinCube traverses the lattice in a bottom-up/breadth-first fashion computing a node in each step, and identifying redundant tuples at the same time. The algorithm uses appropriate bitmap vectors that enable easy identification of tuples already characterized as BST on the set of the grouping attributes of the currently processed node during the computation of other nodes in lower lattice levels. This optimization saves time and makes the algorithm more effective, since these tuples need not be stored. At each node N, MinCube first filters the tuples that have already been characterized in a previous step as BST on the set SD of N's attributes. Then, it sorts the remaining tuples with respect to SD and proceeds with computation. It processes the sorted tuples sequentially, but it saves no result until it has processed a whole partition of tuples that share the same values in SD. If it finds such a partition that contains only one tuple, then it labels the latter BST on SD and all its supersets.

MinCube constructs the optimal result, building a Minimal BST-Condensed Cube. However, it pays a high cost that deteriorates its efficiency: traversing the lattice in a bottom-up/breadth-first fashion, it has to compute each node from scratch, without being able to take advantage of node commonalities or to share computational costs among multiple nodes. This problem is addressed by BU-BST, which is elaborated in the next subsection.

### 6.4. Algorithm BU-BST and RBU-BST

Like MinCube, Algorithm BU-BST [Wang et al. 2002] is based on the notion of a BST-Condensed Cube. It is more efficient than MinCube, since it constructs the cube faster, but the resulting BST-Condensed Cube is not minimal, since it fails to detect all totally redundant tuples.

The objective behind BU-BST was to combine the speed of BUC (fastest bottom-up algorithm) with the MinCube idea of pruning totally redundant tuples in order to reduce the storage space needs. Similarly to BUC, BU-BST is recursive and traverses the cube lattice in a bottom-up/depth-first fashion. Its structure is identical with that of BUC, hence, we do not repeat all the details here (refer back to Section 4.7). The most important difference from BUC is at the point when a recursive call is made to

| A | B | C | M | SDSET |
|---|---|---|---|---|
| * | * | * | 370 | {} |
| 0 | 1 | 1 | 30 | {A, AB, AC, ABC} |
| 1 | 1 | 1 | 100 | {A, AB, AC, ABC} |
| 2 | 3 | 1 | 60 | {A, AB, AC, ABC} |
| 4 | 5 | 1 | 70 | {A, AB, AC, ABC} |
| 6 | 5 | 2 | 90 | {A, AB, AC, ABC} |
| * | 1 | * | 150 | {} |
| * | 1 | 1 | 150 | {} |
| * | 3 | 1 | 60 | {B, BC} |
| * | 5 | * | 160 | {} |
| * | 5 | 1 | 70 | {} |
| * | 5 | 2 | 90 | {} |
| * | * | 1 | 280 | {} |
| * | * | 2 | 90 | {} |

**Fig. 23**. The BU-BST Cube of R.

the algorithm with input, some partition that contains only one tuple. In BU-BST, this tuple is characterized as a BST on the appropriate sets of the grouping attributes, and the algorithm ignores the tuple in all higher levels. Figure 23 illustrates the BU-BST cube of R (Figure 1a)—the cube of R constructed by BU-BST. Expectedly, the size of this BU-BST cube is larger than the size of the Minimal BST-Condensed Cube of Figure 22, since BU-BST fails to identify on time, some totally-redundant tuples.

It is interesting to note that it is the authors of BUC who first identified the existence of many single-tuple partitions. They even introduced an optimization in BUC: whenever a single-tuple partition is identified, a method WriteAncestors is called and automatically updates all higher levels in the lattice, interrupting the recursion earlier, and avoiding unnecessary recursive calls. Method WriteAncestors however, writes the same tuple multiple times in the output, once for each node in the higher levels that contains it. BU-BST goes one step further and does not only interrupt the recursion early, but also condenses the cube, avoiding the storage of redundant tuples multiple times.

As mentioned earlier, BU-BST does not guarantee finding a Minimal BST-Condensed Cube. The reason is that, similarly to BUC, it has to decide an ordering for the original fact table dimensions, according to which it traverses the lattice in a depth-first fashion. Depending on the node-visit order, it is possible that a redundant tuple is identified late, after it has been written to the output. Such storage of redundant information is the price that the algorithm has to pay for the sake of faster computation. Hence, the condensation percentage of the cube finally stored depends on the choice of the dimension ordering. A good choice may result in a near Minimal BST-Condensed Cube. Toward this end, as in BUC, two heuristics are proposed: using dimensions in decreasing order of their cardinalities or in increasing order of their maximum number of duplicates.

The choice of a good dimension ordering is so crucial that it has led the authors of BU-BST to propose a variation, called RBU-BST, which computes the data cube twice. During the first pass, no output is written, but statistics are gathered. Based on these, RBU-BST makes the best choice for ordering the dimensions in the second pass. Hence, a near Minimal BST-Condensed Cube is created, achieving up to 80–90% condensation [Wang et al. 2002]. Since the time consumed for writing the output takes a considerable fraction of the total execution time, the two passes of RBU-BST may be a competitive solution.

### 6.5. Algorithm QC-DFS

QC-DFS [Lakshmanan et al. 2002] is an extension of BU-BST that identifies at least all partially redundant tuples in the cube. In particular, it takes all cube tuples that

| A | B | C | M | SDSET |
|---|---|---|---|---|
| * | * | * | 370 | {} |
| 0 | 1 | 1 | 50 | {A, AB, AC, ABC} |
| 1 | 1 | 1 | 100 | {A, AB, AC, ABC} |
| 2 | 3 | 1 | 60 | {A, AB, AC, ABC} |
| 4 | 5 | 1 | 70 | {A, AB, AC, ABC} |
| 6 | 5 | 2 | 90 | {A, AB, AC, ABC} |
| * | 1 | * | 150 | {} |
| * | 1 | 1 | 150 | {} |
| * | 3 | 1 | 60 | {B, BC} |
| * | 5 | * | 160 | {} |
| * | 5 | 1 | 70 | {} |
| * | 5 | 2 | 90 | {} |
| * | * | 1 | 280 | {} |
| * | * | 2 | 90 | {} |

**Fig. 24**. The QC-Table of R.

are produced by the aggregation of the same set of fact table tuples (which include all partially redundant tuples) and assigns them into an equivalence class of cells with identical aggregate values. By doing so for all such sets of tuples, QC-DFS generates a so called Quotient Cube and stores it in a relational table, called QC-Table [Lakshmanan et al. 2002].

Figure 24 illustrates the QC-Table[5] that corresponds to the fact table R in Figure 1a. *Class-id* stores a unique identifier for each class of equivalent tuples and *Lower-Bounds* stores an expression that encapsulates the lower boundaries of the class, by capturing in all paths, how far down in the cube lattice the corresponding QC-Table tuple can be used as is (by simple projections on its dimension values). For example, the class with id = 5 (in Figure 24) actually represents five redundant tuples in the uncompressed cube of Figure 1b ($\langle 4, 5, 1, 70\rangle$ in node ABC, $\langle 5, 1, 70\rangle$ in BC, $\langle 4, 1, 70\rangle$ in AC, $\langle 4, 5, 70\rangle$ in AB, and $\langle 4, 70\rangle$ in A). These are the tuples produced by the tuple $\langle 4, 5, 1, 70\rangle$ (stored in the fifth row of the QC-Table itself), by projecting A out of that tuple (as indicated by the A part in the Lower-Bounds expression A $\vee$ BC), and by projecting B, C, and BC out of it (as indicated by the BC part in the expression A $\vee$ BC).

### 6.6. Algorithm CURE

Algorithm CURE has recently been proposed [Morfonios and Ioannidis 2006a] as the first comprehensive solution that combines the following advantages: It is a pure ROLAP method, fully compatible with the relational model, which makes it easy to implement over any existing relational engine. Furthermore, it has been designed not only for "flat" datasets but also for handling those whose dimension values are organized in hierarchies. Additionally, it applies an efficient external partitioning algorithm that allows the construction of cubes over very large fact tables whose size may exceed the size of main memory by far. Finally, it stores cubes in a very compact format, eliminating all kinds of redundancy from the final result. These features make CURE the most promising algorithm for cube construction in the ROLAP framework [Morfonios and Ioannidis 2006a].

The creators of CURE started from the observation of several other researchers that a cube contains a large amount of redundant data and unified all types of redundancy identified so far under a simple framework: *a value that is stored in a data cube is called redundant if it is repeated in the same attribute somewhere else in the cube as well*. According to this, CURE recognizes two types of redundancy: *dimensional redundancy*, which appears whenever a specific dimension value is repeated in different tuples,

[5]In this example, we have computed a Quotient Cube with respect to cover partition, which seems more interesting in practice (refer to the original paper [Lakshmanan et al. 2002] for more details).


**Fig. 25**. The schema of NT, TT, and CAT relations.

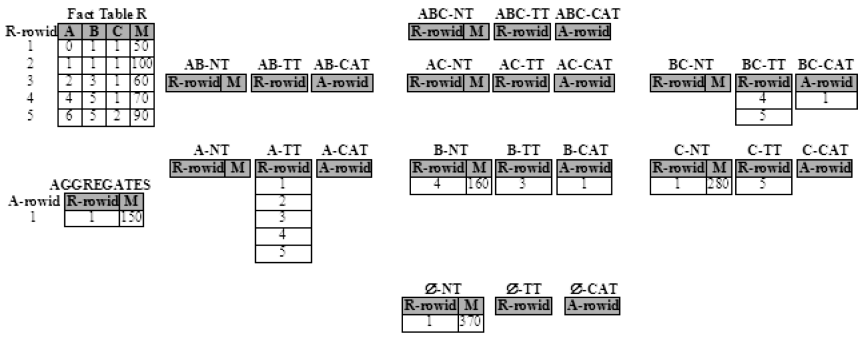


**Fig. 26**. Example of a CURE cube.

and *aggregational redundancy*, which appears whenever a specific aggregate value is repeated in different tuples.

To remove the aforementioned types of redundancy and store the final cube more compactly, CURE separates cube tuples into three categories: *Normal Tuples* (NTs), *Trivial Tuples* (TTs), and *Common Aggregate Tuples* (CATs) [Morfonios and Ioannidis 2006a]. A cube tuple t that has been produced by aggregation of a tuple set S in the original fact table (say R) is NT if it is dimensionally but not aggregationally redundant, TT if it comes from a singleton set S ($|S| = 1$), and CAT if it is aggregationally redundant and nontrivial ($|S|>1$). For example, according to these definitions, tuple $\langle 5, 160\rangle$ stored in node B of the cube shown in Figure 1b is an NT, since its aggregate value, 160, is unique in the cube; tuple $\langle 3, 60\rangle$ is a TT, since it has been generated by a singleton set of tuples $S = \{\langle 2, 3, 1, 60\rangle\}$ in the fact table R appearing in Figure 1a; and tuple $\langle 1, 150\rangle$ is a CAT, since it shares the same aggregate values with tuple $\langle 1, 1, 150\rangle$ in node BC but is nontrivial, coming from the aggregation of a nonsingleton set of tuples $S = \{\langle 0, 1, 1, 50\rangle, \langle 1, 1, 1, 100\rangle\}$ in R.

To condense the cube, CURE uses at most three relations per cube node, one for each category of tuples (clearly, empty relations are not physically stored). Figure 25 shows the schema of these relations, assuming that each cube tuple consists of Y aggregate values. According to the notation used, R-rowid denotes a row-id reference pointing to the first tuple in the fact table R that has contributed to the generation of the corresponding cube tuple (through this row-id, CURE avoids the storage of dimensional redundancy), Aggr-i ($i \in [1, Y]$) is the i-th aggregate value, and A-rowid represents a row-id reference pointing to a tuple in a relation called AGGREGATES, which is used for the storage of the common aggregate values of CATs. The schema of AGGREGATES is identical to the schema of relation NT.

For example, omitting the details of the construction algorithm, which can be found elsewhere [Morfonios and Ioannidis 2006a], the CURE cube that corresponds to the fact table R (Figure 1a) appears in Figure 26. Note that this cube contains exactly the same information as the uncompressed cube of R (Figure 1b), albeit stored in a much more compact format. In order to uncompress the data of a specific node and restore the original information, we have to follow R-rowid and A-rowid references, and fetch the corresponding tuples from relations R and AGGREGATES, respectively. Then, in the former case, we only need to project every tuple fetched from R on the

attributes included in the node queried, while in the latter case, we first need to follow an additional R-rowid found in every tuple fetched from AGGREGATES before that. For instance, node B (Figure 1b) has three tuples in total, one of each category. To uncompress tuple ⟨4, 160⟩ stored in B-NT, we have to fetch the tuple in R with R-rowid 4 (tuple ⟨4, 5, 1, 70⟩) and project it on B. Combining the resulting dimension value B = 5 with the aggregate value M = 160 stored in B-NT generates tuple ⟨5, 160⟩ (the third tuple of node B in Figure 1b). Similarly, the tuple stored in B-TT indicates that we have to fetch the tuple with R-rowid 3 in R (tuple ⟨2, 3, 1, 60⟩). Since in this case, we restore a TT, we need to reconstruct the aggregate value as well; hence, we project the tuple fetched on both B and M, which generates tuple ⟨3, 60⟩ (the second tuple of node B in Figure 1b). Finally, the tuple stored in B-CAT indicates that we have to fetch the tuple with A-rowid 1 in relation AGGREGATES (tuple ⟨1, 150⟩). The latter specifies that we need to fetch the tuple with R-rowid 1 in R (⟨0, 1, 1, 50⟩) and project it on B. Combining the resulting dimension value B = 1 with the aggregate value M = 150 found in the tuple fetched from relation AGGREGATES finally generates tuple ⟨1, 150⟩ (the first tuple in node B of Figure 1b).

Interestingly, CURE stores a TT only in the most specialized node NS to which it belongs and considers it shared among NS and its ancestors in the cube lattice that have the grouping attributes of NS as a prefix. In Figure 26 for example, any TT stored in node A-TT belongs to node A and, indirectly, to nodes that have A as a prefix, namely AB, AC, and ABC. Note that none of them stores the same TTs physically, since this would be redundant. Therefore, during query answering over a node N, accessing TTs only in the relation N-TT is not enough. Additionally, it is necessary to access TTs in the corresponding TT relations of some of N's descendants. If N includes x grouping attributes, then the total number of TT relations that need to be accessed is x+1. These nodes are N itself and all those whose label is generated by recursively removing the rightmost attribute of N. For example, restoring TTs of node ABC requires accessing relations ABC-TT, AB-TT, A-TT, and Ø-TT in Figure 26.

Finally note that TRS-BUC [Morfonios and Ioannidis 2006b] is a cubing algorithm that can be seen as a simpler version of CURE, in the sense that it does not support many of the key features of CURE. For instance, TRS-BUC cannot handle hierarchies, it does not provide advanced partitioning techniques, and it does not identify all types of redundancy (it identifies only TTs and stores all the remaining tuples in an uncompressed format). Therefore, we only mention it here for completeness, but we do not study it any further.

## 7. PUBLISHED COMPARISONS

In this section, we study the comparisons that have been published so far for the data cube implementation algorithms presented here. As with the presentation of the algorithms, we divide this overview into three parts: computation, selection, and integrated methods.

### 7.1. Computation Methods

The following list presents all published comparisons that concern the computation methods of Section 4. Since these methods construct identical full cubes, the comparison metric is construction time.

—PipeSort and PipeHash [Agarwal et al. 1996] have been compared with NaiveSort and NaiveHash, which are two implementation variations of Algorithm GBLP that use sorting and hashing, respectively. As expected, the results show that PipeSort and PipeHash dramatically improve the efficiency of the corresponding "naive"

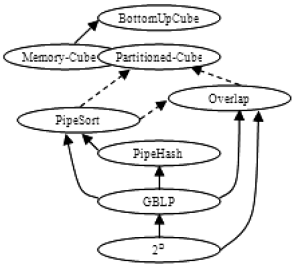


**Fig. 27**. Graph depiction of published results comparing computation methods.

versions. Moreover, the interesting conclusion of this study is that PipeSort outperforms PipeHash, since sorting costs can be better shared among different nodes constructed in a pipelined fashion, than partitioning costs. PipeHash behaves better only when each group-by operation decreases the number of tuples significantly, which occurs when the original fact table is dense. This is attributed to the fact that, in this case, hashing cost in low-level nodes is negligible.

—In the paper where Overlap is introduced [Agarwal et al. 1996], there is a comparison among Overlap, $2^D$, and GBLP. As expected, Overlap outperforms the other algorithms, since it has better scalability properties.

—There has also been a theoretic comparison of the complexity of algorithms Partitioned-Cube, Overlap, and PipeSort [Ross and Srivastava 1997]. According to that, the "winner" is Partitioned-Cube, since it has linear complexity with respect to the number of dimensions. Overlap follows, having quadratic complexity, with PipeSort being the worst, with exponential complexity with respect to the number of dimensions.

—The execution time of Memory-Cube, which is the heart of Partitioned-Cube, has been experimentally compared with that of BUC [Beyer and Ramakrishnan 1999]. As the number of dimensions and their cardinalities increase, BUC outperforms Memory-Cube, since the data cube gets sparser, and thus has more totally redundant tuples, leading to earlier recursion breaks. Memory-Cube is better only when the data distribution is very skewed, creating dense data areas that do not allow BUC to cut recursion early.

These results form a directed graph (Figure 27), where each node represents a computation method and each edge denotes the result of some comparison. Edges are directed towards the method that has been proven better in the average case. Dashed edges denote theoretical comparisons, whereas solid edges denote experimental comparisons. The node that represents Partitioned-Cube intentionally overlaps with Memory-Cube, since the latter is the heart of the former. Fortunately, the graph is acyclic. The "champions" that arise are Partitioned-Cube and BUC.

We consider the comparisons of Figure 27 reliable for the following reasons:

—Algorithms $2^D$ and GBLP are expectedly at the bottom, since they were presented in an introductory paper focusing mainly on definitional concepts rather than on any form of optimization.

—Algorithms PipeSort and PipeHash were proposed in the same paper [Agarwal et al. 1996] by the same group. Hence, in all likelihood, their implementation has been unbiased, making the corresponding experimental results and comparison between the two algorithms rather reliable.

—The comparison among Partitioned-Cube, PipeSort, and Overlap is based on analytical rather than experimental results (as indicated by the dashed edges in Figure 27). Hence, it should be undisputed.

—In an independent thread of our research [Morfonios and Ioannidis 2006b] we implemented Partitioned-Cube/Memory-Cube and BUC and confirmed the trends observed in the experiments of the original papers [Ross and Srivastava 1997; Beyer and Ramakrishnan 1999].

These remarks cover all the edges that appear in the graph of Figure 27, making us confident about its validity.

### 7.2. Selection Methods

The following list presents all published comparisons that concern the selection methods of Section 5:

—Analytic formulas that describe the performance of Algorithms r-Greedy and Inner-Level Greedy [Gupta et al. 1997] show that the latter is preferable to the former for values of r up to 2, where r is the upper bound on the number of nodes and indexes selected in each iteration of r-Greedy, as explained in Section 5.4. Nevertheless, if we are willing to sacrifice time performance for a higher quality result, then 3-Greedy and 4-Greedy are better solutions.

—PBS has been compared with BPUS [Shukla et al. 1998]. Their time complexity is $O(n \times \log n)$ and $O(k \times n^2)$, respectively, where n is the number of lattice nodes and k denotes the number of nodes that are finally materialized. Although PBS is faster, the advantage of BPUS is that it always guarantees that the quality of the result will be near the optimal, while PBS can do so only for the class of SR-hypercube lattices.

—Inverted-Tree Greedy has been experimentally compared with an alternative algorithm that guarantees finding the optimal solution of the selection problem under a maintenance-cost constraint based on the $A^*$ heuristic [Gupta and Mumick 1999]. The latter has cost that is exponential in the number of nodes of the cube lattice and has been proposed by the creators of Inverted-Tree greedy mainly for comparison reasons. The experimental study has shown that Inverted-Tree Greedy almost always finds the same solution as $A^*$—the optimal solution—in much less time.

—PGA has been compared with BPUS [Nadeau and Teorey 2002]. Their time complexity with respect to the number of dimensions D of the fact table is $O(k^2 \times D^2)$ and $O(k^2 \times 2^{2D})$, respectively, where k is the number of nodes finally selected. Clearly, the former is more efficient than the latter, but it does not provide any guarantees for the quality of the solution it finds. The creators of PGA have shown experimentally that the quality of its solution is worse than, but usually close to, the quality of the solution of BPUS.

—The literature [Kotsis 2000] also offers a comparison among algorithms GBLP, Key, and V-Aggregator. The first one is a computation method, the second a selection method, and the third an integrated method. We present the comparison here, since it has been mainly focused on the size of the stored cube. The results show that Key and V-Aggregator lead to a considerable reduction of the data cube size, while their execution time is in general lower than that of GBLP. Furthermore, query response times are only a little worse when redundancy has been removed from the cube by Key or V-Aggregator and simple projections are necessary for the computation of nodes or tuples that have not been stored. Hence, Key and V-Aggregator are the methods of preference.

—The creators of DynaMat [Kotidis and Roussopoulos 1999] have shown experimentally that DynaMat outperforms a system that uses a view-selection algorithm, which statically selects the optimal set of views given a workload, and some space and update-time constraints. Consequently, DynaMat is expected to outperform all heuristic algorithms that statically select suboptimal sets of views as well. They attribute the superiority of DynaMat to its dynamic adaptation to the query workload and the partial materialization of (a large number of) views instead of the complete materialization of a smaller number of views.

—As also mentioned in Section 5.11, experiments with randomized algorithms [Kalnis et al. 2002] have shown that 2PO is the most promising among them, converging faster to a solution of better quality than the rest. Interestingly, the total benefit of 2PO has been found usually very close to, and sometimes even better than, the total benefit of BPUS, which is a systematic algorithm of higher complexity.

Unfortunately, the comparisons among selection methods that exist in the literature are few and sparse. Therefore, we do not provide for them a graph similar to that in Figure 27.

### 7.3. Integrated Methods

The following list presents all published comparisons that concern the integrated methods of Section 6:

—Algorithms MinCube, BU-BST and RBU-BST that construct a so-called BST-Condensed Cube have been experimentally compared with BUC [Wang et al. 2002]. All three of them result in a considerable reduction of the necessary storage space. MinCube produces the smallest result, since it constructs a Minimal BST-Condensed Cube. RBU-BST comes very close to that, while BU-BST comes third. Size reduction increases when the number of dimensions grows but decreases in the presence of skew. Concerning execution time, BU-BST is the winner, RBU-BST comes second with almost double execution time, and MinCube comes last, having exponential complexity in the number of dimensions. All three algorithms, however, outperform BUC, since they write a smaller output to the disk.

—QC-DFS [Lakshmanan et al. 2002] has been experimentally compared with MinCube. As expected, QC-DFS produces smaller cubes than MinCube, since QC-DFS identifies more types of redundancy. Smaller output costs benefit its execution time as well, making it terminate faster than MinCube.

—CURE [Morfonios and Ioannidis 2006a] has been experimentally compared with BUC and BU-BST. CURE identifies all types of redundancy; hence it produces cubes in a more compressed format than the other two. Smaller output costs combined with its efficient partitioning scheme benefit its execution time as well. Consequently, it has been found faster than BUC and BU-BST.

For the same reasons described in Section 7.2, we do not provide a graph similar to that in Figure 27 here either.

## 8. PARAMETER SPACE OF THE DATA CUBE IMPLEMENTATION PROBLEM

In the previous sections, we presented the most important data cube implementation methods published up to now, focusing on ROLAP technology. These methods exhibit several commonalities and differences, and each has its advantages and disadvantages. Their study inspired us to attempt to place them all in a space formed by independent dimensions that correspond to the important characteristics of these algorithms. Such

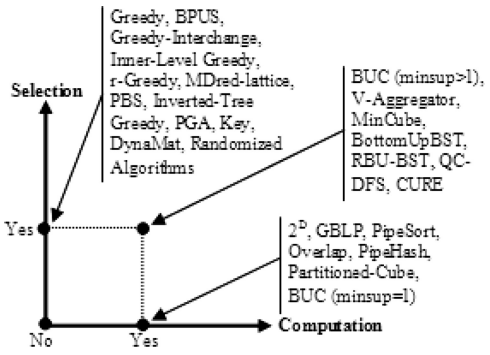


**Fig. 28**. First classification of the methods.

placement would allow exploration of their properties and identification of features that differentiate one algorithm from another, making some of them more effective. To the best of our knowledge, this is the first attempt to draw the parameter space of the data cube implementation problem.

### 8.1. General Description

Figure 28 shows the two orthogonal problems of data cube computation and selection and a gross classification of approaches related to each one of them. The possible points that arise in the space formed can be described by the Cartesian product {No, Yes}×{No, Yes}. Value *No* in the dimension of Computation (respectively, Selection) denotes that the corresponding algorithm is not related to the computation (respectively, selection) of the cube, while value *Yes* denotes that it is. Thus, computation methods (Section 4) are associated with point ⟨Yes, No⟩, selection methods (Section 5) are associated with ⟨No, Yes⟩, and integrated methods (Section 6) with ⟨Yes, Yes⟩. Clearly, no algorithm corresponds to ⟨No, No⟩. In the following subsections, we investigate each point shown in Figure 28, further analyzing it into more detailed orthogonal dimensions.

### 8.2. Dimensions of Computation Methods

*8.2.1. Analysis.* The computation methods presented in Section 4 follow the same general strategy, which consists of three steps:

—Transform the original lattice (e.g., Figure 4) into a tree and choose a plan to traverse it.

—Partition the tuples of the original fact table into segments that fit in memory and transfer them from disk, one by one, for further processing.

—Aggregate the data that is brought in using some in-memory algorithm.

These three steps are mutually independent: Traversal of the cube lattice is not related to the strategy used to partition the original fact table and transfer the data. Similarly, neither of these affects the in-memory processing algorithm used for aggregation and vice versa. Such independence leads to Figure 29, which shows three orthogonal dimensions of the data cube computation problem along with their possible values. Among them, the value *None* appears at the axes' origin, to denote methods that do not deal with the corresponding issue. Note that, there is no specific intrinsic ordering among values in the same dimension, but using one as in Figure 29 makes visualization easier. The number of all possible points that exist in the space formed is $5\times3\times3 = 45$. Further details for each dimension are given below.

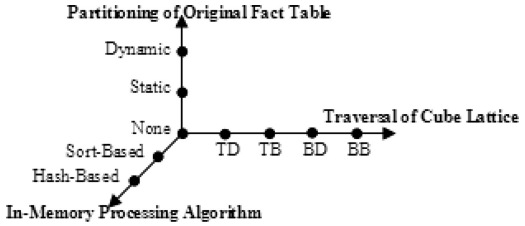


**Fig. 29**. The orthogonal dimensions of the data cube computation problem.

**Traversal of Cube Lattice.** This dimension shows the order in which an algorithm "visits" the lattice nodes and computes the corresponding aggregations. This order includes both the traversal direction (top-down/bottom-up) and the manner the lattice levels are visited (depth-first/breadth-first). Figure 29 shows the possible values of this dimension in a two-letter abbreviation format:

—A first letter "T" denotes the **T**op-down traversal direction, while "B" stands for **B**ottom-up.

—A second letter "D" denotes the **D**epth-first manner of visiting the lattice levels, while "B" stands for **B**readth-first.

**Partitioning of Original Fact Table.** This dimension shows the way in which an algorithm partitions the data into segments that fit in main memory, in order to transfer them from disk for further processing. As mentioned earlier, in many cases, the original fact table whose cube is computed does not fit in memory. In addition, it is possible that even nodes in interim levels of the lattice do not fit in memory. Computation methods are obliged to take this fact into account; otherwise they are vulnerable to thrashing. The possible values identified for this dimension (Figure 29) concern the way in which partitioning is applied:

—*Static* partitioning indicates that tuples are segmented offline, at the beginning of computation, based on some static algorithm, and cannot be reorganized, even if some of them do not fit in main memory. In this case, the corresponding algorithms suffer from thrashing.

—*Dynamic* partitioning indicates that data is recursively partitioned until each generated segment truly fits in memory.

**In-Memory Processing Algorithm.** This dimension shows the strategy that an algorithm uses, to compute aggregations in memory and produce the final result. We have identified two possible values for this dimension (Figure 29):

—Value *Sort-Based* denotes that the in-memory algorithm used for aggregations is based on sorting.

—Value *Hash-Based* denotes that the in-memory algorithm is based on hashing.

*8.2.2. Method Placement.* Based on the previous analysis, we can place all computation algorithms (point ⟨Yes, No⟩ of Figure 28) within the space of orthogonal parameters analyzed here. Figure 30 shows the result of such placement, which we further explain and justify.

—**Algorithm $2^D$** (Section 4.1) has been proposed as an example for the introduction of the data cube structure and has not been designed for practical use. So, it

| Partitioning | None | | | Static | | | Dynamic | | |
|---|---|---|---|---|---|---|---|---|---|
| Memory Alg. / Traversal | None | Sort-Based | Hash-Based | None | Sort-Based | Hash-Based | None | Sort-Based | Hash-Based |
| None | $2^D$ | | | | | | | | |
| TD | GBLP (1) | Memory-Cube | | | PipeSort | | | Partitioned-Cube | |
| TB | GBLP (2) | | | | Overlap | PipeHash | | | |
| BD | | | | | | | | BUC-S [minsup=1] | BUC-H [minsup=1] |
| BB | | | | | | | | A-BUC-S [minsup=1] | A-BUC-H [minsup=1] |

**Fig. 30**. Placement of computation methods.

contains no optimization techniques, it suggests no certain plan for the lattice traversal, it proposes no partitioning scheme, and it does not specify a particular in-memory algorithm. Hence, we place $2^D$ at point ⟨*None, None, None*⟩.

—**Algorithm GBLP** (Section 4.2) is a small improvement over $2^D$, introducing the use of the smallest-parent optimization. Hence, it proposes a top-down traversal direction. It is not associated with a particular manner of visiting the lattice levels however, so we can place it at two possible points, depending on the given implementation: ⟨*TD, None, None*⟩ and ⟨*TB, None, None*⟩. Note that the manner of visiting the lattice levels affects memory requirements: in general, breadth-first scanning imposes greater memory requirements than depth-first. As already mentioned however, GBLP does not focus on memory management, so it makes no specific choice.

—**Algorithm PipeSort** (Section 4.3) transforms the original lattice into a tree and traverses its paths in a depth-first fashion, so it corresponds to value *TD* in dimension "Traversal of Cube Lattice." For the aggregation process, it sorts the first node of each path and propagates the results to lower levels using pipelining. Thus, the value in dimension "In-Memory Processing Algorithm" is *Sort-Based*. Finally, for the pipelined computation, the algorithm statically allocates a single memory page for each node that belongs to the currently processed path. Hence, we place PipeSort at point ⟨*TD, Static, Sort-Based*⟩.

—**Algorithm Overlap** (Section 4.4) transforms the original lattice into a tree by choosing for each node its lattice parent with which it shares the longest common prefix. Then, it traverses the tree in a top-down and breadth-first fashion, and computes each node from its tree parent giving priority to nodes that have more grouping attributes and lower memory requirements. The partitioning scheme assumes that data is uniformly distributed and is therefore static. Finally, aggregate evaluation is based on sorting. From this description, it follows that Overlap is at point ⟨*TB, Static, Sort-Based*⟩.

—**Algorithm PipeHash** (Section 4.5) transforms the original lattice into a tree, using the smallest-parent optimization. Then, it iteratively selects the largest subtree that is not yet processed, for which it estimates that, assuming uniform data distribution, all partitions of its root fit in memory. It partitions the root of the selected subtree and, for each partition, evaluates the necessary aggregations, visiting nodes in a breadth-first fashion and using a hash-based in-memory algorithm. Hence, we place PipeHash at point ⟨*TB, Static, Hash-Based*⟩.

—Viewed as a separate computation method, **Algorithm Memory-Cube** (Section 4.6) does not apply a partitioning algorithm, because it assumes that its entire input fits in memory. In general, as noted in Section 4.6 and also described in the literature [Ross and Srivastava 1997], Memory-Cube functions in a way similar to PipeSort: it creates paths traversing the lattice in a top-down/depth-first fashion and uses sorting,

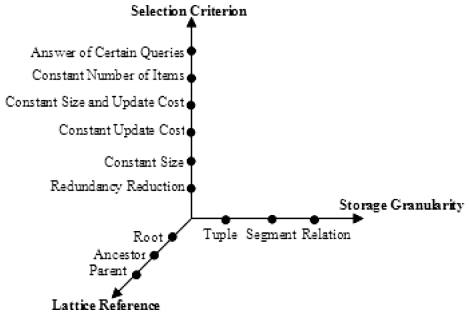


**Fig. 31**. The orthogonal dimensions of the selection problem.

to make adjacent in main memory tuples that aggregate. Thus, we place it at point ⟨*TD, None, Sort-Based*⟩.

—**Algorithm Partitioned-Cube** (Section 4.6) uses Memory-Cube for the aggregation process. It also recursively applies dynamic partitioning on its input, until all generated partitions fit in memory. Then, it simply calls Memory-Cube, which performs all aggregations. Partitioned-Cube collects the partial results of all recursive calls and returns their union. Hence, we place it at point ⟨*TD, Dynamic, Sort-Based*⟩.

—**Algorithm BUC** (Section 4.7) appears four times in Figure 30 in the following variations: BUC-S [minsup=1], BUC-H [minsup=1], A-BUC-S [minsup=1] and A-BUC-H [minsup=1]. Abbreviation A-BUC stands for Apriori-BUC, which is a variation that traverses the lattice in a bottom-up/breadth-first fashion instead of the bottom-up/depth-first fashion of the standard version of BUC. Versions labelled S are sort-based, while those labelled H are hash-based. In practice however, only the former have been implemented. All four variations use recursive dynamic partitioning until all generated partitions fit in memory. Note that, in this subsection, we study computation methods; hence, minsup is set to 1, referring to versions of BUC that compute an entire cube.

### 8.3. Dimensions of Selection Methods

*8.3.1. Analysis.* Using similar reasoning, we identify the following orthogonal dimensions that model selection of a data cube subset for materialization:

—Granularity of the items selected for storage.

—Criterion, based on which, selection is made.

—Node used as a reference for nonmaterialized data. This node corresponds to the relation where all information necessary for on-the-fly computation of nonprecomputed data is stored.

These three parameters are mutually independent. Figure 31 shows these three orthogonal dimensions of the selection problem along with their possible values. Note that, unlike Figure 29, we do not include value *None* among them, since it is not used in practice. Furthermore, there is again no intrinsic ordering of values in the same dimension. The one used in the figure serves only for easy visualization. The number of all possible points that exist in the space formed is $3 \times 6 \times 3 = 54$. Further details for each dimension are given below.

**Storage Granularity.** This dimension shows the granularity of items that are selected for computation and storage (or nonstorage). Along this dimension, we have identified

the following possible values (Figure 31):

—Value *Tuple* denotes that the selection criterion is applied on each individual record.
—Value *Relation* denotes that entire tables/nodes are selected for storage.
—Value *Segment* denotes that the selected items are sets of tuples that belong to lattice nodes. *Segment* is an intermediate, more general value, which degenerates to *Tuple* or *Relation* at the boundaries.

**Selection Criterion.** The "Selection Criterion" dimension represents the objective of the selection process. An algorithm applies the selection criterion to decide whether the item currently processed (tuple, segment, or relation) needs to be stored or can be ignored. The values that we have identified for this dimension are the following (Figure 31):

—*Redundancy Reduction* identifies redundant parts of the cube that can be easily derived from data already stored, which are then ignored.
—*Constant Size* places a constraint on the storage space available, and ignores parts of the cube that do not fit.
—*Constant Update Cost* sets a constraint on the time available for incrementally updating the items (tuples, segments, or relations) stored in the cube.
—*Constant Size and Update Cost* expresses that the selected items must fit in some given storage space and also that they are updateable during a given update time window.
—*Constant Number of Items* sets an upper bound on the number of items that can be precomputed and stored.
—*Answer of Certain Queries* is based on an expected query workload and identifies parts of the cube that are unlikely to be needed for those queries and can therefore be ignored.

**Lattice Reference.** For each part of the cube that is not stored, some reference must exist to the node where the information necessary for its on-the-fly computation is stored. The possible values for this dimension are the following (Figure 31):

—*Root*, when the data required is in the lattice root.
—*Parent*, when the data required for parts of a node resides in one of its parents.
—*Ancestor*, when data required for parts of a node resides in some intermediate level between the root and the node's parents. Again, this value is general and degenerates to *Root* or *Parent* at the boundaries.

Note that, in this article, we assume that the set of dimensions in the fact table is a superkey [Codd 1970]; hence, the fact table includes no duplicates and as a consequence, it contains the same information as the lattice root. Under this assumption, we have simplified our model by using the value *Root* even for methods that use references to the fact table. Our model can easily adapt to cases for which this assumption does not hold. Such adaptation is straightforward and is not further discussed.

*8.3.2. Method Placement.* Based on the previous analysis, we can place all selection algorithms proposed in the literature (point ⟨No, Yes⟩ of Figure 28) within the space of orthogonal parameters analyzed above. Figure 32 shows the results of such placement, which is further explained and justified below.

—**Algorithm Greedy** (Section 5.1) selects entire lattice nodes for computation and storage, until a certain predefined number of nodes has been selected. The root of

| Granularity / Reference / Criterion | Tuple | | | Segment | | | Relation | | |
|---|---|---|---|---|---|---|---|---|---|
| | Root | Ancestor | Parent | Root | Ancestor | Parent | Root | Ancestor | Parent |
| Red/ncy Reduction | | | | | | | Key | | |
| Constant Size | | | | | | | PBS | BPUS<br>Greedy-Interchange<br>Inner-Level Greedy<br>r-Greedy<br>PGA | |
| Constant Update Cost | | | | | | | | Inverted-Tree Greedy | |
| Constant Size and Update Cost | | | | | DynaMat | | | Randomized Algorithms | |
| Constant Number of Items | | | | | | | | Greedy | |
| Answer of Certain Queries | | | | | | | | MDred-lattice | |

**Fig. 32**. Placement of selection methods.

the lattice always belongs to the set of materialized nodes, since the information it contains cannot be derived from its descendants. If the result of a query belongs to a nonmaterialized node, it can be computed on the fly using its smallest sized materialized ancestor. In the best case, this ancestor is a parent of the missing node, while in the worst, it is the root. Hence, we place Greedy at point ⟨*Relation, Constant Number of Items, Ancestor*⟩.

—**Algorithm BPUS** (Section 5.2) is a variation of Greedy that differentiates only in the selection criterion. So, instead of imposing a limit on the number of materialized nodes, there is an upper bound on storage space that can be occupied. Thus, we can place BPUS at point ⟨*Relation, Constant Size, Ancestor*⟩.

—**Algorithms Greedy-Interchange**, **Inner-Level Greedy**, **r-Greedy**, and **PGA** (Sections 5.3, 5.4, and 5.8) are variations of BPUS; consequently, we place them at the same point, namely ⟨*Relation, Constant Size, Ancestor*⟩.

—**Algorithm MDred-lattice** (Section 5.5) selects entire lattice nodes as well. Nodes are chosen based on knowledge of certain queries that the system has to respond to. As in the case of the Greedy algorithm, if a query needs a node that has not been stored, then the necessary data can be computed on the fly using its smallest sized materialized ancestor. Again, in the best case, this ancestor is a parent of the missing node, while in the worst, it is the root. Hence, we place MDred-lattice at point ⟨*Relation, Answer of Certain Queries, Ancestor*⟩.

—**Algorithm PBS** (Section 5.6) also selects entire nodes satisfying a constraint on the total storage space that they can occupy. It gives priority to smaller nodes, however, so it proceeds in a bottom-up fashion, forming a frontier that separates the nodes selected from those not selected. Hence, if a node has not been stored, we can easily conclude that none of its ancestors has been stored either, apart from the root, which is always stored. Hence, we place PBS at point ⟨*Relation, Constant Size, Root*⟩.

—**Algorithm Inverted-Tree Greedy** (Section 5.7) is the first member of the family of greedy algorithms that introduced the view-selection problem under a maintenance-cost constraint. Therefore, we place it at point ⟨*Relation, Constant Update Cost, Ancestor*⟩.

—**Algorithm Key** (Section 5.9) identifies and avoids storing totally-redundant nodes: nodes that include a so-called observational key in their grouping attributes. Key is similar to PBS in that, if a node is totally redundant, then all of its ancestors up to the lattice root are totally redundant as well and need not be saved. Hence, we place Key at point ⟨*Relation, Redundancy Reduction, Root*⟩.

| Partitioning | None | | | Static | | | Dynamic | | |
|---|---|---|---|---|---|---|---|---|---|
| Memory Alg. / Traversal | None | Sort-Based | Hash-Based | None | Sort-Based | Hash-Based | None | Sort-Based | Hash-Based |
| None | | | | | | | | | |
| TD | | | V-Aggregator (1) | | | | | | |
| TB | | | V-Aggregator (2) | | | | | | |
| BD | | | | | | | | BUC-S [minsup>1]<br>(R)BU-BST<br>QC-DFS<br>CURE | BUC-H [minsup>1] |
| BB | | | | | MinCube | | | A-BUC-S [minsup>1] | A-BUC-H [minsup>1] |

**Fig. 33**. Placement of integrated methods (1).

—**Algorithm DynaMat** (Section 5.10) selects particular segments of relations under the constraint that they fit in a given storage space and are updateable during a given update window. In this sense, we place DynaMat at point ⟨*Segment, Constant Size and Update Cost, Ancestor*⟩.

—**The Algorithms of Randomized Search** (Section 5.11) select entire relations under a constant size and update cost constraint. Therefore, we place them at point ⟨*Relation, Constant Size and Update Cost, Ancestor*⟩.

### 8.4. Dimensions of Integrated Methods

*8.4.1. Analysis.* After studying the orthogonal dimensions that define the parameter space of computation and selection problems separately, we can combine the results, in order to identify the independent dimensions that describe the space where integrated methods "live." This space is defined by six orthogonal dimensions, namely "Traversal of Cube Lattice," "Partitioning of Original Fact Table," "In-Memory Processing Algorithm," "Storage Granularity," "Selection Criterion," and "Lattice Reference." The number of potentially different combinations of algorithm characteristics in the space is equal to the product of the corresponding numbers in Sections 8.2.1 and 8.3.1, namely $45 \times 54 = 2430$.

These six dimensions cannot be visualized easily, but we can try to understand their meaning. They are six degrees of freedom that define an entire family of methods. Among all these possible methods, only a few have been studied in the literature. In the following subsection, we place all integrated methods proposed so far at the appropriate points within the 6-dimensional space.

*8.4.2. Method Placement.* We have split the 6-dimensional information into Figures 33 and 34, which are complementary and show the method placement. These figures are similar to Figures 30 and 32, respectively, and must be read together, since each method appears in both. Our reasoning behind this method placement is presented below.

—**Algorithm BUC** (Section 6.1) is again placed at four points in Figure 33, which coincide with those in Figure 30, for the same reasons explained in Section 8.2.2. This time, however, parameter minsup is greater than 1, which implies that BUC performs some form of selection as well. It ignores entire segments of tuples that do not satisfy some minimum support criterion. Recall that if a segment does not qualify during the computation of a node, it will also not qualify in all its ancestors, since support decreases in a bottom-up direction. Hence, all nonstored information can only be recovered using the fact table. Algorithm BUC with minsup$>$1 materializes a subcube (the so-called Iceberg-Cube), appropriate for answering certain queries that

| Granularity | Tuple | | | Segment | | | Relation | | |
|---|---|---|---|---|---|---|---|---|---|
| Criterion \ Reference | Root | Ancestor | Parent | Root | Ancestor | Parent | Root | Ancestor | Parent |
| Redncy Reduction | MinCube / (R)BU-BST / CURE | QC-DFS | V-Aggregator (1 +2) | | | | | | |
| Constant Size | | | | | | | | | |
| Constant Update Cost | | | | | | | | | |
| Constant Size and Update Cost | | | | | | | | | |
| Constant Number of Items | | | | | | | | | |
| Answer of Certain Queries | | | | (A-) BUC-(S/H) [minsup>1] | | | | | |

**Fig. 34**. Placement of integrated methods (2).

include a HAVING clause. Thus, we place all four versions of BUC at point ⟨*Segment, Answer of Certain Queries, Root*⟩.

—**Algorithm V-Aggregator** (Section 6.2) traverses the lattice in a top-down fashion using hashing for in-memory processing. In its original presentation, it assumes that data fits in main memory, thus it does not propose a partitioning scheme. During the computation of a node, it does not store partially redundant tuples, which are tuples that appear also in the node's parent. Hence, in the computation subspace, we place it at two possible points that differ only in the way of visiting the lattice levels, depending on the specific implementation: ⟨*TD, None, Hash-Based*⟩ and ⟨*TB, None, Hash-Based*⟩. Moreover, in the selection subspace, we place it at point ⟨*Tuple, Redundancy Reduction, Parent*⟩.

—**Algorithm MinCube** (Section 6.3) traverses the lattice in a bottom-up/breadth-first fashion, ignoring totally redundant tuples (or BSTs in its own terminology). These are the tuples that have not been subject to aggregation and have the same measurement value at all levels up to the root. The in-memory processing algorithm is sort-based. After sorting the data according to the grouping attributes of the node at hand, the algorithm scans all tuples that have not yet been identified as totally redundant one-by-one, and applies the aggregation. For this purpose, it is enough to statically allocate one memory page. Hence, we place MinCube at point ⟨*BB, Static, Sort-Based*⟩ in the computation subspace and at point ⟨*Tuple, Redundancy Reduction, Root*⟩ in the selection subspace.

—**Algorithm BU-BST** (Section 6.4) is similar to BUC-S (i.e., the sort-based version of BUC) in the computation subspace and to MinCube in the selection subspace. Thus, we place it at the corresponding points of Figures 33 and 34, respectively. Similarly, we place **RBU-BST**, which is a simple variation of BU-BST, at the same points as well.

—**Algorithm QC-DFS** (Section 6.5) is an extension of BU-BST that identifies more types of redundant tuples: it identifies cube tuples that are produced by aggregation of the same set of tuples in the original fact table and assigns them into an equivalence class of cells with identical aggregate values. Then it stores the aforementioned equivalence class only once, sharing the same tuple t among a set of nodes SN indicated by a lower-bound expression (see Section 6.5 for more details). Tuple t is physically stored in the most detailed node of SN that is an ancestor of all the remaining nodes of SN. Therefore, we place QC-DFS at point ⟨*BD, Dynamic, Sort-Based*⟩ in the computation subspace and at point ⟨*Tuple, Redundancy Reduction, Ancestor*⟩ in the selection subspace.

—**Algorithm CURE** (Section 6.6) is another integrated method based on BUC-S that removes all types of redundancy, substituting redundant information with row-id references pointing to the fact table (which contains the same information with the root of the lattice if we assume that it has no duplicates). Consequently, we place CURE at point ⟨*BD, Dynamic, Sort-Based*⟩ in the computation subspace and at point ⟨*Tuple, Redundancy Reduction, Root*⟩ in the selection subspace.

### 8.5. Interesting Properties

After defining the parameter space of the data cube implementation problem and identifying its orthogonal dimensions, we can go one step further and search for interesting properties within different combinations of values. Interesting properties are those that appear to have a positive effect on the efficiency of the data cube implementation algorithm. The benefit of their detection is twofold: On the one hand, it can provide an explanation for why some existing algorithms outperform the others. On the other hand, it can lead to the development of new methods that combine appropriate properties to achieve better results. In the rest of this subsection, we first present some general observations, and then identify groups of interesting properties of various combinations of dimensions.

**General Observations.** Our first remark is that, in general, integrated methods provide more comprehensive solutions than separate computation and selection methods. Their advantage is that they draw more effective plans, since they face the data cube implementation problem as a whole. Selection is incorporated into computation, making it possible to avoid unnecessary actions sooner and leading to more optimizations.

Furthermore, we note that, in the computation subspace, value *None* should be avoided. An algorithm is considered comprehensive when it has nontrivial values along all dimensions. Otherwise, it does not take into account all problem parameters and can never be optimal. Value *None* is seen only in algorithms used for the introduction of the data cube structure and in methods assuming that the original fact table and all interim data structures fit in main memory. Such algorithms do not represent solutions that are practically applicable.

**Traversal of Cube Lattice.** Top-down lattice traversal is better combined with selection methods that do not store partially redundant items (tuples, segments, or relations), since this kind of redundancy occurs only in parent-child node pairs and nothing can be deduced for ancestor nodes in levels further up. On the contrary, bottom-up traversal is better combined with selection methods that do not store totally redundant items, since this kind of redundancy occurs between a node and all of its ancestors, all the way up to the lattice root. Early detection of such redundancy leads to earlier pruning and decreases complexity of higher levels. Visiting nodes in a bottom-up and breadth-first fashion makes it possible to identify and ignore all redundant items at the cost of a longer cube computation time. On the contrary, visiting the nodes in a bottom-up and depth-first fashion accelerates the corresponding algorithms, but some redundancy cannot be identified early, and redundant items are stored in the result.

Furthermore, combining values *TD* for "Traversal of Cube Lattice" and *Sort-Based* for "In-Memory Processing Algorithm" is particularly effective, especially when dynamic partitioning is also applied. The reason is that sorting can be pipelined with the computation of entire node paths. Each node needs only a single memory page for this, thus leading to small memory requirements. Dynamic partitioning is important

when the original data does not fit in main memory and external sorting and merging is unavoidable.

**Partitioning of Original Fact Table.** Along the dimension "Partitioning of Original Fact Table," value *Dynamic* seems to result in more robust and effective algorithms. These algorithms adapt better to sparse cubes and skewed data. The assumption that data is uniformly distributed is rarely valid in practice. Skew can result in partitions that are too big to fit in memory. Static partitioning does not take this into consideration, making algorithms prone to thrashing. On the other hand, dynamic partitioning saves on I/O costs and makes a more effective use of system resources.

**In-Memory Processing Algorithm.** It is also worth mentioning that, in general, sort-based in-memory processing algorithms impose smaller memory requirements than their hash-based counterparts, since they make use of no additional data structure during cube computation. The original fact table itself is usually already very large and additional memory requirements for keeping hash tables makes the situation even harder. Furthermore, it has been shown in the literature [Agarwal et al. 1996] that sorting costs are better shared among different nodes computed in a pipelined fashion than hashing costs. Hence, sort-based methods seem to be a better choice for cube implementation.

**Selection Criterion.** Among selection methods, only those having value *Redundancy Reduction* along dimension "Selection Criterion" result in a fully precomputed cube. At query time, if some required data is not stored in a specific node, it can be retrieved on the fly with a simple projection, using data in the node pointed to by the lattice reference. On the contrary, all other methods need additional on-the-fly aggregations of data stored in the reference node. Avoiding such aggregation leads to better query response times.

**Storage Granularity and Lattice Reference.** Finally, we must note that values *Segment* and *Ancestor* along dimensions "Storage Granularity" and "Lattice Reference", respectively, provide greater freedom to selection algorithms and tend to lead to superior condensation rates.

### 8.6. Further Remarks

In this subsection, we provide some additional comments in order to highlight some subtle issues and avoid possible misunderstandings.

First, we emphasize that methods with the same value in one dimension of the parameter space that we have identified do not necessarily function in exactly the same way with respect to that dimension. For instance, BUC uses three different sorting functions (CountingSort, QuickSort, and InsertionSort), based on the volume of data. Hence, even if it has value *Sort-Based* in dimension "In-Memory Processing Algorithm," it performs sorting operations in a different way from Partitioned-Cube, which has the same value in this dimension, but uses only QuickSort. The difference is in implementation details and is not related to the general philosophy of the algorithms. Method placement in the parameter space is based only on general properties and not on such particularities. This also explains how multiple methods can coexist at a single point of the parameter space. They may share general concepts and techniques, but each method has its own identity and differs in several implementation details.

Second, the values identified along each dimension do not necessarily exhaust the problem parameter space. They characterize the algorithms found in the literature and possible variations that we have conceived. We cannot preclude, however, the development of pioneer techniques in the future, which will correspond to new values

in some dimensions. Our current space mapping is flexible enough to adapt to such novelties.

Third, we have seen that some methods can be placed at multiple points in the space. Such placement can be attributed to two reasons:

(1) Some issues have been left unspecified by the originators of the algorithm, leaving different options open. We have already taken such cases into account during placement of the algorithms at the appropriate points, and we have placed some methods at multiple points as well. For example, in Figure 30, we have placed BUC at four points, depending on the given implementation.

(2) There are algorithms that combine the advantages of multiple techniques. Some such combinations have already been proposed in the literature:

—BUC can be effectively combined with PBS [Beyer and Ramakrishnan 1999]. Neither segments that do not satisfy certain minimum support conditions nor entire nodes that exceed certain size bounds would be stored, condensing the resulting cube more than each algorithm alone would have been able to do. The two algorithms are fully compatible, since they both traverse the lattice in a bottom-up fashion.

—Algorithm Key has been combined with V-Aggregator [Kotsis 2000] for the construction of a more condensed cube.

—In the presentation of most selection algorithms, for the sake of simplicity, we have assumed that queries are uniformly distributed over all nodes of the cube. Nevertheless, associating every possible query with some probability measure in its benefit expressions is straightforward and has appeared in practice. For instance, there is a variation of PBS, called PBS-U, that takes into account not only the estimated size of each node $v_i$, but also the probability $p_i$ that an incoming query requires $v_i$. It chooses nodes based not on their absolute size $|v_i|$, but on their weighted size $|v_i|/p_i$. Thus, it uses a double "Selection Criterion": *Constant Size* and *Answer of Certain Queries*.

The discovery of such effective combinations is very interesting for further research, but exceeds the scope of this article and is left for future work.

## 9. CONCLUSIONS AND FUTURE WORK

In this article, we have presented a comprehensive overview of existing algorithms for efficient data cube implementation in a ROLAP environment. Furthermore, we have studied the parameter space of the problem and identified six orthogonal parameters/dimensions: "Traversal of Cube Lattice," "Partitioning of Original Fact Table," "In-Memory Processing Algorithm," "Storage Granularity," "Selection Criterion," and "Lattice Reference." We have placed the existing algorithms at the appropriate points within the problem space, based on their properties. Interestingly, the algorithms form clusters, whose study has led to the identification of particularly effective values for the space parameters.

We believe that such clustering can serve as a starting point and inspiration to propose new algorithms that incorporate the above findings and implement the data cube more efficiently. This is part of our current and future work.

## REFERENCES

ACHARYA, S., GIBBONS, P. B., AND POOSALA, V. 2000. Congressional samples for approximate answering of group-by queries. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 487–498.

AGARWAL, S., AGRAWAL, R., DESHPANDE, P., GUPTA, A., NAUGHTON, J. F., RAMAKRISHNAN, R., AND SARAWAGI, S. 1996. On the computation of multidimensional aggregates. In *Proceedings of Very Large Data Bases (VLDB)*. 506–521.

AGRAWAL, R., MEHTA, M., SHAFER, J. C., SRIKANT, R., ARNING, A., AND BOLLINGER, T. 1996. The quest data mining system. In *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*. 244–249.

BARALIS, E., PARABOSCHI, S., AND TENIENTE, E. 1997. Materialized views selection in a multidimensional database. In *Proceedings of Very Large Data Bases (VLDB)*. 156–165.

BARBARÁ, D. AND SULLIVAN, M. 1997. Quasi-cubes: Exploiting approximations in multidimensional databases. *SIGMOD Record 26,* 3, 12–17.

BARBARÁ, D. AND SULLIVAN, M. 1998. A space-efficient way to support approximate multidimensional databases. In *Tech. Rep., ISSE-TR-98-03, George Mason University*.

BEYER, K. S. AND RAMAKRISHNAN, R. 1999. Bottom-up computation of sparse and iceberg cubes. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 359–370.

CHAUDHURI, S. AND DAYAL, U. 1997. An overview of data warehousing and OLAP technology. *SIGMOD Record 26,* 1, 65–74.

CODD, E. F. 1970. A relational model of data for large shared data banks. *Comm. ACM 13,* 6, 377–387.

FENG, Y., AGRAWAL, D., ABBADI, A. E., AND METWALLY, A. 2004. Range cube: Efficient cube computation by exploiting data correlation. In *Proceedings of International Conference on Data Engineering (ICDE)*. 658–670.

FU, L. AND HAMMER, J. 2000. Cubist: A new algorithm for improving the performance of ad-hoc olap queries. In *ACM International Workshop on Data Warehousing and OLAP (DOLAP)*. 72–79.

GIBBONS, P. B. AND MATIAS, Y. 1998. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 331–342.

GRAY, J., BOSWORTH, A., LAYMAN, A., AND PIRAHESH, H. 1996. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of International Conference on Data Engineering (ICDE)*. 152–159.

GUNOPULOS, D., KOLLIOS, G., TSOTRAS, V. J., AND DOMENICONI, C. 2000. Approximating multidimensional aggregate range queries over real attributes. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 463–474.

GUPTA, H. 1997. Selection of views to materialize in a data warehouse. In *Proceedings of International Conference on Database Theory (ICDT)*. 98–112.

GUPTA, H., HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. D. 1997. Index selection for olap. In *Proceedings of International Conference on Data Engineering (ICDE)*. 208–219.

GUPTA, H. AND MUMICK, I. S. 1999. Selection of views to materialize under a maintenance cost constraint. In *Proceedings of International Conference on Database Theory (ICDT)*. 453–470.

HAAS, P. J., NAUGHTON, J. F., SESHADRI, S., AND STOKES, L. 1995. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of Very Large Data Bases (VLDB)*. 311–322.

HAN, J., PEI, J., DONG, G., AND WANG, K. 2001. Efficient computation of iceberg cubes with complex measures. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 1–12.

HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. D. 1996. Implementing data cubes efficiently. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 205–216.

IOANNIDIS, Y. E. AND KANG, Y. C. 1990. Randomized algorithms for optimizing large join queries. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 312–321.

JOHNSON, T. AND SHASHA, D. 1997. Some approaches to index design for cube forest. *IEEE Data Eng. Bull. 20,* 1, 27–35.

KALNIS, P., MAMOULIS, N., AND PAPADIAS, D. 2002. View selection using randomized search. *Data Knowl. Eng. 42,* 1, 89–111.

KARAYANNIDIS, N., SELLIS, T. K., AND KOUVARAS, Y. 2004. Cube file: A file structure for hierarchically clustered olap cubes. In *Proceedings of International Conference on Extending Database Technology (EDBT)*. 621–638.

KOTIDIS, Y. AND ROUSSOPOULOS, N. 1998. An alternative storage organization for rolap aggregate views based on cubetrees. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 249–258.

KOTIDIS, Y. AND ROUSSOPOULOS, N. 1999. Dynamat: A dynamic view management system for data warehouses. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 371–382.

KOTSIS, N. 2000. Multidimensional aggregation in olap systems. PhD thesis. Department of Computer Science, University of Strathclyde.

KOTSIS, N. AND MCGREGOR, D. R. 2000. Elimination of redundant views in multidimensional aggregates. In *Proceedings of Data Warehousing and Knowledge Discovery (DaWaK)*. 146–161.

LAKSHMANAN, L. V. S., PEI, J., AND HAN, J. 2002. Quotient cube: How to summarize the semantics of a data cube. In *Proceedings of Very Large Data Bases (VLDB)*. 778–789.

LAKSHMANAN, L. V. S., PEI, J., AND ZHAO, Y. 2003. Qc-trees: An efficient summary structure for semantic olap. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 64–75.

LAWRENCE, M. 2006. Multiobjective genetic algorithms for materialized view selection in olap data warehouses. In *Proceedings of Genetic and Evolutionary Computation Congress (GECCO)*. 699–706.

LEE, M. AND HAMMER, J. 1999. Speeding up warehouse physical design using a randomized algorithm. In *Proceedings of Design and Management of Data Warehouses (DMDW)*. 3.

LIANG, W., WANG, H., AND ORLOWSKA, M. E. 2001. Materialized view selection under the maintenance time constraint. *Data Knowl. Eng. 37,* 2, 203–216.

MORFONIOS, K. AND IOANNIDIS, Y. 2006a. Cure for cubes: Cubing using a rolap engine. In *Proceedings of Very Large Data Bases (VLDB)*. 379–390.

MORFONIOS, K. AND IOANNIDIS, Y. 2006b. Supporting the data cube lifecycle: The power of ROLAP. *Int. J. VLDB.* http//dx.doi.org/10.1007/s00778-006-0036-8.

NADEAU, T. P. AND TEOREY, T. J. 2002. Achieving scalability in olap materialized view selection. In *Proceedings of ACM International Workshop on Data Warehousing and OLAP (DOLAP)*. 28–34.

POOSALA, V. AND GANTI, V. 1999. Fast approximate answers to aggregate queries on a data cube. In *Proceedings of International Conference on Statistical and Scientific Database Management (SSDBM)*. 24–33.

ROSS, K. A. AND SRIVASTAVA, D. 1997. Fast computation of sparse datacubes. In *Proceedings of Very Large Data Bases (VLDB)*. 116–125.

ROUSSOPOULOS, N., KOTIDIS, Y., AND ROUSSOPOULOS, M. 1997. Cubetree: Organization of and bulk updates on the data cube. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 89–99.

SHANMUGASUNDARAM, J., FAYYAD, U. M., AND BRADLEY, P. S. 1999. Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. In *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*. 223–232.

SHAO, Z., HAN, J., AND XIN, D. 2004. Mm-cubing: Computing iceberg cubes by factorizing the lattice space. In *Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM)*. 213–222.

SHUKLA, A., DESHPANDE, P., AND NAUGHTON, J. F. 1998. Materialized view selection for multidimensional datasets. In *Proceedings of Very Large Data Bases (VLDB)*. 488–499.

SHUKLA, A., DESHPANDE, P., NAUGHTON, J. F., AND RAMASAMY, K. 1996. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proceedings of Very Large Data Bases (VLDB)*. 522–531.

SISMANIS, Y., DELIGIANNAKIS, A., ROUSSOPOULOS, N., AND KOTIDIS, Y. 2002. Dwarf: shrinking the petacube. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 464–475.

SISMANIS, Y. AND ROUSSOPOULOS, N. 2004. The complexity of fully materialized coalesced cubes. In *Proceedings of Very Large Data Bases (VLDB)*. 540–551.

THEODORATOS, D., DALAMAGAS, T., SIMITSIS, A., AND STAVROPOULOS, M. 2001. A randomized approach for the incremental design of an evolving data warehouse. In *Proceedings of the 20th International Conference on Conceptual Modeling (ER'01)*. 325–338.

THEODORATOS, D. AND SELLIS, T. K. 1997. Data warehouse configuration. In *Proceedings of Very Large Data Bases (VLDB)*. 126–135.

UCHIYAMA, H., RUNAPONGSA, K., AND TEOREY, T. J. 1999. A progressive view materialization algorithm. In *Proceedings of ACM International Workshop on Data Warehousing and OLAP (DOLAP)*. 36–41.

VITTER, J. S. AND WANG, M. 1999. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 193–204.

VITTER, J. S., WANG, M., AND IYER, B. R. 1998. Data cube approximation and histograms via wavelets. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)*. 96–104.

WANG, W., FENG, J., LU, H., AND YU, J. X. 2002. Condensed cube: An efficient approach to reducing data cube size. In *Proceedings of International Conference on Data Engineering (ICDE)*. 155–165.

XIN, D., HAN, J., LI, X., AND WAH, B. W. 2003. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proceedings of Very Large Data Bases (VLDB)*. 476–487.

YU, J. X., YAO, X., CHOI, C.-H., AND GOU, G. 2003. Materialized view selection as constrained evolutionary optimization. *IEEE Trans. Syst., Man, Cybernetics, Part C 33,* 4, 458–467.

ZHAO, Y., DESHPANDE, P., AND NAUGHTON, J. F. 1997. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*. 159–170.

Received July 2005; revised January 2007; accepted March 2007