CURE for Cubes: Cubing Using a ROLAP Engine*

Konstantinos Morfonios
Dept. of Informatics and Telecom. Univ. of Athens
kmorfo@di.uoa.gr

Yannis Ioannidis
Dept. of Informatics and Telecom. Univ. of Athens
yannis@di.uoa.gr

ABSTRACT

Data cube construction has been the focus of much research due to its importance in improving efficiency of OLAP. A significant fraction of this work has been on ROLAP techniques, which are based on relational technology. Existing ROLAP cubing solutions mainly focus on "flat" datasets, which do not include hierarchies in their dimensions. Nevertheless, the nature of hierarchies introduces several complications into cube construction, making existing techniques essentially inapplicable in a significant number of real-world applications. In particular, hierarchies raise three main challenges: (a) The number of nodes in a cube lattice increases dramatically and its shape is more involved. These require new forms of lattice traversal for efficient execution. (b) The number of unique values in the higher levels of a dimension hierarchy may be very small; hence, partitioning data into fragments that fit in memory and include all entries of a particular value may often be impossible. This requires new partitioning schemes. (c) The number of tuples that need to be materialized in the final cube increases dramatically. This requires new storage schemes that remove all forms of redundancy for efficient space utilization. In this paper, we propose CURE, a novel ROLAP cubing method that addresses these issues and constructs complete data cubes over very large datasets with arbitrary hierarchies. CURE contributes a novel lattice traversal scheme, an optimized partitioning method, and a suite of relational storage schemes for all forms of redundancy. We demonstrate the effectiveness of CURE through experiments on both real-world and synthetic datasets. Among the experimental results, we distinguish those that have made CURE the first ROLAP technique to complete the construction of the cube of the highest-density dataset in the APB-1 benchmark (12 GB). CURE was in fact quite efficient on this, showing great promise with respect to the potential of the technique overall.

1. INTRODUCTION

Modern data analysis "mines" knowledge from data stored in database systems discovering trends useful for decision making. To achieve this, analysts pose complex queries that extensively use aggregation in order to group together "similarly behaving tuples". The response time of such queries over extremely large fact tables in modern data warehouses can be prohibitive. This inspired Gray et al. [6] to propose the pre-computation of the data cube, which is a data structure that consists of the results of group-by aggregate queries on all possible combinations of the dimension-attributes over a fact table in a data warehouse.

A common representation of the data cube that captures the computational dependencies among different group-by queries is the cube lattice [9]. Figure 1 illustrates the cube lattice of a fact table R with three dimensions (A, B, and C). Every node in the cube lattice represents a group-by query and is labeled with its grouping attributes, which consist of the subset of dimensions that participate in the group-by clause of the corresponding query. If we denote the number of dimensions of a fact table with D, then the number of all cube lattice nodes is 2^D. Hence, a naive implementation method that computes each node separately and stores the result has exponential time and space complexity.

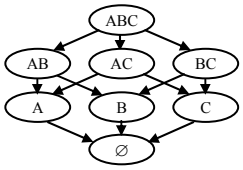


Figure 1. Example of a cube lattice

To overcome this problem, implementation of the complete data cube has been studied using various data structures to construct and store the cube. On one hand, ROLAP and MOLAP methods use materialized views and multidimensional arrays, respectively, focusing mainly on the efficient sharing of computational costs (like sorting or hashing) during cube construction. On the other hand, more recent approaches exploit specialized tree-like data structures in order to compute and store cubes more efficiently. In this paper, we focus on ROLAP methods and ignore the other categories for the following reasons: (a) MOLAP methods are poor performers when data is sparse, which is the case in most real-life applications. Although challenged by some, this has been observed by many researchers [2, 18]. (b) Complex tree-like data structures appear to have superior performance for cube construction and storage, but are currently not supported by any widely used product, hence, requiring nontrivial implementation effort.

ROLAP methods appear to strike the right balance on several fronts. They are based on materialized views and can be incorporated into any existing relational engine with minimal cost. Moreover, it has been shown that several ROLAP methods behave well in most kinds of datasets, including sparse ones, and some of them are capable of condensing the final cube by removing redundancy.

Unfortunately, current ROLAP cubing methods have focused mainly on supporting "flat" data, while many real-life applications deal with fact tables with each dimension consisting of several attributes organized hierarchically. For example, a dimension

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

VLDB '06, September 12–15, 2006, Seoul, Korea.

"Region" may contain values at different levels of detail, forming the hierarchy "City" → "Country" → "Continent". Hierarchies offer greater flexibility to analysts, since they describe the data at different granularities and form the basis for common operations, like roll-up and drill-down. On the other hand, hierarchies introduce several complications into cube construction that cannot be handled by straightforward extensions of existing techniques. (a) The number of nodes in a cube lattice increases dramatically and its shape is more involved. These require new forms of lattice traversal for efficient execution. (b) The number of unique values in the higher levels of a dimension hierarchy may be very small; hence, partitioning data into fragments that fit in memory and include all entries of a particular value may often be impossible. This requires new partitioning schemes. (c) The number of tuples that need to be materialized in the final cube increases dramatically. This requires new storage schemes that remove all forms of redundancy for efficient space utilization.

In this paper, we propose CURE (**C**ubing **U**sing a **R**OLAP **E**ngine), a novel ROLAP cubing method that addresses the issues above and constructs complete data cubes over very large datasets with arbitrary hierarchies. We demonstrate the effectiveness of CURE through experiments on both real-world and synthetic datasets. Among the experimental results, we distinguish those that have made CURE the first ROLAP technique to complete the construction of the cube of the highest-density dataset in the APB-1 benchmark (12 GB) [17]. CURE was in fact quite efficient on this, showing great promise with respect to the potential of the technique itself and of ROLAP in general. CURE stretches ROLAP to its limits, for the first time in the face of hierarchies, indicating that it may not be inherently inferior.

CURE contributes a novel lattice traversal scheme, an optimized data partitioning method, and a suite of relational storage schemes for all forms of redundancy. The last two are useful to "flat" datasets as well, but they are mostly necessary in the presence of hierarchies. In more detail:

- **Lattice Traversal with Dimension Hierarchies:** To the best of our knowledge, CURE is essentially the first comprehensive ROLAP solution capable of constructing a complete cube not only at the leaf level of each dimension hierarchy, but also at all higher levels, pre-computing group-by queries at all granularities. To achieve this, CURE uses an efficient way of traversing an extended lattice that includes dimension hierarchy levels (first proposed elsewhere [9]), which enables great cost sharing of sorting operations through pipelining.
- **External Partitioning:** We propose an efficient algorithm for partitioning fact tables that store hierarchical data of any size into memory-fitting segments, while computing a very small subset of the cube using inexpensive additional resources. Exploiting this early-computed data, CURE accelerates the construction of the final cube significantly, making it feasible even when the original fact table is extremely large. Existing techniques, partition data according to values in a single dimension and require that segments of tuples with the same value in this dimension fit in memory. However, as shown in Section 4, this is not always possible in cases that include hierarchies, due to small domain sizes at coarse granularities.
- **Efficient Storage:** Unlike previous ROLAP methods that rely only on avoiding redundant-tuple storage for cube size reduction, we further study alternative schemes for storing non-redundant data efficiently as well. To the best of our knowledge, CURE is the only ROLAP method that condenses the cube both by rejecting all kinds of redundancy and by further exploiting appropriate data representations.

The rest of this paper is organized as follows: After summarizing related work in Section 2, in Sections 3, 4, and 5, we study the problem of handling hierarchies and revisit external partitioning and efficient storage, respectively, under the new perspective. In Section 6, we combine everything and present CURE in pseudo-code. In Section 7, we describe the results of our experimental evaluation and finally, we conclude in Section 8.

2. RELATED WORK

Data cube construction has been the focus of much research due to its importance in improving the performance of OLAP tools. After Gray et al. [6] proposed the data cube structure, a plethora of papers has been published in this area.

There are several ROLAP cubing methods proposed so far [1, 2, 6, 12, 13, 15, 18, 19, 24], which are well-documented in the existing literature and their detailed description exceeds our purpose. BUC [2] is the most influential method in the ROLAP context attributing its success to a very efficient execution plan that enables sharing sorting costs during construction of different nodes. Both BU-BST [24] and QC-Tables [13] are BUC-based, i.e., they use the same execution plan. However, they do not support hierarchies, they have not been tested over very large data sets, and they do not store cube tuples efficiently. CURE is BUC-based as well, while also dealing with all of these problems.

Among ROLAP cubing techniques, only PipeSort and PipeHash [1, 19] have (superficially) discussed supporting hierarchies. Both of them, however, represent rather straightforward and non-scalable solutions, they have already been outperformed by all subsequent ROLAP methods, and neither handles efficient storage. Hence, CURE appears to be the first ROLAP method that studies the problem comprehensively and proposes a practical solution.

Furthermore, to the best of our knowledge, all results published so far for ROLAP cubing assume that the original fact table fits in memory. Disk-based extensions have been discussed rarely [2, 18], but only for "flat" data and without any accompanying performance results. On the contrary, CURE's partitioning is applicable over very large hierarchical data, which is also shown experimentally even in cases that data sizes far exceed memory resources.

With respect to cube size reduction in ROLAP, Key [12], BU-BST [24], and QC-Tables [13] study the effect of removing redundant tuples from the cube. They only focus on what to avoid storing but not on how to store the data finally materialized. Like existing methods, CURE removes all kinds of redundancy but also employs efficient storage schemes that further compress the final result. Orthogonal to the above is the ability of BUC [2] to construct iceberg cubes, i.e., cubes that do not store data produced by aggregation of a small number of tuples. Being BUC-based, CURE is able to construct iceberg cubes as well.

Regarding MOLAP methods, they use multidimensional arrays for cube construction and storage [20, 26] as an alternative to relational materialized views. None of them handles hierarchies or redundancy, however, hence they are considered impractical.

The inability of existing ROLAP and MOLAP methods to solve all aspects of the cubing problem efficiently has given rise to a third category of algorithms that use sophisticated, complex tree-like data structures for cube construction and storage [5, 8, 14, 22, 25]. Among them, Dwarf [22] is the most promising, being able to deal with hierarchies [21] while removing several kinds of redundancy from cube data, which gives it polynomial scaling [23]. QC-Trees [14] have similar redundancy-reduction capabilities as well. As we explained earlier, the methods of this category cannot be directly used currently, since to the best of our knowledge, the complex data structures they exploit are not supported by any widely-used product and their implementation is far from straightforward. Hence, CURE is the only solution that shares common properties with such sophisticated methods, including polynomial storage requirements, while still being ROLAP compatible and, hence, easily put into an existing server. A comparison among CURE, Dwarf and QC-Trees would be interesting, since it would reveal the foundational strengths and weaknesses of the two underlying philosophies, but it exceeds the purpose of this paper and is left for future work. The aim of this paper is to find the best, easy-to-implement (i.e., ROLAP), comprehensive cubing method that is a potentially viable competitor of the elite of the existing methods that exploit more sophisticated (and costly) data structures.

Finally, apart from the methods that construct complete cubes, there are methods that select subsets of nodes for partial construction (e.g., [9]) and others that compute a small number of predefined nodes [4, 16]. Clearly, the functionality of such methods is orthogonal to that of CURE, since selecting *what* to materialize is orthogonal to deciding *how* to materialize it efficiently. Hence, although they are interesting and their combination with CURE seems possible, their study exceeds the scope of this paper.

3. HIERARCHICAL EXECUTION PLANS

Consider a fact table with D dimensions. If we denote the number of levels of the i-th dimension with $\mathcal{L}_i$, the number of all cube nodes is given by the product $\prod_{i=1}^{D}(\mathcal{L}_i+1)$, which is greater than or equal to 2^D (equality holds when all dimensions are flat, i.e. when $\mathcal{L}_i = 1, \forall\ i \in [1, D]$). Assume, for example, that the dimensions of the fact table R (whose lattice appears in Figure 1) are organized in hierarchies as follows: $A_0 \rightarrow A_1 \rightarrow A_2$, $B_0 \rightarrow B_1$, and C_0. The number of nodes in this example is equal to $(3+1)\cdot(2+1)\cdot(1+1) = 24 > 2^3 = 8$. Clearly, finding an efficient execution plan becomes more complex when using hierarchies. Taking the importance of hierarchies into account [10, 11], we propose below an efficient in-memory method for the construction of a hierarchical cube (a cube whose dimensions form hierarchies).

3.1 Comparison of Alternative Plans

Cubing algorithms that compute the cube using only the most detailed level of every dimension are not viable in practice, since then, for common roll-up and drill-down operations, the underlying system must further aggregate materialized aggregates on the fly, which is computationally expensive. Moreover, executing the cubing operation several times, once for every possible combination of the hierarchy levels of the cube's dimensions, is not practical either. Efficiency of all cubing methods depends on their ability to share computational costs among as many cube nodes as possible. Hence, independent construction of different sub-cubes can never be optimal overall.

The arguments above make it clear that the ideal cubing method must construct all nodes of the hierarchical data cube in a pipelined fashion using as few data passes as possible. Hence, a cube lattice that includes hierarchies must be found, together with an efficient way to traverse it and prune it into a tree that shows the order of execution, i.e., creating the so-called execution plan of the corresponding cubing method.

With respect to pruning a lattice into a tree, BUC [2] has been found to be the winner among all ROLAP cubing algorithms that compute complete and flat cubes. Its efficiency is primarily due to the way it traverses the cube lattice, namely bottom-up and depth-first. Such a traversal is ideal for flat cubes and can easily be extended properly to efficiently handle hierarchies as well.

With respect to identifying an appropriate lattice that incorporates hierarchies, there are two main alternatives: A straightforward solution is to consider every level of each dimension as a separate dimension and construct the lattice as if it were flat. In this case, nodes including more than one levels of the same dimension (e.g., A_1A_2 or $A_0B_0B_1$) should be omitted, since they repeat trivial information. The second alternative is the lattice introduced for hierarchies in the context of view selection [9], which natively reflects relationships between different levels of the same dimension.

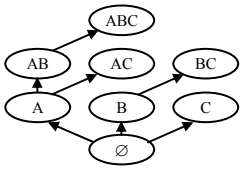


Figure 2. The "flat" execution plan of BUC (P_1)

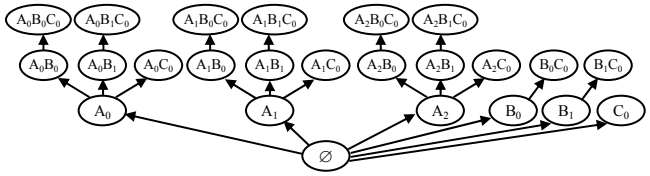


Figure 3. A straightforward hierarchical execution plan (P_2)

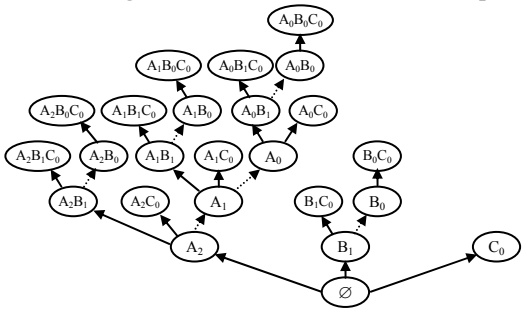


Figure 4. CURE's hierarchical execution plan (P_3)

To compare the two alternatives, consider the example of R, which is a flat fact table whose lattice appears in Figure 1 and one BUC-based execution plan (called P_1) appears in Figure 2. For R with hierarchies, a BUC-based traversal over the first and second types of lattices produces execution plans P_2 and P_3, illustrated in Figure 3 and Figure 4, respectively. Note that P_2 is actually the shortest possible extension of P_1 (height remains equal to 3), while P_3 is the tallest possible (its height is equal to 6), since it pushes the computation of any node as high as possible.

To compare the quality of P_2 vs. P_3 let us first study P_1, the execution plan of the standard BUC algorithm. According to it, BUC

first scans through all tuples of R and computes the ALL node at the bottom (marked as "∅" in Figure 2), which consists of only one tuple. Then, it sorts R according to A and isolates the first set of tuples S_{A1} that share the same value in A (note that $|S_{A1}| \leq |R|$). It aggregates the measures of these tuples, outputs the result, and proceeds recursively to node AB passing S_{A1} as input. In that call, it re-sorts S_{A1} according to B (all values in A are the same) and isolates the first set of tuples S_{A1B1} that share the same value in (both A and) B (note again that $|S_{A1B1}| \leq |S_{A1}|$). Then, it aggregates the measures of these tuples, outputs the result, and proceeds recursively to node ABC, passing S_{A1B1} as input. Subsequently, it re-sorts S_{A1B1} according to C and so on. After finishing the processing of S_{A1}, BUC repeats the same steps for all the remaining sets of tuples in R (S_{A2}, S_{A3}, …), each of which consists of tuples that have the same value in A. After constructing all nodes that contain A in their grouping attributes, BUC proceeds with B, which induces a new sorting operation on the original data of R (this time according to B). For every set of tuples that share the same value in B the measures are aggregated and execution proceeds to BC and so on. Finally, the same is repeated for node C.

A close look reveals that recursive calls at the top of P_1 (and similarly of P_2 and P_3) tend to be cheaper than recursive calls at the bottom, since the size of the tuple sets that are passed to them as input tends to decrease (recall that $|R| \geq |S_{A1}| \geq |S_{A1B1}|$) incurring smaller sorting costs. Hence, in a "taller" plan, expensive sorting costs are pushed to the bottom and are better shared among more nodes, making cube construction cheaper overall. As mentioned above, P_3 is the tallest extension of P_1 and hence the solution that better shares sorting costs.

The reason for this behavior is that having separated tuples at a coarse-grained level according to their dimension values, it is cheaper to separate them further at finer levels than to do so from scratch. For example, having found all tuples with value "Europe" at level "Continent" of dimension "Region", we can isolate all tuples with value "England" at level "Country", and among them all tuples with value "London" at level "City" in a computationally cheaper way than by searching the original data. The opposite is clearly impossible. Hence, the execution plan that makes a smarter use of pipelining is P_3 (Figure 4).

CURE uses execution plans like P_3, which have been formally defined as the result of BUC-based pruning of cube lattices that capture relationships between different levels of dimensional hierarchies natively. These can also be constructed more directly by using the following two rules:

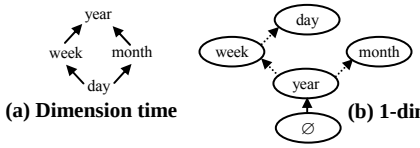
Rule 1: A solid edge connects a node N_1 to N_2 ($N_1 \rightarrow N_2$), if N_2 has the same grouping attributes at the same hierarchy levels with N_1 plus one more, which must be at the top, least detailed level (e.g., A_2 is connected via solid edges with A_2B_1 and A_2C_0).

Rule 2: A dashed edge connects a node N_1 to N_2, if their grouping attributes differ only in the rightmost dimension, whose hierarchy level in N_2 must be at one level below that of N_1 (e.g., A_2 is connected via a dashed edge with A_1, while A_0B_1 with A_0B_0).

In general, solid edges pass the execution towards nodes with more grouping attributes, while dashed edges towards nodes at lower hierarchy levels. In both cases, execution proceeds from less detailed nodes towards more detailed ones.

3.2 Handling Complex Hierarchies

Up to this point, we have deliberately used examples of only simple (i.e., linear) hierarchies for the sake of simplicity. However, there are also cases that dimensions form complex (i.e., nonlinear) hierarchies. A common example of such a case is illustrated in Figure 5a, in which dimension "time" forms a complex hierarchy.



(a) Dimension time **(b) 1-dimensional cube**

Figure 5. Example of a complex hierarchy

Note that, in the case of simple hierarchies, rule 2 defined above guarantees that at most one dashed edge may start from (respectively, end at) each node. The proof of this is straightforward. On the contrary, in the case of complex hierarchies, it is also possible to have multiple dashed edges starting from (respectively, ending at) the same node representing branches like the ones in Figure 5a. The former side-effect is desirable, since it passes execution to more detailed nodes and guarantees that the final execution plan covers all nodes. However, the latter side-effect turns the execution plan into a graph (not a tree), which is unacceptable. To overcome this, we propose a modified version for the second rule:

Rule 2 (modified): A dashed edge connects a node N_1 to N_2, if their grouping attributes differ only in the rightmost dimension **RD**, whose hierarchy level in N_2 must be at one level below that of N_1, **provided that the cardinality of the specific level of RD in N_1 is the maximum among the cardinalities of all its sibling levels in the complex hierarchy.**

Using the heuristic of maximum cardinality, we resolve tie breaks in favor of edges that start from more detailed nodes, which incur smaller additional sorting costs. For example, the 1-dimensional cube whose only dimension is "time" (Figure 5a) appears in Figure 5b. In this example, according to the original version of rule 2, node day should be connected to both node week and node month. According to the modified version of rule 2, however, the month→day edge is discarded, since the cardinality of month is lower than that of week. In this way, CURE can also handle complex hierarchies. In the rest of this paper, we will not study complex hierarchies any further due to space limitations.

3.3 Node Enumeration

Before closing this section, we find it useful to describe the enumeration scheme we have used in our implementation for uniquely identifying every node in the execution plan of CURE. We do this through an example that enumerates all nodes in Figure 4. Recall the dimension hierarchies: $A_0 \rightarrow A_1 \rightarrow A_2$, $B_0 \rightarrow B_1$, and C_0. Extending them with an extra level ALL, which we omitted before due to its simplicity (it only contains a single value), we take: $A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow$ ALL, $B_0 \rightarrow B_1 \rightarrow$ ALL, and $C_0 \rightarrow$ ALL. Renaming level ALL to A_3, B_2, and C_1, respectively, we finally take: $A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3$, $B_0 \rightarrow B_1 \rightarrow B_2$, and $C_0 \rightarrow C_1$. Hence, if we denote with $\mathcal{L}_i$ the number of levels of the i-th dimension, including this time the level ALL, we get: $\mathcal{L}_1 = 4$, $\mathcal{L}_2 = 3$, and $\mathcal{L}_3 = 2$. Ordering the dimensions in the order ABC, we can define for each a factor F_i ($i \in [1, D]$), according to formula (1) below. Then, the identifier of any node N whose i-th dimension is at level L_i ($L_i \in [0, \mathcal{L}_i\text{-}1]$) is given by formula (2).

$$F_i = \begin{cases} 1, \text{ if } i = 1 \\ F_{i-1} \cdot \mathcal{L}_{i-1}, \text{ otherwise} \end{cases} \quad \textbf{(1)} \qquad id(N) = \sum_{i=1}^{D} F_i \cdot L_i \quad \textbf{(2)}$$

In our example, we get $F_1 = 1$, $F_2 = 4$, and $F_3 = 12$. The unique identifiers of all 24 nodes appear in Figure 6.

It is also interesting that an identifier generated by formula (2) presented above can be easily decoded using modulo operations to give the levels L_i of all dimensions in the corresponding node. In our example, identifier 21 gives $L_3 = (21 \bmod F_3) = 1$, $L_2=(21 \text{ div } F_3) \bmod F_2=2$, and $L_1=((21 \text{ div } F_3) \text{ div } F_2) \bmod F_1=1$, which denotes node A_1, since B_2 and C_1 are synonyms to ALL.

Node	L_1	L_2	L_3	id
$A_0B_0C_0$	0	0	0	0
$A_1B_0C_0$	1	0	0	1
$A_2B_0C_0$	2	0	0	2
B_0C_0	3	0	0	3
$A_0B_1C_0$	0	1	0	4
$A_1B_1C_0$	1	1	0	5
$A_2B_1C_0$	2	1	0	6
B_1C_0	3	1	0	7

Node	L_1	L_2	L_3	id
A_0C_0	0	2	0	8
A_1C_0	1	2	0	9
A_2C_0	2	2	0	10
C_0	3	2	0	11
A_0B_0	0	0	1	12
A_1B_0	1	0	1	13
A_2B_0	2	0	1	14
B_0	3	0	1	15

Node	L_1	L_2	L_3	id
A_0B_1	0	1	1	16
A_1B_1	1	1	1	17
A_2B_1	2	1	1	18
B_1	3	1	1	19
A_0	0	2	1	20
A_1	1	2	1	21
A_2	2	2	1	22
∅	3	2	1	23

Figure 6. Example of node enumeration

4. EXTERNAL PARTITIONING

In the previous section, we presented an efficient execution plan for in-memory construction of hierarchical cubes. In this section, we introduce an efficient external partitioning algorithm especially suited for such plans, which makes CURE disk-based. As shown below, existing techniques are inadequate. We consider this as a significant drawback, since using efficient in-memory aggregation is not enough when the data does not fit in memory. Our main motivation has been the challenge of making CURE able to construct a complete hierarchical cube for the fact table of the APB-1 benchmark [17] in its highest density, which consists of about 496 million tuples occupying 12 GB. To the best of our knowledge, no other ROLAP method has accomplished this task.

In general, partitioning a large fact table on some subset of dimensions SD separates its tuples into smaller disjoint segments (hereafter called partitions) that fit in memory. The separation process must guarantee that tuples with common values in the dimensions of SD must be assigned into the same partition; otherwise the results of the independent aggregations over the disjoint partitions would have to be merged, which is expensive. We call the partitions that obey this rule **sound on SD**. A partition sound on SD is also called sound on the node whose grouping attribute set is equal to SD. Clearly, a partition sound on some node N is also sound on all the ancestors of N in the cube lattice.

Taking a look at the execution plan of CURE (Figure 4), we observe that there are always D nodes directly constructed from data in the original fact table R, where D is the number of dimensions. In the example of Figure 4, these nodes are A_2, B_1, and C_0. As mentioned above, partitioning R on the values of any of these nodes generates partitions sound on that node and its ancestors in the cube lattice. Hence, a straightforward implementation must perform D partitioning operations, one for each dimension at the first level of CURE's execution plan. This incurs at least D+1 reads (one read for partitioning and one read per dimension for loading the partitions during construction) and D writes of R, which can be very costly taking into account that R is usually very large. A possible optimization projects out the dimensions already partitioned, which are not necessary in cube construction, e.g. when partitioning on B we do not need to store A. This incurs (D+3)/2 reads and (D+1)/2 writes, which is still proportional to D.

Such solutions are not only expensive but may also be infeasible. Note that at the first level of CURE's execution plan appear nodes with a single dimension at its top hierarchy level, which has therefore low cardinality (cardinality decreases, since higher levels are less detailed). A low cardinality in combination with a large number of tuples in R may have as result the inability of the partitioning algorithm to generate memory-sized sound partitions. Assume that in the example of Figure 4 R's size is |R| = 10 GB, the memory size is |M| = 1 GB, and the cardinality of A_2 is $|A_2|$ = 5. Assuming a uniform data distribution, the partitioning algorithm has to generate at least |R|/|M| = 10 (sound) partitions to make them fit in memory. This is though impossible since the number of partitions is limited by $|A_2|$ (we cannot generate more sound partitions than the number of different values). Clearly, the aforementioned partitioning method is not viable. In the rest of this section, we describe CURE's solution that not only makes partitioning always feasible, but also makes cube construction faster. The solution is based on the following observations:

Observation 1: Although generating partitions sound on the top level (LT) of the first dimension (say A) may be infeasible, the same task becomes feasible with greater probability if performed on some level L<LT. This is true, since cardinalities tend to increase at lower hierarchy levels making the creation of memory-sized sound partitions possible. Such partitions are sound on node A_L and all of its ancestors in the cube lattice.

Observation 2: The cube node $\mathcal{N}$ that has D grouping attributes (like R) all at the base hierarchy level, except for the first one (say A) that is at level L+1 (L is the level on which partitioning was based before) is approximately $|A_0|/|A_{L+1}|$ times smaller than R. As L increases, this factor becomes considerable and, in practice, $\mathcal{N}$ is several orders of magnitude smaller than R, which makes it fit in memory with great probability. In the previous example, if $|A_0B_0C_0|$ = 10 GB, |M| = 1 GB, $|A_0|$ = 5,000, $|A_1|$ = 500, and $|A_2|$ = 5, then L = 1, $\mathcal{N} = A_2B_0C_0$, and $|\mathcal{N}| \approx 10$ MB << |M|.

Observation 3: We can use node $\mathcal{N}$ to construct all nodes that include A_i in their grouping attributes $\forall$ i ∈ [L+1, LT] along with all nodes that do not include A at all. The proof is based on that we can use a detailed node to construct less detailed ones (at least for non-holistic [6] aggregate functions).

Based on these, CURE selects the maximum level L ∈ [0, LT] in the hierarchy of the first dimension A of R such that partitioning on A_L generates memory-sized sound partitions on A_L and $\mathcal{N} = A_{L+1}B_0C_0\ldots$ fits in memory (if L = LT, then $A_{LT+1} \equiv$ ALL and A is projected out in $\mathcal{N}$). Such an L exists in all situations we have seen in practice. However, in the rare case that it does not exist, the partitioning algorithm can be extended properly to work on pairs of dimensions. We omit this extension due to space limitations.

After selecting L, CURE partitions R on A_L and during partitioning constructs $\mathcal{N}$ in memory (using hashing, which enables construction with one pass). It then uses the generated partitions to construct all nodes that include A_i (i ∈ [0, L]) in their grouping attributes (according to observation 1) and $\mathcal{N}$ to construct all the rest (according to observation 3). In this way, CURE performs partitioning with only 2 reads, 1 write, and an inexpensive construction of $\mathcal{N}$, which is many times smaller than R.

Note that BUC uses the heuristic of ordering the dimensions of R in decreasing cardinality to improve its efficiency, which also makes CURE's partitioning more efficient and the existence of a

proper L more probable due to the following reasons. Bringing a dimension with many unique values in the first position increases $|A_0|$. Furthermore, such a dimension generates many tuples that may be hard to scan through and interpret. Hence, it is likely that an analyst defines several hierarchy levels with lower cardinalities on this dimension to create coarser-grained views that are easier to interpret, thus decreasing $|A_{L+1}|$. Both facts increase factor $|A_0|/|A_{L+1}|$, which makes the existence of a proper L more probable (according to observation 2). Assume for example that a fact table SALES includes dimension Product organized in three levels barcode→brand→economic_strength with the following cardinalities 10,000→1,000→10, and that |M|=1 GB. Then Table 1 shows that CURE can partition SALES even if its size is 1 TB.

Table 1. Example of CURE's partitioning efficiency

|R|	L	# of Partitions	Partition Size	$\|A_0\|/\|A_{L+1}\|$	$\|\mathcal{N}\|$
10 GB	2	10	1 GB	10,000	1 MB
100 GB	1	100	1 GB	1,000	100 MB
1 TB	1	1,000	1 GB	1,000	1 GB

5. EFFICIENT CUBE STORAGE

First Kotsis and McGregor [12] and then several other researchers [5, 13, 14, 22, 24] have realized that a great portion of the data in a cube is redundant. They have used terms like prefix/ suffix/partial/total redundancy, equivalent tuples, or BSTs. A detailed description of these terms exceeds our purpose. Alternatively, in an attempt to express all these terms under a global definition, we state that: *A value that is stored in a data cube is called redundant if it is repeated in the same attribute elsewhere in the cube as well.* According to this, we can generally recognize two types of redundancy: **Dimensional redundancy** appears whenever a specific dimension value is repeated in different tuples. **Aggregational redundancy** appears whenever a specific aggregate value is repeated in different tuples.

Clearly, removing redundant data produces a smaller cube and benefits computational efficiency as well, since smaller cubes require fewer aggregations and induce smaller output costs. However, avoiding redundancy is not the only factor that affects cube size. Another equally important factor concerns the storage format of non-redundant data. Existing ROLAP methods that avoid redundancy store the entire cube as a monolithic relation of fix-sized tuples, which is far from compact. Unlike such methods, CURE strikes on both factors simultaneously, avoiding the storage of redundancy, while storing non-redundant data in a very compact relational form. Note that storing tuples efficiently is more critical in hierarchical cubes, since they consist of more nodes and of denser areas at coarse-grained levels, which generate large numbers of non-redundant tuples. In the following subsections, we describe CURE's efficient storage format of non-redundant data and then an algorithm that classifies cube tuples according to the type of redundancy they contain.

5.1 Storage Format

Existing ROLAP methods that identify redundancy use a single D-dimensional relation for storing non-redundant data, which introduces a large number of NULL values for tuples that belong to nodes of lower dimensionality. Instead, we propose storing tuples separately, according to the node they belong to. Every such tuple t stored in a cube node N has been produced by the aggregation of a tuple set S in the original fact table (say R). Hence, without further optimizations, t should be stored as shown in Figure 7, assuming that it consists of X dimensions and Y aggregates.

Figure 7. Basic tuple format

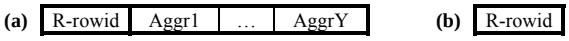


Figure 8. Normal and trivial tuple formats

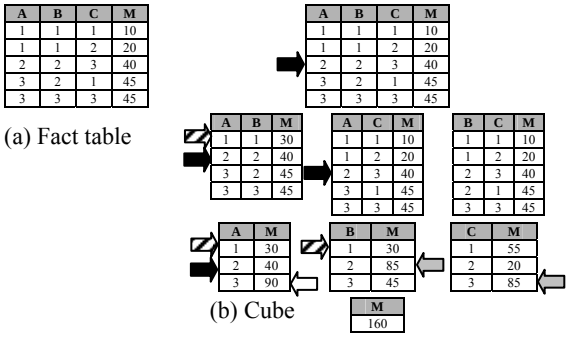


Figure 9. Fact table R and its data cube

Clearly, t has the same dimension values with every tuple $t_S \in S$ projected on the grouping attributes of N, hence every cube tuple is dimensionally redundant. To overcome this, we propose (Figure 8a) replacing all dimension values of t with a row-id reference (R-rowid) pointing to any $t_S \in S$. In our implementation, R-rowid stores the minimum row-id of the tuples in S. Note that replacing dimension values by a row-id is useful only if the size of the former is smaller than the size of the latter. This may not be true for tuples that belong to nodes of one or two dimensions. Such nodes, however, are few and relatively small compared to more detailed nodes at higher lattice levels. Hence, although CURE can decide dynamically which format is preferable, the cases when the storage of redundant data is beneficial are so rare and the benefits so small, that we treat them uniformly with the others.

Having dealt with dimensionally redundant data, we can further focus on aggregational redundancy in order to apply additional optimizations. As described below, we can classify cube tuples into three categories, according to the type of aggregational redundancy they contain. CURE uses (at most) three tables per node, one for each category. Their schema is described below.

Normal Tuples (NTs): If t is only dimensionally but not aggregationally redundant, we call it normal. The most compact format for NTs is the one of Figure 8a, since we cannot avoid storing the aggregates. For example, if Figure 9a shows the tuples stored in R (whose lattice appears in Figure 1) and Figure 9b shows the corresponding cube (in an uncompressed form), then tuple <3, 90> in node A (pointed by the white arrow) is NT, since there is no other tuple in the entire cube with an aggregate value equal to 90.

Trivial Tuples (TTs): If t comes from a singleton set S (|S| = 1), no aggregation is necessary for its computation, but just a simple projection of the sole $t_S \in S$ on N's grouping attributes. In this case, we call t trivial. Note that, if t is trivial, its aggregate values are equal to the measures of t_S, hence TTs are aggregationally redundant and their aggregates can be retrieved from the original tuple they come from. Hence, TTs can be minimally stored using just row-ids and discarding all aggregate values (Figure 8b).

Interestingly, it can be proven that a TT that belongs to N belongs also to all the ancestor nodes of N in the cube lattice, since it comes from the simple projection of a single tuple that has not

matched with any other tuples in the original fact table and hence cannot match either for the generation of a more detailed tuple. This property holds for hierarchical cube lattices as well, hence also for CURE's execution plan (Figure 4), which is a pruned lattice. This is beneficial, since it means that any TT can be stored once, only in the least detailed node N_{LD} it belongs to, and be shared among this node and its ancestors that form an entire sub-tree rooted at N_{LD}. This remark gives another advantage to taller execution plans (Figure 4) against shorter ones (Figure 3), since the former maximize the size of such sub-trees resulting into greater storage savings. In the example of Figure 9b, all cube tuples with value A = 2 (pointed by black arrows) are TTs, since they have been produced by a simple projection of the single tuple <2, 2, 3, 40> in R. Storing only one TT in node A (the least detailed one) is enough to represent them all, due to the property mentioned above. This tuple can then be considered as shared among nodes A, AB, AC, and ABC (that form an entire sub-tree rooted at A) and can be easily retrieved on demand. Note that TTs are similar to BSTs [24] but are stored far more efficiently.

Common Aggregate Tuples (CATs): If t is aggregationally redundant and non-trivial (|S| > 1) we call it CAT. By definition, there must be at least one more CAT t' such that t and t' have common aggregate values. The existence of CATs can be attributed to two reasons, namely common source and coincidence:

- **Common source CATs** attribute equality of their aggregates to the fact that they have been produced by the same set of tuples of R. In Figure 9b, tuples <1, 1, 30> in AB, <1, 30> in A, and <1, 30> in B (pointed by striped arrows) are common source CATs, since they have been produced by the same tuple set S = {<1, 1, 1, 10>, <1, 1, 2, 20>} of R.
- **Coincidental CATs** are the CATs that have the same aggregates, although they have been produced by different tuple sets of R. In Figure 9b, tuples <2, 85> in B and <3, 85> in C (pointed by gray arrows) are examples of coincidental CATs.

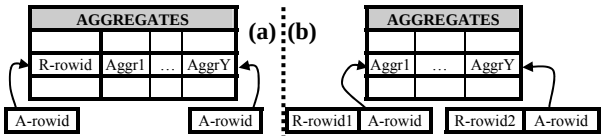


Figure 10. Alternative CAT formats

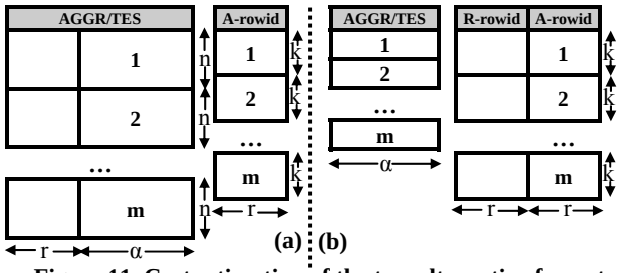


Figure 11. Cost estimation of the two alternative formats

To avoid storing the aggregate values of CATs redundantly, we propose the use of an additional relation AGGREGATES to store such common values only once and the replacement of all aggregate values in CATs with a row-id (A-rowid) pointing at the corresponding tuple in AGGREGATES (below we investigate the conditions under which such storage is beneficial). However, the decision of a specific schema for AGGREGATES depends on the type of CATs that prevails. As shown below, format (a) (Figure 10a) produces a more compact cube, if common source CATs prevail. Otherwise, format (b) (Figure 10b) is better, provided that Y > 1. The difference is in the storage of R-rowid, which points at the first tuple $t_S \in S$ that has contributed to the generation of the corresponding CAT. If two CATs have common source then their R-rowids are equal and can be stored only once in relation AGGREGATES (Figure 10a). Otherwise, their R-rowids differ and format (b) is more compact, since format (a) would induce the storage of a second tuple in relation AGGREGATES, which is long and mainly redundant.

Up to this point we have decided the optimal formats for NTs and TTs and the question that arises is how to choose the most compact between the alternative formats (a) and (b) of CATs. The answer to this question depends on the data. Assume that there are m different combinations of aggregate values of the form <Aggr1, …, AggrY> stored in AGGREGATES (Figure 11), and that each combination is pointed on average by k CATs produced by n different tuple sets in the fact table. Let r denote the size of a row-id and α the size of a combination of aggregates. If C_a and C_b denote the storage space occupied by format (a) and (b), respectively, then format (a) is preferable when the following inequality holds:

$$C_a < C_b \Leftrightarrow (n(r+\alpha)+kr)m < (\alpha+2kr)m \Leftrightarrow (k-n)r > (n-1)\alpha$$

Setting $n-1 \approx n$ and $\alpha = Yr$ (Y is the number of aggregates), we get:

$$(k-n)r > nYr \Leftrightarrow \frac{k-n}{n} > Y \Leftrightarrow \frac{k}{n} > Y+1$$

The last inequality indicates that format (a) is a better choice when k is several (Y+1) times larger than n, i.e., when the majority of CATs are common source. Otherwise, if coincidental CATs prevail, as also intuitively expected, format (b) is the right choice. Using similar reasoning, we can also prove that the storage of all CATs in the NT format is better than using format (a) when $\frac{k}{n} < \frac{Y+1}{Y}$ and preferable to format (b) when $\frac{k}{k-1} > Y$. Since k and n are integers and $k \geq n$ the former can be true only when k = n, i.e., when all CATs are coincidental, while the latter holds when either Y = 1 (we store only one aggregate) or k = 1 (there are no CATs). The rule below summarizes the above results. Interestingly, as shown in the following subsection, CURE can gather statistics during cube construction that enable it to test these criteria and decide dynamically the most efficient storage format.

if common source CATs prevail store them in format (a)
else if Y = 1 store CATs as NTs
else store CATs in format (b)

5.2 Tuple Classification Algorithm

As already mentioned, the storage format used to condense the cube is critical, since it decides the amount of storage savings and thus the practicability of a solution. Nevertheless, the use of any sophisticated format is useless if the cubing method cannot classify tuples into the proper class (NT, TT, or CAT), according to the type of redundancy they originally contained. In this subsection, we describe the algorithms CURE uses to identify all types of redundancy in order to classify tuples to a proper type.

As also described elsewhere [24], identification of TTs is easy for BUC-based algorithms. Recall that such algorithms are recursive. The input of each recursive call that produces a tuple t of node N consists of a set of tuples of the original fact table that have the same values in the grouping attributes of N and must be aggre-

gated to produce t. Whenever the input of such a call consists of only a single tuple t_S, then the resulting tuple t is TT and can be produced by simply projecting t_S on N's grouping attributes. Furthermore, the bottom-up lattice traversal guarantees that N is the least detailed node to which t belongs. Hence, CURE can simply output the row-id of t_S in the corresponding TT relation of N (relation that stores TTs using the format in Figure 8b) and prune recursion to avoid redundant storage of the same TT in N's ancestors. Pruning saves both storage and computational costs.

Having handled all TTs with the previous algorithm, what remains is the separation of NTs from CATs (which is not necessary in the rare case that coincidental CATs prevail and Y = 1, since in this case CATs are stored as NTs, as explained above). Recall that both NTs and CATs are produced by the aggregation of a tuple set S of the original fact table consisting of multiple tuples (|S| > 1), hence the size of S is not a correct criterion, as is in the case of TTs. Our solution is given below.

Figure 12. The signature structure

The task of separating NTs from CATs can be performed with the use of some meta-data kept during aggregation. We organize the meta-data accompanying each aggregated tuple in a structure called *signature* (Figure 12). The signature is a minimal representation of an aggregated tuple holding information which is necessary for the separation of NTs from CATs. Assuming that a tuple t of node N is produced by the aggregation of a tuple set S of the original fact table, then Aggr-y (y ∈ [1, Y]) denotes the y-th aggregate value produced by the aggregation of the tuples in S, R-rowid is the minimum row-id of the tuples in S, and NodeId is the unique identifier of N produced as described in Section 3.3. Actually, studying CURE's storage schema described in the previous subsection, we can argue that the meta-data in a tuple's signature is enough for storing the corresponding tuple in the proper format. Hence, CURE actually needs only the meta-data and not the data itself, which would increase memory requirements.

Assume for the moment that CURE has unlimited resources to hold all signatures of all aggregated (non-trivial) tuples in a large in-memory temporary relation, called *signature pool*. Then, during cube computation it outputs to disk only TTs and for all the other tuples it outputs nothing; instead, it keeps their signatures in the pool. In this way, upon return from all recursive calls, CURE has gathered in the pool all signatures of all non-trivial tuples. What remains is a final step to separate them into NTs and CATs. CURE performs this task using signature sorting in order to bring in adjacent memory places signatures of tuples with the same aggregates (and optionally produced by the same tuple set if format (a) of Figure 10a is used). This means that all signatures must be sorted according to the fields <Aggr1, …, AggrY> (and optionally R-rowid). During sorting, CURE can also calculate inexpensive and accurate statistics that enable it to choose the most efficient storage format for CATs, using the criteria described above. After the sorting operation has finished, CURE scans through the sorted pool comparing adjacent signatures and identifies every set of signatures S_{SIG} that have the same aggregate values (and optionally the same R-rowid). Clearly, there are two possibilities:

- **If $|S_{SIG}| = 1$**, the sole signature in S_{SIG} represents an NT and CURE stores a new tuple t_{NT} = <R-rowid, Aggr1, …, AggrY> in the NT relation of the node indicated by the field NodeId of the signature. Note that t_{NT} can be produced by the signature itself, which proves that only the meta-data is enough.
- **If $|S_{SIG}| > 1$**, the signatures in S_{SIG} represent CATs whose independent storage would be redundant. To avoid this, CURE uses one of the formats mentioned above. If the statistics have indicated that the majority of CATs are common source, it stores a new tuple t_{AG} = <R-rowid, Aggr1, …, AggrY> in the common relation AGGREGATES and t_{AG}-rowid in all relations CAT of the corresponding nodes indicated by the field NodeId of the signatures. Otherwise, if the majority of CATs has been found coincidental (and Y>1), it stores a tuple t_{AG} = <Aggr1, …, AggrY> in AGGREGATES and a pair <R-rowid, t_{AG}-rowid> in the proper CAT relations. Note that the decision on the format can be made once and used globally afterwards.

Up to this point, we have assumed that the entire signature pool fits in memory. This is "cheaper" than holding the entire cube, but it is still non-realistic. A naive solution to this problem would be to flush the pool on the disk and perform external operations. However, this would incur great I/O costs harming CURE's efficiency. An efficient heuristic solution can be based on the observation that CURE's recursive calls are "well instrumented" due to its execution plan (Figure 4) that enables pipelining, which means that cube tuples generated by the same data are constructed in recursive calls near in time with great probability. Hence, we actually need not hold all signatures in memory, but only a proper "working set". The devised CURE algorithm uses a limited signature pool. As long as this pool is not full, CURE keeps adding signatures, as described above. When it becomes full, it sorts all signatures available and flushes to disk cube tuples classified as NTs or CATs, based only on information resident in memory. In this way, the pool becomes empty again waiting for the following signatures. This heuristic makes CURE efficient at the expense of possibly storing redundantly some repeated data. However, our experimental evaluation has shown that the size of the signature working set is relatively small and that using a pool of only 1,000,000 signatures occupying approximately (Y+2)*4 MB can generate a cube whose size is very close to the optimal, which would be generated by the original version of the algorithm. Hence, the pool size can be thought of as an input parameter. The trade-off is obvious; a zero-length pool prohibits the identification of CATs, while an unlimited pool enables CURE identify them all.

5.3 Extensions and Improvements

To further enhance CURE's efficiency we can use several implementation variations. For example, if the underlying ROLAP engine supports bitmap indexing, which is true in many widely used servers, we can change the format of relation TT (and probably CAT if it uses format (a)) without affecting their ROLAP compatibility. Instead of storing each row-id (which consumes several bytes) separately, we can use such a bitmap to index the tuples that need to be retrieved for answering queries on the corresponding node N. A potential problem in this case is that we have to waste some space for the storage of zero bits for all tuples that do not belong to N. Hence, this variation makes sense only if the number of row-ids stored originally is large enough.

Furthermore, we have seen that it is beneficial to sort all row-ids in TT relations (in a post-processing step) according to the order of the tuples they point at. This produces sequential scans during query answering. Our experiments have shown that such a post-

processing step is inexpensive compared to the cube construction time and results into great savings during cube usage. Note that the use of bitmap indices achieves such a sorting indirectly.

Another option is to trade off some storage space for faster query response times. This can be achieved by changing the schema of the NT relations to physically store the actual dimension values of NTs instead of the row-ids of the corresponding tuples.

Finally, we have noticed that the bottleneck in query answering is actually in the access of two relations, the original fact table and the relation AGGREGATES, since all cube nodes point at tuples stored in them. Hence, it is beneficial during query answering to cache as many tuples from them as possible. This property is unique in CURE, since in other ROLAP methods there is no simple rule to indicate which relations to cache in order to enhance efficiency overall. Similarly, instead of indexing the entire cube, which is expensive, we can index just the original fact table consuming much cheaper resources. Further investigation of caching and indexing exceeds the purpose of this paper.

6. THE CURE ALGORITHM

In the previous sections, we described independently the most important parts of CURE, focusing mainly on our contributions to ROLAP cubing when dealing with hierarchies. In this section, we gather everything together and present CURE and some auxiliary functions in pseudo-code (Figure 13). We omit details, which have been elaborated above.

```
Algorithm CURE(inputRelation)
1:  for (d = dim; d < numOfDims; d++)
2:      topLevel[d] = GetHierarchyMetaData(d);
3:      baseLevel[d] = 0;
4:      levels[d] = topLevel[d];
5:  end for
6:  if (inputRelation.size() < memorySize)
7:      input = Load(inputRelation);
8:      ExecutePlan(input, 0, levels);
9:  else
10:     L = SelectPartitionLevel();
11:     [partRelations,numOfParts,nodeRelation]=Partition(inputRelation,L);
12:     levels[0] = L; // Start from level L of the first dim
13:     for (i = 0; i < numOfParts; i++)
14:         partition = Load(partRelations[i]);
15:         FollowEdge(partition, 0, levels);
16:     end for
17:     levels[0] = topLevel[0]; // Start from the top level of the first dim
18:     baseLevel[0]=L+1;      // and do not proceed below level L+1
19:     N = Load(nodeRelation);
20:     ExecutePlan(N, 0, levels);
21: end if
22: FlushSignatures(signaturePool);

Algorithm ExecutePlan(input, dim, levels)
1:  if (input.count() == 1)
2:      WriteTT(input[0], dim, levels[dim]);
3:      return;
4:  end if
5:  Aggregate(input); //Places result in outputRec
6:  if (signaturePool.full()) FlushSignatures(signaturePool); end if
7:  AddNewSignature(signaturePool, outputRec);
8:  for (d = dim; d < numOfDims; d++)
9:      FollowEdge(input, d, levels); // Follow all solid edges
10: end for
11: if (dim >= 1 && levels[dim-1] > baseLevel[dim-1])
12:     levels[dim-1]--;
13:     FollowEdge(input, dim-1, levels); // Follow a dashed edge
14:     levels[dim-1]++;
15: end if

Algorithm FollowEdge(input, dim, levels)
1:  start = 0;
2:  Sort(input, dim, levels[dim]); // Re-sort the current segment
3:  while((count = GetNextSegment(input, dim, levels[dim], start)) > 0)
4:      outputRec.dim[dim] = input[start].dim[dim][levels[dim]];
5:      ExecutePlan(input[start ... start+count], dim+1, levels);
6:      start += count;
7:  end while
8:  outputRec.dim[dim] = ALL;
```

Figure 13. Algorithm CURE and auxiliary functions

Algorithm CURE initializes some global arrays that contain metadata about hierarchy levels and sets all dimensions to their top levels (lines 1-5), which complies with its execution plan (Figure 4). Then, it checks if the input fact table fits in memory (line 6), in which case it loads it and calls ExecutePlan to construct the entire cube (lines 7-8). Otherwise, it selects the partitioning level L of the first dimension (line 10), as described in Section 4, and partitions the input relation on L (line 11). Function Partition returns the relation names of the partitions created (partRelations), their number (numOfParts), and the relation name of node $\mathcal{N}$ (nodeRelation), which is the node constructed in memory. Upon return from Partition CURE sets the initial level of the first dimension to L (line 12) to construct only the nodes dictated by observation 1 (Section 4) and proceeds with the computation of the sub-cubes of all partitions (lines 13-16). Afterwards, based on observation 3, it constructs all remaining nodes using $\mathcal{N}$ (lines 17-20). Finally, it flushes to disk all signatures gathered in the signature pool, calling FlushSignatures in line 22. This function sorts signatures and separates NTs from CATs, as explained in Section 5.2. Note that it is called not only in the end, which would need infinite resources, but also within ExecutePlan (line 6).

Algorithm ExecutePlan implements the bottom-up and depth-first traversal mentioned in Section 3.1. It first checks if the size of its input is equal to 1, which reveals a TT. In this case it early stops recursion (lines 1-4). Otherwise, it aggregates all input tuples and stores a new signature in the pool, after checking that it fits (lines 5-7). Then, it follows all solid edges of the execution plan (lines 8-10) and a dashed edge, if one exists (lines 11-15).

Finally algorithm FollowEdge sorts its input tuples according to their values in level levels[dim] of dimension dim (line 2) and separates them in smaller segments that have the same value (line 3). For every such segment it calls ExecutePlan in order to pass it as input to higher lattice levels (line 5).

7. EXPERIMENTAL EVALUATION

To evaluate the efficiency of the proposed techniques, we have implemented CURE and the most efficient methods in ROLAP, namely BUC and BU-BST. The former identifies no redundancy, while the latter identifies TTs but does not use efficient storage for non-redundant data. (We have not implemented QC-Tables [13], since the relational representation of the so-called Quotient Cubes has been shown to have many problems [14]. This can be solved with the use of QC-Trees [14], but these are tree-like data structures and, hence, outside the scope of this paper.) We call CURE+ the variation of CURE that applies a post-processing step to sort and replace row-ids with bitmap indices, as explained in Section 5.3. We have run our experiments on a Pentium 4 (2.8 GHz) PC with 512 MB memory under Windows XP. In this section, we present the results of our experimental evaluation.

Flat Cubes: In our first set of experiments we have evaluated the efficiency of all algorithms in constructing flat cubes over real and synthetic datasets. Additionally, we have tested the effectiveness of the generated cube formats in query answering, which is important, since condensing a cube is pointless if it cannot provide fast query response times. The workloads we have used consist of 1,000 random node queries, which perform no selection.

Real datasets: We have experimented with two widely used real-world datasets, namely CovType [3] and Sep85L [7]. The former has 10 dimensions and 581,012 tuples, while the latter has 9

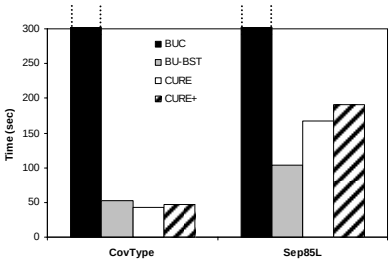


Figure 14. Real datasets-Construction Time

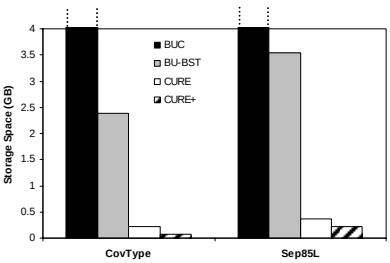


Figure 15. Real datasets-Storage Space

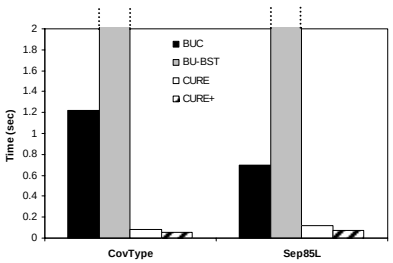


Figure 16. Real datasets-Average QRT

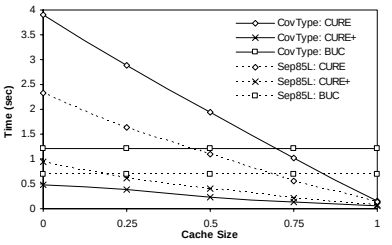


Figure 17. Effect of caching on Av. QRT

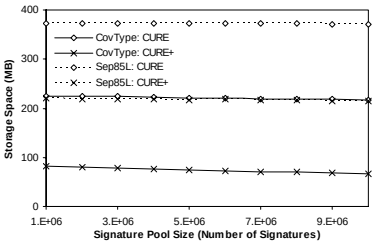


Figure 18. Pool size vs. Storage Space

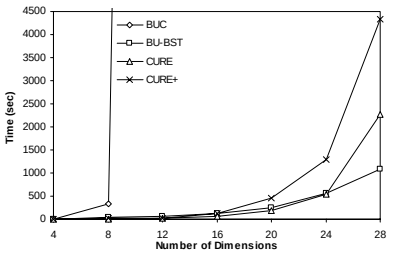


Figure 19. Dimensionality vs. Constr. Time

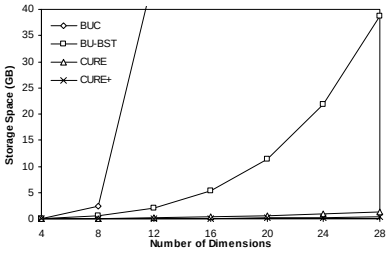


Figure 20. Dimensionality vs. Storage Space

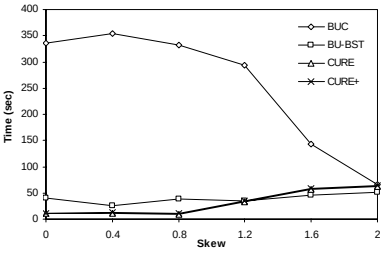


Figure 21. Skew vs. Construction Time

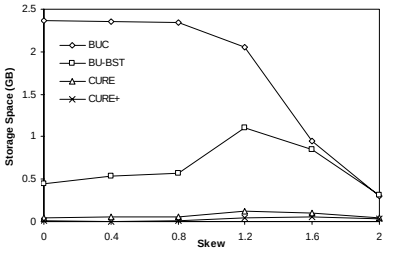


Figure 22. Skew vs. Storage Space

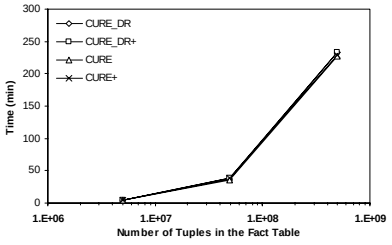


Figure 23. Construction Time (APB-1)

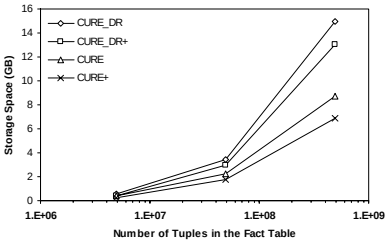


Figure 24. Storage Space (APB-1)

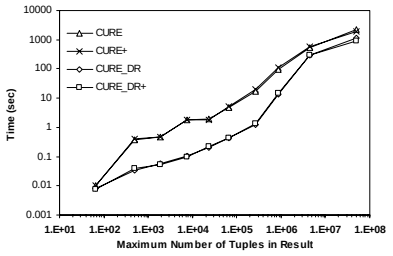


Figure 25. Average QRT (APB-1 density 4)

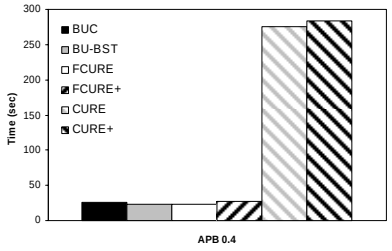


Figure 26. Flat vs. Hier. cube (Construction Time)

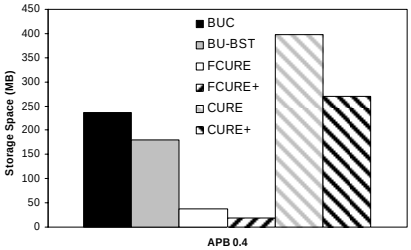


Figure 27. Flat vs. Hier. cube (Storage Space)

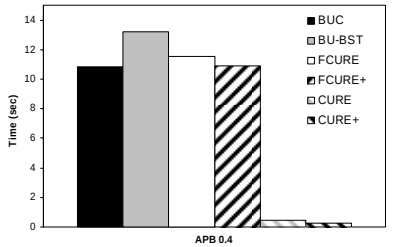


Figure 28. Flat vs. Hier. cube (Average QRT)

dimensions and 1,015,367 tuples. Figure 14 shows the time spent on the construction of the corresponding cubes, Figure 15 their storage space requirements, and Figure 16 the average query response time in a workload of the type mentioned above. Note that in both datasets the size of the CURE cube (cube constructed by CURE) is an order of magnitude smaller than the size of the BU-BST cube, since BU-BST removes only specific kinds of redundancy and mainly because it does not use an efficient schema for non-redundant data. The BUC cubes exceed the ranges of the graph. Interestingly, in the Sep85L dataset the construction time of CURE is a little worse than the one of BU-BST. This is attributed to the fact that Sep85L contains some dense areas that generate many non-trivial tuples, which forces CURE to pay a greater cost for sorting signatures. Nevertheless, the small penalty in construction time is compensated by great storage savings and efficiency in answering queries. The same holds for the time cost of

the post-processing step of CURE+, which is negligible compared to the improvements it offers. Furthermore, BU-BST cubes are two and three orders of magnitude worse than their BUC and CURE counterparts, respectively, in query answering. This is attributed to their monolithic format that stores all cube tuples in a single relation incurring a sequential scan of the entire cube for answering any query. This could be improved only by indexing the entire BU-BST cube in order to cluster together tuples that belong to the same node. However, indexing the entire cube is non-trivial both in computation time and in storage space. CURE also outperforms BUC in query answering due to its smaller size and mainly due to the use of caching of the original fact table. Recall that for BUC and BU-BST there is no rule on what to cache in order to improve efficiency overall. The effect of caching is further studied in Figure 17. In this graph the x-axis shows the portion of the original fact table cached. CovType is sparser, which produces more tuples in each node and more accesses in the original fact table, deteriorating CURE's performance in small cache sizes. Nevertheless, even when no caching is used, CURE+ is close to BUC outperforming it in CovType.

Finally, Figure 18 illustrates the effect of the pool size in the ability of CURE to identify redundancy. Apparently, the final result size is monotonically decreasing with memory size. However, improvement is minor, which confirms our claim that the signature "working set" is relatively small.

Synthetic datasets: To test the effect of dimensionality, skew, and size of the original data on the scaling of the proposed methods, we have generated synthetic datasets of various parameters. Figure 19 and Figure 20 illustrate the effect of dimensionality on construction time and storage space, respectively. In these experiments, the number of tuples in the original fact table is T=500,000, the zipf factor is Z=0.8, and the cardinality of the i-th dimension is C_i=T/i. Clearly, on storage space, CURE and especially CURE+ are the undisputed winners. Furthermore, in moderate dimensionalities (up to D=24) CURE is slightly faster than BU-BST compensating signature sorting by smaller output costs. The efficiency of CURE+ degrades faster due to the large number of nodes that need to be processed in the post-processing step. In very high dimensionalities (D>24) CURE is outperformed by BU-BST, since the latter stores all tuples in a single relation, while the former generates (at most) three relations per node. Theoretically, the maximum number of relations created by CURE is $3 \cdot 2^D$, which seems prohibitive for D=28. However, in practice this number is much smaller, since in great dimensionalities there are many TTs, which saves the generation of a very large number of relations. In our experiment, CURE constructed 88,932 relations which is four orders of magnitude smaller than the theoretical $3 \cdot 2^{28} = 805,306,368$.

Figure 21 and Figure 22 show the effect of skew on the efficiency of the studied methods. In this experiment we have set D=8, T=500,000, and C_i=T/i, while varying Z from 0 (uniform distribution) to 2. Although in the existing literature BUC-based methods have been shown to degrade in high skew values, we have confirmed the remark of others [2] that using CountingSort instead of QuickSort for tuple sorting is very helpful. Moreover note that in low Z values the cube is sparse, which generates many TTs decreasing the size of CURE and BU-BST. In moderate Zs dense areas appear and the size of both methods increases. Finally, in really high skews the cube becomes so dense that the total number of cube tuples is very small, hence the sizes decrease again. Note that the efficiency of BUC seems to improve in high Zs due to the great saving in its output costs. The fact that in Z=2 the sizes of BUC and BU-BST are approximately equal denotes that in this case there are no TTs and CURE attributes its storage savings to the identification of dimensional redundancy and CATs.

Varying the number of tuples so that they still fit in main memory has not revealed any interesting trends and the corresponding graphs are omitted. In summary, CURE slightly wins BU-BST in the time field, while the cube it produces is much more compact.

Hierarchical Cubes: In this set of experiments, we have evaluated the efficiency of several variations of CURE in constructing hierarchical cubes and the effectiveness of the corresponding formats in query answering. (Recall that neither BUC nor BU-BST support hierarchies.) The datasets we have used are synthetic and have been produced by the data generator of the APB-1 benchmark [17], which is a standard in OLAP. The generated fact table has two measures (Unit Sales and Dollar Sales) and four dimensions organized in hierarchies as follows (in parenthesis we show the corresponding cardinalities). Product: Code (6,500) → Class (435) → Group (215) → Family (54) → Line (11) → Division (3), Customer: Store (640) → Retailer (71), Time: Month (17) → Quarter (6) → Year (2), and Channel: Base (9). The size of the fact table is tuned by a density factor varying between 0.1 and 40. The lowest density factor generates a fact table consisting of 1,239,300 tuples occupying approximately 30 MB (in binary format). The same figures in the highest density are 400 times larger (495,720,000 tuples and 12 GB). The total number of nodes in the cube is $(6+1)\cdot(2+1)\cdot(3+1)\cdot(1+1) = 168$. Note that the base-level cardinality of all dimensions is very low; this implies that any naive partitioning algorithm would fail. However, the partitioning algorithm of CURE is able to handle this case smoothly.

Figure 23 and Figure 24 show the construction time and the storage space, respectively, for a low (0.4), a medium (4), and the highest possible (40) density. The values along the x-axis indicate the number of tuples in the corresponding fact tables. CURE_DR is a variation of CURE that removes no dimensional redundancy from NTs trading some storage space for query efficiency. Evidently, all variations of CURE scale very well attributing their performance in the efficient execution plan, the external partitioning algorithm, and the effective storage format they use, which reduces output costs. Constructing a full hierarchical cube for the APB-1 benchmark in its highest density in approximately 3h 50min using very limited resources (256 MB of memory for loading partitions and caching signatures) is impressive. In the field of storage space CURE+ is the winner constructing a cube that occupies 6.86 GB (recall that the original fact table size has been 12 GB).

Figure 25 illustrates the average query response times for all formats under a workload of all possible (168) node queries in APB-1 with density factor 4 separated into ten equal-sized sets that have been produced by ordering the queries according to the number of tuples they return. The first set contains the 17 smallest queries and so on. Note that CURE_DR and CURE_DR+ cubes take less than 0.5 seconds on average to answer 60% of all node queries possible (that return up to 10^5 tuples) and less than 10 seconds for 80% (that return up to 10^6 tuples). Such query response times should be considered very fast for heavy workloads like the ones described here. Note that while testing our software we have used a widely accepted commercial database server, which has taken 12 hours to answer 20 small and moderate que-

ries, whose maximum result size has been 534,654 tuples. Note also that queries with smaller results, which can be answered very efficiently, have more practical interest for analysts, since they are easier to interpret. On the contrary, queries that return many millions of tuples are impractical and would be more interesting if they were combined with some selection of specific ranges (accelerated by indexing techniques). Our experiments with APB-1 in density factor 40 have shown similar trends; hence they are not explicitly shown. Furthermore, expectedly, in APB-1 with density factor 0.4, whose fact table fits in main memory, the results have been orders of magnitude better, due to caching.

Moreover, we have investigated the trade-offs between constructing flat (only at the finest level of detail) and hierarchical cubes over hierarchical data. The dataset we have used is APB-1 in density 0.4, which fits in memory. FCURE is the version of CURE that generates flat cubes ignoring hierarchies. Clearly, the construction of a flat cube is faster (Figure 26) and occupies less storage space (Figure 27), but a hierarchical cube offers greater advantages in answering roll-up/drill-down queries fast (Figure 28). In all cases some variation of CURE provides the best solution.

Finally, we have noticed that answering count iceberg queries (which contain a predicate of the form HAVING count(*) > min_count in their SQL syntax) over a CURE cube is orders of magnitude more efficient than doing so over any other format, since in this case TTs can be ignored (recall that the count for TTs is always 1). We omit the exact figures due to space limitations.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have studied ROLAP cubing in the presence of hierarchies and presented CURE, a novel ROLAP cubing method that addresses these issues and constructs complete data cubes over very large datasets with arbitrary hierarchies. To achieve this we have introduced a novel and efficient execution plan suitable for hierarchical cube construction and revisited partitioning and size reduction methods complicated due to the existence of hierarchies. The effectiveness of CURE has been demonstrated through experiments on both real-world and synthetic datasets (including the APB-1 benchmark in its highest density), which have given very promising results with respect to the potential of CURE overall.

In the future, we are planning to compare CURE directly with Dwarf and QC-Trees, prominent cubing methods that use specialized tree-like data structures. We expect this comparison to reveal the fundamental strengths and weaknesses of the two underlying techniques. Furthermore, we are planning to investigate indexing for accelerating selective queries. Finally, we will further study incremental updating for redundant tuples in CURE cubes. Our initial investigation has resulted in efficient methods for updating NTs and TTs, and we are currently working on CATs.

9. REFERENCES

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnam, and S. Sarawagi. *On the Computation of Multidimensional Aggregates.* In VLDB 1996.

[2] K. Beyer and R. Ramakrishnan. *Bottom-Up Computation of Sparse and Iceberg CUBEs.* In SIGMOD 1999.

[3] J. A. Blackard. *The Forest CoverType Dataset.* ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype

[4] Z. Chen, V. R. Narasayya. *Efficient Computation of Multiple Group By Queries.* In SIGMOD 2005.

[5] Y. Feng, D. Agrawal, A. Abbadi, A. Metwally. *Range CUBE: Efficient Cube Computation by Exploiting Data Correlation.* ICDE 2004.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals.* Proceedings of the 12th International Conference on Data Engineering, 1996.

[7] C. Hahn, S. Warren, and J. London. *Cloud reports.* http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html

[8] J. Han, J. Pei, G. Dong, K. Wang. *Efficient Computation of Iceberg Cubes with Complex Measures.* In SIGMOD 2001.

[9] V. Harinarayan, A. Razaraman and J. D. Ullman. *Implementing Datacubes Efficiently.* In SIGMOD 1996.

[10] H. V. Jagadish, L. Lakshmanan, and D. Srivastava. *What can Hierarchies do for Data Warehouses?* In VLDB 1999.

[11] N. Karayannidis, T. Sellis, Y. Kouvaras. *CUBE File: A File Structure for Hierarchically Clustered OLAP Cubes.* EDBT 04.

[12] N. Kotsis and D. R. McGregor. *Elimination of Redundant Views in Multidimensional Aggregates.* In DaWaK 2000.

[13] L.V.S. Lakshmanan, J. Pei, J. Han. *Quotient Cube: How to Summarize the Semantics of a Data Cube.* VLDB 2002.

[14] L.V.S. Lakshmanan, J. Pei and Y. Zhao. *QCTrees: An Efficient Summary Structure for Semantic OLAP.* SIGMOD 2003.

[15] C. Li, G. Cong, A. K. H. Tung, S. Wang. *Incremental maintenance of quotient cube for median.* KDD 2004.

[16] X. Li, J. Han, and H. Gonzalez. *High-Dimensional OLAP: A Minimal Cubing Approach.* In VLDB 2004.

[17] OLAP Council. *APB-1 OLAP Benchmark.* http://www.olapcouncil.org

[18] K. A. Ross and D. Srivastava. *Fast Computation of Sparse Datacubes.* In VLDB 1997.

[19] S. Sarawagi, R. Agrawal and A. Gupta. *On Computing the Data Cube.* Research report 10026. IBM Almaden Research Center, San Jose, California 1996.

[20] Z. Shao, J.Han, and D.Xin. *MM-Cubing: Computing Iceberg Cubes by Factorizing the Lattice Space.* In SSDBM 2004.

[21] Y. Sismanis, A. Deligiannakis, Y. Kotidis, N. Roussopoulos. *Hierarchical Dwarfs for the Rollup Cube.* DOLAP 2003.

[22] Y. Sismanis, A. Deligiannakis, N. Roussopoulos and Y. Kotidis. *Dwarf: Shrinking the petacube.* In SIGMOD 2002.

[23] Y. Sismanis, and N. Roussopoulos. *The Complexity of Fully Materialized Coalesced Cubes.* In VLDB 2004.

[24] W.Wang, H.Lu, J.Feng, J.Xu Yu. *Condensed Cube: An Effective Approach to Reducing Data Cube Size.* ICDE 2002.

[25] D. Xin, J. Han, X. Li, and B. W. Wah. *Star Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration.* In VLDB 2003.

[26] Y. Zhao, P. M. Deshpande and J. F. Naughton. *An Array–Based Algorithm for Simultaneous Multidimensional Aggregates.* In SIGMOD 1997.