The VLDB Journal (2010) 19:257–282
DOI 10.1007/s00778-009-0160-3

REGULAR PAPER

Revisiting the cube lifecycle in the presence of hierarchies

Konstantinos Morfonios · Yannis Ioannidis

Received: 25 October 2008 / Revised: 10 June 2009 / Accepted: 7 July 2009 / Published online: 19 September 2009

Abstract On-line analytical processing (OLAP) typically involves complex aggregate queries over large datasets. The data cube has been proposed as a structure that materializes the results of such queries in order to accelerate OLAP. A significant fraction of the related work has been on Relational-OLAP (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, as shown in this paper, the nature of hierarchies introduces several complications into the entire lifecycle of a data cube including the operations of construction, storage, indexing, query processing, and incremental maintenance. This fact renders existing techniques essentially inapplicable in a significant number of real-world applications and mandates revisiting the entire cube lifecycle under the new perspective. In order to overcome this problem, the CURE algorithm has been recently proposed as an efficient mechanism to construct complete cubes over large datasets with arbitrary hierarchies and store them in a highly compressed format, compatible with the relational model. In this paper, we study the remaining phases in the cube lifecycle and introduce query-processing and incremental-maintenance algorithms for CURE cubes. These are significantly different from earlier approaches, which have been proposed for flat cubes constructed by other techniques and are inadequate for CURE due to its high compression rate and the presence of hierarchies. Our methods address issues such as cube indexing, query optimization, and lazy update policies. Especially regarding updates, such lazy approaches are applied for the first time on cubes. We demonstrate the effectiveness of CURE in all phases of the cube lifecycle 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 and usage 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.

Keywords Data cube · Query processing · Incremental maintenance · Lazy update

1 Introduction

It is well known that business analysts and decision makers are not interested in individual data items but mainly focus on summaries generated by properly aggregating large collections of such data. Analysis of such summaries usually reveals new knowledge in the form of hidden trends and patterns that could then be exploited to obtain business advantage. The need for effectively supporting such analysis on a large scale has resulted in the advent of on-line analytical processing (OLAP) and data mining technologies [5], which have motivated a plethora of studies on efficient, accurate, and effective techniques to address relevant new challenges in data management.

Contemporary on-line analysis tools and spreadsheet applications implement features that allow users to explore

Work done while K. Morfonios was at the Department of Informatics and Telecommunications, University of Athens.

K. Morfonios (✉)
IBM Almaden Research Center, 650 Harry Road,
San Jose, CA 95120, USA
e-mail: kmorfon@us.ibm.com

Y. Ioannidis
Department of Informatics and Telecommunications,
University of Athens, Athens, Greece
e-mail: yannis@di.uoa.gr

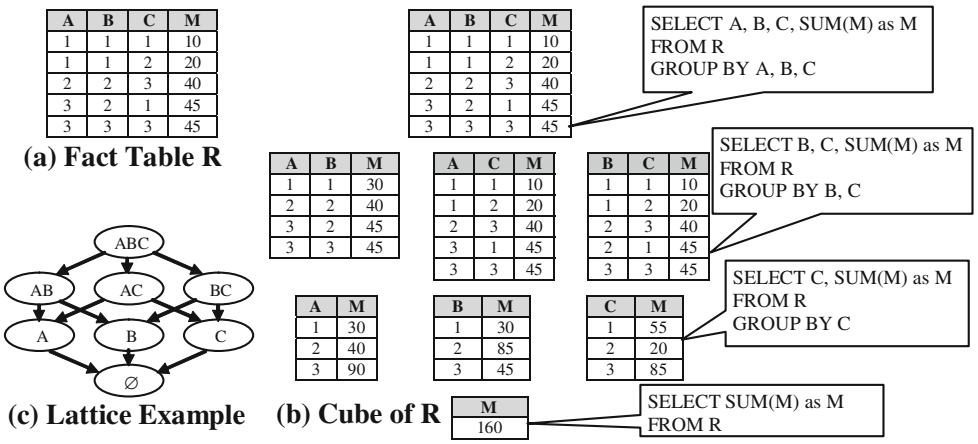

Fig. 1 Fact table R and its data cube

the underlying data, looking at it from different viewpoints and at different granularities, while interacting with it in an ad-hoc fashion by roll-up/drill-down operations. As an example, consider a business analyst of a shoe-store chain who sees that the total 2008 revenue for the chain was ⟨$10,200,000⟩. By drilling down, the analyst can see how this amount is split among states, e.g., ⟨CA, $2,000,000⟩, ⟨CO, $4,200,000⟩, and ⟨MT, $4,000,000⟩. Being disappointed by California sales, the analyst may further focus on this state and examine its sales revenue per season: ⟨CA, winter, $1,000,000⟩, ⟨CA, spring, $350,000⟩, ⟨CA, summer, $150,000⟩, and ⟨CA, fall, $500,000⟩. This detailed view indicates that sales are fine in winter, but drop during the other seasons, especially summer. In turn, this observation can lead to a new strategy that separates California from the other states and promotes different kinds of shoes there, given the warmer climate.

The data may also undergo some data mining analysis, e.g., to identify clusters of shoe-sales characteristics with similar revenue behavior. For instance, Colorado and Montana may cluster together regarding their overall yearly revenue, while at a more refined level, they may be similar only with respect to specific shoe types and this with a few months of delay for Montana. Such trends could lead to stock surplus transfers from Colorado to Montana but only for shoe types that are popular in both.

From a system's perspective, OLAP or data mining explorations such as the ones above generate large numbers of ad-hoc queries that make extensive use of grouping and aggregation in different subspaces of the multidimensional space of the problem concerned [9]. Unfortunately, on-the-fly aggregation over large volumes of data can be computationally very expensive; essentially, the success of OLAP and data mining techniques depends on the efficiency of the underlying systems in supporting aggregate queries in real-world settings.

To overcome the limitations and inefficiencies of traditional data systems that serve as underlying infrastructures for OLAP, Gray et al. [9] proposed the data cube: a structure that stores precomputed results of group-by aggregate queries on all possible combinations of the dimension-attributes over a fact table in a data warehouse. For example, Fig. 1a shows a fact table R and Fig. 1b shows the corresponding data cube (in uncompressed form). In this figure, we have arranged the group-by views that form the cube, called cube nodes, in positions reminiscent of the so-called cube lattice [12] (Fig. 1c), which indicates computational dependencies among different nodes. Every node in the cube lattice represents a group-by query of the form shown in Fig. 1b 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. Clearly, materializing the results of a large number of aggregate queries in a cube promises better response times and therefore a better infrastructure for OLAP.

1.1 Problem description

Intuitively, materializing the entire cube is ideal for fast access to aggregated data. Nevertheless, it poses considerable costs in computation and maintenance time, as well as in storage space. In order to overcome this problem and balance the tradeoff between query-response times and cube-resource requirements, implementation of the complete data cube has been studied using various data structures to construct and store the cube. Orthogonal to the issue of *how* to construct and store a cube is the issue of selecting *what* portion of the cube to materialize (i.e., which particular cube nodes to construct and store). Partial materialization of the cube is a popular topic in the related literature [12,27,33,37]. Combining partial materialization techniques with our work seems possible, but exceeds the scope of this work.

In general, existing data-cube implementation methodologies can be partitioned into four main categories, depending on the data structures and formats they use to compute and store a data cube. On the one hand, Relational-OLAP (**ROLAP**) and Multidimensional-OLAP (**MOLAP**) methods

use materialized views and multidimensional arrays, respectively, focusing mainly on efficient sharing of computational costs (like sorting or hashing) during cube construction. On the other hand, **Graph-Based** approaches exploit specialized graphs to compute and store cubes efficiently. Finally, **Approximation-Based** methods use various in-memory representations (e.g., histograms), inspired mainly from statistics.

With the exception of ROLAP, all the other methodologies generate cube formats that are inherently compact and allow fast data access:

- MOLAP methods store only aggregate values; the dimension values are stored only once and are indirectly implied by the coordinates of the position of cells in the multidimensional arrays used. This property compresses the cube [42] and allows fast access to aggregates, since it is known a-priori which cells to visit for answering any particular query.
- Graph-based methods exploit specialized structures that compress the cube by removing redundant information. At the same time, these structures work as indices as well and allow efficient manipulation of cube data [35].
- Approximation-based methods [39] use compact in-memory cube representations that inherently compress a cube at a price of some small loss of information and allow fast calculation of approximate answers.

Unlike their competitors, ROLAP methods are not inherently characterized by such compression features. They need some additional machinery to compress the cube and additional indices to manipulate it efficiently. Motivated by the above requirements, this paper focuses on ROLAP and attempts to stretch it to its limits when dealing with cubes in the presence of data hierarchies. The remaining methodologies are out of the scope of the paper, as their solutions cannot be incorporated into the ROLAP framework.

Existing ROLAP cubing algorithms have several weaknesses. A recent study [24] reveals that most of these algorithms focus mainly on construction and storage of flat cubes, i.e., cubes constructed over flat datasets. Issue 1: The lifecycle of a data cube does not involve off-line construction and storage only, but also every-day cube usage, including query answering and incremental maintenance. Issue 2: Real-world datasets are not always "flat" but are usually organized in hierarchies. For example, a dimension "Region" may contain values at different levels of detail, forming the hierarchy "City"→"Country"→"Continent". The first issue has been studied elsewhere [23], but without considering the additional complications introduced by hierarchies. On the other hand, the second issue has been studied with respect to construction and storage only [22], neglecting the remaining phases in the cube lifecycle. So, revisiting the cube lifecycle from a new perspective is mandatory to solve both issues in a comprehensive fashion; this is the main purpose of this paper.

Hierarchies are rather common in real-world applications and quite significant, as they offer great flexibility in describing the data at different granularities and form the basis for roll-up and drill-down operations [13]. However, as briefly mentioned above as well, their nature introduces several complications into all phases of the cube lifecycle that cannot be handled by straightforward extensions of existing techniques, rendering existing algorithms essentially inapplicable in a significant number of real-world applications [22]:

- The number of nodes in a cube lattice increases dramatically and its shape is more involved. For instance, consider a fact table with D dimensions. If L_i denotes the number of levels of the i-th dimension, the product $\prod_{i=1}^{D}(L_i+1)$ gives the total number of cube nodes. This product is greater than or equal to 2^D. Equality holds for flat data, i.e., when $L_i = 1, \forall\, i \in [1, D]$. For example, compare the number of nodes in Fig. 1c (flat dataset) with the number of nodes in Fig. 7 (hierarchical dataset).
- 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.
- The number of tuples that need to be stored in the cube increases dramatically.

To overcome these problems, this paper builds upon previous work on the CURE (**C**ubing **U**sing a **R**OLAP **E**ngine) algorithm [22] and develops comprehensive ROLAP solutions that address efficiently the entire cube lifecycle and can be implemented easily over existing relational servers. These CURE-based families of algorithms provide fast computation of a fully-materialized cube in compressed form, are incrementally updateable, and exhibit fast query-response times that can be improved by low-cost indexing and caching, even when dealing with very large datasets with arbitrary hierarchies. The efficiency of our methods is demonstrated through comprehensive experiments on both synthetic and real-world datasets, whose results have shown great promise for the performance and scalability potential of the proposed techniques, with respect to both the size and dimensionality of the fact table. Among the experimental results, we distinguish those that have made CURE the first ROLAP technique to complete the construction and usage of the cube of the highest-density dataset in the APB-1 benchmark (12 GB) [26]. 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 presence of hierarchies, indicating that ROLAP may not be inherently inferior.

1.2 Paper contribution and outline

With respect to cube construction, CURE contributes a novel lattice-traversal scheme, an optimized data-partitioning method, and a suite of relational storage schemes for all forms of redundancy [22]. Furthermore, as shown in this paper, the nature of hierarchies imposes additional challenges to the every-day usage of cubes as well, making existing techniques inapplicable for the manipulation of CURE cubes. In order to make CURE a comprehensive solution capable of treating efficiently all phases in the lifecycle of a cube even in the presence of hierarchies, in this paper, we further propose novel techniques for fast query answering and incremental maintenance over CURE cubes, providing answers to the aforementioned challenges as well. In more detail, our main contributions can be summarized as follows:

- **Query answering:** We develop an algorithm for answering arbitrary queries using an (unindexed) CURE cube and show that its practicality is limited in real-world applications that typically involve selective queries over large datasets. To overcome this, we investigate the effect of indexing on CURE cubes and propose an extension of the original algorithm that is based on low-cost indices. We show that indexing the entire cube, which is potentially very expensive in hierarchical data, is not necessary; thanks to the particular storage format of CURE cubes, indexing the original fact table only is enough for efficient query processing.
- **Query optimization:** We examine customized query optimization policies that use cost estimations to assess the benefits of using an index and identify which set of indices should be combined for a given query.
- **Incremental maintenance:** We study different approaches to incremental maintenance of CURE cubes and conclude that the common, eager tactics, which refresh a cube periodically during a dedicated window of time, are not efficient for CURE, due to its storage format and the nature of hierarchies. Alternatively, we propose a novel lazy method that performs only some lightweight operations during an update and modifies the actual data only on demand, during query processing. The additional cost at query time is marginal, while the amortized update cost is only a small fraction of that of the eager approach, making the lazy approach the method of choice. To the best of our knowledge, this is the first time a lazy method has been employed for cube processing. Finally, we propose a hybrid combination of the eager and the lazy method, which is very promising under certain conditions.

The rest of this paper is organized as follows: In Sect. 2, we offer a brief overview of related work. In Sect. 3, we mainly focus on the storage format of CURE cubes, originally presented elsewhere [22], which is necessary for the self-containment of this paper, since building efficient algorithms for accessing cube data strongly depends on the storage format. In Sect. 4, we study issues related to query processing and optimization over CURE cubes, and, in Sect. 5, we investigate incremental maintenance of CURE cubes and develop an eager, a lazy, and a hybrid solution. Our study on algorithms for efficient usage of CURE cubes is under the perspective of the additional challenges imposed by hierarchies. Finally, in Sect. 6, we present our conclusions and the directions of our future work.

2 Related work

Cube construction has been the focus of much research due to its importance in improving the performance of OLAP tools. After Gray et al. [9] proposed the data cube, a plethora of papers has been published in this area, which are well documented in the literature [24] and their detailed description exceeds our purpose.

There are several ROLAP cubing methods proposed so far [2,3,9,16,17,22,23,28,31,40]. Among them, BUC [3] is the most influential method attributing its success to a very efficient execution plan that enables sharing sorting costs during construction of different nodes. BU-BST [40], QC-DFS [17], and TRS-BUC [23] 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 [22] is BUC-based as well, while also dealing with all of these problems.

Moreover, among ROLAP cubing techniques other than CURE, only PipeSort and PipeHash [2,31] have (superficially) discussed supporting hierarchies. Both of them, however, represent rather straightforward and nonscalable 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 algorithms other than CURE assume that the original fact table fits in memory. Disk-based extensions have been discussed rarely [3,28], 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.

With respect to cube-size reduction in ROLAP, Key [16], BU-BST [40], QC- DFS [17], and TRS-BUC [23] 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 [3] 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.

Hence, regarding cube construction and storage, CURE seems to be the most promising algorithm overall in the ROLAP framework [22]. This fact justifies the requirement for some additional attention on its query and update performance, which is a main topic in this paper that extends previous work on CURE [22] in order to make it a comprehensive ROLAP solution that deals with all aspects of the cube lifecycle for the first time in the presence of hierarchies.

Regarding MOLAP methods, they use multidimensional arrays for cube construction and storage [32,42] as an alternative to relational materialized views. The structural differences between ROLAP and MOLAP are so marked, however, that any application of MOLAP techniques on CURE seems impossible.

A third category of cube-related algorithms includes methods that use graph-based structures for cube construction and storage [8,11,15,18,30,35,41]. These structures usually act as indices as well, generating fast query and update times with the use of customized algorithms. Among them, Dwarf [35] is the most promising, being able to deal with hierarchies [34] while removing several kinds of redundancy from cube data, which gives it polynomial scaling [36]. QC-Trees [18] have similar redundancy-reduction capabilities as well. CURE is the only solution in the ROLAP framework that shares some common properties with such sophisticated methods, including polynomial storage requirements. As explained above, a direct comparison of CURE with methods of this category goes beyond the scope of this paper.

Furthermore, approximation-based methods assume that decision support does not need 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, sampling, and histograms. Some examples can be found elsewhere [1,39]. Studying them further is beyond the scope of this paper, since we are interested in methods that produce accurate results.

Additionally, apart from the methods that construct complete cubes, there are methods that select subsets of nodes for partial construction and others that compute only some predefined nodes (e.g., [12,27,33,37]). As mentioned above, selecting *what* to materialize is orthogonal to deciding *how* to materialize it. Hence, although methods that perform partial aggregation are interesting and their combination with CURE seems possible, their study exceeds the scope of this paper.

Apart from the publications related to cube computation and storage, there is also some work on cube usage, i.e., on query answering using data stored in a cube, and on incrementally updating a data cube, following the updates of the fact table.

To the best of our knowledge, query processing over condensed ROLAP cubes has been studied for two algorithms, BU-BST [7,40] and TRS-BUC [23]. Interestingly, neither of them supports hierarchies. The solution proposed for the former is based on indexing the entire data cube. However, the additional cost of indexing all cube nodes is considerable and the situation would be even worse in the presence of hierarchies, since the number of nodes increases dramatically. On the other hand, TRS-BUC identifies and compresses some types of redundant tuples also identified by CURE. According to the terminology of CURE, originally defined elsewhere [22] and briefly described in Sect. 3.1, these redundant tuples are called Trivial Tuples (TTs) and are stored as row-id references pointing to tuples in the original fact table R. Based on this property of TTs, the creators of TRS-BUC have first proposed indexing only R, an idea beneficial for CURE as well to an even greater extent, since every tuple stored in CURE actually uses row-id references to R (not only TTs). Hence, here we apply essentially the same indexing technique, further enhanced by a query optimization method not discussed there. Specialized query optimization based on expected access costs is mandatory for CURE due to the existence of hierarchies, since many dimensions have small domains. As a consequence, a large number of tuples in R may have the same value in a particular dimension making the use of an index costly.

Regarding the incremental maintenance of a cube, most solutions follow the paradigm of Mumick et al. [25], who first proposed separating the process into two phases: *propagation*, during which the delta cube is constructed, and *refresh*, during which the original and the delta cube are merged. Recently, a general method for updating (uncompressed) cubes has been proposed [19], based on selecting particular nodes of the delta cube for construction during propagation. This method identifies propagation as the dominating factor during update and tries to optimize it. On the contrary, we have seen that constructing a compressed delta cube using CURE is very fast and that the dominating process is refresh; hence, although combining our methods with smarter propagation seems possible, here we focus on refresh. Interestingly, existing methods proposed for the refresh phase of condensed ROLAP cubes are not applicable in the case of CURE, due to the additional complications imposed by its high compression rate and mainly the existence of hierarchies. In particular, refreshing a BU-BST cube [7] is again based on indexing the entire cube, a solution already rejected as impractical for CURE cubes. Furthermore, refreshing QC-Tables (cubes constructed by QC-DFS) [20,21] or TRS-BUC cubes [23] is based on brute-force scanning or decompressing the entire cube, respectively; both

methods are very expensive, and the situation would be worse for the case of CURE cubes.

3 Cube storage

Any algorithm that uses data stored in a cube depends on the storage format produced by the algorithm that constructed the cube, since the storage format affects the access methods that can be applied. Hence, in this section, we provide an overview of the storage format produced by CURE, which has been originally presented elsewhere [22]. Based on it, in the following sections, we present specialized algorithms for querying and updating CURE cubes in the presence of hierarchies.

3.1 Storage format

First Kotsis and McGregor [16] and then several other researchers [8,17,18,23,35,40] 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 base-single-tuples (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* [22]. 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.

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 nonredundant data. CURE strikes on both factors, avoiding the storage of redundancy, while storing nonredundant 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 nonredundant tuples. Below, we describe CURE's efficient storage format of nonredundant data.

Most existing ROLAP methods that identify redundancy use a single D-dimensional relation for storing nonredundant data, which introduces a large number of NULL values for tuples that belong to nodes of lower dimensionality. Instead, CURE follows the example of TRS-BUC [23] and stores 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 Fig. 2, assuming that it consists of X dimensions and Y aggregates.

Fig. 2 Basic tuple format

Fig. 3 Normal and trivial tuple formats

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, CURE replaces all dimension values of t with a row-id reference (R-rowid), pointing to any $t_S \in S$ (Fig. 3a). 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. Assume for example a node consisting of a single two-byte dimension (say of type 'small integer'). If a row-id reference is four bytes long, the basic tuple format of Fig. 2 for this case is more compact than the format of Fig. 3a, since Dim1 would take 2 bytes/tuple, whereas R-rowid would take 4 bytes/tuple. 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 CURE treats them uniformly with the others. Moreover, note that CURE uses row-ids not only for the efficient storage of some tuples, as performed by TRS-BUC for the so-called totally-redundant tuples only [23], but for all tuples, since dimensional redundancy exists in every cube tuple.

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

Normal tuples (NTs): A tuple t is called normal if it is only dimensionally but not aggregationally redundant. The most compact format for NTs is the one of Fig. 3a, since CURE cannot avoid storing the aggregates. For example, if Fig. 4a shows the tuples stored in R and Fig. 4b shows the corresponding cube (in an uncompressed form), then tuple $\langle 3, 90\rangle$ in node A is an NT, since there is no other tuple in the entire cube with an aggregate value equal to 90.

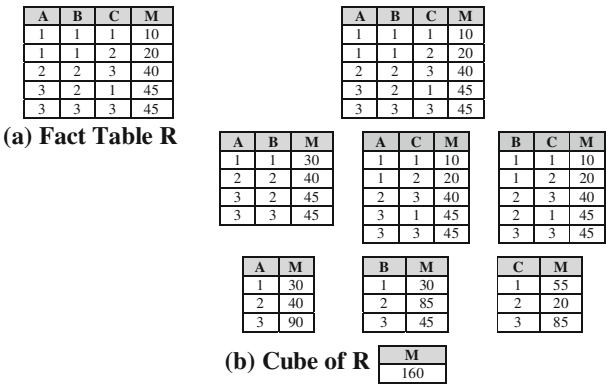

Fig. 4 Fact table R and its uncompressed cube

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, t is called 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 (Fig. 3b).

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 (an example of such a plan appears in Fig. 7), 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 subtree rooted at N_{LD}. In the example of Fig. 4b, all cube tuples with value A = 2 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 subtree rooted at A) and can be easily retrieved on demand.

Note that TTs are similar to BSTs and totally redundant tuples recognized by BU-BST [40] and TRS-BUC [23], respectively; however, they are stored far more efficiently.

Common aggregate tuples (CATs): A tuple t is called CAT, if it is aggregationally redundant and nontrivial ($|S| > 1$). 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:

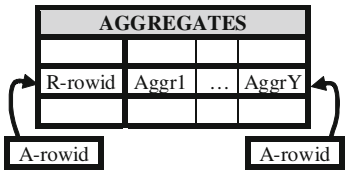

Fig. 5 Format of relation AGGREGATES and of CATs

- **Common source CATs** attribute equality of their aggregates to the fact that they have been produced by the same set of tuples of the fact table. In Fig. 4b, tuples ⟨1, 1, 30⟩ in AB, ⟨1, 30⟩ in A, and ⟨1, 30⟩ in B are common source CATs, since they have been produced by the same tuple set $S = \{\langle 1, 1, 1, 10\rangle, \langle 1, 1, 2, 20\rangle\}$ of R.
- **Coincidental CATs** are the CATs that have the same aggregates, although they have been produced by different tuple sets of the fact table. In Fig. 4b, tuples ⟨2, 85⟩ in B and ⟨3, 85⟩ in C are examples of coincidental CATs.

To avoid storing the aggregate values of CATs redundantly, CURE uses an additional relation AGGREGATES to store such common values only once and replaces all aggregate values in CATs with a row-id (A-rowid) pointing to the corresponding tuple in AGGREGATES. The specific schema chosen for AGGREGATES depends on the type of CATs that prevails. As explained elsewhere [22], the format shown in Fig. 5 produces a more compact cube, if common source CATs prevail. Since this is the most common case, in the rest of this paper we use this format. For more details on the other possible formats, please refer elsewhere [22].

For example, omitting the details of construction [22], the CURE cube of R in Fig. 4a appears in Fig. 6. Note that this cube contains the same information like the uncompressed cube of R in Fig. 4b, albeit stored in a much more compact format. In order to decompress 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 in relations R and AGGREGATES, respectively. Then, in the former case we only need to project every tuple fetched from R on the grouping attributes of the node queried, while in the latter we first need to follow an additional R-rowid found in every tuple fetched from AGGREGATES before that. For instance, node A (Fig. 6) has three tuples in total, one of every category. In order to decompress the tuple ⟨4, 90⟩ stored in A-NT, we fetch the tuple in R with R-rowid 4 (i.e., tuple ⟨3, 2, 1, 45⟩) and project it on A. Combining the resulting dimension value A = 3 with the aggregate value M = 90 stored in A-NT, we generate tuple ⟨3, 90⟩ (the third tuple of node A in Fig. 4b). Similarly, the tuple stored in A-TT indicates that we have to fetch the tuple with R-rowid 3 in R (i.e., tuple ⟨2, 2, 3, 40⟩). Since we restore a TT, we need to reconstruct the aggregate value as well; hence, we project the

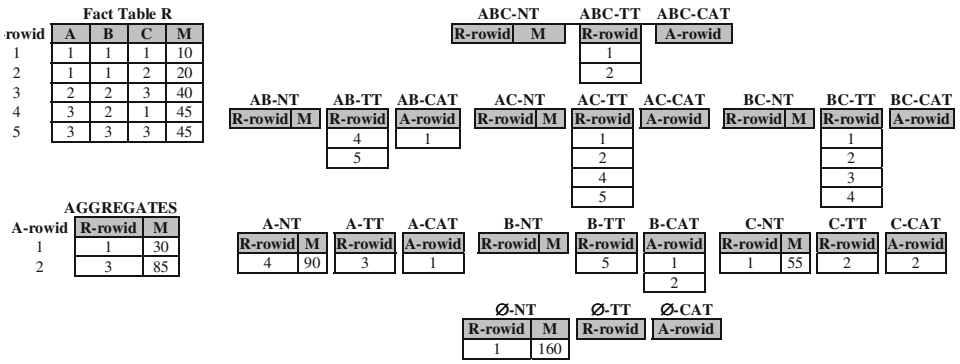

Fig. 6 Fact table R and its CURE cube

tuple fetched on both A and M and generate tuple ⟨2, 40⟩ (the second tuple of node A in Fig. 4b). Finally, the tuple stored in A-CAT indicates that we have to fetch the tuple with A-rowid 1 in relation AGGREGATES (i.e., tuple ⟨1, 30⟩). The latter specifies that we need to fetch the tuple with R-rowid 1 in R (i.e., ⟨1, 1, 1, 10⟩) and project it on A. Combining the resulting dimension value A = 1 with the aggregate value M = 30 found in the tuple fetched from relation AGGREGATES we generate tuple ⟨1, 30⟩ (the first tuple in node A of Fig. 4b).

Interestingly, CURE stores a TT only in the most specialized node N_S it belongs to and considers it shared among N_S and its ancestors in the cube lattice that have the grouping attributes of N_S as a prefix. In Fig. 6 for example, the 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 this TT physically, since this would be redundant. Therefore, during query answering over a node N, additionally accessing TTs of some of N's descendants is necessary. If N has X grouping attributes, then the total number of TT relations that need to be accessed is $X+1$. These are the TT relations of N itself and of all nodes that come up by removing the rightmost attribute iteratively. For example, restoring TTs of node ABC requires accessing relations ABC-TT, AB-TT, A-TT, and Ø-TT in Fig. 6.

Finally, in order to further enhance CURE's efficiency we can use several implementation variations. For example, if the ROLAP server supports bitmap indexing, which is common, we can change the format of relations TT and CAT without affecting 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. Furthermore, we have seen that it is beneficial to sort all row-ids in TT relations according to the order of the tuples they point at. This produces sequential scans during query answering. Our experiments have shown that such a sorting operation 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.

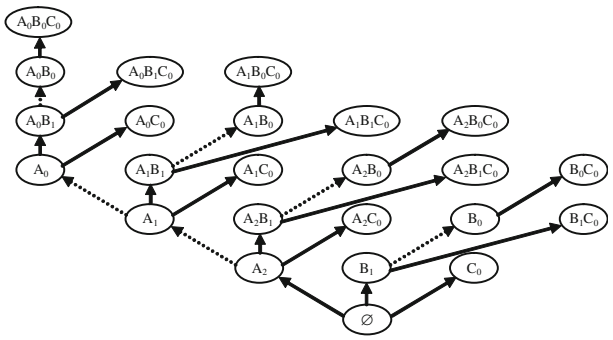

Fig. 7 The hierarchical execution plan of CURE

3.2 Construction and tuple classification algorithm

First constructing an uncompressed cube and then compressing it according to the aforementioned format would be inefficient. To avoid such performance penalties, CURE uses a specialized data structure, called *signature pool*, and a customized algorithm that classifies cube tuples into the proper class (NT, TT, or CAT) during construction [22]. Moreover, it applies several advanced techniques in order to deal with the challenges introduced by the nature of hierarchies [22]: (a) It traverses efficiently an extended lattice that includes dimension hierarchy levels, which enables pipelining and extensive sharing of sorting costs. Assume, for example, that the dimensions of the fact table R (Fig. 4a) are organized in hierarchies as follows: $A_0 \rightarrow A_1 \rightarrow A_2$, $B_0 \rightarrow B_1$, and C_0. Then, Fig. 7 presents the execution plan of CURE. (b) It introduces an efficient algorithm for partitioning fact tables that store hierarchical data of any size into memory-fitting segments. (c) It captures all types of redundancy and uses alternative schemes for storing nonredundant data efficiently, as described above.

Further details related to the CURE construction algorithm are beyond the scope of this paper and can be found elsewhere [22]. For the sake of self-containment, we simply repeat below the most indicative experimental results originally presented in the introductory publication of CURE [22].

In particular, we have experimented with the behavior of CURE and CURE+ under different conditions, and we have compared[1] their performance against the most efficient methods in ROLAP, namely BUC [3], BU-BST [40], BU-BST+ [7], and TRS-BUC [23]. CURE+ is a variation of CURE that applies a post-processing step to sort and replace row-ids with bitmap indices, as explained in Sect. 3.1. Furthermore, BU-BST+ is a variation of BU-BST that has actually been implemented as well at some point [7] and further incorporates the relatively straightforward storage of cube nodes as separate views instead of using a single monolithic relation for the entire cube. We have not implemented QC-DFS, another BUC-based algorithm proposed in the existing literature [17], since the relational representation of the so-called Quotient Cubes it constructs has been shown to have many problems [18]. This can be solved with the use of QC-Trees [18], but these are graph-based data structures and, hence, outside the scope of this paper. Note that, among the algorithms we have implemented, BUC identifies no redundancy, whereas BU-BST, BU-BST+, and TRS-BUC identify some redundancy, TTs in particular; nevertheless, none of them uses efficient storage for nonredundant data as CURE and CURE+ do.

We present the most indicative results of our experimental evaluation related to cube construction and storage. Note that, in the following analysis of the experimental results, we mainly focus on the behavior of CURE and CURE+. An extensive analysis of the behavior of the other algorithms is beyond the scope of this paper and can be found elsewhere [23].

Flat cubes: In our first set of experiments we have evaluated the efficiency of all algorithms in constructing flat cubes. We have experimented with two widely used real-world datasets, namely CovType [4] and Sep85L [10]. CovType describes forest cover-type data and consists of 10 dimensions and 581,012 tuples. Its dimensions and their cardinalities are as follows: Horizontal-Distance-To-Fire-Points (5,827), Horizontal-Distance-To-Roadways (5,785), Elevation (1,978), Vertical-Distance-To-Hydrology (700), Horizontal-Distance-To-Hydrology (551), Aspect (361), Hillshade-3pm (255), Hillshade-9am (207), Hillshade-Noon (185), and Slope (67). Sep85L describes surface synoptic weather reports and consists of 9 dimensions and 1,015,367 tuples. Its dimensions and their cardinalities are as follows: Station-Id (7,037), Longitude (352), Solar-Altitude (179), Latitude (152), Present-Weather (101), Day (30), Weather-Change-Code (10), Hour (8), and Brightness (2). In both datasets, we have arranged the dimensions in a decreasing cardinality order, for greater efficiency, as proposed elsewhere [3] and verified in early experimentation.

[1] Since, among these methods, CURE and CURE+ are the only ones that support hierarchies, such a comparison is meaningful on flat datasets only.

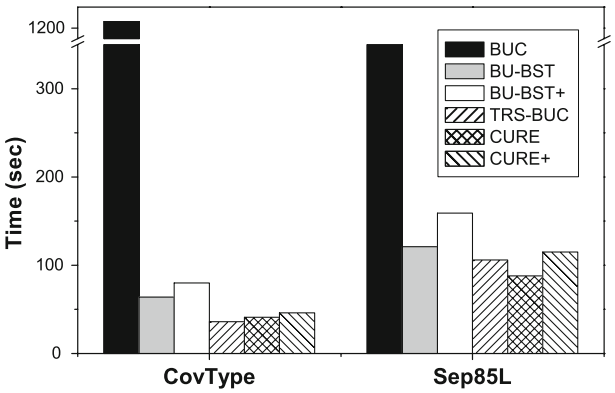

Fig. 8 Construction time (real datasets)

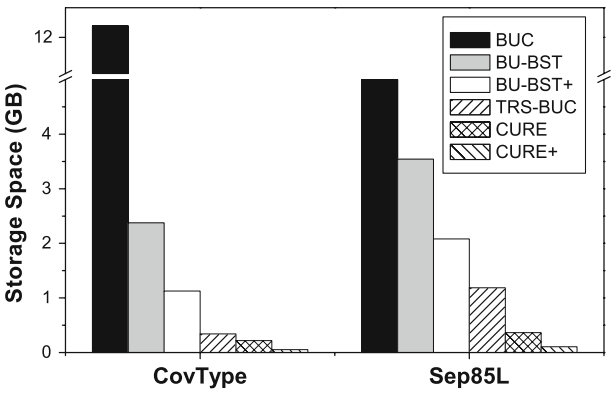

Fig. 9 Storage space (real datasets)

Figure 8 shows the time spent on the construction of the corresponding cubes, and Fig. 9 their storage space requirements. Note that in both datasets the size of the cubes constructed by CURE and especially CURE+ is much smaller than the sizes of the cubes of the other formats. Interestingly, the size of CURE+ cubes is approximately six to ten times (10–20 times, respectively) smaller than the size of the corresponding TRS-BUC cubes (BU-BST+ cubes, respectively). CURE and CURE+ attribute their storage efficiency to that they remove all types of redundancy and mainly to the efficient schema they use for nonredundant data. With respect to time, CURE seems to be very close to TRS-BUC. There are essentially two contradictory factors that decide their relative performance. On the one hand, TRS-BUC is simpler; it identifies only some types of redundancy, and does not spend any time on operations like sorting signatures stored in a signature pool, as CURE does. Such lightweight behavior gives an advantage to TRS-BUC. On the other hand, TRS-BUC stores a larger cube; therefore, it pays greater output costs. Such behavior gives it a disadvantage. The relative impact of these contradictory factors makes TRS-BUC a little faster than CURE in CovType, but a little slower in Sep85L.

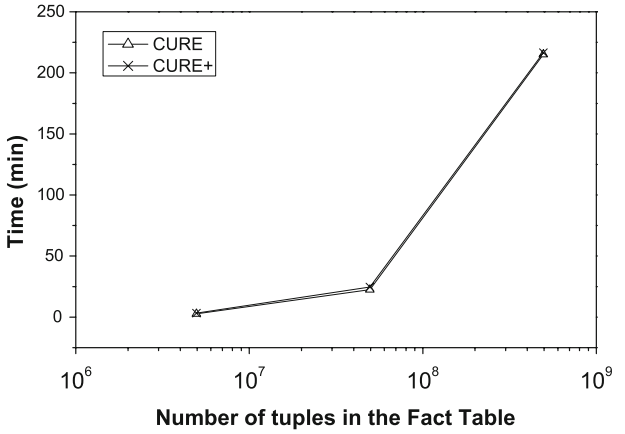

Fig. 10 Construction time (APB-1)

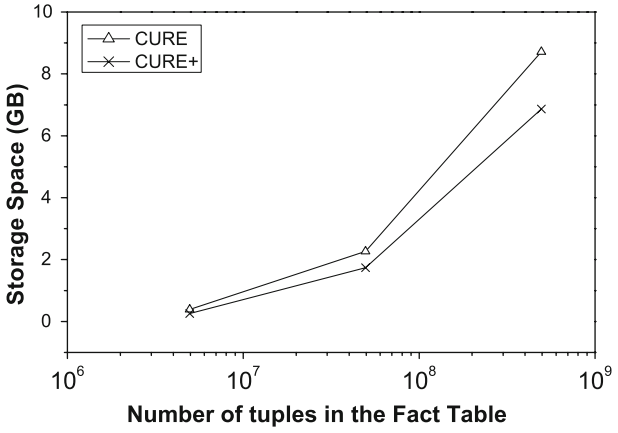

Fig. 11 Storage space (APB-1)

Moreover, expectedly, CURE+ is always a little slower than CURE, paying the additional penalty of the post-processing step. Nevertheless, the small penalty in construction time is compensated by great storage savings (also by greater efficiency in answering queries, as shown in the next section); thus, we consider it negligible compared to the improvements it offers.

Hierarchical cubes: In this set of experiments, we have evaluated the efficiency of CURE/CURE+ in constructing hierarchical cubes. Recall that the other BUC-based methods that we have used for comparison above do not support hierarchies; hence, they are omitted. The datasets we have used are synthetic and have been produced by the data generator of the APB-1 benchmark [26], which is a standard in OLAP [38]. 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 for the highest density factor are 400 times larger (495,720,000 tuples and 12 GB). The total number of nodes in the cube is (6+1)×(2+1)×(3+1)×(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.

Figures 10 and 11 show the construction time and the storage space, respectively, for a low (0.4), a medium (4), and the highest possible (40) density factor. The values along the *x*-axis, which is logarithmic, indicate the number of tuples in the corresponding fact tables. Evidently, both CURE and

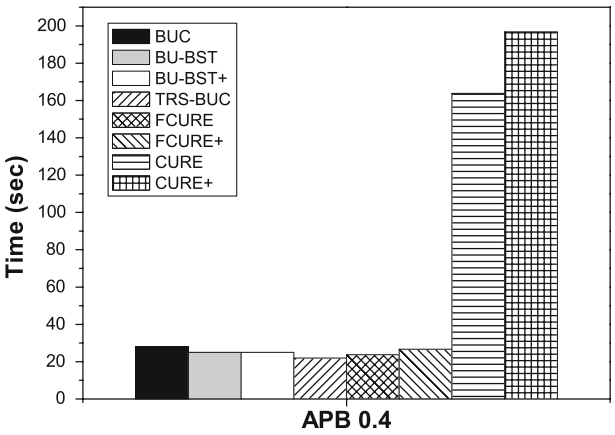

Fig. 12 Flat versus hierarchical cube: construction time

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 an APB-1 dataset in its highest density factor in approximately 3.5 h using very limited resources (256 MB of memory) is impressive. With respect to 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).

Moreover, we have investigated the tradeoffs between constructing flat (only at the finest level of detail) and hierarchical cubes over hierarchical data. The dataset we have used is APB-1 with density factor 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 (Fig. 12) and occupies less storage space (Fig. 13); however, as we will show in Sect. 4.2, a hierarchical cube offers greater advantages in answering roll-up/drill-down queries fast. Hence, the overall tradeoff seems to be in favor of hierarchical cubes.

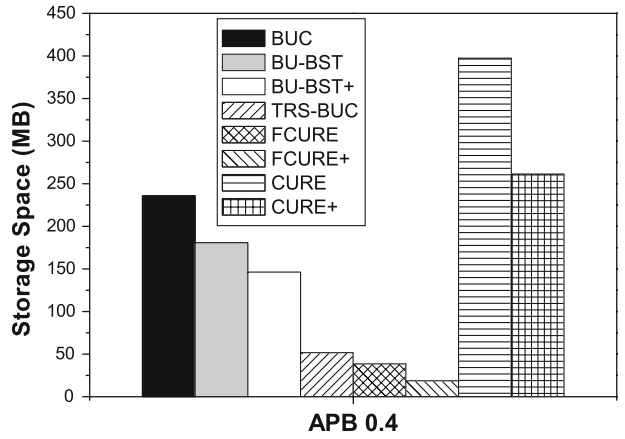

Fig. 13 Flat versus hierarchical cube: storage space

4 Query answering

In the previous section, we focused on the highly compressed storage format produced by CURE, a ROLAP algorithm that constructs efficiently complete cubes over large datasets with arbitrary hierarchies. In order to make CURE a comprehensive solution capable of dealing with all phases in the cube lifecycle, in this and the following section, we study algorithms for efficient usage of CURE cubes, i.e., algorithms for query answering and incremental maintenance. Interestingly, as shown below, existing techniques for cube usage are not practical if applied over CURE cubes, due to CURE's high compression rate and mainly due to the additional challenges imposed by the existence of hierarchies. In order to overcome this problem, in this section, we study customized methods for efficient queries on top of CURE cubes, and in the following section, we study novel update algorithms, including some lazy policies never applied in techniques associated with cubes before, and demonstrate the effectiveness of our solutions through experiments on both real-world and synthetic datasets.

4.1 Querying unindexed CURE cubes

Based on the storage format of CURE presented in Sect. 3.1, we develop an algorithm for answering group-by queries on top of unindexed CURE cubes (Algorithm 1), namely QU-CURE (Query Unindexed CURE). A general form of a group-by query that can be answered using a data cube is the following:

SELECT S_1, S_2,..., S_n, *f(M)*
FROM R
WHERE W_1 op_1 v_1 *AND ...AND* W_k op_k v_k
GROUP BY S_1, S_2,..., S_n
HAVING f(M) op v

Algorithm 1 QU-CURE(R, AGG/TES, N, Cond)

```
1: for each tuple t in relation N-NT do
2:   if t.aggr satisfies Cond then
3:     Fetch tuple t_R from R indicated by t.R-rowid;
4:     if t_R.dim satisfies Cond then
5:       t_out.dim = Project(t_R, N);
6:       t_out.aggr = t.aggr;
7:       Output(t_out);
8:     end if
9:   end if
10: end for
11: for each descendant node N_D of N (including N itself) whose
    grouping attributes are a prefix of the grouping attributes of N do
12:   for each tuple t in relation N_D-TT do
13:     Fetch tuple t_R from R indicated by t.R-rowid;
14:     if t_R.aggr satisfies Cond and t_R.dim satisfies Cond then
15:       t_out.dim = Project(t_R, N);
16:       t_out.aggr = t_R.aggr;
17:       Output(t_out);
18:     end if
19:   end for
20: end for
21: for each tuple t in relation N-CAT do
22:   Fetch tuple t_A from AGGREGATES indicated by t.A-rowid;
23:   if t_A.aggr satisfies Cond then
24:     Fetch tuple t_R from R indicated by t_A.R-rowid;
25:     if t_R.dim satisfies Cond then
26:       t_out.dim = Project(t_R, N);
27:       t_out.aggr = t_A.aggr;
28:       Output(t_out);
29:     end if
30:   end if
31: end for
```

In this query, f is an aggregate function, assumed to be identical to the aggregate function used in the cube construction. Also $S = \{S_1, \ldots, S_n\}$ is the subset of dimensions of the original fact table R participating in the GROUP BY clause and $W = \{W_1, \ldots, W_k\}$ is the subset of the ones participating in the WHERE clause. Clearly, the most specialized node N that needs to be accessed for such a query is the one with grouping attributes $S \cup W$. If $W \subseteq S$, then $S \cup W = S$, so the node with S as its grouping attributes holds all information necessary to answer the query and no aggregation needs to be performed at query time. Otherwise, selection on the dimensions in W must be performed on the node with $S \cup W$ as its grouping attributes and the tuples selected must be aggregated and projected on S to produce the result. In our study, we assume that the dimension set in the WHERE clause of a given query is a subset of the dimension set in the SELECT clause ($W \subseteq S$); otherwise, a step of postaggregation would be necessary. We omit this step, since it adds nothing to the intuition of the algorithm. The input parameters of QU-CURE consist of the fact table R, the relation AGGREGATES, the cube node queried N, and an expression Cond that indicates selection conditions, as expressed by the WHERE and HAVING clauses in SQL syntax.

QU-CURE consists of three phases, one for each category of tuples. In the first phase (lines 1–10) it processes NTs by accessing every tuple t in the NT relation of N (line 1) and checking whether the aggregate values of t (t.aggr) satisfy the condition Cond (line 2). If they do, the algorithm fetches the tuple t_R from R indicated by the row-id stored in t (t.R-rowid) (line 3) and then, it checks whether the dimension values of t_R (t_R.dim) satisfy the condition Cond (line 4). If the answer is positive, the algorithm has found a new tuple (t_{out}) that qualifies the selection criteria imposed by the original query and hence writes it in the output (line 7). The dimension values of t_{out} come from the projection of t_R on the grouping attributes of N (line 5), while the aggregate values from the normal tuple t (line 6).

In the subsequent phases the algorithm processes TTs (lines 11–20) and CATs (lines 21–31). The steps are similar to these of phase 1; hence, we do not explain them here in detail. Just note that, in phase 2, the algorithm accesses TTs not only in the TT relation of node N, but also in the TT relations of all of its descendants in the cube lattice whose grouping attributes are a prefix of the grouping attributes of N (line 11). As described in Sect. 3.1, this is attributed to that CURE stores TTs only in the most specialized node they belong to.

4.2 Experimental evaluation for the unindexed case

Clearly, the performance of cube usage is tightly coupled with the underlying format used for cube storage. In order to evaluate the quality of the storage format of CURE with respect to query answering and the efficiency of the proposed technique for answering queries over CURE cubes, we have implemented QU-CURE, and compared the query-response times it generates with the query-response times generated by the cube formats of the best algorithms among CURE's ROLAP competitors, namely BUC [3], BU-BST [40], BU-BST+ [7], and TRS-BUC [23]. These are the same algorithms also used for comparison in Sect. 3.2. In this subsection, we present the results of our experimental evaluation on answering node queries, which are queries that contain no WHERE part in their SQL syntax. Clearly, such queries cannot be accelerated by indexing. More selective queries, which contain a WHERE part and can be accelerated by indices, will be the subject of the following subsections.

Flat cubes: In our first set of experiments we have evaluated the efficiency of the cube formats generated by the aforementioned methods in query answering. Such an evaluation is clearly 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. Once more we have experimented with CovType [4] and Sep85L [10], the two real-world datasets

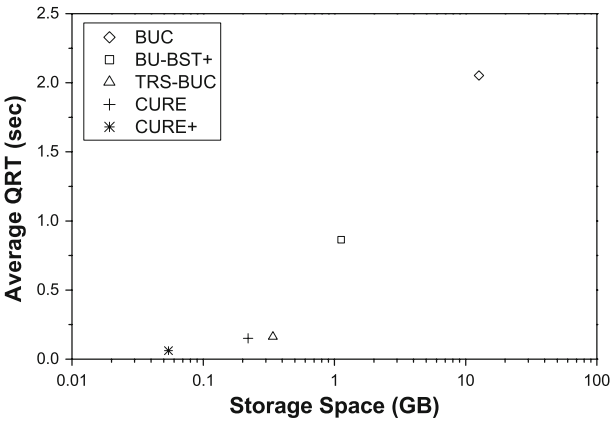

Fig. 14 Average QRT versus storage space (CovType)

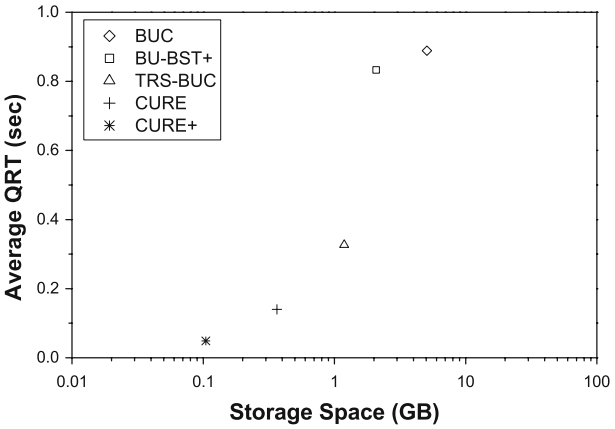

Fig. 15 Average QRT versus storage space (Sep85L)

we have also used in Sect. 3.2. Figures 14 and 15 show the average query-response times of all algorithms for the aforementioned workloads with respect to the storage-space requirements of each algorithm (presented in a logarithmic scale). In other words, these figures indicate query performance as a function of the resources consumed by the different algorithms. In these figures, we have excluded BU-BST, since it resides outside the scale of the graphs. Clearly, these graphs indicate that CURE and CURE+ are the undisputed winners, since they achieve the best query-response times, while consuming remarkably less storage resources. In more detail, CURE and especially CURE+ exhibit the best performance with respect to query answering, outperforming TRS-BUC, the method of choice so far in the existing literature [23]. The reason is that CURE uses row-ids for all types of tuples, not only for TTs, as TRS-BUC does. Hence, it redirects all disk accesses (not only some) to a single relation and can, therefore, exploit caching to a greater extent (the effect of caching has been studied elsewhere [22]). CURE+ is even better, because it stores row-ids in a sorted fashion, producing sequential scans and greater locality of reference. Hence, we conclude that its great storage savings and

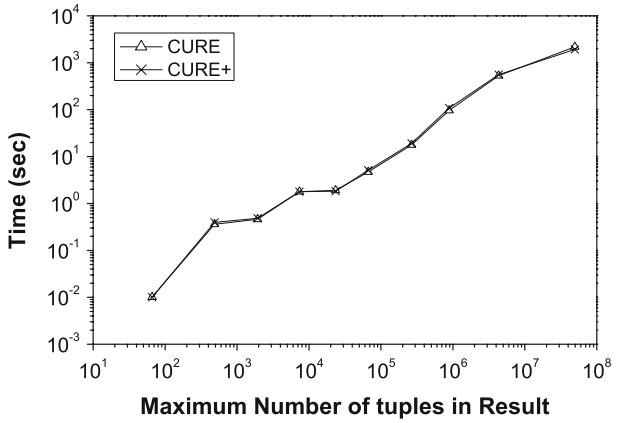

Fig. 16 Average QRT (APB-1 density factor 4)

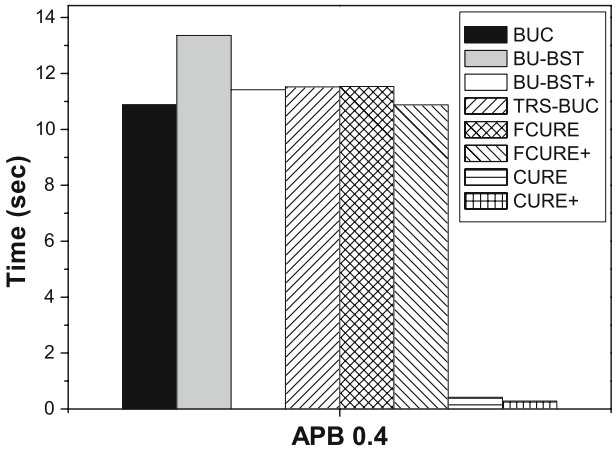

Fig. 17 Flat versus hierarchical cube: average QRT

efficiency in answering queries compensates the small penalty in construction time.

Hierarchical cubes: In this set of experiments, we have evaluated the efficiency of the storage formats of CURE and CURE+ in query answering over hierarchical cubes. Recall that the other methods used for comparison above do not support hierarchies. The datasets we have used have been once again produced by the data generator of the APB-1 benchmark [26], which is a standard in OLAP [38], as explained in Sect. 3.2.

Figure 16 illustrates the average query-response times for CURE and CURE+ 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. Both axes are logarithmic; this fact hides the advantage of CURE+ over CURE.

Note that both CURE and CURE+ cubes take less than 1 s on average to answer 30% of all node queries possible and less than 10 seconds for 60% (that return up to 10^5 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 h to answer 20 small and moderate queries, 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). The study of such range queries is the topic of the following subsections. 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.

Finally, we have further investigated the tradeoffs between constructing flat (only at the finest level of detail) and hierarchical cubes over hierarchical data (Fig. 17). The dataset we have used is APB-1 with density factor 0.4, which fits in memory. FCURE is the version of CURE that generates flat cubes ignoring hierarchies, as also explained in Sect. 3.2. Clearly, although the construction of a flat cube is faster and occupies less storage space (as shown in Figs. 12 and 13, respectively, in Sect. 3.2), a hierarchical cube offers greater advantages in answering roll-up/drill-down queries. The corresponding response times produced by querying the hierarchical cubes are approximately two orders of magnitude faster than the response times over the other formats (Fig. 17). Such behavior gives us strong evidence that the overall tradeoff is in favor of hierarchical cubes, since query answering is associated with the every-day usage of a data cube, whereas cube construction is an off-line procedure.

4.3 Querying indexed CURE cubes

In Sect. 4.1, we presented an algorithm for query answering over CURE cubes that relies on brute-force scanning of tuples that belong to the queried node N without using indices in order to accelerate selective queries. Clearly, as also shown in the experimental evaluation of Sect. 4.2, this algorithm can be fast only if N contains a small number of tuples, which induces a small number of disk seeks, or if the fact table R is small enough, which enables caching a considerable portion of it in main memory. However, such conditions do not always hold in real-world applications; hence, we need to develop a better solution, which is the topic of the rest of this section.

In order to reduce I/O costs during query answering, we have to adopt the use of indices for filtering out tuples that do not satisfy selection criteria without accessing them. However, indexing an entire cube (performed by others [7]) is not a panacea due to the considerable overhead it adds in terms of computational and storage costs, as also shown elsewhere [23]. Note that the overhead of indexing an entire cube becomes even greater when data is organized in hierarchies, because, in this case, the number of cube nodes increases dramatically. Hence, our primary design goal is to accelerate arbitrary selective queries over a CURE cube without building indices on top of every node. Moreover, we wish to maintain the advantages of ROLAP compatibility of CURE by using common index structures implemented in relational engines, e.g., B^+-Trees and bitmap indices.

Towards this end, a careful study of QU-CURE (Algorithm 1) indicates that the most expensive operation during query answering over a CURE cube involves fetching tuples from the fact table R (lines 3, 13, 24). Clearly, R is considerably larger than any cube node, since it holds at least as many tuples as they do (and usually many more), whose size in bytes is several times larger than any cube tuple, given that CURE stores tuples compactly (Fig. 6). Thus, randomly accessing tuples in R can generate long disk seeks. Note that due to that CURE rejects dimensional redundancy from the cube, as described above, restoring any cube tuple requires accessing R, which makes such expensive operations very common and threatens performance overall.

Based on that the main bottleneck in query answering over CURE cubes is the access of tuples in R, we propose indexing solely R (in the spirit of TRS-BUC [23]) in order to filter out early during query processing a large number of such operations. By doing so, we achieve our primary goal to accelerate arbitrary selective queries over any node, without indexing the entire cube. Actually, this solution does not index the cube, but only R, which is not a part of the cube. Interestingly, the fact that R may be already indexed for other purposes generates potentials for efficient reuse of existing resources. Note that indexing R in the case of TRS-BUC cubes accelerates only the access of TTs (called totally-redundant tuples in the context of TRS-BUC), since they are the only type of tuples substituted with row-ids in the corresponding format [23]. On the contrary, the advanced storage format of CURE uses row-ids for all types of tuples; hence, the benefits from indexing only R are expectedly larger for CURE cubes.

For indexing the fact table, we can use any ROLAP compatible method. In this paper, we have used a method, which has been found efficient elsewhere [23]. Figure 18 illustrates an example of indexing R (the fact table in Fig. 4a), based on it. According to this solution, if D is the number of dimensions of R, we construct D separate B^+-Trees, one for each dimension. This enables later combining any subset of them for answering arbitrary queries over any cube node. (The use

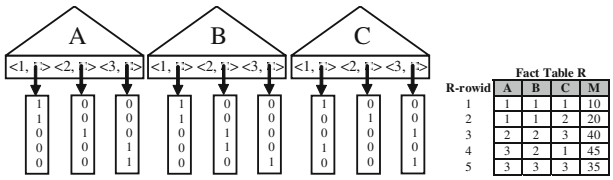

Fig. 18 Example of indexing R

of such combinations is the topic of the following subsection). At the leaf level of the B^+-Tree of dimension Δ, we associate every value δ_i in the domain of Δ with a bitmap vector indicating the tuples in R with $\Delta = \delta_i$. We physically store such bitmap vectors in a compressed format. For more details, please refer elsewhere [23].

Algorithm 2 QI-CURE(R, AGG/TES, N, Cond, *RS*)

```
1: for each tuple t in relation N-NT do
2:   if t.R-rowid belongs to RS and t.aggr satisfies Cond then
3:     Fetch tuple t_R from R indicated by t.R-rowid;
4:     if t_R.dim satisfies Cond then
5:       t_out.dim = Project(t_R, N);
6:       t_out.aggr = t.aggr;
7:       Output(t_out);
8:     end if
9:   end if
10: end for
11: for each descendant node N_D of N (including N itself) whose
    grouping attributes are a prefix of the grouping attributes of N do
12:   for each tuple t in relation N_D-TT do
13:     if t.R-rowid belongs to RS then
14:       Fetch tuple t_R from R indicated by t.R-rowid;
15:       if t_R.aggr satisfies Cond and t_R.dim satisfies Cond then
16:         t_out.dim = Project(t_R, N);
17:         t_out.aggr = t_R.aggr;
18:         Output(t_out);
19:       end if
20:     end if
21:   end for
22: end for
23: for each tuple t in relation N-CAT do
24:   Fetch tuple t_A from AGGREGATES indicated by t.A-rowid;
25:   if t_A.R-rowid belongs to RS and t_A.aggr satisfies Cond then
26:     Fetch tuple t_R from R indicated by t_A.R-rowid;
27:     if t_R.dim satisfies Cond then
28:       t_out.dim = Project(t_R, N);
29:       t_out.aggr = t_A.aggr;
30:       Output(t_out);
31:     end if
32:   end if
33: end for
```

Based on the above, we have extended QU-CURE (Algorithm 1) and developed QI-CURE (Query Indexed CURE, Algorithm 2), an algorithm that benefits from low-cost indexing over the fact table R. Note that the extended algorithm uses an additional input parameter *RS*, which is a superset of the row-ids of the tuples in R that satisfy condition Cond. The construction of *RS* is based on the use of indices

over R and will be further examined in the following subsection. For the moment, let us take its existence for granted.

The differences between QI-CURE (Algorithm 2) and QU-CURE (Algorithm 1) exist in lines 2, 13, and 25 of the former. In these lines, QI-CURE additionally checks if fetching a tuple t_R from R can generate an output tuple that potentially satisfies the condition Cond. Note that it performs this test without fetching t_R, based only on the information of *RS*. If the R-rowid of t_R is not a member of *RS*, the potential output tuple that would be generated by fetching t_R would not satisfy Cond, by definition of *RS*. In this case, the algorithm does not fetch t_R, saving an unnecessary disk seek operation. In other words, QI-CURE exploits the additional information of *RS* in order to prune early during its execution the usually expensive access of tuples that do not qualify the selection criteria of a given query. Recall that earlier we have identified such access costs as a main bottleneck in query answering over CURE cubes. Interestingly, the set *RS* can be efficiently represented as a bitmap vector, where zero (one) values denote that the corresponding tuple in R does not (may) satisfy Cond. This representation of a set is both compact and offers efficient containment tests with the use of simple bitwise operations.

4.4 Query optimization issues

Taking into account the particular structures described in the previous subsection for efficiently indexing a fact table R in order to accelerate queries over the corresponding CURE cube, in this subsection, we study query-optimization issues customized for CURE and propose an algorithm for constructing *RS*, the aforementioned input parameter of algorithm QI-CURE (Algorithm 2).

Assume a group-by query Q (of the form presented in Sect. 4.1) over a node N, and let the WHERE part in the SQL syntax of Q look as follows:

WHERE $(\Delta_1\ op_1\ \delta_1)$ *AND* $(\Delta_2\ op_2\ \delta_2)$ *AND...AND* $(\Delta_k\ op_k\ \delta_k)$

In this expression Δ_i $(i \in [1,k])$ denotes a dimension of R, op_i is a comparison operator (e.g., $>$, $=$, $<$), and δ_i a value in the domain of Δ_i. Furthermore, let the predicates $(\Delta_i\ \text{op}_i\ \delta_i)$ be ordered in a decreasing selectivity order, i.e., let the most selective predicate (the one that filters out the largest number of tuples) be first and so on (this may require query rewriting). Moreover, suppose that |N| is the number of tuples in node N, *CR* the average cost of fetching a tuple from R, and S_i the selectivity factor of the i-th predicate, i.e., an estimation of the fraction of tuples in R that satisfy the condition it expresses. Finally, let I_i be an estimation of the cost of accessing the index of R that corresponds to dimension Δ_i in order to find the set of row-ids of the tuples in R that satisfy the condition expressed by the i-th predicate. Then, the cost C_0 of fetching from R all tuples indicated by R-rowids stored in N without using any index is given by the formula $C_0 = |\text{N}| \times CR$. Similarly, the cost C_1 of fetching from R only the tuples indicated by R-rowids in N that satisfy the first predicate is given by the formula $C_1 = |\text{N}| \times CR \times S_1 + I_1$. Generally, if we define $I_0 = 0$, $S_0 = 1$, and assume that data is uniformly distributed and that values in different dimensions are independent, the cost C_i of fetching from R only the tuples indicated by R-rowids stored in N that satisfy the first i predicates ($i \in [0,k]$) is given by formula (4.1).

$$C_i = |\text{N}| \times CR \times \prod_{n=0}^{i} S_n + \sum_{n=0}^{i} I_n \tag{4.1}$$

Note that the first term in formula (4.1) decreases with i, since $S_i \leq 1$, whereas the second one increases, since $I_i \geq 0$. Our prototype optimizer uses formula (4.1) in order to estimate the value $i \in [0,k]$ that minimizes C_i and, equivalently, the indices that we need to access for optimally answering the given query Q. If RS_0 is a bitmap vector of |R| bits all set to 1, RS_i is a bitmap vector whose m-th bit is set to 1 if the m-th tuple in R satisfies the condition expressed by the i-th predicate, and our optimizer indicates that the number of indices that must be accessed is n, then the input parameter *RS* of QI-CURE is given by formula (4.2).

$$RS = \bigcap_{i=0}^{n} RS_i \tag{4.2}$$

Finding the individual RS_i using the index structures proposed in the previous subsection is straightforward and computing their intersection is easy using bitwise operations. After the discussion in this subsection, let us repeat the definition of *RS* given above: *RS is a superset of the row-ids of the tuples in R that satisfy condition Cond*. Now, it should be clear why *RS* is generally a "superset of" and not "the set of". It is because we expect that usually $n < k$. *RS* becomes exactly the set of the row-ids of the tuples in R that satisfy Cond when the optimizer decides that $n = k$.

Note that superficially setting $n = k$ without using the aforementioned formulas may be satisfactory for flat cubes, because the domains of the dimensions are usually large and hence S_i and I_i are usually minor for any $i \in [1,k]$ keeping the cost C_i relatively small. Nevertheless, when hierarchies exist, the domains of the dimensions are usually small, especially at the higher hierarchy levels. A small domain implies that a large number of tuples in R have the same value in a given dimension, increasing both S_i and I_i, and hence C_i as well. The potential large cost of C_i explains why the use of the aforementioned formulas is mandatory for CURE, which is designed for hierarchical cubes as well.

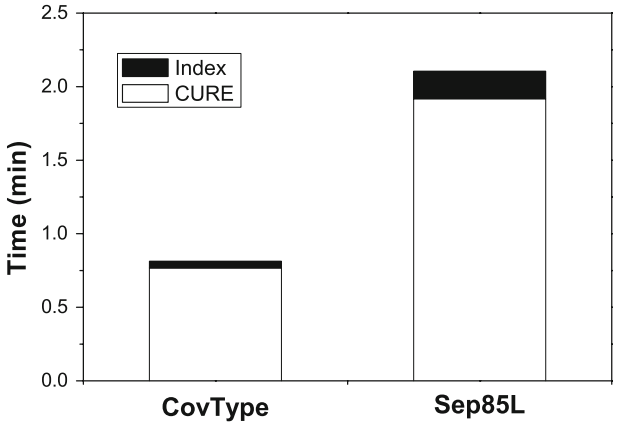

Fig. 19 Construction time (real datasets)

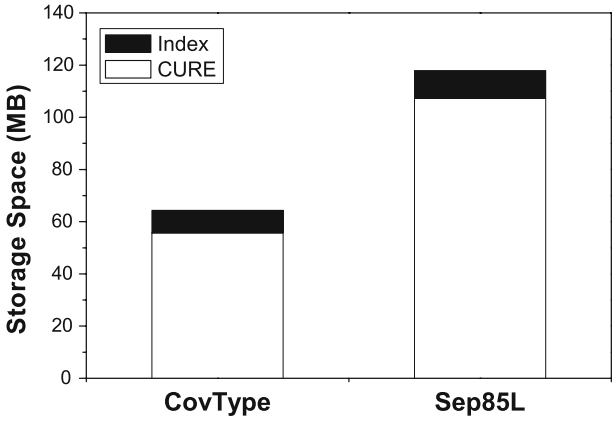

Fig. 20 Storage space (real datasets)

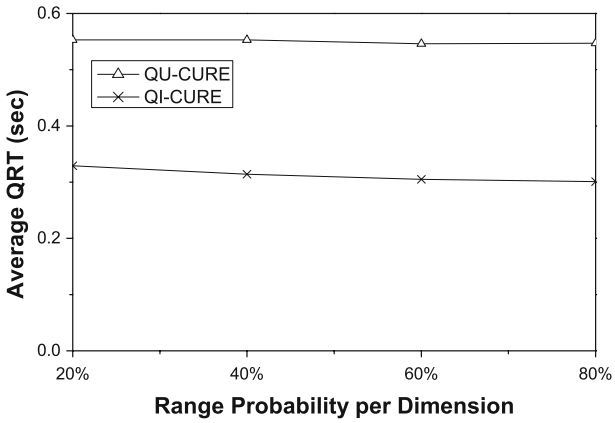

Fig. 21 Average QRT (CovType)

4.5 Experimental evaluation for the indexed case

In order to evaluate the efficiency of the proposed method for answering queries over an indexed CURE cube, we have implemented QI-CURE and compared it with QU-CURE. In this subsection, we have excluded the other BUC-based algorithms from our experimental evaluation for the following reasons:

- None of them supports hierarchies.
- Most of them (the exception is TRS-BUC) are based on indexing the entire cube.
- All of them have already been found essentially inferior to CURE both in cube construction and in query answering over unindexed cubes.

Furthermore, since we have found that the version of CURE called CURE+ is in most cases preferable, hereafter we focus on it. In this new context, we have decided to omit the symbol "+" and use the name CURE for CURE+, for the sake of simplicity. Below, we present the most indicative results of our experimental study. We omit the rest for the sake of brevity.

Flat cubes: Figure 19 illustrates the time required for constructing and indexing CURE cubes over the two real-world datasets used throughout this paper, namely CovType and Sep85L. The white part at the bottom of each column in this figure depicts the cube-construction time, while the black part at the top shows the additional time spent on indexing the corresponding fact table. Note that the cube-construction algorithm that generates the white part has been the topic of another publication [22] briefly presented in Sect. 3.2; hence, we should mainly focus on the black part, which is generated by the indexing algorithm proposed for CURE cubes in Sect. 4.3. Similarly, Fig. 20 illustrates the storage-space requirements of each CURE cube and of the corresponding indices. Again, the white part at the bottom of each column represents the storage space necessary for materializing the cube itself, while the black part at the top shows the additional space required for storing the index over the corresponding fact table. Clearly, both figures indicate that the resources required for cube construction and storage dominate the additional resources required for indexing the fact table. This gives strong evidence that we have achieved one of our initial design goals, i.e., to use a ROLAP compatible indexing scheme on top of CURE that does not induce significant overhead on the cubing process.

Furthermore, Fig. 21 presents average query-response times produced by QI-CURE and QU-CURE over the indexed and the unindexed version of the CURE cube of CovType, respectively. Figure 22 does the same for Sep85L. The workloads in these experiments consist of 500 random queries and the values shown in the figures are averages. Three factors determine the selectivity of these workloads, namely *point probability pp*, *range probability rp*, and *all-values probability ap*. Factor *pp* denotes the probability that the values of a dimension involved in a query are filtered by an equality condition in the corresponding WHERE clause.

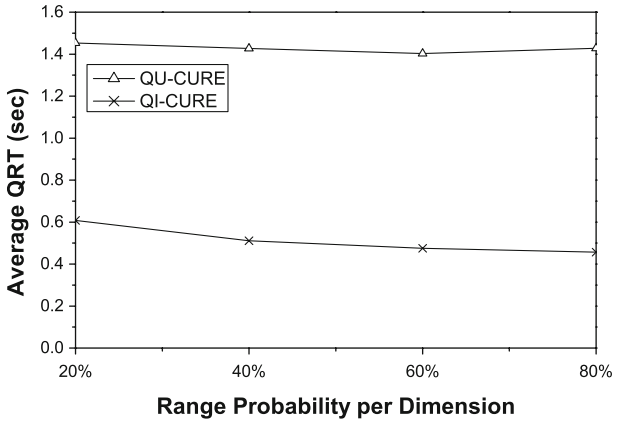

Fig. 22 Average QRT (Sep85L)

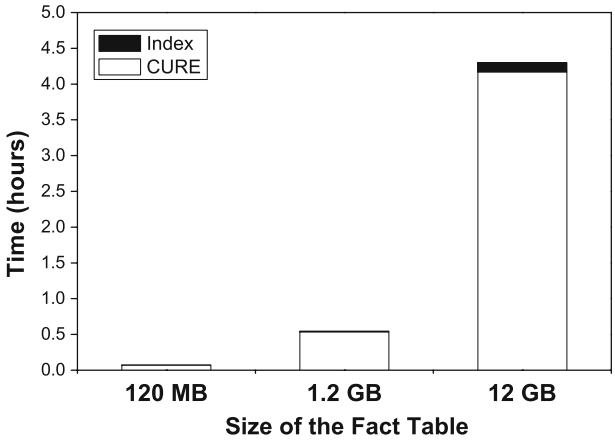

Fig. 23 Construction time (APB-1)

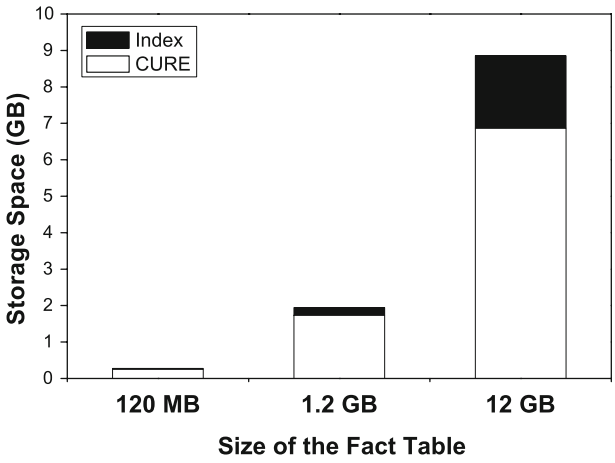

Fig. 24 Storage space (APB-1)

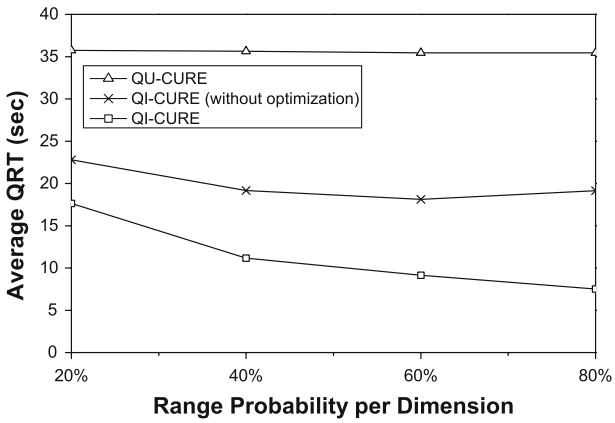

Fig. 25 Average QRT (APB-1, $|R| \approx 12$ GB)

Similarly, *rp* is the probability that the values of a dimension are filtered by a range condition, while *ap* is the probability that the values of a dimension are not filtered at all. Clearly, $pp + rp + ap = 1$. In our experiments, we have set $pp = 0.2$ and varied the values of *rp* (and hence of *ap* as well, since they are dependent, $ap = 1 - pp - rp$). The horizontal axis in both figures represents the different values of *rp* that we have used. Clearly, in our setting, selectivity increases with *rp* (more selective queries return less tuples). The reason for setting *pp* in a relatively small value and varying *rp* is that a large value of *pp* produces very selective queries that generate very small or even empty result sets, which are not indicative. Finally, we have set the selectivity factor (defined in Sect. 4.4) of every range predicate to 20%.

Clearly, answering selective queries over CURE cubes benefits when the fact table is indexed. Expectedly, the difference increases with selectivity in favor of QI-CURE. The increase is, however, marginal and this is attributed to that the corresponding fact tables fit in memory, which makes the cost of fetching tuples from them relatively low. Hence, further filtering the access to fact-table tuples does not offer much. Due to lack of publicly available real-world datasets much larger than CovType and Sep85L, we have further experimented with synthetic ones, generated by the data generator of the APB-1 benchmark [26]. Our conclusions follow.

Hierarchical cubes: Figures 23 and 24 illustrate the construction time and the storage space, respectively, required for constructing and indexing a CURE cube over a synthetic APB-1 dataset of a low (0.4), a medium (4), and the highest possible (40) density factor. The values along the horizontal axis indicate the sizes of the corresponding fact tables. Again, the white part of each column represents the resources consumed by the original CURE algorithm, while the black part shows the additional cost of indexing. Once more, we see that our indexing technique does not induce significant overhead, even when applied over the most challenging APB-1 dataset (of density factor 40).

Furthermore, we have experimented with query answering over the indexed and unindexed CURE cubes of the aforementioned synthetic datasets. The characteristics of the workloads we have used are the same, as described above for the real-world datasets. Figure 25 shows the results for

the most challenging APB-1 dataset (of density factor 40), which consists of approximately 5×10^8 tuples occupying 12 GB in binary format. Trends for factors 0.4 and 4 are similar, hence omitted. Clearly, the use of indices offers great benefits to QI-CURE, increasing with selectivity. In order to test the impact of query optimization in the efficiency of QI-CURE, in this experiment, we have also tried an alternative run with the optimizer turned off. The line labeled "QI-CURE (without optimization)" in Fig. 25 represents the corresponding result. Expectedly, accessing an index for every predicate in the WHERE clause of a query is not a good solution. According to the discussion in Sect. 4.4, the access of an index may be expensive; hence, a proper selection of which indices to use is mandatory. Recall that our selection is based on formula (4.1).

5 Incremental maintenance

Data cubes are usually constructed off-line in order to accelerate aggregate queries and consequently the every-day usage of the data stored in a data warehouse. Nevertheless, such data is not static in the general case but changes over time, as new tuples are inserted in the fact table. Such modification must be reflected in the data stored in the cube as well; otherwise, query results become inaccurate.

Typically, data in a warehouse is periodically refreshed in batch mode during a window of time, which must be kept as small as possible. Hence, reconstructing the entire cube from scratch every time the fact table changes is clearly not practical; incremental maintenance must be applied instead. In this section, taking into account the importance of refreshing cube data, we study incremental maintenance of CURE cubes following two alternatives: (a) an *eager* approach that updates a CURE cube in a single off-line process and (b) a *lazy* approach that performs only some lightweight operations during the off-line phase, preparing the actual update of cube tuples on-line the first time they are accessed during query answering. Finally, we combine the eager and the lazy approach into an even more promising hybrid method. To the best of our knowledge, the idea of lazily updating a cube is novel.

In the following presentation, we denote by *deltas* the new tuples inserted in the fact table since the last time the fact table and the cube were completely consistent. We also make the realistic assumption that the *delta fact table* is much smaller than the original one. Moreover, we assume that the aggregate functions applied are self-maintainable [25]. A set of aggregate functions is self-maintainable if the new value of the functions can be computed solely from the old values of the aggregation functions and from the changes to the base data. Aggregate functions can be self-maintainable with respect to insertions only, deletions only, or both. A self-maintainable aggregate function is always distributive; conversely, a distributive aggregate function is always self-maintainable with respect to insertions but not necessarily with respect to deletions. The COUNT function can help certain distributive aggregate functions to become self-maintainable with respect to deletions as well. With respect to algebraic aggregate functions, they can be expressed as a scalar function of distributive aggregate functions (e.g., avg = sum/count); hence, by keeping their (self-maintainable) distributive parts separately, they can also become self-maintainable. For more details on function types, please refer elsewhere [25]. Finally, we focus only on insertions; deletions and updates can be treated similarly with some additional machinery. Briefly, for deletions, we assume that tuples in the fact table are not physically deleted but are simply marked as deleted. Otherwise, row-id references stored in the cube pointing to deleted tuples would be invalidated. These are useful to restore dimension values for cube entries but not aggregate values, which are obtained from the cube entries themselves, without requiring access to the fact table, based on the CURE properties described in Sect. 3.1. Moreover, to support deletions, we also need to materialize in the cube the result of the COUNT aggregate function, even if the user does not explicitly declare it in the definition of the cube schema. COUNT is required for two reasons: (a) for making a larger number of aggregate functions self-maintainable with respect to deletions, as explained above and (b) for monitoring when an NT/CAT becomes a TT, which occurs when COUNT drops to 1 due to deletions. With respect to updates, we treat them as a combination of delete and insert.

5.1 An eager approach

The most common scenario of updating a ROLAP cube with a set of deltas involves the construction of the delta cube, built using only the deltas, and a subsequent refresh phase for merging the original cube with the delta one. However, existing techniques are not practical in the case of CURE cubes, mainly due to that they have been designed for flat rather than hierarchical cubes and, as a consequence, they do not take care of the additional challenges imposed by the nature of hierarchies, as explained in detail below.

According to the approach proposed elsewhere [7] for accelerating the refresh phase in the case of BU-BST cubes, a collection of a large number of indices is necessary, one for every different node of the original cube. These indices serve the search for matches between tuples of the delta and the original cube (two tuples match when they have the same dimension values). However, as shown in the existing literature [23], indexing every cube node separately is generally very expensive in terms of both computational and storage resources even in the flat case; the situation becomes even worse in the presence of hierarchies, since the number of

cube nodes increases dramatically. Hence, the approach of BU-BST would invalidate the benefits of the CURE algorithm, including construction speed and compression rate, being thus unsuitable.

On the other hand, the algorithm proposed for the refresh phase of updating a TRS-BUC cube is an attempt in the past to avoid indexing the entire cube [23]. The algorithm scans the tuples of the original cube and searches for matches with tuples in the delta cube (the BU-BST approach does the opposite). Proper in-memory hash tables built for indexing the delta cube accelerate the procedure, based on that indexing the delta cube is much cheaper than indexing the original one. In order to check for tuple matches the algorithm decompresses the TRS-BUC cube nodes while scanning the tuples stored in them, which is equivalent with restoring all redundant information that had been removed from the cube during construction. Decompressing tuples is necessary, since TRS-BUC replaces some kinds of tuples (only TTs) with row-id references, as also performed by CURE for all tuples. Clearly, row-ids in the original and the delta cube are not comparable, because they reference tuples stored in different fact tables; hence restoring redundant information seems inevitable. Unfortunately, this approach is not applicable in the case of CURE either. As mentioned above, TRS-BUC removes only some dimensional redundancy from the cube by identifying only TTs and stores all the remaining tuples in an uncompressed format. On the contrary, CURE identifies and removes not only some but the entire dimensional redundancy, constructing a much more compressed cube. Furthermore, CURE has been designed for hierarchical cubes as well, which include a dramatically larger number of both nodes and tuples compared to flat cubes. Hence, although restoring the entire cube during update may be practical in the case of TRS-BUC, it is impractical in the case of CURE due to the reasons analyzed above.

Taking into account the increased requirements imposed by the nature of hierarchies, we target at an update algorithm that is not based on the existence of indices on the entire cube, while it still does not require decompressing the entire cube during the refresh phase. A way to achieve these design goals is to make row-id references stored in the original and the delta cube comparable. This means that if a tuple dt stored in some node N of the delta cube matches with a tuple t in the same node of the original cube, the associated row-id stored in the delta cube should not point to the first tuple dft in the delta fact table that has contributed to the construction of dt; instead, the aforementioned row-id must point to the first tuple ft in the original fact table that has contributed to the construction of t. Since t and dt match, the projections of dft and ft on the grouping attributes of N will match as well, which ensures the correctness of the proposed solution.

In other words, we propose elimination of the need for decompressing the original cube by incorporating smarter compression into the construction of the delta cube. This requires (a) small modifications in the CURE algorithm that constructs the delta cube and (b) the existence of indices over the original fact table R, for example of the form described in Sect. 4.3, which are the same indices that are used by QI-CURE as well. We call UE-CURE (Update Eagerly CURE, Algorithm 3) the modified version of CURE. Among the input parameters of this algorithm, R denotes the original fact table and DR the delta fact table.

Algorithm 3 UE-CURE(OriginalCube, R, DR)

```
1: Call CURE(DR) to generate the tuples of DeltaCube (without storing
   them) and
2: for each tuple dt of any node N CURE generates do
3:    Find the set S of tuples in R that match with dt;
4:    if (S == ∅) then
5:       Append dt in node N of the OriginalCube;
6:    else
7:       dt.R-rowid = FindMinRowid(S);
8:       Store the modified dt in node N of the DeltaCube;
9:    end if
10: end for
11: for each node N of the DeltaCube traversed in a bottom-up and
   breadth-first fashion do
12:    for each tuple dt that belongs to node N do
13:       t = FindMatchingTuple(OriginalCube, N, dt);
14:       t_new = Merge(t, dt);
15:       Replace t by t_new in the OriginalCube;
16:       if t was TT in the OriginalCube then
17:          Write t (as TT) in the parent nodes of N in the OriginalCube;
18:       end if
19:    end for
20: end for
```

The key idea is that, for every tuple dt that UE-CURE constructs for some node N of the delta cube (Algorithm 3, lines 1-2), it must further use the indices of R in order to find the set S of tuples in R whose projection on the grouping attributes of N matches with dt (line 3). (We will give more details about the construction of S below.) If S is empty (line 4), dt does not match with any tuple in R, so it cannot match with any tuple in node N of the original cube either. In this case merging is trivial and degenerates to appending dt in node N of the original cube (line 5). Otherwise, if S is not empty (line 6), the aggregation of the tuples in it must have given a tuple t in node N of the original cube during its construction. According to the properties of CURE, the row-id that substitutes the dimension values of t in the compressed format of CURE must be the smallest row-id of the tuples in S. Knowing S enables UE-CURE to find this smallest row-id without accessing any tuple in the node N of the original cube, but only the indices of R (line 7). Hence, UE-CURE can store dt in the delta cube (line 8) replacing its dimension values with the smallest row-id of the tuples in S, which

coincides with the row-id of the matching tuple t stored in node N of the original cube.

After constructing the delta cube, UE-CURE merges the matching tuples, which are by construction those sharing the same row-id values, using a merge-scan algorithm (lines 11-20), based on that CURE sorts cube tuples according to the row-id values, as proposed in Sect. 3.1 for performance reasons. Note that, during merging, any tuple t that becomes updated and was stored as an NT in the original cube remains an NT, whereas TTs and CATs may be stored either as NTs or CATs, depending on the modifications performed on other cube tuples as well. In order to classify the updated tuples and separate NTs from CATs, UE-CURE follows the exact same strategy of the original CURE, i.e., it uses a specialized structure called signature pool (originally defined elsewhere [22]). Hence, UE-CURE preserves the CURE format unaffected and generates an updated cube identical to the one that would be produced by full reconstruction, a highly desirable property of any incremental-maintenance algorithm. This operation is incorporated in functions Merge and Replace (lines 14–15). Furthermore, any updated tuple t that was stored as TT in some node N of the original cube must be written as TT in all of N's parents one level above in the execution plan of CURE (lines 16–18), since it will not be shared any more between N and its ancestors. (Recall that an example of an execution plan of CURE appears in Fig. 7.) This property makes it necessary that the merge-scan algorithm visits the nodes in the execution plan in a bottom-up and breadth-first fashion (line 11). Note that the aforementioned algorithm is customized only for CURE cubes, since CURE is the only method that replaces all dimension values with row-id references.

Let us now return to the topic of efficiently constructing the aforementioned set S and explain how UE-CURE takes advantage of the pipelined execution of the original CURE (Fig. 7). Assume that UE-CURE has just generated a tuple $\mathrm{d}t = \langle a_2 \rangle$ of node A_2 in the example of Fig. 7 (we omit the aggregate values in the vector that represents $\mathrm{d}t$ for the sake of simplicity) and has found the corresponding set $S_{\langle a_2 \rangle}$ of tuples in R that match with $\mathrm{d}t$, i.e., the set of tuples in R that have the same value a_2 in A_2. Then, according to the particular execution plan, UE-CURE proceeds to node $\mathrm{A}_2\mathrm{B}_1$ and, for some value b_1 in the domain of B_1, it generates tuple $\langle a_2, b_1 \rangle$. Clearly, the following equation holds $S_{\langle a_2,b_1 \rangle} = \mathrm{S}_{\langle a_2 \rangle} \bigcap S_{\langle b_1 \rangle}$. Hence, in order to find the set of tuples $S_{\langle a_2,b_1 \rangle}$, UE-CURE must simply find $S_{\langle b_1 \rangle}$ and intersect it with $S_{\langle a_2 \rangle}$, which is already materialized from the previous step. Taking into account the particular structures used for indexing R (Sect. 4.3), this is cheaper than finding $S_{\langle a_2,b_1 \rangle}$ from scratch. Generalizing this example, whenever UE-CURE generates another tuple $\mathrm{d}t$ in some node N of the delta cube, it also accesses a single index structure, the one associated with the right-most dimension of N, finds the set of tuples in R that have the same value with $\mathrm{d}t$ in the right-most dimension of N, and intersects it with a similar set constructed in the previous node visited in the execution plan.

Furthermore, if UE-CURE finds some set S empty, it concludes not only that the newly generated tuple does not match with any tuple in the original cube, but also that all of its specializations in the higher levels of the execution plan do not match either. The proof is straightforward: the result of the intersection of any set with an empty set is always the empty set. This property allows an optimization that avoids the scan of indices in higher levels, if a tuple has already been found nonmatching in some lower level.

A potential problem of the method is that the number of tuples in R with a specific value in a particular dimension may be very large, especially when this dimension is at a high hierarchy level and has, therefore, a small domain. In such a case, the number of row-ids associated with the particular value in the corresponding index of R will be equally large, which may produce large sets during the operation of UE-CURE. This fact generates some concerns, since the construction and intersection of large sets may be prohibitively expensive. In order to overcome this problem, we propose the construction of a limited number of additional indices using information stored in some cube nodes, trading some construction time and storage space for update efficiency. The key point is that $|\mathrm{N}| \leq |\mathrm{R}|$ for any node N, since N stores aggregated data; hence, the indices built based on the data of N are smaller as well, resulting into smaller sets during the operation of UE-CURE. Note that the additional indices discussed here should not reference tuples stored in N, since this would be useless according to the description of UE-CURE. Instead, they should reference tuples in R, provided that the row-ids of the corresponding tuples appear in some tuple stored in N. In other words, the references stored in these indices are the subset of the references stored in the indices of R that appear in the R-rowid column of some tuple in N. This is functional because UE-CURE does not actually need the entire set S of tuples in R with a particular value in a specific dimension, but only the smallest row-id among them. N stores exactly this smallest row-id; hence, the corresponding index built using data of N can accelerate the search for the particular row-id during update.

Having decided to construct additional indices, the problem that arises is which nodes to select in order to build the corresponding indices. In order to solve this problem we formulate the following rule:

Rule 5.1: Assume D dimensions $\Delta_1, \ldots, \Delta_D$ and let $\Delta_{i,j}$ denote that the i-th dimension is at level j. (ALL denotes the maximum level). Then the construction of the sets associated with node

$$\mathrm{N} = \Delta_{1,l_1} \ldots \Delta_{k,l_k} \Delta_{k+1,\mathrm{ALL}} \ldots \Delta_{D,\mathrm{ALL}}$$

with $l_k \neq$ ALL and $k \leq D$ (k is the order of N's right-most dimension) can be accelerated by indices constructed based on data of a node

$$\text{IN} = \Delta_{1,l'_1} \ldots \Delta_{k,l'_k} \Delta_{k+1,l'_{k+1}} \ldots \Delta_{D,l'_D}$$

if $l'_i \leq l_i, \forall\, i \in [1, k]$ and $l'_i = 0, \forall\, i \in (k, D]$.

The proof of rule 5.1 is based on that we can use the indices of a node IN to accelerate the construction of sets in N, provided that IN is at least as detailed as N and all of its ancestors in the execution plan of UE-CURE (Fig. 7) that have the grouping attributes of N as a prefix. The latter is required to preserve the correctness of UE-CURE; otherwise, the incremental construction of sets using intersection would be invalidated. For example, the sets of node A_2 can be constructed using indices based on $A_2B_0C_0$, $A_1B_0C_0$, or $A_0B_0C_0$, but not on A_2, A_1B_0, or $A_1B_1C_0$.

Furthermore, let |IN| denote the number of tuples in node IN, d_{IN} the number of grouping attributes of IN, k the order of the right-most dimension of N, l_k its hierarchy level, and C_{k,l_k} its cardinality (i.e., its domain size). Then, we can estimate the cost CI_{IN} of constructing indices using data in IN and the cost $CS_{\text{N,IN}}$ of constructing a set in node N using the indices of IN (given that N and IN obey rule 5.1) with the following formulas:

$$CI_{\text{IN}} = |\text{IN}| \times d_{\text{IN}} \tag{5.1}$$

$$CS_{\text{N,IN}} = \frac{|\text{IN}|}{C_{k,l_k}} \tag{5.2}$$

Formula (5.1) expresses the cost of constructing a number of d_{IN} indices referencing a number of |IN| tuples each. Formula (5.2) says that the cost of constructing a set at a node N based on indices of IN is proportional to the number of tuples in IN that have the same value in the right-most dimension of N, assuming a uniform distribution.

Moreover, let *SN* be the set of all cube nodes, *SIN* the set of the indexed nodes, and IN(N) the smallest node IN $\in$ *SIN* that obeys rule 5.1 with respect to N. Then, the total cost *CI* of indexing, and the total cost *CS* of constructing all sets in all nodes after the insertion of a delta tuple are clearly given by formulas (5.3) and (5.4), which can be evaluated based on formulas (5.1) and (5.2), respectively.

$$CI = \sum_{\text{IN} \in SIN} CI_{\text{IN}} \tag{5.3}$$

$$CS = \sum_{\text{N} \in SN} CS_{\text{N,IN(N)}} \tag{5.4}$$

CI (*CS*) increases (decreases) with the number of nodes in *SIN*. Our experiments (e.g., Fig. 26, whose details will be explained in Sect. 5.4) show that the graph *CS* versus *CI* has a knee after the addition of a relatively small number of nodes in *SIN*. Based on this, we propose the use of a greedy algorithm called SelectSIN (Algorithm 4) for the selection of a proper set *SIN*.

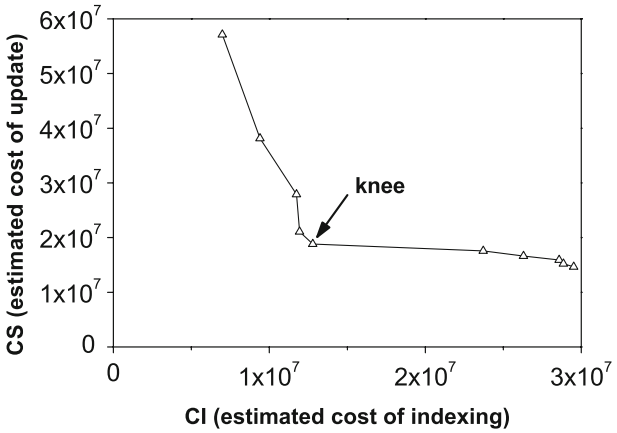

Fig. 26 The graph *CS* versus *CI*

Algorithm 4 SelectSIN(R, *SN*)

1: Set *SIN* = {R};
2: **repeat**
3: Find node IN (where IN $\in$ *SN* and IN $\notin$ *SIN*) that maximizes the difference CS^{SIN} - $CS^{SIN \cup \{\text{IN}\}}$;
4: Set *SIN* = *SIN* $\cup$ {IN};
5: **until** a knee has been found
6: Return *SIN*;

5.2 A lazy approach

In the previous subsection, we developed UE-CURE, an eager algorithm for the incremental maintenance of CURE cubes. As also revealed by our experimental evaluation presented in Sect. 5.4, the dominating factor in UE-CURE is the time taken for the identification of tuple sets in R with a particular value in a given dimension. Unfortunately, the cost of this key operation is usually considerable and increases with the size of R, raising concerns for the practicality of UE-CURE. Additionally, recall that we have already rejected other solutions based on indexing the entire cube or decompressing it during merging. Hence, the following question arises: *is the compression rate of CURE cubes so high and the challenges imposed by the nature of hierarchies so demanding that they prevent the development of any efficient algorithm for the incremental maintenance of CURE cubes?* Working on this question, we have come up with the following answer: *the incremental maintenance of CURE cubes seems prohibitively expensive if we think conventionally trying to develop algorithms following the well-known strategies (i.e., eager tactics)*. In other words, the problem is so demanding that we need to think unconventionally and develop an algorithm that significantly deviates from policies common in data cubing. Such an algorithm is the topic of this subsection.

The key idea of our alternative approach is that no data stored in a cube can ever be considered obsolete until some query retrieves it. In this sense, the eager approaches seem

too pessimistic and conservative: they spend a considerable amount of time accessing and refreshing a-priori every last detail of the entire cube, although some parts of it may be queried in the far future, if ever.

Our lazy approach presented here takes a completely different route. During the off-line window available for the incremental maintenance of the entire data warehouse, we simply construct the delta cube using the original CURE algorithm with no modification whatsoever. Taking into account that the delta fact table is usually much smaller than R, the construction of the delta cube can be thought of as a lightweight operation, especially since performed by an efficient algorithm like CURE. By constructing the delta cube, we prepare the system for incrementally updating data on-line, the first time retrieved by a query, say Q. Note that lazy approaches have been successfully used in the past for updating other types of entities stored in a data warehouse, e.g., materialized views and indices [6,29,43]. Nevertheless, to the best of our knowledge, they have never been applied in algorithms related to data cubes. Furthermore, to the best of our understanding, existing lazy methods update the entire entity, if necessary, before accessing it. In our approach, we only update a part of it, the one consisting of the tuples that belong to the result set of the corresponding query Q.

To understand why a lazy approach is more promising, recall that one of the main goals of (eager) UE-CURE is to merge the original and the delta cube without decompressing the former in order to avoid the considerable additional cost. However, during query answering the cube tuples selected by Q are anyway decompressed, as explained in Sect. 4. Hence, shifting merging during query answering achieves an easier search for tuple matches between the original and the delta cube, without additional costs.

Having constructed the delta cube off-line, our lazy approach needs some simple modifications in the query answering algorithms originally presented in Sect. 4. Let us use UL-CURE (Update Lazily CURE) to denote the extended query answering algorithm (Algorithm 5). The input parameters of UL-CURE are a superset of the parameters of QI-CURE (Algorithm 2): DR denotes the delta fact table and D_AGGR the relation AGGREGATES of the delta cube. Instead of answering Q by selecting tuples from the original cube only, UL-CURE first selects the tuples in the delta cube that satisfy the conditions of Q (line 1), and keeps them in some memory structure H that allows fast look-ups (e.g., a hash table). Then, it selects the tuples from the original cube that satisfy Q's conditions and for every such tuple t (lines 2–3), it checks whether t matches with any tuple dt in H (line 4), before writing t in the output. If it finds no match (line 5), it simply outputs t (lines 6–7). Otherwise (line 8), it removes dt from H (so that it does not find it again in subsequent steps), updates the aggregate values of t by merging them with the corresponding values of dt, and writes the updated tuple t_{out} in the output (lines 9–11). Subsequently, UL-CURE checks whether updating the cubes on the disk is safe (line 12). If Q queries node N, the condition that must hold for safety is that neither t nor dt is a TT physically stored in a descendant node DN of N in the execution plan of CURE (something possible due to the properties of TTs). Otherwise, if either t or dt (or both) is a TT that indirectly belongs to N and is actually stored in DN, then UL-CURE must not update either of them on the disk, because fixing the data in nodes DN and N would not be enough, since a large number of nodes may exist in-between in the execution plan and updating a TT on disk in this case would cause a loss of tuples in these nodes. On the other hand, if the safety condition holds, UL-CURE updates the original and the delta cube as follows: it replaces the old tuple t with t_{out} and removes dt from delta cube so that it does not find it again in subsequent queries (lines 13–14). (Interestingly, the update on disk exhibits some desirable locality of reference, since the block containing t is already resident in memory.) Moreover, if either t or dt was a TT (physically stored in N) before merging, UL-CURE must perform an additional step of writing it also as TT in N's parents in the execution plan of CURE (lines 15–20); otherwise, tuples would be lost, as already explained above.

After writing to the output all tuples of the original cube that satisfy the conditions of Q, UL-CURE makes a final pass of the remaining tuples in H (lines 24–30). These are tuples from the delta cube that satisfy the query conditions, but have not matched with any tuple in the original cube. UL-CURE writes these tuples to the output (line 25). Furthermore, it appends them in the original cube (line 27) and removes them from the delta cube (line 28), after checking whether this operation is safe (line 26).

Note that UL-CURE preserves the CURE format using a signature pool, as also performed by CURE and UE-CURE. This behavior is incorporated in functions Replace and Append (Algorithm 5, lines 13 and 27). Finally, note that after UL-CURE updates a tuple on the disk, it can simply use it in subsequent queries without paying any additional costs for updating it again. This feature is particularly beneficial if there are hot spots in the cube data, i.e., parts of the cube frequently queried.

5.3 A hybrid approach

Our experiments in Sect. 5.4 show that the lazy approach (UL-CURE) is much more efficient than the eager one (UE-CURE), since the latter takes more time even than reconstruction. Nevertheless, the use of UE-CURE is sometimes inevitable, fortunately, in trivial cases that UE-CURE behaves well: Assume that during an update window a delta fact table

Algorithm 5 UL-CURE(R, DR, AGGR, D_AGGR, N, Cond, *RS*)

```
1: Fill H with the set of delta tuples in N that satisfy Cond returned by
   QU-CURE(DR, D_AGGR, N, Cond);
2: Call QI-CURE(R, AGGR, N, Cond, RS) and
3: for each tuple t in its result set do
4:    dt = SearchForMatching(t, H);
5:    if (dt == NULL) then
6:       t_out = t;
7:       Output(t_out);
8:    else
9:       Remove dt from H;
10:      t_out = Merge(t, dt);
11:      Output(t_out);
12:      if neither t nor dt is a TT physically stored in a descendant of
         N then
13:         Replace t by t_out in the original cube;
14:         Remove dt from the delta cube;
15:         if t was TT in the original cube then
16:            Write t (as TT) in the parent nodes of N in the original
               cube;
17:         end if
18:         if dt was TT in the delta cube then
19:            Write dt (as TT) in the parent nodes of N in the delta cube;
20:         end if
21:      end if
22:   end if
23: end for
24: for each tuple dt remaining in H do
25:    Output(dt);
26:    if dt is not a TT physically stored in a descendant of N then
27:       Append dt in the original cube;
28:       Remove dt from the delta cube;
29:    end if
30: end for
```

DR_1 arrives, forcing the system to construct the corresponding delta cube DC_1, according to the description in Sect. 5.2. Then, during queries, UL-CURE updates the original cube and removes some tuples from DC_1. However, UL-CURE does not guarantee that it will consume all tuples in DC_1 incorporating them in the original cube, because this depends on the particular workload. Therefore, if during a subsequent update window another delta fact table DR_2 arrives, the system will not be able to handle it. In this case, merging DR_1 and DR_2 and reconstructing the corresponding delta cube from scratch would be incorrect, since some tuples of DC_1 have already been incorporated in the original cube. Thus, in this case, the only solution is to eagerly update the remaining tuples in DC_1 with the tuples of DC_2 using UE-CURE. Since deltas are usually small, UE-CURE must be efficient on that. Our experiments in Sect. 5.4 confirm this hypothesis.

In conclusion, we propose a hybrid method for updating CURE cubes: UL-CURE updates the original cube in a lazy fashion, while UE-CURE eagerly updates the delta cube, if the latter is not entirely incorporated in the original one during the on-line phase.

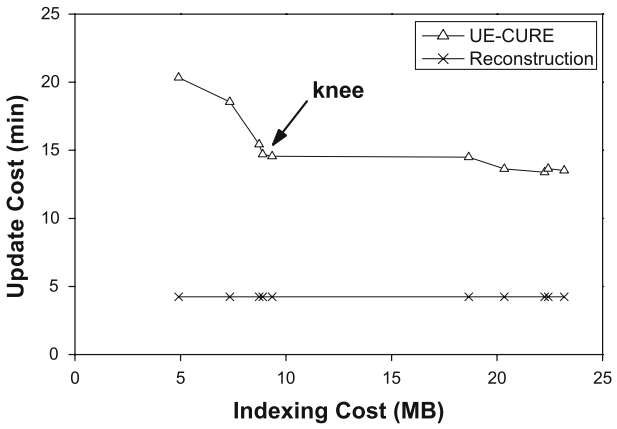

Fig. 27 Update time of UE-CURE

5.4 Experimental evaluation

In order to evaluate the efficiency of the proposed methods, we have implemented the two approaches for incrementally updating CURE cubes, namely UE-CURE and UL-CURE. In this subsection, we focus on CURE and exclude the other BUC-based algorithms from the following experiments for the same reasons already explained in Sect. 4.5. Below, we present the most indicative results of our experimental study. We omit the rest that provide no further intuition on the behavior of the proposed techniques for the sake of brevity.

As explained in Sect. 5.1, the update speed of UE-CURE becomes faster by constructing additional indices over nodes selected by algorithm SelectSIN (Algorithm 4). Figure 26 illustrates the graph *CS* vs. *CI* (i.e., estimated update cost vs. estimated index cost) for the ten first nodes selected by SelectSIN for accelerating the update process of an APB-1 dataset of density factor 0.4 ($|R| \approx 120$ MB). Clearly, the graph has a knee on the fifth point, suggesting that indexing only the five first nodes proposed by SelectSIN achieves a profitable tradeoff. Furthermore, Fig. 27 presents the actual efficiency of UE-CURE when updating a fact table of APB-1 of density factor 0.4 ($|R| \approx 5 \times 10^6$ tuples) with 10^4 delta tuples. The update cost appears with respect to the storage space traded for constructing additional indices over up to the ten first nodes proposed by SelectSIN. The graph has a knee at the fifth point as well, confirming the worthiness of the selection of SelectSIN. Unfortunately, even after this optimization, the update efficiency of UE-CURE is approximately three times worse than reconstruction, verifying our claim that an eager approach is not a viable solution. (The situation becomes worse as |R| increases). However, note that in such a case that involves a relatively small R, although the update cost of UE-CURE is higher than the cost of reconstruction, it is still reasonable as an absolute number (15 min are not prohibitive). Such behavior supports our claim that UE-CURE can be practically used in a hybrid

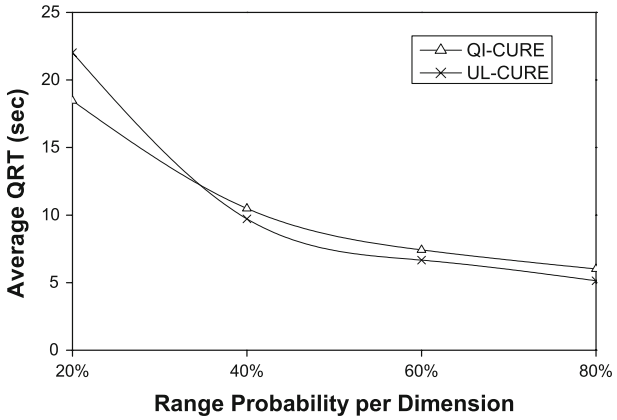

Fig. 28 Overhead of UL-CURE

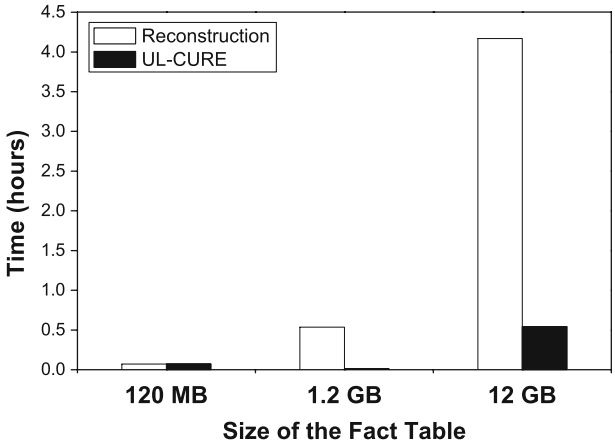

Fig. 29 Update time of UL-CURE

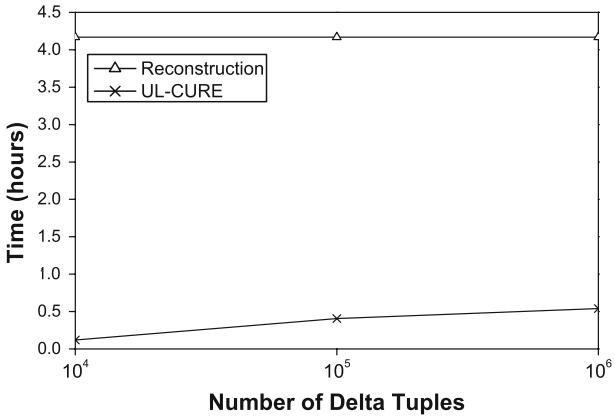

Fig. 30 Effect of delta size on UL-CURE

solution (Sect. 5.3) only for updating small delta cubes with new delta tuples, if necessary.

Unlike UE-CURE that generally generates performance concerns, the lazy approach UL-CURE behaves very efficiently, even when updating the cube that corresponds to the most challenging dataset used in this paper, i.e., the dataset constructed by the data generator of APB-1 in its highest density factor 40 ($|R| \approx 5 \times 10^8$ tuples), with 10^6 delta tuples. Figure 28 illustrates the efficiency of UL-CURE in answering queries while updating the corresponding cube compared with the simple QI-CURE that only answers queries over the reconstructed cube. The workloads are the same also used in the experiment of Fig. 25. Interestingly, UL-CURE only adds some marginal overhead in queries of low selectivity (large output) and even outperforms QI-CURE as selectivity increases. This is attributed to that selective queries produce smaller result sets and, therefore, a smaller number of tuple matches between the original and the delta cube. This trend decreases the cost of update. Furthermore, the size of the original cube (the one refreshed by UL-CURE) is smaller than the size of the reconstructed cube (the one accessed by QI-CURE for answering queries); hence access costs over the former are lower than over the latter, decreasing the overall cost of UL-CURE.

The workloads used in the previous experiment do not guarantee that UL-CURE updates the entire cube incorporating every last detail of the delta cube into it. In order to study the behavior of UL-CURE under such conditions, we generated a workload of all possible node queries on top of the cubes constructed over APB-1 datasets of densities 0.4, 4, and 40, respectively. Recall that a node query includes no WHERE clause in the corresponding SQL syntax and, therefore, reconstructs an entire node of the corresponding cube. Such workloads guarantee that UL-CURE refreshes the entire cube. Figure 29 shows the amortized cost for updating the three cubes mentioned above compared to the cost of full reconstruction. In the horizontal axis appear the sizes of the corresponding fact tables. Note that from the (amortized) cost of UL-CURE we have subtracted the pure cost of query answering, keeping only the additional cost for refreshing the cube and the construction time of the delta cube (which is negligible, approximately equal to 30 sec). The conclusion is that in the small dataset (to the left), whose fact table fits in memory, the amortized update cost is relatively high and comparable to the reconstruction cost. This is attributed to the fact that the part of the algorithm responsible for query answering is very fast, since it benefits from caching incurring no I/O. On the other hand, the part of the algorithm that is responsible for the update procedure stores the updated and the new values on disk, incurring I/O, which dominates overall. On the contrary, the situation in the larger datasets is different: the I/O for query answering dominates the I/O for pure update, making the amortized cost of UL-CURE approximately an order of magnitude smaller than reconstruction time.

Finally, in order to study the effect of the size of the delta fact table on the performance of UL-CURE, we have varied the number of delta tuples from 10^4 to 10^6. Figure 30

shows the results on the most challenging dataset used in this paper (APB-1 of density factor 40). Note that the horizontal axis is logarithmic. Clearly, the amortized update cost of UL-CURE scales well with the size of the delta fact table. Although reconstruction time does not depend on the delta size, we have added it in the figure for comparison reasons.

6 Conclusions and future work

In this paper, for the first time, we have studied the entire ROLAP data cube lifecycle in the presence of hierarchies. In particular, we have built upon previous work on construction and storage of CURE cubes [22] and have developed a suite of efficient algorithms for query processing and incremental updating over them. These are significantly different from earlier approaches, which have been proposed for flat cubes constructed by other techniques but are inadequate for CURE, due to its high compression rate and the presence of hierarchies. We have addressed issues such as indexing, query optimization, and lazy update policies. Especially regarding updates, such lazy approaches have been applied for the first time on cubes. We have demonstrated the effectiveness of CURE in all phases of the cube lifecycle 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 and usage 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 itself and of ROLAP in general.

In the future, we would like to investigate the power of using aggregate queries in different types of applications. Some examples include applications for classification over large and continuously changing datasets or for improved searches in the web. Using CURE to implement such tasks efficiently seems very promising.

References

1. Acharya, S., Gibbons, P.B., Poosala, V.: Congressional samples for approximate answering of group-by queries. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 487–498 (2000)
2. Agarwal, S., Agrawal, R.,Deshpande, P., Gupta, A.,Naughton, J.F., Ramakrishnan, R., Sarawagi, S.: On the computation of multidimensional aggregates. In: Proceedings of Very Large Data Bases (VLDB), pp. 506–521 (1996)
3. Beyer, K.S., Ramakrishnan, R.: Bottom-up computation of sparse and iceberg cubes. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 359–370 (1999)
4. Blackard, J.A.: The forest covertype dataset. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype
5. Chen, M.S., Han, J., Yu, P.S.: Data mining: an overview from a database perspective. IEEE Trans. Knowl. Data Eng. **8**(6), 866–883 (1996)
6. Colby, L.S., Griffin, T., Libkin, L., Mumick, I.S., Trickey, H.: Algorithms for deferred view maintenance. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 469–480 (1996)
7. Feng, J., Si, H., Feng, Y.: Indexing and incremental updating condensed data cube. In: Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM), pp. 23–32 (2003)
8. Feng, Y., Agrawal, D., Abbadi, A.E., Metwally, A.: Range cube: Efficient cube computation by exploiting data correlation. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 658–670 (2004)
9. Gray, J., Bosworth, A., Layman, A., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 152–159 (1996)
10. Hahn, C., Warren, S., London, J.: Synoptic cloud reports. http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html
11. Han, J., Pei, J., Dong, G., Wang, K.: Efficient computation of iceberg cubes with complex measures. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 1–12 (2001)
12. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 205–216 (1996)
13. Jagadish, H.V., Lakshmanan, L.V.S., Srivastava, D.: What can hierarchies do for data warehouses? In: Proceedings of Very Large Data Bases (VLDB), pp. 530–541 (1999)
14. Karayannidis, N., Sellis, T.K., Kouvaras, Y.: Cube file: a file structure for hierarchically clustered olap cubes. In: Proceedings of International Conference on Extending Database Technology (EDBT), pp. 621–638 (2004)
15. Kotidis, Y., Roussopoulos, N.: An alternative storage organization for rolap aggregate views based on cubetrees. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 249–258 (1998)
16. Kotsis, N., McGregor, D.R.: Elimination of redundant views in multidimensional aggregates. In: Proceedings of Data Warehousing and Knowledge Discovery (DaWaK), pp. 146–161 (2000)
17. Lakshmanan, L.V.S., Pei, J., Han, J.: Quotient cube: How to summarize the semantics of a data cube. In: Proceedings of Very Large Data Bases (VLDB), pp. 778–789 (2002)
18. Lakshmanan, L.V.S., Pei, J., Zhao, Y.: Qc-trees: an efficient summary structure for semantic olap. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 64–75 (2003)
19. Lee, K.Y., Kim, M.H.: Efficient incremental maintenance of data cubes. In: Proceedings of Very Large Data Bases (VLDB), pp. 823–833 (2006)
20. Li, C., Cong, G., Tung, A.K.H.,Wang, S.: Incremental maintenance of quotient cube for median. In: Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD), pp. 226–235 (2004)
21. Li, C., Tung, K.H., Wang, S.: Incremental maintenance of quotient cube based on galois lattice. J. Comput. Sci. Technol. **19**(3), 302–308 (2004)
22. Morfonios, K., Ioannidis, Y.: CURE for cubes: cubing using a ROLAP engine. In: Proceedings of Very Large Data Bases (VLDB), pp. 379–390 (2006)
23. Morfonios, K., Ioannidis, Y.E.: Supporting the data cube lifecycle: the power of ROLAP. VLDB J. **17**(4), 729–764 (2008)
24. Morfonios, K., Konakas, S., Ioannidis, Y.E., Kotsis, N.: ROLAP implementations of the data cube. ACM Comput. Surv. **39**(4), 12:1–12:53 (2007)
25. Mumick, I.S., Quass, D., Mumick, B.S.: Maintenance of data cubes and summary tables in a warehouse. In: Proceedings of

ACM Special Interest Group on Management of Data (SIGMOD), pp. 100–111 (1997)
26. OLAP Council: Apb-1 olap benchmark. http://www.olapcouncil.org
27. Pedersen, T.B., Jensen, C.S., Dyreson, C.E.: The treescape system: Reuse of pre-computed aggregates over irregular olap hierarchies. In: Proceedings of Very Large Data Bases (VLDB), pp. 595–598 (2000)
28. Ross, K.A., Srivastava, D.: Fast computation of sparse datacubes. In: Proceedings of Very Large Data Bases (VLDB), pp. 116–125 (1997)
29. Roussopoulos, N., Economou, N., Stamenas, A.: Adms: a testbed for incremental access methods. IEEE Trans. Knowl. Data Eng. **5**(5), 762–774 (1993)
30. Roussopoulos, N., Kotidis, Y., Roussopoulos, M.: Cubetree: Organization of and bulk updates on the data cube. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 89–99 (1997)
31. Sarawagi, S., Agrawal, R., Gupta, A.: On computing the data cube. In: Research report 10026. IBM Almaden Research Center, San Jose (1996)
32. Shao, Z., Han, J., Xin, D.: Mm-cubing: computing iceberg cubes by factorizing the lattice space. In: Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM), pp. 213–222 (2004)
33. Shukla, A., Deshpande, P., Naughton, J.F.: Materialized view selection for multidimensional datasets. In: Proceedings of Very Large Data Bases (VLDB), pp. 488–499 (1998)
34. Sismanis, Y., Deligiannakis, A., Kotidis, Y., Roussopoulos, N.: Hierarchical dwarfs for the rollup cube. In: Proceedings of ACM International Workshop on Data Warehousing and OLAP (DOLAP), pp. 17–24 (2003)
35. Sismanis, Y., Deligiannakis, A., Roussopoulos, N., Kotidis, Y.: Dwarf: shrinking the petacube. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 464–475 (2002)
36. Sismanis, Y., Roussopoulos, N.: The complexity of fully materialized coalesced cubes. In: Proceedings of Very Large Data Bases (VLDB), pp. 540–551 (2004)
37. Timko, I., Dyreson, C.E., Pedersen, T.B.: Pre-aggregation with probability distributions. In: Proceedings of ACM International Workshop on Data Warehousing and OLAP (DOLAP), pp. 35–42 (2006)
38. Vassiliadis, P., Sellis, T.K.: A survey of logical models for olap databases. SIGMOD Rec. 28(4):64–69 (1999)
39. Vitter, J.S., Wang, M.: Approximate computation of multidimensional aggregates of sparse data using wavelets. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 193–204 (1999)
40. Wang, W., Feng, J., Lu, H., Yu, J.X.: Condensed cube: an efficient approach to reducing data cube size. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 155–165 (2002)
41. Xin, D., Han, J., Li, X., Wah, B.W.: Star-cubing: computing iceberg cubes by top-down and bottom-up integration. In: Proceedings of Very Large Data Bases (VLDB), pp. 476–487 (2003)
42. Zhao, Y., Deshpande, P., Naughton, J.F.: An array-based algorithm for simultaneous multidimensional aggregates. In: Proceedings of ACM Special Interest Group on Management of Data (SIGMOD), pp. 159–170 (1997)
43. Zhou, J., Larson, P.Å., Elmongui, H.G.: Lazy maintenance of materialized views. In: Proceedings of Very Large Data Bases (VLDB), pp. 231–242 (2007)