The VLDB Journal (2008) 17:729–764
DOI 10.1007/s00778-006-0036-8

REGULAR PAPER

Supporting the data cube lifecycle: the power of ROLAP

Konstantinos Morfonios · Yannis Ioannidis

Received: 24 August 2005 / Revised: 29 January 2006 / Accepted: 11 July 2006 / Published online: 19 December 2006

Abstract The lifecycle of a data cube involves efficient construction and storage, fast query answering, and incremental updating. Existing ROLAP methods that implement data cubes are weak with respect to one or more of the above, focusing mainly on construction and storage. In this paper, we present a comprehensive ROLAP solution that addresses efficiently all functionality in the lifecycle of a cube and can be implemented easily over existing relational servers. It is a family of algorithms centered around a purely ROLAP construction method that provides fast computation of a fully materialized cube in compressed form, is incrementally updateable, and exhibits quick query response times that can be improved by low-cost indexing and caching. This 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.

Keywords ROLAP · Data cube · Compressed storage · Query processing · Incremental updating · Indexing · Caching

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

K. Morfonios (✉) · Y. Ioannidis
Department of Informatics and Telecommunications, University of Athens, Athens, Greece
e-mail: kmorfo@di.uoa.gr

Y. Ioannidis
e-mail: yannis@di.uoa.gr

1 Introduction

The data cube is one of the most important concepts in OLAP technology. It consists of the results of group-by aggregate queries on all possible combinations of the dimension attributes over a fact table in a data warehouse. Materialization of summary views on the cube is critical for improving the response time of OLAP queries and of operators such as roll-up, drill-down, and pivot.

A common representation of the data cube that illustrates the computational dependencies among the different group-bys is the cube lattice [9]. Figure 1 presents cube lattice of a fact table R with three dimensions (A, B, and C), where a lattice node label is a concatenation of the corresponding dimension names. If we denote the number of dimensions of a fact table with D, the number of all cube lattice nodes is 2^D. Hence, a naive implementation method that computes each node separately and simply stores the result has exponential time and space complexity. To overcome this, a wide range of methods that provide efficient cube implementation (with respect to both computation and storage) has been proposed, which are compatible with either the ROLAP or the MOLAP architecture. In this paper, we follow the majority of efforts so far and focus on ROLAP.

Data cube implementation, however, is not an end in itself. Cubes may be constructed off-line, but their lifecycle does not stop there. They are used for answering queries in order to improve response times. Additionally, data in the original fact table changes and data cubes must follow suit. Reconstructing the entire cube from scratch periodically is impractical; instead, incremental update methods must be employed.

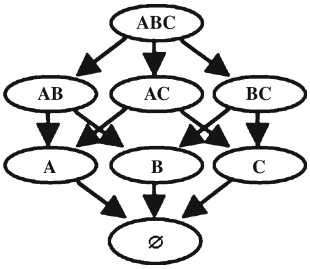


Fig. 1 Example of a cube lattice

The ideal data cube implementation must address efficiently all aspects of cube functionality in order to be viable. Nevertheless, existing ROLAP methods have only provided partial solutions, focusing mainly on (off-line) computation and storage, neglecting phases in the cube lifecycle that are related to its every-day usage and may, hence, be more important. This has given rise to several hierarchical data structures that provide efficient computation, storage, query answering, and updating of cubes, such as Dwarf [24], which is a highly compressed data structure that eliminates prefix and suffix redundancies efficiently. The main disadvantage of these approaches is that they are not ROLAP, i.e., they require specialized development effort.

Hence, the question that arises is the following: *Is the data cube problem so complex that it cannot be solved efficiently in its entirety using relational techniques?* If so, the extra effort and cost to implement more sophisticated methods is inevitable. In this paper, we investigate the power of ROLAP and propose three new cubing methods [PRT-PC, PRS-PC, and TRS-BUC] that comply with the relational model. Our experiments on both synthetic and real-world datasets demonstrate that TRS-BUC is the best among them and, to a large extent, dominates in performance all previously existing ROLAP techniques on all aspects of the data cube problem, sometimes by significant margins.

Note that, as a cube construction algorithm alone, TRS-BUC is similar to existing methods, only further enhanced with an improved storage scheme that provides significant reduction in storage space requirements and sometimes reduces construction time as well. This is not, however, the main contribution of this paper. The purpose here is not to propose another construction algorithm but an entire suite of construction, querying, and update algorithms that are based on TRS-BUC and enjoy the benefits of the improved storage scheme that TRS-BUC generates.

In brief, the greatest advantage of TRS-BUC is that it replaces a large number of tuples with row-id references to tuples in the fact table, thus redirecting many disk accesses during cube operations to a single relation. This redirection allows TRS-BUC to benefit from efficient caching and indexing, as the fact table can become the single point of attention for these optimizations. For example, with respect to indexing, cubes constructed by TRS-BUC are amenable to indexing schemes that accelerate cube usage overall while consuming only a limited amount of additional resources. In contrast, in all previous ROLAP techniques, indexing the cube is equivalent to indexing every one of its nodes. As we show later, this generates significant overheads in terms of both computational complexity and storage space requirements, rendering these techniques relatively impractical for real-world applications. Similar issues arise with respect to caching as well. Hence, the particular storage format of TRS-BUC provides great advantages to the entire lifecycle of data cubes.

In particular, our contributions can be summarized as follows:

- *Cube construction*: We incorporate redundancy reduction into the dominant ROLAP cube construction methods and devise three new cubing algorithms, which exhibit considerable reduction in the size of the cube to be stored as well as some minor reduction in its construction time. We study their behavior under large and multi-dimensional datasets and show that TRS-BUC is the first ROLAP method that scales well up to at least 25 dimensions.
- *Query answering*: Under a comprehensive query model, which is broader than the models used in the past for the same purpose, we evaluate novel algorithms for answering queries on top of the cubes produced by the new methods and demonstrate that the cube resulting from TRS-BUC exhibits significantly better query execution performance compared to all earlier techniques, including those considered as "champions" with respect to construction time and storage space. Furthermore, we prove that the class of *count iceberg queries* can be answered using TRS-BUC as efficiently as on top of an actual iceberg cube.
- *Incremental maintenance*: We introduce novel incremental update algorithms and demonstrate that those based on TRS-BUC exhibit again significantly better performance compared to their counterparts. Moreover, they produce a cube identical to the one that would be produced by full reconstruction, i.e., TRS-BUC preserves its compact format unaffected, regardless of the frequency of updates.
- *Indexing and caching*: We propose indexing and caching schemes and study their effect on both query

answering and incremental updating of ROLAP cubes. Our experiments show that TRS-BUC is the only known method that can benefit significantly from such techniques, consuming inexpensive additional resources.

Overall, our results give very positive indications on the power of ROLAP.

The rest of this paper is organized as follows: In Sects. 2, 3, and 4, we study the problems of cube construction, query answering, and incremental maintenance, respectively. In Sect. 5, we study indexing techniques for ROLAP cubes using traditional B^+-Trees and bitmap indices, again in the context of the entire cube lifecycle. Finally, in Sect. 6, we present related work and, in Sect. 7, we conclude and exhibit the directions of our future work.

A	B	C	M
3	2	1	30
2	3	1	20
1	2	3	10
1	1	3	50

Fig. 2 Fact table R

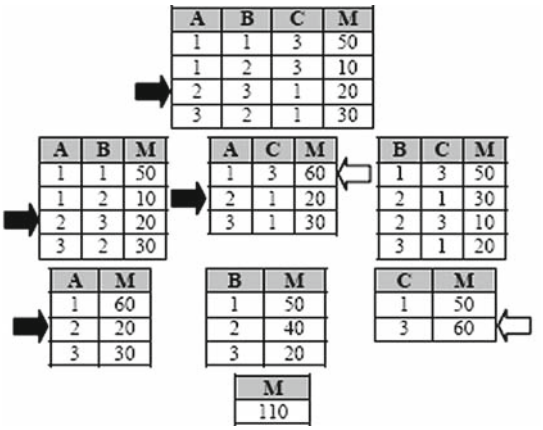


Fig. 3 Cube of R

2 Cube construction

2.1 Cube redundancy

First Kotsis and McGregor [12] and then several other researchers have realized that a great portion of the data in a cube is redundant [5,13,14,24,28]. Removing it results in a condensed, yet still fully materialized cube. Existing methods that avoid redundancy achieve impressive reductions in the size of the cube stored, which benefit computation performance as well, because output costs are significantly lower and in some cases, because early identification of redundant tuples enables pruning of parts of the computation. Additionally, at query time, neither aggregation nor decompression takes place, but only a simple redirection of retrieval to other nodes in the lattice.

In this paper, we distinguish two notions of redundancy: (a) A tuple in a node is *partially redundant* when it can be produced by simple projection of a single tuple from its *parent node* in the cube lattice without aggregation. Partially-redundant tuples are generated by the same set of tuples in the original fact table and have, therefore, the same aggregate values. Such redundant tuples are taken into account by other algorithms as well [5,12–14,24]. (b) A partially-redundant tuple in a node is further characterized as *totally redundant* when it can be produced by projection of a single tuple from *any ancestor node* all the way up to the *original fact table* without aggregation. The concept has also been identified by earlier efforts as well; for example, totally-redundant tuples coincide with those called Base Single Tuples in BU-BST [28]. Both notions of redundancy are better explained with the example below.

Example: Assume a fact table R consisting of three dimensions (A, B, C) and one measure M (Fig. 2). Figure 3 presents the corresponding cube. Tuple $\langle 2, 20\rangle$ in cube node A is totally redundant, since it can be produced by a simple projection of tuple $\langle 2, 3, 1, 20\rangle$ in R. The black arrows in Fig. 3 point at several tuples in an entire subcube where the same information with respect to A is repeated. Note that A = 2 "functionally determines" the values in all other fields, thus the corresponding tuple is redundant in all nodes containing dimension A. Similarly, tuple $\langle 3, 60\rangle$ in C can be evaluated from a simple projection of tuple $\langle 1, 3, 60\rangle$ in its parent node AC (white arrows in Fig. 3). However, C = 3 does not "functionally determine" the values in all the other fields, hence, tuple $\langle 3, 60\rangle$ is partially but not totally redundant.

Note that the notion of a partially-redundant tuple denotes a relationship between the tuple's node and its parent only. No conclusions can be drawn about other ancestor nodes in higher lattice levels. On the contrary, the definition of a totally-redundant tuple implies (by induction) that it is totally redundant in all of its ancestors up to the lattice root and is also found in the original fact table. Hence, bottom–up computation methods are efficient mainly in discovering totally-redundant tuples, since they can construct each node from the fact table, while top–down methods can exploit all partial

redundancy, since they use parent–child relationships. Our methods avoid such redundancies, as described below.

2.2 Suite of algorithms investigated

According to published comparisons, the most efficient methods in the ROLAP context that construct full and uncompressed cubes are Partitioned-Cube (PC) [21] and BottomUpCube (BUC) [2], which compute the cube by traversing the cube lattice in a top–down and a bottom–up manner, respectively. They both apply recursive partitioning on the original data until each generated segment fits in main memory, making effective use of resources and providing robustness and reasonable scaling.

Inspired by the discussion in Sect. 2.1, we combine the "champions" in cube computation PC and BUC with redundancy reduction and derive three new cubing algorithms: PRT-PC, PRS-PC, and TRS-BUC. The first two are derivatives of (top–down) PC and exploit all partial redundancy, while the third one is a derivative of (bottom–up) BUC and exploits total redundancy.

We also examine two earlier ROLAP techniques that exploit redundancy: BU-BST [28] and QC-DFS [13]. They are both similar to TRS-BUC, in that they are recursive algorithms based on BUC that take advantage of total redundancy (QC-DFS deals, in fact, with all partial redundancy and more). As explained in Sect. 2.5, however, they both differ from TRS-BUC in some aspects that prove crucial for performance overall.

2.3 Algorithms PRT-PC and PRS-PC

Partially-Redundant-Tuple-PC (PRT-PC) and Partially-Redundant-Segment-PC (PRS-PC) are two cubing methods that combine the efficiency of PC [21] with redundancy reduction. PC follows a divide-and-conquer strategy when the input fact table does not fit in memory: at appropriate points of its execution, it uses an efficient hash-based method to recursively partition the fact table into fragments that fit in main memory and performs sorting and aggregation on each memory-sized fragment independently. The hash-based method guarantees that all tuples that need to be aggregated together, i.e., tuples with the same values in the dimensions that are processed at that point, are assigned into the same partition. Similarly to all cubing approaches presented in the literature, PC makes the realistic assumption that no such group of tuples is so big that it cannot fit in memory. This implies essentially that there is no single value in any dimension that has more tuples associated with it than what fits in memory. For each memory-sized partition InRel, algorithm Memory-Cube is called (Algorithm 1 without the gray lines), which is the heart of PC and performs aggregation. Memory-Cube computes an entire subcube inside main memory, using pipelining to construct entire lattice paths of nodes that share a common prefix. In this way, it shares sorting costs and minimizes computational penalties.

Our modifications to the original Memory-Cube algorithm, denoted with gray color in Algorithm 1, are based on the observation that identification of partially-redundant tuples can be easily incorporated into it at almost no extra cost. In particular, Memory-Cube takes a prefix-ordered path and sorts the in-memory relation segment InRel according to the attribute ordering of the first node of the path (line 1). It then makes a single scan through the data (starting at line 9), storing into accumulator variables aggregates at all granularities found on the path (starting at line 18). Aggregates at finer granularities are combined with those at coarser granularities (line 26) when the corresponding grouping attributes change (tested in line 19). Results are output immediately with no check for redundancy (in the original algorithm, instead of lines 21–25, a single call to Flush(Accumulators[i]) exists), so this is the part where we have intervened. After sorting the tuples (line 1) and starting the pipelined computation of an entire lattice path (line 3), we identify tuples that propagate from parent to child without being aggregated. If such a tuple does not match the tuple in the current accumulator (line 19), the accumulator is flushed to disk (lines 22, 24) and the propagated tuple enters the empty accumulator (line 26). In this case, flag isAggregated is set to false (line 27). Otherwise, the matching tuples are aggregated (line 30) and flag isAggregated is set to true (line 31). Note that an aggregated tuple is flushed to disk indeed according to the conventional algorithm (line 22), but its row-id is kept to be used as a reference if the tuple is found redundant in descendant nodes. On the contrary, non-aggregated tuples are partially redundant and can be immediately sidelined (line 24) and substituted by references to the ancestor node where the original tuple is actually stored; this is the reason why we keep track of each tuple's AnscestorRef (lines 17, 22). These references can be implemented using row-ids, in purely relational fashion. (In this paper, we assume that the underlying system supports row-ids, which is common in practice. Otherwise, an additional integer column in the fact table would be required, which would hold a unique number for every tuple, e.g., taken from a sequencial number generator, playing the role of a row identifier. We do not study this alternative any further.)

Figure 4 shows the *PRT-PC cube* (the cube constructed by PRT-PC) of R (Fig. 2). To facilitate under-

Algorithm 1 PRT/PRS-PcMemoryCube(InRel, GroupingAttrs)

```
1:  Sort(InRel);
2:  CombineTuplesSharingAllValues(InRel, GroupingAttrs);
3:  for each sort order SO do
4:     InitializeAccumulators(Accumulators);
5:     t = GetFirstTuple(InRel);
6:     Combine(t, Accumulators[0]);
7:     previous_t = t;
8:     startSortingFrom_t = t;
9:     for each subsequent tuple t in InRel do
10:       j = FindFirstDifferentSortOrderAttribute(t, previous_t);
11:       if j < GetCommonPrefixSize(SO, NextSOrder(SO)) then
12:          SortSegment(InRel, startSortingFrom_t, previous_t, NextSOrder(SO));
13:          startSortingFrom_t = t;
14:       end if
15:       if HasDiffInGroupingAttr(t, Accumulators[0]) then
16:          propagated_t = Accumulators[0]; {Propagate this tuple to the next level}
17:          propagated_t.AnscestorRef = Flush(Accumulators[0]); {Keep tuple's row-id}
18:          for each subsequent Accumulators[i] do {For all descendants in the lattice path}
19:             if HasDiffInGroupingAttr(propagated_t, Accumulators[i]) then
20:                current_t = Accumulators[i];
21:                if Accumulators[i].isAggregated then {Flush a "normal" tuple to disk and keep its row-id to use as a
                   reference if it is found partially redundant in descendant nodes}
22:                   current_t.AnscestorRef = Flush(Accumulators[i]);
23:                else {Redundant tuple found. Flush its ancestor's rowid (AnscestorRef)}
24:                   FlushRedundant(Accumulators[i]);
25:                end if
26:                Combine(propagated_t, Accumulators[i]); {Accumulator has been flushed and a new tuple is added}
27:                Accumulators[i].isAggregated = false;
28:                propagated_t = current_t; {Propagate current t to the next level}
29:             else
30:                Combine(propagated_t, Accumulators[i]);
31:                Accumulators[i].isAggregated = true; {Aggregation took place}
32:                breakLoop; {Break tuple propagation}
33:             end if
34:          end for
35:       end if
36:       Combine(t, Accumulators[0]);
37:       previous_t = t;
38:    end for
39:    FinalizeAccumulators(Accumulators);
40: end for
```

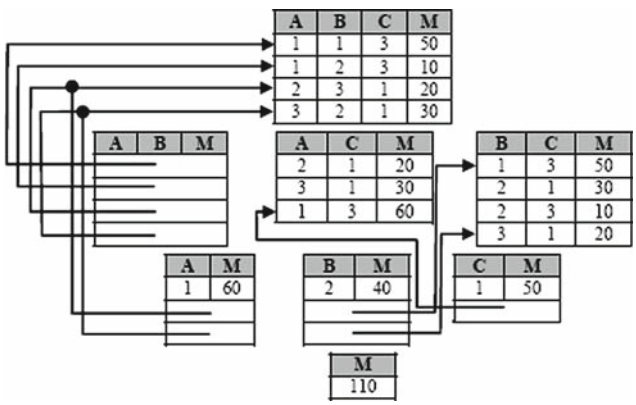


Fig. 4 The PRT-PC cube of R

standing of this figure and the underlying algorithm, Fig. 6 describes the construction steps according to PRT-PC assuming that the entire fact table fits in one partition. In the description of the algorithm's actions, acc-X stands for accumulator X. In Fig. 4, note the large number of partially-redundant tuples that have been replaced by mere row-id references, depicted as arrows pointing to tuples in ancestor nodes. These references are actually stored in different tables (to preserve relational compatibility), but we show them within the normal views for simplicity. Note also that, for the construction of this cube, three paths are traversed (in accordance with the original PC algorithm): **ABC** → AB → A → Ø, **CA** → C, and **BC** → B. Figure 6 illustrates the construction of all tuples in the first path only. The rest is similar and, thus, omitted. Finally, note that the root node of each path has no parent, thus nodes ABC, AC, and BC include no references in Fig. 4.

PRS-PC extends PRT-PC, identifying subsets of consecutive partially-redundant tuples, called *partially-*

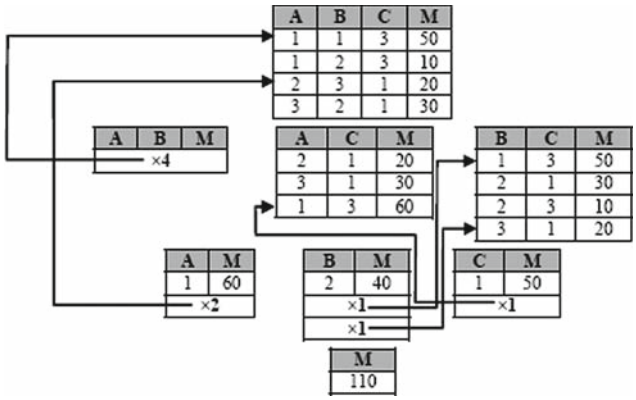


Fig. 5 The PRS-PC cube of R

redundant segments. By grouping such tuples into a segment (performed in FlushRedundant, line 24), PRS-PC substitutes several row-id references with a single one (pointing to the first tuple of the segment) and a count (denoting the segment size), thus achieving greater compactness at no extra cost. Figure 5 presents the *PRS-PC cube* of R. The extension of PRT-PC to PRS-PC is straightforward; hence we omit a detailed illustration of the algorithm like the one in Fig. 6.

We should note that there are aggregate functions that are not amenable to such processing. In general, three types of aggregate functions are recognized [6]: *distributive* (e.g., count, sum, min, max), *algebraic* (e.g., avg), and *holistic* (e.g., median). Aggregates at higher lattice levels can be used to compute aggregates at lower levels only for distributive and algebraic functions. Like PC, PRT/PRS-PC take advantage of aggregates already computed, hence, they can evaluate distributive and algebraic functions, but not holistic ones.

2.4 Algorithm TRS-BUC

Totally-Redundant-Segment-BUC (TRS-BUC) combines BUC [2] with elimination of *totally-redundant segments* (subsets of consecutive totally-redundant tuples) in the same spirit as BU-BST [28]. Our modifications to the original BUC algorithm are once more denoted with gray color in Algorithm 2. Similarly to all other BUC-based methods presented in the next subsection, TRS-BUC proceeds recursively from the bottom of the lattice and works its way up towards the larger, less aggregated nodes, in a depth-first manner.

Initially, a main function partitions the original fact table into pieces that fit in main memory (using a hash-based method that is similar to the one used by PC, as explained in the previous subsection), and then, for each partition InRel, it calls TRS-BUC (Algorithm 2) with input parameters InRel and dim = 0. In the first step, TRS-BUC aggregates all measures of the entire InRel

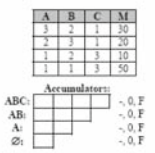
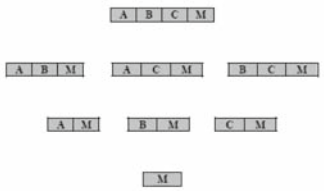
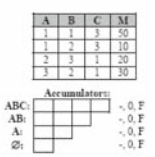
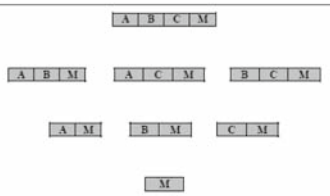
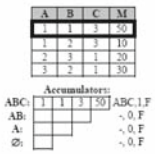
Step	Action	Memory Image	Disk Image
1	Load fact table into main memory. Start with path ABC→AB→A→Ø and initialize four accumulators (one for each node in the path). The size of each accumulator is equal to the size of a tuple that belongs to the corresponding node. For each accumulator keep the following meta-data: an ancestor reference AnscestorRef (consisting of the node name and a row-id) and a flag isAggregated indicating if the tuple has been aggregated in the accumulator. The initial values of the meta-data indicate that the accumulators are empty.		
2	Sort R according to the root node in the current path, i.e., ABC.		
3	Start scanning R. Since acc-ABC is empty, just insert the first tuple of R into it and update its meta-data. The value <ABC, 1, F> of the meta-data indicates that the tuple under construction in the corresponding accumulator is the $\mathbf{1}_{st}$ generated for node **ABC** and has not been aggregated (hence, isAggregated is equal to **F**, i.e., false).		

Fig. 6 **a** Steps 1–3 of PRT-PC applied on R

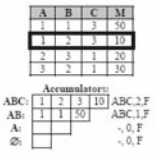
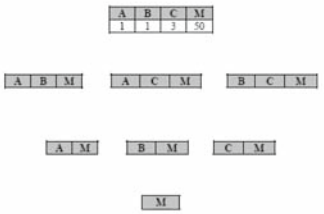
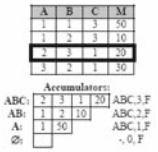
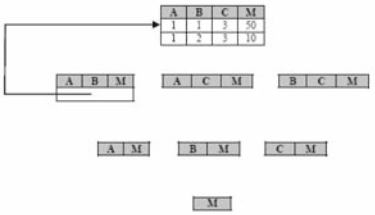
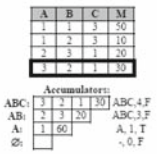
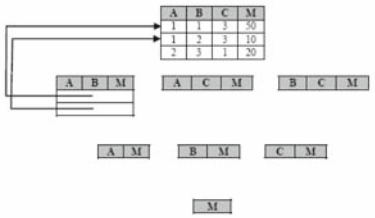
Step	Action	Memory Image	Disk Image
4	Continue with the second tuple of R. Compare its dimension values with the ones of acc-ABC. Since they differ, (a) flush acc-ABC to disk, (b) propagate the relevant part of its contents to acc-AB, and (c) insert the second tuple into acc-ABC and update its meta-data. Although not aggregated, the flushed tuple is stored as normal because ABC is the root of the path, as explained in the text. The same holds for all the tuples stored in node ABC in steps 5, 6, and 7. Since acc-AB was empty, the propagated values from acc-ABC are not propagated any further. The meta-data of acc-AB (<ABC, 1, F>) indicate that the corresponding tuple (<1, 1, 50>) can be produced by a simple projection of the first normal tuple of node ABC, since it has not been aggregated in AB, at least up to now.		
5	Continue with the third tuple of R. Again, since its dimension values differ from those of acc-ABC, (a) flush acc-ABC to disk, (b) propagate the relevant part of its contents to acc-AB, and (c) insert the third tuple into acc-ABC and update its meta-data. Compare the dimension values propagated from acc-ABC to acc-AB with those already in AB. Since they differ, repeat the process of flushing the latter to disk, propagating the relevant part of its contents to acc-A, and inserting the propagated contents of acc-ABC into AB. (The flushed tuple is stored as a reference to the 1_{st} tuple of node ABC, as indicated by the corresponding accumulator meta-data <ABC, 1, F> in step 4.) Again, since acc-A was empty, the propagated values from acc-AB are not propagated any further.		
6	Continue with the fourth tuple of R in identical fashion, flushing and propagating again until acc-A. This time, the dimension values propagated from acc-AB are equal with those already in A (A=1 in both). Hence, perform aggregation, update acc-A, and do not propagate to acc-Ø. Note that the meta-data associated with acc-A (<A, 1, T>) indicate that tuple <1, 60> is the $\mathbf{1}_{st}$ normal tuple that has been aggregated in node **A**. Hence, this time is-Aggregated is set to **T** (true).		

Fig. 6 **b** Steps 4–6 of PRT-PC applied on R

into a single tuple t (line 1). If t is totally redundant (i.e., generated from a single tuple in InRel, tested in line 2), TRS-BUC stores it only into the currently processed node CurrentNode (lines 3–16) and prunes recursion early (line 17). Note that the original BUC algorithm applies a similar optimization technique as well when identifying a cube tuple t generated from a single fact-table tuple. The main difference is in that BUC also projects t and stores it into all the ancestor nodes of CurrentNode in the cube lattice (since, in BUC, instead of lines 3–16, a single call to a function called WriteAncestors exists). Hence, although BUC achieves early pruning of recursive calls as well, it physically stores projections of the same cube tuple into multiple nodes, which is redundant. TRS-BUC not only overcomes this drawback, but it further compresses the final result by grouping subsets of totally-redundant tuples generated by consecutive fact-table tuples into segments. (Recall that PRS-PC applies similar techniques for identifying partially-redundant segments.) To achieve this, TRS-BUC keeps some additional information for each lattice node it processes: (a) The first tuple of the fact table that belongs to the most recently found redundant segment RS (CurrentNode.frstRdntTpl), (b) the last tuple of the fact table identified up-to-now that belongs to RS (CurrentNode.lastRdntTpl), and (c) a counter that tracks the size of RS (CurrentNode.redundantCount). Using this information, whenever TRS-BUC identifies a

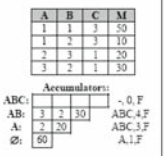
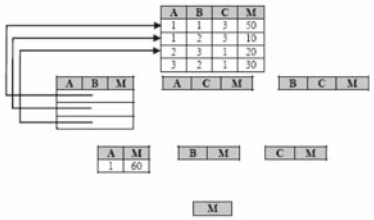
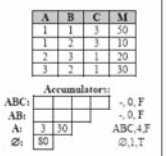
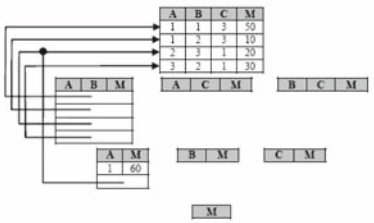
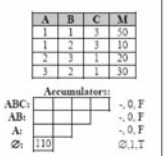
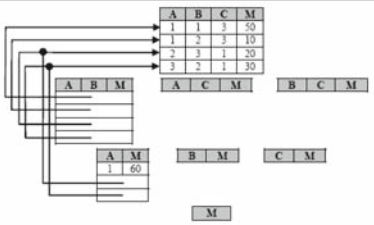
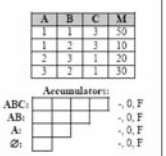
Step	Action	Memory Image	Disk Image
7	Since there are no more tuples in R, start finalizing accumulators through downward propagation all the way through the other end. Flush acc-ABC to disk, propagate the relevant part of its contents to AB, and then make it empty. For both acc-AB and acc-A the dimension values propagated from their parent differ from those already there, hence, flush new tuples to disk (stored in a reference or normal fashion, respectively, as indicated by the meta-data). For acc-Ø, since it was empty, just insert the propagated values from acc-A.		
8	Since acc-ABC is empty (and there are no more tuples in R), flush acc-AB to disk, propagate the relevant part of its contents to acc-A, and then make it empty. Propagation results in one more (reference) tuple flushed to node A, as before. Since acc-Ø has no dimension values, any tuple matches with its contents. Hence, perform aggregation and stop further propagation.		
9	Since acc-ABC and AB are empty (and there are no more tuples in R), flush acc-A to disk, propagate its measure to acc-Ø, aggregate it with the value already there, and then make it empty.		
10	Flush acc-Ø to disk and make it empty. At this point, all tuples in the nodes of the first path ABC→AB→A→Ø have been generated. Computation continues with the next path, CA→C. We omit the subsequent steps, since they are similar to the ones already presented.		

Fig. 6 **c** Steps 7–10 of PRT-PC applied on R

totally-redundant tuple t (line 2), it also checks whether or not it is a neighbor of the marginal tuples of RS (line 3). If it is, then t belongs to RS and the algorithm modifies the meta-data associated to RS (lines 4–9) and updates the previously stored redundant tuple (line 10). Otherwise, a new redundant segment has been found. In this case, TRS-BUC updates the corresponding variables to keep track of the newly discovered redundant segment (lines 12–14) and flushes to disk redundancy information consisting of a row-id reference pointing to Current-Node.frstRdntTpl and a count that denotes the size of RS (line 15).

If t is not found to be totally redundant in line 2, TRS-BUC flushes a normal tuple to disk (line 19) and proceeds recursively, like BUC. For each dimension d between dim and numDims (line 20), it sorts its input relation InRel with respect to dimension d (line 22) and partitions it into disjoint partitions (line 23), each of which contains the tuples in InRel that share the same value on d. Note that BUC-based algorithms do not sort the original fact table in the beginning according to all its dimensions. Instead, in each recursive call, they sort independently the input relation InRel according to several individual dimensions iterating over d (line 22). After the execution of Partition (line 23), dataCount[d] contains the number of records for each distinct value of the dimension d. Line 25 iterates through the partitions, which one after the other become the input relation to the next recursive call (line 28), driving computation to nodes in upper lattice levels. Once the algorithm has processed all the partitions, it repeats the entire procedure for the next dimension.

Algorithm 2 TRS-BUC(InRel, dim)

```
1:  Aggregate(InRel, t); {Places result of aggregation in tuple t}
2:  if InRel.count() == 1 then {if t is totally redundant}
3:    if Consecutive(t, CurrentNode.frstRdntTpl) || Consecutive(t, CurrentNode.lastRdntTpl) then {t belongs to an
      already found TR segment. Update redundancy information}
4:      CurrentNode.redundantCount++;
5:      if t < CurrentNode.frstRdntTpl then
6:        CurrentNode.frstRdntTpl = t;
7:      else
8:        CurrentNode.lastRdntTpl = t;
9:      end if
10:     UpdatePreviousRedundant(CurrentNode.frstRdntTpl, CurrentNode.redundantCount);
11:   else {New redundant segment found}
12:     CurrentNode.frstRdntTpl = t;
13:     CurrentNode.lastRdntTpl = t;
14:     CurrentNode.redundantCount = 1;
15:     WriteRedundant(CurrentNode.frstRdntTpl, CurrentNode.redundantCount);
16:   end if
17:   return; {Prune recursion}
18: end if
19: WriteNormal(t); {t is not totally redundant. Write it as normal}
20: for (d = dim; d < numDims; d++) do {Proceed bottom–up}
21:   C = cardinality[d];
22:   Sort(InRel, d);
23:   Partition(InRel, d, C, dataCount[d]);
24:   k = 0;
25:   for (i = 0; i < C; i++) do
26:     c = dataCount[d][i];
27:     t.dim[d] = InRel[k].dim[d];
28:     TRS-BUC(InRel[k ... k+c], d+1); {Recursive call}
29:     k += c;
30:   end for
31:   t.dim[d] = ALL;
32: end for
```

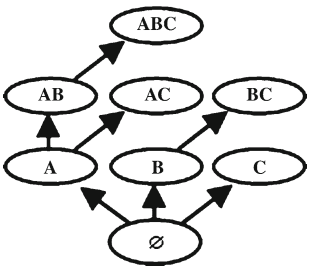


Fig. 7 Execution plan of BUC-based methods

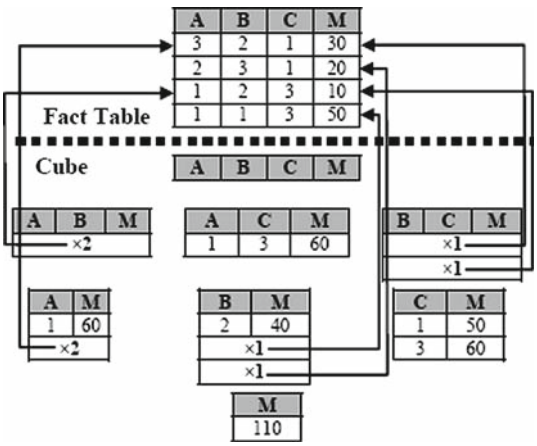


Fig. 8 Fact table R and the corresponding TRS-BUC cube

Figure 7 indicates the plan of recursive calls of TRS-BUC as it traverses the cube lattice of R in a bottom–up and depth-first manner. Figure 8 presents the *TRS-BUC cube* of R (Fig. 2). Note that, unlike Figs. 4 and 5, Fig. 8 does not only show a cube, but also the corresponding fact table (appearing on the top). This is necessary, since TRS-BUC is the only method that uses references pointing to tuples in the fact table. Furthermore, it stores redundant tuples only in the most specialized node they belong to (as also mentioned above), where they are implicitly shared between this node and its ancestors with which it has a common prefix. For instance, the ⟨*row-id reference*, *count*⟩ pair stored in node A (Fig. 8) is not repeated in nodes AB, AC and ABC. Hence, node AB contains virtually four tuples in total (two of them are pointed by a reference physically stored in it and two more are "inherited" from node A). Similarly, node

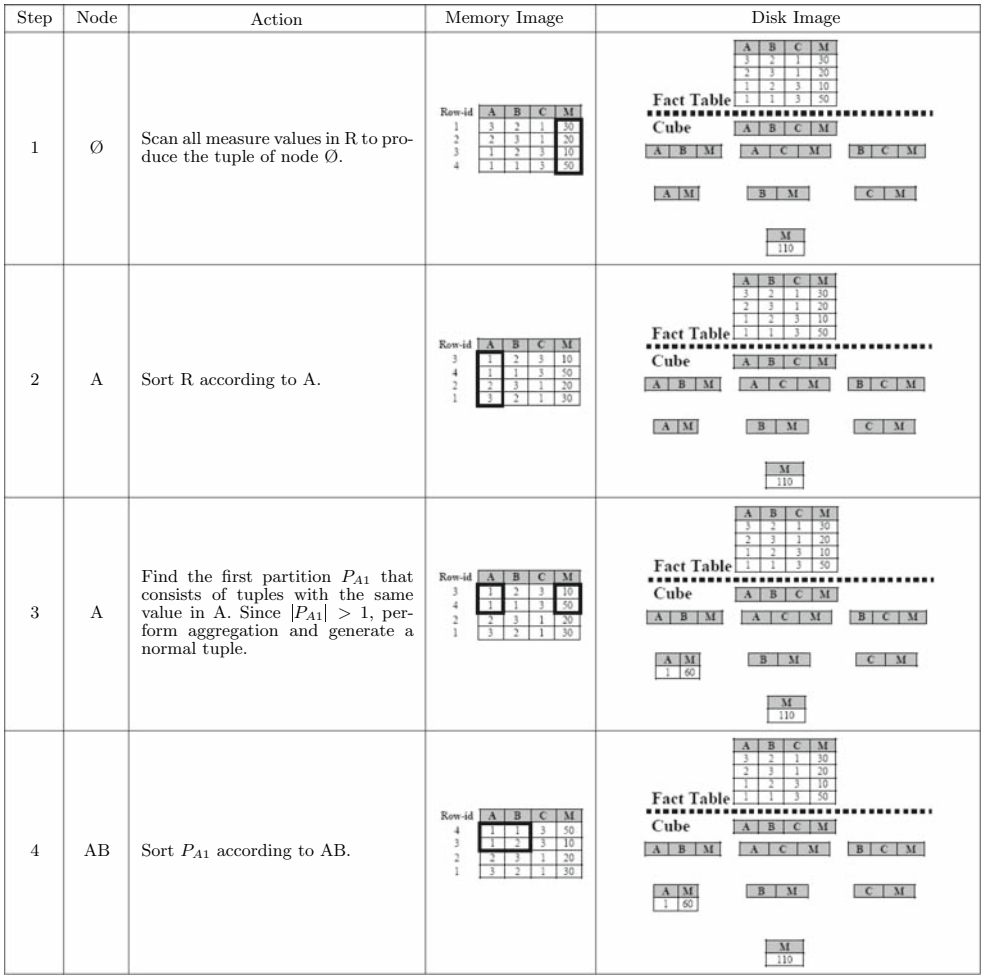


Fig. 9 **a** Steps 1–4 of TRS-BUC applied on R

ABC, which appears empty in Fig. 8, contains four tuples as well, all "inherited" from nodes AB and A. Unlike redundant tuples, normal tuples are not shared among different nodes. Note that, as in Fig. 5, in Fig. 8, we show ⟨*row-id reference*, *count*⟩ pairs within the normal views only for simplicity. Such pairs are actually stored in different tables. The semantics of these pairs are similar to the ones of Fig. 5. Each row-id reference points to the first tuple of the corresponding totally-redundant segment in the original fact table, while count holds the size of this segment. They slightly differ in that, as explained above, such a pair in TRS-BUC always points to a segment of tuples in the original fact table (not in any other cube node) and is only stored in the most specialized node it belongs to (not in any of its ancestors).

Finally note that, as with other BUC-based methods, TRS-BUC computes each aggregate value from scratch, without using other values already computed. Hence, it can deal with all types of aggregate functions, including holistic ones.

Figure 9 is similar to Fig. 6 and follows the construction of the TRS-BUC cube of R shown in Fig. 8, assuming that all of R fits in main memory. In the description of the algorithm's actions, TR stands for 'totally-redundant'.

2.5 Algorithms BU-BST and QC-DFS

Unlike PC (which has not been studied before in conjunction with redundancy reduction), BUC has already been extended by previous efforts in order to deal with redundancy as well. Within the ROLAP world, the potential competitors of our BUC-based method (TRS-BUC) are Bottom-Up-Base-Single-Tuple (BU-BST) [28] and Quotient-Cube-Depth-First-Search (QC-DFS) [13]. The former identifies totally-redundant tuples (exactly the same as TRS-BUC) producing a condensed cube, hereafter called BU-BST cube. The latter is an extension of BU-BST that identifies at least all partially-redundant tuples in the cube. In particular, it takes all cube tuples

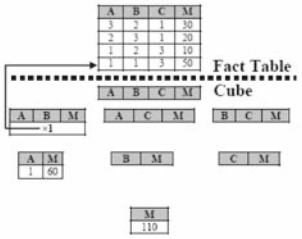
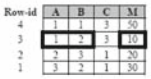
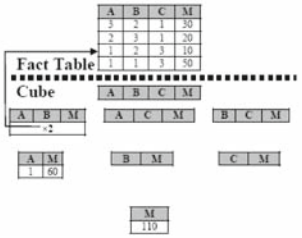
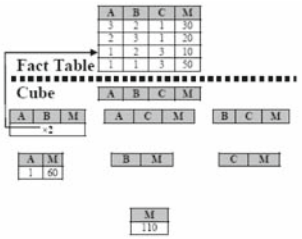
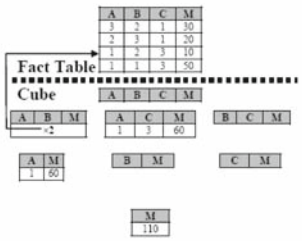
Step	Node	Action	Memory Image	Disk Image
5	AB	Find the first partition P_{A1B1} that consists of tuples with the same value in AB. Since $\|P_{A1B1}\| = 1$ generate a TR tuple and prune recursion (do not proceed to ABC).		
6	AB	Find the second partition P_{A1B2} that consists of tuples with the same value in AB. Since $\|P_{A1B2}\| = 1$ generate a TR tuple and prune recursion (do not proceed to ABC). Note that the tuple flushed to disk in this step is consecutive with the previous one in the fact table. Hence, we update only a reference and a count.		
7	AC	Since all tuples in partition P_{A1} have been processed according to AB, proceed to node AC. Hence, sort P_{A1} according to AC.		
8	AC	Find the first partition P_{A1C1} that consists of tuples with the same value in AC. Since $\|P_{A1C1}\| > 1$ perform aggregation and generate a normal tuple.		

Fig. 9 **b** Steps 5–8 of TRS-BUC applied on R

that are produced by the aggregation of the same set of tuples in the original fact table (which include all partially-redundant tuples) and assigns them into an equivalence class of cells with identical aggregate value. By doing so for all such sets of tuples, QC-DFS generates a so-called Quotient Cube and stores it in a relational table, called QC-Table [13].

Figures 10 and 11 present the *BU-BST cube* and *QC-Table* of R (Fig. 2), respectively. In the BU-BST cube, *CID* denotes the node to which a normal (non-redundant) tuple belongs and *SID* the most specialized node (further to the top) to which a redundant tuple belongs. In the QC-Table, *Class-id* stores a unique identifier for each class of equivalent tuples and *Lower-Bounds* stores an expression[1] that encaptulates the lower boundaries of the class, i.e., how far down the cube lattice, in all paths, the corresponding QC-Table tuple can be used as is (i.e., by simple projections on its dimension values). For example, the class with id = 6 (in Fig. 11) actually represents five redundant tuples in Fig. 3 ($\langle 3,2,1,30\rangle$ in node ABC, $\langle 2,1,30\rangle$ in BC, $\langle 3,1,30\rangle$ in AC, $\langle 3,2,30\rangle$ in AB, and $\langle 3,30\rangle$ in A). These are the tuples produced by the tuple $\langle 3,2,1,30\rangle$ stored in the 6th row of the QC-Table itself, by projecting A out of that tuple (as indicated by the A part in the Lower-Bounds expression A $\vee$ BC), and by projecting B, C, and BC out of it (as indicated by the BC part in the expression A $\vee$ BC). Note that, in this example, we have computed a Quotient Cube with respect to cover partition [13], which seems more interesting in practice (refer to the original paper [13] for more details).

Although BU-BST and QC-DFS are similar to TRS-BUC in that they are all derivatives of BUC that handle redundancy, they also have the following

[1] Given the non-atomic nature of the logical expressions in the Lower-Bounds attribute, QC-DFS may be perceived as not purely ROLAP, but we consider this a minor point.

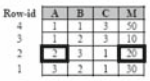
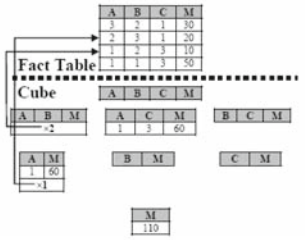
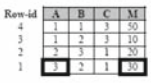
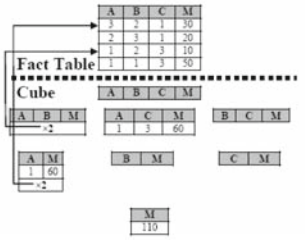
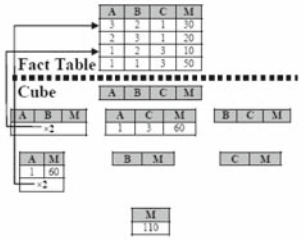
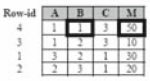
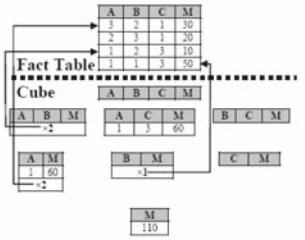
Step	Node	Action	Memory Image	Disk Image
9	A	Since all tuples in partition P_{A1} have been processed according to both AB and AC, go back to node A and find the second partition P_{A2} that consists of tuples with the same value in A. Since $\|P_{A2}\| = 1$, generate a TR tuple and prune recursion (do not proceed to AB or AC).		
10	A	Find the third partition P_{A3} that consists of tuples with the same value in A. Since $\|P_{A3}\| = 1$, generate a TR tuple and prune recursion (do not proceed to AB or AC). Note that the tuple flushed to disk in this step is consecutive with the previous one. Hence, we update only a reference and a count.		
11	B	Since all tuples in R have been processed for all nodes that contain A in their grouping attributes, proceed with B and sort R according to B.		
12	B	Find the first partition P_{B1} that consists of tuples with the same value in B. Since $\|P_{B1}\| = 1$, generate a TR tuple and prune recursion (do not proceed to BC).		

Fig. 9 **c** Steps 9–12 of TRS-BUC applied on R

differences with it, which prove crucial for performance overall:

- TRS-BUC groups each subset of (consecutive) redundant tuples into a segment and does not treat each such tuple individually.
- It does not physically store the tuples that belong to a redundant segment, but substitutes them with a single row-id reference (pointing to the first tuple of the segment) and a count (holding the segment size), in the spirit of PRS-PC. The storage cost of such references is only a few bytes, quite small compared to the size of the actual segment. More interestingly, the use of row-id references redirects a large number of disk accesses during cube operations to a single relation (the fact table), which benefits query processing and incremental updating as well, as we show in the following sections. (Recall that row-id references in TRS-BUC always point to tuples in the fact table, as described in Sect. 2.4.)
- It stores each lattice node as a separate view and does not use a single relation to model the entire cube. Thus, it achieves better clustering and avoids storage of missing dimensions (i.e., *-values).

Initial experiments to assess the impact of the above differences have shown that TRS-BUC outperforms its counterparts in both construction time and storage space requirements. This is illustrated in Figs. 12 and 13, which show, respectively, the cube construction time and storage space when the number of tuples in the original fact table ranges from 250,000 to 750,000. The particular experiments were on a six-dimensional projection of SEP85L [7], which is a commonly used real-world dataset, as also explained in the next subsection.[2]

Clearly, with respect to construction time (Fig. 12), TRS-BUC and BU-BST are quite similar, while they

[2] The data for this experiment and the executable that generated QC-Tables were kindly provided by Li et al. [15,16].

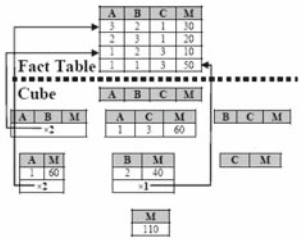
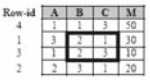
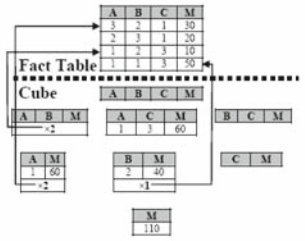
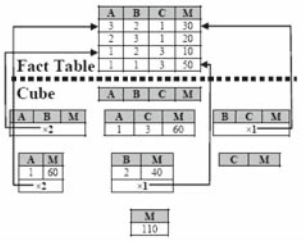
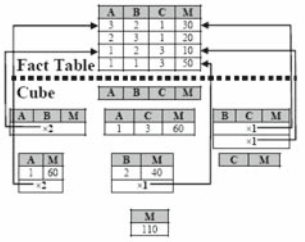
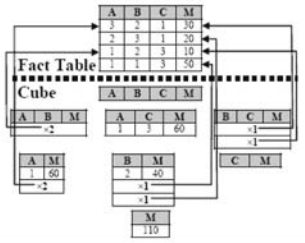
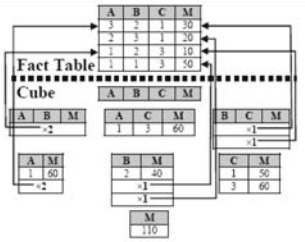
Step	Node	Action	Memory Image	Disk Image
13	B	Find the second partition P_{B2} that consists of tuples with the same value in B. Since $\|P_{B2}\| > 1$, perform aggregation and generate a normal tuple.		
14	BC	Sort P_{B2} according to BC.		
15	BC	Find the first partition P_{B2C1} that consists of tuples with the same value in BC. Since $\|P_{B2C1}\| = 1$, generate a TR tuple. (BC has no ancestors in the execution plan (Figure 7); hence, recursion would be pruned anyway.)		
16	BC	Find the second partition P_{B2C2} that consists of tuples with the same value in BC. Since $\|P_{B2C2}\| = 1$, generate a TR tuple. (BC has no ancestors; hence recursion would be pruned anyway.) Unlike the situation in step 10, this tuple is not consecutive to the previous one, hence, the reference has to be stored separately.		
17	B	Since all tuples in partition P_{B2} have been processed according to BC, go back to node B and find the third partition P_{B3} that consists of tuples with the same value in B. Since $\|P_{B3}\| = 1$, generate a TR tuple and prune recursion (do not proceed to BC).		
18	C	Since all tuples in R have been processed for all nodes that contain B in their grouping attributes, proceed with C and sort R according to C. In a similar spirit, the two (normal) tuples in node C are found. Note that C has no ancestors in the execution plan (Figure 7), hence, recursion stops.		

Fig. 9 **d** Steps 13–18 of TRS-BUC applied on R

both outperform QC-DFS. This is attributed to the fact that they discover only totally-redundant segments and tuples, respectively, whose identification is inexpensive in BUC-like methods. On the contrary, QC-DFS searches for cube tuples generated by the same set of fact-table tuples (these cube tuples include but are not

A	B	C	M	SID	CID
*	*	*	110		∅
1	*	*	60		A
1	1	3	50	AB	
1	2	3	10	AB	
1	*	3	60		AC
2	3	1	20	A	
3	2	1	30	A	
*	1	3	50	B	
*	2	*	40		B
*	2	1	30		BC
*	2	3	10		BC
*	3	1	20	B	
*	*	1	50		C
*	*	3	60		C

Fig. 10 The BU-BST cube of R

Class-id	A	B	C	M	Lower-Bounds
1	*	*	*	110	∅
2	1	*	3	60	A ∨ C
3	1	1	3	50	AC
4	1	2	3	10	A ∨ C
5	2	3	1	20	AC ∨ BC
6	3	2	1	30	A ∨ BC
7	*	2	*	40	∅
8	*	*	1	50	∅

Fig. 11 The QC-Table of R

limited to partially-redundant tuples), incurring significant additional computational costs. With respect to storage space (Fig. 13),[3] although QC-DFS identifies more redundant tuples, the final QC-Tables are larger than the corresponding TRS-BUC cubes. The main reasons correspond to our observations above, i.e., using *D*-dimensional representations for redundant tuples instead of small references, storing additional attributes (Class-id and Lower-Bounds, which are both necessary for answering queries), and physically storing even missing (i.e., projected-out) dimensions. On the other hand, the BU-BST cubes are even larger than the QC-Tables, suffering from the same overheads but also from identifying only total redundancies.

Based on the above, it appears that QC-DFS dominates BU-BST on storage space (roughly by a factor of 2 in the dataset examined) but is significantly inferior on construction time (by almost an order of magnitude), probably losing in the overall tradeoff one might say. This has been an observation of the creators of QC as well; they recognized the deficiencies of a relational approach to their concept and, soon after its

[3] We believe that the final cube size is a more representative metric of space requirements than the number of tuples materialized, which has been used elsewhere [5], since to a large extent, performance is determined by the exact footprint of the cube on the disk.

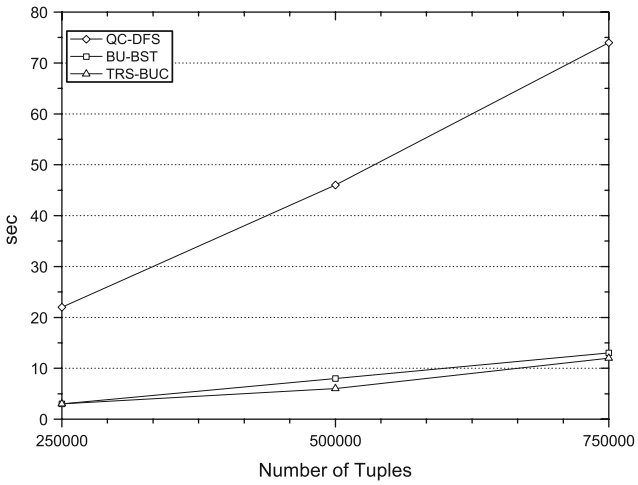


Fig. 12 Comparison of BUC-based methods that remove redundancy—construction time

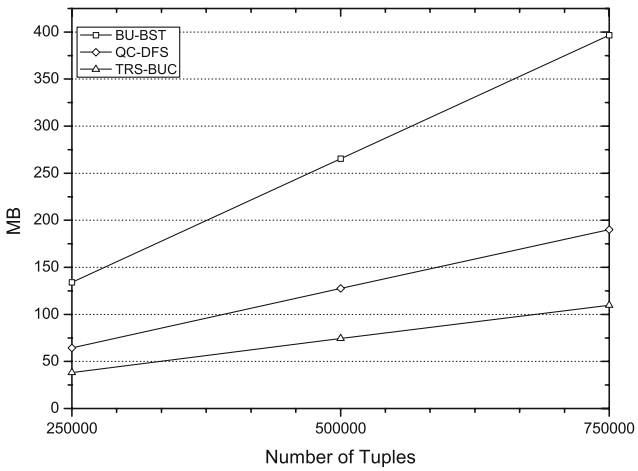
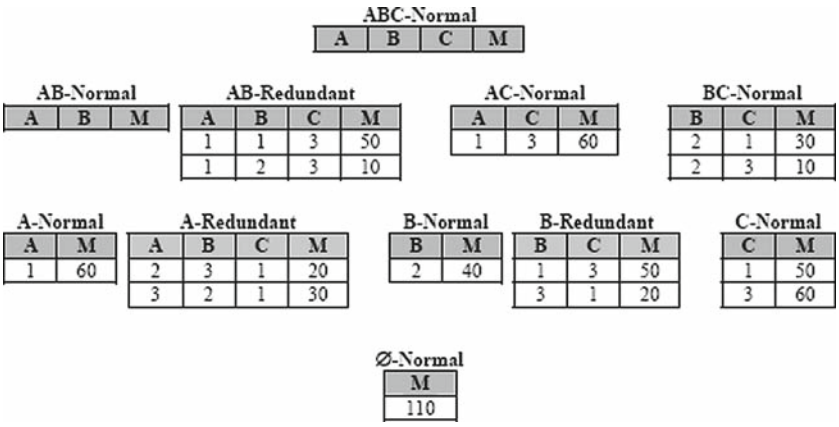


Fig. 13 Comparison of BUC-based methods that remove redundancy—storage space

original introduction [13], they proposed a specialized, non-ROLAP, data structure, called QC-Tree [14], to represent Quotient Cubes efficiently. QC-Trees have not only offered to Quotient Cubes faster construction and greater compression than QC-Tables, but also opportunities for efficient indexing, query answering, and incremental maintenance, which do not seem to exist with QC-Tables. At least with respect to incremental maintenance of QC-Tables, the best update algorithm that has been presented [16] requires a full scan of the data cube for each new insertion into the fact table, which is prohibitive for real-world datasets.

Given the apparent favorable construction-time/storage-space tradeoff for BU-BST and the fact that QC-DFS is not the ideal realization of Quotient Cubes, we have concentrated on BU-BST as the main repre-

Fig. 14 The BU-BST+ cube of R



sentative of earlier ROLAP methods that handle redundancy and have not considered QC-DFS any further.

Among the three differences between TRS-BUC and BU-BST mentioned above, we consider the two first as most significant. As we show in the following sections, substituting a large number of tuples with row-id references pointing to segments of tuples in the fact table gives TRS-BUC great advantages in all aspects of the cube lifecycle. The third difference, concerns the relatively straightforward storage of cube nodes as separate views. This approach to storage has actually been incorporated into a version of BU-BST as well at some point [4]. In this paper, we use BU-BST+ to denote this version of the algorithm and examine it separately in order to compare the gains that arise from just this optimization with those that arise from all three optimizations together, as incorporated into TRS-BUC. Figure 14 illustrates the *BU-BST+ cube* of R (Fig. 2). Note that every node consists of two views, one for the normal tuples and one for the redundant ones. Exceptions to this rule are nodes that contain the right-most dimension in their grouping attributes (in our example dimension C), i.e., nodes C, AC, BC, and ABC. These nodes have no ancestors in the BUC-based execution plan (Fig. 7), hence storing tuples in them as redundant offers no benefits to BU-BST+. Furthermore, note that redundant tuples in the cube format of BU-BST+ are actually longer than the normal ones stored in the same node. This is attributed to the fact that (as also mentioned above) totally-redundant tuples are stored only in the most specialized node to which they belong and are shared between this node and its ancestors in the execution plan (Fig. 7). Since the most specialized node is the one with the smallest number of grouping attributes, more dimension values need to be stored there, otherwise information would be lost and redundant tuples would not be restorable in the ancestor nodes. Interestingly, TRS-BUC does not suffer from either problem, due to the use of row-ids.

2.6 Experimental evaluation

We have used C++ to implement seven algorithms[4] to study the effects of redundancy reduction on the most efficient, purely ROLAP, computation methods: BUC, BU-BST, BU-BST+, *TRS-BUC*, PC, *PRT-PC*, and *PRS-PC*. The first four are in the BUC-family and the remaining three in the PC-family (the new algorithms are in *italics*). The fact that C++ has a hard limit of 2,048 (2^{11}) files that can be simultaneously open in a process causes some overhead to all algorithms except BU-BST for opening and closing an exponential (in the number of dimensions) number of files, potentially multiple times. Fortunately, all these algorithms exhibit a locality of reference to the nodes they produce, due to their top–down or bottom–up lattice traversal, which ensures that a recently opened file will be used again in the near future with great probability. Hence, we have used a heuristic policy of closing the least-recently opened file to remain under the limit of 2,048, which has minimized the cost of these file operations.

We have run our experiments on a Pentium 4 (2.8 GHz) PC with 512 MB memory under Windows XP. We have studied the execution time and result size of the seven methods on real and synthetic datasets under different conditions. The real datasets used are CovType [3] and SEP85L[5] [7]. CovType, which describes forest cover-type data, has ten dimensions and 581,012 tuples. The 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, which describes

[4] We have used the same code for components with the same functionality in all algorithms.

[5] SEP85L is also known as weather dataset.

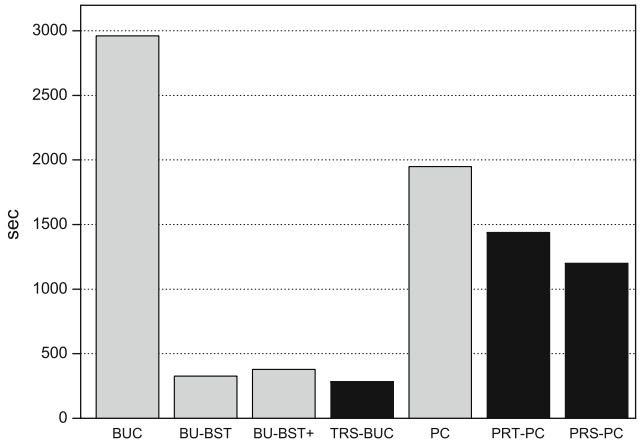


Fig. 15 CovType—construction time

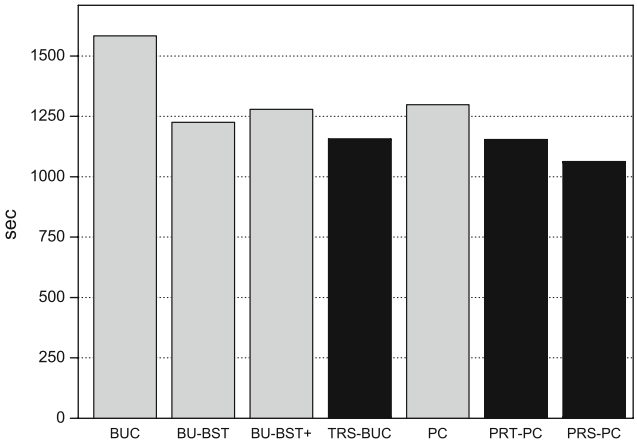


Fig. 17 SEP85L—construction time

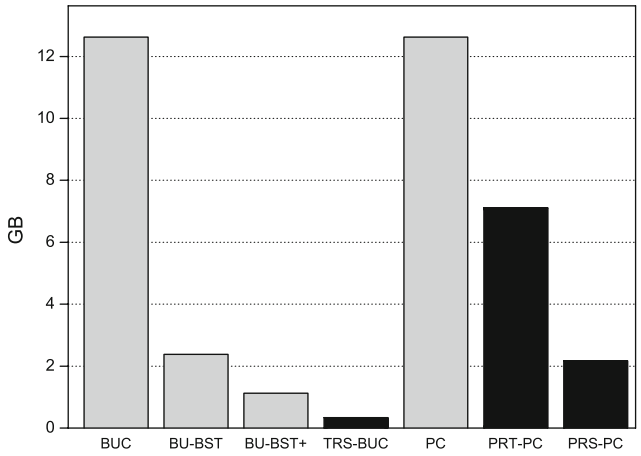


Fig. 16 CovType—storage space

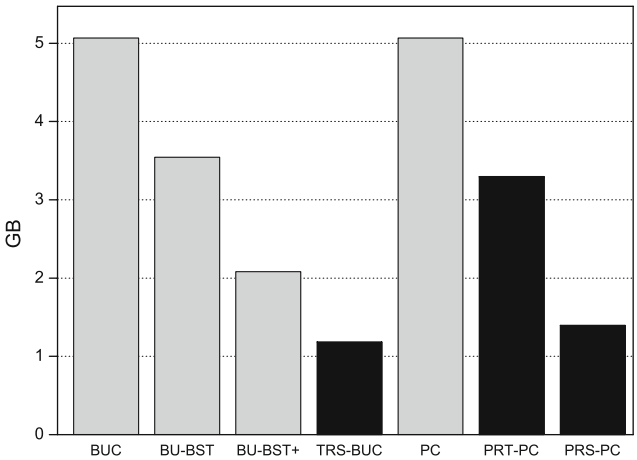


Fig. 18 SEP85L—storage space

surface synoptic weather reports, has nine dimensions and 1,015,367 tuples. The 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 [2] and verified in early experimentation. In Figs. 15–18, we present the behavior of the seven algorithms on these datasets. Our three algorithms are in black color, while the existing ones in gray. These graphs indicate that the proposed algorithms outperform the original ones, both on execution time and on storage space. Clearly, avoiding redundancy has impressive benefits, especially when done at the segment level. Note that, for CovType, TRS-BUC is the undisputed winner, whereas for SEP85L, PRS-PC and TRS-BUC are essentially equivalent. Furthermore, as expected, BU-BST behaves better than BU-BST+ with respect to time, whereas BU-BST+ has an advantage with respect to storage space. Clearly, this is due to the fact that BU-BST+ pays an additional performance penalty for managing multiple files instead of one, like BU-BST, but benefits from that by avoiding the storage of missing dimensions.

To understand the different trends by the various algorithms observed in the two real-world datasets, we have also used synthetic datasets, where each dimension is independent and follows the generalized Zipf distribution [31] sharing the same Z parameter (which affects skew) with all other dimensions. According to this distribution, for a fact table with T tuples, the frequency T_r of the rth most frequent value in the ith dimension is equal to $T_r = T\frac{\frac{1}{r^z}}{\sum_j \frac{1}{j^z}}$, where $j \in [1, C_i]$ and C_i denotes the cardinality of the ith dimension. Furthermore, we have set C_i to be equal to T/i, ordering again the dimensions in decreasing cardinality. In our experiments, we have varied the following parameters: number of dimensions D,

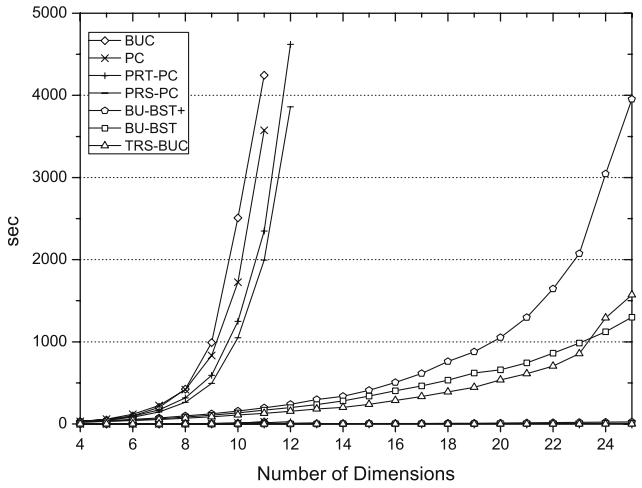


Fig. 19 Synthetic dataset ($T = 5 \times 10^5$, $Z = 0.8$)— construction time

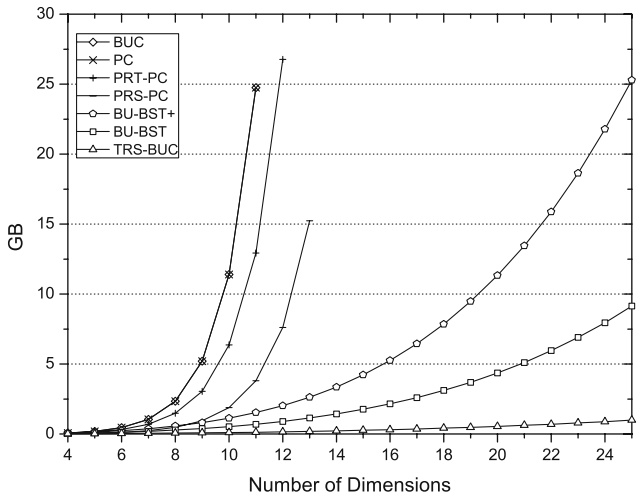


Fig. 20 Synthetic dataset ($T = 5 \times 10^5$, $Z = 0.8$)—storage space

skew in the data Z, and number of tuples in the original fact table T. The results are as follows:

Number of dimensions: We have created moderately sized ($T = 5 \times 10^5$ tuples) and skewed (Zipf factor $Z = 0.8$) datasets while varying the number of dimensions from 4 to 25. To the best of our knowledge, this has been the first attempt to study the behavior of ROLAP cubing methods beyond the limit of ten dimensions. Figures 19 and 20 present the behavior of the seven algorithms under such conditions. The results are an average of five different experimental sets with the same characteristics. TRS-BUC is the only algorithm that handles multi-dimensional datasets well with respect to both time and storage space. BU-BST scales well with respect to time only, whereas BU-BST+ has a moderate behavior on both parameters. The remaining four algorithms have not been tested to the limits, because their output has exceeded our storage capacity (≈ 45 GB), indicating their scalability problems. Beyond $D = 10$, their performance deteriorates, since they create an exponential number of views. BU-BST stores the entire cube as a single relation, so it does not suffer from this. TRS-BUC and BU-BST+ could potentially store an exponential number of views (2^{D+1}) as well, with up to 2^D of them simultaneously open. It has been shown in practice (Fig. 21), however, that the majority of these views is totally redundant and generates no output, turning primarily TRS-BUC and to some extent BU-BST+ into scalable solutions as well. With respect to time, BU-BST slightly outperforms TRS-BUC after about $D = 24$. Beyond this threshold, the cost of managing multiple files starts outweighing the other benefits of TRS-BUC. The difference, however, is still small. On the other hand, TRS-BUC prevails by far with respect to storage space, since the size of the cube it produces is almost unaffected by dimensionality. This is attributed to the use of constant-sized references instead of entire tuple segments. The fact that BU-BST+ does not behave similarly indicates what we have claimed earlier, i.e., that the benefits from storing tuples in multiple views are less significant than the use of references. Even when $D = 25$, the TRS-BUC cube consumes less than 1 GB, whereas BU-BST+, which is the next most efficient technique, consumes more than 9 GB.

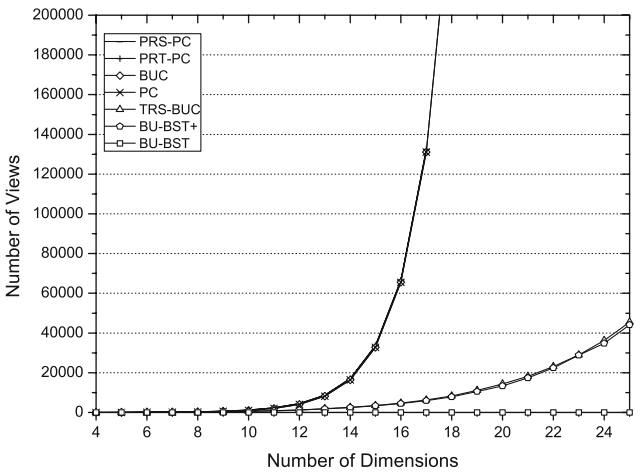


Fig. 21 Synthetic dataset ($T = 5 \times 10^5$, $Z = 0.8$)—number of views

Data skew: We have created 8-dimensional datasets of 5×10^5 tuples with decreasing cardinalities, while varying the Zipf factor Z from 0 (uniform distribution) to 2.4. The behavior of the seven algorithms under such conditions, presented in Figs. 22 and 23, is again an average of five experimental sets. For uniform distributions, the BUC-family is once again the winner. As skew increases, however, all three representatives of the PC-family re-

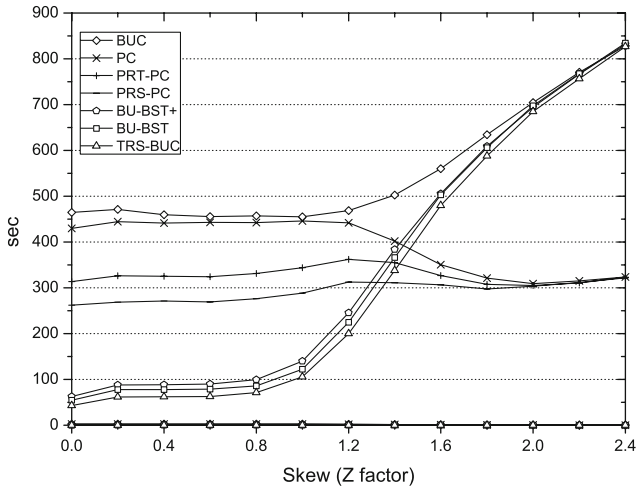


Fig. 22 Synthetic dataset ($T = 5 \times 10^5$, $D = 8$)— construction time

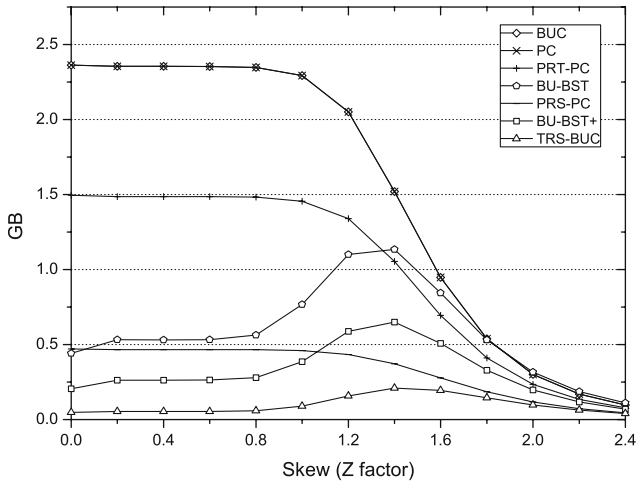


Fig. 23 Synthetic dataset ($T = 5 \times 10^5$, $D = 8$)—storage space

main almost unaffected and even seem to accelerate in really high-skew datasets, in contrast to their counterparts, which slow down considerably. The reason is that, when skew increases, some very dense areas are created within the cube (together with some very sparse ones, which do not affect efficiency, however). Formation of such dense areas results in heavy aggregations and less redundancy. By traversing the lattice in a top–down fashion, the PC-family takes advantage of results already computed and performs aggregation efficiently. On the contrary, the BUC-family wastes time in computing the same aggregations multiple times, since as mentioned already, nodes are constructed from scratch with no reuse of intermediate results.

The above can also explain the different trends in the CovType and SEP85L datasets. An informal examination of SEP85L reveals that it contains three dimensions that are highly correlated, namely Station-Id, Longtitude, and Latitude (since the same station is always located at the same Longtitude-Latitude coordinates). This generates some very dense areas in the corresponding cube deteriorating the efficiency of BUC-based methods. On the other hand, a similar informal examination of CovType has not revealed any obvious correlations that would generate dense clusters. Consequently, the corresponding cube is sparser, which clearly helps the performance of BUC-based methods.

Furthermore, it is interesting that the storage space required for TRS-BUC, BU-BST, and BU-BST+ increases until a maximum is reached and then decreases for larger Z values. To understand this trend, note that the cube size is affected by two parameters moving in opposite directions with Z: (p1) the number of redundant tuples and (p2) the average size of aggregated segments (sets of non-redundant tuples that aggregate). As Z increases, the original fact table becomes denser, which implies that the number of redundant tuples decreases while the average size of aggregated segments increases. Thus, as Z increases, p1 causes an increase of the cube size, since normal tuples are more expensive than redundant ones, whereas p2 causes a decrease of the cube size, since larger aggregated segments create fewer normal tuples. Initially, p1 is the dominant factor, so cube sizes increase overall, but later on p2 dominates, so cube sizes decrease overall.

Fact table size: We have created 8-dimensional, moderately-skewed ($Z = 0.8$) datasets with decreasing cardinalities varying the fact table size from 10^6 to 10^7 tuples. The former corresponds to a fact table of approximately 36 MB, while the latter to one that is ten times larger. In order to investigate the effect of partitioning, we have limited input buffers to approximately 100 MB. This means that only datasets with less than 3×10^6 tuples can fit in main memory. Figures 24 and 25 present the behavior of the seven algorithms under such conditions. Clearly, both execution time and storage space of all algorithms grow linearly with size. TRS-BUC is again the winner, having almost constant size, followed by BU-BST+, BU-BST and PRS-PC. Note that, once again, we have not been able to test BUC and PC to the limit, because their output has exceeded our storage capacity. However, their trend is obvious. Furthermore, note that the slope of all graphs in Fig. 24 increases between the second and the third measurement point. It is at this point where the entire fact table stops fitting in main memory, causing additional scans of the data. However, as expected, given the use of dynamic partitioning when the original data does not fit in memory, performance of all seven algorithms is not dramatically affected.

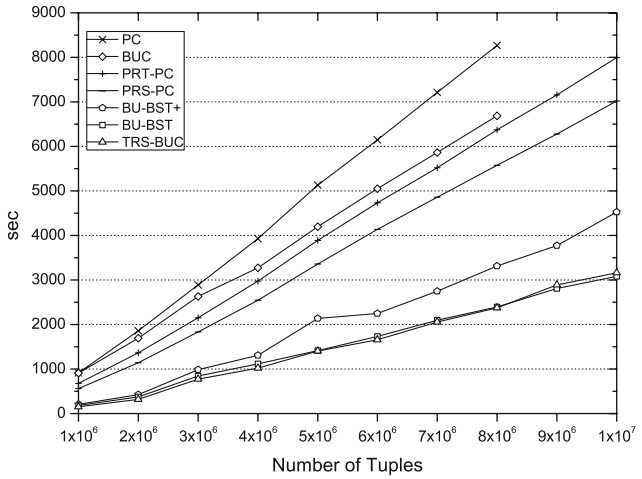


Fig. 24 Synthetic dataset ($D = 8, Z = 0.8$)—construction time

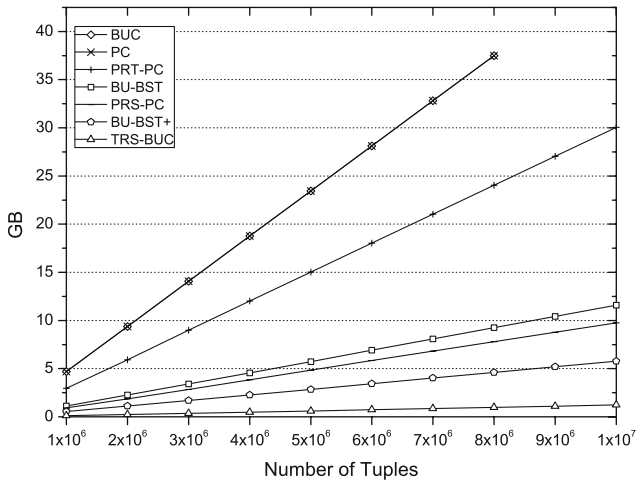


Fig. 25 Synthetic dataset ($D = 8, Z = 0.8$)—storage space

From all the above, we conclude that (a) skew is the critical parameter and (b) TRS-BUC exhibits the best or closely to the best performance in both construction time and storage space, making it the undisputed winner.

3 Query answering

The main reason for constructing a cube is to improve query response times. Constructing a condensed cube is not beneficial if its format cannot lead to query answering efficiency. Hence, query performance should be taken into account as well for the choice of a particular cubing algorithm (and its corresponding storage format). In this section, we study query performance over non-indexed cubes; the effect of indexing on the entire cube lifecycle (including query processing as well) is studied in Sect. 5.

Intuitively, one expects that a full BUC or PC cube behaves best in query answering, since all data is precomputed and no extra costs are necessary for accessing tuples in lattice nodes other than the ones queried. On the other hand, answering queries over condensed cubes generates additional costs for restoring non-materialized redundant tuples. In this section, we introduce some new algorithms for answering queries over condensed cubes and study the effect of such additional costs. We conclude that, although it stores a condensed cube, the format produced by TRS-BUC does not degrade query performance. On the contrary, a wide variety of query types runs faster over TRS-BUC cubes, because smaller tables are scanned and, more importantly, because a large number of access operations is redirected to a single relation (the fact table), which offers great opportunities for special optimizations.

The seven algorithms studied so far actually generate six different cube formats, since BUC and PC store identical, fully materialized cubes. Moreover, the BU-BST cube is monolithic and lacks any nice tuple clustering. Combined with its large size, this forces long sequential scans of the entire cube for answering any query. Our experiments have shown that response times over BU-BST are one to two orders of magnitude higher than those over the other formats. Finally, the cubes of PRT-PC and PRS-PC differ only in that the latter groups segments of consecutive references. Referencing entire redundant segments instead of tuples results in fewer disk seeks, as multiple consecutive tuples are fetched with one head movement. Hence, as initial experiments have confirmed, PRS-PC always outperforms PRT-PC. Based on all the above, in the rest of the section, we further consider in detail only the formats created by BUC, BU-BST+, TRS-BUC and PRS-PC.

3.1 Query model

In order to study query performance, we need to model the queries that we expect to deal with. A general form of a *group-by query* that can be answered using a data cube is presented in Fig. 26, where 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

```
SELECT S1, S2, ..., Sn, f(M)
FROM R
WHERE W1 op1 v1 AND ... AND Wk opk vk
GROUP BY S1, S2, ..., Sn
HAVING f(M) op v
```

Fig. 26 Group-by query model

the WHERE clause. Clearly, the most specialized node 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 $W \subseteq S$, since query time aggregations would affect all algorithms in the same way.

In a query, we may be interested in particular ranges of values for each dimension $S_i \in S$, affecting query selectivity. If n_i is the number of values in the range of interest of S_i, then $n_i \in [1, C_i]$, where C_i is the cardinality of S_i. Depending on the values of the n_i's, we identify three cases: (a) Setting $n_i = 1$ for each $S_i \in S$ produces a *point query*. In this case, $W = S$ and all operators are set to "=" in the WHERE clause. Point queries are the most selective. (b) Setting $n_i = C_i$ for each $S_i \in S$ produces a *node query*, meaning that the result is the entire node corresponding to the dimensions in S. In this case $W = \emptyset$. Node queries are the least selective. (c) Everything in-between is a *range query*. In this case $W \neq \emptyset$ and there is at least one i for which $n_i \in [2, C_i - 1]$. This is equivalent with setting at least one operator to ">" or "<" in the WHERE clause.

Existing cubing methods have been evaluated on point and range queries with $W = S$ [14,24]. To the best of our knowledge, our work is the first to study all three query types, including node and range queries with $W \subset S$. Note that node queries cannot be accelerated by indexing, since all node tuples must be returned. Moreover, low selectivity (large output) introduces great overheads to tree-like cubing formats (like Dwarf [24]), due to multiple traversals of the trees, which is not the case in our methods.

Furthermore, our query model also includes *iceberg queries*, which produce answers only for groups with large f aggregate values. We call *count iceberg queries* those that, in SQL syntax, contain a predicate of the form *HAVING count(M)* $> v$, for some $v > 0$. The TRS-BUC and BU-BST+ formats have a great advantage on them, because they require no redundancy to be restored (since, for totally-redundant tuples, count = 1). In fact, the performance of count iceberg queries over a (full) TRS-BUC or BU-BST+ cube is identical to that over a specialized *iceberg cube* produced by BUC [2] and is thus very efficient. This property of methods that handle totally-redundant tuples has not been studied before.

```
SELECT S1, S2, ..., Sn, f(M)
FROM R
CUBE BY S1, S2, ..., Sn
HAVING f(M) op v
```

Fig. 27 Subcube query model

Finally, our query model also includes *subcube queries* of the form presented in Fig. 27. Such queries can be answered by visiting 2^n cube nodes and are thus conceptually equivalent to 2^n node queries. They include no WHERE clause, as it would require run-time aggregations, which again would affect all algorithms in the same way.

3.2 Group-by and count-iceberg queries

In this subsection, for each cube format of concern, we present algorithms for answering general group-by queries (of the form described above) on some node S. Answering such queries over a BUC cube (Algorithm 3) is straightforward: Scan all tuples in node S and return those that satisfy the selection and HAVING criteria.

TRS-BUC adds an extra phase (Algorithm 4, lines 6-18): After scanning all normal tuples in S (lines 1-5), it follows all appropriate references that point to segments of consecutive tuples in the original fact table and fetches the corresponding redundant tuples. Those

Algorithm 3 BucGroupBy(node S)

```
1: for each tuple t in S do
2:   if t satisfies selection and HAVING conditions then
3:     Write(t);
4:   end if
5: end for
```

Algorithm 4 TRSBucGroupBy(node S)

```
1: for each normal tuple t in S do
2:   if t satisfies selection and HAVING conditions then
3:     Write(t);
4:   end if
5: end for
6: if not iceberg then
7:   for each descendant node N of S whose grouping attributes
     are a prefix of the grouping attributes of S (in the order used
     for cube construction) do
8:     for each pointer p in N's references do
9:       Fetch the redundant tuple set T pointed by p;
10:      for each tuple t in T do
11:        if t satisfies selection then
12:          t = Project(t, S);
13:          Write(t);
14:        end if
15:      end for
16:    end for
17:  end for
18: end if
```

that satisfy the selection criteria are projected on S and returned with the result. Note that the references followed are not only the ones stored in S, but also those stored in the descendants of S whose grouping attributes are a prefix of the grouping attributes of S. For example, if S = ABC (dimensions are sorted in the same order used for cube construction), then the nodes whose references must be followed are ABC, AB and A. This is necessary, since pointers to totally-redundant segments belong to an entire subcube, but are stored only once, according to the property of total redundancy described in Sect. 2.4. This property implies that the sets of row-id references accessed by the algorithm in every node are disjoint, which guarantees the correctness of Algorithm 4, since no duplicates are returned in the result set. As mentioned earlier, in the case of count iceberg queries this additional phase is not necessary (line 6), which has great impact on performance.

Furthermore, answering queries over a TRS-BUC cube can benefit greatly from low-cost caching. Since in TRS-BUC cubes, all references point to tuples in the original fact table, caching any portion of it is beneficial. Redundant tuples can then be retrieved from memory instead of being fetched from the disk. Although caching the entire cube seems infeasible, caching (some portion of) the original fact table is quite likely. The other cube formats (including BU-BST+) do not have a similar property and are therefore unable to make analogously effective use of any memory available: any cached portion of the cube is useful to them only for queries directly accessing that portion.

Moreover, note that the main steps for answering queries over a BU-BST+ cube are similar to the corresponding steps over a TRS-BUC cube, since the two formats differ only in the way they store redundant tuples. Hence, transforming Algorithm 4 to operate on BU-BST+ cubes simply involves removing the lines that access row-ids (namely, lines 8, 9 and 16) and modifying line 10 as follows: "**for each** tuple t in N **do**".

Finally, PRS-PC adds its own extra phase to Algorithm 3 (Algorithm 5, lines 6-14): After scanning all normal tuples in S (lines 1-5), it follows the references stored in S and fetches segments of redundant tuples from its ancestors. In this case, only references stored in S are processed, since, as mentioned, the knowledge that a tuple is partially redundant cannot be extended to other nodes.

3.3 Subcube queries

The algorithms for answering subcube queries (of the form presented in Sect. 3.1) are similar to the ones presented in Sect. 3.2 for group-by queries. The main difference is that instead of accessing node S only, all nodes of the subcube rooted at S must be accessed. The algorithms for processing subcube queries over BUC and PRS-PC cubes just call the corresponding group-by query algorithms iteratively (Algorithms 6 and 8, line 3). This is not the case for TRS-BUC (Algorithm 7), however, where an optimization is possible (lines 10-18): Whenever a redundant tuple is restored, it is not

Algorithm 5 PRSPcGroupBy(node S)

```
1: for each normal tuple t in S do
2:   if t satisfies selection and HAVING conditions then
3:     Write(t);
4:   end if
5: end for
6: for each pointer p in S's references do
7:   Fetch the redundant tuple set T pointed by p;
8:   for each tuple t in T do
9:     if t satisfies selection and HAVING conditions then
10:      t = Project(t, S);
11:      Write(t);
12:    end if
13:  end for
14: end for
```

Algorithm 6 BucCubeBy(node S)

```
1: PS_S = PowerSet(GroupingAttributes(S));
2: for each node N in PS_S do
3:   Call BucGroupBy(N);
4: end for
```

Algorithm 7 TRSBucCubeBy(node S)

```
1: PS_S = PowerSet(GroupingAttributes(S));
2: for each node N in PS_S do
3:   for each normal tuple t in node N do
4:     if t satisfies HAVING conditions then
5:       Write(t);
6:     end if
7:   end for
8:   if not iceberg then {Find Ancestors of N that belong to
     PS_S}
9:     A_SET = Anscestors(N, PS_S);
10:    for each pointer p in N's references do
11:      Fetch the redundant tuple set T pointed by p;
12:      for each tuple t in T do
13:        for each node A in A_SET do
14:          t' = Project(t, A);
15:          Write(t');
16:        end for
17:      end for
18:    end for
19:  end if
20: end for
```

Algorithm 8 PRSPcCubeBy(node S)

```
1: PS_S = PowerSet(GroupingAttributes(S));
2: for each node N in PS_S do
3:   Call PRSPcGroupBy(N);
4: end for
```

only returned for the currently processed node, but also for its ancestors that belong to the subcube rooted at S. Thus, redundant segments are accessed only once. Caching the original fact table is very beneficial in this case as well. The algorithm for answering subcube queries over a BU-BST+ cube is again similar with the algorithm for TRS-BUC. It simply involves removing the lines that access row-ids (namely, lines 11, 12, and 17) and modifying line 10 as follows: "**for each** redundant tuple t in node N **do**".

3.4 Experimental evaluation

In this subsection, we present the results of our experimental evaluation conducted to test query response times over BUC, BU-BST+, TRS-BUC, and PRS-PC cubes. We have run different types of queries over cubes of real and synthetic datasets. The trends indicated by the results of all datasets are similar, so in the subsequent presentation we concentrate on the real datasets. We have created random queries using the following parameters: *Dimension Probability* decides the percentage of dimensions that participate in the GROUP BY clause of a query. *Selectivity* affects the number of tuples that match the WHERE conditions. As mentioned earlier, for each grouping attribute S_i, we may be interested in a set of values whose cardinality ranges between 1 and C_i. The corresponding *selectivity factor* belongs to the interval $[1/C_i, 1]$. With respect to selectivity, we denote queries using a 3-dimensional vector $\langle x, y, z\rangle$, called *selectivity vector*, where $x, y, z \in [0, 1]$ and $x + y + z = 1$. Factor x denotes the percentage of grouping attributes for which we set an equality condition in the WHERE clause, and y the corresponding percentage of range conditions. Factor z denotes the percentage of grouping attributes that do not participate in the WHERE clause. Thus, selectivity vector $\langle 1, 0, 0\rangle$ represents a point query and $\langle 0, 0, 1\rangle$ a node query. Any other combination represents range queries. Note that as x increases, more equality conditions appear, producing more selective queries. On the contrary, larger z means lower selectivity and more tuples in the result. The effect of y depends on the selectivity factor. Larger selectivity factor means broader ranges and more selected tuples. In our experiments, for each range condition, we have set the selectivity factor to 0.1. Finally, we have varied parameter v of Figs. 26 and 27 between 1 and 100 to experiment with different types of iceberg queries. In all experiments, we have executed a warm-up phase consisting of 1% of the total number of queries to warm-up memory. This phase is excluded from the following results. All numbers presented are averages of 500 queries.

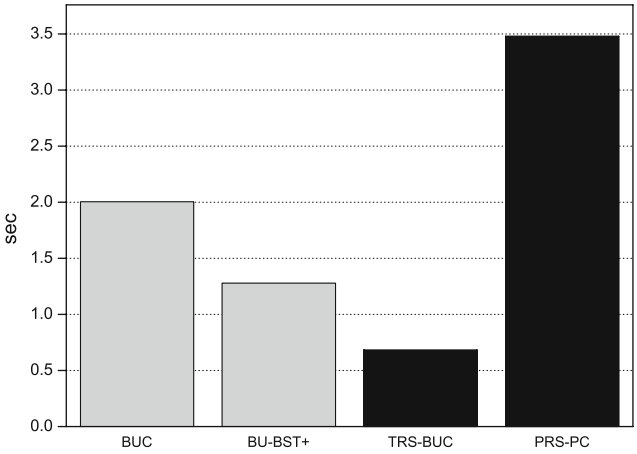


Fig. 28 CovType—average QRT

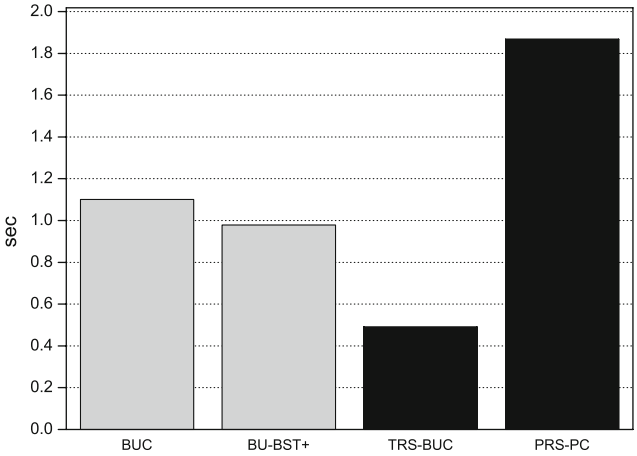


Fig. 29 SEP85L—average QRT

Average query: This is a group-by query with dimension probability 0.4 and selectivity vector $\langle 1/3, 1/3, 1/3\rangle$. In such queries, $W \subseteq S$ and selectivity is moderate. Figures 28 and 29 show the average query response time (QRT) of 500 queries run over CovType and SEP85L cubes, respectively. Clearly, these figures indicate that TRS-BUC outperforms all other formats, despite the fact that it restores redundant tuples referenced by row-ids on-line. The reasons for this are mainly its caching ability, due to the substitution of redundant tuples by row-ids that point to tuples in the fact table only, as well as the small size of the stored cube itself. As mentioned before, PRS-PC cannot take advantage of caching and the extra cost of restoring tuples is evident. Queries over SEP85L produce smaller answers and are thus faster, but otherwise offer no additional intuition on the cube formats. Hence, the rest of this section contains only the results for CovType.

Dimension probability: Varying the dimension probability from 0.2 to 0.6 has generated the chart of

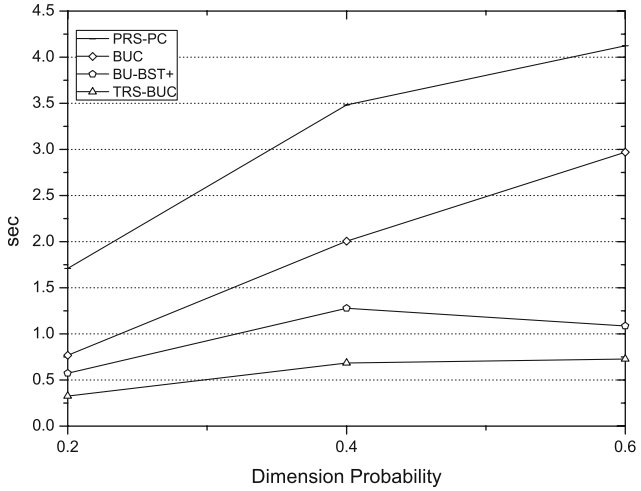


Fig. 30 CovType—effect of dimension probability on QRT

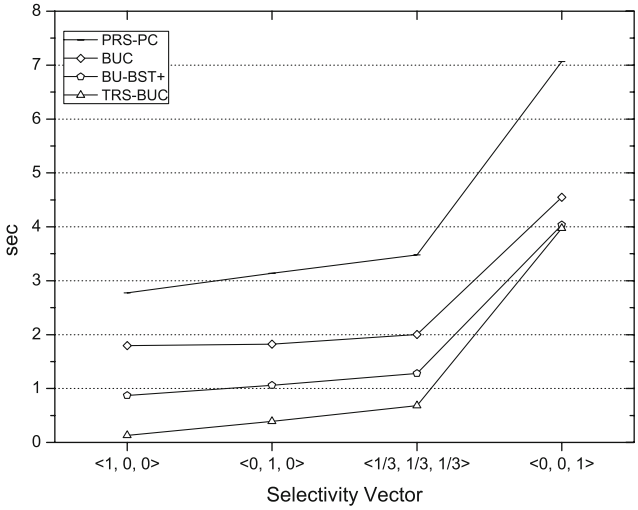


Fig. 31 CovType—effect of selectivity on QRT

Fig. 30, where once more TRS-BUC exhibits the best performance among all other formats. Interestingly, the slope of all graphs in it decreases with the dimension probability. This is attributed to the fact that increasing the number of grouping attributes generates two contradictory trends: On one hand, it implies access to larger nodes, since on average more dimensions appear in the GROUP BY clause of the corresponding queries. This produces more disk accesses, which increases the average query response time. On the other hand, the queries become more selective, since on average more dimensions appear in the WHERE clause as well. This produces smaller result sets, which decreases the average query response time. It seems that in lower values of the dimension probability the first trend dominates, but it fades in larger ones.

Selectivity: Figure 31 presents the effect of selectivity on query response time. Clearly, higher selectivity (to the left) gives an advantage to TRS-BUC, which is lost when more tuples are generated. In node queries, where selectivity is minimum, BU-BST+ and TRS-BUC are essentially equivalent, even if TRS-BUC still performs slightly better, since the latter is forced to restore and project more redundant tuples on the node queried, incurring additional costs.

Iceberg queries: As mentioned earlier, TRS-BUC and BU-BST+ cubes answer count iceberg queries very efficiently, since in this case, redundant tuples are not accessed at all. Figure 32 confirms their great advantage over the other cubes that do not have this property.

Caching: Figure 33 shows the effect of caching on the average group-by query over TRS-BUC cubes, which are the only ones that can take full advantage of a cache. Cache size is indicated as the portion of the fact table that fits in memory, ranging from 0 (no caching used)

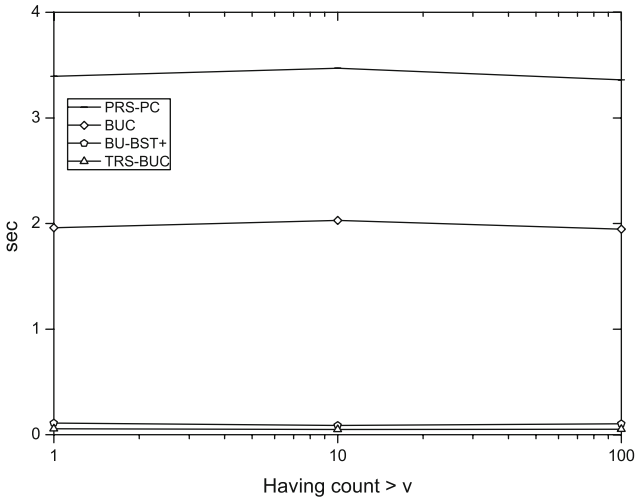


Fig. 32 CovType—iceberg QRT

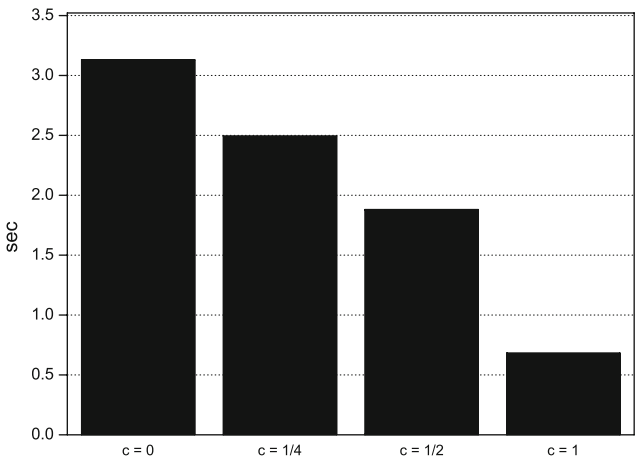


Fig. 33 CovType—effect of caching on TRS-BUC for QRT

to 1 (the entire fact table is cached). As expected, performance improves as cache size becomes larger for group-by queries.

Subcube queries: During our experimentation with subcube queries, we have generally found that they take much longer than group-by queries, since (a) they visit an exponential number of nodes, which generates greater reading costs, and (b) they do not have a WHERE clause (as explained in Sect. 3.1), which generates considerable output costs as well, since they are not selective. As a result, the response time of all algorithms is dominated by these factors and increases significantly with dimensionality, bringing the graphs of all algorithms very close to each other, with TRS-BUC behaving only slightly better. Given the lack of any particular differentiation among storage formats and algorithms, we omit the actual graphs.

4 Incremental maintenance

A cube contains aggregated data of a particular instantiation of a fact table. Hence, all updates to the fact table must be propagated to the cube as well in a batch mode, as is typical in data warehouses. Clearly, full reconstruction of the cube is prohibitively expensive, so the cube must be incrementally updated. In this section, we study the problem of incremental maintenance of the data cube and present novel algorithms for updating the four formats discussed in Sect. 3 (BUC, BU-BST+, TRS-BUC, and PRS-PC). As with query answering, initial experiments have shown that the BU-BST format is not amenable to efficient incremental maintenance, hence, it is excluded from the subsequent detailed analysis. Incremental maintenance has been studied in the past, in the context of BU-BST+, and has been approached through the use of indexing of the entire cube [4]. As shown in Sect. 5, however, such indexing introduces considerable overhead, rendering this technique relatively impractical.

Following common practice, we assume that the number of updated (delta) tuples is small compared to the size of the fact table. Furthermore, we assume that the aggregate functions involved in cube construction are *self-maintainable* [20]. 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, but 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. 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 refer elsewhere [20].

In this section, we present algorithms only for insertions; deletions and updates require some additional machinery but can be treated following similar approaches.

4.1 Algorithms

Having a fact table R, its corresponding data cube C(R) (in BUC, BU-BST+, TRS-BUC or PRS-PC format), and a set of delta tuples DT, there are (at least) three possible ways to obtain an updated cube. (a) *Merge method*: Construct the delta cube C(DT) in the format of C(R) and then merge it with C(R), producing C(R+DT). (b) *Direct method*: For each tuple t of each node N of C(R), find the tuples in DT that match with t on the dimension values (if any) and combine them into a new tuple. Aggregate the remaining tuples of DT according to N and add them to the result. (c) *Reconstruction method*: Take the union of R and DT and reconstruct C(R+DT) from scratch.

Since DT is small, we assume that we can construct a hash-table H with all its tuples in order to accelerate the Merge and Direct methods. The same holds for every node of C(DT) as well, if considered separately, since every such node contains at most as many tuples as DT. (In our initial experimentation, we have found that the alternative of building hash-tables on top of R and the nodes of C(R) performs much worse, due to the considerable amount of resources it requires.) Algorithms 9–12 customize the above general description for the Merge and Direct methods according to the needs of the corresponding cube formats.

Algorithm 9 describes the BUC and PRS-PC Merge methods. First, it constructs the delta cube C(DT) (line 1). Then it visits each node N of C(R) in a bottom–up fashion (line 2) and, for each tuple t in N (line 5), it searches for a matching tuple t' (a tuple with the same values in all dimension attributes) in the corresponding node N' of C(DT) (line 6). This search is accelerated through a hash-table H(N') (constructed in line 4). If such a matching tuple t' exists, the algorithm aggregates t and t' (line 8), replaces t with the resulting tuple t_N (line 9), and removes t' from the hash-table (line 10) to exclude it from the subsequent steps. Upon exiting the

Algorithm 9 BucOrPRSPcMerge(C(R), DT)

```
1: Create C(DT); {Construct the cube of deltas}
2: for each node N of C(R) in a bottom–up direction do
3:   Let N' be the corresponding node o C(DT);
4:   H(N') = Hash(N'); {Create hash-table}
5:   for each tuple t in N do {Both normal and redundant for
     PRS-PC}
6:     t' = FindMatchingTuple(t, H(N')); {Use hash-table H(N')
       to find if any tuple t' in N' matches with t}
7:     if t' ≠ NULL then
8:       t_N = Combine(t, t'); {Aggregate t and t'}
9:       Replace(t, t_N); {Write t_N as normal}
10:      Remove(t', H(N')); {Remove t' from hash-table}
11:    end if
12:  end for
13:  for each tuple t' remaining in H(N') do
14:    WriteNormal(t', N); {Redundancy may be lost...}
15:  end for
16: end for
```

Algorithm 10 BucOrPRSPcDirect(C(R), DT)

```
1: for each node N of C(R) in a bottom–up direction do
2:   H(DT) = Hash(DT); {Create hash-table }
3:   for each tuple t in N do {Both normal and redundant for
     PRS-PC}
4:     ST = FindMatchingTupleSet(t, H(DT)); {Use hash-table
       H(DT) to find all tuples in DT that match with t}
5:     if ST ≠ then
6:       t_N = Combine(t, Tuples(ST)); {Aggregate t and all tu-
         ples in ST}
7:       Replace(t, t_N); {Write t_N as normal}
8:       Remove(Tuples(ST), H(DT)); {Remove all tuples in ST
         from hash-table }
9:     end if
10:  end for
11:  for each tuple t remaining in H(DT) do
12:    ST = FindMatchingTupleSet(t, H(DT));
13:    t_N = Combine(Tuples(ST));
14:    WriteNormal(t_N, N); {Redundancy may be lost...}
15:    Remove(Tuples(ST), H(DT));
16:  end for
17: end for
```

loop that iterates over all tuples in N, it scans through all remaining tuples in H(N') (line 13) and writes them as normal tuples into node N (line 14). These tuples are the ones stored in node N' of C(DT) that have not been matched with any tuples of C(R). Note that the BUC and PRS-PC Merge methods are identical, because PRS-PC cannot take advantage of the knowledge that some delta tuple is partially redundant in C(DT) in line 14. In particular, recall that partially-redundant tuples are stored as pointers to ancestor nodes. Such a pointer, pointing to some tuple in a node of C(DT), cannot be mapped to a new pointer, pointing to some tuple in the corresponding node of C(R+DT), because the second node has not yet been constructed, since the lattice is traversed in a bottom–up fashion. Bottom–up

Algorithm 11 TRSBucMerge(C(R), DT)

```
1: Create C(DT); {Construct the cube of deltas}
2: for each node N of C(R) in a bottom–up direction do
3:   Let N' be the corresponding node o C(DT);
4:   H(N') = Hash(N'); //Create hash-table
5:   for each tuple t in N do
6:     t' = FindMatchingTuple(t, H(N')); {Use hash-table H(N')
       to find if any tuple t' in N' matches with t}
7:     if t' ≠ NULL then
8:       t_N = Combine(t, t'); {Aggregate t and t'}
9:       Replace(t, t_N); {Write t_N as normal}
10:      if t was redundant in C(R) then
11:        WriteRedundant(t, Parents(N));
12:      end if
13:      if t' was redundant in C(DT) then
14:        WriteRedundant(t', Parents(N'));
15:      end if
16:      Remove(t', H(N')); {Remove t' from hash-table }
17:    end if
18:  end for
19:  for each tuple t remaining in H(N') do
20:    if Normal(t) then
21:      WriteNormal(t, N);
22:    else
23:      WriteRedundant(t, N);
24:    end if
25:  end for
26: end for
```

traversal is unavoidable, otherwise, pointers would be invalidated and their restoration would be impossible.

The same problem exists for the PRS-PC Direct method as well, which is again identical to its BUC counterpart (Algorithm 10). Algorithm 10 is similar to algorithm 9, thus not explicitly elaborated. This inability of PRS-PC to take advantage of newly discovered redundancy deteriorates its format slowly, eventually transforming it into the BUC format.

On the contrary, TRS-BUC has no such problem. Pointers refer to totally-redundant segments in the fact table, which is always given. Hence, no new redundant segment is lost and the TRS-BUC format is preserved. Algorithm 11 describes the TRS-BUC Merge method and Algorithm 12 the TRS-BUC Direct method. The former is similar to Algorithm 9, but when two matching tuples t and t' are found (line 7) the aggregated tuple t_N is written in node N (line 9) and if t is redundant in C(R) (line 10), t is written as redundant in all the parents of N as well. The same holds if t' is redundant (line 13). Recall that totally-redundant tuples are only stored in the most specialized node they belong to, but are also redundant in all its ancestor nodes as well. So, since t is replaced by t_N in N, it must be written as redundant in the parents of N, otherwise cube tuples would be lost. If t remains redundant in that level, then it will remain unaffected when the algorithm visits the corresponding nodes. Otherwise, if it matches

Algorithm 12 TRSBucDirect(C(R), DT)

```
1: for each node N of C(R) in a bottom–up direction do
2:   DT'= NotYetRedundantTuples(DT,N); {Hash only DT's tuples that have not yet been identified as redundant in N's descendants}
3:   H(DT') = Hash(DT');
4:   for each tuple t in N do
5:     ST = FindMatchingTupleSet(t, H(DT')); {Use hash-table H(DT') to find all tuples in DT' that match with t}
6:     if ST ≠ then
7:       t_N = Combine(t, Tuples(ST)); {Aggregate t and all tuples in ST}
8:       Replace(t, t_N); {Write t_N as normal}
9:       if t was redundant in C(R) then
10:        WriteRedundant(t, Parents(N));
11:      end if
12:      Remove(Tuples(ST), H(DT')); {Remove all tuples in ST from hash-table}
13:    end if
14:  end for
15:  for each tuple t in H(DT') do
16:    ST = FindMatchingTupleSet(t, H(DT'));
17:    if Size(ST) == 1 then
18:      WriteRedundant(t, N);
19:    else
20:      t_N = Combine(Tuples(ST));
21:      WriteNormal(t_N, N);
22:    end if
23:    Remove(Tuples(ST), H(DT'));
24:  end for
25: end for
```

again something else from C(DT), it will be once more replaced by an aggregated tuple and written again as redundant in the "grandparents" of N. This guarantees that no redundancy is lost and preserves the TRS-BUC format unaffected after any number of updates. (Keeping updated redundant tuples for use in parent nodes has also been used elsewhere [4].)

Having the descriptions of algorithms 9–11 as a basis, Algorithm 12 is relatively straightforward to follow, hence, we elaborate on it no further.

Finally, note that, at the abstract level used for the presentation of algorithms above, the merge and direct methods for BU-BST+ are identical to those of TRS-BUC. Nevertheless, differences do exist at finer levels of detail, which affect performance. For example, line 5 in Algorithm 11 (line 4 in Algorithm 12 as well) states "**for each** tuple t in N **do**", indicating that the algorithm accesses all tuples that belong to N, i.e., both normal and redundant. If t is redundant, however, TRS-BUC fetches it from the fact table, while BU-BST+ reads it from the corresponding view. Similarly, function WriteRedundant called in lines 11, 14, and 23 of Algorithm 11 (lines 10, 18 of Algorithm 12 as well) behaves differently in TRS-BUC and in BU-BST+, since these two cubing methods store redundant tuples differently.

4.2 Experimental evaluation

Here, we present the most representative results of our experimental evaluation of the above incremental update algorithms. The graphs in Figs. 34 to 40 illustrate the performance of these algorithms over BUC, BU-BST+, TRS-BUC, and PRS-PC cubes of the SEP85L dataset.

Figure 34 presents the average time required to update a cube with delta size set to 1% of the fact table size. In general, the Merge and Direct methods have similar performance. They both fail on PRS-PC and take much longer than reconstruction because the reference targets in PRS-PC are not all in the original fact table but may be scattered across all cube nodes, forcing expensive disk seeks. Furthermore, as already explained, PRS-PC cannot exploit caching. On the contrary, incremental updates on BUC, BU-BST+, and TRS-BUC have considerable savings. BUC and BU-BST+ use no references, whereas TRS-BUC benefits from the concentration of all reference targets in the original fact table as well as from low-cost caching and achieves the best performance overall.

These conclusions are also confirmed by Figs. 35, 36, 37 and 38, which show the effect of the delta size on update performance. Incremental update methods benefit the most when the delta size is small, which is the situation in practice. As delta size becomes comparable to the fact table size, however, it is better to reconstruct the cube. Note that the Merge method scales better than the Direct both for BU-BST+ (Fig. 36) and for TRS-BUC (Fig. 37). The reason is that, given that the delta cube is small and compact, searching for tuple matches between pairs of nodes is faster than doing so between the nodes of C(R) and DT.

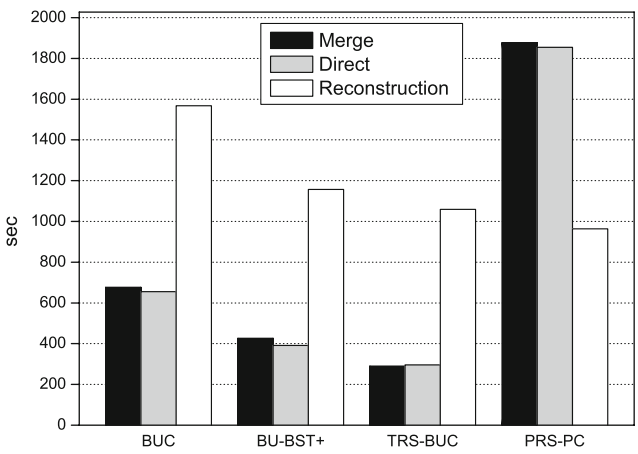


Fig. 34 Incremental update performance (delta size = 1%)

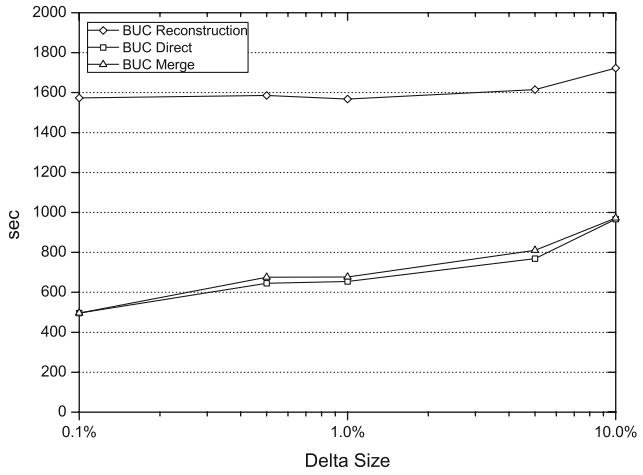


Fig. 35 BUC—effect of delta size on incremental update

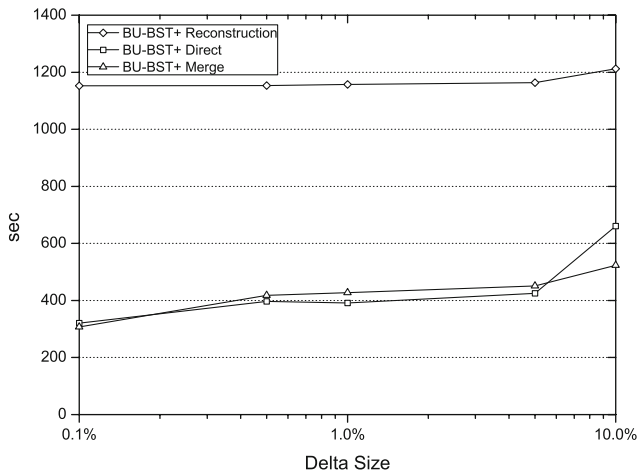


Fig. 36 BU-BST+—effect of delta size on incremental update

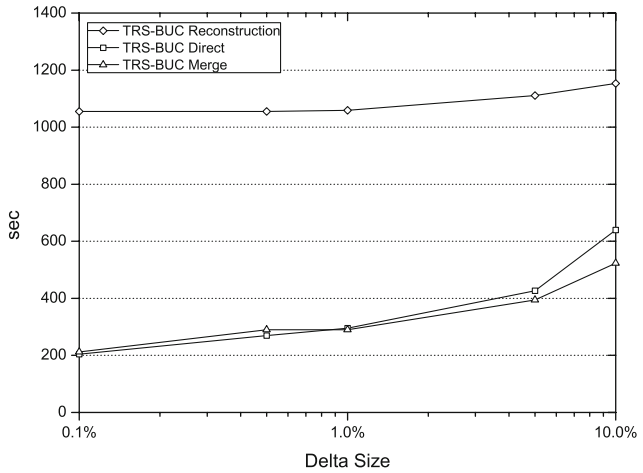


Fig. 37 TRS-BUC—effect of delta size on incremental update

Finally, Fig. 39 shows the TRS-BUC update performance when ten consecutive updates take place. In each update, the delta size is 1% of the fact table size. Clearly,

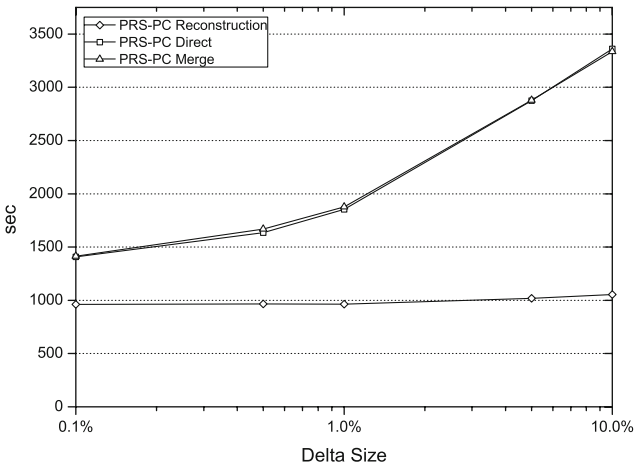


Fig. 38 PRS-PC—effect of delta size on incremental update

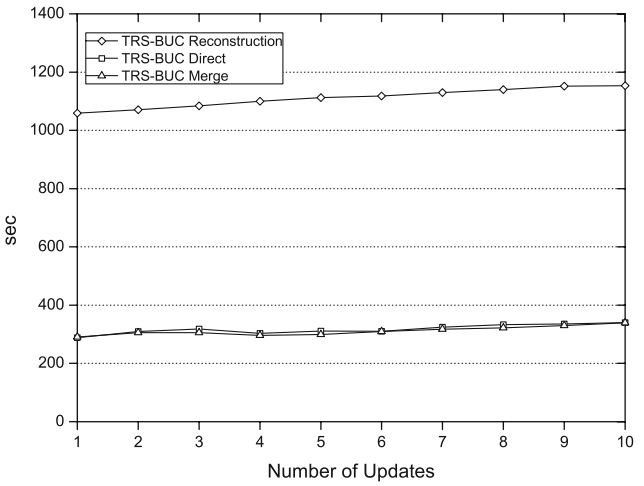


Fig. 39 Cumulative incremental updates (delta size = 1%)—time

incremental update methods scale well with the number of updates, remaining almost unaffected. The corresponding cube sizes after each update are shown in Fig. 40. This graph shows three identical lines, which confirm, as expected, that the TRS-BUC format is preserved, independent of the update approach.

5 Indexing

As mentioned earlier, cubes are constructed to improve the response times of OLAP queries. However, view materialization may not be enough, since cube nodes are usually large and sequential scans may be costly. In this section, we study indexing ROLAP cubes as a potential solution and conclude that TRS-BUC is the only cube format that can benefit from a simple indexing method consuming inexpensive resources.

To the best of our knowledge, the only effort so far to index a ROLAP cube (in particular BU-BST+) uses

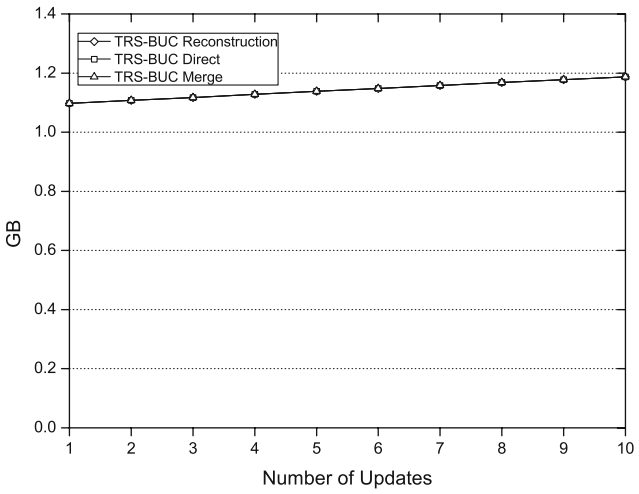


Fig. 40 Cumulative incremental updates (delta size = 1%) — storage space

a collection of Zkd B-Trees, called *CuboidTree* [4]. The *Z*-code of a tuple in a cube node is computed by interleaving the bits of the binary representation of the dimension values involved and is used as a key into a Zkd B-Tree index. This method does not capture naturally the order of tuples based on their dimension values (due to bit interleaving) making the range query algorithms rather complicated. Moreover, the size of a CuboidTree is comparable to the size of the indexed cube itself, imposing considerable overhead. Finally, the use of large *Z*-codes (up to 40 bytes for 10 dimensions) generates additional delays. Given the above and the fact that our focus has not been on indexing structures themselves, in this paper, we simply use a collection of B^+-Trees to index each cube node. B^+-Trees are supported by all relational servers and their effectiveness has been proven. Moreover, they preserve any sorting of tuples along each dimension, hence, benefiting both point and range queries.

5.1 Algorithms

A particular cube node S with **n** grouping attributes can be indexed by a collection of **n** B^+-Trees, one per attribute. In the leaf nodes of the *i*th B^+-Tree ($i \in [1, n]$), each attribute value is associated with a set of row-id references pointing to the tuples of S whose *i*th attribute has the corresponding value. Hence, the $\langle key, value \rangle$ pair inserted in the *i*th B^+-Tree for tuple t consists of the *i*th grouping attribute value and the row-id of t, respectively. The construction algorithm for such B^+-Tree indices is straightforward: For each cube node, scan all tuples and, for each grouping attribute, insert a $\langle key, value \rangle$ pair into the corresponding B^+-Tree.

Row-id reference sets can be physically stored in the leaf nodes of the corresponding B^+-Tree using either plain linked lists or bitmap vectors. Both solutions are ROLAP compatible, hence suitable for our methods. For instance, Fig. 41 shows node ABC (also shown in Figs. 3, 4 and 5) indexed by row-id linked lists (at the top) or bitmap vectors (at the bottom). In our study, we have experimented with both alternatives, as shown in the following subsection.

For the implementation of linked lists, we have used existing functionality in Berkeley DB [26], the widely accepted open-source implementation of B^+-Trees. For the implementation of bitmap vectors, we have used our own binary files. A naive solution would construct a separate file for each bitmap, i.e., for each distinct value in an attribute's domain (e.g., as indicated in the logical representation of bitmap vectors in Fig. 41). This would incur great overhead for file management during construction. To avoid this, in our implementation, we physically store all bitmap vectors associated with the values of a specific attribute of a node in the same file. This file consists of interleaved blocks from different bitmap vectors organized in linked lists, one list per

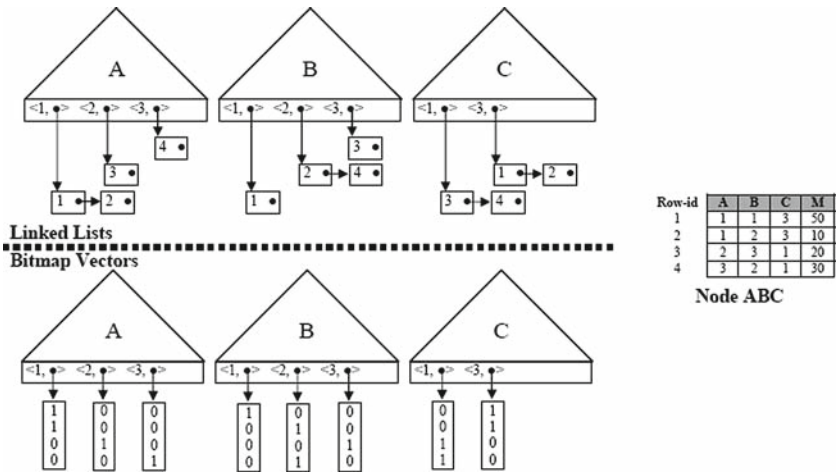


Fig. 41 Example of indexing node ABC using linked lists and bitmap vectors

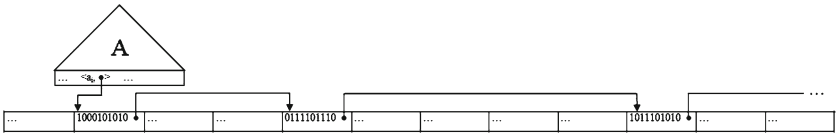


Fig. 42 Physical organization of a bitmap vector indexing value a_i

value. We use B$^+$-Trees to index the starting block of each list, indicating the beginning of the corresponding bitmap vector in the file, and references from one block to the next one that belongs to the same list. For example, the eight bitmap vectors that are shown as separate in the logical representation of Fig. 41, would actually be stored in three files, one per attribute. Figure 42 shows an example that gives more details on the actual representation of a bitmap vector associated with value a_i. In this figure, the B$^+$-Tree references the first block of the corresponding bitmap vector. Subsequent blocks of the same vector can be found by following references stored at the end of each block (illustrated as arrows in Fig. 42). These references allow accessing all blocks of the same bitmap vector without scanning the entire file. Our implementation further compresses the bitmaps using run-length encoding.

It is straightforward to prove that, in both alternatives (linked lists and bitmap vectors), a D-dimensional cube needs a total of $D \times 2^{D-1}$B$^+$-Trees for full indexing of all of its nodes. Without loss of generality, suppose that the levels of the cube lattice (Fig. 1) are numbered in a bottom–up fashion; then the ith level ($i \in [0, D]$) consists of c(D, i) nodes with i grouping attributes each, where c(D, i) is the number of combinations $\binom{D}{i} = \frac{D!}{i! \times (D-i)!}$. Hence, the total number NI of indices that need to be constructed is computed as follows:

$$
\begin{aligned}
NI &= \sum_{i=0}^{D} i \times c(D,i) = \sum_{i=0}^{D} i \times \frac{D!}{i! \times (D-i)!} \\
&= 0 + \sum_{i=1}^{D} i \times \frac{D!}{i! \times (D-i)!} = \sum_{i=1}^{D} \frac{D!}{(i-1)! \times (D-i)!} \\
&= \sum_{i=1}^{D} D \times \frac{(D-1)!}{(i-1)! \times [(D-1)-(i-1)]!} \\
&= D \times \sum_{i=1}^{D} c(D-1, i-1)
\end{aligned}
$$

Setting $i' = i - 1$ and $D' = D - 1$ the above formula gives:

$$
NI = D \times \sum_{i'=0}^{D'} c(D', i') = D \times 2^{D'} = D \times 2^{D-1} q.e.d.
$$

Algorithm 13 FetchNormalTuplesUsingIndex(node S, Criteria Cr)

```
1: bitmap = CreateBitmapVector(S);
2: SetAllBits(bitmap, 1);
3: for each grouping attribute ga of S do
4:   if exists selection condition on ga then
5:     Use BTree(S,ga) to find the rows in node S that satisfy
       Cr(ga);
6:     bitmap = bitmap ∩ rows;
7:   end if
8: end for
9: for each bit in bitmap do
10:  if bit==1 then
11:    Fetch the corresponding tuple t from node S;
12:    Write(t);
13:  end if
14: end for
```

TRS-BUC needs D additional indices for the original fact table as well, since all references replacing redundant tuples in cube nodes point to tuples in it. The indirection achieved through the use of references is so beneficial that for all practical purposes, as verified by our experiments, we do not actually need to index the TRS-BUC cube. In particular, we have already seen that the bottleneck in query answering over TRS-BUC cubes is in restoring redundant tuples from the original fact table. Hence, instead of constructing $D \times (1+2^{D-1})$ indices for the entire cube and fact table, just D B$^+$-Trees indexing the original fact table achieve essentially the same and often better performance. Furthermore, since the original fact table "lives" outside the cube, it is highly likely that it is already indexed for other purposes, implying that effective reuse of existing resources may just be enough.

Algorithm 13 provides a general sketch of a method that uses B$^+$-Tree indices for fetching *normal* tuples from a cube node S (in which pre-computed aggregated tuples have been materialized) that satisfy some selection criteria Cr. This algorithm is independent of the particular cube format (BUC, BU-BST, BU-BST+, TRS-BUC, PRS-PC, etc.). The main idea is to find the intersection of the row-id sets of tuples that satisfy the selection criteria of each dimension involved. Whether such row-id sets are stored as linked lists or bitmap vectors in the leaf nodes of the B$^+$-Trees, Algorithm 13 uses an in-memory bitmap vector (created in line 1) whose bits are initialized to 1 (line 2). This bitmap is iteratively (and very efficiently) intersected with the sets of rows

Algorithm 14 FetchRedundantTuplesUsingIndex(node S, node S', Criteria Cr)

1: bitmap = CreateBitmapVector(S);
2: SetBitsForRedundantTuples(bitmap, S, S');
3: Steps 3–14 are identical to the corresponding steps in Algorithm 13 with S' in place of S

that satisfy the selection criteria on the corresponding grouping attribute (lines 3–8). All (aggregated) tuples in S indicated by the final bitmap are those that satisfy all the selection criteria and are, hence, fetched and sent to output (lines 9–14).

Algorithm 14 is a variation of Algorithm 13 and uses B^+-Tree indices for fetching *redundant* tuples from a cube node S (which are references to normal tuples stored in node S', possibly the fact table) that satisfy some selection criteria Cr. The only modification is in line 2: Instead of initializing the entire bitmap vector with 1s, Algorithm 14 sets only the bits that correspond to redundant tuples in S that reference normal tuples in S', and the whole vector is then intersected with vectors expressing the selection criteria. Thus, among the redundant tuples of S, only those that satisfy the selection criteria are finally restored from S', saving unnecessary disk seeks and avoiding a major bottleneck in query answering.

Depending on the cube format used, a proper combination of calls to Algorithms 13 and 14 can be used for answering any query over an indexed cube. For example, in BUC a single call to Algorithm 13 is enough, since BUC stores only normal tuples, while in TRS-BUC we also need an additional call to Algorithm 14 for restoring redundant tuples referencing normal tuples that are actually stored in the original fact table.

Note that instead of B^+-Trees, we could have used more complex multidimensional structures (e.g., R-Trees). Such a choice, however, would not really affect performance, as the critical factor is actually the format of the indexed cube.

5.2 Experimental evaluation

In this subsection, we present the results of our experimental evaluation of the effect of indexing on cube usage. We have repeated all the experiments described in the previous sections (including cube construction, query answering, and incremental updating) adding collections of B^+-Trees, using either row-id linked lists or bitmap vectors. Below, we present our experiments for both cases (for the sake of brevity, we show the most representative graphs only). Note that "TRS-BUC Top" is the version of TRS-BUC that uses indexing only on the original fact table, while "TRS-BUC All" constructs a complete set of indices. Furthermore, the performance of BU-BST with indexing is one to two orders of magnitude worse than that of BU-BST+; hence, here we show results only for the latter.

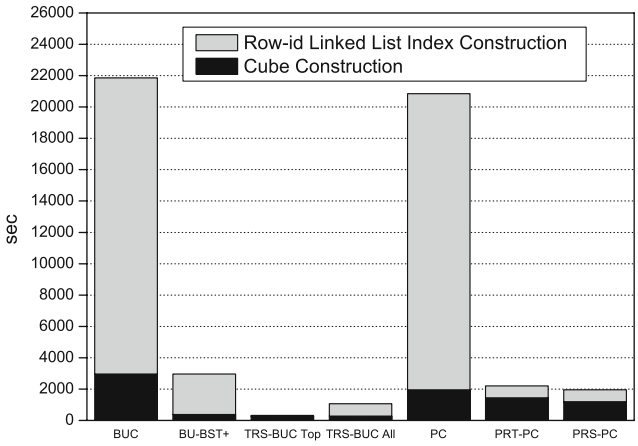


Fig. 43 CovType Cube + linked list index construction time

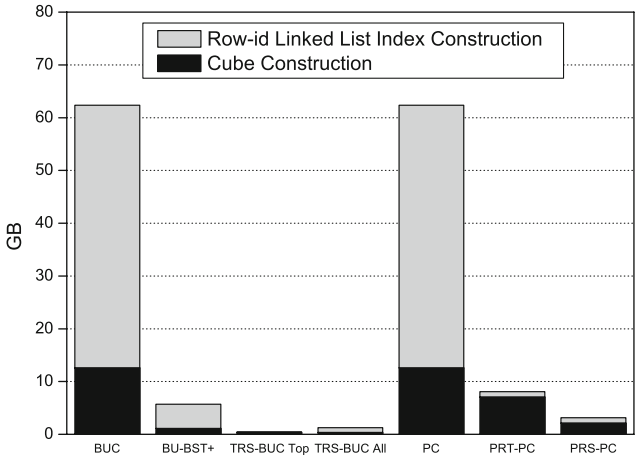


Fig. 44 CovType Cube + linked list index storage space

Cube construction: Figures 43 to 48 demonstrate the effect of indexing on cube construction. The first pair of graphs corresponds to the use of row-id linked lists, while the second pair to the use of bitmap vectors, respectively. Each bar in these graphs consists of two parts. The black part in the bottom represents pure cube construction, as presented in Figs. 15 and 16. The gray part illustrates the additional time and space resources spent for indexing. Interestingly, the use of bitmaps (Figs. 45 and 46) offers great savings compared to the use of row-id linked lists (Figs. 43 and 44), due to the compression capabilities that they offer, as reported elsewhere as well [10]. This is better illustrated in Figs. 47 and 48, which zoom in on the total index construction time and storage space for the most efficient methods.

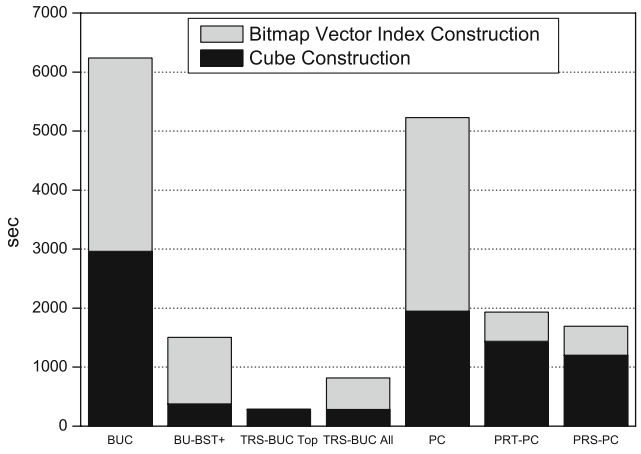


Fig. 45 CovType Cube + bitmap index construction time

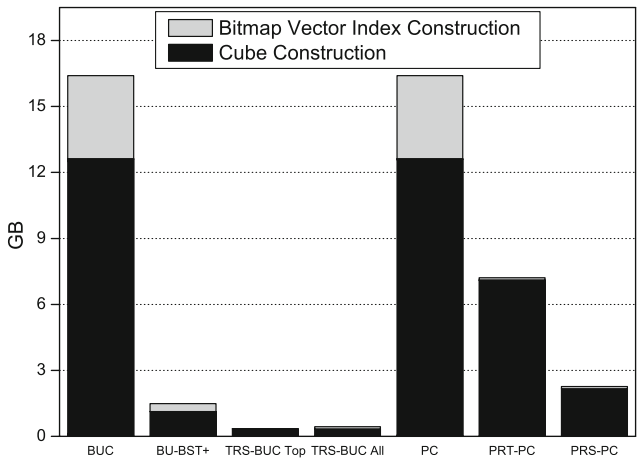


Fig. 46 CovType Cube + bitmap index storage space

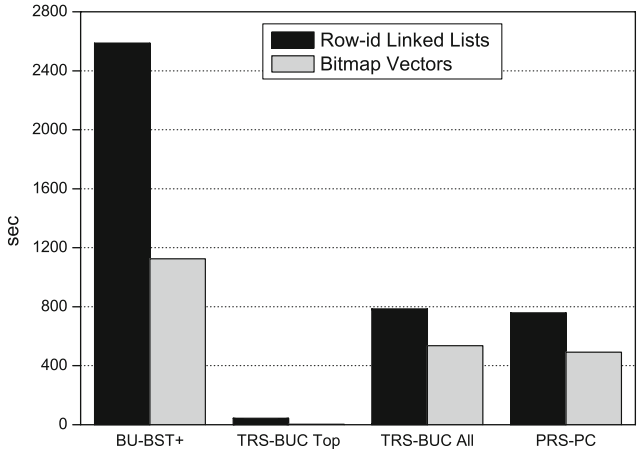


Fig. 47 CovType Zoom in on index construction time

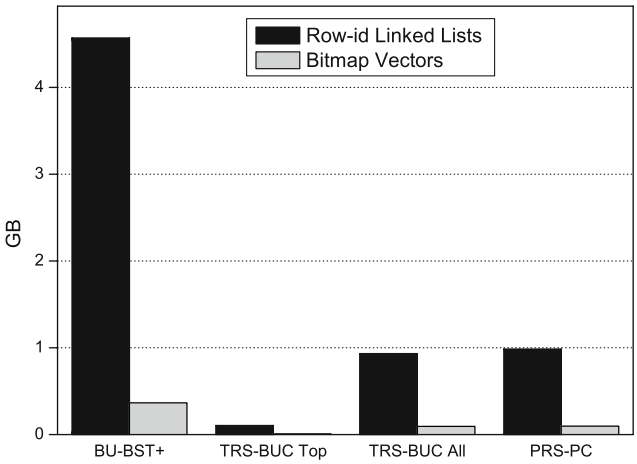


Fig. 48 CovType Zoom in on index storage space

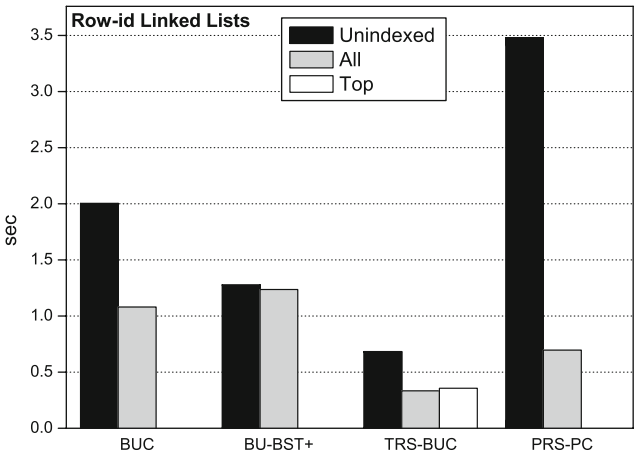


Fig. 49 CovType Cube + linked list index average QRT

It is clear that only the cube formats that replace redundant tuples with references (like TRS-BUC, PRT-PC, and PRS-PC) can be practically fully indexed without imposing any unreasonable overhead to cube construction and storage. Especially TRS-BUC Top seems to prevail by consuming very limited additional resources. In BU-BST+ the overhead of index construction and storage is comparable to the construction and storage of the cube itself, hence non-trivial, whereas in the other formats (BUC and PC) the amounts of additional time and space to construct and store the corresponding indices is prohibitively large. Especially in the construction of bitmap vectors (in Fig. 47 and Fig. 48), TRS-BUC Top is approximately 342 times faster and produces a result that is approximately 43 times smaller than that of BU-BST+, which is the best existing method among the ones compared here. The improvement is clearly impressive.

Query answering: Concerning query answering, Figs. 49 and 50 show the results for the average query workload. In these graphs, the black bars correspond to unindexed cubes, as presented in Fig. 28. Interestingly, these figures demonstrate that not only TRS-BUC Top consumes fewer resources, but it also exhibits very fast query response times, which are essentially the same

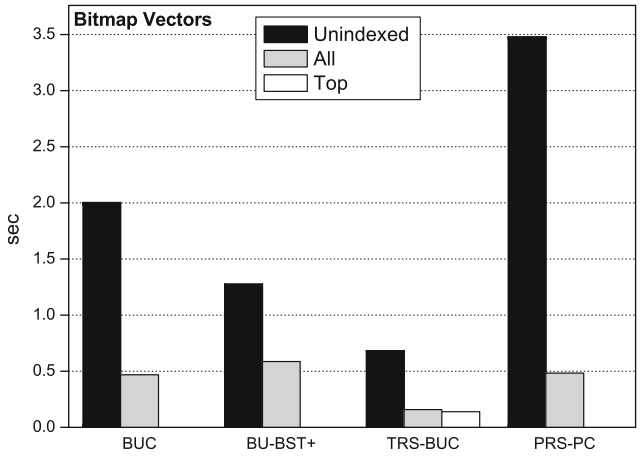


Fig. 50 CovType Cube + bitmap index average QRT

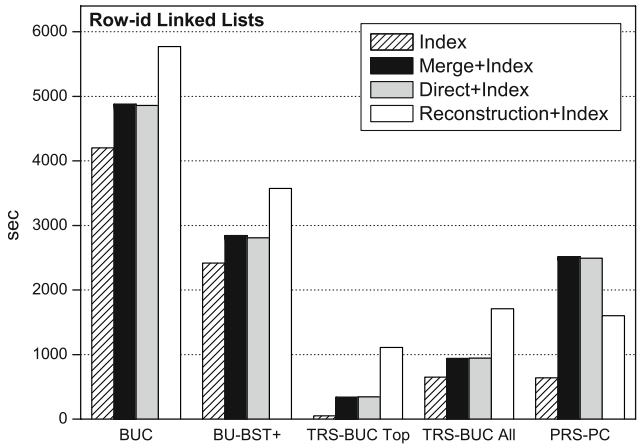


Fig. 51 SEP85L Cube + linked list index incremental update performance (delta size = 1%)

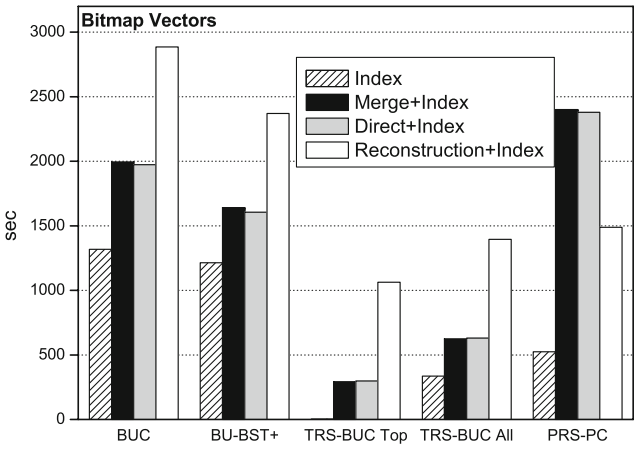


Fig. 52 SEP85L Cube + bitmap index incremental update performance (delta size = 1%)

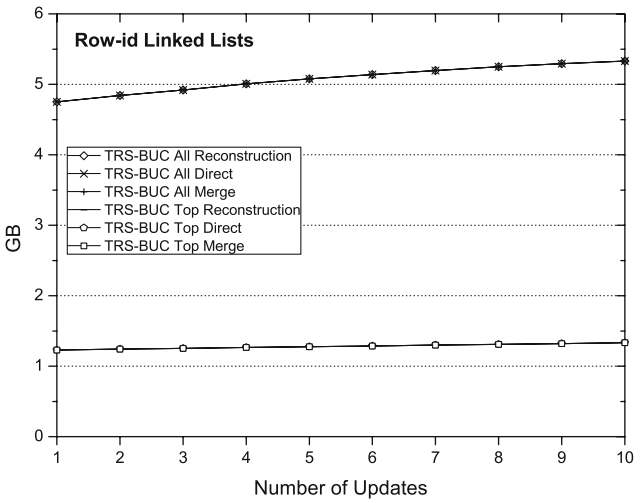


Fig. 53 SEP85L Cube + linked list index cumulative incremental update storage space (delta size = 1%)

and sometimes better than the query response times of TRS-BUC ALL. Especially when combined with bitmap vectors, TRS-BUC Top provides query response times that are more than a factor of 4 faster than the unindexed case (Fig. 50), a considerable improvement. The query response times of all the other formats are always worse. Clearly, these properties make TRS-BUC Top the best choice. Experiments with different parameter values have exhibited similar trends as well; hence, the corresponding graphs are omitted.

Incremental maintenance: Finally, Figs. 51 to 54 demonstrate the results of our experiments with incremental maintenance of indexed cubes. In Figs. 51 and 52, the dashed bars correspond to pure index maintenance costs, while the other bars to the total cost of updating both the indices and the cube itself. Figures 53 and 54 extend Fig. 40 and show two groups of three identical lines: the bottom line corresponds to the three versions of TRS-BUC Top, whereas the top line to the three versions of TRS-BUC All. Clearly, TRS-BUC Top imposes minimal additional costs for incremental updates as well, especially when bitmap vectors are used.

Overall, the results indicate that TRS-BUC is essentially the only method that can be practically indexed, using limited amounts of additional resources, and can take advantage of such indices for all phases of the data cube lifecycle.

6 Related work

The problem of efficient data cube implementation has attracted much attention in the database community. After Gray et al. proposed the cube-by operator [6], a plethora of papers has been published in this area. Most cube construction and storage methods comply with the

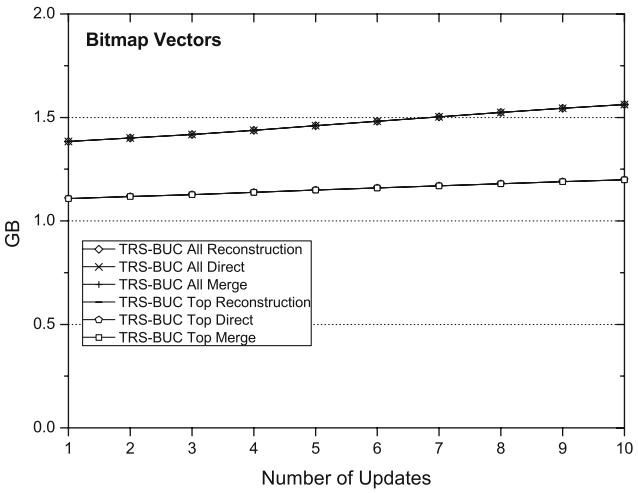


Fig. 54 SEP85L Cube + bitmap index cumulative incremental update storage space (delta size = 1%)

	ROLAP	Non-ROLAP Construction + ROLAP Storage	Non-ROLAP
Full/Iceberg Cubes	BUC PC	H-Cubing Star-Cubing MM-Cubing	
Reduced Cubes	TRS-BUC PRT -PC PRS-PC BU-BST BU-BST+ QC-DFS	Range-Cubing	Dwarf QC-Tree

Fig. 55 Classification of advanced cubing methods

ROLAP architecture [1,2,4,6,9,12,13,21,23,28] and are surveyed in detail elsewhere [19], and fewer with the MOLAP architecture [11,30]. The problem of efficient storage has also been addressed through other, orthogonal approaches, whose study exceeds the purpose of this paper. Examples include approximation methods [27] as well as partial cubing methods, which produce only a thin layer of the cube, keeping additional indices that help cube node construction at query time [17].

As mentioned earlier, among the ROLAP techniques that can handle large fact tables and store the entire cube on disk, the "champions" are BUC [2] and PC [21]. Our work has started from them and has resulted in PRT-PC and PRS-PC, which incorporate redundancy reduction into PC, and TRS-BUC, which does the same into BUC. The methods that are closer to our work are BU-BST/BU-BST+ [28,4] and QC-DFS [13], which have extended BUC in the spirit that TRS-BUC does but with several crucial differences, as described in Sect. 2.5. In this paper, we have shown that TRS-BUC tackles the data cube problem in a comprehensive fashion, providing efficient algorithms for all phases of the cube lifecycle. To the contrary, the original algorithms behave well only in (off-line) computation and storage, whereas earlier redundancy-reducing BUC extensions, i.e., BU-BST/BU-BST+ and QC-DFS, produce relational cube representations (BU-BST/BU-BST+ cubes, and QC-Tables, respectively) that have inferior performance next to TRS-BUC cubes. Note that query processing and incremental maintenance has been studied earlier for BU-BST/BU-BST+ cubes as well [4]. The particular solutions proposed, however, assume that the entire cube is indexed, which as shown in this paper, is not very practical.

The above algorithms can be classified as purely ROLAP, as it is not only their storage of the cube that is relational but their processing of the fact table during construction as well, consisting mostly of standard relational operations (e.g., sorting and projecting) and using mostly in-memory relations (see Figs. 6 and 9). This makes these techniques easy to integrate in any existing relational server. In contrast to them, there have been several proposals recently that continue to store the cube in relational views but construct it with the help of specialized in-memory data structures that represent the fact table compactly and are traversed in efficient, non-relational ways, e.g., H-Cubing [8], Star-Cubing [29], MM-Cubing [22], and Range-Cubing [5]. They have all been presented primarily under the assumption that the complex data structures they use fit in main memory. Ideas on how to transform some of these techniques into dealing with large datasets are mainly based on partitioning the original fact table and/or on constructing the corresponding data structures externally and paging them in and out of disk during their traversal [8,22,29]. The effectiveness of these ideas remains open, however, as to the best of our knowledge, they have not been experimentally evaluated. On the contrary, our experiments have indicated (Fig. 24) that TRS-BUC operates efficiently on memory-fitting segments of the fact table generated by the partitioning scheme of the original BUC method [2], without suffering a great performance loss.

Furthermore, with the exception of Range-Cubing [5], all the other methods using special in-memory structures (namely, H-Cubing [8], Star-Cubing [29], and MM-Cubing [22]) construct complete or iceberg cubes, with no redundancy reduction. They are capable of performing Apriori pruning [2], however, as part of their iceberg-cube construction functionality. Hence, in principle, they could be extended to identify and remove totally-redundant tuples (count = 1) and produce TRS-BUC cubes. Although using a different construction algorithm, this approach could take advantage of all the benefits that the TRS-BUC cube representation has for the remaining phases of the cube lifecycle: efficient querying, indexing, and incremental updating.

This approach would probably outperform TRS-BUC on small and dense datasets, as these algorithms put particular emphasis on dense areas of the fact table, but would probably lose on large or sparse datasets given the redundancy-reduction characteristics of TRS-BUC. Preliminary experimentation indeed concurs with the above intuition. On the other hand, Range-Cubing [5] does remove redundancy, i.e., redundancy that arises from cube tuples that are produced by the same set of fact-table tuples and belong to any path between two specified nodes in the cube lattice, called *range*. Essentially, a Range-Cube is identical to a QC-Table that only stores atomic values in the Lower-Bounds column, i.e., lower bounds of equivalent classes that do not include disjunctions ("$\vee$") in their encapsulating logical expression (Sect. 2.5, Fig. 11). This means that Range-Cubes are expected to be larger than QC-Tables and also suffering from some of the same deficiencies (e.g., the need for additional meta-data associated with every tuple and the monolithic format). Given that QC-Tables are outperformed by TRS-BUC, all indications are that Range-Cubes will be outperformed as well.

Moving even further away from ROLAP, there is a small number of additional techniques that depend on specialized hierarchical data structures not only for in-memory construction of the cube but for the entire cube lifecycle, including storage on disk, querying, and updating. The main representatives in this category are Dwarf [24] and QC-Tree [14]. Dwarf [24] seems to be the strongest algorithm overall, since it is the only one that guarantees a polynomial time and space complexity with respect to dimensionality [25]. It is based on a highly compressed data structure that eliminates *prefix* and *suffix redundancies* efficiently. Prefix redundancy occurs when two or more tuples in the cube share the same prefix. For example, in the cube of Fig. 3, there are six tuples that have the same value $A = 1$ in their first dimension. These are tuples $\langle 1,1,3,50\rangle$ and $\langle 1,2,3,10\rangle$ in node ABC, tuples $\langle 1,1,50\rangle$ and $\langle 1,2,10\rangle$ in AB, tuple $\langle 1,3,60\rangle$ in AC, and tuple $\langle 1,60\rangle$ in A. This group exhibits prefix redundancy where the prefix size is 1, i.e., it involves just the first dimension. There are also other examples where the length of the redundant prefix is greater than 1. Dwarf recognizes this kind of redundancy and stores every unique prefix just once. This does not affect the total number of cube tuples but reduces the space required to store each tuple. Prefix redundancy is not captured by TRS-BUC or any of the other methods investigated in this paper. Fortunately, it is relevant only to the storage of normal (non-redundant) tuples, which are few, as demonstrated by our experimental study, making the penalty of not dealing with it relatively negligible. On the other hand, suffix redundancy is in some sense complementary to prefix redundancy and occurs when two or more cube tuples share the same suffix, i.e., the same dimension and aggregate values on the right. For example, in the cube of Fig. 3, tuples $\langle 1,1,3,50\rangle$ in node ABC and $\langle 1,3,50\rangle$ in BC exhibit suffix redundancy, as they share the same values in their right-most dimensions (B and C) and also in their aggregated measure. Suffix redundancy occurs primarily when multiple cube tuples are generated by the same set of tuples in the fact table, which of course contribute the exact same aggregate value every time. Hence, it is similar to the notion of partial redundancy of segments, which is dealt with by several algorithms examined in this paper and has been shown to benefit performance substantially.

An advantage of Dwarf is that it does not only store a data cube compactly, but also serves as an index that can accelerate point and selective range queries. TRS-BUC needs additional, even if inexpensive, external indices to attempt to compete with Dwarf's performance on such queries. Furthermore, as also mentioned above, Dwarf has polynomial time and space complexity [25], whereas TRS-BUC has no proof of polynomial behavior, although our experimental evaluation has shown that it is not cursed by dimensionality. On the other hand, TRS-BUC promises better results in node and subcube queries of low selectivity, since it clusters tuples in each node and, unlike Dwarf, it requires no multiple tree traversals. Furthermore, TRS-BUC preserves its format after incremental updates, so full reconstruction is never necessary, whereas frequent incremental updates would slowly deteriorate Dwarf's clustering. Hence, both TRS-BUC and Dwarf appear to be viable solutions to the cubing problem, each with its own advantages and disadvantages, which should be studied in detail in future work.

The other major technique that employs a specialized data structure for external storage of the cube is QC-Tree [14]. As mentioned in Sect. 2.5, QC-Tree implements Quotient Cubes in non-relational fashion to overcome the limitations of QC-DFS [13], which stores the cube in QC-Tables. QC-Tree has similar properties with Dwarf, capturing prefix redundancy and grouping together classes of tuples generated by the aggregation of the same set of fact-table tuples. On the other hand, it shares some of Dwarf's weaknesses compared to TRS-BUC. Again, the main distinction in favor of TRS-BUC compared to both Dwarf and QC-Tree is that it is ROLAP compatible, which makes it easier to implement over existing relational servers, taking advantage of all the nice properties of a mature technology. A comprehensive comparison of all three techniques is necessary to draw any precise conclusions on the exact trade-offs among them. This is beyond the scope of this paper,

which focuses on pure ROLAP approaches, but is part of our plans for future work.

The above overview of existing techniques and their characteristics is summarized visually in the table illustrated in Fig. 55. The table classifies (a) BUC [2] and PC [21], the "champions" of ROLAP techniques that store the entire cube on disk, (b) the most widely accepted methods that have been proposed subsequently, and (c) the techniques introduced in this paper. This table also helps explain the intuition behind particular choices we have made in designing our methods and sets a clear boundary on the scope of this paper: pure ROLAP algorithms that are capable of compressing the final cube. Comparing across the columns of the table, primarily the ROLAP and non-ROLAP columns, is the subject of a separate study.

Finally, in this paper, we have focused on all the phases of the cube lifecycle but have dealt with only flat datasets, ignoring the orthogonal issue of dealing with datasets whose dimensions are organized in hierarchies. In an independent thread of our research [18], we have essentially studied only construction and storage of cubes and have proposed the CURE algorithm, which deals effectively with the challenges introduced by the nature of hierarchies.

7 Conclusions and future work

In this paper, we have incorporated redundancy reduction into the best existing pure ROLAP methods for cube implementation and have proposed a suite of novel algorithms that deal with all aspects of cube usage, including efficient construction, storage, query answering, incremental updating, indexing, and caching. To the best of our knowledge, this is essentially the first such comprehensive approach to the problem in the ROLAP context, treating all the above aspects in an independent fashion. We have created comprehensive testing environments, broader than the ones used in the past, and have experimented with both synthetic and real-world datasets. Our extensive evaluation has shown that TRS-BUC dominates or nearly dominates its competitors in all aspects of the cube problem: It provides fast computation of a fully materialized cube in compressed form, is incrementally updateable, and exhibits quick query response times that can be accelerated by inexpensive indexing and caching. TRS-BUC appears to be the first ROLAP cubing method that packages all these nice properties into a single comprehensive solution giving strong indication of the power of ROLAP.

Our future work directions include extension of TRS-BUC to store groups of nodes/views in the same file. This presents an interesting implementation problem, since we have seen that file management generates considerable overhead in datasets with high dimensionality. We are also planning to investigate whether or not our intuition that TRS-BUC storage requirements grow polynomially with dimensionality, as indicated by Fig. 20, can be proven analytically. Finally, we intend to compare TRS-BUC with the most prominent non-ROLAP methods (e.g., Dwarf and QC-Tree) in order to study the comparative advantages and disadvantages of the corresponding general philosophies.

Acknowledgments We would like to thank Cuiping Li, Gao Cong, Anthony K. H. Tung, and Shan Wang for supplying their implementation of QC-DFS and allowing us to use it in our experiments [15,16]. Furthermore, we would like to thank Jianlin Feng for explaining to us some details of BU-BST [4,28], especially regarding the use of separate views to store each node of the cube in a later version of the algorithm [4]. Finally, we would also like to thank the anonymous referees for their valuable comments.

References

1. 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)
2. 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)
3. Blackard, J.A.: The forest covertype dataset. ftp://ftp. ics.uci. edu/pub/machine-learning-databases/covtype
4. 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)
5. 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)
6. 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)
7. Hahn, C., Warren, S., London, J.: Edited synoptic cloud reports from ships and land stations over the globe. http://cdiac. esd.ornl.gov/cdiac/ndps/ndp026b.html
8. 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)
9. 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)
10. Johnson, T.: Performance measurements of compressed bitmap indices. In: Proceedings of Very Large Data Bases (VLDB), pp. 278–289 (1999)

11. 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)
12. 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)
13. 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)
14. 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)
15. 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)
16. 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)
17. Li, X., Han, J., Gonzalez, H.: High-dimensional olap: a minimal cubing approach. In: Proceedings of Very Large Data Bases (VLDB), pp. 528–539 (2004)
18. Morfonios, K., Ioannidis, Y.: Cure for cubes: Cubing using a rolap engine. In: Proceedings of Very Large Data Bases (VLDB) (2006)
19. Morfonios, K., Konakas, S., Ioannidis, Y., Kotsis, N.: Rolap implementations of the data cube (submitted)
20. 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)
21. Ross, K.A., Srivastava, D.: Fast computation of sparse datacubes. In: Proceedings of Very Large Data Bases (VLDB), pp. 116–125 (1997)
22. 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)
23. 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)
24. 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)
25. Sismanis, Y., Roussopoulos, N.: The complexity of fully materialized coalesced cubes. In: Proceedings of Very Large Data Bases (VLDB), pp. 540–551 (2004)
26. Sleepycat Software: The berkeley database (berkeley db). http://www.sleepycat.com
27. Vitter, J.S., Wang, M., Iyer, B.R.: Data cube approximation and histograms via wavelets. In: Proceedings of International Conference on Information and Knowledge Management (CIKM), pp. 96–104 (1998)
28. Wang, W., Lu, H., Feng, J., 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)
29. 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)
30. 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)
31. Zipf, G.K.: Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology. Addison-Wesley, Reading (1949)