PMC: Select Materialized Cells in Data Cubes

Hongsong Li[1], Houkuan Huang[1], and Shijin Liu[2]

[1] School of Computer and Information Technology,
Beijing Jiaotong University, Beijing, China, 100044
mlhs@163.com, hkhuang@center.njtu.edu.cn
[2] Ticket Center, Passenger Traffic Department,
Urumchi Railroad Bureau, Urumchi, China, 830011
LiukingLiuking@21cn.com

Abstract. QC-Trees is one of the most storage-efficient structures for data cubes in a MOLAP system. Although QC-Trees can achieve a high compression ratio, it is still a fully materialized data cube. In this paper, we present an improved structure PMC, which allow us to partially materialize cells in a QC-Trees. There is a sharp contrast between our partially materialization algorithm and other extensively studied materialized view selection algorithms. If a view is selected in a traditional algorithm, then all cells in this selected view are to be materialized. Our algorithm, however, selects and materializes data by cells. Experiments results show that PMC can further reduce storage space occupied by the data cube, and can shorten the time for update the cube. Along with further reduced space and update cost, our algorithm can ensure a stable query performance.

1 Introduction

Using pre-aggregated data in data cube is an efficient method to improve query performances of data cubes. Usually, to attain better storage and update performance, only a part of views are materially stored in the database. Apparently, there exists performance difference between materialized view and non-materialized view. Moreover, to select materialized views, prior knowledge about the query model of users should be attained. This kind of knowledge, however, are usually not available.

QC-Trees is one of the most compress efficient structures of data cubes. This structure has significant merits. The more sparse the data cube is, the more storage space can be saved. Since typically high-dimension data cubes are sparse, a high compression ratio can be easily achieved.

In this paper, we present PMC and corresponding algorithms to select only a part of cells in QC-Tree to be materialized so that better space and update efficiencies are achieved. In contrast with traditional view materialization algorithms, cells (not views) are selected to be materialized. In this way, a low bound of query can be ensured by our structure. This low bound is regulated by a parameter of selection algorithm, which can be used for controlling the tradeoff between storage, update and query performances.

This paper includes following contributions:

Firstly, we present a structure PMC (Partially Materialized Cube), which can be seen as an extension or a generalization of QC-Tree. To our knowledge, our work is the first attempt to partially materialize data cube at the cell level.

Secondly, corresponding algorithms are presented, including selection of materialized cells, queries and update process for PMC. Besides the costs of storage and maintenance are reduced, the query performance can be stably ensured by a parameter of the selecting algorithm.

The paper is organized as follows: related work are reviewed in Section 2. In section 3, we present PMC structure and corresponding algorithms including select, update and query algorithms. Then in Section 4 we conduct experiments to evaluate PMC algorithm, while conclusions are presented in Section 5.

2 Related Work

Online analytical processing (OLAP) is an essential data analysis service and can provide critical insights into huge amount of application data. Data Cube, which was proposed by J. Gray[2], has been extensively applied to the implementation of OLAP. Data in a data cube can be stored in special data structures(MOLAP) or in tables of relational databases(ROLAP).

Using pre-aggregated data in data cube is an efficient method to improve query performance of OLAP. However, this method brings serious space and maintenance problems. Although this situation can be relived by materializing only a subset of all views, which has been extensively studied in recent years, this strategy results in significant difference of query performance between materialized views and non-materialized views. Another flaw of this strategy is that the cost model for selecting views often needs prior knowledge about how users query the data. However, it is very difficult to attain this knowledge.

For MOLAP systems, data cubes are typically stored in (sometimes compressed) data arrays[10]. Although this method can help to achieve better queries performance, the size of these data cube become more huge. A lot of papers tried to compress data cube by compressing the arrays[6], or only providing inaccurate answers for queries[1][7].

Recently several new structures, such as condensed cube[9], Dwarf[8], Quotient cube[4] and QC-Trees[5], are proposed to substantially reduce the size of the data cube. These structures have similar idea: when several different level aggregation values have the same certain characteristics, these structures can store them in one storage unit. In other words, they use one memory unit to present several cells in data cube, if these cell are computed by the same subset of the fact table's tuples.

In 2002, W. Wang[9] proposed the concept of BST(Base Single Tuple). If a cell SD in the data cube only covers one tuple r in the base table, then r is called the BST of SD. If r is a BST on some (preferably maximized) SD, the aggregation function should be applied only, and exactly once, to tuple r. Moreover, for every BST r, only one base tuple needs to be stored in the Condensed Cube.

Dwarf[8] has a similar and better idea. It adopts directed acyclic graph(DAG) to reduce prefix redundancy. To reduce suffix redundancy, it places Dwarf, several different level cells which can be aggregated from one set of base tables, in one storage unit.

In Quotient cube[4], the set of cells of a Data Cube are partitioned into certain number of equivalent classes. One equivalent classes, which contain one or some cell(s) all have the same certain characteristics, will be coalesced into one storage unit.

QC-Trees[5], which is the Quotient cube with *cover* equivalence, is one of the most efficient structures among them. For example, a cell c covers a base table tuple t whenever there exists a roll-up path from t to c. If the set of tuples in the base table covered by cell c and cell d are the same, then c and d are *cover* equivalent and their aggregation values are equal. In QC-Trees, both of them will be partitioned into one class. Another point of QC-Trees is that all cells in one class can be represented by the finest cell (called the *upper bound* of this class) in the class.

3 Partially Materialized Cells

In this section, we develop a data structure PMC(Partially Materialized Cells) to partially materialize QC-Tree at the cell level. Algorithms are also presented for the constructing , updating and querying of PMC. In this section, a part of algorithms are abridged for space reasons.

3.1 Motivation

In the QC-Tree, a node represents one or several cell(s) whose aggregation is not NULL in the data cube. On the other hand, each non-NULL cell can find the only corresponding node in the QC-Tree.

We noticed that a node in QC-Tree may possess many children (by edges or links), whose labels may belong to one of several dimensions. Moreover, the value of the node[1] is equal to the aggregates of all children which belong to any one of dimensions. For example, in Fig. 1, node 1 has 6 children: node 2,8,11(by edge) and node 5, 7, 10 (by link). The labels of node 2,8 belong to dimension *Store*, while 5,11 belong to *Product* and 7,10 belong to *Season*. The value of node 1 and the aggregation of all its children on any one dimension are the same: 27.

The point is, since the value of the node is equal to the aggregation of its children, the value of the node needn't to be stored if the cost of aggregating from its children is trivial. When querying, we can retrieve the value by temporarily aggregating the values of its children.

Thus, much of storage space can be saved. Moreover, the maintenance performance of the data cube can be significantly improved.

[1] In the following part of this paper, when we mention "the value of a node" , we are refering to "the aggregation value of the cell which is correspond to the node ".

Store	Product	Season	Price
S1	P1	s	$6
S1	P2	s	$12
S2	P1	f	$9

Table 1. The base table

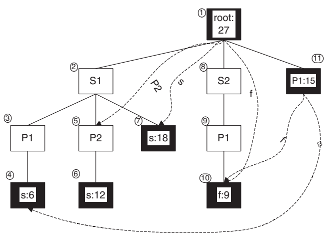


Fig. 1. The QC-Tree for Table 1

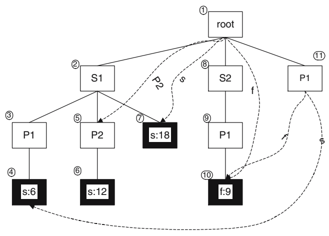


Fig. 2. The PMC for Table 1

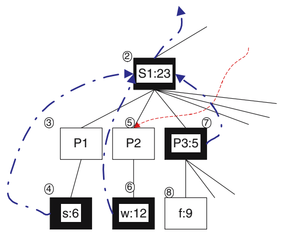


Fig. 3. A part of a slight complex PMC

For example, in a marketing management data warehouse, data are collected under the schema `sales(Store, Product, Season, Price)`. The base table, which holds the sales records, is shown in Fig. 1. Attributes `Store`, `Product` and `Season` are *dimensions*, while attribute `Price` is a *measure* and $SUM(Sale)$ as the aggregate function.

The *data cube*, with `Store`, `Product` and `Season` as dimensions and $SUM(Sale)$ as an aggregate function, is a set of results returned from being grouped by each subset of these three dimentions. Each group-by corresponds to a set of cells, described as tuples over the group-by dimensions.

A QC-Tree example for Table 1 is shown in Fig. 1, while Fig. 2 shows the PMC from the same base data, where we set the threshold for query costs of PMC to 2. Non-materialized nodes are marked with white boxes, while materialized nodes are marked with black boxes. The edges are denoted by solid lines, the link by the read thin broken lines. Compared to Fig. 1, node 1 and node 11 in Fig. 2 are no longer to be materialized because the cost of computing each of their values is not exceed 2. For example, A query for node 11 can be answered by aggregating values of two other nodes:node 4 and node 10. In fact, QC-Tree can be seen as the special case of PMC with threshold= 1.

Upon update problem, we have following considerations:

For a single cell, if it is to be updated, logically all cells roll up from it should be updated to ensure consistency. QC-Trees can achieve a better update performance than normal data cube just because a cell in QC-Trees may present several coarser cells whose cover set are identical with it. When the base cell is updated, values of those coarser cells needn't be modified because they are linked to the base cell. In other words, one modified action on the base cell also update all other cells with the same cover set, which is the reason why update operation can be reduced in QC-Trees.

So the more other coarser cells are linked to the cell, the more the update cost is reduced. In QC-Trees, an aggregated cell is only linked to one cell when both cells' cover sets are identical. In PMC, a non-materialized node can be linked to one or several materialized node(s). As a result, PMC goes a further step and makes more cells non-materialized.

Therefore, there are less nodes are to be update in PMC than in QC-Trees when maintaining. As a result, PMC can achieve better update performance than QC-Tree.

For example, in Fig. 1, if the value of node 10 are changed, the values of node 1 and node 11 should also be updated . However, in Fig. 2, such a change on node 9 will not cause a update on node 1 and node 11.

3.2 The Structure of PMC

PMC is similar to QC-Tree , except that

- each node in PMC has an additional value *cost*, which denotes the cost of retrieving the value of the node. For materialized node, the cost is set to 1. For non-materialized node, the cost is set to the time of access its descendants' values when aggregating through edges and(or) links.
- all non-materialized nodes don't store their value.
- (optional) each materialized node has a number of update-links which point to its direct materialized descendants. These update-link are used for accelerating the process of update.

In this paper, the value of *cost* of a node is defined as the number of accessing (materialized) node(s)' value(s) in the course of get the value of the node. For a materialized node, since its value is stored in the node itself, its cost is 1. To get the value of a non-materialized node, we need drill down and get the value by aggregating its descendants' values. Therefor, if a node has a large *cost*, we will spend much on retrieving its value when answering queries. In order to avoid this situation, we have this kind of nodes materialized so that we can retrieve any node's value easily.

Fig. 3 shows a part of a more complex PMC, where the update-links are marked by the blue thick broken lines.

Without loss of generality, we assume the type of aggregate function is SUM. In fact, other common aggregate functions like COUNT, MAX, MIN can also be the aggregate function of PMC, so does AVG, which can be denoted as SUM/COUNT.

3.3 Construction of PMC

The construction of PMC is in four steps.

- Firstly, PMC is constructed like a QC-tree. The difference is that our algorithm needn't compute the aggregate value for each temporary class. In a

```
Function $PointQuery$($CurrentNode$)
local variable: $localvalue$
initiate $localvalue$;
if $CurrentNode$ is materialized then
    $localvalue$=$CurrentNode.Vaule$;
else
    Find the drill dimension $D$ whose query cost is minimum;
    for each child $CD$ along dimension $D$ of $CurrentNode$ do
        $localvalue$=aggregate($localvalue$ , $PointQuery$($CD$));
    endfor;
endif;
return localvalue;

Function $Select$($CurrentNode$)
for every edge's child nodes $cd$ of $CurrentNode$ do
    $Select$($cd$);
endfor;
//re-computer $CurrentNode$'s cost
if cost($CurrentNode$) > $Threshold$ then
    $SetMaterialized$($CurrentNode$);
    $BuildUpdateLinks$($CurrentNode$,$CurrentNode$);
    compute and store $CurrentNode$'s aggregation value;
    /*now the cost of $CurrentNode$ becomes 1*/
endif;
return(cost($CurrentNode$));
```

Fig. 4. Function *PointQuery* and Function *Select*

QC-Tree, the aggregates of all temporary classes have to be computed, and a majority of them are repetitional and useless in the course of constructing QC-Tree.

– *Cost* of each node is initiated in the second step. In this step, only leaf nodes of the tree are materialized, so the *cost* of a node is in fact equal to the number of the base tuple covered by the node.
– In third step, function *select* is executed for the tree and selects the set of nodes to be materialized. The parameter $C_{threshold}$ of *select* is used to control this process. Those nodes whose *cost* are larger than $C_{threshold}$ will be materialized so that their *costs* are reduced to 1. By doing so, the cost of query of all cells in PMC can be limited below $C_{threshold}$.
– Finally, we perform *BuildUpdateLinks* to build update links for materialized nodes. The update links are set among materialized nodes. Each materialized node has update links which point to those nodes whose value is computed from this node. Of course, if we want build a PMC without update-links, we can jump over this step.

In Fig. 4, we present *select* function, which selects a set of nodes and then materializes them so that there is no node whose *cost* exceeds $Cost_{threshold}$ in PMC. It's interesting that if this $Cost_{threshold}$ is set to 1, the corresponding PMC is identical with the QC-Trees. It shows that QC-Tree is in fact the special case of PMC with $Cost_{threshold} = 1$.

For the sake of briefness, we do not show the detail of *BuildUpdateLinks*.

3.4 Query

Point Query The answering for point query is rather easy. If the node is materialized, we'll get the answer from the node immediately. Otherwise, the answer will be retrieved by computing its descendants' value. It's clear that the cost of this computation will not exceed $Cost_{threshold}$. Algorithm for point query is shown in Fig. 4.

3.5 Maintenance of PMC

To the PMC without update-links, the process of maintenance is similar to that of QC-Tree. The difference is that only materialized nodes need to be *updated* in PMC. As a result, less temporary "*update*" classes are produced in PMC and thus the maintenance of PMC should be slight faster than that of QC-Tree.

If we want a better update performance, update-links can be build for acceleration. In following part of this subsection, we'll address the issues concerned with the maintenance of the PMC with update-links.

There are three types of maintaining operations on the base table: insert, delete and update. Therefore, the change of source data can be represented simply by tuples inserted into, deleted from, and updated in the base table.

Update. For a newly coming tuple, if there exits a corresponding node in PMC, then a update operation starts. It's clear that the corresponding node is a leaf node. From subsection 3.3, we know that leaf nodes are materialized. Thus, The node is materialized and has update links.

In PMC, values of nodes are update along update links. When the value is to be updated, a (materialized) node will call the update functions of those nodes which are pointed by its update links. The update value for the current node will be passed as a parameter along these update functions. Thus, computing aggregate values of newly coming tuples for temporary classes is avoided.

If execute the update function for a leaf node, all its materialized ancestors will be updated.

If PMC is updated tuple by tuple, we can simply call the update functions of corresponding leaf nodes to accomplish the update of PMC.

Batch Update. In theory, batch update can be accomplished by perform *update* operations tuple by tuple. This strategy, however, is inefficient. In this way, if a node is updated for t times, then all its ancestors' value are bound to be modified for t times. In other words, if a node covers n newly coming tuples, the update function of this node will be executed for n times in the course of maintenance.

Therefore, for batch update, we adopt another strategy. Fig. 5 shows the function *BatchUpdate*. Each materialized node has a variable *tempV*, and it is initiated as 0. When the update function of this node is executed, the parameter $Value$, which is passed from one of its descendant, will used to cumulatively modify (aggregate) *tempV* instead of the value of this node. Moreover, other update functions will not be called by this update function. Only when it is the last time the update function of this node is executed, will *tempV* be used

```
Function $BatchUpdate$($CurrentNode$, $Value$)
$CurrentNode.Count_{to\_update}$--;
if $CurrentNode.Count_{to\_update}$ > 0 then
    $tempV$+=$Value$;
else
    $SetNewValue$($CurrentNode$, $CurrentNode.tempV$);
    for every $UpdateLink$ from $CurrentNode$ do
        $Update$($UpdateLink$.TargetNode, $CurrentNode.tempV$);
    endfor;
end if;
```

Fig. 5. Function *BatchUpdate*

to change the value of this node. And then, its ancestors' update functions are called along update links and take *tempV* as parameter.

This strategy has following merits:

1. No matter how many times will a node be updated , all its ancestors' update functions will be executed only *once*.

2. Since values for update in PMC are passed along update links, we needn't compute aggregate values of temporary classes in ΔDFS. Thus, a lot of data access and computations are avoided. Compared to it, computation for the initial value of $Count_{to\_update}$ needn't access the value of raw data and its complexity can be neglected.

Insertion and Deletion. When there is no tuple in the base table such that it has the same dimension values as the newly coming tuple, new classes should be split from a old class, or be created. In this paper, we view an insertion operation as the combination of (possibly) a node creation and a update operation.

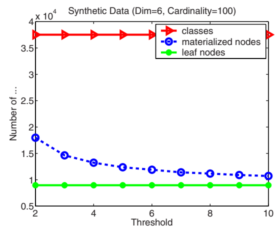
Similar to insertion operation, we view a deletion operation as the combination of an update operation and possible mergence or deletion of old node(s).

Due to the limitation of space, we skip the detail of this part.

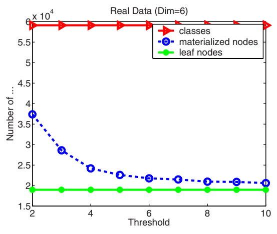
4 Experimental Results

In this section, synthetic and real data sets are used for examining the algorithms we presented in this paper. We focus on three main performance issues(storage, queries answering and maintenance) of PMC and compare them with those of the QC-Trees.

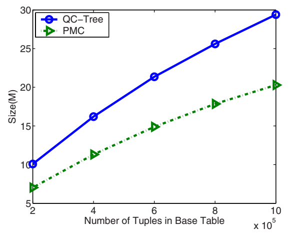
There are two classes of data sets used in the experiments. The first one is synthetic data with Zipf distribution. The synthetic data set contains 1,000,000 tuples and 6 dimensions, the cardinality of each dimension is 100. The other one is the real data set containing weather conditions at various stations on land for September 1985[3]. The weather data set contains 1,015,367 tuples and 6 dimensions. The attributes are as follows with cardinalities listed in parentheses: longitude (352), solar-altitude (179), latitude(152), present-weather (101),



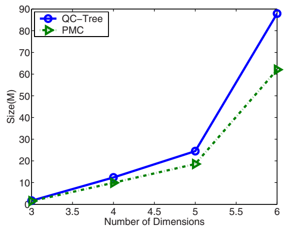
(a) threshold vs. storage (synthetic data)



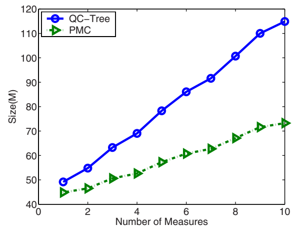
(b) threshold vs. storage (real data)



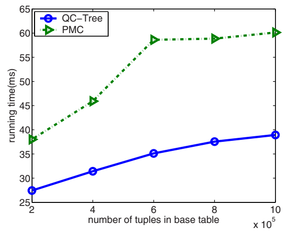
(c) number of base tuples vs. storage (synthetic data)



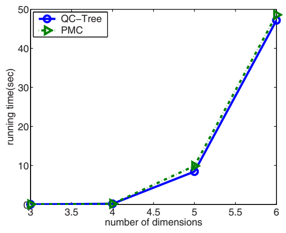
(d) number of dimensions vs. storage (real data)



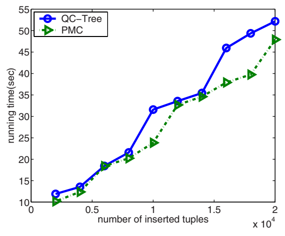
(e) number of measures vs. storage (real data)



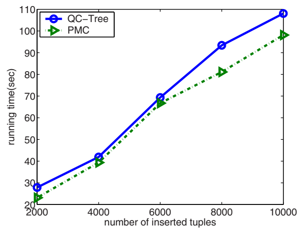
(f) Query Performance (synthetic data)



(g) Query Performance (real data)



(h) Maintenance Performance (synthetic data)



(i) Maintenance Performance (real data)

Fig. 6. Experimental Results

weather-change-code(10), and hour (8). All experiments are running on a Pentium 4 PC with 256MB main memory and the operation system is Windows XP.

4.1 Storage

In QC-Tree, the number of data units is equal to the number of classes, while in PMC it is equal to the number of materialized node. Fig. 6(a) and 6(b) illustrate the value of thresholds versus the number of data storage units needed. The results show that small thresholds(like 2 or 3) can achieve satisfying balance between storage and query. In Fig. 6(a) and 6(b), number of tuples in the base table is 20000 .

Fig. 6(c) shows the results of storage vs. number of base tuples, while Fig. 6(d) shows storage vs. the number of dimensions and Fig. 6(e) shows storage vs. the number of measures. The results illustrate that PMC can compresses the data cube more efficiently than QC-Tree.

4.2 Queries Answering

In this experiment, we compared query answering performance of PMC and QC-tree. We randomly generated 1,000 point queries and range queries on each data set. Fig. 6(f) and 6(g) show the times for answering point queries on synthetic data and range queries on real data separately. Other experiments have similar results.

The results show that the query performance of PMC is only a little worse than that of QC-Tree.

4.3 Maintenance

To test the performance of the incremental maintenance of PMC, we generated different size of data, inserted them into a size-fixed PMC. The tuples are batch inserted. The results are shown in Fig. 6(h) and 6(i).

The results show that the maintenance performance of PMC is better than QC-Tree. The reasons are: (1). our algorithm will conduct less aggregate operation when generating temporary classes; (2)there are less materialized nodes to be update in PMC.

5 Conclusions

In this paper, we present algorithms to partially materialize cells in the data cube. Compared with extensively studied view selection algorithms, our algorithms deal with cells in compressed structure of data cube. Along with reduced space and update cost, our algorithm can ensure a stable query performance.

Our algorithms can achieve a better tradeoff between storage, update and query performances of a Data cube. As a example in this paper, our algorithms are applied to QC-tree. In fact, other cell-level structure, such Dwarf or Condensed Cube, can also adapt our algorithms with appropriate modifications.

References

1. K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In Proc, SIGMOD, pages 359-370, 1999.
2. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In Proc, ICDE, pages 152-159, 1996.

3. C. Hahn et al. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. cdiac.est.ornl.gov/ftp/ndp026b/SEP85L.Z,1994.
4. Laks V. S. Lakshmanan, Jian Pei, Jiawei Han, Quotient Cube: How to Summarize the Semantics of a Data Cube. In Proc, VLDB, pages 778-789, 2002.
5. Laks V. S. Lakshmanan, Jian Pei, Yan Zhao: QC-Trees: An Efficient Summary Structure for Semantic OLAP. In Proc, SIGMOD, pages 64-75, 2003.
6. J. Li, Y. Li, J. Srivastava: Efficient Aggregation Algorithms on Very Large Compressed Data Warehouses. J. Comput. Sci. Technol. 15(3): 213-229 (2000).
7. J. Shanmugasundaram, U. Fayyad, and P. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In Proc, SIGKDD, pages 223-232, 1999.
8. Y. Sismanis, A. Deligiannakis, N. Roussopoulos, Y. Kotidis, Dwarf: shrinking the PetaCube. In Proc, SIGMOD, pages 464-475, 2002.
9. W. Wang, H. Lu, J. Feng, and J. Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In Proc, ICDE, pages 155-165, 2002.
10. Y. Zhao, P. DeshPande, and J. Naughton, An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In Proc, SIGMOD, pages 159-170, 1997.