QC-Trees: An Efficient Summary Structure for Semantic OLAP*

Laks V.S. Lakshmanan† Jian Pei‡ Yan Zhao†
† University of British Columbia, Canada, {laks, yzhao}@cs.ubc.ca
‡ State University of New York at Buffalo, jianpei@cse.buffalo.edu

ABSTRACT

Recently, a technique called quotient cube was proposed as a summary structure for a data cube that preserves its semantics, with applications for online exploration and visualization. The authors showed that a quotient cube can be constructed very efficiently and it leads to a significant reduction in the cube size. While it is an interesting proposal, that paper leaves many issues unaddressed. Firstly, a direct representation of a quotient cube is not as compact as possible and thus still wastes space. Secondly, while a quotient cube can in principle be used for answering queries, no specific algorithms were given in the paper. Thirdly, maintaining any summary structure incrementally against updates is an important task, a topic not addressed there. In this paper, we propose an efficient data structure called *QC-tree* and an efficient algorithm for directly constructing it from a base table, solving the first problem. We give efficient algorithms that address the remaining questions. We report results from an extensive performance study that illustrate the space and time savings achieved by our algorithms over previous ones (wherever they exist).

1. INTRODUCTION

The data cube [8] is an essential facility for data warehousing and OLAP. Conceptually, a data cube is a multi-level, multi-dimensional database with various multiple granularity aggregates. It is a generalization of the group-by operator, and contains group-bys corresponding to all possible combinations of a list of dimensions.

EXAMPLE 1 (DATA CUBE). In a marketing management data warehouse, data are collected under the schema sales(Store, Product, Season, Sale). The *base table*, which holds the sales records, is shown in Figure 1. Attributes Store, Product and Season are called *dimensions*, while attribute Sale is called a *measure*.

Store	Product	Season	Sale
$S1$	$P1$	s(spring)	6
$S1$	$P2$	s(spring)	12
$S2$	$P1$	f(fall)	9

Figure 1: Base table sales for a data warehouse.

A *data cube* grouped by Store, Product, Season using an aggregate function such as AVG(Sale) is the set of results returned from the 8 group-by queries, with each subset of {Store, Product, Season} forming a group-by. Each group-by corresponds to a set of *cells*, described as tuples over the group-by dimensions. The cells in the data cube $Cube_{Sales}$ are shown in Figure 2(a). Here, symbol "$*$" in a dimension means that the dimension is generalized such that it matches any value in its domain. ■

Two basic semantic relations among cells of a data cube are the *drill-down* relation and its dual, the *roll-up*. A cell c_1 rolls up to cell c_2 (c_2 drills down to c_1), if c_2 generalizes c_1 in some dimensions, i.e., in all dimensions where c_1 and c_2 have different values, c_2 has value "$*$". E.g., in the data cube in Figure 2(a), cell $(S1, P1, s)$ rolls up to $(S1, *, s)$. Cell $(S1, *, s)$ represents a higher level aggregate (i.e., the sales of *ALL products* in store $S1$ and in spring) than cell $(S1, P1, s)$ (i.e, the sales of product $P1$ in store $S1$ and in spring). A cube's cells together with the drill-down (or roll-up) relation form a lattice. Figure 2(b) shows the lattice for the data cube cells in Figure 2(a), where the top element, *false*, is not shown.

Because of their importance for efficient query answering and support for exploration and analysis, there have been numerous works on computing data cubes efficiently from a base table [1, 5, 20, 30], constructing a cube with various constraints [5], compressing [24, 25, 27] and approximating a data cube [3, 4, 26], etc. However, there are some inherent problems that current techniques cannot handle well.

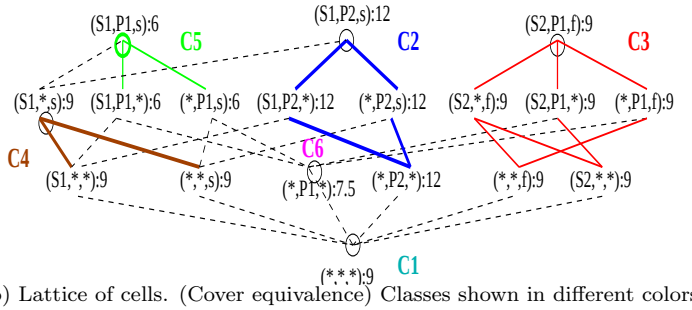
(1) Many kinds of critical semantic relationships among aggregate cells in a data cube are not captured. E.g., in Figure 2, tuple $(S2, P1, f)$ is the *only* contributor to the aggregates in cells $(S2, *, f)$, $(S2, P1, *)$, $(*, P1, f)$, $(*, *, f)$, and $(S2, *, *)$. A one-step roll-up from cell

*Research supported by a grant from NSERC (Canada).

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

Store	Product	Season	AVG(Sale)
$S1$	$P1$	s	6
$S1$	$P2$	s	12
$S2$	$P1$	f	9
$S1$	$*$	s	9
$S1$	$P1$	$*$	6
$*$	$P1$	s	6
...	...	...	...
$*$	$*$	f	9
$S2$	$*$	$*$	9
$*$	$*$	$*$	9

(a) Cells in data cube $Cube_{Sales}$.



(b) Lattice of cells. (Cover equivalence) Classes shown in different colors, and as maximal sets of cells connected by solid lines for black and white copies; upper bounds circled; $C_1 = \{(*,*,*)\}$ & $C_6 = \{(*, P1, *)\}$ – singleton classes; *false* cell not shown.

Figure 2: Data cube $Cube_{Sales}$

$(S2, P1, f)$ along any dimension will not give the user any new fruitful aggregate information. This kind of semantic relationships among aggregate cells is critical for achieving effective OLAP services.

(2) In practice, a data cube lattice is often huge. E.g., even without any hierarchy in any dimension, a 10-dimension data cube with a cardinality of 100 in each dimension leads to a lattice with $101^{10} \approx 1.1 \times 10^{20}$ cells. Assuming a sparsity of 1 out of a million, we still have a lattice with 1.1×10^{14} non-empty cells! Suppose a manager wants to identify exceptions by browsing the data. The manager has no idea on which dimensions to roll-up or drill-down. Many steps in her exploration may be just fruitless, because they solely rely on roll-up/drill-down. To provide an effective navigation service, a more powerful metaphor is needed.

(3) Compression techniques should help since they cut down the size. However, almost all approaches proposed previously are *syntactic*, in that even the roll-up/drill-down relation is lost in the compressed representation. For both query answering and browsing the cube, it may need to be uncompressed, causing significant overhead.

In summary, one needs a summary structure on the data cube that captures the cube lattice's inherent roll-up/drill-down links as well as interesting regularities that will help cut down the size. Conventional techniques do not meet these goals. Without a proper semantic summarization, users may not be able to fully understand and make use of the information from a huge data cube. This motivates our study on the effective semantic summarization in a data cube to support OLAP.

Recently, a study [14] proposed a novel conceptual structure called quotient cube, for semantic summarization of data cubes. The idea is to partition the cube cells by grouping cells with identical aggregate values (and satisfying some additional conditions) into (equivalence) classes *while keeping the cube's drill-down links and lattice structure*.

A quotient cube is a representation of a cube lattice in terms of classes of cells. The drill-down is captured as a partial order among classes. Each class contains cells that have the same aggregate value(s). Thus, each class captures a regularity – group of adjacent cells having the same aggregates. In addition, whereas a user drills down from a cell to another in the cube lattice, in a quotient cube, he drills down from a class to another! Conceptually, one can also drill down *into* a class, asking to open it up and inspect its internal structure. For example, one quotient cube w.r.t. the data cube in Figure 2(a) is highlighted in Figure 2(b) and shown in Figure 3. It has 6 classes. Every cell in a class has the same aggregate value. Figure 3 also demonstrates a drill-down into class C_3 in the quotient cube.

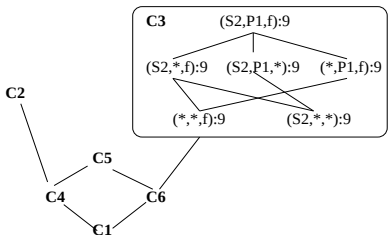


Figure 3: Quotient cube showing drill-down into a class' internal structure.

Compression comes from having to store only the lower and upper bound cells for each class. E.g., $(*,*,f)$, $(S2,*,*)$ are the lower bounds and $(S2, P1, f)$ is the upper bound of class $C3$ (see Figure 2(b)). Semantics is preserved since classes form a lattice with a drill-down consistent with the cell drill-down. [14] proposed a number of ways of constructing quotient cubes based on the partitioning method used.

We can answer various queries and conduct various browsing and exploration operations using a quotient cube directly. E.g., we have found that it can support advanced analysis such as *intelligent roll-up* (i.e., finding the most general contexts under which the observed patterns occur) [23] and what-if queries [2]. For brevity, we only give the intelligent roll-up example.

For example, a manager may ask an intelligent roll-up query "*in the data cube in Figure 2, starting from* $(S2, P1, f)$*, what are the most general circumstances where the average sales is still* 9*?*" The answer should be $(*,*,*)$ *except for* $(*, P1, *)$ *and* $(*, P2, *)$. If we search the lattice in Figure 2(b), we may have to search the 7 descendent cells of $(S2, P1, f)$. However, if we search in the class lattice in Figure 3, we only need to search at most 2 classes, and not any specific cell in any class. Thus, semantic compression reduces the search

space for analysis and exploration. Furthermore, presenting the answer in the form of classes also facilitates understandability.

Finally, drill-down from apparently unrelated cells may lead to the same class of cells. E.g., in Figure 2, a drill-down from cell $(*, *, *)$ as well as one from $(*, P1, *)$ (in both cases, by setting dimension 3 to f) both lead to the same class of cells, C_3. This means first drilling down from $(*, *, *)$ to $(*, P1, *)$ and from there to $(*, P1, f)$ versus drilling down from $(*, *, *)$ to $(*, *, f)$ both yield equivalent cells. This may suggest an interesting pattern for the user.

While the quotient cube is an interesting idea, the study in [14] leaves many problems unanswered. In particular, the following important questions remain.

- Given that quotient cubes can be constructed in more than one way, what kind of quotient cubes should be used to construct a general purpose data warehouse?

  Our contributions. In this paper, we recommend using cover partition (Section 2.2) for constructing a quotient cube-based data warehouse. We show that such a data warehouse can support a variety of OLAP queries effectively and compress the data cube substantially.

- How to implement a quotient cube-based data warehouse efficiently? How to store and index data in a quotient cube and how to use them to answer various queries efficiently?

  Our contributions. We devise (Section 3) *QC-tree*, an effective data structure to compress and index quotient cubes w.r.t. coverage partitions. We give an efficient algorithm (Section 3.2) for constructing a QC-tree directly from the base table. We develop efficient algorithms (Section 4) for answering point queries, range queries, and iceberg queries.

- How to incrementally maintain a quotient cube-based data warehouse?

  Our contributions. We develop scalable algorithms (Section 3.3) to incrementally maintain a QC-tree, and establish their correctness. We ran extensive tests (Section 5) for (i) comparing the time and space efficiency, as well as query answering speed of our QC-tree algorithm with the recently proposed Dwarf Cube [25], and (ii) for measuring the extent of savings of our incremental maintenance algorithms over a recompute, as well as demonstrating their scalability. We discuss our experiments and explain our results.

In Section 2.1, we review the main concepts and important properties of quotient cubes. We discuss related work in Section 6. Section 7 summarizes and concludes the paper.

2. QUOTIENT CUBES

In this section, we briefly review the concepts and important properties of quotient cubes proposed in [14], to which we refer the reader for details. Then, we consider a specific kind of quotient cubes, *cover quotient cubes*. They have some nice properties enabling them to be used as a general purpose summary structure for answering various aggregate queries.

2.1 Quotient cubes and their properties

The key idea behind a quotient cube is to create a summary structure by carefully partitioning the set of cells of a cube into classes such that cells in a class all have the same aggregate measure value and in addition, satisfy some desirable properties. We wish to use each class as a concise summary for all its cells. In addition, a quotient cube captures the roll-up/drill-down semantics between cells, present in the original cube, between its classes. Summary structures obtained from arbitrary partitions do *not* achieve this.

As an example, suppose in Figure 2(b), we partition cells solely on the basis of equality of aggregate values (ignore the colors for now). Let C be the class of all cells with aggregate value 9 and D contain the only cell with aggregate 7.5. We can drill down from cell $(*, *, *)$ in C to cell $(*, P1, *)$ in D, and then again drill down from $(*, P1, *)$ in D to $(*, P1, f)$ in C! What went wrong here is that cells $(*, *, *)$ and $(*, P1, f)$ belong to the same class (C), but not $(*, P1, *)$ that lies in between these two cells. Let $\prec$ be the cube lattice partial order. A class that contains cells c, d but not e, where e is some cell such that $c \prec e \prec d$, is said to have a *hole*. Classes without holes are called convex and a partition all of whose classes are convex is a *convex partition*. The partition in the above example is non-convex. The partition illustrated in Figure 2(b) *is* convex.

One of the main results of [14] is that convex partitions on cube lattices lead to a set of classes that themselves form a lattice, called the *quotient lattice*. The ordering among classes is defined as follows: for classes C, D, $C \prec D$ whenever there are cells $c \in C$ and $d \in D$ such that $c \prec d$ in the original cube lattice. Thus, the use of a quotient lattice in place of the original cube lattice leads to a summary structure, the quotient cube, that captures the roll-up/drill-down semantics of the original cube, in terms of this partial order among its classes.

In [14], techniques were developed for defining partitions for various aggregate functions. They define two cells c, d to be equivalent w.r.t. to an aggregate function f, denoted $c \equiv_f d$, whenever they satisfy one of the following conditions: (i) c and d have the same f-value *and* c is a parent or child of d, *or* (ii) there is a cell e such that c and d are both equivalent to e.

2.2 Quotient cubes w.r.t. cover partitions

Among other things, [14] proposed a generic way of partitioning cells without reference to any particular aggregate function, based on the so-called notion of "cover". A cell c *covers* a base table tuple t whenever there exists a roll-up path from t to c, i.e., $c \prec t$ in the cube lattice. The cover set of c is the set of tuples in the base table covered by c. E.g., in Figure 1, the first two tuples both can be rolled up to cell $(S1, *, s)$, since they both agree with this cell on every dimension where

its value is not "$*$". Indeed, the cover set of $(S1, *, s)$ is $\{(S1, P1, s), (S1, P2, s)\}$. Two cells c and d are *cover equivalent*, denoted $c \equiv_{Cov} d$, whenever their cover sets are the same. E.g., in Figures 1 and 2(b), the cells $(S1, *, s)$ and $(S1, *, *)$ have identical cover sets and are cover equivalent. In fact, the (cover equivalence) class containing these cells is $\{(S1, *, s), (S1, *, *), (*, *, s)\}$. We define a class upper (lower) bound to be any maximal (minimal) element in the class. We have the following.

LEMMA 1 (COVER PARTITIONS). *[14]* The partition induced by cover equivalence is convex. Cover equivalent cells necessarily have the same value for any aggregate on any measure. Each class in a cover partition has a unique upper bound. ∎

The quotient cube of a data cube w.r.t. the cover partition is called *cover quotient cube*. Figure 3 shows the cover quotient cube of the cube lattice of Figure 2(b), where the class associated with *false* is not shown. There are 6 classes besides it. They are shown along with their upper bounds (circled in Figure 2(b)). E.g., the upper bound of class C_3 is $(S2, P1, f)$. We refer to immediate successors (predecessors) w.r.t. $\prec$ in the quotient lattice as *parents* (*children*). E.g., C_6 is a child of $C3$ and C_5 in Figure 3. *In the rest of the paper, we mainly focus on cover partitions, and cover quotient cubes, unless otherwise specified.*

3. QC-TREES

In this section, we develop *QC-tree*, a compact data structure to store and search a quotient cube efficiently as well as algorithms for constructing a QC-tree directly from the base table and for incrementally maintaining it under base table updates.

3.1 The QC-tree structure

We seek a compact data structure for representing and implementing quotient cube. Such a data structure should (a) retain all the essential information in a quotient lattice, yet be concise; (b) enable efficient answering of various kinds of queries including point, range, and iceberg queries; and (c) afford efficient maintenance against updates. *QC-tree* (short for quotient cube tree) developed here meets all the requirements.

The key intuition, which will be established in stages in the rest of the paper, is that the set of class upper bounds in a quotient cube w.r.t. cover partition captures all essential information about classes in the quotient cube. We represent this information compactly as a tree by prefix sharing, where bounds are represented as strings. We then add links for capturing some additional drill-down relations.

Figure 4 shows the QC-tree associated with the quotient lattice of Figure 3. Let Q be a quotient cube and $\mathbf{B}$ its set of class upper bounds. Each bound can be represented as a string w.r.t. a given dimension order, by omitting all "$*$" values. E.g., $(S1, P1, f)$ and $(*, P2, *)$ would be represented as $S1 \cdot P1 \cdot f$ and $P2$, respectively, w.r.t. the dimension order `Store-Product-Season`.[1] In

Class	Upper Bound
C_1	$(*, *, *)$
C_2	$(S1, P2, s)$
C_3	$(S2, P1, f)$
C_4	$(S1, *, s)$
C_5	$(S1, P1, s)$
C_6	$(*, P1, *)$

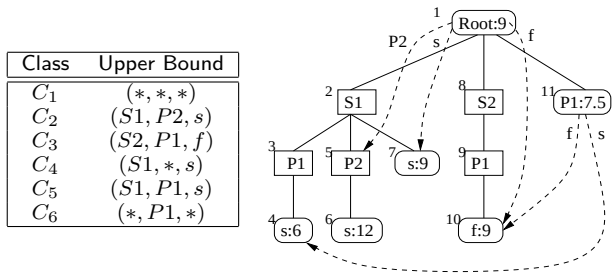


Figure 4: Classes and QC-tree for the quotient cube of Figure 2.

the sequel, we shall assume an arbitrary fixed dimension ordering and identify a bound with its string representation.[2] Given a sequence of values $z_1 \cdot \ldots \cdot z_k$, some of which may be "$*$", we use $\langle z_1, \ldots, z_k \rangle$ to denote the node in the QC-tree which corresponds to the string representation of this sequence, omitting "$*$"s. E.g., for $* \cdot P1 \cdot *$, $\langle *, P1, * \rangle$ is node 11 in Figure 4, while for $S1 \cdot * \cdot s$, $\langle S1, *, s \rangle$ is node 7.

DEFINITION 1 (QC-TREES). The *QC-tree* for a quotient cube Q is a directed graph (V, E) such that:

1. $E = E' \cup E''$ consists of tree edges E' and links E'' such that (V, E') is a rooted tree.
2. each node has a (dimension) value as its label.
3. for each class upper bound b, there is a unique node $v \in V$ such that the string representation of b coincides with the sequence of node labels on the path from the root to v in the tree (V, E'). This node stores the aggregate value associated with b.
4. suppose C, D are classes in Q, $b_1 = (x_1, \ldots, x_n)$ and $b_2 = (y_1, \ldots, y_n)$ are their class upper bounds, and that C is a child of D (i.e., C directly drills down to D) in Q. Then for every dimension Dim_i on which b_1 and b_2 differ, there is either a tree edge or a link (labeled y_i), but not both, from node $\langle x_1, \ldots, x_{i-1} \rangle$ to node $\langle y_1, \ldots, y_i \rangle$. ∎

Definition 1 is self-explanatory, except that the last condition looks technical: it says whenever class C directly drills down to class D in the quotient cube, one can drill down either via a tree edge or a link from some prefix of the path representing C's upper bound to a node which will eventually lead to D's upper bound.

In Figure 4, there is a node representing every class upper bound of the quotient lattice of Figure 2(b). E.g., node 7 represents $(S1, *, s)$ and stores the corresponding aggregate value 9. One can drill down from C_6 (whose upper bound is $(*, P1, *)$) to C_5 (whose upper bound is $(S1, P1, s)$) in the quotient lattice. The two upper bounds differ in two dimensions: `Store` and `Season`.

[1] Strictly speaking, we should represent each value v as $D = v$, D being the dimension name. We omit this obvious formality for simplicity.

[2] Heuristically, dimensions can be sorted in the cardinality ascending order, so that more sharing is likely achieved at the upper part of the tree. However, there is no guarantee this order will minimize the tree size.

The first difference (i.e., a drill-down from class C_6 to class C_5 by specifying $S1$ in dimension `Store`) is indicated by a tree edge from node $1 = \langle * \rangle$ to node $2 = \langle S1 \rangle$ in the QC-tree. The second difference (i.e., a drill-down from C_6 to C_5 by specifying s in dimension `Season`) is denoted by a link from node $11 = \langle *, P1 \rangle$ to $4 = \langle S1, P1, s \rangle$ with label s. There is no link from node 11 to 6 or 7 since there is no drill-down relationship there. In Figure 3, there are certain direct drill-down relationships between classes that are *not* a child-parent pair: e.g., we can drill down from C_1 to C_2 by specifying `Product`=P2. Such drill-downs exactly correspond to the drill-down links in the QC-tree (e.g., the link from node 1 to node 5 labeled P2). *Note that the parent-child relationship in a quotient lattice is reversed in a QC-tree*: e.g., C_1 is a child of C_6 in the lattice (Figure 3) while C_1 is the root of the tree (Figure 4) and has child C_6. For consistency, we always refer to the lattice relationship and speak lattice child and lattice parent.

THEOREM 1 (UNIQUENESS OF A QC-TREE). Let Q be a data cube and QC be the corresponding cover quotient cube. Let $R : D_1, \ldots, D_n$ be any order of dimensions. Then: (1) for each class C, there is a unique path P from the root in the QC-tree (in order R) such that the sequence of the node labels on P is exactly the non-$*$ dimension values in ub, where ub is the upper bound of C; and (2) the QC-tree is unique. ∎

For any cell in a data cube, we can quickly obtain its aggregate values by simply following the path in the QC-tree corresponding to its class upper bound. We defer the details of this to Section 4.

3.2 Construction of a QC-tree

The construction of a QC-tree is in two steps. First, a variant of the Depth-First Search algorithm in [14] is used to identify all classes in a quotient cube. For each class C, two pieces of information are maintained: the upper bound and the ids of C's lattice child classes. *Compared to algorithm Depth-First Search, a key advantage of QC-tree construction algorithm is that it only maintains the upper bounds. No information about lower bounds is ever needed.* Depth-First Search generates a list of *temporary classes*. Some of them could be "redundant", i.e., they share the same upper bound as previous ones. Instead of pursuing a post-processing to merge the redundant classes and obtain the (real) classes, we identify the redundant temporary classes in the next step. If the upper bound of a temporary class exists in the tree when the upper bound is inserted, the redundancy is identified and removed.

As the second step, we sort all the upper bounds in dictionary order, where a user-specified order of dimensions is used. Within each dimension, we assume values are ordered arbitrarily except "$*$" precedes other values. Then, the upper bounds are inserted into the QC-tree, and drill-down links are built whenever necessary, according to Definition 1. Figure 5 shows the algorithm for QC-tree construction, which the following example illustrates.

ALGORITHM 1. *[Construct QC-tree]*
Input*: base table B .*
Output*: QC-tree for cover quotient cube of B.*
Method*:*

```
1.  b = (*, ..., *);
2.  call DFS(b, B, 0, −1); //(*, ..., *) has no lattice child;
3.  Sort the temp classes w.r.t. upper bounds
    in dictionary order (''*'' precedes other values)
    //Insert the temp classes one by one into QC-tree
4.  Create a node for the first temp class as the root;
5.  last = first temp class's upper bound;
6.  while not all temp classes have been processed
      current = next class's upper bound;
      if (current ≠ last)
        insert nodes for current;
        last = current;
      else
        Let ub be the current's child class's upper bound,
          lb be the lower bound of current;
        Find the first dim D s.t. ub.D = * && lb.D ≠ *;
        Add a drill-down link with label D from ub to last;
7.  return ;
```

Function $DFS(c, B_c, k, chdID)$

```
// c is a cell and B_c is the corresponding partition of the
base table;
1.  Compute aggregate of cell c;
2.  Compute the upper bound d of the class containing c, by
    ''jumping'' to the appropriate upper bounds;
3.  Record a temp class with lower bound c,upper
    bound d and child chdID. Let clsID be its class ID;
4.  if there is some j < k s.t. c[j] = all and d[j] ≠ all
       return; //such a bound has been examined before
5.  else for each k < j < n s.t. d[j] = all do
       for each value x in dimension j of base table
         let d[j] = x;
         form partition B_d;
         if B_d is not empty, call DFS(d, B_d, j, clsID);
6.  return;
```

Figure 5: QC-tree Construction.

EXAMPLE 2 (QC-TREE CONSTRUCTION). Let us build the QC-tree for the base table in Figure 1. In the first step, we identify all (temporary) classes by a depth-first search starting from cell $(*, *, *)$. We calculate the aggregate of cell $(*, *, *)$. Since there is no single dimension value appearing in all tuples in the base table, $(*, *, *)$ forms a class with itself as the upper bound. Then, the search explores the first dimension, `Store`. The tuples in the base table are sorted on this dimension. There are two stores. Thus two partitions are generated. In the first partition, i.e., the tuples about sales in store $S1$, dimension value s appears in every tuple, so we identify $(S1, *, s)$ as an upper bound. The search recurs by exploring various products.

class ID	Upper Bound	Lower Bound	Lattice Child	Agg
i_0	(*,*,*)	(*,*,*)	-1	9
i_5	$(*, P1, *)$	$(*, P1, *)$	i_0	7.5
i_1	$(S1, *, s)$	$(S1, *, *)$	i_0	9
i_9	$(S1, *, s)$	$(*, *, s)$	i_0	9
i_2	$(S1, P1, s)$	$(S1, P1, s)$	i_1	6
i_6	$(S1, P1, s)$	$(*, P1, s)$	i_5	6
i_3	$(S1, P2, s)$	$(S1, P2, s)$	i_1	12
i_8	$(S1, P2, s)$	$(*, P2, *)$	i_0	12
i_4	$(S2, P1, f)$	$(S2, *, *)$	i_0	9
i_7	$(S2, P1, f)$	$(*, P1, f)$	i_5	9
i_{10}	$(S2, P1, f)$	$(*, *, f)$	i_0	9

Figure 6: Temporary classes returned by a depth-first search.

The temporary classes returned from the depth-first search are shown in Figure 6. They are sorted in dic-

tionary order and inserted into a tree. First, we process classes i_0, i_5, i_1 and create the corresponding nodes in the QC-tree (nodes 1, 11, and 7 in Figure 4). Then, class i_9 comes in. Its upper bound has already been inserted (see class i_1). We only need to build a drill-down link. Class i_0 is its lattice child, so we compare the upper bounds of classes i_9 and i_0 to get the drill-down link label, s, and so add a drill-down link with label s from $(*, *, *)$ (node 1) to $(S1, *, s)$ (node 7). Similarly, a link labeled f from node 11 (class i_5) to 10 (class i_7) is added. ∎

3.3 Incremental maintenance of a QC-tree

Many applications encounter frequent updates to base data. Thus, fast maintenance of QC-tree is important. In this section, we discuss algorithms for incremental maintenance against insertions and deletions. Modifications can be simulated by deletions and insertions.

3.3.1 Insertions

In a cover quotient cube, an insertion of a base table tuple may cause updating of the measure of an existing class, splitting of some classes, or creating of a new class, never merging of classes.

First, consider inserting one tuple into the base table. Let QC be the quotient cube, t be the new tuple. Two situations may happen.

Case 1: *t has the same dimension values as an existing tuple in the base table.* Each class, *all* of whose member cells cover this tuple, should have its aggregate measure updated to reflect the insertion of the new tuple.

Case 2: *there is no tuple in the original base table such that it has the same dimension values as the newly inserted tuple.* E.g., let the newly inserted tuple be $t = (S2, P2, f)$. *What are the cells whose cover set is changed by t?* Clearly, they are generalizations of $(S2, P2, f)$, e.g., $(*, P2, f)$, $(S2, *, f)$, ..., $(*, *, *)$. Let C be a class in the QC, let $ub = (x_1, \ldots, x_n)$ be its upper bound, and $t = (y_1, \ldots, y_n)$ be the newly inserted tuple. Let $t \wedge ub = (z_1, \ldots, z_n)$ be the greatest lower bound of t and ub, defined as $z_i = x_i$ if $x_i = y_i$, and $z_i = *$ otherwise, $(1 \leq i \leq n)$, i.e., $t \wedge ub$ agrees with ub on every dimension that ub and t agree on, and is "$*$" otherwise. All the classes in QC can be divided into three categories based on whether a class has a member cell that covers t.

Category 1: *the class does not contain any member cell that covers t.* Class C is in category 1 iff $t \wedge ub$ is not in C. For class C_5 in our running example, $t \wedge ub = (*, *, *)$ which is not in C_5. Since the cover set of its member cells does not change, this class remains unchanged.

Category 2: *the upper bound of the class covers t,*i.e., every member cell of the class covers t. Class C is in this category iff $t \wedge ub = ub$, e.g., C_1 in QC. Then the only change is that the tuple t is added to the cover set of *every* member cell of this class, signifying an update to the measure. No other change to the class is necessary.

Category 3: *the upper bound of the class does not cover t, but some member cell in the class does.* A class C is in this category iff $ub \wedge t \neq ub$ but $ub \wedge t$ belongs to the class C, e.g., for class C_3 in QC, the member cell $(S2, *, f)$ and all its descendants in C_3 cover $(S2, P2, f)$, while the upper bound $(S2, P1, f)$ of C_3 doesn't.

Since tuple t is added to the cover set of some but not all member cells, the class C should then be split into two classes. First, we form a new class with upper bound $ub \wedge t$ and all its descendants inside C as member cells. The remaining cells in C form a second class, with the same upper bound as the old C. For example, $(S2, *, f)$, $(*, *, f)$, and $(S2, *, *)$ in class C_3 form a separate class C' with upper bound $ub \wedge t = (S2, *, f)$. All other member cells (whose cover sets have not been changed) form another class, C'', which contains $(S2, P1, f)$, $(S2, P1, *)$, and $(*, P1, f)$, whose upper bound is still $(S2, P1, f)$. Parent child relationships are then easily established by merely inspecting the upper bounds of classes C', C'' as well as all parent and child classes of the old class C in the original QC lattice.

Finally, we need to create a new class for t. The member cells of the new class are those aggregate cells whose cover sets contain only t.

It is worth noting that we can show that new classes created by our strategy are always convex and are maximal.

So far, our discussion considered only the quotient cube, not the QC-tree. Besides, when we have multiple insertions, updating the quotient cube one by one is inefficient. We next propose a batch insertion algorithm for QC-trees which directly updates the QC-tree. The idea is that during a depth-first generation of temp classes on the set of inserted tuples $\triangle DB$, a process henceforth refered as $\triangle DFS$, we simultaneously record the classes to be updated on the original QC-tree QC and the new classes to be created (or split) as necessary (see Figure 7).

```
ALGORITHM 2 (INSERTION IN QC-TREE).
Input: a QC-tree T, base table ΔDB;
Output: new QC-tree;
Method:
 1.  Let b = (*, ..., *); call △DFS(b, ΔDB);
 2.  Sort the temp classes s.t.  the upper bounds are
     in the dictionary order,* goes to the first
 3.  last = NULL;
 4.  while not all temp classes have been processed
       current = next class;
       if (current's upper bound ≠ last's upper bound)
          if (current is an updated class)
             update the corresponding node in QC-tree;
          if (current is a new class)
             insert current's upper bound into QC-tree
          if (current is a split class)
             split current's upper bound from the old class;
          last = current;
       else
          Let ub be the current's lattice child class's
             upper bound,ub' be the upper bound of current;
          Find the first dim D s.t.ub.D = * && ub'.D ≠ *;
          Add a link with label D from ub to last;
 6.  return;
```

Figure 7: Batch Insertion.

Recall that in the DFS function of Algorithm *Construct QC-tree*, for a given cell c, its upper bound c' is found based on repeating values in the partition table (step 2 of Function DFS). However, in $\triangle DFS$, instead of recording c' as an upper bound, several steps need to be done:

1. find the upper bound ub of the class containing c in QC
2. if ub cannot be found (c does not exist), then record a *new* temporary class with upper bound c'.
3. else, find $ub \wedge c'$. Three possible results of $ub \wedge c'$ lead to three cases:
Case 1: $ub \wedge c' = ub$. Record ub and its corresponding node for measure *update*.
Case 2: $ub \wedge c' = c'$. That means c' itself is an upper bound. We need to split the class C in QC that ub falls in. Record a *split* class with upper bound c', containing all cells of C that are below c'. The remainder of C forms another class, with ub as its upper bound.
Case 3: Neither of these two situations happen, i.e., c' and ub are incomparable. Then let $c'' = ub \wedge c'$, and record a *new* temporary class with upper bound c''.

The batch insertion algorithm is given in Figure 7.

clsID	Upr Bnd	Lwr Bnd	Lattice child	Status	Old Upr Bnd
0	(*,*,*)	(*,*,*)	-1	Update	(*,*,*)
4	(*,P2,*)	(*,P2,*)	0	Split	(S1,P2,s)
1	(S2,*,f)	(S2,*,*)	0	Split	(S2,P1,f)
7	(S2,*,f)	(*,*,f)	0	Split	(S2,P1,f)
2	(S2,P2,f)	(S2,P2,f)	1	Insert	–
5	(S2,P2,f)	(*,P2,f)	4	Insert	–
3	(S2,P3,f)	(S2,P3,f)	1	Insert	–
6	(S2,P3,f)	(*,P3,*)	0	Insert	–

Figure 8: Temporary Classes Created by ΔDFS.

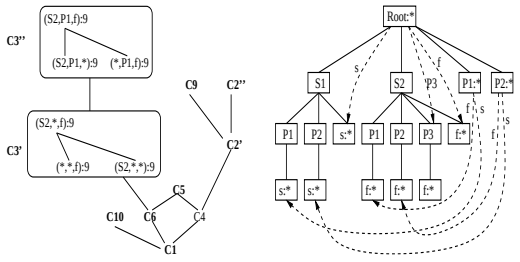


Figure 9: The QC-tree after the updates.

Example 3 (Batch update of a QC-tree). Suppose we want to update the QC-tree w.r.t. the insertion of two tuples $\Delta DB = \{(S2, P2, f), (S2, P3, f)\}$ to the base table. We apply the ΔDFS on ΔDB. Figure 8 shows the temporary classes created by the ΔDFS. Figure 9 shows the updated quotient lattice and QC-tree. ∎

Both tuple insertion and batch insertion involve checking whether a cell c belongs to the original quotient cube, and possibly to a class whose upper bound is given. This amounts to asking a point query using QC-trees. While we discuss algorithms for query answering in Section 4, here it suffices to bear in mind that, even though our query answering algorithms are very efficient, it pays to minimize the number of such "query invocations". From this standpoint, tuple-by-tuple insertion makes repeated invocations of certain queries, so intuitively batch insertion ought to be more efficient. We quantify this intuition empirically in Section 5.

3.3.2 Deletions

In a cover quotient cube, deletion will not create any new classes, but it may cause classes to merge, or cause an outright deletion of a whole class. We assume all the tuples in ΔDB exist in the base table. If not, it is easy to check and remove from ΔDB any tuple that is not in the base table. A frequent operation we need to perform in the deletion algorithm is checking whether the cover set of a cell is empty, and whether the cover sets of a parent-child pair of cells is the same. Owing to the property of cover partitions, we can perform both checks efficiently by simply maintaining a count for each cell.

Since the algorithm for maintenance under deletion is similar to that for insertion, we illustrate the ideas via the following example, and omit the algorithm for brevity.

Example 4 (Deletion). Let QC be a cover quotient cube for the set of tuples $\{(S1, P1, s), (S1, P2, s), (S2, P1, f), (S2, P2, f), (S2, P3, f)\}$. Suppose $\Delta DB = \{(S2, P2, f), (S2, P3, f)\}$.

First, we call ΔDB to compute the classes and upper bounds of ΔDB, and order them in the reverse dictionary order ("*" appears last). We delete the classes (with upper bounds) $(S2, P2, f)$, $(S2, P3, f)$ from the original QC, since their cover set will be empty after the base table update (i.e., their count is zero). We will update classes $(S2, *, f)$, $(*, P2, *)$, but their cover sets do not become empty. So we have to see whether they can be merged. Drill down from the class of $(S2, *, f)$ to the class of $(S2, P1, f)$ (which is in the original quotient cube lattice). $(S2, P1, f)$ has the same cover set (i.e., count) as $(S2, *, f)$, so we will merge the classes of $(S2, *, f)$ and $(S2, P1, f)$. Drill down from the class of $(*, P2, *)$ to the class of $(S1, P2, *)$. $(S1, P2, *)$ falls into the class with upper bound $(S1, P2, s)$ whose count is exact the same as $(*, P2, *)$. So merge $(*, P2, *)$ into $(S1, P2, s)$. During the merge, we need to add a link labelled $P2$ from $(*, *, *)$ to $(S1, P2, s)$. ∎

One of the key properties of our algorithms for insertion and deletion is that they yield the correct QC-tree reflecting the update and do so efficiently.

Theorem 2. (Correctness) Let DB be a base table and T be the QC-tree associated with the cover quotient cube of DB. Let ΔDB be a set of tuples that is inserted into DB. Then the QC-tree produced by our batch insertion algorithm is identical to the QC-tree obtained by running the QC-tree construction algorithm outlined in Section 3.2 on $DB \cup \Delta DB$. A similar statement holds for deletion. ∎

4. QUERY ANSWERING

Fundamentally, queries over a data warehouse can be classified into three basic categories – point queries, range queries, and iceberg queries (or boolean combinations thereof). We propose efficient algorithms to answer various queries using QC-trees.

The key idea for an efficient use of QC-trees in query answering comes from the fact that for every cell c with

a non-empty cover set, the QC-tree is guaranteed to have a path representing the class upper bound of c (see Theorem 1). We can trace this path very fast and the last node, being a class node, will contain the aggregate measure associated with c. But we may not *know* the class upper bound of c, so how to find this path fast?

4.1 Point queries

A point query is given a cell c (called the query cell), find its aggregate value(s). We assume c is presented in the dimension order used to build the QC-tree. Let $c = (*, \ldots, v_1, \ldots, *, \ldots, v_2, *, \ldots, v_k, \ldots, *)$, where the values $v_i \neq$ '$*$' appear in dimensions $\ell_i, i = 1, \ldots, k$. First consider the case where the cover set of c is non-empty so the answer to the point query is not null. Consider the following procedure.

> Start at the root and set the current value to v_1. At any current node N, look for a tree edge or link labeled by the current value, say v_i; if it exists set current value to v_{i+1} and current node to the child of N using the edge found; repeat as long as possible.

LEMMA 2 (POINT QUERY ANSWERING). Let c be as above and let N be any current node such that there is no tree edge or link out of N labeled by the current value v_i. Suppose c has a non-empty cover set. Then there is a dimension $j < \ell_i$ such that: (i) j is the last dimension for which N has a child, and (ii) there is exactly one child of N for dimension j. ∎

The above lemma suggests an efficient method for answering point queries. Follow the iterative procedure above. Whenever we cannot proceed further, check the last dimension j for which the current node N has a child. If $j \geq \ell_i$, c cannot appear in the cube. If $j < \ell_i$, then move to the unique child of N corresponding to the last dimension j and repeat the procedure above. While the algorithmic details will be given shortly, note that this scheme never visits more than one path. And whenever the query has a non-null answer, this path corresponds to the class upper bound of the query cell. Intuitively, this makes the method very efficient, regardless of the size of the base table.

We next give the algorithm and illustrate it on the QC-tree of Figure 4.

EXAMPLE 5 (ILLUSTRATING ALGORITHM 3). Using our running example QC-tree of Figure 4, suppose we want to answer a query $(S2, *, f)$. From the root node, we find there is a child labelled $S2$, node 8. From node 8, we cannot find any tree edges or links labelled f. So we pick the child on the last (in this case, only) dimension, which is $P1$, node 9. From node 9, we *can* find a child labelled f, node 10, and being a class node, it has aggregate value in it. So this value is returned. As another example, consider the query $(S2, *, s)$. From the root node, we find there is a child labelled $S2$, node 8. From node 8, we cannot find any tree edges or links labelled s. so pick the child on the last dimension, which is $P1$, node 9. from node 9, we cannot reach a node labelled s. Then pick the child on the last dimension, which is f, node 10. But f's dimension is *not* later than s's dimension. So query fails, and we return NULL. As a last example, consider the query $(*, P2, *)$. From the root node, we find there is a link labelled $P2$. We have examined all non-* values in the query. But $P2$ has no aggregate value. So we pick the child on the last dimension, which is s. s has aggregate value, so return it. ∎

ALGORITHM 3. *[Point query answering]*
Input: *a point query* $q = (*, .., v_1, ., *, .., v_k, .., *)$ *where* $v_1 \ldots v_k$ *are the only non-* values and QC-tree* T.
Output: *aggregate value(s) of* q.
Method:

```
// process the dimension values in q one by one. find a
route in QC-tree by calling a function searchRoute.
 1.  Initialize newRoot. newRoot=the root node of T
 2.  for each value v_i in q && newRoot ≠ NULL
        // reach for the next node with label v_i
        newRoot = searchRoute(newRoot, v_i);
 3.  if newRoot = NULL, return null;
     else if it has aggregate value(s)
       return its aggregate(s);
     else
       Keep picking the child corresponding to the last
       dimension of the current node, until we reach a
       node with aggregate values, and return them;

Function searchRoute(newRoot, v_i)
// find a route from newRoot to a node labeled v_i:
if newRoot has a child or link pointing to N labeled v_i
  newRoot = N;
else
  Pick the last child N of newRoot labeled by a value
    in the last dimension, say j;
  if (j < the dimension of v_i) call searchRoute(N, v_i);
  else return null;
```

Figure 10: Point query answering.

4.2 Range queries

A range query is of the form $(*, \ldots, v_1, *, \ldots, \{u_{P1}, \ldots, u_{p_r}\}, \ldots, v_i, *, \ldots, \{w_{q_1}, \ldots, w_{q_s}\}, *, \ldots, v_k, *, \ldots)$. We need to find the aggregates associated with each point (cell) falling in the given range. Note that we have chosen to enumerate each range as a set. This way, we can handle both numerical and hierarchical ranges.

An obvious method to answer such queries is to compile the range query into a set of point queries and answer them one by one. We can do much better by dynamically expanding one range at a time, while at once exploring whether each partial cell we have expanded thus far has a chance of being in the cube. E.g., if we determine that based on what we have seen so far cells with value v_i dimension i do not exist, we can prune the search space dramatically.

The algorithm of answering range query is given in Figure 11 and is illustrated in the following example. Experimental evaluation of point and range query answering algorithms compared with those for Dwarf can be found in Section 5.

EXAMPLE 6 (RANGE QUERIES). Suppose we want to answer a range query $(\{S1, S2, S3\}, \{P1, P3\}, f)$ in the QC-tree shown in Figure 4.
(1) We begin from root $(*, *, *)$. We find the $S3$ cannot be reached from (*,*,*), the cells that begin with $S3$ will be pruned immediately.
(2) From node $S1$, only node $P1$ exists. Since $P3$ does

ALGORITHM 4. *[Range query answering]*
Input: *a range query* $q = (*, \ldots, v_1, *, \ldots, \{u_{P1}, \ldots, u_{p_r}\}, \ldots, v_i, *, \ldots, \{w_{q_1}, \ldots, w_{q_s}\}, *, \ldots, v_k, *, \ldots)$ *and QC-tree* T.
Output: *Set of answers.*
Method:

```
      //initialization.
 1.   Let newRoot be the root node of T, results be ∅;
 2.   results = rangeQuery(q, newRoot, 0);
 3.   return results;

Function rangeQuery(q, newRoot, i)
// base case;
if i > the last non-* dimension in q,
  if newRoot = NULL. do nothing;
  if newRoot has aggregate value
    Add its aggregate to results
  else
    Keep picking the child with a value on the last
    dimension until we reach a node with aggregate value,
    add its aggregate to results.
  return;
//recursion;
if in q, i is not a range dimension
    Call searchRoute(newRoot, v_i),
    Let newRoot be the return node
    if newRoot is not NULL
        Call rangeQuery(q, newRoot, i + 1)
else, for each value v_im in the range
    Call searchRoute(newRoot, v_im),
    Let newRoot be the return node
    if newRoot is not NULL
        Call rangeQuery(q, newRoot, i + 1)
```

Figure 11: Range query answering.

not exist, we do not care about this branch any more. Then we try to reach f from $P1$, but fail. Nothing would be returned for the branch.
(3) From node $S2$, again, no node with label $P3$ can be expanded. Only $P1$ can be reached. Continuing to search f is successful. A query result with aggregate value is thus returned. ∎

4.3 Iceberg queries

Iceberg queries are queries that ask for all cells with an aggregate measure greater than a user-specified threshold. These queries may arise either in isolation or in combination with constraints on dimensions, which have a range query flavor in the general case. E.g., one can ask "for all stores in the northeast, and for every day in [4-1-02, 6-1-02] find all cells with SUM > 100,000". First, let us consider pure iceberg queries, which leave dimensions unconstrained. For handling them, we can build an index (e.g., B+tree) on the measure attribute into the QC-tree. Then the pure iceberg query can be answered very quickly: read all (class) node id's from the index and fetch the nodes.

Suppose now we have a constrained iceberg query (as in the example above). We have two choices. (1) Process the range query ignoring the iceberg condition and for each class node fetched, simply verify the iceberg condition. (2) Use the measure attribute index to mark all class nodes satisfying the iceberg condition. Retain only these nodes and their ancestors, to get a subtree of the QC-tree.[3] Process the range query on this subtree. For brevity, we suppress further details.

[3]Since the root is retained, this will be a tree.

5. EMPIRICAL EVALUATION

In this section, we present an extensive empirical evaluation to examine the algorithms we developed in this paper. All experiments are running on a Pentium 4 PC with 512MB main memory and running windows 2000. We used both synthetic and real data sets to evaluate the algorithms. Issues, such as compression ratio, queries performance and incremental maintenance are concerned. We compared QC-tree to QC-table and Dwarf[4] on various algorithms wherever they exist. For the limitation of space, we only present representative results.

5.1 About the datasets

In order to examine the effects of various factors on the performance of the algorithms, we generated synthetic data sets with Zipf distribution. All synthetic data are generated with a Zipf factor 2.

In addition, we also used the real dataset containing weather conditions at various weather stations on land for September 1985 [10]. It contains $1,015,367$ tuples ($\sim$ 27.1MB) and 9 dimensions. The attributes with cardinalities of each dimension 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).

5.2 Compression ratio and time

In this experiment, we explore the compression benefits as well as the construction time of QC-trees. To show the compression ratio, we compare the storage size of QC-trees, QC-table and Dwarf according to the proportion of the original data cube generated by algorithm BUC [5], where QC-table is to store all upper bounds plainly in a relational table.

Figure 12(a)-(c) illustrate the compression ratio versus the major factors data sets, namely the number of tuples in the base table, the cardinality, and the number of dimensions. The results show that Dwarf and QC-table can achieve comparable compression ratio. That is, even a quotient cube itself can compress the data cube effectively. A QC-tree compresses the data cube better in most of the cases. Only in data cubes with very a small number of dimensions or a very low cardinality, Dwarf may be better. In such cases, on average the number of cells per class is very close to 1 and thus not much compression can be achieved.

From Figure 12(a) and (b), one can see that the three compression methods are insensitive to the number of tuples in base table and the cardinality, in terms of compression ratio. That indicates these compression methods are good for various applications. Interestingly, Figure 12(c) shows that the higher the dimensionality, the better the compression ratio. This happens because the data gets sparser when dimensionality increases while the number of tuples remains the same. This illustrates all three methods have the potential to scale well for large number of dimensions. Consistent compression effects can also be observed on real data sets, as shown in

[4]Since the original Dwarf code was unavailable, we implemented it as efficiently as possible.

#Dim	4	5	6	7	8	9
Cube	31.9	111.0	440.3	1,005.9	2,716.4	5,990.2
QC-Tab	11.4	30.4	90.5	130.3	223.0	305.8
QC-Tree	12.5	27.8	67.5	103.4	170.2	241.2
Dwarf	16.4	33.0	91.6	163.7	300.4	423.8

Figure 15: Experiment Result of Storage Size(MB) on Weather Data

Figure 15, where the real size of the data cube and the compressions are given.

As shown in Figure 12, all three methods are scalable on the base table size in terms of construction time. QC-table and QC-tree are consistently better than Dwarf. That can be explained by the fact that (1) a quotient cube is substantially smaller than the complete cube, so constructing QC-tables and QC-trees access less data; and (2) the depth-first search computation of classes and upper bounds is effective and efficient.

5.3 Query answering performance

In this experiment, we compared query answering performance using QC-tree and Dwarf. First, we randomly generated $10,000$ different point queries based on the synthetic data set. Figure 13(a) shows that the increase of cardinality reduces the efficiency of Dwarf. But QC-tree is insensitive to the increase. Then, we generated 1,000 point queries based on the weather data. Figure 13(b) shows the result.

To test the effect of range query, we randomly generated 100 range queries. On the synthetic dataset, a range query contains 1-3 range dimensions and each range dimensions has 30 range values. So the maximum case is that a range query would be equivalent to $27,000$ point queries. In the weather dataset experiment, a range query contains 1-3 range dimensions and each range dimensions has range values exactly the same as the cardinality of that dimension. Figure 13(c) and (d) show the performance.

From these results, one can see that both methods are scalable for query answering, and QC-tree is clearly better. The main reason why we are faster than Dwarf is because: (i) when we query a cell c with n dimensions, Dwarf needs to access exactly n nodes. In the QC-tree, typically fewer than n would be accessed. For a simple example. if we query $(*, P1, *)$, in Dwarf, 3 nodes have to be traversed. In QC-tree, only root node and node $P1$ are visited; (ii) since the size of QC-tree is smaller than dwarf, less I/O is needed in a QC-tree.

5.4 Incremental Maintenance

To test the scalability of the incremental maintenance of QC-trees, we fixed the base table size and varied the size of incremental part. Only the results on insertions into both synthetic and real data sets are reported in Figure 14. The results on deletions are similar. Limited by space, they are omitted.

We compare three methods: (1) recomputing the QC-tree; (2) inserting tuples one by one; and (3) inserting in batch. Clearly, the incremental maintenance algorithms are much better than the reamputation. Moreover, maintenance in batch is more scalable than maintenance tuple by tuple. The main work of our incremental maintenance algorithm is locating upper bounds, which means doing point queries, in the original QC-tree to decide if we need to update, split, or create new classes. The previous experiments have shown that answering point queries in QC-tree is extremely efficient. Therefore, these savings are passed on for the case of incremental maintenance.

6. DISCUSSION

6.1 Related work

Three categories of previous researches are related, namely the foundation of data cubes and OLAP, the implementation and maintenance of data cubes, and query answering and advanced analysis using data cubes.

The data cube operator was firstly proposed by Gray et al. in [8]. The quotient cube notion [14] can be regarded as applying the Galois closure operators and formal concept analysis [6] to the data cube lattice.

Many approaches (e.g., [1, 5, 20, 30]) have been proposed to compute data cubes efficiently from scratch. In general, the algorithms speed up the cube computation by sharing partitions, sorts, or partial sorts for group-bys with common dimensions. In [11], Harinarayan et al. study the problem of choosing views to materialize data cubes under space constraints. [5] proposes computing iceberg cube and use BUC to handle monotonic constraints. Methods to compress data cubes are studied in [24, 25, 27]. Moreover, [3, 4, 26] investigated various approximation methods for data cubes.

How to implement and index data cubes efficiently is a critical problem. In [21, 25], two important methods, Cubetree and Dwarf, are proposed. The general idea is to explore the sharing among dimension values of aggregate cells, so that the required storage space can be reduced substantially, while the performance of query-answering still retains.

A data cube may need to be updated timely to reflect the changes of the base table. The possible updates include insertions and deletions of data records, and changes of the schema (e.g., changes of the domains and hierarchies in some dimensions). [9, 17, 18, 19] study the maintenance of views in a data warehouse. In [12, 13], Hurtado et al. study the problem of data cube maintenance under dimension updates. Moreover, In [28, 29], the problem of temporal view maintenance is explored.

Various methods have been proposed to answer aggregate queries efficiently. Some typical examples include [7, 15, 16]. To facilitate the query answering, various indexes have been proposed. In [22], Sarawagi provides an excellent survey on related methods.

Some recent studies aim at supporting advanced analysis in data warehouses and data cubes. [23, 2] are some interesting examples.

6.2 Comparison with Dwarf

Recently, a compact data structure, Dwarf, was proposed in [25] to store a data cube efficiently. The main strengths of Dwarf are (a) sharing of prefixes among cells is exploited to compress the cube substantially and

(b) rows from the same group-by view are clustered together to speed up query answering. In this section, we outline the main differences between Dwarf and QC-trees.

First, the two data structure store different data. Dwarf stores the complete data cube (albeit in a highly compressed form) while QC-tree only stores the upper bounds of a quotient cube (again in compressed form). Since a quotient cube is often much smaller than the complete cube, a QC-tree is expected more efficient than Dwarf. This is also verified by our empirical evaluations on both synthetic and real data sets. A QC-tree consistently achieves better compression ratio than Dwarf.

Second, the query answering mechanisms in Dwarf and QC-tree are different. In Dwarf, the query answering requires exact matches of the query points and the cells stored. In QC-tree, once the upper bound of the query point is identified, the query is answered. Since the QC-tree structure only records the upper bounds, its average fan-out (i.e., average number of children per node) is expected to be substantially smaller than that of Dwarf. Thus, the query answering in QC-tree is faster. The advantage is even clearer when the data cube is larger (i.e., larger cardinality, more dimensions and more tuples.)

Third, the QC-tree supports more complicated OLAP operations efficiently. For example, since the directed links for drill-down between classes are stored explicitly in a QC-tree, the user can directly browse the classes of quotient cube efficiently. Moreover, since the QC-tree has efficient incremental maintenance algorithms, it is efficient to populate a hypothetical sub-cube and answer the "what-if" queries efficiently.

7. CONCLUSIONS

The technique of quotient cube as a compact summary structure that preserves the essential semantics of a data cube was recently proposed in [14]. However, a straightforward implementation of quotient cube is not as efficient and compact as it can be. Besides, issues of efficient query answering using quotient cubes and their incremental maintenance were not addressed. In this paper, we proposed using the cover quotient cube as it caters for all aggregate functions, and showed some nice properties for them. We proposed a very compact data structure called QC-tree, which compresses the size of a quotient cube even further. It is elegant and lean in that the only information it keeps on classes are their upper bound and measure(s). We provided fast algorithms for query answering and incremental maintenance. Our empirical evaluations show that under many circumstances, QC-trees lead to a better compression and construction time than Dwarf. Our query answering algorithms really outperform Dwarf in most cases. For view maintenance, our incremental algorithms are typically an order of magnitude, or more, faster than recomputation. We are currently developing a prototype system building on our research results.

8. REFERENCES

[1] S. Agarwal et al. On the computation of multidimensional aggregates. In VLDB'96.

[2] A. Balmin et al. Hypothetical queries in an olap environment. In VLDB'00.

[3] D. Barbara and M. Sullivan. Quasi-cubes: Exploiting approximation in multidimensional databases. SIGMOD Record, 26:12–17, 1997.

[4] D. Barbara and X. Wu. Using loglinear models to compress datacube. In WAIM'00.

[5] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In SIGMOD'99.

[6] C. Carpineto and G. Romano:. Galois: An order-theoretic approach to conceptual clustering. In ICML'93.

[7] S. Cohen et al. Rewriting aggregate queries using views. In PODS'99.

[8] J. Gray et al. Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In ICDE'96.

[9] A. Gupta et al. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, Eds., SIGMOD'93.

[10] 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.

[11] V. Harinarayan et al. Implementing data cubes efficiently. In SIGMOD'96.

[12] C.A. Hurtado et al. Maintaining data cubes under dimension updates. In ICDE'99.

[13] C.A. Hurtado et al. Updating olap dimensions. In DOLAP'99.

[14] L.V.S. Lakshmanan et al. Quotient cube: How to summarize the semantics of a data cube. In VLDB'02.

[15] A.Y. Levy et al. Answering queries using views. In PODS'95.

[16] A.O. Mendelzon and A.A. Vaisman. Temporal queries in olap. In VLDB'00.

[17] I.S. Mumick et al. Maintenance of data cubes and summary tables in a warehouse. In Joan Peckham, editor, SIGMOD'97.

[18] D. Quass et al. Making views self-maintainable for data warehousing. In Proc. 1996 Int. Conf. Parallel and Distributed Information Systems.

[19] D. Quass and J. Widom. On-line warehouse view maintenance. In SIGMOD'97.

[20] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In VLDB'97.

[21] N. Roussopoulos et al. Cubetree: Organization of and bulk updates on the data cube. In SIGMOD'97.

[22] S. Sarawagi. Indexing OLAP data. IEEE Data Eng. Bulletin, 20:36–43, 1997.

[23] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional OLAP data. In VLDB'01.

[24] J. Shanmugasundaram et al. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In KDD'99.

[25] Y. Sismanis et al. Dwarf: Shrinking the petacube. In SIGMOD'02.

[26] J. S. Vitter et al. Data cube approximation and historgrams via wavelets. In CIKM'98.

[27] W. Wang et al. Condensed cube: An effective approach to reducing data cube size. In ICDE'02.

[28] J. Yang and J. Widom. Maintaining temporal views over non-temporal information sources for data warehousing. In EDBT'98.

[29] J. Yang and J. Widom. Temporal view self-maintenance. In EDBT'00.

[30] Y. Zhao et al. An array-based algorithm for simultaneous multidimensional aggregates. In SIGMOD'97.

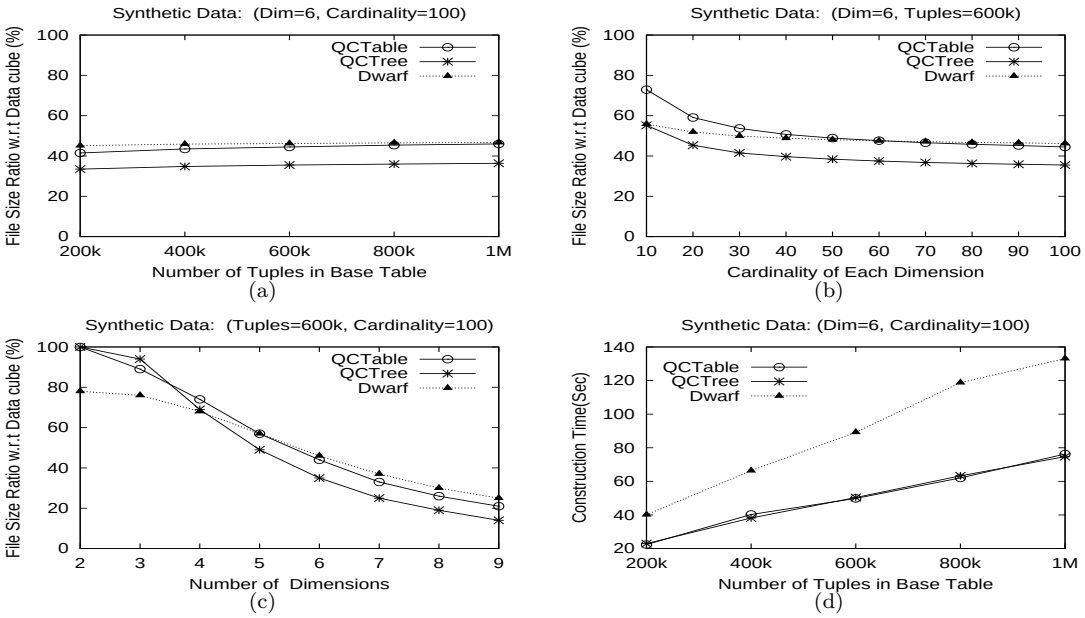


Figure 12: Evaluating compression ratio and running time on synthetic data

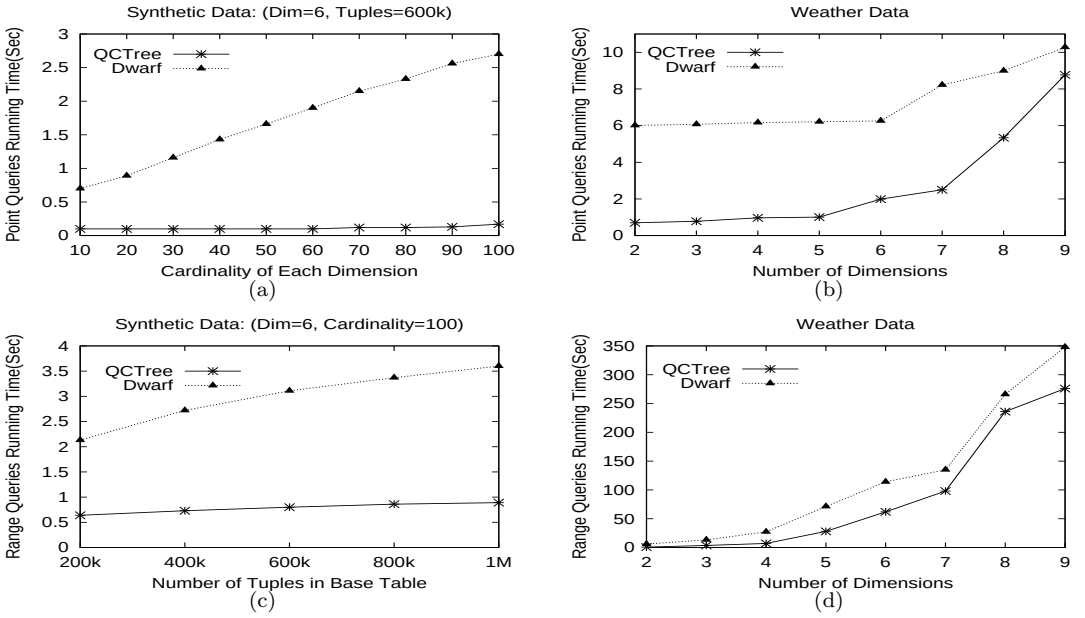


Figure 13: Query performance

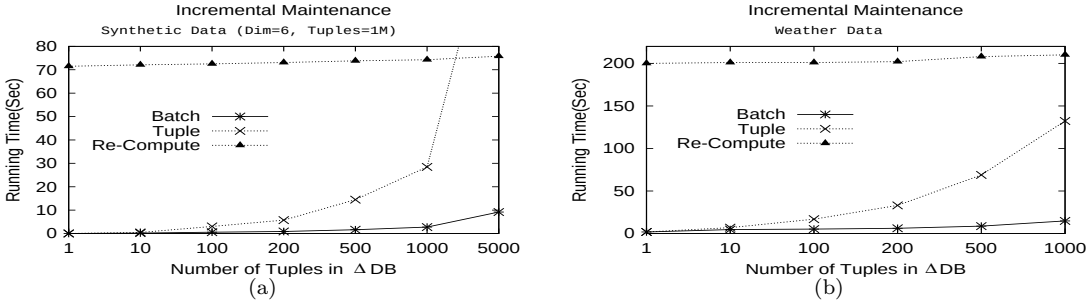


Figure 14: Performance of Batch Incremental Maintenance