Quotient Cube and QC-Tree: Efficient Summarizations for Semantic OLAP

by

Yan Zhao

B.Eng., Shanghai Jiao Tong University, China, 1993

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

September 2003

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, BC
Canada V6T 1Z4

Date: Oct. 8th. 2003

Abstract

Most data cube compression approaches focused on reducing the cube size but overlooked preserving the semantics of data cubes. A technique called quotient cube was proposed recently as a summary structure for a data cube that retains its semantics. It can be constructed very efficiently and it leads to a significant reduction in the cube size. However, many issues have not been addressed regarding the application of quotient cubes. For instance, efficient storage and index structures were not provided; No specific algorithm for query answering was given; Incremental maintenance against updates was not discussed. In this thesis work, we are aiming at developing proper solutions to the above problems. Firstly, we introduce the idea of sorted list to index quotient cubes and provide associated query answering algorithms. Secondly, since a tabular representation with additional index is neither compact nor efficient to maintain, we propose a more efficient data structure called QC-tree to store and index quotient cubes. We present a depth-first search based algorithm for constructing QC-trees directly from base tables. Thirdly, we devise efficient algorithms for answering various queries and incrementally maintaining QC-trees against base table updates. An extensive experimental study illustrates the space and time savings achieved by our algorithms. Finally, we implement a quotient cube based data warehouse system to demonstrate our research accomplishments.

Contents

List of Tables

List of Figures

Acknowledgments

First and foremost, I am very grateful to Dr. Laks V.S. Lakshmanan, my supervisor, for his guidance and great support throughout this thesis work. Sincere appreciation goes to Dr. Jian Pei. His inspiration and support is the key to the success of this work. Deep thanks to Dr. George K. Tsiknis for spending time to read and review my work. I would like to acknowledge Dr. Raymond T. Ng and Dr. Alan Wagner for being my thesis committee members and for their advice and comments. Thanks to all the members in database lab, especially Zhimin Chen, Pingdong Ai, and Zheng Zhao for their invaluable help and suggestion.

YAN ZHAO

The University of British Columbia

September 2003

To my family

Chapter 1

Introduction

1.1 OLAP and Data Cubes

A data warehouse is a "subject-oriented, integrated, time-varying, non-volatile collection of data that is used primarily in organizational decision making" [1]. It becomes one of the essential elements of decision support and hence attracts more attentions in both industry and research communities. Powerful analysis tools are well developed, and consequently reinforce the prevalent trend toward data warehousing systems. On-line analytical processing (OLAP) systems, which typically dominated by stylized queries that involve group-by and aggregates operators, are representative applications among these tools.

1.1.1 Data Cubes

The data in OLAP systems is usually modelled as a multidimensional structure. Table 1.1 is a ***base table*** of a simple sales data warehouse. The attributes Time, Product and Location are ***dimensions*** identifying and categorizing the data. The

attribute Sales is a numerical ***measure*** that people wish to analyze. Dimensions usually associate with ***hierarchies***, which organize data according to ***levels***. For instance, a Time dimension might have Day, Month, Quarter and Year as the hierarchy levels.

Location	Product	Time	Sales
Vancouver (Van)	Book (b)	99.1.1 (d1)	9
Vancouver (Van)	Food (f)	99.2.1 (d2)	3
Toronto (Tor)	Book (b)	99.1.1 (d2)	6

Table 1.1: Base Table for a Sales Data Warehouse.

Location	Product	Time	Avg(Sales)
Van	b	d1	9
Van	f	d2	3
Tor	b	d2	6
Van	b	*	9
Van	*	d1	9
*	b	$d1$	9
...	...	...	...
Tor	*	*	6
*	f	*	3
*	*	d1	9
*	*	*	6

Table 1.2: Cells in a Data Cube

Data cube was introduced by Gray et al. in [9]. It is a generalization of the group-by operator over all possible combinations of dimensions with various granularity aggregates. Each group-by corresponds to a set of ***cells***, described as tuples over the group-by dimensions. The cells in the data cube of Table 1.1 is shown in Table 1.2. Here, the symbol "*" in a dimension means that the dimension is generalized such that it matches any value in its domain.

Roll-up and ***drill-down*** are two basic semantic relations of a data cube. A cell with higher-level aggregates can drill down to a cell with lower-level aggregates, e.g., cell $(Van, *, d1)$ drills down to cell $(Van, b, d1)$. A cell with lower-level aggregates

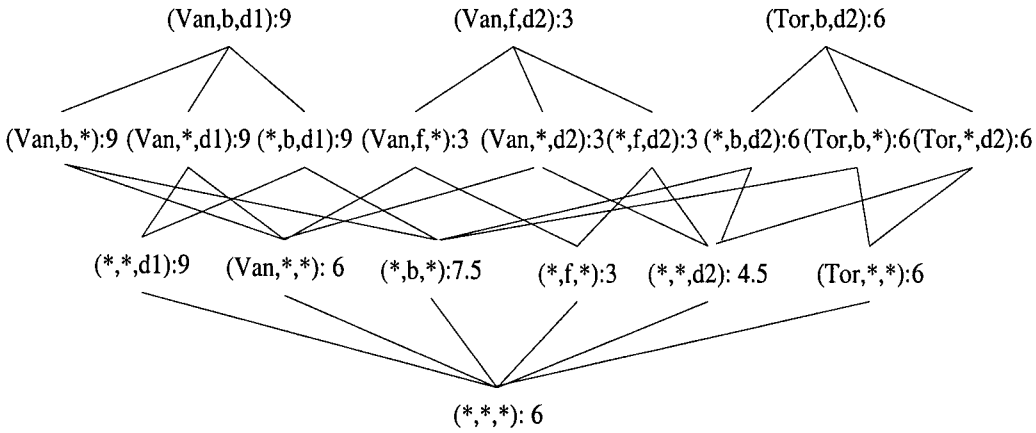


Figure 1.1: Data Cube Lattice

can roll up to a cell with higher-level aggregates, e.g., cell $(Tor, b, *)$ rolls up to cell $(*, b, *)$. A cube's cells together with the roll-up/drill-down relations form a lattice structure. Figure 1.1 shows the ***cube lattice*** for the data cube presented in Table 1.2.

Definition 1.1 (Ancestor-descendant relation). In a data cube lattice, let c and d be two cells such that $c \neq d$. c is an ancestor of d and d is a descendant of c, $c \succ d$, if, in each dimension D such that $d.D \neq *, c.D = d.D$

1.1.2 Basic OLAP Queries

A typical data cube exploration involves three important categories of queries. The most common queries are point queries, which is usually used in roll-up/drill-down operations. Range queries apply aggregate operations over selections which are specified by dimension value ranges. Iceberg queries enable users imposing some constraints on certain dimensions and/or on (aggregate) measure attributes while querying a data cube. The following sections discuss these three types of queries

and provide some illustrative examples.

Point Queries

A point query looks for the aggregate measure value in a specific cell in the cube. Consider the data cube in Figure 1.1 as an example. The following point query finds the total amount of sales in the location Van in the day $d1$. The result of that query is the cell $(Van, *, d1)$.

```
SELECT location, time, SUM(Amount)
FROM sales
WHERE (location=Van) AND (time=d1)
```

Point queries are essential for data cube. Any other queries can be rewritten as a set/series of point queries.

Range Queries

In some situations, we are interested in a group of points in the data cube. A range query looks for a list of cells whose dimension values are in a specific range. For example, the following range query finds the list of sales in Van and Tor of product b(book).

```
SELECT location, product, SUM(Amount)
FROM sales
WHERE (Product=b) AND ((location=Van) OR (location=Tor))
GROUP BY location
```

The answer to the above query should contain two tuples: $(Van, b, *)$ and $(Tor, b, *)$. A range query can have multiple dimensions with range values. For example, the following query looks for sales of product b and f in the location Van and Tor.

```
SELECT location, product, SUM(Amount)
FROM sales
WHERE ((location=Van) OR (location=Tor)) AND ((Product=b) OR (Product=f))
GROUP BY location, product
```

Clearly, a range query Q can be rewritten to a set of point queries $P(Q)$. Given a schema and the cardinality of every dimension (i.e., values in every dimension), $P(Q)$ can be uniquely determined from Q. That is, given a range query Q, we can derive $P(Q)$ directly without looking at the data records in the base table.

Iceberg Queries

Both point queries and range queries retrieve data from the dimension values to the measure values. In some applications, users may be interested in queries from measure values to dimension values. That motivates the iceberg queries. An iceberg query returns all cells in the cube whose aggregate values satisfy a user-specified threshold. For example, the following iceberg query looks for all locations, products and time and their combinations such that the corresponding average sales are over 6 dollars. The HAVING clause in the iceberg query is also called an iceberg condition.

```
SELECT location, product, time, AVG(Amount)
FROM sales
CUBE BY location, product, time
HAVING AVG(Amount) >= 6
```

Many queries on a data cube are combinations of the above three kinds of queries. For example, one may compose a query for all the locations whose total sales are over 2000 dollars. This is a combination of an iceberg query and a set of range queries.

1.2 Problems and Contributions

An important feature of OLAP systems is its ability to efficiently answer decision support queries. To improve query performance, an optimized approach is to materialize the data cube instead of computing it on the fly. The inherent problem with this approach is that the huge size of most data cubes limits the efficiency and applicability. The space and time costs of the materialization are extremely high. For example, 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! Reducing the size of data cubes thus becomes one of the essential aspects for achieving effective OLAP services.

Another essential aspect is the critical semantics among aggregate cells in a data cube. A basic semantics as we mentioned before is given by the roll-up/drill-down relations which can be captured by the lattice structure. The other semantics is defined in term of the cell similarities, e.g., cells $(Van, b, d1)$ and $(Van, b, *)$ register the same average sales. A typical navigation of data cubes entails this semantics.

Previous work on compressing the cube should help to cut down the cube size. However, almost all approaches proposed before are *syntactic*, in the sense that even the roll-up/drill-down relations are lost in the compressed representation.

Therefore, seeking a lossless, concise and effective summary of a cube that preserves the inherent semantics and lattice structure is our main goal. A novel conceptual structure called quotient cube was proposed in [32] as a 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 classes while keeping the cube's roll-up/drill-down relations and lattice structure.

While the quotient cube is an interesting idea, the study in [32] leaves many questions unanswered. In particular, the following questions are still open:

- What kind of quotient cubes are suitable for a general purpose data warehouse given that quotient cubes can be constructed in more than one way?
- How to store and index data in quotient cubes and how to use them to answer various queries?
- How to construct quotient cubes as well as the index structures efficiently?
- How to incrementally maintain a quotient cube-based data warehouse?
- How to implement a practical quotient cube-based data warehouse system?

Motivated by providing effective solutions to these interesting questions, this thesis work makes the following contributions:

- We show that quotient cube w.r.t cover partition can be used to construct a quotient cube-based data warehouse. It can support a variety of OLAP queries effectively and compress the data cube substantially.
- We introduce sorted list as a feasible structure to index the tabular representation of quotient cubes and propose efficient algorithms for query answering.
- We devise QC-tree, an effective data structure to compress and index quotient cubes. We give an efficient algorithm to construct a QC-tree directly from the base table. We develop efficient algorithms to answer point queries, range queries, and iceberg queries.
- We develop scalable algorithms to incrementally maintain a QC-tree, and establish their correctness.

- Based on our research results, we develop a prototype data warehouse system, demonstrating the efficiency and the practicability of quotient cubes and QC-trees.

1.3 Outline

The rest of the thesis is organized as the following:

- In Chapter 2, we describe research background and related work.
- Chapter 3 focuses on quotient cubes. We first review the main concepts and important properties of quotient cubes. We next propose an efficient structure to index the tabular representation of quotient cubes, following with the associated query answering algorithms.
- We develop QC-tree, a novel storage and index structure for quotient cubes in Chapter 4. We discuss an efficient approach to compute a QC-tree directly from the base table. Algorithms of various query answering and incremental maintenance are also proposed in this chapter.
- Experimental analysis is presented in Chapter 5. Our evaluations demonstrate the achievements of the substantial compression of quotient cubes and QC-trees and efficiency of the algorithms.
- Chapter 6 describes a prototype data warehouse system entirely based on our research results of quotient cubes and QC-trees.
- Chapter 7 summarizes this thesis work and indicates some possible directions of future work.

Chapter 2

Background and Related work

Three categories of previous research are related to this thesis work: 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 [9]. The quotient cube notation [32] can be regarded as applying the Galois closure operators and formal concept analysis [7] to the data cube lattice. Many approaches (e.g., [2, 6, 19, 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. [6] proposes computing iceberg cube and use BUC to handle monotonic constraints. Methods to compress data cubes are studied in [23, 24, 27]. Moreover, [4, 5, 26] investigat various approximation methods for data cubes.

Undoubtedly, a data cube has to be updated timely to reflect the changes of the base table. The possible updates include insertions and deletions of data

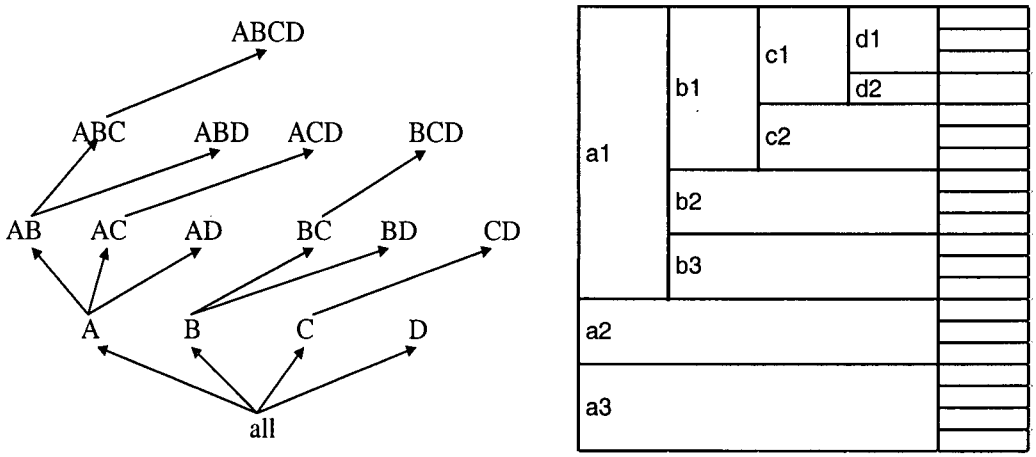
records, and changes of the schema (e.g., changes of the domains and hierarchies in some dimensions). [10, 16, 17, 18] study the maintenance of views in a data warehouse. In [12, 13], Hurtado et al. study the problem of data cube maintenance under dimension updates. Moreover, in [28, 29], the problem of temporal view maintenance is explored.

How to implement and index data cubes efficiently is a critical problem. Various methods have been proposed to answer aggregate queries efficiently. Some typical examples include [8, 14, 15, 25, 20, 24]. [21] provides an excellent survey on related methods. Some recent studies aim at supporting advanced analysis in data warehouses and data cubes. [22, 3] are two interesting examples.

It is worthwhile to further discuss several previous works closely related to this thesis work. [6] proposed an efficient bottom-up method to compute sparse and iceberg cubes which inspires our depth-first search algorithm. Works on lossless compression of data cubes such as [24, 27] are certainly of clear relevance to us. However, the distinction between our work and these previous approaches is significant. Our goal is not only to compress the cube size, but most importantly, to seek an exact and concise summarization of a data cube that preserves the semantics and lattice structure.

2.1 Bottom-Up Computation of Sparse Cubes

BUC [6] builds the data cube bottom-up; i.e. it computes the cube from a group-by on a single dimension, then on a pair of dimensions, then on three dimensions, and so on. In other words, it proceeds from the bottom of the cube lattice, partitions the corresponding base table and explores the cells way up. Suppose we have a 4-dimension base table. Let A, B, C, and D represent the dimension name and let



(a) BUC Processing Tree (b) BUC Partition

Figure 2.1: BUC Algorithm [6]

$a1, a2, ...$ denote the dimension value. The processing tree and partition mechanism of the BUC algorithm is shown in Figure 2.1[6].

Our depth-first search algorithms in computing quotient cubes and QC-trees follow the similar processing steps as BUC.

2.2 Condensed Cube

The concept of condensed cube was devised by Wang et al. in [27]. It is based on two key ideas, namely "Base Single Tuple" compression and "projected single tuple" compression, which come from the observation that a number of partitions(cells) contain only one tuple in the base table. The combination of the two compression ideas is similar to the cover equivalence. It is perhaps the closest in spirit to our work.

The algorithms proposed for computing condensed cubes in [27] include an exhaustive one and an efficient heuristic but suboptimal one. Although it was men-

tioned that condensed cubes could be used to answer queries efficiently, no specific query answering algorithms were provided in [27].

2.3 Dwarf

The Dwarf structure was proposed in [24] to store a data cube efficiently. By adopting two key approaches, namely prefix sharing and suffix coalescing, Dwarf reduces the size of data cubes impressively.

There are some essential differences between Dwarf and our approach. Dwarf is a syntactic compression method, without taking any semantics among cells into account. Quotient cubes and QC-trees are fundamentally semantic, preserving the cube lattice structures. Dwarf stores all cube cells while Quotient cubes and QC-trees use classes(upper bounds) as the storage base. These essential differences lead to distinguished mechanisms in the algorithms of construction, query answering and incremental maintenance.

Chapter 3

Quotient Cube

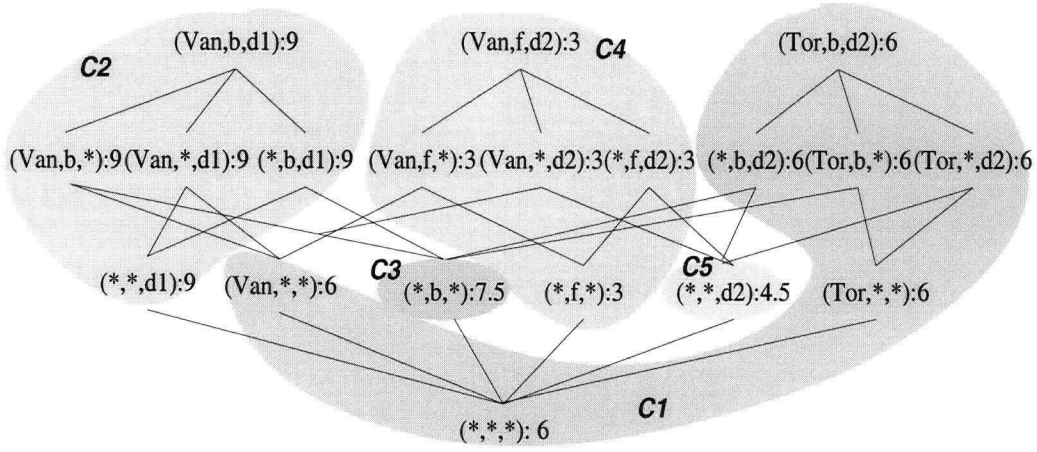
A quotient cube is a summary structure for a data cube that preserves its semantics. It can be constructed very efficiently and achieves a significant reduction in the cube size.

In this chapter, we first review the concepts and important properties of quotient cubes briefly and then discuss a straightforward representation of quotient cubes, called QC-table. Then we focus on the issues, which were not addressed in [32], such as how to store and index quotient cubes and how to answer basic queries using quotient cubes.

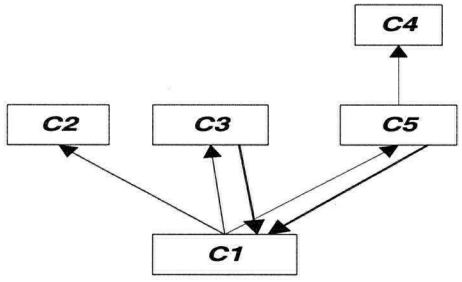
3.1 Cube Lattice Partitions

The key idea behind a quotient cube is to create a summary structure by carefully partitioning the set of cells of a cube into equivalent classes while keeping the cube's roll-up and drill-down semantics and lattice structure.

The first attempt to partition cells came from the observation that many cells have same aggregate values. However, a partition of the cube that is based solely



(a) Partition Based on Aggregate Value AVG



(b) Quotient Lattice

Figure 3.1: Bad Partition $Cube_{Sales}$

on the aggregate measure value does not necessarily preserve the original cube semantics.

Figure 3.1 shows the problem. Suppose we partition the cells solely based on equality of AVG values. The partition result cannot yield a lattice. We can drill down from class cell (*,*,*) in C_1 to cell (*,b,*) in class C_3, and then drill down from cell (*, b, *) in class C_3 back to cell (Tor, b,*) in class C_1. We also can drill down from class C_1 to C_5 and then drill down from class C_5 to C_1. What went wrong here is that cells $(*,*,*)$ and (Tor,b,$*$) belong to the same class C_1, but not $(*, b, *)$

that lies in between these two cells.

Let $\prec$ be the cube lattice partial order. A class that contains cells c, c' but not c'', where c'' is some cell such that $c \prec c'' \prec c'$, is said to have a ***hole***. So what is the generic partition that leads to classes with no holes?

Definition 3.1 (Convex Partitions). [32] Let $\mathcal{P}$ be a partition and let $C \in \mathcal{P}$ be a class. We say C is convex provided, whenever C contains cells c and c' with $c \succeq c'$, then every intermediate cell c'', such that $c \succeq c'' \succeq c'$, necessarily belongs to C. We say $\mathcal{P}$ is ***convex*** provided all its classes are.

In addition to the requirement of being convex, it is also required that classes are ***connected***: for every two cells c_1 and c_2 in a class, we can start from c_1 and finally arrive at c_2 by a series of roll-up/drill-down operations, and all the intermediate cells are also in the same class. A partition is connected provided all its classes are.

Definition 3.2 (Connected Partition). [32] Let f be any aggregate function. Then we define the equivalence $\equiv_f$ as the reflexive transitive closure of the following relation R: for cells c, c' in the cube lattice, $c \; R \; c'$ holds iff: (i) the f-value at c and c' is the same, and (ii) c is either a parent or a child of c'.

Various partitions can be defined for various aggregate functions. We note that many aggregate functions of practical interest, such as MIN, MAX, COUNT, SUM[1], and TOP-k, are ***monotone***.

[1]When the domain of the measure attribute does not contain both positive and negative values

Theorem 3.1 (Monotone Aggregate Functions). *[32]: Let f be a monotone aggregate function and ($\mathcal{L}$, $\preceq$) a cube lattice. Then there is a unique maximally convex and connected partition on $\mathcal{L}$.*

For monotone aggregate functions, forming the partition using Definition 3.2 is guaranteed to produce a unique convex and connected partition on the cube lattice. Semantically, this means there is a unique way to partition the cube lattice maximally while preserving the cube semantics.

A fundamental partition of cells without reference to any particular aggregate function, is the one based on the so-called notion of "cover".

Definition 3.3 (Cover). [32]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, denoted as $cov(c)$.

In Table 1.1, the first two tuples both can roll up to cell $(Van, *, *)$, since they both agree with this cell on every dimension where its value is not "$*$". So, the cover set of $(Van, *, *)$ is $\{(Van, b, d1), (Van, f, d2)\}$.

Two cells c and c' are cover equivalent, $c \equiv_{cov} c'$, whenever their cover sets are the same, i.e. $cov(c) = cov(c')$. E.g., in Figures 3.2(a), the cells $(*, *, d1)$, $(Van, b, *)$, $(Van, *, d1)$, $(*, b, d1)$ and $(Van, b, d1)$ have identical cover sets and are cover equivalent.

Lemma 3.1. *[32] 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 cover equivalence $\equiv_{cov}$ coincides with the equivalence $\equiv_{count}$. Thus the way to maximally cover partition the cube lattice while preserving the cube semantics is unique.

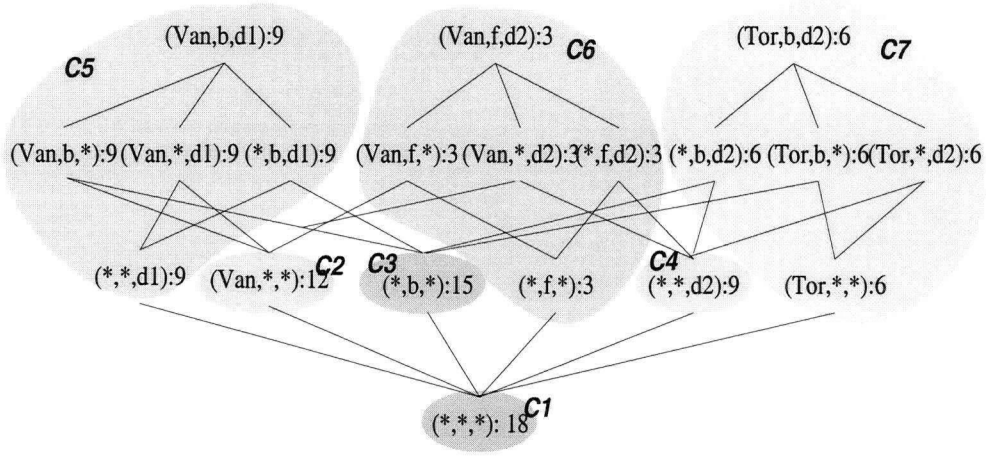
Definition 3.4 (Quotient Cube). [32] For a cube lattice($\mathcal{L}$, $\preceq$), a ***quotient cube lattice*** is obtained by a convex and connected partition $\mathcal{P}$, denoted ($\mathcal{LP}$, $\preceq$) , such that the elements of $\mathcal{LP}$ are the classes of cells in $\mathcal{L}$ w.r.t $\mathcal{P}$. For two classes C and D in the quotient lattice, $C \succeq D$ exactly when $\exists c \in C, \exists d \in D$, such that $c \succeq d$ w.r.t $\mathcal{L}$.

The quotient cube of a data cube w.r.t. the cover partition is called ***cover quotient cube***. Figure 3.2(b) shows the cover quotient cube of the cube lattice in Figure 3.2(a).

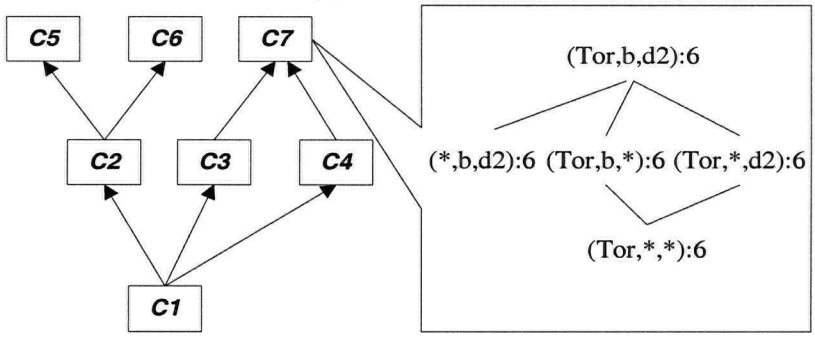
Cover quotient cubes can be served for constructing a general purpose data warehouse regardless of the aggregate functions used. We mainly focus on cover quotient cubes in the rest of the discussion, unless otherwise specified.

3.2 Computing Quotient Cubes

An efficient algorithms for computing quotient cubes was given in [32] and is presented in Figure 3.3. The main part of the algorithm is a depth-first search, which combines the cube computation with base table partitions and equivalent cells determinations, accomplished by first determining class upper bounds and "jumping" to them. In general, the methods of determining class upper bounds depend on the aggregate functions used in the cube. However, in a cover quotient cube it can be computed without reference to aggregate functions.



(a) Cover Partition



(b) Quotient Lattice

Figure 3.2: Quotient Cube w.r.t. Cover Partition

Suppose we are at a cell c. Let B_c be the partition of base table that matches the conditions of cell c. How can we determine the upper bounds ub_c of the class that c belongs to? ub_c agrees with c on every dimension where the value of c is not "*". For any dimension i where the value of c is "*", suppose a fixed value say x_i, appears in dimension i of all tuples in the partition B_c. Then the value of ub_c on every such dimension i is the repeating value x_i of that dimension. For example, if the base table contains just the tuples (a_2, b_1, c_1, d_1), (a_2, b_1, c_2, d_1), and (a_1, b_2, c_2, d_2). Then from the cell $c = (a_2, *, *, *)$, we can "jump" to the upper

bound $(a_2, b_1, *, d_1)$ since the value b_1 appears in dimension b of every tuple and value d_1 appears in dimension d of every tuple in the partition B_c.

The depth-first processing tree is very similar to the BUC [6] algorithm. Note that, because of the "jumping" part, cells in between bounds are never visited in the computation. Compared to BUC, considerable savings can be achieved.

Algorithm 3.1. [32] [Computing Quotient Cubes]
Input: Base table B .
Output: Cover Quotient Cube of B.
Method:
1. $b = (*, \ldots, *)$;
2. call $DFS(b, B, 0)$;
3. merge temp classes sharing common upper bounds.
4. output classes, and their bounds, but only output true lower bounds, by removing lower bounds that have descendants in the merged class;

Function $DFS(c, B_c, k, pID)$
// c is a cell and B_c is the corresponding partition of the base table;
1. Compute aggregate of cell c, collect dimension-value statistics info for count sort;
2. Compute the upper bound ub of the class c, by ''jumping'' to the appropriate upper bounds;
3. Record a temp class with lower bound c,upper bound ub;
4. if there is some $j < k$ s.t. $c[j] = all$ and $ub[j] \neq all$ return; //such a bound has been examined before
5. else for each $k < j < n$ s.t. $ub[j] = all$ do
   for each value x in dimension j of base table
   let $ub[j] = x$;
   form B_d;
   if B_d is not empty, call $DFS(ub, B_d, j)$;
6. for each $k < j < n$ s.t. $c[j] = all$ do
   partition B_c according to values in dimension j;
   let $d = c$;
   for each value x in dimension j such that it occurs in B_c;
   let $d[j] = x$;
   if d is not covered by ub then call $DFS(d, B_d, j)$
7. return;

Figure 3.3: Algorithm - Computing Quotient Cubes

3.3 QC-Table

We now discuss the structures for storing as well as indexing quotient cubes and the methods for answering various queries.

3.3.1 Tabular Representation

A direct approach to store a quotient cube is the tabular representation called QC-table. As illustrated in Table 3.1, a quotient cube can be represented by storing, for each class, its upper bound(s), and lower bound(s). For example, class C_5 has upper bound $(Van, b, d1)$, lower bound $(Van, b, *)$ and $(*, *, d1)$. This representation shows that a cell belongs to a class if and only if it lies between its upper bound and one of its lower bounds.

Class	Upper Bound	Lower Bound	Agg
1	(*,*,*)	(*,*,*)	18
2	(Van,*,*)	(Van,*,*)	12
3	(*,b,*)	(*,b,*)	15
4	(*,*,d2)	(*,*,d2)	9
5	(Van,b,d1)	(Van,b,*)(*,*,d1)	9
6	(Van,f,d2)	(Van,*,d2)(*,f,*)	3
7	(Tor,b,d2)	(Tor,*,*),(*,b,d2)	6

Table 3.1: QC-Table

3.3.2 Sorted Lists

In this section, we propose a sorted list method to index QC-tables with respect to cover partition. We first present the structure and some properties of sorted lists, and then propose query answering algorithms for basic queries.

The Structure of Sorted Lists

The ancestor-descendant relation is a partial order in the set of upper bounds. To obtain a global order for the index on classes, which retains the ancestor-descendant relation, a ***degree*** of a cell c can be defined as the number of dimensions that c has value "*", denoted as $dgr(c) = |\{D|c.D = *\}|$. Then, we can sort all upper bounds of a quotient cube in a degree ascending order L, where bounds with same degree can be accommodated arbitrarily. We denote $c_1 \prec_L c_2$ for two bounds c_1 and c_2 if c_1 is before c_2 in L. In words, c_1 has more "*" then c_2.

Lemma 3.2. *Let L be a degree ascending order over a set of cells. $c_1 \prec_L c_2$ holds for cells c_1 and c_2 if $c_1 \prec c_2$.*

Proof: Since $c_1 \prec c_2$, c_2 can roll up to c_1. So all dimensions D such that $c_1.D \neq c_2.D$, $c_1.D = *$. Thus, c_1 has more * than c_2. $c_1 \prec_L c_2$ holds. ■

Given a degree ascending order L over the set of upper bounds of a cover quotient cube, we can assign bound-id to upper bounds as follows. For each upper bound ub, its bound-id $ub.bid = (m + 1)$, where m is the number of upper bounds before ub in L. Since each class has a unique upper bound in a cover quotient cube, the class-id of a class can be represented exactly by the bound-id.

Definition 3.5 (Sorted List). A ***sorted list*** $L(d.v)$ for each dimension d and value v (where $v \neq *$) is a list of bound-ids of all upper bounds having value v in dimension d in the degree descending order L.

In summary, a quotient cube can be stored and indexed using

1. a degree ascending order L

2. a class table QC-table

3. a set of linked lists $L(d.v)$, where v is a value of dimension d.

Dimension Value	Sorted List $L(v)$	Bitmap Vector $B(v)$
Van	2-5-6	0100110
Tor	7	0000001
b	3-5-7	0010101
f	7	0000010
d1	5	0100000
d2	4-6-7	0001011

Table 3.2: The Sorted Lists and Bitmap Vectors of a Quotient Cube

Example 3.1 (The Sorted List Structure). The quotient cube in Figure 3.2 can be indexed as follows.

- A degree ascending order on upper bounds.

  $L = (*,*,*), (Van,*,*), (*,b,*), (*,*,d2), (Van,b,d1), (Van,f,d2), (Tor,b,d2)$

- A class table as shown in Table 3.1

- Sorted lists as shown in Table 3.2 ■

Let n be the number of dimensions, *card* be the total cardinality of dimensions, and m_{cls} be the number of classes. Therefore, there are at most *card* sorted lists. Each upper bound can be in at most n sorted lists. The total size of sorted lists is at most nm_{cls} integers. However, some dimensions of many bounds may be "*". Thus, the real size of sorted lists is often much smaller than nm_{cls}.

Answering Point Queries Using Sorted Lists

Given a point query of cell c, we need to identify the upper bound of the class that c belongs to. For example, to find the aggregate value of query point $(Tor,*,*)$ in

the quotient cube shown in Figure 3.2, we can identify the upper bound "closest" to the query cell c, which is $(Tor, b, d2)$. The following lemma verifies the idea.

Lemma 3.3. *Let c be a query cell and ub be a class upper bound such that $c \preceq ub$ and there exists no class upper bound ub' such that $c \preceq ub' \prec ub$. Then, c is in the class of which ub is the upper bound.*

Proof: Suppose that c is not in the class of ub. Let ub' be the upper bound of the class that c belongs to. So $c \prec ub'$. The quotient cube lattice preserves the roll-up/drilldown weak congruence. Since $c \prec ub$, $c \prec ub' \prec ub$ must be hold. A contradiction. ■

Lemma 3.3 suggests an interesting potential for answering point queries: we only need to search the set of upper bounds and find the closest ancestor of the query cell.

Example 3.2 (Answering Point Queries Using Sorted Lists). Let us consider answering a point query $c = (Tor, b, *)$ in the quotient cube given in Figure 3.2. The sorted lists are shown in the middle column of Table 3.2. Clearly, value Tor and b appear in every ancestor cell of c. Thus, we only need to find the common upper bounds in sorted lists $L(1.Tor)$ and $L(2.b)$. On the other hand, since we want the closest ancestor, we only need to find the common upper bound with the smallest bound-id. By comparing the sorted list $L(1.Tor) = 7$ and $L(2.b) = 3 - 5 - 7$, the smallest common bound-id is 7. Thus, we determine that the upper bound of the class that c belongs to is $(Tor, b, d2)$ and the aggregate value of c is 6. The following lemma verifies the ideas in this example.

Lemma 3.4. *Let* $c = (*, ..., v_1, ., *, .., v_k, ..)$ *be the query point, where* $v_1, ..., v_k$ *be all non-* dimension values appearing in* c*. Let* $d_{v_1}, ..., d_{v_k}$ *denote the dimensions with non-* value. Let* ub *be the upper bound such that* ub *:* bid *is the smallest bound-id common in* $L(d_{v_1}.v_i), ..., L(d_{v_k}.v_j)$*. Then,* c *is in the class in which* ub *is the upper bound.*

Proof: First we show $c \preceq ub$. ub is the upper bound such that $ub : bid$ is the smallest bound-id common in $L(d_{v_1}.v_i), ..., L(d_{v_k}.v_j)$, So in dimension $d_{v_1}, ..., d_{v_k}$, ub must have value $v_1, ..., v_k$. Therefore, $c \preceq ub$.

Then we show there exists no ub' such that $c \preceq ub' \preceq ub$. Suppose there is another ub' such that $c \preceq ub' \preceq ub$. By Definition and Lemma 3.2, $ub'.bid$ appears in $L(d_{v_1}.v_i), ..., L(d_{v_k}.v_j)$ and $ub' \prec_L ub$. That leads to a contradiction that $ub.bid$ is not the smallest bound-id common in $L(d_{v_1}.v_i), ..., L(d_{v_k}.v_j)$. According to Lemma 3.3, c is in the class of which ub is an upper bound. ■

We know that answering a point query c is to identify the upper bound of the class that c belongs to. Thus the question becomes *how to find the smallest bound-id common in a set of sorted lists.* Next, we present the algorithm of point query.

Algorithm 3.2 presents a k-way search method. Here k is at most the number of dimensions in the data cube. In the worst case, it reads (kl) upper bounds, where l is the maximal length of sorted lists.

To further improve the performance, we propose an optimization technique called ***segment index***. The idea is to divide a sorted list into segments, building index using the first bound-id in each segment and only search segments which may contain common bound-ids.

Algorithm 3.2. [Point Query Answering Using Sorted Lists]
Input: A point query $q = (*, ..., v_1, ., *, .., v_k, ..)$
Output: The aggregate values of cell q
Method:

```
1.  Load k sorted list L_1,...,L_k for non-* value in q
2.  Call function SBid(L_1,...,L_k) to find the smallest bound-id bid common in
L_1,...,L_k;
3.  Find the class bid and its aggregate values and return;

Function SBid(L_1,...,L_k)
1.  for (1 ≤ i ≤ k), let h[i] be the first bound-id in L_i;
2.  while (true ) do
      if (h[1] = ... = h[k]) then return (h[1]);
      let maxbid = max h[1],...,h[k];
      for (1 ≤ i ≤ k)
        if (h[i] < maxbid) then read next from L_i
          until the first bound-id ≥ maxbid;
        meanwhile, if the list ends, return;
```

Figure 3.4: Algorithm - Point Query Answering Using Sorted Lists

Example 3.3 (Segment Index). Let us consider finding the smallest bound-id common in the sorted lists L_1=1-5-6- 10-23-45-52-65-72-92-103-128-...and L_2=75-92-98-.... Algorithm 3.2 has to read 12 bound-ids (10 bound-ids from L_1 and 2 from L_2) from before it determines that the smallest common bound-id is 92.

We can build segment index for the sorted lists. Suppose that a segment holds 4 bound-id, i.e., the first segment of L_1 holds bound-ids 1, 5, 6, and 10. The index for L_1 is $Idx(L_1)$ =1-23-72-... and the index for L_2 is $Idx(L_2)$=75-... The first bound-id in L_2 is 75, which is larger than the index values of the first 3 segments of L_1. Thus, there is no need to read the first 2 segments of L_1. We can start from the third segment of L_1. In total, only 7 bound-ids are read, i.e., the first 3 bound-ids in $Idx(L_1)$, the first bound-id in $Idx(L_2)$, the first 2 bound-ids in the third segment in L_1 and the first 2 bound-ids in L_2. ■

For a sorted list containing l bound-ids, if each segment has m bound-ids, then

the space overhead of the index is $\lceil \frac{l}{m} \rceil$. Clearly, if m is too small, the index is space-costly and the saving can be trivial. On the other hand, if m is too big, then the chance that a segment can be skipped is small. An appropriate value of m will produce good pruning effect with a moderate space overhead.

Answering Range Queries Using Sorted Lists

A range query is of the form $(*, \ldots, v_1, *, \ldots, \{u_{P1}, \ldots, u_{p_r}\}, \ldots, v_i, *, \ldots, \{w_{q_1}, \ldots, w_{q_s}\}, *, \ldots, v_k, *, \ldots)$. It can be rewritten as a set of point queries. In usual, the set of query cells share same values in some dimensions(point dimensions) and are different in some other dimensions (range dimensions). The k-way search algorithm (Algorithm 3.2) can be extended to answer a set of point queries by going through the related sorted lists only once.

Suppose given a 4-dimension range query $(x_1, v_1, ...v_l, x_2, w_1, ...w_k)$. We will have two sorted lists for two point dimensions $L(x_1), L(x_2)$ and two grouped sorted lists for two range dimensions, $LG(v) = L(v_1), ...L(v_l)$ and $LG(w) = L(w_1), ...L(w_k)$. We can apply Algorithm 3.2 to find the smallest bound-id common in $L(x_1), L(x_2), LG(v)$ and $LG(w)$ with two slight changes.

- Make modifications to deal with the grouped sorted lists, ie., $LG(v), LG(w)$. Every member inside one group can be used to represent the whole group as long as it contains the common bound-id. In other words, we will find the smallest bound-id common in $L(x_1)$,$L(x_2)$, any member list in $LG(v)$ and any member list in $LG(w)$.

- Apparently the result of a range queries is usually more than one cell. Once a common bound-id is found, a point cell is added to the answer. An additional bookkeeping on which point cell has already been selected is needed to avoid

possible overlap. For instance, we find a common bound-id in $L(x_1)$,$L(x_2)$, $LG(v) : L(v_3), LG(w) : L(w_2)$. So point cell (x_1, v_3, x_2, w_2) is selected. Later on, we may find a common bound-id in $L(x_1)$,$L(x_2)$, $LG(v) : L(v_3), LG(w) : L(w_2)$ again. This time, we do not record it as an answer and throw it away.

The optimizations of building segment index can be extended correspondingly.

3.3.3 Bitmap Vectors

As an alternative, sorted lists can be implemented using bitmap vectors. For example, the sorted lists and bitmap vectors in the quotient cube in Figure 3.2 are shown in Table 3.2, .

The k-way search now becomes the AND operation of bitmap vectors until a bit 1 is found. The algorithms of answering various queries as well as their optimizations can be translated correspondingly.

3.4 Discussion

In summary, a quotient cube is a representation of a cube lattice in terms of classes of cells. The roll-up/drill-down relations are captured as a partial order among classes. Each class contains cells that have the same aggregate value(s). Thus, a user can drill down from class to class in a quotient cube in the same way he/she drills down from cell to cell in a cube lattice. Conceptually, one can also ***drill down into*** a class, asking to open it up and inspect its internal structure. For example, in the quotient cube of Figure 3.2(b), we can drill-down into class C_7 and investigate the internal structure of the class.

We can answer various queries and conduct various browsing and exploration

operations using a quotient cube. As one interesting point, 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) [22]. For example, a manager may ask an intelligent roll-up query like "*in the data cube in Figure 1.1, starting from* $(Tor, b, d2)$*, what are the most general circumstances where the average sales is still* 6*?* The answer should be $(*, *, *)$ except for $(*, *, d2)$ and $(*, b, *)$. If we search the lattice in Figure 1.1, we may have to search the 8 descendent cells of $(Tor, b, d2)$. However, if we search in the class lattice in Figure 3.2(b), we only need to search at most 3 classes, and not any specific cells in any classes. This example shows that semantic compressions reduce the search space for analysis and exploration dramatically.

Chapter 4

QC-Tree

In last chapter, we discussed quotient cubes and a direct tabular representation called QC-table, an index structure called sorted list and corresponding algorithms of answering queries. While the proposals are interesting, many open issues are unaddressed and some significant improvements can be achieved. Firstly, a tabular representation with additional index is not as compact as possible and thus still wastes space. Secondly, if we develop more effective storing and indexing structures, efficient algorithms for query answering should also be possible. Thirdly, an important topic about maintaining quotient cubes incrementally against updates is not addressed yet.

In this section, we develop QC-tree, a compact data structure to store and index quotient cubes efficiently. We also develop algorithms to construct a QC-tree directly from the base table and incrementally maintain a QC-tree under various updates to the base table. We show how various queries can be answered efficiently using the QC-tree structure.

4.1 The QC-Tree Structure

We would like to seek a compact data structure that (a) retains all the essential information in a quotient lattice, yet be concise; (b) enables efficient answering of various kinds of queries including point, range, and iceberg queries; and (c) allows for efficient maintenance against updates. The ***QC-tree*** (short for quotient cube tree) structure we develop in this chapter meets all the above requirements.

The key intuition behind QC-trees, is that *the set of class upper bounds in a quotient cube w.r.t. cover partition captures all the essential information about classes in the quotient cube.*

Consider the set of upper bounds B in a cover quotient cube(we continue to focus on cover quotient cubes, unless otherwise specified). First, we represent each bound as a string w.r.t. a given dimension order, by omitting any "*" values. E.g., $(Van, b, d1)$ and $(*, b, *)$ would be represented as $Van \cdot b \cdot d1$ and b, respectively, w.r.t. the dimension order Location-Product-Time.

Next, we construct a tree by means of sharing common prefixes. Each bound is represented by some nodes in the QC-tree. And the sequence of node labels associated with the path from the root to the bottom corresponds exactly to the string representation of the bound. Given a sequence of values $z_1 \cdot \ldots \cdot z_k$, where 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 $* \cdot b \cdot *$, $\langle *, b, * \rangle$ is node 10 in Figure 4.1, while for $Van \cdot b \cdot d1$, $\langle Van, b, d1 \rangle$ is node 4. Moreover, with the last node representing a bound, we can associate aggregate information, e.g, sum, count, etc. of that class.

Different dimension order leads to different size of QC-trees. Heuristically, dimensions can be sorted in 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.

To capture the drill-down relations in a quotient lattice, additional links are needed. Consider any two bounds $u, u' \in B$ such that there is a drill-down relation (u, u') in quotient lattice. One of the following situations can arise.

1. The string representation of u is a prefix of that of u'. In this case, the tree constructed so far already contains a path from the root passing through nodes x_u and x'_u such that x_u represents u and $x_{u'}$ represents u'. It is important to note that in this situation, the path above acts as a drill-down path that takes us from the class of u to that of u'.

2. A second possibility is that the string representation of u may not be a prefix of that of u' and a drill-down from u to u' cannot be captured in the tree constructed. A straightforward solution to capture those drill-down relations is to explicitly add them in.

A drill-down link is a pointer and has a dimension value as label. The dimension value label of the link is always identical to the label of the node that the link points to. Drill-down links are always "downward". That means if there is a link from node N_1 to node N_2, the label of the link is always on a dimension after the dimension of N_1. For example, the link from node $b : 15$ to $d1 : 9$ with label $d1$ means that drilling down from $(*, b, *)$ using dimension value $d1$ will arrive at a class with upper bound $(Van, b, d1)$.

Definition 4.1 (QC-Trees). The ***QC-tree*** for a quotient cube Q is a directed graph (V, E) such that:

Class	Upper Bound
C_1	(*,*,*)
C_2	(Van,*,*)
C_3	(*,b,*)
C_4	(*,*,d2)
C_5	(Van,b,d1)
C_6	(Van,f,d2)
C_7	(Tor,b,d2)

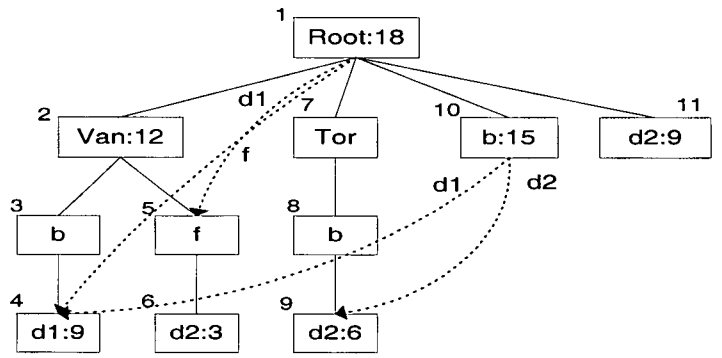


Figure 4.1: Classes and QC-Tree for the Quotient Cube in Figure 1.1

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 D_i on which b_1 and b_2 differ, there is either a tree edge or a link (labelled 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 4.1 is self-explanatory, except that the last condition needs more explanations: Condition 4 simply states that 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 4.1, there is a node representing every class upper bound of the quotient lattice of Figure 3.2(b). For instance, node 10 represents $(*, b, *)$ and stores the corresponding aggregate value 15. One can drill down from C_3 (whose upper bound is $(*, b, *)$) to C_7 (whose upper bound is $(Tor, b, d2)$) in the quotient lattice. The two upper bounds differ in two dimensions: Location and Time. The first difference (i.e., a drill-down from class C_3 to class C_7 by specifying Tor in dimension Location) is indicated by a tree edge from node $1 = \langle * \rangle$ to node $7 = \langle Tor \rangle$ in the QC-tree. The second difference (i.e., a drill-down from C_3 to C_7 by specifying $d2$ in dimension Time) is captured by a link from node $10 = \langle *, b \rangle$ to $9 = \langle Tor, b, d2 \rangle$ with label $d2$.

Lemma 4.1. $\forall$ *node in QC-tree, no two out-arcs have the same label*

Proof: Out-arcs of a node in a QC-tree are either tree edges or drill-down links. We can treat tree edges as special drill-down links that link a parent node to its child. Let N be a node in a QC-tree and let $C(N)$ be the class which N represents. Suppose N has two out-arcs with the same label(dimension D and dimension value v) pointing to two different nodes N_1 and N_2. We know that the two nodes with the same label must belong to two different classes. Let N_1 belongs to class C_1 and N_2 belongs to class C_2. These two out-arcs can be represented by the drill down operation $Drilldown(C(N), D, v)$. By the semantics of OLAP, operation $Drilldown(C(N), D, v)$ has only one result cell. On the other hand, in a quotient cube, one cell can only be a member of one class. So C_1 and C_2 cannot be two different classes. A contradiction. ■

Theorem 4.1 (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 quotient 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.*

Proof: The uniqueness of the path in the QC-tree representing *ub* follows the third condition in Definition 4.1. For the second claim, the uniqueness of a QC-tree is guaranteed by the following: (1) the cover quotient cube QC is unique; (2) each class is uniquely represented by its only upper bound in the QC-tree; and (3) the drill-down links are unique according to the fourth condition in Definition 4.1. ■

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.

4.2 Construction of a QC-Tree

The construction of a QC-tree can be done in two steps. First, a variant of the Depth-First Search algorithm in Section 3.1 is used to identify all classes as well as upper bounds in a quotient cube. For each class C, two pieces of information are maintained: the upper bound and the id of C's lattice child class from which C is generated. Compared to Algorithm 3.1, *a key advantage of QC-tree construction algorithm is that it only maintains the upper bounds. No information about lower bounds is ever needed.* In particular, step 6 in Algorithm 3.1 is pruned. Considerable savings thus can be achieved. The new 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 redundancy and obtain the *real classes*, we identify the redundant temporary classes in the next step. We then sort all the upper bounds in the dictionary order, where a user-specified order of dimensions is used. Within each dimension, we assume values are ordered with "$*$" preceding other values.

As the second step, the upper bounds are inserted into the QC-tree. If the upper bound of a temporary class exists in the tree when the upper bound is inserted, the redundancy is identified and a directed link is built according to Definition 4.1. The efficiency of insertion is expected due to the prefix sorting. The sort order also guarantees the upper bounds always come after all their drill-down parents.

Figure 4.2 shows the algorithm of QC-tree construction. The following example illustrates the algorithm in detail.

ID	UB	LB	Chd	Agg
i_0	$(*,*,*)$	$(*,*,*)$	$--$	18
i_1	$(Van,*,*)$	$(Van,*,*)$	i_0	12
i_2	$(Van,b,d1)$	$(Van,b,*)$	i_1	9
i_3	$(Van,f,d2)$	$(Van,f,*)$	i_1	3
i_4	$(Tor,b,d2)$	$(Tor,*,*)$	i_0	6
i_5	$(*,b,*)$	$(*,b,*)$	i_0	15
i_6	$(Van,b,d1)$	$(*,b,d1)$	i_5	9
i_7	$(Tor,b,d2)$	$(*,b,d2)$	i_5	6
i_8	$(Van,f,d2)$	$(*,f,*)$	i_0	3
i_9	$(Van,b,d1)$	$(*,*,d1)$	i_0	9
i_{10}	$(*,*,d2)$	$(*,*,d2)$	i_0	9

Temporary Classes

ID	UB	LB	Chd	Agg
i_0	$(*,*,*)$	$(*,*,*)$	$--$	18
i_{10}	$(*,*,d2)$	$(*,*,d2)$	i_0	9
i_5	$(*,b,*)$	$(*,b,*)$	i_0	15
i_1	$(Van,*,*)$	$(Van,*,*)$	i_0	12
i_2	$(Van,b,d1)$	$(Van,b,*)$	i_1	9
i_6	$(Van,b,d1)$	$(*,b,d1)$	i_5	9
i_9	$(Van,b,d1)$	$(*,*,d1)$	i_0	9
i_3	$(Van,f,d2)$	$(Van,f,*)$	i_1	3
i_8	$(Van,f,d2)$	$(*,f,*)$	i_0	3
i_4	$(Tor,b,d2)$	$(Tor,*,*)$	i_0	6
i_7	$(Tor,b,d2)$	$(*,b,d2)$	i_5	6

Sorted Temporary Classes

Table 4.1: Temporary Classes Returned by the Depth-First Search

Example 4.1 (QC-Tree Construction). Let us build the QC-tree for the base table in Table 1.1. In the first step, we identify all temporary classes by the depth-first search starting from cell $(*,*,*)$. We calculate the aggregate of cell $(*,*,*)$. Since there is no dimension value appearing in all tuples in the base table, $(*,*,*)$ forms a class with itself as the upper bound. Then, the search exploits the first

Algorithm 4.1. [Constructing a 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);
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 $DFS(c, B_c, k, pID)$
//c is a cell and B_c is the partition of the base table

```
1.  Compute aggregate of cell c;
2.  Compute the upper bound d of the class c, by ''jumping'' to the
    appropriate upper bounds;
3.  Record a temp class with lower bound c, upper bound d and
    and parent pID. Let bID 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 B_d;
        if B_d is not empty, call DFS(d, B_d, j, bID);
6.  return;
```

Figure 4.2: Algorithm - QC-Tree Construction

dimension, `Location`. The tuples in the base table are sorted in this dimension. There are two locations, so two partitions are generated. In the first partition, i.e., the tuples about sales in location Van, no identified dimension value appears in every tuple, so $(Van, *, *)$ is an upper bound. The search recurs by exploring various dimension values.

The temporary classes returned from the depth-first search is shown in Table 4.1(a). They are sorted in the dictionary order as in Table 4.1(b) and inserted into a tree. First, we process classes $i_0, i_{10}, i_5, i_1, i_2$ and create the corresponding nodes in the QC-tree (nodes 1,11,10,2,3,4 in Figure 4.1). Then, class i_6 comes in. Its upper bound has already been inserted (its upper bound is equal to the upper bound of previous class, i_2). We only need to build a drill-down link. Class i_5 is the lattice child, so we compare the upper bound of i_5 and the lower bound of i_6 to get the drill-down link label $d1$, and then add a drill-down link with label $d1$ from $(*, b, *)$ (node 10) to $(Van, b, d1)$ (node 4). As another example, we consider class i_8. Its upper bound has been inserted. By comparing its upper bound and that of its child class i_0, (*,*,*), we get the drill-down link label f. So a link from node 1 to 5 is added. ■

One of the major cost in this algorithm is the step of sorting temporary classes in the sense that the number of temporary classes could be very large. To improve the performance, we propose two optimization approaches.

Optimization 1 When temporary classes are created, we can hash them to appropriate disk buckets based on the values in the first (or first few) dimension. Next as we need to sort temporary classes, we never have to sort across buckets. The set of temporary classes in one bucket can always fit in memory, even though the set of all temporary classes may not be fit in memory.

Optimization 2 By observing the temporary classes, we find that we actually can skip this step entirely while keeping the prefix insertion efficient. Table 4.1(a) shows the temporary classes that we created. i_0, i_1, i_2, i_3, i_4, i_5 and i_{10} are the real classes whose upper bounds need to be inserted by prefix sharing. The interesting point is that those upper bounds are already ordered (because of the depth-first search we used). Therefore, the only work we need to do is to identify those redundant classes in order to maintain the drill-down links. It turns out identifying redundant classes is not difficult either. Let's look at temp class i_6. i_6 is created from its lattice child i_5 by exploring $(*, b, *)$ to $(*, b, d1)$. The exploration is on dimension 3 (replace * with $d1$). Compare i_6's upper bound $(Van, b, d1)$ and its lower bound $(*, b, d1)$. The first dimension that they differ is on dimension 1, which is before 3. So we can say i_6 is a redundant class. In general, the redundant classes can be identified by the following method. For a temp class C, let d_i be the dimension on which C explored from its lattice child, let d_j be the first dimension that the upper bound and lower bound of C differ. If $d_j < d_i$, C is a redundant class. Otherwise it is a real class.

4.3 Query Answering Using QC-Trees

Typical queries over a data warehouse can be classified into three categories, i.e., 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 Lemma 4.1). 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. Therefore, it is important that we can find this path fast.

4.3.1 Point Queries

In a point query, a cell c is given (called the query cell) and we are asked to 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 $\ell_i, i = 1, \ldots, k$.

To answer the point query, one can start at the root and set the current value to v_1. At any current node N, look for a tree edge or link labelled 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.

The procedure is straightforward. However, the question is *what if at any current node, a tree edge or link with label v_i cannot be found.*

Lemma 4.2 (Point Query Answering). *Let c be a point query as above and let N be any current node such that there is no tree edge or link out of N labelled 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.*

Proof: Suppose to the contrary.

Case 1: Suppose the last dimension j for which N has a child is $\geq \ell_i$. In this case, there is no such upper bound that agrees with c on non-* dimensions $\ell_1, \ldots, \ell_i$. c has an empty cover set. A contradiction.

Case 2: $j < \ell_i$, but N has more than one child for dimension j. (1) if N is a class node. N represents the class upper bound $ub_N = (*, \ldots, v_1, \ldots, v_2, *, \ldots, N, *, \ldots)$. From Definition 4.1, there must be a tree edge or a link labelled v_i out of N. A contradiction. (2) if N is not a class node. Let $x_{j_1}, x_{j_1}, ..., x_{j_m}$ be the children of N for dimension j. Node x_{j_1} represents the upper bound $(*, \ldots, v_1, \ldots, v_2, *, .., N, .., x_{j_1}, ..)$, x_{j_2} represents the upper bound $(*, \ldots, v_1, \ldots, *, \ldots, v_2, *, \ldots, N, \ldots, x_{j_2}, \ldots)$, and so on. Since there is no such class upper bound as $(*, \ldots, v_1, \ldots, v_2, *, \ldots, N, *, \ldots)$ and j is not the last dimension of the base table, because of the cover equivalence, there must be a class upper bound $ub = (*, \ldots, v_1, \ldots, *, \ldots, v_2, *, \ldots, N, \ldots, x_l, \ldots)$, where dimension $l > j$. So N must necessarily have a child for a dimension $> j$. A contradiction.

Thus, we get contradictions in both cases. ■

Lemma 4.2 suggests an efficient method for answering point queries. First, we follow the above straightforward procedure. 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.

For a point query, this answering 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 explain it using the QC-tree example of Figure 4.1.

Example 4.2 (Point Query Answering). Using our running example QC-tree

Algorithm 4.2. [Point Query Answering]
Input: a point query $q = (*, .., v_1, ., *, .., v_k, .., *)$ 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 labelled v_i

```
  if newRoot has a child or link pointing to N labelled v_i
    newRoot = N;
  else
    Pick the last child N of newRoot in the last dimension, say j;
  if (j < the dimension of v_i) call searchRoute(N, v_i);
  else return null;
```

Figure 4.3: Algorithm - Point Query Answering

of Figure 4.1, suppose:

- We want to answer the query $(Tor, *, d2)$. From the root node, we find there is a child labelled Tor, node 7. From node 7, we cannot find any tree edges or links labelled $d2$. So we pick the child on the last (in this case, only) dimension, which is b, node 8. From node 8, we can find a child labelled $d2$, node 9. Being a class node, it has a aggregate value in it. This value is returned.

- Answer another query $(Tor, *, d1)$. From the root node, we find there is a child labelled Tor, node 7. From node 7, we cannot find any tree edges or links labelled $d1$. so pick the child on the last dimension, which is b, node 8. From node 8, we cannot reach a node labelled $d1$. Then pick the child on the

last dimension, which is $d2$, node 9. But $d2$'s dimension is *not* later than $d1$'s dimension. So query fails.

- We consider the query $(*, f, *)$. From the root node, we find there is a link labelled f. We have examined all non-* values in the query. But f has no aggregate value, so we pick the child on the last dimension, which is $d2$. $d2$ has aggregate values, thus return it. ■

4.3.2 Range Queries

A range query is of the form $(*, \ldots, v_1, *, \ldots, \{u_{P1}, .., u_{p_r}\}, \ldots, v_i, *, \ldots, \{w_{q_1}, .., w_{q_s}\}, *, \ldots, v_k, *, \ldots)$. 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 categorical 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 in dimension i do not exist, we can prune the search space dramatically.

The algorithm of answering range query is given in Figure 4.4. Experimental evaluation of point and range query answering algorithms can be found in Chapter 5.

Example 4.3 (Range Query Answering). Let's answer range queries using our running QC-tree of Figure 4.1.

- Suppose we want to answer a range query $([Van, Tor, Edm], [b, f], d1)$

  (1) We begin from root $(*, *, *)$. We find that Edm cannot be reached from

Algorithm 4.3. [Range Query Answering]
Input: a range query q
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 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_{i_m} in the range
      Call searchRoute(newRoot, v_{i_m}),
      Let newRoot be the return node
      if newRoot is not NULL
        Call rangeQuery(q, newRoot, i + 1)
```

Figure 4.4: Algorithm - Range Query Answering

(*,*,*), the cells that begin with Edm will be pruned immediately.

(2) From node Tor, only node b exists. Since f does not exist, we do not care about this branch any more. Then we try to reach $d1$ from b, but fail. Nothing is returned.

(3) From node Van, both b, f can be reached. Searching $d1$ from b is successful, but searching $d1$ from f is failed. One result with aggregate value is returned.

- Suppose we want to answer a range query $(*, [b, f], d1)$. the order of processing

nodes is:

(1) $root \rightarrow b \rightarrow d1$. Return aggregate value.

(2) $root \rightarrow f$, but cannot reach $d1$. No result returns. ■

4.3.3 Iceberg Queryies

Iceberg queries are queries that ask for all cells with a measure satisfying the threshold, whether the aggregate function is monotone (e.g., SUM, MIN, etc.). 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.[1] Process the range query on this subtree.

4.4 Incremental Maintenance

Many emerging applications (like analysis of streaming data) require fast incremental maintenance of the data cube. Thus, maintaining a QC-tree structure incremen-

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

tally against updates is an important task. In general, updating a base table can fall into three categories: insertion, deletion and modification.

4.4.1 Incremental Maintenance of Quotient Cubes

An insertion or deletion of a base table will perform a series of the following operations on the quotient classes in a quotient cube.

Updating a class The quotient class structure is unchanged. Only the associated aggregate values need to be modified.

Creating a class Some new classes that solely contain part or all of the newly inserted tuples may be created.

Deleting a class Some original classes may no longer cover any remained tuples. After the deletion of the base table, these classes should be outright removed.

Splitting a class One class can be split into two classes whenever some member cells cover different base table tuples than others. The original upper bound(s) remains the upper bound(s) of one class. The cells with different cover set form another class with a new upper bound.

Merging classes Two classes can be merged to one class as long as they are equivalent to each other and the merge will not create holes.

We next use our cover quotient cube to illustrate the idea in detail.

Insertion

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

Consider inserting one single tuple into the base table. Let QC be the quotient cube, t be the new tuple. Two situations may arise.

Case 1 t has the same dimension values as an existing tuple in the base table. Each class 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. For example, let the newly inserted tuple be $(Van, s, d2)$. *What are the cells whose cover set is changed by t?* Clearly, they are generalizations of $(Van, s, d2)$, e.g., $(*, s, d2)$, $(Van, *, d2)$, $\ldots$, $(*, *, *)$. All the classes in QC can be divided into three categories based on whether a class has a member cell that covers t. 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.

- 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_3 in our example, $t \wedge ub = (*, *, *)$ which is clearly not in C_3. Since the cover set of its member cells does not change, this class remains the same when t inserted.
- 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_4 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 needed.

- Category 3 the upper bound of the class does not cover t, but some member cells in the class do. 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_6 in QC, the member cell $(Van, *, d2)$ and all its descendants in C_6 cover $(Van, s, d2)$, while the upper bound of C_6, $(Van, f, d2)$ does not.

  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, covering the same tuple set as C. For example, $(Van, *, d2)$ in class C_6 form a separate class C' with upper bound $ub \wedge t = (Van, *, d2)$. All other member cells (whose cover sets have not been changed) are in another class, C'', which contains $(Van, f, d2)$, $(Van, f, *)$, $(*, f, d2)$ and $(*, f, *)$, whose upper bound is still $(Van, f, d2)$. 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 quotient lattice.

Let ΔDB be a set of tuples inserted into the base table. To reflect the insertion, three tasks should be accomplished upon the original quotient cube QC:

1. Find those quotient classes such that all of the member cells cover a subset of ΔDB. Perform class updating correspondingly.

2. Find those quotient classes such that some of the member cells cover a subset

of ΔDB while others do not. Perform class splitting correspondingly.

3. Create new classes whose member cells cover only a subset of ΔDB.

Lemma 4.3. *In the new quotient cube created by our insertion strategy, all classes are maximally convex and connected.*

Proof: Let DB be the original base table. Let ΔDB be a set of tuples inserted into the base table.

(1) Our insertion strategy guarantees the member cells in each class cover the same tuples. The operations of updating, splitting and creating do not break up the convexity of the classes. All cells in each class are connected to each other.

(2) Next we prove the maximality.

Suppose to the contrary. Two classes C and D can be merged. All classes in the new quotient cube fall into 3 categories: (A) The classes that only cover some tuples in DB. They are the old classes in the original quotient cube that are not modified by the insertion. (B) The classes that only cover some tuples in ΔDB. They are the newly created classes. (C) The classes that cover some tuples in DB and some tuples in ΔDB. They are the updated or split classes.

- Case 1: Both C and D are in category(A). Since all classes in the original cover quotient cube are maximal, C and D can not be merged. A contradiction.

- Case 2: C is in category(A), and D is in category(B). All the member cells in D only cover the newly inserted tuple in ΔDB while cells in C only covers some tuples in DB. C and D can not be merged. A contradiction.

- Case 3: C is in category(A), and D is in category(C). Suppose D is updated from old class D' or split from old class D'. The cover set of D must contain

some of the new inserted tuples in ΔDB. Since C only covers the tuples in DB, their cover sets cannot be the same. So C and D can not be merged. A contradiction.

- Case 4: Both C and D are in category(B). Because they cover the different subset of ΔDB, they cannot be merged. A contradiction.

- Case 5: C is in category(B), and D is in category(C). C is the newly created class who covers a subset of ΔDB only. Suppose D is updated from old class D' or split from old class D'. D must contain some old tuples which are covered by D' while C only covers the new tuples. So C and D can not be merged. A contradiction.

- Case 6: Both C and D are in category(C). Suppose C is updated from old class C' or split from old class C', D is updated from old class D' or split from old class D'. C must contain some new tuples in ΔDB and the cover set of C'. D must contain some new tuples in ΔDB and the cover set of D'. Since the cover sets of C' and D' are different, the cover sets of C and D must be different. So C and D can not be merged. A contradiction.

We get contradictions in all cases. Thus no two classes can be merged. All classes are maximal. ■

Deletion

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 the delete set exist in the base table. If not, it is easy to check and remove from delete set 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. Class deleting or merging is determined by this operation. Owing to the property of cover partitions, we can perform both checks efficiently by simply maintaining a count for each class.

Consider deleting one tuple from the base table. Let QC be the quotient cube, t be the deleted tuple. All the classes in QC can be divided into two categories based on whether a class has a member cell that covers t. Due to the property of cover quotient cubes, there is no such case that some of the cells in a class cover t while others do not. Let C be a class in QC, let $ub = (x_1, \ldots, x_n)$ be its upper bound, and $t = (y_1, \ldots, y_n)$ be the deleted tuple. Let $t \wedge ub = (z_1, \ldots, z_n)$, be the greatest lower bound of t and ub.

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 example, let the deleted tuple be $t = (Tor, b, d2)$. Class C_2, C_5, C_6 in our example fall into this category. Since the cover set of its member cells does not change, this class remains the same when t deleted.

Category 2 the upper bound of class C covers t, thus every member cell of the class covers t. For example, let the deleted tuple be $t = (Tor, b, d2)$. Class C_3, C_4, C_1, C_7 are in this category. First, update the measures and maintain the count of C. Then based on the "count", we can check whether the class deleting or merging should be performed.

Case 1: $C.count = 0$ Class C no longer covers any tuple in base table. Thus C should be deleted,e.g., C_7 will be deleted since it only covers the deleted

tuple.

Case 2: $C.count > 0$ There are two possibilities. If $C.count$ equal to the count of one of its lattice parent class, say C', C can be merged to C', for instance, C_3 can be merged to C_5, C_4 can be merged to C_6 and C_1 can be merged to C_2. Otherwise, C stays the same.

Let ΔDB be a set of tuples deleted from the base table. To reflect the deletion, two tasks should be accomplished upon the original quotient cube QC:

1. Find those classes whose member cells cover a subset of ΔDB in QC. Perform class updating.

2. Based on the updated "count", check whether the class deleting or merging should be performed.

Lemma 4.4. *In the new quotient cube created by our deletion strategy, all classes are maximally convex and connected.*

Proof:

(1) Our deletion strategy guarantees the member cells in each class cover the same tuples. The operations of updating and deleting do not break up the convexity and connectedness of the classes. Since a mergence happens between two classes with parent-child relation, the convexity is retained and all cells in each class are connected to each other.

(2) Next we prove the maximality.

Suppose to the contrary. Two classes C and D can be merged. All classes in the new quotient cube fall into 2 categories: (A) The classes that cover some tuples in

ΔDB before the deletion. They are updated classes whose "count" are decreased. (B) The classes that do not cover any tuples in ΔDB before the deletion. They are either the non-changed old classes in the original quotient or the classes merge the updated classes. Their "count" and cover set remain unchanged.

- Case 1: Both C and D are in category(A). Let C' be the original class of C. Let D' be the original class of D. The cover set of C' and D' are different. Deleting some common tuples from the cover set of C' and D', the results should not be the same. C and D can not be merged. A contradiction.

- Case 2: C is in category(B), and D is in category(A). Let D' be the original class of D. To keep the convexity, a mergence happens between two classes with parent-child relation. Thus, either C is a lattice parent of D or D is a lattice parent of C. Since D is an updated classes and cannot be merged into any of its lattice parent class, C can not be a lattice parent of D. That leaves the only possibility that C is a lattice child of D. Because C is a lattice child of D' in the original quotient cube, $C.count > D'.count$. Since $D'.count > D.count$, we can get $C.count > D'.count$. Thus it is impossible to merge C and D. A contradiction.

- Case 1: Both C and D are in category(B). Since all classes in the original cover quotient cube are maximal, C and D can not be merged. A contradiction.

We get contradictions in all cases. Thus no two classes can be merged. All classes are maximal. ■

Modification

There are basically two kinds of modification of a base table.

- Only measure values are modified. e.g., update tuple $(Tor, b, d2, 6)$ to $(Tor, b, d2, 12)$. In this case, only those classes in the original quotient cube that cover t should have their aggregate measures updated to reflect the modification.

- Dimension values are changed. e.g., update tuple $(Tor, b, d2, 6)$ to $(Tor, f, d2, 12)$. Modifications can be simulated by deletions and insertions in this case.

4.4.2 Incremental Maintenance of QC-Trees

To complete the maintenance of quotient classes, direct operations on a QC-tree need to be derived. Table 4.2 shows the corresponding operations on a QC-tree against quotient classes.

Quotient Class	**QC-tree**
Updating class C	Simply update the aggregate values of corresponding class nodes.
Creating class C	Inserting corresponding nodes into QC-tree and maintain drill-down links when necessary.
Deleting class C	If all the nodes (including the class node) of class C are shared with other classes, we only need to delete the aggregate value in the class node and set the class node to be non-class. If some of the nodes are only owned by C, delete them.
Splitting C' from C	Inserting upper bound of C' into QC-tree and add drill-down links from C' to C if necessary.
Merging C' into C	First, delete the class C'. And then from the lattice child of C', add link(s) pointed to C if necessary.

Table 4.2: QC-Tree Maintenance

We next propose algorithms for incremental maintenance of QC-trees based on the strategy we discussed in Section 4.4.1. Apparently, when multiple tuples need to be inserted or deleted, updating the QC-tree tuple by tuple may not be efficient. Our algorithms therefore are designed for the purpose of batch update.

Insertion

The idea is that during a depth-first generation on the set of inserted tuples ΔDB, we will simultaneously record the classes to be updated on the original QC-tree QC and the new classes to be created(or split) when necessary. We henceforth refer to the process as ΔDFS.

Recall in the DFS function of the quotient cube construction algorithm, for a given cell c, its class's upper bound c' can be easily found based on the partition table (step 2 of Function DFS). However, in ΔDFS, instead of simply recording c' as an upper bound, several additional 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, do $ub \wedge c'$. 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 belongs to. 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 the above situations happens, i.e., c' and ub are incomparable. We need to split the class C in QC that ub belongs to. Let $c'' = ub \wedge c'$, and record a *split* temporary 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.

ΔDFS identifies the upper bounds of all affected classes in the original quotient cube (as in step 3) and all newly created classes (as in step 2).

Algorithm 4.4 (Batch Insertion in QC-Trees).
Input: a QC-tree T, base table ΔDB;
Output: new QC-tree;
Method:

```
1.  Let b = (*,...,*); call ΔDFS(b, ΔDB);
2.  Sort the temp classes s.t.the upper bounds are in the dictionary order
    (* precedes other values )
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 current in QC-tree;
        if (current is a new class)
           insert current into QC-tree
        if (current is a split class)
           split current 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 4.5: Algorithm - Batch Insertion

The batch insertion algorithm is given in Figure 4.5.

Example 4.4 (Batch Insertion). Suppose we want to update the QC-tree of Figure 4.1 against the insertion of two tuples $\Delta B = \{(Van, b, d2, 3), (Van, s, d2, 12)\}$ in the base table. We apply the ΔDFS on ΔDB. Figure 4.6(b) shows the temporary classes created by the ΔDFS. Figure 4.6(b) shows the updated QC-tree.

(1) For temporary class 0, 1, 5 and 8, we update the aggregate of the nodes in QC-tree.

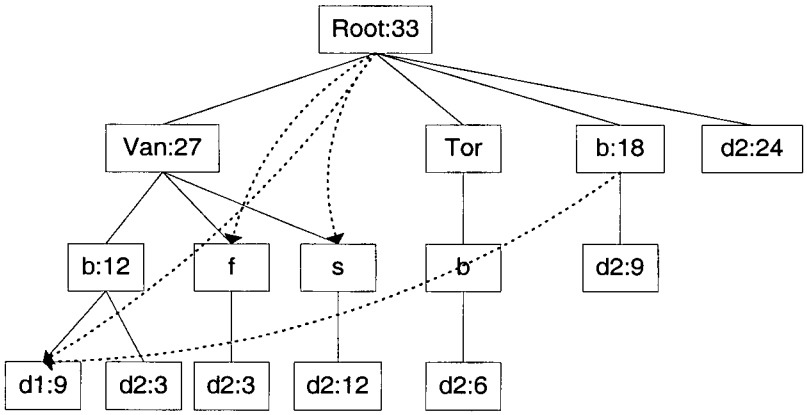
(2) For temporary class 2, we need to split it from the original class $(Van, b, d1)$. We add aggregate values on node 3 which represents $(Van, b, *)$.

(3) For temporary class 3, 4 and 6, we insert corresponding nodes.

(4) For class 7, since its upper bound is the same as class 4, we should add a link

ID	Upr Bnd	Lwr Bnd	Lattice child	Status	Old Upr Bnd
0	(*,*,*)	(*,*,*)	-1	Update	(*,*,*)
1	(Van,*,*)	(Van,*,*)	0	Update	(Van,*,*)
2	(Van,b,*)	(Van,b,*)	1	Split	(Van,b,d1)
3	(Van,b,d2)	(Van,b,d2)	1	Insert	–
4	(Van,s,d2)	(Van,s,*)	0	Insert	–
5	(*,b,*)	(*,b,*)	0	Update	(*,b,*)
6	(*,b,d2)	(*,b,d2)	5	Split	(Tor,b,d2)
7	(Van,s,d2)	(*,s,*)	0	Insert	–
8	(*,*,d2)	(*,*,d2)	0	Update	(*,*,d2)

(a) Temporary Classes Created by ΔDFS



(b) The QC-tree after the insertion.

Figure 4.6: Batch Insertion of QC-Tree

labelled s from its lattice child (*,*,*). ■

Deletion

The maintenance algorithm for deletions is similar to that for insertions. Again the first step of the algorithm is to call a ΔDFS to determine the temporary classes and upper bounds. For a given cell c, its class's upper bound c' can be found by the following steps which are slightly different from the steps in insertion:

1. Find the upper bound ub of the class containing cell c from QC.

2. Determine c's upper bound c' based on ΔDB.

3. Compare ub with c'. Because of the coverage property, we know that there are only two possibilities. Either ub is the same as c' or ub is a roll-up of c'. So the jump can only reach to ub. Record the temp class with ub as its upper bound.

Next when we process the temp classes, an additional step to check whether we can delete or merge the processing classes needs to be performed. Because of the property of cover partitions, this check can be performed efficiently by simply maintaining a count for each class. If the count of the class is zero(which means the cover set of this class is empty), the class should be deleted. If the count is non-zero, we need to check if it can be merged with its parent classes in the quotient lattice. The algorithm in Figure 4.7 shows the details.

Example 4.5. Let QC be a cover quotient cube for the set of tuples $(Van, b, d1, 9)$, $(Van, f, d2, 3)$, $(Tor, b, d2, 6)$, $(Van, b, d2, 3)$,and $(Van, s, d2, 12)$. Suppose $\Delta DB = (Van, b, d2, 3)$, $(Van, s, d2, 12)$.

First, we call ΔDFS to compute the temporary classes and upper bounds, and sort them in the reverse dictionary order ("*" appears last). We delete the classes (with upper bounds) $(Van, b, d2)$, $(Van, s, d2)$ 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 $(Van, b, *)$, $(Van, *, *)$, $(*, b, d2)$, $(*, b, *)$,$(*, *, d2)$. Since their cover sets are not empty, we next check whether they can be merged. Drill down from the class of $(Van, b, *)$ to $(Van, b, d1)$. $(Van, b, d1)$ has the same cover set (i.e., count) as $(Van, b, *)$, so $(Van, b, *)$ can be merged to $(Van, b, d1)$. We then process the rest classes one by one. $(Van, *, *)$,$(*, b, *)$ and $(*, *, d2)$ can not be merged to any class, while $(*, b, d2)$ can be merged to $(Tor, b, d2)$. ■

Algorithm 4.5 (Batch Deletion in QC-Trees).
Input: a QC-tree T, base table ΔDB;
Output: new QC-tree;
Method:

```
1.  Let b = (*,...,*); call ΔDFS(b, ΔDB);
2.  Sort the temp classes s.t.the upper bounds are in the dictionary order
    (* precedes other values)
3.  last = NULL;
4.  while not all temp classes have been processed
      current =next class
      IF current = last
        delete the drill-down link from current's lattice child to last
      ELSE
        update the aggregate value.
      if ub.count == 0
        delete the class of ub.
      else
        if ub has no *, return; // then no need to do merge
        else //drill-down to lattice parents
          (i) while not all * dimensions been proceeded in ub
            (1) Replace * with a value within the domain, form ub'.
            (2) Query ub' in QC.
            (3) if cannot find ub', back to (1)
                if ub' has the same count as ub
                  then merge ub into ub', exit current loop
                if ub' has different count as ub
                  choose another * dimension, back to step (i)
          Until no * dimension can be replaced.
          //ub cannot merge to any other class
```

Figure 4.7: Algorithm - Batch Deletion

One of the key properties of our algorithms for insertions and deletions is that they yield the correct QC-trees reflecting the update and do so efficiently.

Theorem 4.2 (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 4.2 on $DB \cup \Delta DB$. A similar statement holds for deletion.*

Proof: Lemma 4.3 and Lemma 4.4 show that in the quotient cube QC generated by our insertion and deletion strategy, all classes are maximally convex and connected. Theorem 3.1 shows that maximally convex and connected partition of cube lattice is unique. Thus QC is exactly the same as the quotient cube created by re-computing on $DB \cup \Delta DB$ or $DB \cap \Delta DB$. From Theorem 4.1, the produced QC-tree is thus identical to the QC-tree obtained by running the QC-tree construction algorithm. ■

Both batch insertions and deletions 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. Minimizing the number of such queries is of great benefit to the efficiency of the algorithm. Tuple-by-tuple insertion makes repetitions of certain queries. In example 4.4, suppose we insert the two tuples one by one. Cells such as $(*,*,*)$,$(Van,*,*)$,$(Van,*,d2)$, and $(*,*,d2)$ would be queried twice. Batch algorithms avoid such redundant work, so intuitively batch insertion ought to be more efficient. We demonstrate this empirically in Chapter 5.

4.5 Discussion

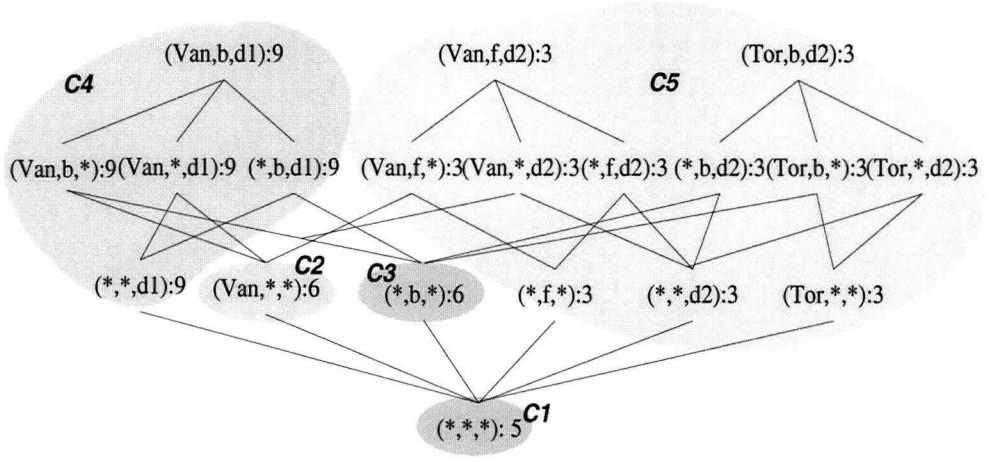
Quotient cubes and QC-trees are fundamental semantic approaches to lossless cube compression. A quotient cube represents a cube lattice in terms of classes, which capture the drill-down relations among cube cells. QC-trees preserve the semantics of quotient cubes by only storing class upper bounds with additional drill-down relations. In a QC-tree structure, one can easily determine whether two cells are equivalent to each other and what is the class upper bound of a cell. One of the key properties of QC-trees is that *not all cells in a data cube exactly match a path*

in the QC-tree, e.g., cell $(Tor, *, d2)$ does not have any exact matching path in Figure 4.1, making QC-tree a distinguished approach that preserves the semantics of data cubes.

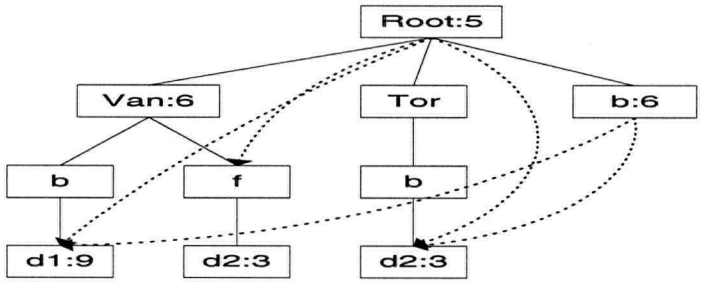
4.5.1 QC-Tree w.r.t Convex Partition

So far, the QC-tree structure we discussed is based on quotient cube w.r.t. cover equivalence. In fact, QC-trees are independent of the equivalence relation used, as long as the equivalence is convex. To illustrate this independence, we consider a different equivalence, namely an equivalence w.r.t. the aggregate function AVG. This equivalence relation $\equiv_{avg}$, was defined in [32] as follows. For two cells c, d in a cube lattice, define $c \sim d$ whenever (i) the aggregate values (AVG) at c and d are the same and (ii) either c is a parent of d or a child of d in the cube lattice. The relation $\equiv_{avg}$ is then the reflexive transitive closure of $\sim$. In [32], it was shown that this equivalence relation is convex.

Example 4.6. In order to make a good example, we slightly change the measure values of Table 1.1 while the dimension values remain the same. Figure 4.8(a) shows the corresponding cube lattice with distinct $\equiv_{avg}$ equivalence classes. The corresponding QC-tree is shown in Figure 4.8(b). First, in comparing this lattice with that in Figure 4.1 for cover equivalence ($\equiv_{cov}$), we find that classes 4, 6 and 7 of $\equiv_{cov}$ are now merged to one (new) class 5. So, the total number of equivalence classes is 5. $(*, *, d2)$ is no longer a class upper bound. These differences manifest themselves in the corresponding QC-tree in Figure 4.8(b) naturally. For example, while the QC-tree in Figure 4.1 shows the root node having a child labelled $d2$, this is not the case in Figure 4.8(b). In its place, we have a direct dotted drill-down link from the root to the node labelled $d2$. ■



(a) Partition based on aggregate value AVG



(b) QC-tree

Figure 4.8: Quotient Cube and QC-Tree w.r.t $\equiv_{avg}$

In general, given an appropriate convex equivalence relation, the corresponding QC-tree can be constructed, by following the same general principles as the cover equivalence.

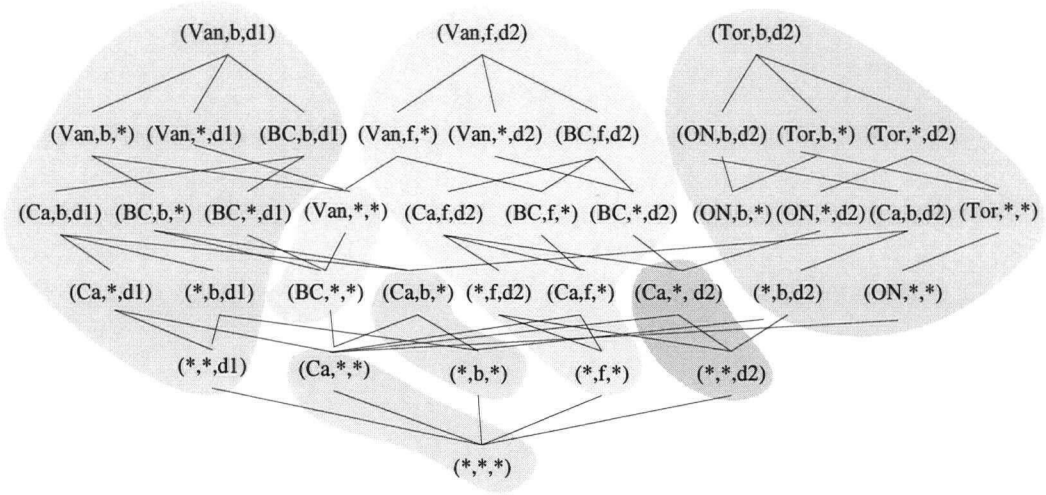
4.5.2 Hierarchy

In OLAP systems, dimensions usually associate with hierarchies, which organize data according to levels. For instance, in Table 1.1, dimension Location might have Country, Province and City as the hierarchy levels.

A data cube keeps its lattice structure in the case of hierarchy. Quotient cubes based on convex and connected partition preserve the drill-down and roll-up semantics as well. QC-tree can be constructed using the same approach, sharing common prefixes of upper bounds in each quotient class except that upper bounds are represented by a string including the information of hierarchy levels. For example, upper bound $(Van, *, *)$ corresponds to a string $Ca \cdot BC \cdot Van$, while upper bound $(Tor, b, d2)$ corresponds to a string $Ca \cdot ON \cdot Tor \cdot b \cdot d2$.

The algorithms of the QC-tree construction, query answering and incremental maintenance can be applied to the situation of dimension hierarchy correspondingly.

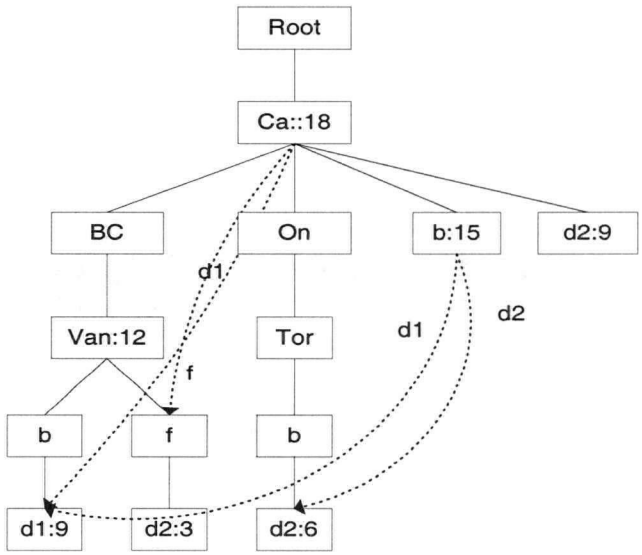
Example 4.7. In Table 1.1, we associate a three-level hierarchy on dimension Location, namely Country-Province-City, and dimension Product and Time remain the same(no hierarchy or hierarchy level is 1). Figure 4.9 shows the cover partition of the cube lattice. The corresponding quotient classes and QC-tree are shown in Figure 4.9(b) and (c). ■



(a) Quotient Lattice with Dimension Hierarchies

Class	Upper Bound	Agg
1	(Ca,*,*)	18
2	(Van,b,d1)	9
3	(BC,*,*)	12
4	(BC,b,*)	15
5	(Van,f,d2)	3
6	(Ca,*,d2)	9
7	(Tor,b,d2)	6

(b) Quotient Classes



(c) QC-tree

Figure 4.9: Quotient Cube and QC-Tree with Dimension Hierarchies

Chapter 5

Experiments

In this section, we present an extensive empirical evaluation of the algorithms we developed in this thesis. All experiments are running on a Pentium 4 PC with 512MB main memory, 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[1] on various algorithms wherever they exist.

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 [31]. It contains $1,015,367$

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

tuples(about 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 Construction 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 [6], where QC-table is to store all upper bounds plainly in a relational table. We associate 5 aggregate functions to the measure value, namely SUM, COUNT, MAX, MIN and MED, which are quite useful in real life data warehouse.

The overall results show that QC-tree and QC-table can achieve substantial reduction. Figure 5.1(a)-(d) 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 a very 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 5.1(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.

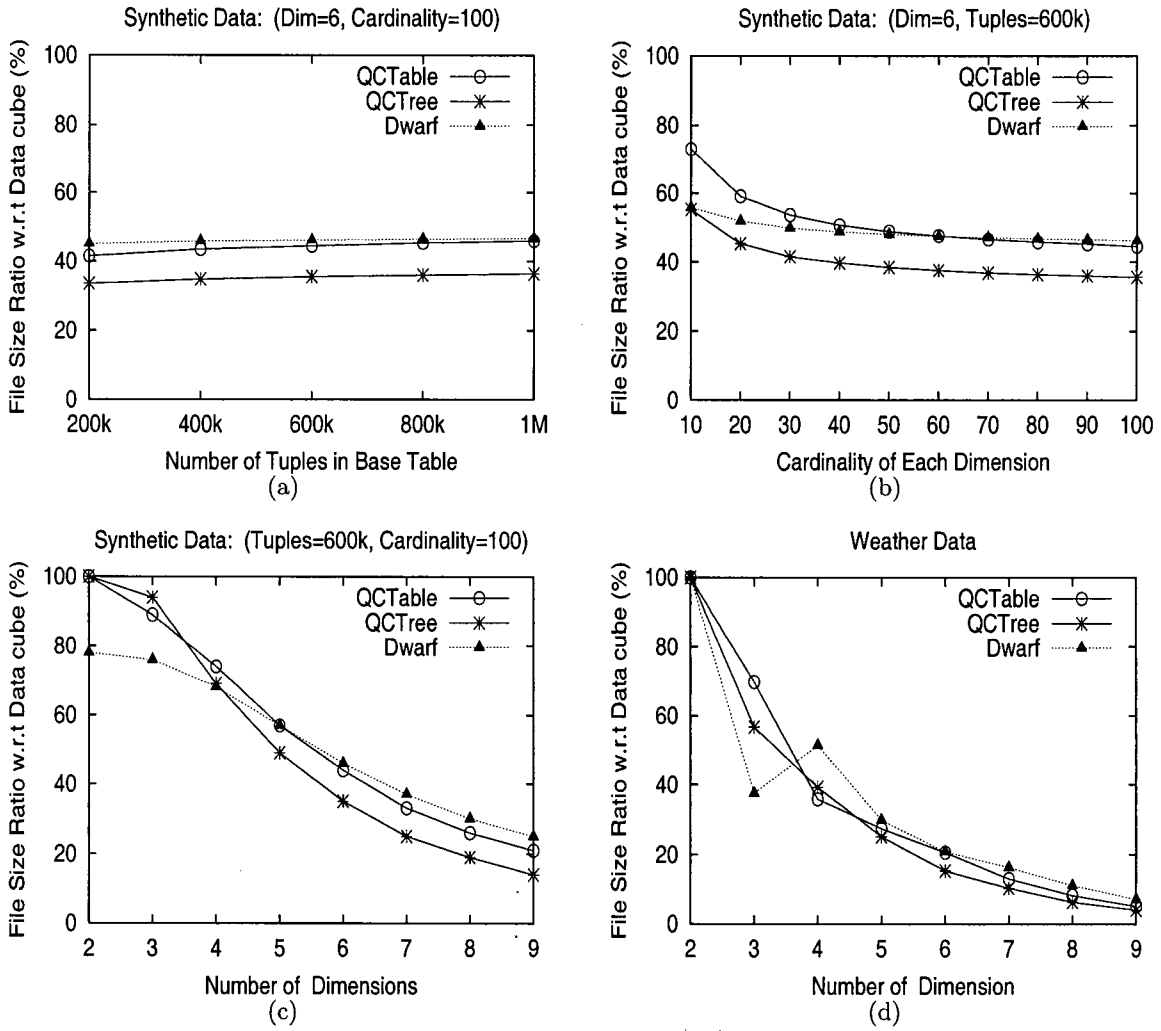


Figure 5.1: Evaluating Compression Ratio

Figure 5.2 shows the construction time. All three methods are scalable on the base table size in terms of the construction time. The depth-first search computation of classes and upper bounds is effective and efficient. A good buffer and I/O manager would help QC-tree and QC-table handle the situation when the number

of dimensions gets large.

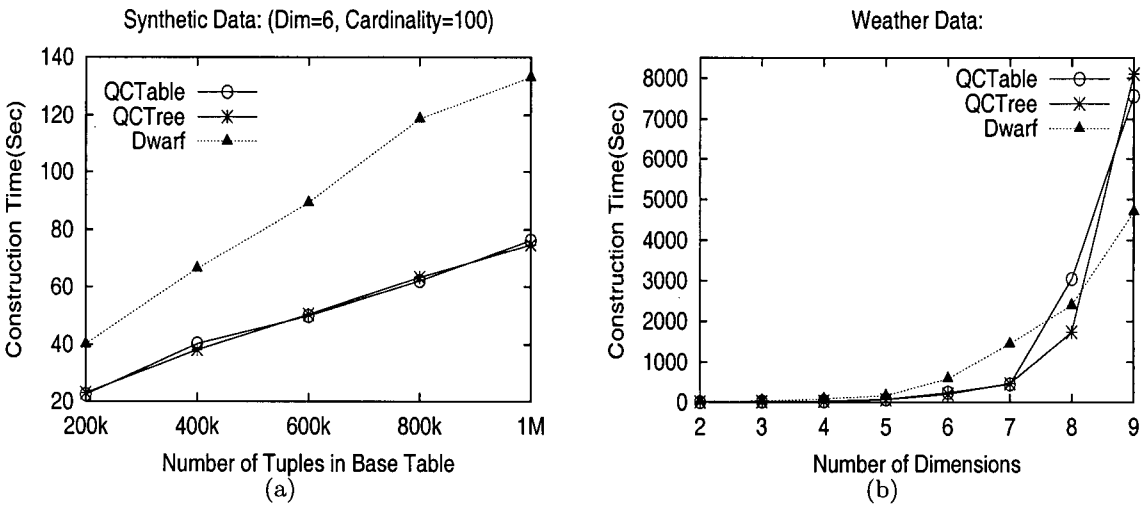


Figure 5.2: Performance of Construction

5.3 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. The results on insertions into both synthetic and real data sets are reported in Figure 5.3. The results on deletions are similar.

We compare three methods: (1) re-computing the QC-tree; (2) inserting tuples one by one; and (3) inserting in batch. Clearly, the incremental maintenance algorithms are much better than the re-computation. 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

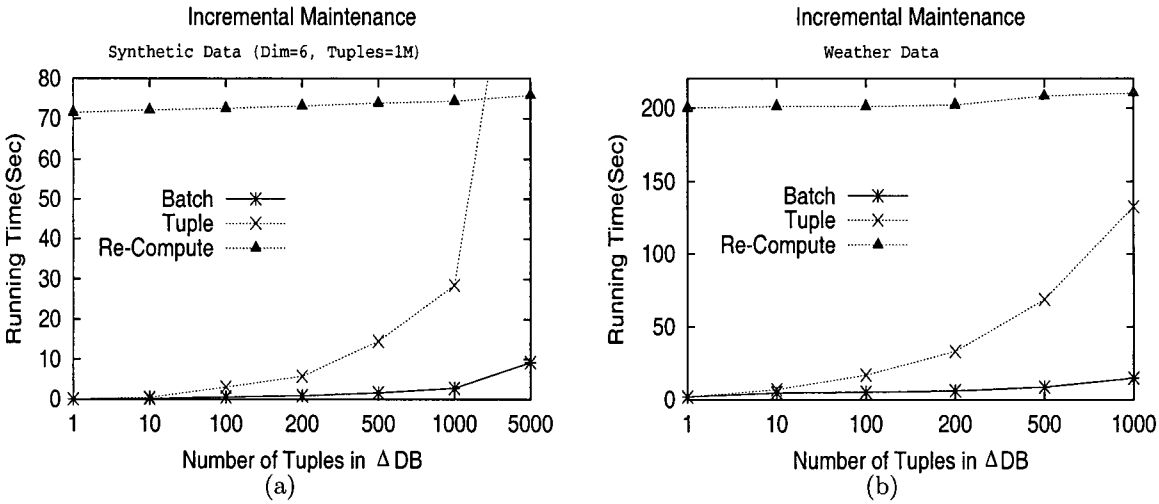


Figure 5.3: Performance of Batch Incremental Maintenance

QC-trees is extremely efficient. Therefore, these savings are passed on for the case of incremental maintenance.

Tuples in ΔDB	10k	20k	30k	40k	50k
Batch	11.70	18.70	25.51	30.02	35.90
Re-compute	73.00	73.00	74.00	74.00	75.00

Table 5.1: Running Time (Sec) of Batch Incremental Maintenance (Synthetic data: Dim=6, Tuples=1M)

Table 5.1 illustrates another experiment when the ΔDB size gets larger. This time, the ΔDB becomes $1\%, 2\%, 3\%, 4\%, 5\%$ of the original base table respectively. The result shows batch insertion is still much faster than re-computing the whole QC-tree.

5.4 Query Answering Performance

In the experiment, we present the performance of query answering using QC-trees and Dwarf.

First, we randomly generated 10,000 different point queries based on the synthetic data set. Figure 5.4(a) shows that the increase of cardinality reduces the efficiency of Dwarf. But QC-trees are insensitive to the increase. Then, we generated 1,000 point queries based on the weather data. Figure 5.4(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 5.4(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 is because: (i) when we query a cell c with n dimensions, Dwarf needs to access exactly n nodes. In QC-tree, typically fewer than n would be accessed. For a simple example, if we query $(*, *, d2)$, in Dwarf, 3 nodes have to be traversed while in QC-tree, only the root node and node $d2$ are visited; (ii) since the size of QC-tree is smaller than Dwarf, less I/O is needed in a QC-tree.

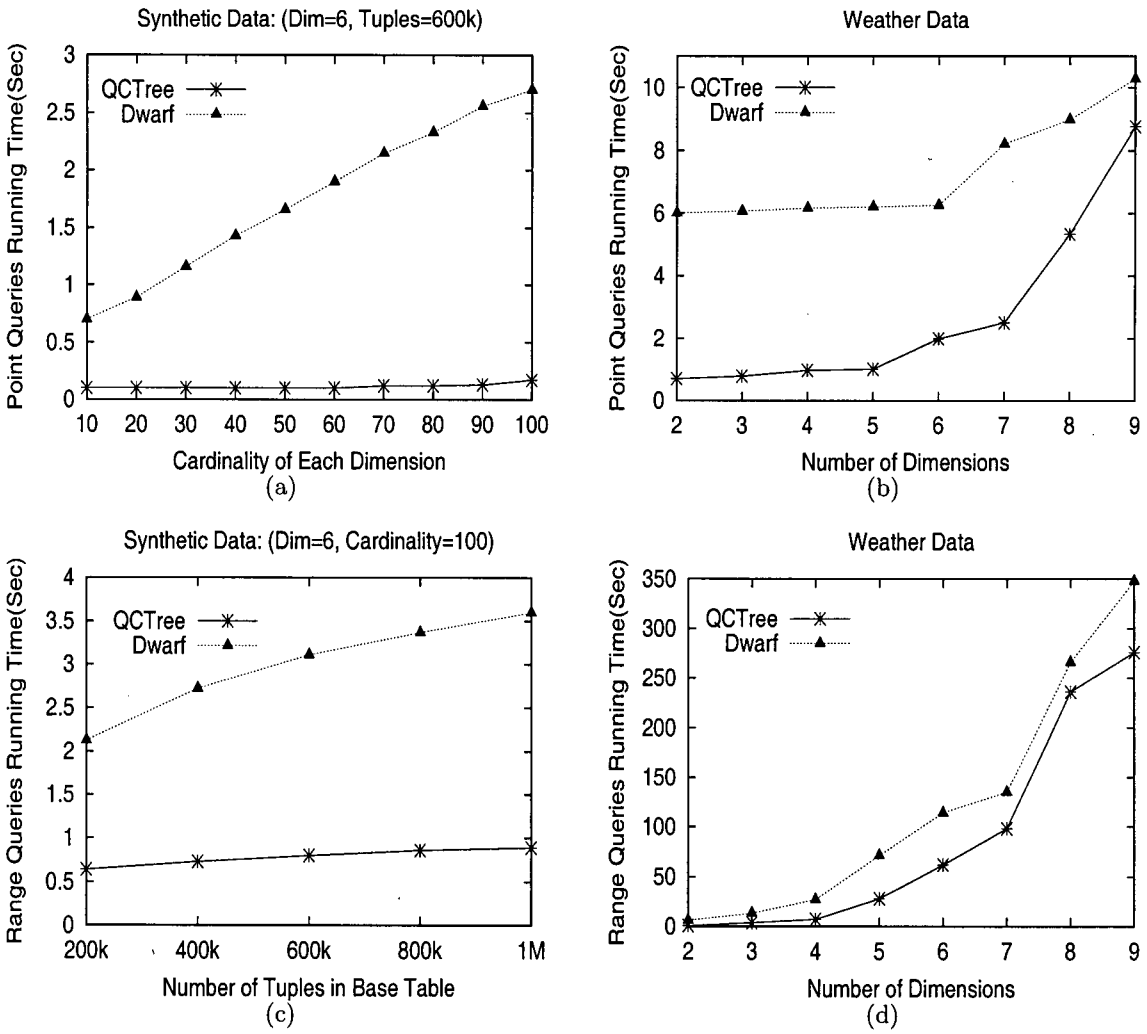


Figure 5.4: Performance of Query Answering

Chapter 6

SOQCET

In the previous chapters, we discussed quotient cubes and QC-trees, systematic approaches to achieve efficacious data cube construction and exploration by semantic summarization and compression. We also developed efficient algorithms for answering queries, and incremental maintenance of QC-trees against updates.

Based on our research achievements on quotient cubes and QC-trees, we implemented SOQCET (short for Semantic OLAP with Quotient CubE and Trees), a prototype data warehousing and OLAP system. In this chapter, we briefly describe the system design and function modules of SOQCET.

6.1 System Architecture

Figure 6.1 shows the structure of SOQCET. The whole system consists of two main parts, a QC-Engine and a user interactive interface. The QC Engine accepts commands generated from the user interface, produced operation and query plans, executes these plans against the data files, and return the answers.

When user issues a command, the command evaluator collects related informa-

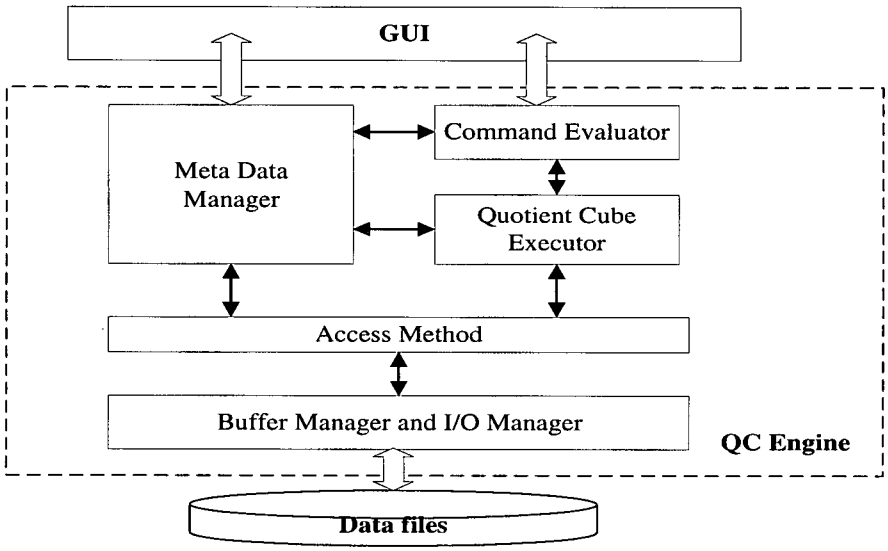


Figure 6.1: Architecture of SOQCET

tion from the meta manager and produces an execution plan that can be understood by the QC executor. The execution plan usually includes information such as command type(creating QC, updating QC or query answering), query type(point query, range query or iceberg query) and the translated command.

Meta manager organizes the data cube schemas including dimensions, measures, hierarchy levels and so on. Hierarchy information is stored in the dimension tables, which are modelled as a snowflake structure.

The QC executor accepts the execution plans, executes them using the corresponding algorithms against the data files and returns the answers back to GUI.

The access methods layer sits on top of the buffer manager, which brings data pages in from disk to main memory as needed in response to read requests.

The buffer manager maintains a buffer pool to bring pages from disk into main memory as needed and decide what existing page in main memory to replace. The replacement policy we used is **least recently used** (LRU).

The disk space manager works on the top of the data files, which is organized by pages in the disk. Buffer manager allocates, de-allocates, reads, and writes pages through this layer.

6.2 Functionalities

SOQCET demonstrates the techniques of building quotient cubes, maintaining quotient cubes against updates, using quotient cubes to answer various queries and conduct cube navigation and exploration.

Meta Data Viewer For each data cube that exists in the system, we can display the dimensions, measures, base table tuples, hierarchies and levels.

Quotient Cube Constructor Quotient cubes(QC-trees) can be created by specifying a base table along with the dimensions, measures and aggregate functions.

Cube Viewer It is a tool to help people doing Roll-up/drill-down navigation of data cubes. one can start from a particular cell and roll-up or drill-down along every level of each dimensions.

Query Generator It helps user write various OLAP queries, e.g. point query, range query and iceberg query. User can filter dimension values and specify measure thresholds to create arbitrary queries. A preliminary intelligent roll-up query can be generated by this tool as well. The query answers will be found by QC Engine and displayed in Cube Viewer.

Lattice Drawer A quotient cube can be represented by a quotient lattice with each node corresponding to a class. Roll-up/drill-down semantics are captured

between classes. This tool can draw a quotient lattice according to the classes that user specify. User can also click on each node(class) in the lattice and *drill down into* it and investigate the internal structure of the class. The internal cells of a class form a lattice as well.

Quotient Cube Maintainer One can incrementally maintain the quotient cube against the update of the base table.

Chapter 7

Conclusions and Future work

7.1 Conclusions

The quotient cube is a compact summary structure that preserves the essential semantics of a data cube. Motivated by the facts that quotient cubes are effective in reducing the size of a data cube and useful in supporting various OLAP queries (including advanced analysis like intelligent roll up), in this thesis work, we addressed several important questions about the quotient cubes which were not answered in [32].

Firstly, indexing a quotient cube and using it to answer complex queries was an open problem. In this thesis, we introduced sorted list to index the tabular representation(QC-table) of quotient cubes. A k-way search algorithm was proposed for efficient query answering using sorted list.

Secondly, a tabular implementation of quotient cubes with an additional index structure is not as efficient and compact as it could be. We provided a very compact data structure called QC-tree, which compresses the size of quotient cubes even

further. It is elegant and lean in the sense that the only information it keeps on quotient classes are their upper bounds and measures. The semantics of quotient cubes are retained in QC-trees. One can easily determine whether two cells are equivalent and what is the class upper bound of a cell. To efficiently construct a QC-tree, we proposed a depth-first based algorithm computing the quotient classes and building the QC-tree structure directly from the base table.

Thirdly, our query answering algorithms facilitate QC-trees to process various queries and conduct various browsing and exploration operations. In particular, basic queries over a data warehouse, i.e. point query, range query and iceberg query can be answered very efficiently.

Finally, issues of incremental maintenance of quotient cubes were not addressed prior to this work. We proposed fast algorithms for maintaining QC-trees against batch updates. The algorithms, which yield the correct QC-tree reflecting the update, are typically much faster than the re-computation and tuple-by-tuple update.

Empirical evaluations show that QC-trees attain substantial reduction of the data cube size. The performances of presented algorithms are impressive in terms of the running time of the QC-tree construction, query answering and incremental maintenance.

7.2 Future Work

As for future work, we plan to improve and explore some of the issues listed below.

7.2.1 Advanced Queries

We have shown quotient cubes can be used to answer advanced queries such as intelligent roll-up [22] in Section 3.4.

[3] proposed another interesting analysis tool called what-if queries to analyze various hypothetical scenarios. For example, an analyst may ask such a question as "*what will the average sales be for Location Vancouver if the sales situation in day 1 had happened in day 2?*". In general, this hypothetical scenario can be modeled as a list of hypothetical modifications on the base table or on the data cube.

Intuitively, efficient algorithms of incrementally maintaining a quotient cube should be helpful to answer what-if queries. We can "virtually update" the quotient cube to a "hypothetical" quotient cube based on the scenario. But problems such as how to handle virtual operations, which may relate to physical replication of the whole or part of the quotient cube, still lack good solutions.

7.2.2 Approximation

In reality, especially in strategic-level OLAP applications, precise answers to an aggregate query are not always necessary, where an approximated data cube can be applied to reduce the time/storage cost and obtain required answers.

Quotient cubes and QC-trees by themselves are lossless and semantic compression concepts. However, it would be interesting to investigate how approximated quotient cubes can enhance the compression while keeping the semantics.

7.2.3 Stream Data

A traditional OLAP system is based on the static, integrated and historical data to facilitate on-line analysis. Many real-time or dynamic systems such as financial applications, network monitoring, telecommunications systems often generate continuously arriving and potentially infinite data. The fundamental difference between stream data and warehouse data creates challenging research issues on extending the

successful OLAP system to stream data system.

With this new trend, it might be interesting to raise the question about whether and how a quotient cube or its variant could somehow support stream data.

Bibliography

[1] W.H. Inmon, Building the Data Warehouse. 1992

[2] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *VLDB'96.*

[3] A. Balmin, T. Papadimitriou, and Y. Papakonstantinou. Hypothetical queries in an olap environment. In *VLDB'00*

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

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

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

[7] Claudio Carpineto and Giovanni Romano:. Galois: An order-theoretic approach to conceptual clustering. In *ICML'93.*

[8] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *PODS'99.*

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In *ICDE'96.*

[10] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD'93.*

[11] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD'96.*

[12] Carlos A. Hurtado, Alberto O. Mendelzon, and Alejandro A. Vaisman. Maintaining data cubes under dimension updates. In *ICDE'99.*

[13] Carlos A. Hurtado, Alberto O. Mendelzon, and Alejandro A. Vaisman. Updating olap dimensions. In *DOLAP'99.*

[14] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS'95.*

[15] Alberto O. Mendelzon and Alejandro A. Vaisman. Temporal queries in olap. In *VLDB'00.*

[16] I.S. Mumick, D. Quass, and B.S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In Joan Peckham, editor, *SIGMOD'97.*

[17] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. 1996 Int. Conf. Parallel and Distributed Information Systems.*

[18] Dallan Quass and Jennifer Widom. On-line warehouse view maintenance. In *SIGMOD'97.*

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

[20] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk updates on the data cube. In *SIGMOD'97.*

[21] S. Sarawagi. Indexing OLAP data. *Bulletin of the Technical Committee on Data Engineering*, 20:36–43, 1997.

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

[23] J. Shanmugasundaram, U. Fayyad, and P. S. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *KDD'99.*

[24] Y. Sismanis, N. Roussopoulos, A. Deligiannakis, and Y. Kotidis. Dwarf: Shrinking the petacube. In *SIGMOD'02.*

[25] D. Sristava, S. Dar, H. V. Jagadish, and A. V. Levy. Answering queries with aggregation using views. In *VLDB'96.*

[26] J. S. Vitter, M. Wang, and B. R. Iyer. Data cube approximation and historgrams via wavelets. In *CIKM'98.*

[27] W. Wang, H. Lu, J. Feng, and J. X. Yu. 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, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD'97.*

[31] C.Hahn etal. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. cdiac.est.ornl.gov/ftp/ndp026b/SEP85L.Z,1994. In *SIGMOD'97.*

[32] L. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. In *VLDB'02.*

[33] L. Lakshmanan, J. Pei, and Y. Zhao. QC-Trees: An Efficient Summary Structure for Semantic OLAP. In *SIGMOD'03.*

[34] L. Lakshmanan, J. Pei, and Y. Zhao. SOCQET: Semantic OLAP with Compressed Cube and Summarization(demo paper) In *SIGMOD'03.*

[35] L. Lakshmanan, J. Pei, and Y. Zhao. Efficacious Data Cube Exploration by Semantic Summarization and Compression(demo paper) In *VLDB'03.*