Quotient Cube: How to Summarize the Semantics of a Data Cube

Laks V. S. Lakshmanan
U. of British Columbia
laks@cs.ubc.ca

Jian Pei
Simon Fraser U.
peijian@cs.sfu.ca

Jiawei Han
U. of Illinois
hanj@cs.uiuc.edu

Abstract

Partitioning a data cube into sets of cells with "similar behavior" often better exposes the semantics in the cube. E.g., if we find that average boots sales in the West 10th store of Walmart was the same for winter as for the whole year, it signifies something interesting about the trend of boots sales in that location in that year. In this paper, we are interested in finding succinct summaries of the data cube, exploiting regularities present in the cube, with a clear basis. We would like the summary: (i) to be as concise as possible, (ii) to itself form a lattice preserving the rollup/drilldown semantics of the cube, and (iii) to allow the original cube to be fully recovered. We illustrate the utility of solving this problem and discuss the inherent challenges. We develop techniques for partitioning cube cells for obtaining succinct summaries, and introduce the *quotient cube*. We give efficient algorithms for computing it from a base table. For monotone aggregate functions (e.g., COUNT, MIN, MAX, SUM on non-negative measures, etc.), our solution is optimal (i.e., quotient cube of the least size). For non-monotone functions (e.g., AVG), we obtain a locally optimal solution. We experimentally demonstrate the efficacy of our ideas and techniques and the scalability of our algorithms.

1 Introduction

The data cube [8] is one of the most influential operators in OLAP. It is instructive to classify works on issues surrounding it into two "generations". In the first generation, the main focus was to devise efficient algorithms for computing the cube – the full cube from scratch [8, 2, 20], choosing views to materialize under space constraints [11], handling sparsity [14], cube compression [16, 19, 17], approximation [3, 4, 18], and computing the cube under user-specified constraints [5]. In the second generation, researchers began to focus attention on extracting more "semantics" from a data cube. E.g., [15] studies the most general contexts under which observed patterns occur and [12] uses the cube structure to generalize association rules to a much broader context.

In this paper, continuing this semantic trend, we are interested in finding a succinct summary of the data cube, exploiting regularities present in the cube, with a clear basis. We would like the summary: (i) to be as concise as possible, (ii) to itself form a lattice preserving the rollup/drilldown semantics of the cube, and (iii) to allow the original cube to be fully recovered. To appreciate the problem, consider the relation `sales(Store, Product, Season, Sale)` (see Figure 1(a)), showing the sales of products in various outlets of, say Walmart. Here, `Sale` is a *measure* attribute, the remaining attributes being *dimensions*. Abbreviating the attributes T (`Store`), P (`Product`), S (`Season`), and L (`Sale`), a cube grouped by T, P, S and using an aggregate function $agg(L)$, is just a shorthand for the 8 group-by queries with each subset of T, P, S forming the group-by. Each group-by such as $\{T, P\}$ corresponds to a set of *cells*, described as tuples over the group-by attributes, e.g., (T = West 10th, P = boots), identifying those tuples in the base table `sales` that match these conditions. The set of all cells of a cube form a lattice (e.g., see Example 1 and Figure 1).

The lattice structure of a cube carries the important semantics of from which cells we can roll up or drill down to which other cells. Another important piece of semantics in a cube is the knowledge of which scenarios (or cells) have a similar behavior. For example, suppose in a cube over `sales`, we know the cells (T = West 10th, P = boots, S = winter) and (T = West 10th, P = boots) register the same average sales. It tells us something significant about the trend of boots sales in the West 10th store in the year

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

in question. We wish to regard these two cells equivalent. *Partitioning a cube into equivalence classes of cells with identical aggregate values helps expose its semantics better.* Besides, it can be used to "reduce" the cube by using classes of cells in place of cells, thus making it more amenable to user comprehension and exploration/analysis. Since the lattice structure of a cube is semantically important, in using partitions to reduce a cube, we should use those partitions that preserve the lattice structure: (i) one can drill down from a cell c to another cell c' in the cube exactly when one can do so from the class of c to that of c' in the reduced cube lattice; and (ii) one can drill down from a class C to a class C' in the reduced cube lattice exactly when one can drill down from *some* cell in C to *some* cell in C' in the original cube lattice.[1] As we will show, not all partitions lead to a (reduced) lattice structure, so we somehow need to choose the partitions right for constructing the reduced cube. The user can then browse the reduced cube (which preserves the original cube's semantics), and (s)he can always "click" on a class whenever (s)he needs to see its internal details. This is to be contrasted with general compression techniques, where the main concern is reducing the cube size, and there is no requirement to preserve the lattice structure of the cube.

The problem of computing the so-called reduced cube using appropriate partitions raises several challenges. Firstly, as we shall show, a blind approach of treating all cells with identical aggregate measure as equivalent, does not help generate a (reduced) lattice. Secondly, classical lattice theory tells us that an equivalence relation that is also a congruence (i.e., it respects the lattice structure) always generates a reduced lattice. Unfortunately, this does not help us, as *none of the known aggregate functions leads to congruences.* Thirdly, assuming we know how to partition a cube lattice right, we need efficient algorithms to compute the reduced cube. For instance, computing the quotient after first computing the cube is expensive and is not an option. We make the following contributions.

- We formalize desirable partitions on a cube lattice using a property called *convexity*. Intuitively, it requires equivalence classes to be free from "holes". We show that a fundamental notion of equivalence called *cover equivalence*, based on base tuples covered by cells, is convex, and formally establish its connection to finding partitions based on COUNT and SUM (Section 4).

- We capture equivalence relations preserving the rollup/drilldown semantics of a cube using a notion called *weak congruence* and show that on a cube lattice, an equivalence relation is a weak congruence if and only if it induces a convex partition (Section 5.1). Unlike for congruences, *all known aggregate functions admit weak congruences.*

- We show the following general result. For any monotone aggregate function f, the equivalence relation, defined based on equality of f-values and preserving connectivity, necessarily induces a convex partition, and hence is a weak congruence. We give examples of non-monotone functions for which such an equivalence relation may or may not induce a convex partition (Section 5.2).

- We formally define the *quotient cube lattice* of a cube w.r.t. a weak congruence, and propose it as a concise semantics-preserving summary of the data cube. We develop algorithms for computing it directly from the base table, for both monotone and non-monotone aggregate functions (Section 6).

- We conducted a comprehensive set of experiments on both synthetic and real data sets. Our results show the efficacy of the quotient cube in achieving considerable reduction in size, while preserving the rollup/drilldown semantics of the original cube lattice. They also validate the effectiveness and scalability of our algorithms (Section 7).

Section 2 motivates the problem and Section 3 summarizes the necessary technical background. Related work is discussed in Section 1.1. Section 8 briefly sketches issues like multiple aggregate functions, query answering, constraints, and incremental maintenance, while Section 9 summarizes the paper and describes future work. Proofs of results are omitted for space limitations, and can be found in the full paper [13].

1.1 Related Work

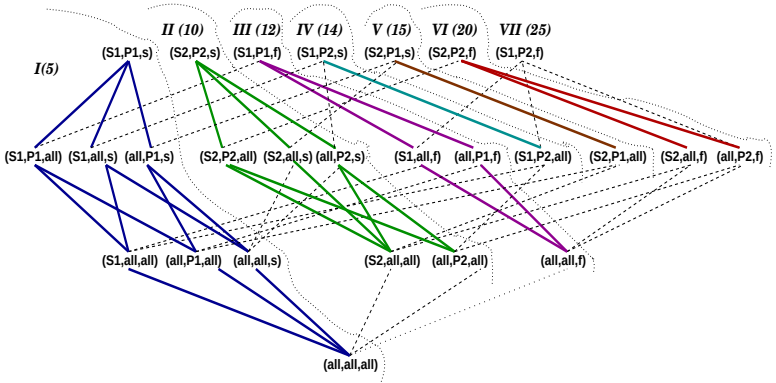
Works on compressing the cube size such as [16, 4, 7, 10] are of clear relevance to us. However, *our main goal is constructing an exact and concise summary of a cube that preserves the cube semantics and lattice structure*, as opposed to just compressing the cube size, which distinguishes it from all these works. The recent work on "condensed cube" by Wang et al. [19] is perhaps the closest in spirit to our work and we compare our work with this in some detail next.

The condensed cube approach uses two ideas of "single base tuple" compression and "projected single tuple " compression as a basis for compressing a data cube. This idea is similar in spirit to some of the goals of this paper. The focus of that paper is on one fixed aggregate function, COUNT (but extends to SUM). Indeed, for COUNT, as we show, the equivalence relation $\equiv_{\text{COUNT}}$ defined here is identical to cover equivalence, another fundamental notion defined in this paper. The *combination* of the two compression ideas above has a flavor similar to cover equivalence. A main technical focus of our paper is characterizing

[1] Clearly, it is important to ensure you cannot *also* drill down from some cell in C' to some cell in C, as that would destroy the lattice structure!

Store	Product	Season	Sale
S1	P1	spring	5
S1	P1	fall	12
S1	P2	spring	14
S2	P2	spring	10
S1	P2	fall	25
S2	P1	spring	15
S2	P2	fall	20

(a)



(b)

Figure 1: Base table `sales`, and its cube lattice and quotient classes (I–VII). Top element *false* not shown.

properties of partitions that generate reduced cubes that are guaranteed to be a lattice, a problem never addressed before. Secondly, by virtue of the theory developed in our paper, we are able to offer a rich set of semantic summarization techniques for most popular aggregate functions, such as MIN, MAX, SUM, COUNT, AVG, and TOP-k. The algorithms proposed for condensed cube in [19] include an exact exhaustive one and an efficient heuristic (and hence, suboptimal) one. By contrast, we provide efficient optimal algorithms for all monotone aggregate functions. Indeed, we compare their algorithm MinCube [19] with our optimal algorithm for computing quotient cube w.r.t. cover equivalence in Section 7. While our algorithms for non-monotone aggregate functions are in general suboptimal (but do satisfy local optimality), we are not aware of any previous algorithms for this case. Lastly, [17] proposes an extremely effective technique for compressing the cube size by means of a data structure called dwarf cube. Their contribution is orthogonal to ours since, unlike us, they are not concerned with semantics in summarization nor preserving lattice structure. It is also complementary to ours, since the structure sharing technique of [17] can be applied to any table, be it the full cube or a reduced cube such as our quotient cube.

2 Motivation

In this section, we illustrate the problem studied in the paper and then give a precise problem statement. Our first example illustrates the concept of a reduced cube that preserves cube semantics and lattice structure.

Example 1 (Illustrating Reduction) Consider Figure 1(a) and the data cube on table `sales` expressed by the query

```
SELECT   Store, Product, Season, MIN(Sale)
FROM     sales
CUBE BY  Store, Product, Season
```

Figure 1(b) shows the cube lattice,[2] with cells not present in the base table, such as $(\texttt{T} = \texttt{S2}, \texttt{P} = \texttt{P1}, \texttt{S} = \texttt{f})$ and (the top element) *false*, omitted. It has 26 distinct cells. The figure also shows a partition of the lattice into 7 disjoint equivalence classes I-VII (shown in solid lines in different colors and separated by curved dotted lines), with the property that all cells in a class have the same MIN(`Sale`) value (shown beside each class). Classes I-VII can indeed be represented as a lattice, viz., the reduced cube lattice, as in Figure 2, where a class C (e.g., II) is below another class C' (e.g., VI) exactly when we can drill down from some cell in C (e.g., $(\texttt{S2}, all, all)$) to some cell in C' (e.g., $(\texttt{S2}, all, f)$), in the original cube (see also Figure 1(b)). For each class, the figure shows the upper and lower bounds, guiding user's exploration. For the sake of completeness, in Figure 2, we also show the class *false*, the equivalence class of cells containing empty set of tuples. We call the reduced cube lattice the "quotient cube" and define it formally in Section 5. ■

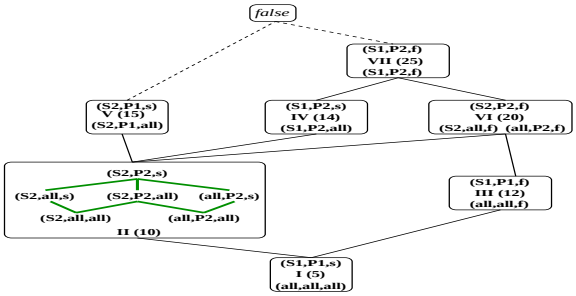


Figure 2: The quotient lattice of Figure 1(b), showing internal structure of class II.

On comparing Figures 1(b) and 2, we observe the following. (1) The quotient cube of Figure 2 is much smaller in size than the original cube of Figure 1(b). The reduction comes from adjacent cube cells shar-

[2] Following the convention of [5], we draw the lattice with $(all, ..., all)$ at the bottom. So we drill down from a cell c to a cell c' exactly when $c \prec c'$ according to the lattice ordering, and c' is above c in the lattice diagram.

ing a common aggregate value, a phenomenon that occurs frequently, as we will show experimentally in Section 7. (2) The quotient cube lattice of Figure 2 preserves the rollup/drilldown semantics inherent in the original cube. E.g., there is an upward edge from class II to class IV since we can drill down from the cell, say $(all, P2, all)$ in class II, to cell$(S1, P2, all)$ in class IV. In general, whenever we can drill down from a cell c to c' in Figure 1(b), we have an upward path from the class of c to that of c' in the quotient lattice of Figure 2. Class I is the bottom element since it contains (all, all, all) from which we can drill down to any cell. There is no path from III to V since one cannot drill down from any cell of III to any cell of V in the original cube. (3) The user may first want to browse the reduced cube lattice of Figure 2 to observe trends, and then (s)he can examine the internal structure of individual classes by "clicking" on them. The figure shows lower/upper bounds of all classes and the internal structure of clicked classes (class II). (4) These observations apply to other aggregate functions too.

Example 2 (Cube with SUM) Consider now the base table in Figure 3(a). Consider a cube query on this table with the aggregate SUM. The cube lattice has 16 cells with non-empty set of tuples. The quotient cube has just 6 classes. Figure 3(b)-(c) shows the cube and quotient cube lattices, with sum values shown beside classes. ■

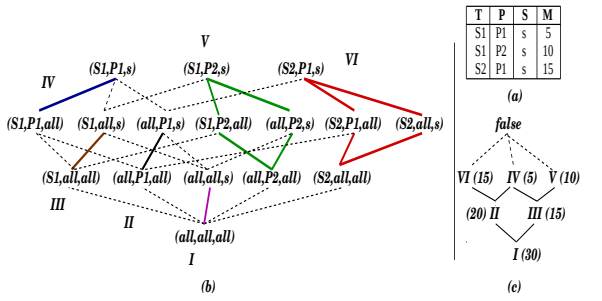


Figure 3: Base Table, Cube, and quotient cube for Example 2.

Many of the remarks on Example 1 apply to this example as well. The main differences are: (1) The aggregate functions are different. (2) There are two distinct classes, III and VI, with the same aggregate value 15. Had we merged these classes into one, the resulting class would have a locally disconnected structure, as can be seen from Figure 3(b). This fails to preserve the rollup/drilldown semantics of the original cube lattice: in the reduced cube (with III and VI merged), there is a drilldown path from II to VI (which is merged with III), and another from III to V; but in the original cube of Figure 3(b), we cannot drill down from any cell in II to any cell in V! Thus, in general, *partitioning cells solely on the basis of equality of aggregate values is undesirable.*

Problem Statement: In this paper, we are interested in the following problem. Given a cube, characterize a good way of partitioning its cells into classes such that: (i) the partition generates a (reduced) lattice, preserving the rollup/drilldown semantics of the cube lattice, and (ii) the partition is optimal in that it results in the fewest possible classes. We consider this problem both for aggregate functions such as MIN, COUNT, SUM,[3] etc. that are monotone and for those like AVG and SUM,[4] etc. that are not. We are interested in efficient algorithms for constructing the quotient cube w.r.t. the optimal partition, directly from the base table.

3 Background

In this section, we review the basic notions needed in the rest of the paper. A *lattice* is a partially ordered set $(\mathcal{L}, \preceq)$ such that every pair of elements in $\mathcal{L}$ has a least upper bound (lub) and a greatest lower bound (glb). Finite lattices can be represented using a lattice diagram with lattice elements as nodes such that there is an upward path from e to e' exactly when $e \preceq e'$. In this case, we call e' an ancestor of e and e a descendant of e'. We say e' is a parent of e (e a child of e') whenever e' is a nearest ancestor of e and $e \neq e'$. (Parents are not necessarily unique.) Every subset of a finite lattice $(\mathcal{L}, \preceq)$ has a lub and a glb. So $\mathcal{L}$ has a unique *top element* and a *bottom element.* Birkhoff [6] is a classic text on lattice theory.

We make use of lattices associated with a data cube instance. Let $b(A_1, ..., A_m, M)$ be a base table, A_i's being dimensions and M being the measure attribute. A cell is a tuple over the dimension attributes, where following [8], we allow the special value *all*. E.g., $(a_1, ..., a_m)$, $(all, a_2, ..., a_{m-1}, all)$, and $(all, ..., all)$ are all cells. The *cube lattice* is defined over the set of cells, where $(x_1, ..., x_m) \preceq (y_1, ..., y_m)$ provided whenever $x_i \neq all$, we have $y_i = x_i$, for all i. Note that for cells c, c', $c \preceq c'$ iff the set of tuples falling in cell c is a superset of those falling in c'. Following the convention of [11, 5], we assume cube lattices have $(all, ..., all)$ at the bottom and *false* as the top. Figures 1(b) and 3(b) show example cube lattices.

4 Cube Lattice Partitions

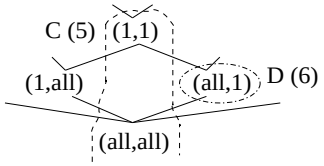
In Section 2, we saw that reduced cubes obtained via cube cell partitions, defined solely on the basis of aggregate measure equality, do not necessarily preserve the original cube's rollup/drilldown semantics. The next example illustrates yet another problem.

Example 3 (Bad Partitions) Figure 4(a)-(b) shows a base table and part of its cube lattice. Suppose we define cell equivalence solely based on equality of SUM values. The figure shows two classes C

[3] When all measure values are positive or all negative.
[4] With both positive and negative measure values.

A	B	M
1	1	5
1	2	−4
2	2	3
2	1	1

(a)



(b)

Figure 4: (a) Base table; (b) Partial cube lattice illustrating problem with SUM partitions.

and D, where C (SUM 5) contains cells (all, all) and $(1,1)$, and D (SUM 6) contains the only cell $(all, 1)$. This partition cannot yield a lattice, since we can drill down from C to D (e.g., from (all, all) to $(1, all)$), but also drill down from D to C (e.g., from $(all, 1)$ to $(1, 1)$. A similar example can be easily created keeping all measures positive, when the aggregate function is AVG. ∎

So how do we partition cube cells right? The following notion formalizes desirable properties of partitions that we are looking for.

Definition 1 [Convex Partitions] 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 $c \preceq c'$, then every intermediate cell c'', such that $c \preceq c'' \preceq c'$, necessarily belongs to C. In words, whenever C contains a pair of cells, it also contains all cells in between. We say $\mathcal{P}$ is *convex* provided all its classes are. ∎

All partitions in Examples 1 and 2 are convex. Intuitively, convexity means "holes" cannot exist in classes. The partition described in Example 3 is *non-convex*. We have the following proposition concerning equivalences w.r.t. count and sum values. For count, notice that we do *not* eliminate duplicates as per multi-set semantics.

Proposition 1 [Count and sum]: The equivalence relation defined solely on the basis of equality of count values is always convex. Suppose the domain of the measure attribute contains only non-negative (or only non-positive) values. Then equivalence defined solely on the basis of equality of sum values is convex. ∎

The proposition follows from the observation that whenever there is a cell c'' in between c and c', i.e., $c \preceq c'' \preceq c'$, c'' must contain all tuples that c' has, and for COUNT and SUM on positive measure, it cannot form a hole.

We say an equivalence class is connected if its local internal structure is a connected DAG. A partition is *connected* provided all its classes are. As seen earlier, partitions defined solely on the basis of equality of SUM values are not connected in general. We can define a fundamental partition of cells is one based on the set of tuples they contain. Specifically, say that two cells c and c' are *cover equivalent*, $c \equiv_{Cov} c'$, provided the set of tuples contained in those cells is the same. Note that cover equivalence of cells implies their equivalence w.r.t. *all* aggregate functions, including SUM and COUNT. The converse does not always hold (e.g., MAX or AVG). We can show:

Lemma 1 [Cover Partition]: Let $\mathcal{P}_{\text{Cov}}$ be the partition associated with the cover equivalence relation $\equiv_{Cov}$. Then $\mathcal{P}_{\text{Cov}}$ is necessarily convex. ∎

The key intuition is that if a cover equivalence class contained a hole, the set of tuples of the cell forming the hole would be bounded from above and from below by the sets of tuples contained in two other cells belonging to the class. But then these two cells must contain the same set of tuples. By Lemma 1, the partition shown in Example 2 (for SUM on positive measures) is indeed an example of a cover partition. The partition shown in Example 1 (for MIN) is different from cover partition. We now give a generic way to define equivalence w.r.t. an arbitrary aggregate function f, in a manner that preserves connectivity. By the *f-value* at a cell c, we mean the value of function f applied to the set of tuples contained in c.

Definition 2 [Connected Partitions] 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'. ∎

Recall that equivalence defined solely w.r.t. equality of sum or count values is *not* identical to cover equivalence. However, for $\equiv_{\text{SUM}}$ and $\equiv_{\text{COUNT}}$, defined as in Definition 2, we can show the following

Lemma 2 [Cover vs. COUNT/SUM equivalence]: The equivalence $\equiv_{\text{COUNT}}$ defined as above coincides with the cover equivalence $\equiv_{Cov}$. Furthermore, suppose the measure attribute domain contains only positive or only negative values. Then on a cube lattice, the equivalence $\equiv_{\text{SUM}}$ defined as above coincides with the cover equivalence. ∎

The above result testifies to the fundamental nature of cover partitions. For SUM, if the measure attribute contains both positive and negative values, it is possible to have a pair of parent and child cells, say c, c', such that the sets of tuples covered by them are not the same, yet, the associated SUM values are the same. This is true even when the values are all non-negative (or all non-positive), since tuples with measure value 0 can create the above situation.

5 Partitions Preserving Semantics

We shall formalize what it means for a partition to preserve the rollup/drilldown semantics of a cube lattice.

5.1 Weak Congruences

Let $\mathcal{L}$ be any cube lattice and $\equiv$ any equivalence relation on its cells. We say that $\equiv$ is a *congruence* provided for every $c, c', d, d' \in \mathcal{L}$, whenever we have $c \equiv c'$, $d \equiv d'$, and $c \preceq d$, we also have $c' \preceq d'$. Informally, a congruence on a lattice is an equivalence relation that respects its partial order in a strong sense. It induces a partition such that a cell in one class is a descendant of a cell in another iff this holds regardless of the choice of representative cells from the classes. While this is a nice property, *in the context of cube, no known natural partitions on the basis of aggregate value equality satisfy this property.* Indeed, even $\equiv_{Cov}$, a fundamental equivalence, is *not* a congruence. To remedy this situation, we propose the following:

Definition 3 [Weak Congruence] Let $(\mathcal{L}, \preceq)$ be any cube lattice and $\equiv$ any equivalence relation on its cells. We say that $\equiv$ is a *weak congruence* provided for every $c, c', d, d' \in \mathcal{L}$, whenever we have $c \equiv c'$, $d \equiv d'$, $c \preceq d$ and $d' \preceq c'$, we also have $c \equiv d$. ∎

Intuitively, a weak congruence is an equivalence relation that respects the lattice partial order in a weaker sense: it says whenever a pair of cells is related in a certain way according to this partial order, cells equivalent to them cannot be related in the opposite way, unless they are all equivalent. The following observation makes the intuition clearer:

- An equivalence relation on a (cube) lattice is a weak congruence iff the partition $\mathcal{P}$ induced by it is such that for any distinct classes $C, D \in \mathcal{P}$, whenever there are cells $c \in C$ and $d \in D$ with $c \preceq d$, then for every $d' \in D$ and $c' \in C$, we have $d' \not\preceq c'$.

In the sequel, we call any lattice partition satisfying the above property a WC partition. Weak congruences induce a "reduced" lattice defined as follows.

Definition 4 [Quotient cube lattice] Let $(\mathcal{L}, \preceq)$ be a cube lattice and let $\equiv$ be a weak congruence on $\mathcal{L}$. Then the *quotient cube lattice* is a lattice, denoted $(\mathcal{L}/\equiv, \preceq)$, such that the elements of $\mathcal{L}/\equiv$ are the equivalence classes of cells in $\mathcal{L}$ w.r.t. $\equiv$. For classes C, D in the quotient lattice, $C \preceq D$ exactly when $\exists c \in C, \exists d \in D$, such that $c \preceq d$ w.r.t. $\mathcal{L}$. ∎

It is straightforward to show that $\mathcal{L}/\equiv$ is indeed a lattice whenever $\equiv$ is a weak congruence. While every congruence is a weak congruence, the practical utility of weak congruences stems from the fact that unlike for congruences, many natural partitions on cube lattices correspond to weak congruences. The following theorem establishes a link between convexity and weak congruence.

Theorem 1 [Convexity and weak congruence]: Let $(\mathcal{L}, \preceq)$ be any lattice, $\equiv$ an equivalence relation on $\mathcal{L}$, and let $\mathcal{P}$ be the partition induced by $\equiv$. Then $\equiv$ is a weak congruence iff $\mathcal{P}$ is convex. ∎

The intuition behind this theorem is that a weak congruence makes it impossible for two equivalence classes of cells to be descendants of each other. Convexity, on the other hand, says there should be no hole in any class of the partition. Intuitively, convexity prevents a class from being both below and above another class, for otherwise, one of the classes would have a hole in it. It follows from this theorem that all examples of partitions that we have seen so far, including those defined w.r.t. MIN, SUM, COUNT, or w.r.t. cover equivalence are WC partitions. An exception is the equivalence given in Example 3, which is not a weak congruence. The significance of the WC property is that it allows us to *unambiguously* construct the quotient lattice of the cube lattice w.r.t. the WC partition. The quotient lattice preserves the rollup/drilldown semantics of the original cube lattice, while typically having a much smaller size.

Given a cube lattice, we seek WC partitions that are as coarse as possible, since coarser partitions yield fewer classes. Given two partitions $\mathcal{P}, \mathcal{P}'$ on any set, we say $\mathcal{P}$ is *coarser* than $\mathcal{P}'$, provided every class of $\mathcal{P}'$ is contained in some class of $\mathcal{P}$. In other words, classes of $\mathcal{P}'$ are not split between classes in $\mathcal{P}$. It is well known that the set of partitions on a fixed set with the "coarser than" ordering is a lattice itself and thus one can speak of partitions that are *maximally coarse* (abbreviated *maximal*) in a given collection of partitions, maximality being defined as for any partial order. Given a cube query with aggregate function f, we can now ask: (i) *is there a unique maximal WC partition on the cube lattice?* (ii) *how efficiently can we compute a maximal WC partition directly from the base table?* These questions are settled in the next sections.

5.2 Role of Aggregate Functions

It turns out answers to these questions tend to depend on the nature of the aggregate function. Recall that aggregate functions are defined over multi-sets of values. For multi-sets S, T, we write $S \subseteq T$ whenever every element of S also occurs in T and with no smaller multiplicity. Say that an aggregate function f is *monotone* whenever one of the following conditions holds: (i) for multi-sets S, T, whenever $S \subseteq T$, we have $f(S) \leq f(T)$; *or* (ii) for multi-sets S, T, whenever $S \subseteq T$, we have $f(S) \geq f(T)$. We have the following:

Theorem 2 [Monotone Aggregate Functions]: Let f be a monotone aggregate function and $(\mathcal{L}, \preceq)$ a cube lattice. Then there is a unique maximally coarse WC partition on $\mathcal{L}$. Furthermore, this partition coincides with the partition induced by the equivalence relation $\equiv_f$ defined in Definition 2. ∎

Theorem 2 is good news for cubes involving monotone aggregate functions. For such functions, it says forming the partition using Definition 2 is guaranteed to produce the unique coarsest WC partition on the

cube lattice. This has the following advantages. Semantically, this means there is a unique way to partition the cube lattice maximally while preserving cube semantics. As we will see in Section 6, it also has important implications for computational efficiency. Before leaving this section, we note that this theorem covers many aggregate functions of practical interest, including MIN, MAX, COUNT, SUM[5], and TOP-k. For AVG, and for SUM on positive and negative measure values, we have seen (Example 3) that an equivalence defined solely on the basis of equality of aggregate values is not convex and hence is not WC either. What can we say of equivalences w.r.t. SUM or AVG that are defined instead according to Definition 2? Our next example settles this question.

A	B	M
1	1	6
1	2	3
2	2	9

b_1

A	B	M
1	1	4
1	2	6
1	3	2

b_2

Figure 5: Two base tables. M is the measure attribute.

Example 4 (Non-monotone aggregate functions) For the base table $b_1(A, B, M)$ in Figure 5, note that the cells (all, all), $(all, 1)$, $(all, 2)$, and $(1, 1)$ all have the same average value, 6. Since they are connected, they are equivalent under $\equiv_{\text{AVG}}$, defined according to Definition 2. However, they are not equivalent to $(1, all)$, with an average value of 4.5, so $\equiv_{\text{AVG}}$ is not a weak congruence for this table. However, for table $b_2(A, B, M)$ in Figure 5, $\equiv_{\text{AVG}}$ generates the equivalence classes $I = \{(all, all), (1, all), (all, 1), (1, 1)\}$ with average 4, $II = \{(all, 2), (1, 2)\}$ with average 6, and $III = \{(all, 3), (1, 3)\}$ with average 2, and can be seen to be a weak congruence. Thus, for AVG, whether or not $\equiv_{\text{AVG}}$ is a weak congruence depends on the base table data! This is also true for SUM when both positive and negative values are allowed. In the full paper, we show there are (usually unnatural) non-monotone aggregate functions g for which $\equiv_g$ is always a weak congruence!

■

Observations: *While for a non-monotone aggregate function f, the partition induced by $\equiv_f$ may or may not be WC, we can always construct a WC partition.* E.g., for table $b_1(A, B, M)$ in Example 4 above, for which $\equiv_{\text{AVG}}$ is not a weak congruence. there are two[6] maximal convex partitions:

- $\mathcal{P}_1$ with classes – $I = \{(all, all), (all, 2)\}$, $II = \{(1, all)\}$, $III = \{(all, 1), (1, 1)\}$, and $IV = \{(2, all), (2, 2)\}$, with averages $6, 4.5, 6$, and 9, and

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

[6] It can be shown there are only two.

ClassId	A	B	C	MIN(M)	Desc
I	S1	P1	s	5	$(A\&B\&C)$
II	S2	P2	s	10	$(A\&C \vee B\&C)$

Figure 6: Representation of Quotient Cube of Figure 1, showing classes I and II.

- $\mathcal{P}_2$ with classes – $A = \{(all, all), (all, 2), (all, 1)\}$, $B = \{(1, all)\}$, $C = \{(1, 1)\}$, and $D = \{(2, all), (2, 2)\}$, with averages $6, 4.5, 6$, and 9.

Both $\mathcal{P}_1$ and $\mathcal{P}_2$ are different from the cover partition:

- $\mathcal{P}_{\text{Cov}}$ with classes: $\alpha = \{(all, all)\}$, $\beta = \{(1, all)\}$, $\gamma = \{(all, 1), (1, 1)\}$, $\delta = \{(2, all), (2, 2)\}$, and $\eta = \{(all, 2)\}$, with averages $6, 4.5, 6, 9$, and 6.

Of these, $\mathcal{P}_2$ is incomparable with $\mathcal{P}$, whereas $\mathcal{P}_1$ is strictly coarser than $\mathcal{P}_{\text{Cov}}$, and so preserves the information in the cover partition. *An important point is that the cover partition is the coarsest partition that at once captures equivalence w.r.t. all possible aggregate functions.*

5.3 Tabular Representation of Quotient Cube

A quotient cube of a base table $b(A_1, ..., A_m, M)$ can be represented by storing, for each class, its upper bound(s), and an indication how far it can be generalized while staying in the class. As an example, Figure 6 shows a representation for class II the quotient cube of Example 1. For class II, it shows its upper bound $(S2, P2, s)$, the MIN(M) value 10, and the fact that we can generalize $(S2, P2, s)$ to as far as $(all, P2, all)$ or $(S2, all, all)$, without leaving class II. That is, it can be expanded on dimensions A and C or on dimensions B and C, and this is captured by the descriptor value $(A\&C \vee B\&C)$. As Figure 2 shows, these are exactly the lower bounds of class II. This representation is precise in the sense that a cell belongs to a class iff it lies between its upper bound and one of its lower bounds. Thus, a quotient cube can be represented by storing each of its classes in this way.

6 Algorithms

In this section, we propose efficient algorithms for computing the quotient cube. An obvious algorithm for computing the quotient cube is as follows. First, compute the cube from the base table. Then, start an exhaustive search from the bottom element, $(all, ..., all)$, and begin identifying classes. At any stage, if a new cell has an aggregate value different from that of a class below it, assign it a new class. Otherwise, add it to the class if the addition will not create a hole in the class, a test that can be expensive to implement. This algorithm is clearly not practical. We can improve this algorithm by: (a) incorporating class formation with cube computation so we do not incur too much I/O, and (b) exploiting the Apriori-like property that whenever a cell (e.g., $(2, all)$) covers an empty set of tuples, any specialization (e.g., $(2, 1), (2, 2)$, etc.) of the cell

will cover empty set as well and hence can be pruned. We refer the reader to [13] for details. In Section 6.1, we develop a much more efficient algorithm based on depth-first search for computing the quotient cube lattice, for monotone aggregate functions. In Section 6.2, we develop another algorithm for non-monotone functions.

6.1 Depth-First Search

Algorithm 1 (Depth-first Search)

Input: base table B, monotone aggregate function f;

Output: Quotient cube;

Method:

Step 1: let $b = (all, ..., all)$; call $DFS(b, B, 0)$;

Step 2: merge those temp classes sharing some common upper bounds: if $\mathbf{C}_1$ and $\mathbf{C}_2$ share a same upper bound c, then merge them;
//e.g., if $MIN((a, b)) = MIN((a, all)) = MIN((all, b))$; the temp classes of the two cells (a, all) and (all, b) should be merged, since they share the upper bound (a, b).

Step 3: output classes, and their bounds, but only output true lower bounds, by removing lower bounds that have descendants in the merged class;
//e.g., when we process DFS on (all, b, all), it may in turn call a DFS on (all, b, c) and then form a temp class $\mathbf{C}_1 = \{(all, b, c), (d, b, c)\}$. Later, when the search branches to (all, all, c), it may form another temp class $\{(all, all, c), (d, b, c)\}$. The two classes share a common upper bound and so are merged. In the merged class, (all, b, c) is not a lower bound anymore and hence should be removed.

Function $DFS(c, B_c, k)$
// c is a cell and B_c is the corresponding partition of the base table;

Step 1: Compute aggregate of cell c by one scan of B_c; in the same scan, collect dimension-value statistics info for CountSort;
//similarly to the BUC algorithm;

Step 2: Compute the set of upper bounds $UB(c)$ of the class of c, by "jumping" to the appropriate upper bounds depending on the aggregate function used;
//see text for details.

Step 3: Record a temp class with lower bound c and upper bound(s) in $UB(c)$;

Step 4: for each upper bound d in $UB(c)$, do

[4.1] *if* there is some $j < k$ s.t. $c[j] = all$ and $d[j]! = all$, *then* such a bound has been examined before; do nothing.
//e.g. suppose when searching (all, all, c, all), we find that (a, all, c, d) is an upper bound. Then it must have been explored in the (a, all, all, all) branch.

[4.2] *else* for each $k < j \leq n$ 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 $DFS(d, B_d, j)$;

Step 5: return

Figure 7: A depth-first search algorithm.

The main idea of the algorithm is as follows. If the aggregate function used in the cube is a monotone function, then from a cell c, we can precisely determine all upper bounds of the class to which c belongs, and indirectly, all lower bounds of this class. Each upper and lower bound forms a border point of a class and together they completely characterize the class. Assuming we know how to "jump" from a cell to upper bounds of its class, note that such a jump is naturally implemented with a depth-first search. We next explain the idea of "jumping".

First, suppose we wish to compute the partition w.r.t. the cover equivalence $\equiv_{Cov}$. Suppose we are at a cell c. An upper bound c' of the class that c belongs to is determined as follows. c' agrees with c on every dimension where the value of c' is not all. For any dimension i where the value of c is all, suppose a fixed value, say x_i, appears in dimension i of all tuples in the partition B_c of the table that matches the conditions of cell c. Then the value of c' on every such dimension i is the repeating value x_i for that dimension. For cover equivalence, this is the only upper bound of the class of c that is reachable from c. E.g., if the base table contains just the tuples $(2, 1, 1, 2, 5)$, $(2, 1, 2, 2, 10)$, and $(1, 2, 2, 2, 20)$, then from the cell $c = (2, all, all, all)$ we can jump to the upper bound $(2, 1, all, 2)$ since the value 1 occurs in every tuple of dimension B, while 2 appears in every tuple of dimension D in the partition B_c of the base table (that contains the first two tuples). Consider a different aggregate function, say MIN. Let v be the MIN value for a cell c. Then every base table tuple contained in cell c which has value v for the measure M is indeed an upper bound of the class that c belongs to. MAX is handled similarly. When positive and negative measure values do not both occur, SUM is handled almost similarly to cover equivalence except tuples with measure value 0, if any, that are in a cell are ignored. In this case, the equivalence $\equiv_{\text{SUM}}$ are coarser than cover equivalence (see remarks following Lemma 2). Upper bounds corresponding to an equivalence relation w.r.t. TOP-k and defined to preserve connectivity as in Definition 2 are determined similarly to MAX except the top k values must be preserved when jumping to an upper bound.

The algorithm is given in Figure 7. The first part deals with cleanup that must be done in a post-processing stage. The second part implements depth-first search. It combines cube computation with equivalence class determination, accomplished by first determining class upper bounds and then jumping to them. Note that *cells in between bounds are never visited in the computation*, achieving considerable savings. We have included short examples beside some of the major steps.

Note that by virtue of the results established in Section 5.2, it follows that the quotient cube constructed by Algorithm 1 is optimal.

6.2 Non-monotone Case

The breadth-first search algorithm outlined in the beginning of Section 6 would work for any aggregate function but is not practical. We next propose a more

efficient algorithm for non-monotone functions. The idea is that cover equivalence always produces a convex partition with the property that all cells in a class have the same aggregate value. It just may not be a maximal such partition, as illustrated in Example 4. We start by computing the cover partition, using Algorithm 1. Then we merge equivalence classes, bottom up, whenever this won't create holes. As stated, it is not clear this leads to an efficient algorithm, since we need to test whether merging a class to the result of previously merging one or more classes would preserve convexity. It turns out that this test can be done efficiently by doing some additional bookkeeping while computing the cover partition.

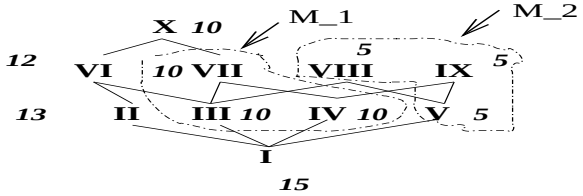


Figure 8: Illustrating Algorithm Class-Merge: part of quotient lattice w.r.t. cover partition, showing AVG values beside classes.

Figure 9 gives the algorithm. It proceeds bottom up, merging a cover class P with another class, or a set of previously merged classes, C, whenever P has the same aggregate measure value as C, P is a parent of some class in C, and all classes that are descendants of P and are ancestors of some class in C are themselves contained in C. Whenever P is found mergeable, we must also include all its child classes with the same measure. The reason is that if these child classes were not already merged to C at a lower level, it can only be because their sole connection with classes in C is established for the first time via P. Besides, if adding P to C will not violate convexity, adding its children cannot either. Figure 8 illustrates the algorithm, where a part of a quotient lattice w.r.t. cover partition is shown, with AVG values shown beside each cover class. Consider class III. While it shares the same aggregate value as IV, they cannot be merged right away since they are not connected. Consider class VII with the same aggregate value as III. It passes the ancestor-descendant test so they are merged. At this time, the child IV of VII can be merged too. Next, consider $C = \{III, VII, IV\}$ and $P = X$. Now, there is a class belonging to the intersection of ancestors of (classes in) C and descendants of P, namely class VI, which does not itself belong to C. So, the algorithm will not merge $P = X$ with C above. For efficiency reasons, the algorithm groups all cover classes at a given level by the aggregate measure value, (by hashing). In the algorithm, by the level of a previously merged class we mean the highest level of any cover class in it. We can show:

Lemma 3 [Correctness of Class-Merge] : Algorithm Class-Merge correctly computes a quotient lattice w.r.t. some maximal convex partition that is coarser than the cover partition. ■

Algorithm 2 (Class-Merge)

Input: base table B, non-monotone aggregate function f;

Output: Quotient cube w.r.t. a maximal convex partition;

Method:

1. Obtain quotient cube Q of B w.r.t. cover partition, using Algorithm DFS.
2. group classes at each level of Q by aggregate value, using hashing.
3. process lattice Q level by level, bottom up:
4. for each unprocessed class C at the current level {
   for each parent class P of C at the next higher level with the same measure value as C {
   - if $((desc(P) \cap anc(C) \subseteq C))$
   - add P as well as all children of P in C's measure group to C; in this case, mark all the latter children "processed"; replace C, P, and the above children by the new merged class; } }

Figure 9: Algorithm Class-Merge: $anc(C)$ denotes ancestors of any cover class in C.

7 Experimental Results

To evaluate the effectiveness of quotient cubes and the efficiency of quotient cube computation, we conducted a comprehensive set of experiments. In this section, we report a summary of our results. All experiments are conducted on a 450Hz PC with 256Mb main memory and running Microsoft Windows NT 4.0. All the programs are coded using Microsoft Visual C++ 6.0. We used both synthetic and real data sets to evaluate the effectiveness as well as run time of the algorithms. We measure effectiveness using a metric called *reduction ratio*, defined as the size of the quotient cube as a proportion of the original cube. *So the smaller the ratio the better.* In our experiments, we stored a quotient cube by explicitly storing both lower bounds and upper bounds for each class, instead of the more compact representation discussed in Section 5.3. This made the implementation easier but at the price of a less compact representation. Still, as the experimental results will show, we were able to achieve substantial reductions in most cases. The algorithms we tested are: QC_Cov, the algorithm for constructing optimal quotient cube w.r.t. cover partition, QC_MIN, the algorithm for constructing optimal quotient cube w.r.t. $\equiv_{MIN}$, and MinCube, the efficient heuristic algorithm for compressing cube, proposed in [19]. Note that the first two are really instances of Algorithm 1 proposed for general monotone aggregate functions, but the run time and reduction ratio will be quite different depending on the aggregate function used. In addition, for testing the effectiveness of quotient cube w.r.t. AVG,

we implemented a version of Algorithm 2. We refer to it as QC_AVG below.

We used the Zipf distribution [21] for generating synthetic data. It is a standard data set used for testing performance of algorithms under a variety of conditions. In addition, we also used the real dataset containing weather conditions at various weather stations on land for September 1985 [9]. This weather dataset has been frequently used in calibrating various cube algorithms, as well as most recently, for demonstrating the effectiveness of the MinCube algorithm of [19]. We ran a comprehensive set of tests on both data sets and performed numerous measurements, with each test repeated thrice. For brevity, we present only a representative set of results here.

7.1 Synthetic Data

First, consider monotone aggregate functions. We used SUM (on positive measures) and MIN as representative examples. Note that QC_Cov exactly corresponds to SUM on positive measures. MinCube is an approximation to the quotient cube w.r.t. SUM. We organize our findings as follows.

Effectiveness: Here, the goal is to gauge the extent of reduction ratio achieved by different algorithms. Of course, it makes no sense to compare QC_Cov (or MinCube) with QC_MIN as they are solving different problems. Yet, we plot the results in a common graph for brevity and also for a comparison of trends. We ran all three algorithms on a synthetic data set with Zipf distribution, with Zipf factor fixed at 2.0 and number of tuples held at 200 K, and the number of dimensions ranging from 2 to 10. The result is shown in Figure 10(a). With number of tuples fixed, as the dimensionality increases, the data gets sparse, so all algorithms achieve higher reduction as sparsity increases. However, QC_Cov achieves much better reduction ratio (about 3 times better) than MinCube. Even in 4 dimensions the reduction ratio is about 55%. Not surprisingly, QC_MIN achieves by far the best reduction ratio. E.g., we found that for 4 dimensions, it achieved a reduction ratio of 0.26%. The intuition is that the number of classes for QC_MIN is strongly correlated with (but not equal to)[7] the number of distinct measure values. In a parallel direction, we also ran tests by freezing number of tuples and dimensions and varying cardinality and observed a very similar behavior w.r.t. the resulting variation in sparsity. We do not show those results.

To measure the effect of data skew, we fixed the number of dimensions at 6, the number of tuples at 200 K, and varied the Zipf factor from 0 (uniform) to 3.0 (high skew) (Figure 10(b)). As data gets more skewed, all algorithms undergo an increase in their reduction ratio, but MinCube undergoes a rapid increase, while QC_Cov increases at a more modest rate and QC_MIN is quite stable. The explanation is that as data gets skewed, tuples tend to get concentrated into a smaller dense region of the data space with corresponding increase in the number of classes. The stability of QC_MIN can again be traced to the strong correlation of its number of classes with the distinct number of measure values.

Scalability: We also measured how effectiveness as well as algorithm run times scale up as the number of tuples goes up. We held the Zipf factor at 2.0, the number of dimensions at 6, and increased the number of tuples from 200 K to 1.5 million. Figure 10(c) shows the results. Reduction ratio goes up for all three algorithms. Again, MinCube shows the sharpest increase while QC_Cov has a modest growth rate and QC_MIN is least affected by the number of tuples. As the number of tuples increases, while other things are constant, data gets more dense. So, a rationale similar to the previous results explains these results too. Also, it is important to note that QC_Cov is an exact algorithm while MinCube is an approximate one.

The run times of the algorithms against number of tuples (under the same conditions) are shown in Figure 10(d), which also shows the run time of BUC (the Bottom-Up Cube algorithm of [5]). All algorithms have an essentially linear scalability. We found that QC_Cov is typically about twice faster than BUC (and about 6 times faster than MinCube). The reason is that QC_Cov computes only a small subset of cells compared to BUC which also translates into less output overhead. QC_MIN is slower than BUC (but still faster than MinCube) since the "jumping" idea, a key part of Algorithm 1 is more expensive for MIN than for Cover.

We tested QC_AVG on both synthetic and real data sets and report only the results on the real data sets.

7.2 Real data

We used the weather data set mentioned above. It contains 1,015,367 tuples (about 27.1 MB). The attributes with cardinalities are as follows: station-id (7,037), longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8), and brightness (2). We generated 8 datasets with 2 to 9 dimensions by projecting the weather dataset on the first k dimensions ($1 \leq k \leq 9$). We ran MinCube and QC_Cov on the full data set. Figure 11(a) shows the results. As expected, both algorithms achieve better reduction ratio as the dimensionality increases (recall the sparsity argument). But again, QC_Cov achieves a much sharper decrease in reduction ratio (i.e., much greater reduction) than MinCube and its trend is more stable. We also tested the algorithms for their run times and found a behavior consistent with the observations for the synthetic data sets above, so we suppress the results.

So far, we have mainly shown performance of quotient cube algorithms for monotone aggregate func-

[7] Because of duplicate tuples.

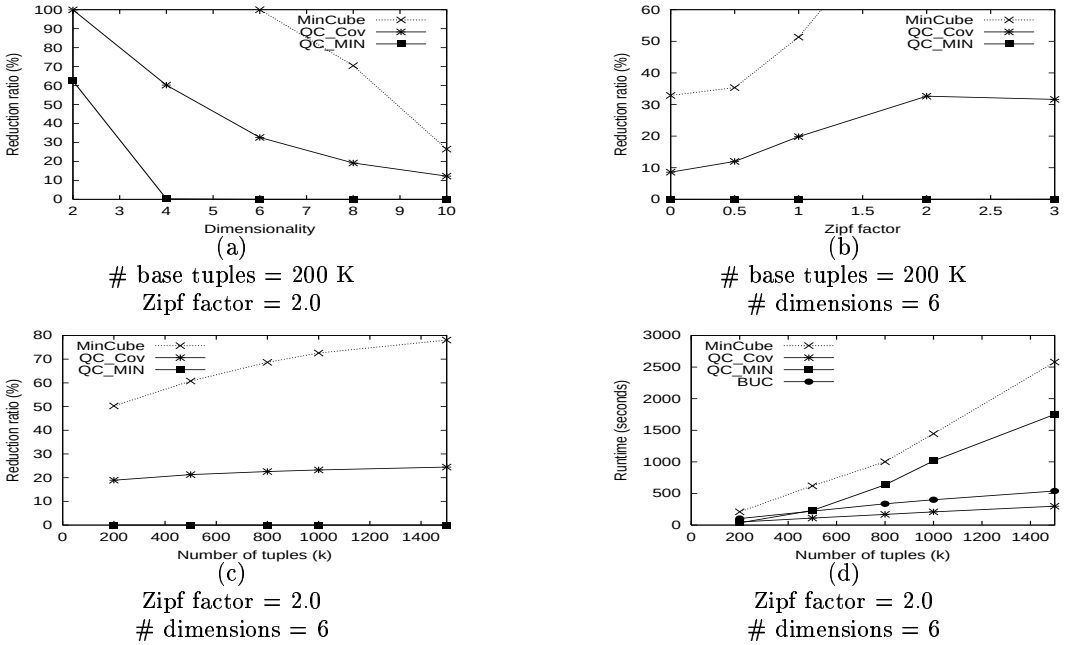


Figure 10: Evaluating reduction ratio and run time of algorithms: Zipf distribution.

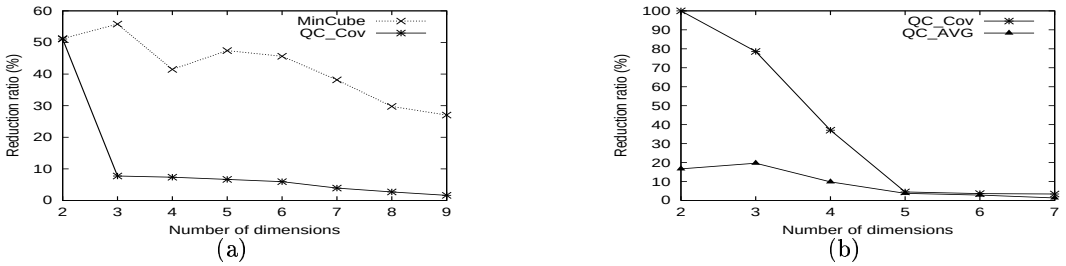


Figure 11: Evaluating reduction ratio of algorithms: the real weather data set.

tions. We also implemented and tested QC_AVG. We found that in general, QC_AVG does achieve substantial reduction (i.e., low reduction ratio) compared with QC_Cov. However, while it is true that the algorithm is indeed polynomial time, the current run time is still high. E.g., for a data set with 10 K tuples and 6 dimensions, it took about 68 seconds. Given this, we report below only results on reduction ratio achieved by QC_AVG on small samples (e.g., 10 K) of the real data set. We varied the dimensionality from 2 to 7. The results, depicted in Figure 11(b), show an interesting behavior. In general, QC_AVG achieves a much better reduction ratio than QC_Cov. However, as dimensionality increases, its additional "gain" on top of QC_Cov is almost negligible, when the number of tuples is fixed. We also observed the run time of QC_AVG goes up as the dimensionality increases as well. This raises an interesting tradeoff. At high dimensionalities, the user may thus prefer to use QC_Cov (and benefit from its very efficient computation) without losing much in the way of reduction ratio. Currently, designing more efficient algorithms for quotient cube w.r.t. AVG is an interesting open problem.

In sum, our experiments show that QC_Cov and QC_MIN are a highly scalable algorithms and achieve a substantially better reduction ratio than MinCube. They also showed the effectiveness of quotient cubes w.r.t. other functions such as MIN and AVG. These points were established on both synthetic and real data sets, and under a variety of conditions.

8 Discussion

The proposal, of quotient cube as a semantics-preserving compression technique, and of efficient algorithms for its computation, is the first step in a large project where we are investigating effective and efficient computation, exploration, analysis, and mining of data cube. In this section, we describe our vision of the project as well as why we believe quotient cube will be useful in practice.

Our current work focuses on the following aspects. (1) We have developed strategies for handling multiple aggregate functions, showing the notion of quotient cube is robust. As a generic point, one can always use the quotient cube w.r.t. the cover partition

for handling queries involving *any* aggregate functions. (2) Efficient algorithms for answering queries against a quotient cube. Our algorithms do not uncompress the cube, but rather rely on the ability to organize the bounds associated with classes intelligently to make query answering fast. (3) Algorithms for incremental maintenance of a quotient cube, based on a precise characterization of when classes must be split or merged in the face of updates. (4) Incorporating constraints (e.g., the well-known iceberg constraint). We have extended the notion of quotient cube to deal with constraints. The full paper [13] addresses these issues in detail. We are currently building a prototype system based on quotient cube.

A possible criticism (which applies to cube compression in general) is if an OLAP server chooses to use a partial materialization strategy for the cube a la [11], what is the relevance of quotient cube. There are two answers to this question. Firstly, it is straightforward to adapt our quotient cube algorithm so it only materializes the chosen views. Secondly, an effective compression technique opens up the possibility of using *full* materialization, as a competitive strategy, given the efficiency of query answering it buys. Furthermore, since it is semantics-preserving, it promotes user exploration much better than partial materialization might.

9 Summary and Future Work

We proposed the quotient cube as a succinct summary of a data cube lattice, preserving its rollup-drilldown semantics. It gives a quick and concise summary to the user based on which they may decide to explore different regions of the cube. Quotient cubes are based on partitions on the cube lattice and we characterized those partitions that lead to a reduced lattice structure. Monotone aggregate functions yield a unique maximal convex partition while non-monotone ones do not. For both cases, we developed efficient algorithms for computing the quotient cube and experimentally demonstrated their utility and effectiveness, as well scalability for the monotone case. In fact, our algorithms significantly outperform the previously proposed MinCube algorithm. There are several interesting questions. Designing scalable algorithms for quotient cube for AVG is important. Can we develop a notion of approximate quotient cube that buys substantial further compression for the price of a small sacrifice in accuracy? Computing quotient cube on streaming data is an important problem. Our ongoing research addresses some of these issues.

Acknowledgements: We are grateful to Hongjun Lu and his group, especially Wei Wang, for both giving us their source code and for their considerable help in clarifying some of the experimental details. Thanks to H.V. Jagadish and Raymond Ng for careful reading of the manuscript and critical feedback. Comments from the anonymous reviewers helped improve the presentation significantly. This work was supported by grants from NSERC, NCE/IRIS, and NSF.

References

[1] R. Agrawal & R. Srikant. Fast algorithms for mining association rules in large databases. *VLDB'94:487-499.*

[2] S. Agarwal, et al. On the computation of multidimensional aggregates. *VLDB'96:506-521.*

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

[4] D. Barbara & X. Wu. Using loglinear models to compress datacube. *WAIM'00:311-322.*

[5] K. Beyer & R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *SIGMOD'99:359-370.*

[6] G. Birkhoff, *Lattice Theory*, 2nd ed., New York, American Math. Soc. (Col. Pub. vol. 25), 1948.

[7] S. Geffner et al. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. *ICDE'99:328-335.*

[8] J. Gray et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. *ICDE'96:152-159.*

[9] C. Hahn et al. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. cdiac.est.ornl.gov/ftp/ndp026b/SEP85L.Z, 1994.

[10] C.-T. Ho et al. Partial-sum queries in data cubes using covering codes. *PODS'97:228-237.*

[11] V. Harinarayan et al. Implementing data cubes efficiently. *SIGMOD'96:205-216.*

[12] T. Imielinski et al. Cubegrades: Generalizing Association Rules. Tec. Rep., Rutgers U., Aug. 2000.

[13] L.V.S. Lakshmanan et al. Quotient Cube: How to summarize the semantics of a data cube. Tech. Rep. UBC, Nov.'01.

[14] K. Ross & D. Srivastava. Fast computation of sparse datacubes. *VLDB'97:116–125.*

[15] G. Sathe & S. Sarawagi. Intelligent Rollups in Multidimensional OLAP Data. *VLDB'01:531-540.*

[16] J. Shanmugasundaram et al. Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions. *KDD'99:223-232.*

[17] Y. Sismanis et al. Dwarf: Shrinking the Petacube. *SIGMOD'02.*

[18] J. S. Vitter et al. Data cube approximation and histograms via wavelets. *CIKM'98:96-104.*

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

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

[21] G.K. Zipf. *Human Behavior and The Principle of Least Effort.* Addison-Wesley, 1949.