

# Incremental Maintenance of Quotient Cube for Median

Cuiping Li<sup>\*</sup>  
Dept. of Computer Science,  
Renmin University of China  
Beijing 100872, China

Gao Cong,  
Anthony K. H. Tung<sup>†</sup>  
Dept. of Computer Science  
Natl University of Singapore  
S'pore 117543, Singapore

Shan Wang  
Dept. of Computer Science,  
Renmin University of China  
Beijing 100872, China

## ABSTRACT

Data cube pre-computation is an important concept for supporting OLAP(Online Analytical Processing) and has been studied extensively. It is often not feasible to compute a complete data cube due to the huge storage requirement. Recently proposed quotient cube addressed this issue through a partitioning method that groups cube cells into equivalence partitions. Such an approach is not only useful for distributive aggregate functions such as SUM but can also be applied to the holistic aggregate functions like MEDIAN.

Maintaining a data cube for holistic aggregation is a hard problem since its difficulty lies in the fact that history tuple values must be kept in order to compute the new aggregate when tuples are inserted or deleted. The quotient cube makes the problem harder since we also need to maintain the equivalence classes. In this paper, we introduce two techniques called **addset data structure** and **sliding window** to deal with this problem. We develop efficient algorithms for maintaining a quotient cube with holistic aggregation functions that takes up reasonably small storage space. Performance study shows that our algorithms are effective, efficient and scalable over large databases.

**Categories and Subject Descriptors:** H.2.8 [Database applications]: Data mining

**General Terms:** Algorithms

**Keywords:** Data Cube, Holistic Aggregation.

## 1. INTRODUCTION

Data cube pre-computation is an important concept for supporting OLAP(Online Analytical Processing) and has been studied extensively. It is often not feasible to compute a complete data cube due to the huge storage requirement. Recently proposed quotient cube [10] addressed this issue through a partitioning method that groups cube cells into equivalence classes, thus reducing computation time and storage overhead.

The intuition behind quotient cube is that many cube cells in a data cube are in fact aggregated from the same set of

base tuples (and thus have the same aggregate value). By grouping these cube cells together, the aggregate values of these cells need only to be stored once. This gives substantial reduction in cube's storage while preserving cube's roll up and drill down semantics.

Consider a data warehouse schema that consists of three dimensions: A, B, and C. The schema includes a single measure M. The domain values for these three dimension attributes are  $\text{domain}(A)=\{a_1, a_2, a_3\}$ ,  $\text{domain}(B)=\{b_1, b_2, b_3\}$ ,  $\text{domain}(C)=\{c_1, c_2, c_3, c_4\}$ . We use relation R in Figure 1(a) as the base relation table. A data cube lattice expressed by the following query:

```
SELECT A, B, C, SUM(M)
FROM R
CUBE BY A, B, C
```

has 22 distinct cube cells as shown in Figure 1 (b). The cube lattice can be partitioned into 9 disjointed equivalence classes, each represented by a circle as shown in Figure 1(b). All cells in an equivalence class are aggregated from the same set of the base relation tuples. For example, cells  $a_1$  and  $a_1b_1$  are both aggregated from tuple 3 and tuple 4, thus they are grouped together and their aggregate values can be stored only once. The nine equivalence classes in Figure 1(b) can be represented with another lattice (so called quotient cube) shown in Figure 1(c), where a class  $C$  (e.g.  $C_1$ ) is above another class  $C'$  (e.g.  $C_2$ ) exactly when we can drill down from some cell in  $C$  (e.g.  $b_1$ ) to some cell in  $C'$  (e.g.  $a_1b_1$ ). Thus the roll up and drill down semantics among the cube cells are preserved.

Such an approach is not only useful for distributive aggregate functions such as SUM but can also be applied to holistic aggregate functions like MEDIAN which will require the storage of a set of tuples for each equivalence partition. Unfortunately, as changes are made to the data sources, maintaining the quotient cube is non-trivial since the partitioning of the cube cells must also be updated. If some tuples are inserted to the base relation R, some equivalence classes need to be updated or split.

Existing proposals for incremental quotient cube maintenance [11] are not able to maintain a quotient cube with holistic aggregation functions such as MEDIAN and QUANTILE. Incremental updating holistic aggregations is difficult since in the case of changes to base tuples, the new aggregate value cannot be computed incrementally based on the previous aggregate value and the new values of the changed tuples. In this paper, we will propose a solution for the incremental maintenance of a quotient cube with holistic aggregation. We identify our contributions as follows:

- We introduce the concept of **addset data structure** that is able to substantially cut down the size of storage space required for each equivalence class. In order

<sup>\*</sup>Work done while the author was visiting National University of Singapore

<sup>†</sup>Contact Author: atung@comp.nus.edu.sg

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD'04, August 22–25, 2004, Seattle, Washington, USA.  
Copyright 2004 ACM 1-58113-888-1/04/0008 ...\$5.00.

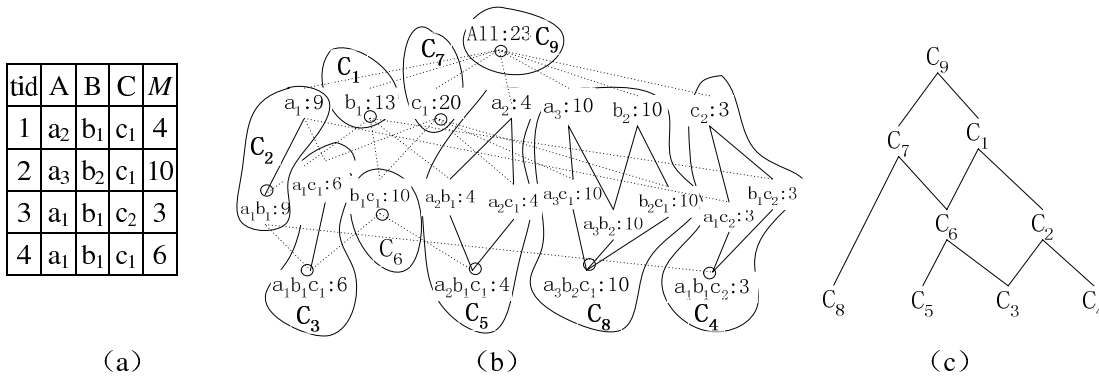


Figure 1: (a)Base table  $R$ ; (b)Cover partition on data cube cells; (c)Quotient cube  $\Phi$

to achieve a compromise between time and space, we enrich the addset data structure with both materialized nodes and pseudo nodes to represent equivalence classes.

- We propose a novel **sliding window technique** to efficiently recompute the updated MEDIAN for all equivalence classes.
- By relating quotient cube to well established theory of Galois lattice [6, 4, 5], we derive principles of maintaining quotient cube. With the principles, addset data structures and sliding window technique, we develop efficient algorithms for maintaining a quotient cube with holistic aggregation MEDIAN that takes up reasonably small storage space.
- We conduct a comprehensive set of experiments on both synthetic and real data sets. Our results show that our maintenance algorithms are efficient in both space and time.

The remaining of the paper is organized as follows. Section 2 gives the background information of our study. Section 3 introduces our techniques for maintaining holistic aggregation function median. Section 4 presents two incremental maintenance algorithms for holistic aggregate function MEDIAN. A performance analysis of our methods is presented in section 5. We give other related works in section 6 and make some conclusions in section 7.

## 2. BACKGROUND

In this section, we will provide the necessary background for discussion in the rest of this paper. We first define some notations in section 2.1 and then briefly explain the maintenance principles of quotient cube in section 2.2.

### 2.1 Notation Definitions

The base relation of a data warehouse is composed of one or more *dimensions*  $D_1, \dots, D_n$  and a *measure*  $M$ . We denote the domain of a dimension  $D_i$  as  $dom(D_i)$

**DEFINITION 1 (RELATIONAL TUPLE).** A tuple  $t$  in a base relation  $R$  of a data warehouse has the form  $t=(tid, dvalue, m)$ , where  $tid$  is the unique tuple identification of  $t$ ,  $dvalue \in dom(D_1) \times dom(D_2) \dots \times dom(D_n)$  is the dimension value set of  $t$ , and  $m$  is the measure value of  $t$ . We use  $t.tid$ ,  $t.dvalue$ , and  $t.m$  to represent each component of  $t$  respectively.  $\square$

As an example, for tuple  $t=(2, a_3b_2c_1, 10)$ ,  $t.tid$  is 2,  $t.dvalue$  is  $\{a_3, b_2, c_1\}$ , and  $t.m$  is 10. For convenience, we use  $a_3b_2c_1$  to represent the dimension value set  $\{a_3, b_2, c_1\}$ .

**DEFINITION 2 (CELL).** A cell,  $c$ , in a data cube is a tuple over the dimension attribute domains where the special value “all” is allowed i.e.  $c \in (dom(D_1) \cup \text{“all”}) \times \dots \times (dom(D_n) \cup \text{“all”})$ . A cell  $c = \{d_1, \dots, d_n\}$  is said to be more **general** than another cell  $c' = \{d'_1, \dots, d'_n\}$  if for all  $i$ , either  $d_i = d'_i$  or  $d_i = \text{“all”}$ .  $\square$

The special value “all” in this case represents a don’t care condition in a particular dimension. For clearer representation, we will assume it to be the default whenever a dimensional value is missing. For example the cell  $\{a_1, \text{all}, c_1\}$  will be represented as  $\{a_1, c_1\}$  (or  $a_1c_1$  to simplify further).

**DEFINITION 3 (MATCHING).** A tuple  $t$  is said to **match** a cell  $c$  if  $t.dvalue$  matches  $c$  in all dimensions except for those dimensions in which the value for  $c$  is “all”. Given a set of cells,  $C$ , a tuple  $t$  is said to match  $C$  if it matches all the cells in  $C$ .  $\square$

For example, the tuple  $t=(2, a_3b_2c_1, 10)$  matches both the cell  $a_3b_2$  and also the set of cells in Class  $C_8$  of Figure 1(b).

All the possible cells in a data cube can be organized into a **lattice** and each cell is represented with an element of a lattice. A lattice is a partially order set  $(\mathcal{L}, \preceq)$  in which every pair of elements in  $\mathcal{L}$  has a Least Upper Bound(LUB) and a Greatest Lower Bound(GLB) within  $\mathcal{L}$ .

Formally, a partially ordered set is defined as an ordered pair  $P = (\mathcal{L}, \preceq)$ , where  $\mathcal{L}$  is called the ground set of  $P$  and  $\preceq$  is the partial order of  $P$ . For the case of data cube, the set of cells are in the set  $\mathcal{L}$  and  $\preceq$  will be defined in Definition 5. Since the lattice theories are well studied, we can borrow some ideas from lattice to design incremental maintaining algorithms by relating data cubes with lattices.

**DEFINITION 4 (LUB AND GLB).** Given a set of elements  $E'$  in a lattice  $(\mathcal{L}, \preceq)$ , the least upper bound(LUB) of  $E'$  is an element  $u \in \mathcal{L}$  such that  $e \preceq u$  for all  $e \in E'$  and there exists no  $u'$  such that  $e \preceq u'$  for all  $e \in E'$  and  $u' \preceq u$ . Likewise, the greatest lower bound(GLB) of  $E'$  is an element  $l \in \mathcal{L}$  such that  $l \preceq e$  for all  $e \in E'$  and there exists no  $l'$  such that  $l' \preceq e$  for all  $e \in E'$  and  $l \preceq l'$ . LUB and GLB are unique.  $\square$

If  $\mathcal{L}$  is finite, then  $(\mathcal{L}, \preceq)$  is a finite lattice. A finite lattice can be represented using a lattice diagram in which elements in  $\mathcal{L}$  are nodes and there is an edge from a node representing an element  $e$  to a node representing another element  $e'$  iff  $e \preceq e'$  and there exists **no** other element  $e''$  such that  $e \preceq e'' \preceq e'$ .

**DEFINITION 5 (CUBE LATTICE).** A cube lattice for a data cube is a finite lattice  $(\mathcal{L}, \preceq)$  in which  $\mathcal{L}$  contains all

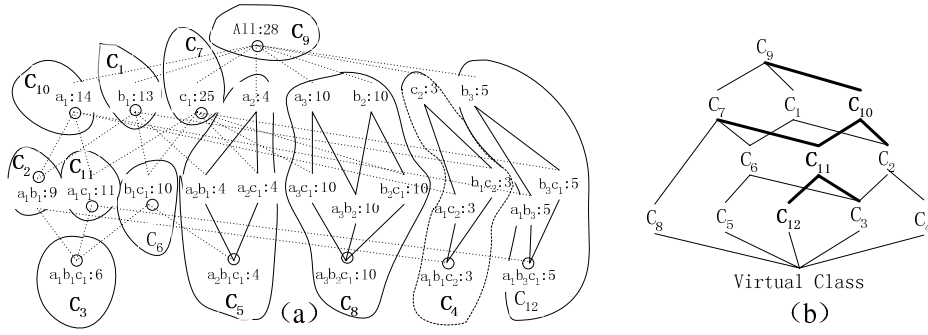


Figure 2: The quotient cube  $\Phi'$  after insertion of  $(5, a_1b_3c_1, 5)$

possible cells in the data cube plus a special cell “false” (the least general cell) and two cells  $c'$ ,  $c$  satisfies  $c \preceq c'$  iff  $c$  is more general than  $c'$  or  $c'$  equals to “false”.  $\square$

An example of a cube lattice is shown in Figure 1(b)(cell “false” is not shown for clarity). For example, the cell “ALL”  $\preceq a_1$  and there is an edge from cell “ALL” to cell  $a_1$ . We can now formally define the concept of an equivalence class of cells in a cube lattice.

**DEFINITION 6 (EQUIVALENCE CLASS).** A set of cells in a cube lattice is said to belong to the same equivalence class,  $C$ , if

1. Given any two cells  $c, c'$  in  $C$  which satisfy  $c \preceq c'$ , any intermediate cell  $c''$  satisfying  $c \preceq c'' \preceq c'$  will also be in  $C$ .
2.  $C$  is the **maximal** set of cells that are matched by the same set of tuples.

$\square$

Since all cells in an equivalence class are matched by the same set of tuples, it is possible to find a unique cell which is the upper bound for the whole class by simply selecting those dimension values that are the same for every tuple in the matching set. For example, all cells in Class  $C_2$  of Figure 1(c) are matched by tuples 3 and 4 in Figure 1(a) and thus the upper bound of Class  $C_2$  is  $a_1b_1$  which is common to all the tuples.

In general, a class  $C$  can be represented by a structure,  $C=(upp, m)$ , in which **upp** is the upper bound of the class and **m** is the aggregated value for the set of tuples that match the cells in  $C$ . We use  $C.upp$  and  $C.m$  to represent each component of  $C$  respectively. For example, class  $C_2$  in Figure 1(c) has the form of  $C_2=(a_1b_1, 9)$ .

## 2.2 Maintenance of Quotient Cube

In this section, we will revise the underlying principles for maintaining the equivalence classes in a quotient cube [11]. We observe that the cube lattice that is formed from the upper bounds of all equivalence classes in the quotient cube in fact has similar structure with a Galois lattice [6, 5]. Because of space limitation, we will not explain the relationship between the Galois lattice and the QC-tree [11] here.

We denote the set of equivalence classes in the original quotient cube as  $\Phi$ , and the new set of equivalence classes as  $\Phi'$ . In this section, we only consider the case that the incremental update is composed of a single tuple  $t$ . We will extend our method for bulk update with a set of new tuples later in Section 4.

New tuple  $t$  can affect an equivalence class in  $\Phi$  in several ways. First, it can cause the aggregate value of the equivalence class to change without affecting the partitioning of the lattices. Second, it might cause the equivalence class to be split, creating some new equivalence classes. The final possibility is that the equivalence class might not be affected at all. When a new tuple matches the upper bound of an equivalence class, the new tuple  $t$  will cause the aggregate value of the equivalence class to be changed. More importantly, we know that the tuple will match every cell in the equivalence class since the class’s upper bound is the most specific cell in the whole equivalence class. The equivalence class in this case needs not to be split.

**PROPOSITION 1 (VALUE MODIFIED CLASS).** Given an equivalence class  $C$  in  $\Phi$ , if a new tuple  $t$  matches  $C.upp$ , then  $C$  needs not to be split but  $C.m$  must be modified.  $\square$

For example, in Figure 2, because  $c_1$  matches  $a_1b_3c_1$  (the new inserted tuple), we update the aggregation sum of  $C_7$  from 20 to 25.

When  $t$  matches only a certain portion of  $C.upp$ , i.e.  $t$  can only match a portion of the cells in  $C$ ,  $C$  must be split into two portions, one in which all cells match  $t$  and one in which all cells do not. A new tuple  $t$  affects a class  $C$  only if there is some **intersection** between  $t$  and  $C.upp$ .

**DEFINITION 7 (INTERSECTION).** We say that a tuple  $t$  has an intersection with (or intersects) a class  $C$  if  $t$  does not match  $C$  but  $t.dvalue \cap C.upp \neq \emptyset$ . Given  $\Phi$ , we use  $intersect(\Phi, t)$  to denote the set of classes in  $\Phi$  that intersect  $t$ .  $\square$

**PROPOSITION 2 (NEW CLASS GENERATOR).** Given a new tuple  $t$ , an existing equivalence class  $C$  must be split if (1)  $C$  intersects  $t$  and  $C$  is the class that contributes the  $GLB\{Y|Y=C'.upp \cap t\}$ ,  $C' \in \Phi$ ; and (2) there does not exist any class  $C'' \in \Phi$  such that  $C''.upp = t \cap C.upp$ ; If these two conditions are satisfied, we call  $C$  a new class generator since the splitting will result in a new equivalence class.  $\square$

The first condition of Proposition 2 ensures that given all classes which generate the same upper bound for the new class  $C_n$ , the one that is the most general (i.e. the GLB) will be the new class generator. For example in Figure 2, given the new tuple  $t = (5, a_1b_3c_1, 5)$ , we have  $C_2 \cap t = C_4 \cap t = \{a_1\}$ . Since  $C_4$  is an upper bound of  $C_2$ ,  $C_2$  will become a new class generator for  $t$  if it satisfies the second condition of Proposition 2. Note that  $C_4$  will definitely not be split since **none of the cells in  $C_4$  matched  $t$** . This is to be expected because  $C_4$  is more specific than  $C_2$  and since even  $C_2.upp$  can not match  $t$ , all cells in  $C_4$  will also not match  $t$ .

Explaining the second condition in Proposition 2 is more simple. Since  $t \cap C$  represents the upper bound of the potential new class, there is no need to generate a new class if such an equivalence class already exists as indicated by the existence of a class  $C'' \in \Phi$  such that  $C''.upp = t \cap C.upp$ .

Note that in the situation in which  $t$  appears in  $R$  for the first time, i.e.,  $t$  has no duplication in  $R$ , in this case, it is a new class itself but there will be no generator for it. To ensure a generator for every new class, a virtual class is added in  $\Phi$ . The upper bound of the virtual class are the union of all the possible dimension value set,  $DV$ .

The split operation of a class is defined as follows:

**DEFINITION 8 (SPLIT OPERATION).** *Given a generator class  $C_g$ ,  $C_g \subseteq \Phi$  and a new tuple  $t$ , a split operation on  $C_g$  based on  $t$  generates a new class  $C_n$  and a modified generator  $C'_g$  as follows:*

- $C_n.upp = C_g.upp \cap t.dvalue$
- $C_n.m = Agg(C_g.m, t.m)$ <sup>1</sup>
- $C'_g.upp = C_g.upp$
- $C'_g.m = C_g.m$

□

The last proposition involves a simple category of equivalence classes that neither match or intersect the new tuple  $t$ .

**PROPOSITION 3 (DUMB CLASS).** *If an equivalence class  $C_d$  in  $\Phi$  is neither a modified class nor a generator, there is no need to change  $C_d$ . We call  $C_d$  as a dumb class.* □

Propositions 1-3 lay the foundation for maintaining a quotient cube. Updating the value of distributive aggregation is relatively simple. The new aggregate value can be computed incrementally based on the previous aggregate value and the new values of the changed tuple [11]. However, to update the value of holistic aggregation, all history tuple values must be kept and the new aggregate value needs to be recomputed even after one tuple is inserted or deleted. Expensive space and time cost make it to be unrealistic to incrementally update a holistic aggregation. In the following section, we will introduce two techniques called **addset data structure** and **sliding window** to deal with this problem.

### 3. TECHNIQUES FOR MAINTAINING MEDIAN

MEDIAN is a holistic function which “has no constant bound on the size of the storage needed to describe a sub-aggregate” [7]. It is obvious that MEDIAN cannot be maintained just by storing the final aggregation result from a set of tuples. One naive approach to maintaining MEDIAN value can be figured out as follows: (1) for each cell, we explicitly store a set of measures from the tuples which match the cell. We call the set of measures, the **measure set** of the cell; (2) for each cell, we update its measure set and recompute the aggregation MEDIAN value when new tuples are inserted.

In the above naive approach, storing the measure set for each cell can become prohibitively expensive because of the large number of cells and tuples. The concept of quotient cube helps to reduce this storage requirement as we can group cells into equivalence classes and store only one measure set for each equivalence class. However, the size of

<sup>1</sup>Agg(a,b) means to apply the corresponding aggregate function to a and b

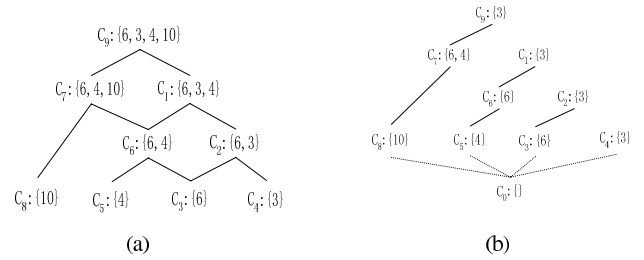


Figure 3: (a)Measure set; (b)Naive Addset

measure sets of quotient cube can be still prohibitively large since each measure set can be large. Moreover, new arriving tuples can result in more equivalence classes which again bring up the storage requirement substantially.

In this section, we present our techniques for updating a quotient cube with aggregate function MEDIAN. We will leave it to readers to see that these techniques are also applicable in the maintenance of other holistic functions like QUANTILE.

### 3.1 Addset data structure

This subsection first describes intuitively how the concept of addset data structure can reduce the storage requirement for maintaining measure set, then proposes a more practical technique of addset data structure including both materialized nodes and pseudo nodes.

Because different equivalence classes in a quotient cube may share some base tuples, there are some redundances among their measure sets. Figure 3(a) shows the quotient cube formed by the base table of Figure 1(a). The measure set of class  $C_9$  is  $\{6,3,4,10\}$  and that of class  $C_7$  is  $\{6,4,10\}$ . It can be observed that  $\{6,4,10\}$  is actually redundant between class  $C_9$  and class  $C_7$ . If we can remove this kind of redundancy, lots of storage space can be spared.

Let us further the discussion to the scenario of updating quotient cube when new tuples come. We find that the tuples matching a newly generated equivalence class are always the superset of the tuples matching its generator (Proposition 2). For example in Figure 2,  $C_{10}$  is a newly generated equivalence class, and  $C_2$  is its generator. The tuples matching  $C_{10}$  are  $T_1 = \{(3, a_1 b_1 c_2, 3), (4, a_1 b_1 c_1, 6), (5, a_1 b_3 c_1, 5)\}$ , while the tuples matching  $C_2$  are  $T_2 = \{(3, a_1 b_1 c_2, 3), (4, a_1 b_1 c_1, 6)\}$ . We have  $T_1 \supset T_2$ . Based on this property, we know that maintaining the list of measures in the new equivalence class can be done by simply storing the difference between the measure set of the new class and that of its generator. We call this difference the **addset** of the new equivalence class.

Assuming that the four base tuples are inserted into a null quotient cube one by one, Figure 3(b) shows the naive addset data structure associated with the quotient lattice of Figure 3(a) (detailed updating algorithm will be explained in section 4). For each new class, it only stores the difference between its measure set and the measure set of its generator. For example, class  $C_5$  is the generator of class  $C_6$ , so class  $C_6$  only stores  $\{6\}$  which is the measure set difference between  $\{6,4\}$  and  $\{4\}$ . Class  $C_0$  is specially introduced as the virtual class so that it can be the generator of the new classes formed by the four base tuples themselves.

Note the space saving we have by adopting the concept of addset in the simple example with only 4 tuples. Instead of storing 18 measures in the naive quotient cube approach in Figure 3(a), we now store only 10 measures (i.e. about 2 times better). The saving is expected to be much more when the number of tuples is large.

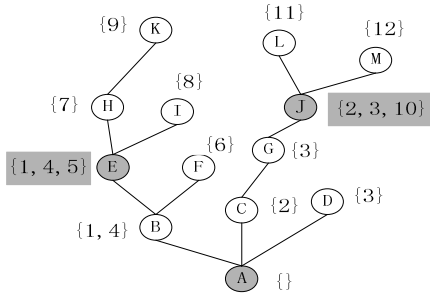


Figure 4: Dynamical Materialization of Addset

There is a linkage between the new class and its generator. Since each new class has a unique generator, the addset data structure is actually a tree. We call it a **family tree**<sup>2</sup>.

We can get the actual measure set of a class by combining the addsets along its family linkage path from the node representing the class to the root. For example, to find the full measure set of class  $C_1$ , we combine the addsets of  $C_1, C_5$  and  $C_6$  i.e.  $\{3\} \cup \{6\} \cup \{4\} = \{3, 6, 4\}$ .

The naive addset data structure can work well if the family path is not very long. However, the computation cost of obtaining the measure set will increase with the length of family path. This may deteriorate performance when the family path is very long. In order to achieve some tradeoff between the space and time, we can dynamically **materialize** some classes in the process of maintenance, i.e., compute the measure sets of these classes and store them explicitly.

Henceforth, we will refer to an equivalence class that stores the addset as a **pseudo (equivalence) class** and a class that stores the actual measure set as a **materialized (equivalence) class**. To obtain the actual measure set of a pseudo class, we only need to trace to its nearest materialized ancestor class instead of the tree root. Figure 4 shows an example of dynamic materialization of addset, the grey circles in Figure 4 represent materialized classes and the blank circles represent pseudo classes. The set of numbers besides a materialized class is its measure set and the set of numbers besides a pseudo class is its addset. (Note that the example in Figure 4 is different from the example in Figure 3 since the latter is too simple to explain the concept of dynamic materialization.) To compute the measure set of class  $K$  in Figure 4, we only need to trace to its first materialized ancestor(class  $E$ ) instead of the root node in naive addset structure.

Next we will address the problems of which classes should be materialized and when are they materialized. Similar to the problem of the materialized view selection [16], we should materialize those classes which can produce the largest benefit. If there is sufficient space, we can materialize more classes to save maintenance time; otherwise, we should keep more classes to be pseudo to save storage space. In this paper, we use a **distance** threshold to control the materialization of pseudo classes. When the distance between a pseudo class and its nearest materialized ancestor exceeds the given threshold, it will be materialized.

**DEFINITION 9 (DISTANCE).** Given a pseudo class  $C_v$  and its nearest actual ancestor  $C_a$ , the distance between  $C_v$

<sup>2</sup>Although the construction of family tree from scratch is not the focus of the paper, interested reader can find the Algorithms 4 and 5 to be described can be used for the purpose when assuming the existing quotient cube to be null

and  $C_a$  is the total number of measures in the addset on the path from  $C_v$  to  $C_a$ . □

For example in Figure 4, assuming that the distance threshold is 3, the distance from classes  $E$  to its nearest materialized ancestor (tree root) is 3 (while there are only two linkages from  $E$  to the root), therefore  $E$  is materialized. Class  $J$  is also materialized because its distance to the root is 3. Once a pseudo class is materialized, its distance becomes 0.

When an equivalence class is generated or modified, we determine whether to materialize the equivalence class or make it a pseudo class. The details will be explained in Section 4.

### 3.2 Sliding Window

The addset data structure (including both materialized and pseudo nodes) seems to be promising to reduce the storage requirement for maintaining MEDIAN values. However, the computation of updated MEDIAN values for all equivalence classes is still expensive by itself. Moreover, extra processing is required to obtain the measure sets for pseudo equivalence classes. In this subsection, we propose a novel *sliding window* technique to compute updated MEDIAN values efficiently.

One important observation contributing to the *sliding window* technique is that given a set of  $n$  measures, the number of elements that are larger and smaller than the median of the  $n$  measures is the same. As a result, by keeping track of the  $k$  ( $1 \leq k \leq n$ ) measure values around the median in a sliding window, we are able to ensure that the median will still lie within the sliding window even with  $k$  insertions. This forms the basis of our sliding window technique for maintenance of MEDIAN.

With the above observation, we will look at how the sliding window technique can be used to compute the median for a pseudo class using the addset and its nearest materialized ancestor. Given a materialized equivalence class  $C_m$  with a sorted measure set  $S = \{x_0, x_1, \dots, x_{n-1}\}$ , the median of one of its descendant pseudo class  $C_d$  is able to be efficiently computed as follows:

(1) Let  $x_{med}$  represent the median of the  $n$  measures. We maintain a sliding window of size  $k$  to keep track of the middle  $k$  measures around  $x_{med}$  in  $S$ . Note that  $k$  must be greater than the distance value between  $C_m$  and  $C_d$  (the reason will be clear later). The sliding window is shown as the area between  $x_{low}$  and  $x_{high}$  in Figure 5, where  $x_{low}$  and  $x_{high}$  are the lowest and the highest measures in the sliding window respectively.

(2) We insert each measure of the addsets of the nodes between  $C_m$  and  $C_d$  into  $S$ . As a new measure  $x$  from an addset is inserted into  $S$ , we adjust  $x_{med}$ ,  $x_{low}$  and  $x_{high}$  according to the following criteria (implemented in algorithm 3 in section 4.1):

1.  $x < x_{low}$ : in this case,  $x_{med}$  needs to move 1/2 position to the left.
2.  $x > x_{high}$ : in this case,  $x_{med}$  needs to move 1/2 position to the right.
3.  $x_{med} < x < x_{high}$ : in this case,  $x_{med}$  needs to move 1/2 position to the right and  $x_{high}$  needs to move one position to the left.
4.  $x_{low} < x < x_{med}$ : in this case,  $x_{med}$  needs to move 1/2 position to the left and  $x_{low}$  needs to move one position to the right.

By doing so, the median for a pseudo class can be efficiently computed only based on the sliding window of its

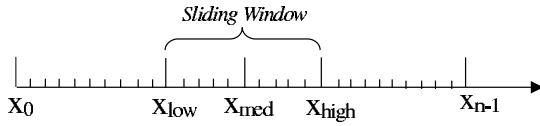


Figure 5: An example of sliding window

nearest materialized ancestor so long as distance between them does not exceed the size of the sliding window. When the distance exceeds the size of the sliding window, extra I/O is required to read more measures to compute new median. In order to avoid the extra I/O, we require the distance threshold defined in section 3.1 to be equal to or smaller than the size of sliding window. When maintaining the quotient cube, we will materialize a pseudo class if its distance to its materialized ancestor exceeds the distance threshold.

Note that the size of the sliding window can be set flexibly by the user. For example, we might let the size of the sliding window to fit within a page so that I/O cost is minimized. Alternatively, we can let the size of sliding window to be the sum of distance threshold and the batch size of insertion. In this case, the new aggregation value of both the materialized classes and pseudo classes can be computed using sliding windows without extra I/O cost. Interestingly, it can be shown that when the size of the sliding window is equal to 1, all equivalence classes in the family tree are materialized classes, which is the implementation of the naive quotient cube maintenance we mentioned at the beginning of section 3. On the other hand, when the size of the sliding window is equal to or larger than the total number of tuples in the base relation, all classes in the family tree are pseudo classes. Although we can obtain the highest space reduction with such a setting, efficiency is affected as we need to sort the whole measure set of a class when computing the median. The size of the sliding window can thus be seen as a parameter to balance the space-time tradeoff in the maintenance of a quotient cube for MEDIAN.

## 4. MAINTENANCE ALGORITHMS

This section illustrates how to maintain the MEDIAN quotient cube incrementally using addset data structure and sliding window technique. Four components  $dist$ ,  $msset$ ,  $parent$  and  $chdlist$  are added to the structure of an equivalence class as defined in Section 2. An equivalence class  $C$  is now represented by the structure  $(upp, m, dist, msset, parent, chdlist)$ .

For a materialized class,  $dist = 0$  and  $msset$  registers the actual measure set. For a pseudo class,  $dist$  refers to the distance to the nearest materialized ancestor and  $msset$  registers the addset relative to its generator. When a new equivalence class is generated or when an existing equivalence class is modified, the values of components  $dist$  and  $msset$  are updated simultaneously. If the value of  $dist$  for a pseudo class is larger than the size of the sliding window, it is converted into a materialized class by backtracking to its nearest materialized ancestor to compute the complete measure set for the pseudo class. Parameters  $parent$  and  $chdlist$  register the parent-child relationship between a new class and its generator in a family tree.

In what follows, we first introduce the algorithm  $Inc\_Single$ , which updates a quotient cube for one new tuple. Based on  $Inc\_Single$ , a more practical algorithm,  $Inc\_Batch$ , which updates a quotient cube in batches will be given.

### 4.1 Single Tuple Maintenance of Insertion

In this section, we first look at algorithm  $Inc\_Single$ , which applies the three propositions in section 2, addset structure and sliding window for updating a quotient cube.

---

#### Algorithm 1 $Inc\_Single(\Phi, t, size)$

{ $\Phi$  is old quotient cube,  $t$  is the new tuple, and  $size$  is the size of sliding window}

1. Create a virtual class  $VC = \{DV, \{DV\}, 0\}$
  2. Divide classes with the same upper bound cardinality into buckets  $B[0] \dots B[n+1]$ ,  $VC$  is in  $B[n+1]$
  3. let  $B'[i] = \emptyset (i=0 \dots n)$  {initialize another bucket set}
  4. for  $i=0$  to  $n+1$  do
  5.   for each class  $C$  in  $B[i]$  do
  6.     if  $t.dvalue$  matches  $C.upp$  then  
       { $C$  is a value modified class}
  7.       ModifyClass( $C, t.m, size$ ), add  $C$  to  $B'[i]$
  8.     else
  9.       MaxMatch =  $C.upp \cap t.dvalue$
  10.       let  $k = |MaxMatch|$
  11.       if  $(k=0)$  then continue
  12.       if  $\neg \exists Z \in B'[k]$  s.t.  $Z.upp = MaxMatch$  then  
       { $C$  is a new class generator}
  13.       split  $C$  into  $C_n$  and  $C'$ ;  $C_n.parent = C'$ ,  
       add  $C_n$  to  $C'.chdlist$ ,  $C_n$  to  $B'[k]$  and  $C'$  to  $B'[i]$ ;  
       if  $(C.dist=0)$  then  $C_n.msset = t.m$ ,  $C_n.dist = 1$ ;  
       else  $C_n.msset = t.m$ ,  $C_n.dist = C.dist+1$ ;  
       if  $(C_n.dist \geq size)$  then materialize  $C_n$ ;
  14.     end if
  15.   end if
  16. end for
  17. end for
  18. df.output( $r, size$ ) { $r$  is the root of the family tree}
- 

---

#### Algorithm 2 ModifyClass( $C, r, size$ )

1. if  $C.dist=0$  then  $C.msset = C.msset \cup r$
  2. else { $C$  is a pseudo class}
  3.   if  $t.dvalue$  does not match  $C.parent.upp$  then  
        $C.msset = C.msset \cup r$ , update the  $dist$  of  $C$  and  
       all its direct pseudo descendants  $C_d$   
       if  $\exists dist \geq size$ , materialize  $C$  or  $C_d$  corresponding  
       to the  $dist$
  5.   end if
  6. end if
- 

Algorithm 1 shows the pseudo code for  $Inc\_Single$ . Having generated a virtual class  $VC$  for reason explained in Section 2,  $Inc\_Single$  divides  $VC$  and all classes of  $\Phi$  into buckets  $B[0], \dots, B[n+1]$  in line 2. A bucket  $B[i]$  contains all equivalent classes  $C$ , such that  $|C.upp| = i$  i.e. there are exactly  $i$  dimensions in  $C.upp$  which do not have "all" as their values. The only exception here is for  $VC$  which is in the  $(n+1)^{th}$  bucket. We will henceforth refer to  $|C.upp|$  as the **cardinality** of  $C$ . A different set of buckets  $B'[0], \dots, B'[n]$  are initialized to store the updated and new equivalent classes for  $\Phi'$  (line 3).

The main loop (lines 4-17) iterates through the classes in each bucket in the order  $B[0], \dots, B[n+1]$ . For each class  $C$  in a bucket  $B[i]$ ,  $Inc\_Single$  first tests for a value modified class (line 6) by checking whether  $C.upp$  is a subset of  $t.dvalues$ . Corresponding update is performed (line 7) for such a case. For example, if tuple  $(a_1 b_1 c_1, 15)$  is added to Figure 3(b), all update is done in line 7 and split will not occur.

Otherwise, a test for a new class generator is done (line 9-12) by computing  $MaxMatch = C.upp \cap t.dvalue$  and testing for its existence in line 12. In between, dumb classes are filtered off if  $C$  does not intersect  $t$  (line 11). Having confirmed that  $C$  is a new class generator,  $C$  will be split based on Definition 8. The new classes,  $C_n$  and updated generator  $C'$  will be added into  $B'[k]$  and  $B'[i]$  respectively. The algorithm ends when all equivalence classes in  $\Phi$  are processed.

Note that checking the buckets in ascending cardinality order is important in verifying two conditions of Proposition 2. This order guarantees that the first encountered class,  $C_f$ ,

**Algorithm 3**  $df\_output(C, k)$   $\{k \text{ is the size of sliding window}\}$

```

1.  if C is a new or modified class then
2.    if  $C.dist=0$   $\{C \text{ is a materialized class}\}$  then
3.      sort measure set  $C.msset$  and get median
4.       $LRDiff=0$ 
5.      put middle  $k$  measures into window  $s[0] \sim s[k-1]$ 
6.    else  $\{C \text{ is a pseudo class}\}$ 
7.      for each data  $d$  in  $C.msset$  do
8.        if  $d < s[k/2]$  then  $LRDiff++$  else  $LRDiff--$ 
9.        if  $s[0] < d < s[k-1]$  then update sliding window
10.     end for
11.     get the new median at  $s[(LRDiff+k)/2]$ 
12.   end if
13.   Output the info of the class
14. end if
15. for each child  $C_{child}$  of  $C$  do
     $\{recursive\ output\}$   $df\_output(C_{child})$  end for

```

which produces  $MaxMatch$  as the intersection of  $C_f.upp$  and  $t$  must be the Greatest Lower Bound (GLB) for all subsequent classes,  $C_s$ , that also have  $C_s.upp \cap t = MaxMatch$ . Also, since  $MaxMatch$  is a subset of  $C.upp$ ,  $k$  will be less than  $i$ . Thus bucket  $B'[k]$  is already updated before classes in  $B[i]$  are processed, making it possible to check for the second condition of Proposition 2 by verifying that  $MaxMatch$  is not already in bucket  $B'[k]$ .

Now we will explain how procedure *ModifyClass* (Algorithm 2) works. If a class satisfies the proposition 1 described in section 2, procedure *ModifyClass* is called. In case that the class is a materialized class, its measure set should be modified (line 1). If it is a pseudo equivalence class, the updating is a bit complicated. First, not all the pseudo classes that satisfy Proposition 1 need to be modified. For example, when a new tuple  $t_5 = (5, a_4b_1c_1, 12)$  is added to Figure 3(b), both equivalence classes  $C_6$  and  $C_1$  satisfy Proposition 1. We only need to modify the addset of  $C_6$  while the addset of  $C_1$  needs not to be modified since the new measure can be obtained from the addset of its parent (i.e.  $C_6$ ). Second, the parameter  $dist$  must be updated for all pseudo equivalence classes. For example in Figure 4, if pseudo class  $B$  is modified, the parameter  $dist$  of class  $F$  must also be updated.

The new median values of all new and modified classes must be computed after the measure sets and addsets are updated. Algorithm 3 computes the median value for the updated equivalence classes in a depth-first order. Note that the depth-first order is extremely important for the sliding window technique to be efficiently adopted. Variable  $LRDiff$  registers the distance that the window should be slid to the left or right. For a materialized class, line 3 sorts all measures and selects the middle measure as the median. Lines 4-5 initialize  $LRDiff$  to 0 and place middle  $k$  measures into the sliding window, which makes preparation for later computation of its pseudo descendants. For a pseudo class, it only needs to compare and slide the window (line 7-11). Since the number of the measures in addsets cannot exceed the size of the sliding window  $k$ , this method needs at most  $k$  comparisons and thus is very efficient. After outputting the information of the current class, the algorithm is recursively called for each of its child (line 15).

## 4.2 Batch Maintenance of Insertions

We next introduce Algorithm *Inc\_Batch* for batch updating of a MEDIAN quotient cube. *Inc\_Batch* is inspired by the BUC algorithm proposed by Bayer and Ramakrishnan [3] which recursively partitions tuples in a depth-first manner so that tuples involved in computing the same cell are grouped together at the time of computation for the cell's

value. The partitioning is performed on different dimensions at each level of the recursion so that different groupings can be formed.

The novelty of Algorithm *Inc\_Batch* over BUC is that *Inc\_Batch* is a maintenance algorithm which performs partitioning on both the existing classes in  $\Phi$  (represented by their upper bounds) and the new set of tuples. We will refer to a partition of the new tuples as a **tuple partition** and a partition of equivalence classes as **class partition**.

To ensure the effectiveness of *Inc\_Batch*, we “synchronize” the tuple and class partitioning in such a way that a particular tuple partition that is being processed at one time is guaranteed to affect only the corresponding class partition that is being processed at the same time. This enhances efficiency in two ways. First, by grouping tuples that share similar dimensional values together, the search for affected equivalence classes needs only to be done once. Second, as the partitioning of equivalence classes done in synchronization with the tuple partitioning, the number of equivalence classes that are being checked is substantially reduced. This “synchronization” is performed in a function of *Inc\_Batch* called **Enumerate()**.

We now explain *Inc\_Batch* in details. The pseudo code of *Inc\_Batch* is shown in Algorithm 4. The main algorithm simply calls the *Enumerate()* function by providing the set of new tuples  $R'$ , the original set of equivalent classes  $\Phi$ , the number of dimensions in the cube and the size of the sliding window. The function *Enumerate()* will then perform recursive partitioning of both the tuples and equivalent classes and update the changes that will be made to various classes in  $\Phi$ . The main algorithm will then output these changes which will produce value modified classes and new classes.

We next look at the function *Enumerate()*. Given the input tuple and class partition,  $input.t$  and  $input.c$ , *Enumerate()* iterates through all the remaining dimensions (from  $dim$  onwards) and partitions both  $input.t$  and  $input.c$  based on the dimensional values of each individual dimension  $D$  (line 3 and 4). The inner loop from line 5 to 11 will then go through each individual dimensional value of  $D$  and recursively call *Enumerate()* to perform further partitioning on the corresponding partitions of the dimensional value.

Finally, we look at procedure *CheckandUpdate* in the first line of function *Enumerate()*. Given the input cell, the tuple partition  $input.t$  and the cell partition  $input.c$ , *CheckandUpdate*'s task is to determine how  $input.t$  will affect the equivalence classes in  $input.c$ . The approach in this procedure is similar to *Inc\_Single* except for some changes due to the batch processing. One main difference is that the cell from the input is used as a representative to compare against the equivalent classes in  $input.c$ .

Algorithm 5 lists the pseudo code for procedure *CheckandUpdate*. The tuple-class comparison is again made in increasing order of cardinality for the equivalent classes. Lines 5 and 6 in the procedure will call procedure *ModifyClass* to update  $C.msset$  or to materialize  $C$  if it detects that a class  $C$  is a value modified class. If  $C.upp \supset cell$ , we compute *uppcell* by appending all dimensional values that have 100% occurrence in  $input.t$  to the *cell*. If *uppcell* equals  $C.upp$ ,  $C.upp$  will be updated from  $input.t$  in future recursion and no action needs to be taken. However, if *uppcell*  $\neq C.upp$ , we will create a temporary class  $C_t$  at Line 13. If the class  $C_t$  is not in  $C.tempset$ , which contains all new classes that are generated from  $C$  and will be output later in the main algorithm of *Inc\_Batch*, we add it to  $C.tempset$  and modify its  $msset$  and  $dist$ . If there already exists a temporary class  $C'_t$  such that  $C'_t.upp = C_t.upp$ ,  $C_t$  is simply discarded since they are in fact the same class.

---

**Algorithm 4:** `Inc_Batch`( $R', \Phi, \text{numDims}, \text{size}$ )
**Input:**

$R'$ : A new set of tuples.  
 $\Phi$ : Existing quotient cube.  
 $\text{numDims}$ : The total number of dimensions.  
 $\text{size}$ : The size of sliding window.

**Output:**

$\Phi'$ : Updated set of equivalence classes.

**Method:**

Enumerate( $\{\}, R', \Phi, \text{numDims}$ )  
 Output value modified classes and new classes.

**Function** Enumerate( $\text{cell}, \text{input}_t, \text{input}_c, \text{dim}$ )**Input:**

$\text{cell}$ : cube cell to be processed.  
 $\text{input}_t$ : a tuple partition.  
 $\text{input}_c$ : a class partition.  
 $\text{dim}$ : the starting dimension for this iteration.

```

1. CheckandUpdate( $\text{cell}, \text{input}_t, \text{input}_c, \text{size}$ )
2. for  $D = \text{dim}$  to  $\text{numDims}$  do
3.   partition  $\text{input}_t$  on dimension  $D$ 
4.   partition  $\text{input}_c$  on dimension  $D$ 
5.   for  $i = 0$  to  $\text{cardinality}[D] - 1$  do
6.      $p_t =$  tuple partition for value  $x_i$  of dimension  $D$ 
7.      $p_c =$  class partition for value  $x_i$  of dimension  $D$ 
8.     if  $|p_t| > 0$  then
9.       Enumerate( $\text{cell} \cup x_i, p_t, p_c, D + 1$ )
10.    end if
11.  end for
12. end for

```

---

**Algorithm 5** CheckandUpdate( $\text{cell}, \text{input}_t, \text{input}_c, \text{size}$ )

```

1. place all measures in  $\text{input}_t$  to measure set  $r$ 
2. add the virtual class  $VC = \{DV, \{DV\}, 0\}$  to  $\text{input}_c$ 
3. sort  $\text{input}_c$  based on ascending cardinality
4. for each class  $C$  in  $\text{input}_c$  do
5.   if  $C.\text{upp} = \text{cell}$  then
6.     ModifyClass( $C, r, \text{size}$ ); break for
7.   else
8.     if  $C.\text{upp} \supseteq \text{cell}$ 
9.       find the upper bound  $\text{uppcell}$  of the class of  $\text{cell}$ 
10.      if  $\text{uppcell} = C.\text{upp}$  then
11.        break for
12.      end if
13.      generate temp class  $C_t, C_t.\text{cpp} = \text{uppcell}$ 
14.      if  $\neg \exists C' \in C.\text{tempset}$  s.t.  $C'.\text{upp} = C_t.\text{cpp}$  then
15.        add  $C_t$  to  $C.\text{tempset}$ 
16.         $C_t.\text{msset} = r$ 
17.         $C_t.\text{dist} = C.\text{dist} + |r|$ 
18.        if  $C_t.\text{dist} \geq \text{size}$ , materialize  $C_t$ 
19.      end if
20.    break for
21.  end if
22. end if
23. end for

```

---

## 5. PERFORMANCE ANALYSIS

To evaluate the efficiency and effectiveness of our update techniques, extensive experiments are conducted. In this section, we report only part of our results due to space limitation. All our experiments are conducted on a PC with an Intel Pentium IV 1.6GHz CPU and 256M main memory, running Microsoft Windows XP. Experiment results are reported on both synthetic and real life datasets.

All run time reported here includes I/O time. We compare our update algorithms with a re-run of the depth-first search algorithm in [10] when an update is made to the original base table. Although we realize that it is not viable to re-generate quotient cube every time the base table is updated, there is no other reasonable benchmark for comparison. Our experiments show that single tuple maintenance algorithm *Inc\_Single* can be up to a hundred time slower than batch maintenance, as such we will only report results for batch maintenance algorithm *Inc\_Batch* which is fed

with the update tuples and existing quotient cube. Note that the order of update tuples does not have any effect on the performance of *Inc\_Batch* while it may affect the performance of algorithm *Inc\_Single*.

### 5.1 Experiments on synthetic datasets

We randomly generated two synthetic datasets with uniform distribution. Both datasets contains 1 million tuples and each tuple has 9 dimensions. Cardinality  $C$  is set at 100 for all 9 dimensions of one dataset and 1000 for the other dataset. Measure for the tuples are randomly generated within the range of 1 to 1000. By default, we set the size of sliding window as 1000, the number of tuples as 200k, the dimensionality of each tuple as 6, the cardinality of each dimension as 100, and the update ration as 50%. An update ratio of  $k\%$  implies  $|\Delta T| = (k\%)*|R|$  tuples are added to the base datasets.

**Efficiency:** We vary the update ratio from 5% to 50%. Figure 6(a) shows the run time of both *Inc\_Batch* (represented with *Inc\_Med\_B*) and the depth-first algorithms (represented with *Dfs\_Med*) on dataset with cardinality  $C=100$ . Figure 6(a) shows that *Inc\_Batch* achieves substantial saving in time than a re-run of the depth-first algorithm. For a update ratio of 50%, we enjoy a 75% saving in processing time. The results clearly indicate that our maintenance algorithm for aggregate function MEDIAN is efficient. The savings in time mainly come from the fact that *Inc\_Batch* reuse previous computation.

Figure 6(b) shows the run time of both algorithms when the dimensionality is increased from 2 to 9. The performance gap between the batch maintenance algorithm *Inc\_Batch* (represented with *Inc\_Med\_B*) and the depth-first algorithm (represented with *Dfs\_Med*) grows with the dimensionality of the dataset.

**Data Skew:** To study the effects of data skew, we vary the distribution of the dimension values in each dimension by changing the zipf factor from 0.0 to 3.0. A zipf factor of 0 means that the dimensional values are uniformly distributed while a high zipf factor will generate a highly skewed dataset. Figure 7 shows the run time of both algorithms as the zipf factor is varied. As the zipf factor increases, the run time of both algorithms decreases. This is because as the zipf factor increases tuples in the dataset are highly similar to each other and the number of equivalence classes will decrease, thus requiring less time for both maintenance and re-computation of the quotient cube.

**Scalability:** We next look at the run time of algorithm *Inc\_Batch* as the number of tuples increases. We increase the number of tuples from 100k to 1 million. Figure 8 shows that although both algorithms have linear scalability, the run time of the incremental maintenance algorithm scales better than a complete re-computation.

**Effectiveness of Addset:** To study the effect of addset in reducing the storage requirement for maintaining the aggregate function MEDIAN, we vary the size of the sliding window from 1 to 200k on both two datasets, which means that the distance threshold also changes from 1 to 200k. The measure set and addset are stored in binary files and thus we use the size of the binary files as a measure for space requirement. Figure 9(a) shows the space requirement for maintaining MEDIAN using *Inc\_Batch*. Table 1 gives more detailed data. As shown in Table 1, when the size of the window is set to 100, the addset only needs 10% of the space compared to the full measure set (when window size equals



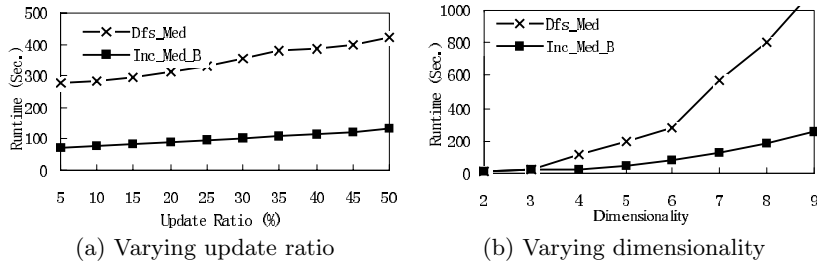


Figure 6: Maintenance Efficiency with C=100

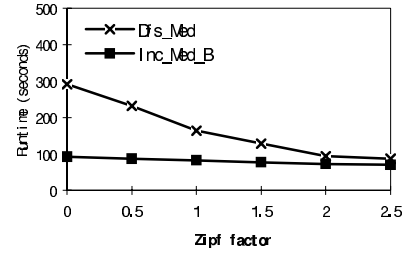


Figure 7: Impact of data skew

1). We observe two tendencies:

Table 1: Space size (M)

Cardinality	Fullset(size=1)	Addset(size=100)
100	59.8	6.4
1000	22.1	5.7

First, as the size of sliding window increases, the space requirement decreases sharply and then levels off. Second, the reduction ratio decreases as the cardinality increases. In other words, the lower the cardinality, the more effective the addset data representation. This is due to the fact that low cardinality dataset are denser which result in more redundancies if the full measure sets are stored.

**Effectiveness of sliding window:** Figure 9(b) shows the run time of algorithm *Inc\_Batch* with respect to varying sliding window sizes. We can see that when the size of the window increases from 1 to 1000, the run time of *Inc\_Batch* decreases. However when the window size continues to increase, the run time begins to increase a bit. This is because too small a window size will result in many materialized classes that require sorting computation. Too big a window size will lead to more backtracking when computing the median for pseudo equivalence classes.

## 5.2 Experiments on real life data

We also evaluate our update techniques on a real life weather dataset<sup>3</sup> which is commonly used in experiments involving computation of data cubes [18, 17, 11]. The dataset contains 1,015,367 tuples and the cardinalities of the dimensions are as follows: station-id (7037), longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8), and brightness (2). We use the first 100k tuples to form the base relation.

Figure 10 shows the maintenance efficiency of both algorithms. As expected, *Inc\_Batch* (represented with Inc\_Med\_B) has the modest run time growth as the update ratio increases. The performance trends revealed by Figure 10 is remarkably similar to those revealed by Figure 6.

We test the effectiveness of addset, and again obtain a sharp decrease in space requirement when the size of sliding window increases. The graph in Figure 11(a) shows that substantial space reduction is obtained even with a sliding window size of 100. Figure 11(b) shows the run time of *Inc\_Batch* with respect to varying window sizes. The result of the algorithm is consistent with the observations we obtain for the synthetic datasets.

In summary, our experiments show that *Inc\_Batch* is a highly efficient algorithm and achieves a substantially better run time reduction than deep-first algorithm. They also show the effectiveness of the addset and sliding window techniques.

<sup>3</sup><http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html>

## 6. RELATED WORKS

Plenty of efforts have been devoted to fast computation of the cube [1, 19]. Since the complete cube consists of  $2^n$  cuboids ( $n$  is the number of dimensions), the size of the union of  $2^n$  cuboids is often too large to be stored due to the space constraints. Thus it is unrealistic to compute the full cube from scratch. There are currently many solutions to the problem, such as choosing views to materialize [8], cube compression [15], approximation [2], handling sparsity [13], and computing the cube under user-specified constraints [3].

Recently, from a different aspect, Wang et al. proposed a concept of condensed cube [18] that explores “single base tuple” and “projected single tuple” to compress a data cube. Lakshmanan et al. proposed a concept of quotient cube [10] that extracts succinct summaries of a data cube based on partition theory. Dwarf [17] identifies prefix and suffix structure redundancies and factors them out by coalescing their storage. All three methods reduced the data cube (hence its computation time and storage overhead) efficiently.

However, as changes are made to the data at the sources, the maintenance work to these compressed data cube is non-trivial. The incremental maintenance of quotient cube is the most challenging since it not only has the largest data compress ratio but also preserves a semantic structure. [11] proposed an efficient data structure called QC-tree. While the important incremental maintenance problem is tackled in the paper, aggregation was considered only in a limited sense. For example, aggregation with holistic aggregation function was not allowed. In this paper, we introduced two techniques called **addset data structure** and **sliding window** to maintain holistic function like MEDIAN. The concept of a sliding window is also used in both [20] and top-k’ view in [21] but no in the context of a QC-tree.

Works on data warehouse maintenance such as [12, 9, 14] are of clear relevance to us. However, none of them addresses the MEDIAN maintenance problem. Our study is also closely related to incremental concept formation algorithms based on Galois lattice [6, 4, 5].

## 7. CONCLUSION

In this paper, we address the problem of updating the existing MEDIAN quotient cube incrementally. We developed a new data structure addset which is able to dramatically cut down the size of storage space required to store measure set for each equivalence class. Moreover, we proposed a sliding window technique to compute the median over not the entire past history of the data, but rather only the sliding windows of middle data from the history. We designed two incremental maintenance algorithms: *Inc\_Single* and *Inc\_Batch*. The former maintains the quotient cube tuple by tuple and the latter maintains the quotient cube in batch. A comprehensive set of experiments on both synthetic and real data sets were conducted. Our results show that our maintenance algorithms are efficient in space and time.

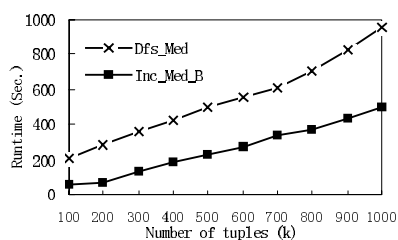


Figure 8: Scalability with the number of tuples

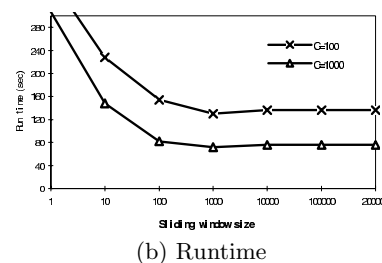
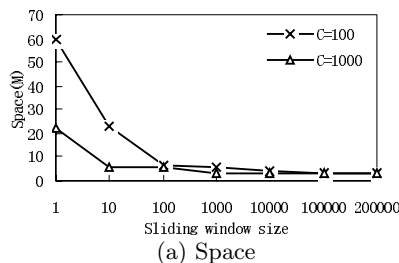


Figure 9: Effectiveness of Addset and Sliding Window

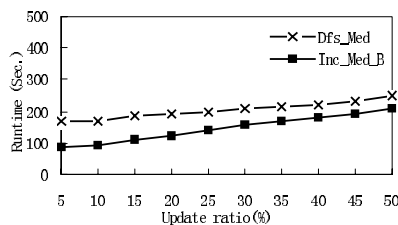


Figure 10: Maintenance Efficiency for Real Data

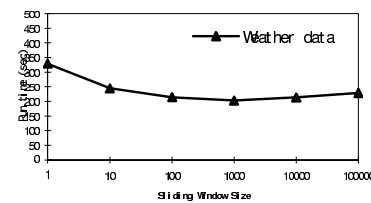
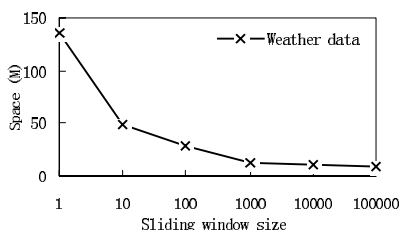


Figure 11: Effectiveness of Addset and Sliding Window on real data

## 8. REFERENCES

- [1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 506–521, Bombay, India, Sept. 1996.
- [2] D. Barbara. Quasi-cubes: Exploiting approximation in multidimensional databases. *SIGMOD Record*, 26:12–17, 1997.
- [3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.
- [4] A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- [5] B. Ganter, R. Wille, and C. Franzke. *Formal concept analysis: mathematical foundations*. Springer-verlag, 1999.
- [6] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on galois lattices. *Computational Intelligence*, 11:246–267, 1991.
- [7] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–54, 1997.
- [8] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 205–216, Montreal, Canada, June 1996.
- [9] W. Labio, U. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proc. of the 26th Int'l Conference on Very Large Databases (VLDB'00)*, 2000.
- [10] L. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, 2002.
- [11] L. Lakshmanan, J. Pei, and Y. Zhao. Qc-trees: An efficient summary structure for semantic olap. In *Proc. Of ACM-SIGMOD Int'l Conference on Management of Data*, 2003.
- [12] I. Mumick, D. Quass, and B. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. Of ACM-SIGMOD Int'l Conference on Management of Data*, 1997.
- [13] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 1997 Int. Conf. Very Large Data Bases (VLDB'97)*, pages 116–125, Athens, Greece, Aug. 1997.
- [14] K. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: trading space for time. In *Proc. Of ACM-SIGMOD Int'l Conference on Management of Data*, 1996.
- [15] J. Shanmugasundaram, U. Fayyad, and P. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *Proc. Of ACM-SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*, 1999.
- [16] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 488–499, New York, NY, Aug. 1998.
- [17] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the petacube. In *Proc. Of ACM-SIGMOD Int'l Conference on Management of Data*, 2002.
- [18] W. Wang, J. Feng, H. Lu, and J. Yu. Condensed cube: An effective approach to reducing data cube size. In *Proc. of 2002 Int'l Conf. on Data Engineering (ICDE'02)*, 2002.
- [19] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 159–170, Tucson, Arizona, May 1997.
- [20] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, 2002.
- [21] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient Maintenance of Materialized Top-k Views. In *Proc. of 2003 Int'l Conf. on Data Engineering (ICDE'03)*, 2003.