Efficient Computation of Iceberg Cubes with Complex Measures *

Jiawei Han† Jian Pei† Guozhu Dong‡ Ke Wang†

† School of Computing Science, Simon Fraser University, B.C., Canada, {han, peijian, wangk}@cs.sfu.ca

‡ Department of Computer Science, Wright State University, Dayton, OH, U.S.A., gdong@cs.wright.edu

Abstract

It is often too expensive to compute and materialize a complete high-dimensional data cube. Computing an iceberg cube, which contains only aggregates above certain thresholds, is an effective way to derive nontrivial multi-dimensional aggregations for OLAP and data mining.

In this paper, we study efficient methods for computing iceberg cubes with some popularly used complex measures, such as *average*, and develop a methodology that adopts a weaker but anti-monotonic condition for testing and pruning search space. In particular, for efficient computation of iceberg cubes with the *average* measure, we propose a *top-k average* pruning method and extend two previously studied methods, Apriori and BUC, to Top-k Apriori and Top-k BUC. To further improve the performance, an interesting hypertree structure, called H-tree, is designed and a new iceberg cubing method, called Top-k H-Cubing, is developed. Our performance study shows that Top-k BUC and Top-k H-Cubing are two promising candidates for scalable computation, and Top-k H-Cubing has better performance in most cases.

1 Introduction

The introduction of data cube [8] can be considered as a landmark in data warehousing since the materialization of multi-dimensional data in large data repositories facilitates fast, on-line data analysis. However, as many researchers pointed out (e.g., [10, 14, 4]), it is prohibitively expensive in both space and time to completely materialize a data cube with high dimensionality. Several methods have been proposed to overcome this difficulty, including (1) *selective materialization of some cuboids* [10], where a *cuboid* is multi-dimensional summarization of only a *subset* of dimensions, and (2) *materialization of only iceberg cubes* [4], where an *iceberg cube* is a subset of a cube containing only those cells whose measure, such as *count*, satisfies certain constraints, such as *minimal support threshold*.

The usefulness of iceberg cubes [4] is obvious. A cube can be viewed as a lattice of cuboids, where cuboids whose group-bys include more dimensions are at a *lower level* than those include fewer dimensions, and the one that includes all the dimensions, called the *base cuboid*, is at the bottom. Most cells of the low level cuboids are likely to contain trivial aggregate values and may not pass certain thresholds, and therefore, do not need to be computed in an iceberg cube. This not only saves processing time and disk space but also focuses analysis only on interesting data since the cells that cannot pass the thresholds are likely to be too trivial to warrant further analysis.

Previous studies [6, 4] on efficient computation of iceberg queries or iceberg cubes have been confined to iceberg queries/cubes with simple measures, such as *count* and *sum*, by exploring the *anti-monotonic property*[1] of iceberg cubes. For example, if the *count* of a cell c in a higher level cuboid C is no higher than v, then the *count* of any of c's descendant cells in the lower level cuboids can never be higher than v, and thus can be pruned by an Apriori-like method [2]. Unfortunately, not all the measures have such *anti-monotonic property*. For example, even if the *average* value in a cell c of a higher level cuboid C is no higher than v, the average value of some of c's descendant cells in the lower level cuboids may still be higher than v.

In this paper, we study how to efficiently compute

* Work supported in part by the Natural Sciences and Engineering Research Council of Canada (A3723) and the Networks of Centres of Excellence of Canada (IRIS-3)

[1] Anti-monotone, first introduced in [12], is different from monotone in that the latter is about *condition satisfaction*, whereas the former is about *condition violation*.

iceberg cubes with non-antimonotonic measures. Before examining this problem, one may ask, "*Is it truly useful to compute iceberg cubes with such measures?*" The answer is a resounding "*yes!*," as shown below.

Example 1 Suppose a sales database has four dimensions: *time, location, customer*, and *product*, and two measures: *price* and *cost* (note: *profit* = *price* − *cost*). The following queries require the computation of iceberg cubes with such *complex* measures.

Q_1: Find groups of sales which contain at least 50 items and whose average item price is at least \$800, grouped by month, city, and/or customer groups.

Q_2: Find groups of sales which contain at least 200 items and whose *total profit*[2] is more than \$6000, grouped by month, city, and/or customer groups.

Q_3: For sales grouped by month, city, and/or customer groups, containing at least 20 items, with an average item price of no less than \$800, find those customer groups on which one can make at least 10% more profit than the average of all the customers. □

Can we find some interesting properties and effective methods so that computation of such iceberg cubes can still be made efficient? This paper investigates this problem, with the following contributions.

1. It develops a mapping which transforms some non-antimonotonic testing conditions to somewhat weaker but anti-monotonic testing conditions so that the search space can be pruned effectively. For example, test on *average* can be mapped to an anti-monotonic, *top-k average* test. Mappings for several other measures are worked out as well.
2. It extends two previously studied methods, Apriori [2] and BUC [4], to Top-k Apriori and Top-k BUC, for computing iceberg cubes with the average measure.
3. To further improve performance for computing iceberg cubes, a hypertree structure, called H-tree, is designed, and a new iceberg cubing method, called Top-k H-Cubing, is developed. The method explores several efficient processing techniques, including tree-based data compression, dynamic link adjustment, and quantitative information merge; these techniques make iceberg cubing highly efficient and scalable, outperforms a high performance cubing technique, BUC [4], in most cases, according to our performance study.

[2] *Profit* could be *negative*, and thus it cannot be handled by iceberg cubes with a *sum* of *nonnegative* values as in [4].

The remaining of the paper is organized as follows. Section 2 introduces the problem of computing iceberg cubes with the average measure. Section 3 explores a weaker but anti-monotonic condition, top-k average, for effective pruning of search space, and presents a binning technique to handle top-k average. Section 4 presents two algorithms, Top-k Apriori and Top-k BUC, which extend Apriori and BUC to compute average iceberg cubes. A hypertree data structure, H-tree, and a new algorithm, Top-k H-Cubing, for efficient computation of average iceberg cubes are presented in section 5. Our performance study is reported in section 6. In section 7, we extend our scope to examine other complex measures and discuss related work. We conclude our study in section 8.

2 Iceberg Cubes with the Average Measure

Example 2 (Iceberg cubes on average) Consider a *Sales_Info* table, given in Table 1, which registers sales related to month, day, city, customer group, product, cost, and price.

Mon.	Day	City	Cust_group	Product	Cost	Price
Jan	10	Toronto	Edu.	HP Printer	500	485
Jan	15	Toronto	Household	Sony TV	800	1200
Jan	20	Toronto	Edu.	Canon Camera	1160	1280
Feb	20	Montreal	Busi.	IBM Laptop	1500	2500
Mar.	4	Vancouver	Edu.	Seagate HD	540	520
...	...	...	...	...	...	...

Table 1: A *Sales_Info* table.

An iceberg cube, *Sales_Iceberg*, which computes Q_1 is presented as follows.

```
CREATE CUBE Sales_Iceberg AS
SELECT month, city, customer-group, AVG(price),
    COUNT(*)
FROM Sales_Info
CUBEBY month, city, customer-group
HAVING AVG(price) >= 800 AND COUNT(*) >= 50
```

Notice that *Sales_Iceberg* differs from its corresponding whole data cube, *Sales_Cube*, in that the former is a restriction of the latter: the former excludes all the cells of the latter whose average price is less than \$800 or whose count is less than 50. It is also different from its corresponding *iceberg query*, formed by replacing CUBEBY with GROUPBY, in that the latter contains only the qualified cells with the three dimensions grouped together, whereas the former contains the qualified cells of *all the possible group-bys* of the three dimensions. □

It is easy to verify that it is highly inefficient and sometimes impossible to first materialize the whole data cube and then select the cells satisfying the

HAVING-clause specified in the iceberg cube since this may lead to a huge number of cells (but most containing only trivial measures) to be computed when the number of dimensions are not too small. To develope an efficient method for computing iceberg cubes, let's first define some terms.

Definition 1 In an n-dimension data cube, a cell $a = (a_1, a_2, \ldots, a_n, measures)$ is called an **m-d cell** (which is a cell in an m-d cuboid), if and only if there are exactly m ($m \leq n$) values among $\{a_1, a_2, \ldots, a_n\}$ which are *not* $*$. It is called a **base cell** (which is a cell in a base cuboid) if $m = n$. A non-base cell stores aggregate values and thus it is sometimes referred to as **aggregate cell**. In an n-dimension data cube, an i-d cell $a = (a_1, a_2, \ldots, a_n, measures_a)$ is an **ancestor** of a j-d cell $b = (b_1, b_2, \ldots, b_n, measures_b)$, and b is a **descendant** of a, if and only if (1) $i < j$, and (2) for $1 \leq m \leq n$, $a_m = b_m$ whenever $a_m \neq *$. In particular, cell a is called a **parent** of cell b, and b a **child** of a, if and only if $j = i+1$ and b is a descendant of a. Given an iceberg cube $ICube$, the (whole) data cube formed by the same specification of $ICube$ without the `HAVING` clause is called the **background cube** of $ICube$, and is denoted as $\mathcal{B}(ICube)$. □

Example 3 Consider the $Sales\_Cube$ of Example 2. $(Jan, *, *, 1200, 2800)$ and $(*, Toronto, *, 800, 1200)$ are *1-d cells*, $(Jan, *, Edu., 600, 250)$ is a *2-d cell*, and $(Jan, Toronto, Busi., 1500, 45)$ is a *3-d cell*. A 3-d cell is a *base cell*, whereas 1-d and 2-d cells are aggregate cells. 1-d cell $a = (Jan, *, *, 1200, 2800)$ and 2-d cell $b = (Jan, *, Busi., 1300, 150)$ are *ancestors* of 3-d cell $c = (Jan, Toronto, Busi., 1500, 45)$; c is a *descendant* of both a and b, b is a *parent* of c, and c a *child* of b.

$\mathcal{B}(Sales\_Iceberg)$, the *background cube* of the iceberg cube, $Sales\_Iceberg$, of Example 2 is defined as,

```
CREATE CUBE Sales_Cube AS
SELECT month, city, customer-group, AVG(price),
    COUNT(*)
FROM Sales_Info
CUBEBY month, city, customer-group
```
□

Definition 2 An iceberg cube, $ICube$, is **anti-monotonic** if and only if for each cell c in $\mathcal{B}(ICube)$, if c *violates* the constraint specified by $ICube$'s `HAVING` clause, so does every descendant of c. □

Example 4 Given the sales table in Example 2, $Count\_Iceberg$, shown below, is *anti-monotonic*.

```
CREATE CUBE Count_Iceberg AS
SELECT month, city, customer-group, COUNT(*)
FROM Sales_Info
CUBEBY month, city, customer-group
HAVING COUNT(*) >= 100
```

Indeed, if a cell c in $Count\_Iceberg$ violates the constraint specified in the `HAVING` clause, i.e., its count is less than 100, then every descendant of c will violate the constraint since the count of each subcube of c must be no larger than that of c.

$Sales\_Iceberg$ in Example 2 is, however, not anti-monotonic. For example, even when the average price of all the items sold in March is less than \$800, e.g., $(March, *, *, 600, 1800)$, the average price for a subset containing only the sales to business people, e.g., $(March, *, Busi., 1300, 360)$, may still satisfy the constraint specified in the `HAVING` clause. □

3 Exploration of Weaker, Anti-monotonic Conditions

An anti-monotonic iceberg cube can be computed efficiently by exploring the Apriori property [2], as shown in [4]. However, since our iceberg cube involves the non-anti-monotonic measure *average*, it does not have the Apriori property. *"Can we find a weaker but anti-monotonic auxiliary condition that may help us compute iceberg cubes efficiently?"*

3.1 Top-k average: An anti-monotonic condition for testing average

Let us examine the following iceberg cube $AvgI$, a generalization of Example 2, defined on a relational table T with i dimensions and one measure M.

```
CREATE CUBE AvgI AS
SELECT A_1, A_2, ..., A_m, AVG(M), COUNT(*)
FROM T
CUBEBY A_1, A_2, ..., A_m
HAVING AVG(M) >= v AND COUNT(*) >= k
```

Definition 3 A cell c is said to have n base cells if it covers n nonempty descendant base cells. The **top-k average** of c, denoted as $avg^k(c)$, is the average value of the *top-k base cells* of c (i.e., the first k cells when all the base cells in c are sorted in value-descending order) if $k \leq n$; or $-\infty$ if $k > n$.[3] □

Lemma 3.1 (Top-k Apriori) *Let c be an m-d cell which fails to satisfy $avg^k(c) \geq v$ in cube $\mathcal{B}(AvgI)$. If a cell c' is a descendant of c, then c' cannot satisfy $avg^k(c') \geq v$ in cube $\mathcal{B}(AvgI)$.* □

This lemma stimulates us to explore the utilization of the auxiliary condition $avg^k(c) \geq v$ as a looser bound for computing iceberg cubes with the `HAVING` clause "$avg(c) \geq v$ `AND` $count(c) \geq k$". The effectiveness of the search space pruning by top-k average is demonstrated in our performance study in section 6.

[3] $-\infty$ can be implemented as $-$ MAXINT in a computer.

3.2 Optimization: A binning technique for top-k average

There is one concern of this top-k average-based pruning: *"will this require us to keep track of top k values for each cell in an m-dimensional space?"* This seems to be a nontrivial cost. If k is small, e.g., $k = 5$, the overhead could be small. However, if k is large, such as 1000, the overhead could be substantial. The following binning technique can be used to reduce the cost of storage and computation of top-k average.

1. **Large value collapsing:** For any measure value v' which is no less than v (i.e., $v' \geq v$) in $avg^k(c) \geq v$, where v' is called a large value, there is no need to store it explicitly. Instead, it is sufficient to store only two measures: (1) *count*, the number of large values, and (2) *sum*, the sum of all large values.

2. **Small value binning:** If the large values registered can make $avg^k(c) \geq v$, there is no need to store small ones (a value v' is small if $v' < v$). Otherwise, we can set up a small set of bins and register two measures, *count* and *sum*, for each bin. The *large-value* group can be considered as a special bin, bin_1. Let the upper value boundary of bin_i be $max(bin_i)$ and the lower one be $min(bin_i)$. For all $1 \leq i < j$, we have $min(bin_i) > max(bin_j)$. To make binning more effective, we can use denser bins for the region relatively closer to v, and sparser bins for the region relatively far away from v.

   For example, suppose $v \geq 0$, one can set up the ranges of five bins as follows: $range(bin[1]) = [v, \infty)$, $range(bin[2]) = [0.95v, v)$, $range(bin[3]) = [0.85v, 0.95v)$, $range(bin[4]) = [0.70v, 0.85v)$, and $range(bin[5]) = [0.50v, 0.70v)$. Notice since we have count and sum of all the cells, that for the remaining range $[-\infty, 0.50v)$ can be derived easily.

The set of bins for a cell c can be used to judge whether $avg^k(c) \geq v$ is false as follows. Let m be the smallest number such that the sum of counts of the upper m bins is no less than k, i.e., $count_m = \Sigma_{i=1}^{m} count(bin_i) \geq k$. We approximate $avg^k(c)$ using,

$$avg'^k(c) = (\Sigma_{i=1}^{m-1} sum(bin_i) + max(bin_m) \times n_k)/k,$$

where $n_k = k - \Sigma_{i=1}^{m-1} count(bin_i)$.

Lemma 3.2 $avg^k(c) \leq avg'^k(c)$. *Consequently, if $avg'^k(c) < v$, then no descendant of c can satisfy the* Having-*condition in AvgI.* □

Notice that binning might lead to a minorly coarser granularity than registering each of individual k values, and hence less sharp pruning, however, with a good binning technique as described above, the blurring effect is quite minor. Moreover, the technique is safe since it will not lead to missing any answer.

Based on this discussion, we denote three pieces of information *sum, count*, and *top-k bins* as **quant-info**, which often need to be accumulated with each cell for efficient computation of average iceberg cubes.

4 Extension of Apriori and BUC for Iceberg Cube with Average

Based on the above discussions, we extend (1) the Apriori association mining algorithm [2], and (2) the BUC iceberg cube computation algorithm [4], to compute iceberg cubes with average.

4.1 Top-k Apriori

Based on Lemma 3.1, we can work out an Apriori-like [2] iceberg cube computation algorithm, as below.

Example 5 (Top-k Apriori) The iceberg cube in Example 2 can be computed by Top-k Apriori as follows.

First, the set of relevant data is obtained by projecting the database on three relevant attributes, *month, city*, and *customer_group*, and one measure *price*. This forms the base cuboid DB.

Scan DB once to accumulate quant-info (i.e., count, sum, and top-k bin measures) for the 0-d cell c_0 of the 0-d cuboid. Output the 0-d cuboid, $R_0 = \{c_0 \mid count(c_0) \geq 50 \ \wedge \ avg(c_0) = sum(c_0)/count(c_0) \geq 800\}$, and keep the 0-*d live set*, $L_0 = \{c_0 \mid avg'^{50}(c_0) \geq 800\}$.

If $L_0 = \emptyset$, the computation terminates. Otherwise, compute 1-d cells as follows. All the 1-d cells are candidate cells, i.e., forming the candidate set C_1, such as $(Jan, *, *, \ldots)$, $(Feb, *, *, \ldots)$, …, $(*, Toronto, *, \ldots)$, $(*, Vancouver, *, \ldots)$, ….

Then scan DB, accumulate quant-info for each c_1 in C_1, output R_1, and keep the live set L_1:

1. $R_1 = \{c_1 \mid c_1 \in C_1 \ \wedge \ count(c_1) \geq 50 \ \wedge \ avg(c_1) = sum(c_1)/count(c_1) \geq 800\}$;

2. $L_1 = \{c_1 \mid c_1 \in C_1 \ \wedge \ avg'^{50}(c_1) \geq 800\}$.

This process continues level-by-level, until the live set L_k or the candidate set C_k for some k is empty. □

Top-k Apriori computes average iceberg cubes by exploring candidate generation and level-wise computation. This is more efficient than first computing the whole background cube and then selecting the cells using constraints. However, it still involves costly processing: (1) it takes m scans of DB where m is the maximum number of dimensions containing nonempty candidate set, and (2) it may generate a huge number of candidate sets.

4.2 Top-k BUC

An efficient iceberg cube computation method *Bottom-Up Cubing* (BUC) [4] builds the cube from lower number of dimension combinations to higher ones. It explores the dimension ordering by putting the most discriminating dimensions first and then recursively partitioning DB according to the ordering. At each step of recursive partition, one can push in the iceberg constraint, such as *min count*, to remove those that cannot satisfy it. This can be applied to computing iceberg cubes with the *average* measure. For example, for computing $AvgI$, one can use $avg^k(c) \geq v$ to test the partitions generated: any partition that cannot pass the test will not need to be considered further.

Example 6 (Top-k BUC) The iceberg cube $AvgI$ of Example 2 can be computed by Top-k BUC as follows.

Star with the base cuboid DB with three dimensions *month, city,* and *customer_group*, and one measure *price.* Let $cardinality(city) > cardinality(month) > cardinality(customer\_group)$. The BUC processing tree is shown in Fig. 1, where C is for *city*, M for *month*, G for *customer_group*, and *num* in "C : *num*" represents the processing order.

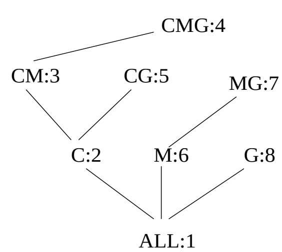

Figure 1: BUC Processing Tree.

Following the processing order indicated in Fig. 1, in the first scan of DB, we (1) accumulate quant-info for "ALL", and (2) project each tuple to the corresponding *city* partition, and (3) accumulate quant-info for each *city*. At the end of the scan, output "ALL" if it passes the count and avg test, and if "ALL" is not alive, i.e., $avg^{50}(ALL) \geq 800$ is false, stop. Otherwise, output city c_i if $count(c_i) \geq 50$ and $avg(c_i) = sum(c_i)/count(c_i) \geq 800$, and mark city c_i live if $avg^{50}(c_i) \geq 800$.

Then, for each live city c_i, scan c_i's partition and project each tuple to its corresponding second dimension M ($month$) and for each CM-partition, accumulate quant-info, and so on. This process continues until CMG is processed or until there exist no live partitions. Then we recurse back and process in the order of CG, M, MG, and finally G, by scanning the corresponding database or partitions. □

Top-k BUC partitions a large database into a set of much smaller data sets by projections over the corresponding dimensions, and localizes the search to partitioned data sets. Without generating candidate sets like Apriori, it may occasionally do some extra work, e.g., if *March* cannot pass $avg^{50}(March) \geq 800$, there is no need to examine the pair of city c_i and *March* by Apriori but BUC still has to examine it (since *March* is in a different partition). However, the trade of accuracy of pruning for locality of reference has been proven highly beneficial in performance [4].

5 Top-k H-Cubing: Top-k Cubing Using a Hyper-Tree Structure

By exploring *dimension partition and constraint push*, Top-k BUC achieves good performance. *Can we further improve the performance?* In this section we introduce a hyper-tree structure, called H-tree, and propose an efficient algorithm, Top-k H-Cubing, for computing *average* iceberg cubes.

5.1 H-tree: A Hyper-Tree Structure

Example 7 (H-tree) Given *Sales_Info* in Table 1 and *Sales_Iceberg* specified in Example 2, a tree structure HT can be built as follows.

1. Tree HT has a root node "*null*", and dimensions are in cardinality-ascending order, i.e. $R : G - M - C$.
2. *A header table* is created, in which each entry records the quant-info for an attribute-value pair.
3. The first tuple, $t_1 = (Edu., Jan, Toronto, 485)$, is inserted into HT, with three nodes, $Edu.$, Jan and $Toronto$ inserted in sequence to form the first branch, and quant-info in the leaf ($Toronto$). Also, price 485 is used to update quant-info for $Edu.$, Jan and $Toronto$ in the header table.
4. Similarly, $t_2 = (Household, Jan, Toronto, 1200)$, is inserted. Since the two leaf nodes have the same label, they are linked by a *side-link*.
5. Since $t_3 = (Edu., Jan, Toronto, 1280)$ has the same attribute values as t_1, t_3 shares the path as t_1, with quant-info in the leaf and header updated.
6. The remaining tuples can be inserted similarly, with the result tree shown in Fig. 2. The tree so formed is called an H-tree. Its construction requires only one scan of the database. □

For lack of space, we omit the rigorous definiton of H-tree. H-tree has some interesting properties which facilitate computing iceberg cubes.

Lemma 5.1 (Properties of H-tree) *Given a relation table T and an iceberg cube creation query $AvgI$*

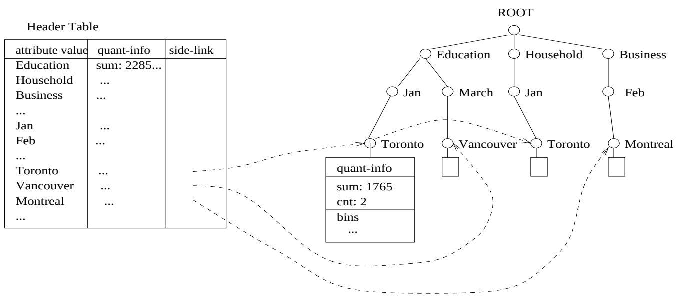

Figure 2: An H-tree.

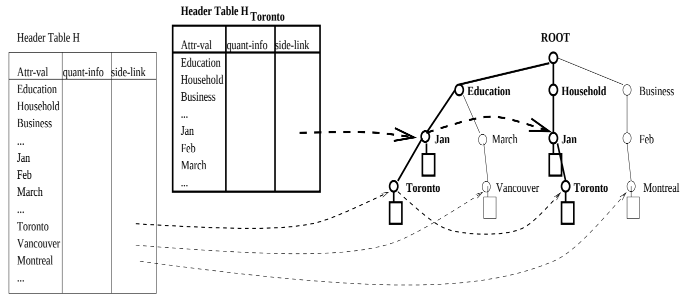

Figure 3: Updated H-tree for computing descendants of $(*, *, Toronto)$.

as in section 3.1, the H-tree HT *has the following properties.*

1. **(Construction cost)** *The* H-tree *can be constructed by scanning the database only once.*
2. **(Completeness)** *The* H-tree *and its header table* H *contain the complete information needed for computing iceberg cube* $AvgI$.
3. **(Compactness)** *Let there be* n *tuples in table* T *and* m *attributes involved in* $AvgI$. *The number of nodes in* H-tree *cannot exceed* $n \times m + 1$. □

5.2 Top-k H-Cubing: Computing iceberg cubes using H-tree

With the compact H-tree structure, one can explore efficient iceberg cube computation, as below.

Example 8 (Top-k H-Cubing) Using the H-tree HT built in Example 7, $AvgI$ can be computed as follows.

Step 1. Compute cells involving dimension C. The *quant-info* in the H-tree tells whether a cell in the form of $(*, *, c)$, where c is a city, passes the top-k average and average tests. For example, the entry $Toronto$ in the header table contains $avg^k(price)$ and $avg(price)$ for $(*, *, Toronto)$. If $avg(price)$ passes the average price threshold, output the cell. If $avg^k(price)$ passes it, the descendants of the cell $(*, *, Toronto)$ should be examined as shown below.

1. The sub-graph of HT containing only the paths related to Toronto, denoted as $HT_{Toronto}$, is an H-tree for sub-cube $(*, *, Toronto)$. $HT_{Toronto}$ is sufficient to compute the iceberg sub-cube w.r.t. Toronto.
2. The *side-link* for $Toronto$ in the header table H links all the paths related to Toronto. By traversing it once, we (1) make a copy of quant-info in every leaf-node labeled $Toronto$ to its parent node in the tree, (2) build a new header table $H_{Toronto}$, which collects quant-info for every attribute-value w.r.t. Toronto, and (3) link all the parent nodes of the leaf-nodes labeled $Toronto$. Fig. 3 is the updated tree.
3. Based on the header table $H_{Toronto}$, output all the cells of the form $(*, m, Toronto)$ or $(g, *, Toronto)$, which pass the average price test, where $m \in M$ and $g \in G$. Also, explore recursively the descendants of the cells of the form $(*, m, Toronto)$ or $(g, *, Toronto)$ which pass the top-k average test.

Similarly, all the cells of the form $(*, *, c)$ as well as their descendants are explored, where $c \in C$. Note that there is no information conflict on either quant-info or side-link, since every time we change the scope of examination, parent nodes copy quant-info from its child under examination and side-links are rebuilt w.r.t. the nodes currently under examination.

Step 2. Compute cells involving dimension M but no C. After examining cells in the form of $(*, *, c)$ $(c \in C)$ and their descendants, we turn to those in the form of $(*, m, *)$ $(m \in M)$ and their descendants, i.e., $(g, m, *)$ $(g \in G)$. This can be done in two steps.

(1) Roll-up quant-info to dimension M. Every leaf node in H-tree merges its quant-info into that of its parent node. All nodes labeled by a common month, should be linked by side-links and also linked to the corresponding row in the header table H. As an optimization, if the quant-info in a child node indicates that $avg'^k(child)$ passes the average measure threshold, the parent node can be marked "*top-k_OK*". Only *sum* and *count* are collected for those marked *top-k_OK*. No binning is needed, since it always passes top-k average checking. In further quant-info rolling up, parents of the nodes marked *top-k_OK* should be treated similarly.

(2) Compute cells involving M but no C. This is similar to Step 1 demonstrated before.

Step 3. For cells involving only G, the last dimension in our consideration, we consult the header table H directly for the result. It is easy to verify that the above process correctly computes the complete iceberg cube. □

Based on the above example and reasoning, we have Algorithm Top-k H-Cubing presented below.

Algorithm 1 (Top-k H-Cubing) Compute iceberg cube with *average* by top-k H-tree-based cubing.

Input: (1) A relational table, T, with attributes A_1, $\ldots$, A_m, and one measure M; and (2) an iceberg cube creation query $AvgI$, specified in section 3.1.

Output: The computed Iceberg cube, $AvgI$.

Method:

1) construct an H-tree HT, let H be the header table;
2) let $c = (\underbrace{*, \ldots, *}_{m})$, **call** htree_cubing($m$, H, c);

procedure htree_cubing(m, H, c);
{
1) **for** $i = m$ **downto** 1 **do** {
2) **for** each $a_i \in A_i$ **if** $avg'^k(M) \geq v$, **then** {
3) let $c[i] = a_i$, **if** $avg(M) \geq v$, output c;
4) **if** $i > 1$ **then** {
5) create a new header table H_{a_i}, only rows for attribute values in $A_1, \ldots, A_{i-1}$ are needed;
6) traverse side links from a_i in H do
   ⋄ collect quant-info for header table H_{a_i};
   ⋄ copy quant-info in child to parent;
   ⋄ link parents of the same label by side-links;
7) **call** htree_cubing($i-1$, H_{a_i}, c);
8) }
9) }
10) **if** $i > 1$ **then** // roll up quant-info
11) traverse side-links from $a_i \in A_i$ in H do
   ⋄ merge quant-info from children to parent;
   ⋄ link parents of the same label by side-links;
12) $c[i] = *$; // re-initialization
13) }
}

Rationale. The correctness of the algorithm is based on that it explores the iceberg cube in a divide-and-conquer style. For each iceberg sub-cube, it forms a virtual H-tree using a header table created and updated on the fly, with proper side-links and quant-info propagated to proper tree nodes. It explores further the remaining sub-cubes by recursion. □

Let us analyze the efficiency of Top-k H-Cubing.

1. Space cost. As shown in Lemma 5.1, an H-tree is a compact structure. During the computation, Top-k H-Cubing needs to create a stack of up to $(m-1)$ header tables, where m is the number of dimensions. The maximal size of the i^{th} header table is $O(\sum_{j=1}^{i-1} cardinality(A_j))$. Therefore, the total size of header tables is $O(m^2 \times cardinality(A_m))$. Even if the cube contains 20 dimensions, each with 100 distinct values, and each header slot takes 20 bytes, the total amount of memory for the header tables will still be less than $20^2 \times 100 \times 20 = 800K$ bytes, which can fit in main memory comfortably.
2. Database scan and tree traverse. Only one scan of the database is needed to construct an H-tree. The remaining computation is main memory-based tree traversal and updates of side-links and header table entries in the H-tree.
3. Data manipulation. The major data manipulations are side-link adjustment and quant-info copying/merging. Comparing with Top-k BUC, whose major work is sorting and quant-info collecting, Top-k H-Cubing's work load is lighter.
4. Pruning and optimization. Both Top-k BUC and Top-k H-Cubing use the same pruning techniques. However, Top-k H-Cubing can further use a *top-k_OK* marking technique to save quant-info computation, which cannot be applied in Top-k BUC.

5.3 Reduction of Database Scans

Even though H-tree compresses a database, one cannot assume that H-tree can always fit in main memory. To handle large databases, projections and partition can be performed first. When the H-tree for a partition can fit in memory, we turn to H-tree -based mining. Here we propose a dual projection method, which requires less memory and disk space but scans DB and each partitioned database only once.

Example 9 For computing all cuboids in Fig. 1, a dual projection scheme can be adopted as follows.

1. In the first scan, we project only on the first dimension C, as BUC, which forms a set of smaller partitions, C_i, for each distinct value c_i in C.
2. When projecting each partition C_i on the second dimension M, we project each tuple t into two partitions: a CM partition C_iM_j for city c_i and month m_j, and an M partition M_j. After this scan, both CM and M projections are generated.
3. Similarly, the projection of CM on G will produce both CMG and CG, and so on.

Thus, the processing tree of Fig. 1 is updated to Fig. 4, where the number immediately after the partition dimension shows the order that the projection is generated, followed by a number showing the order that the projection is processed. □

Dual projection has the following nice properties: (1) to compute an n-dimension data cube, dual projection scans base cell table DB as well as its

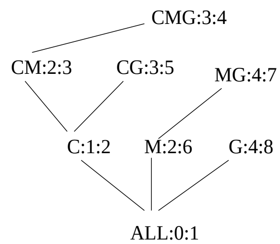

Figure 4: The Processing Tree in Top-k H-Cubing.

partitions only once, while BUC has to scan DB n times, and scan a partition on i^{th} dimension $(n-i)$ times; and (2) when scanning DB and projecting on D_1, it requires the same amount of main memory space and disk space as BUC. When scanning partition D_1 and doing dual projection on D_1D_2 and D_2, the main memory space needed is $2 \times S(dim_2)$, and the projected pages generated is about the sum of the size of two partitions D_1 and D_2, a minimal cost in both cases.

Since H-tree may substantially compress a database, in many cases the entire compressed database in the form of H-tree can fit in main memory. Then, the dual projection technique will not be needed. However, when the database is huge, the dual projection technique can be applied until the main memory-based H-trees can be constructed for the partitions.

6 Performance Analysis

In this section, we report our performance analysis on computing iceberg cubes $AvgI$.

All experiments were conducted on a PC with an Intel Pentium III 500MHz CPU and 128M main memory, running Microsoft Windows/NT. All programs were coded in Microsoft Visual C++ 6.0.

As shown in [4], the performance of Apriori for computing iceberg cube is far weaker than BUC. This is also confirmed by our experiments. Thus we concentrate on the performance comparison of Top-k BUC and Top-k H-Cubing. Top-k BUC is an extension of BUC, implemented similar to [4]. We only implemented main memory-based BUC (no external partitioning). In the experiments reported here, both algorithms have enough main memory to hold data and related structures. Thus we believe the comparison is fair. The runtime reported here includes both I/O time and CPU time.

1. Dataset generator

We designed a synthetic dataset generator, which takes parameters shown in Table 2 and generates data according to our specification. Experiments are conducted on various synthetic datasets generated by

Parameter	Meaning
n	Number of tuples in the dataset
m	Number dimensions
$card[i]$ $(1 \le i \le m)$	cardinality of the i^{th} dimension
m_max, m_min	range of measure
τ	repeat factor

Table 2: Parameters of the data generator.

the generator. The performance testing results are similar. Limited by space, except for performance with respect to the number of tuples, we report here only results on one such dataset, $\mathcal{D}$. There are 10 dimensions and 100,000 tuples in $\mathcal{D}$. The cardinality for every dimension is set to 10. The measure values are in range $[0, 99]$.

We report results on dataset $\mathcal{D}$ since it is typical and challenging. Performance study in [4] indicated that datasets with cardinality 10 are more challenging than those with cardinality 100 or 1,000.[4]

2. Computing iceberg cubes with only the COUNT measure

For all experiments reported in this subsection, the average threshold is set to 0, i.e., every cell passes the average checking.

Figure 5 shows the scalability of Top-k BUC and Top-k H-Cubing as the count threshold decreases from 100 (0.10%) to 8 (0.008%). Both algorithms are scalable, even when the count threshold is pretty low. They have comparable performance for relatively high count threshold. When the count threshold is extremely low, e.g. below 0.02%, Top-k H-Cubing is considerably faster than Top-k BUC .

Figure 6 helps us get an in-depth understanding of the scalability of the two algorithms. For both algorithms, the runtime per cell in result goes down as the count threshold goes down. That explains the scalability of them in most cases. When the count threshold goes lower than 0.02%, the runtime per cell for Top-k BUC reaches the bottom, while that for Top-k H-Cubing keeps on decreasing. This indicates that Top-k H-Cubing incurs a low overhead per cell more consistently than Top-k BUC .

3. Computing iceberg cubes with the Average measure

Figure 7 shows the runtime of the two algorithms with respect to various min_avg thresholds. The count threshold is set to 10.

As can be seen from the figure, the restriction on average helps both algorithms prune search space and thus save runtime. Top-k H-Cubing is usually much

[4] Smaller cardinalities lead to denser data cubes, which result in much larger number of cells satisfying conditions.

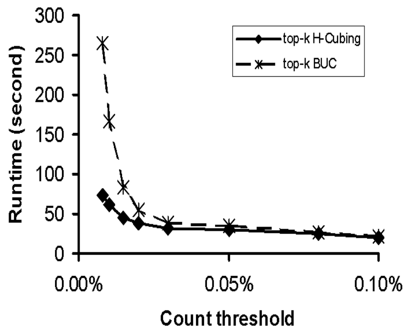

Figure 5: Scalability with respect to count threshold (no min_avg setting).

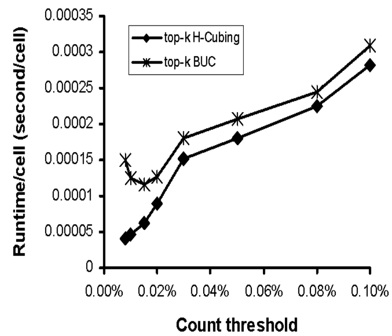

Figure 6: Runtime per cell in result.

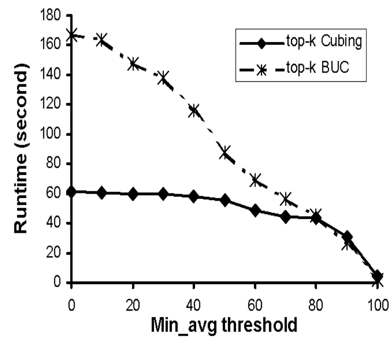

Figure 7: Scalability with respect to min_avg.

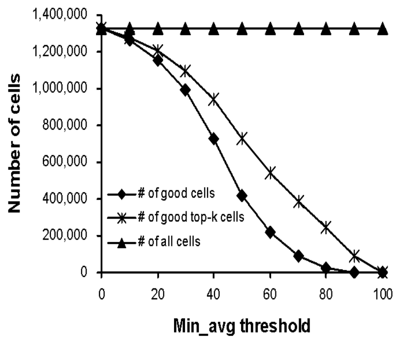

Figure 8: Number of all cells, good top-k cells and good cells with respect to min_avg.

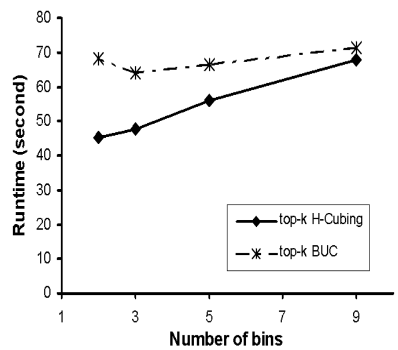

Figure 9: Scalability with respect to number of bins.

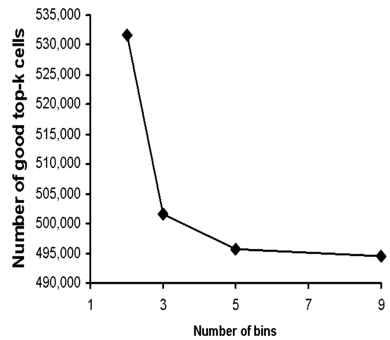

Figure 10: Number of good top-k cells with respect to number of bins.

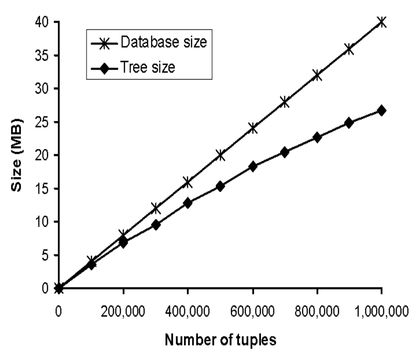

Figure 11: Size of the H-tree and the database w.r.t. the number of tuples.

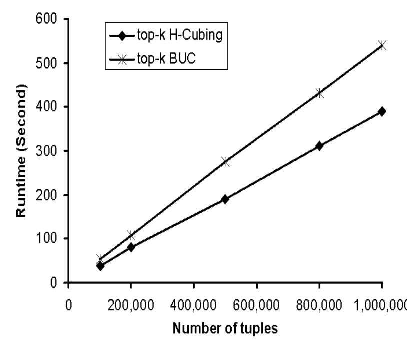

Figure 12: Scalability with respect to number of tuples.

faster than Top-k BUC . When the average threshold approaches zero, Top-k H-Cubing achieves a speed up factor of about 2.5. When the average threshold is pretty high (over 80%), i.e., most of cells are pruned, the two algorithms have similar performance. Top-k BUC catches up with Top-k H-Cubing for high average threshold for the following reasons: When the average threshold increases, the main cost of

Top-k H-Cubing, tree manipulation, could not be dramatically reduced, whereas the main cost of Top-k BUC, sorting (indicated mainly by the number of required sortings), decreases significantly.

Figure 8 helps us understand the pruning effect of the average threshold. It shows that the gap between the number of cells passing the average threshold and that of cells passing the top-k average threshold is quite small. This indicates that top-k average provides a good estimation for average and consequently a high pruning power.

4. Effect of the number of bins

Figure 9 shows the runtime of the two algorithms with respect to the number of bins. More bins incur higher cost in binning, but provide more precise estimation. The figure indicates that it does not pay to use too many bins, as binning cost outweighs the benefit when the number of bins is larger than 4 or 5. Figure 10 shows the number of cells passing top-k average checking with respect to the number of bins. At the very beginning, increasing the number of bins brings down the number of cells passing top-k average checking significantly. However, the marginal benefit becomes weak as the number of bins goes up.

In our experiments, having 5 or fewer bins tends to yield optimal performance.

5. Size of H-tree and scalability with respect to database size

Using settings identical to that for $\mathcal{D}$, we generated several new datasets with up to 1,000,000 tuples. Figure 11 shows the size of the tree with respect to the number of tuples in the database. As can be seen from the figure, the size of the tree is always smaller than the database size and the effect of compression becomes stronger when the database becomes larger.

Figure 12 shows the scalability of both algorithms with respect to the number of tuples. The figures shows that both algorithms are scalable with respect to database size.

7 Discussion

Our previous sections explored the efficient computation of iceberg cubes with the *average* measure. Here we will extend the scope to examine iceberg cubes with other complex measures and discuss related works.

7.1 Computing iceberg cubes with some other complex measures

The key of our method to solving *average* measure problem is to find a function which is weaker but ensures certain anti-monotonic property. This is also true if we wish to extend our scope to handle other complex measures. Below we examine a few typical cases and provide such transformed functions.

1. Compute iceberg cubes whose AVG is no bigger than a value.

Our *AvgI* query is to compute the iceberg cubes whose average is no less than a value v. Can we compute icebergs whose AVG is no bigger than v? Similar to finding $avg^k(c) \geq v$ in *AvgI*, here we can find a weaker, anti-monotonic auxiliary function $avg_k(c) \leq v$, where $avg_k(c)$ is the average of the bottom-k base cell values of a cell c, and the bottom-k base cells are the first k cells when all the base cells in c are sorted in the *value-ascending order*. Then, the bottom-k average Apriori property holds because if the bottom-k average of a cell c is no greater than v, then the average of any of its descendants (with at least k nonempty base cells) cannot be greater than v. Thus, the methods discussed in sections 4 and 5 can be easily extended to this case.

2. Compute iceberg cubes with the AVG constraint only.

The Having-clause in *AvgI* contains both AVG and COUNT constraints. *What will happen if we have only the average constraint, i.e.,* AVG(*price*) $>= v$*?* This is equivalent to $k = 1$ and thus the methods discussed before are still applicable. Since $k = 1$, the top-k average testing becomes effectively the testing of MAX(*price*) $>= v$. Notice that a relatively large k will serve as a good constraint to cut every cell that contains too small number of nonempty base cells, or whose average price cannot pass the threshold. When k is reduced down to 1, the power of the constraint is also reduced to minimum since if the value of a base cell c_i is no less than v, all of c_i's ancestors cannot be pruned. That is why we start our discussion on more useful cases where $k > 1$. Notice this does not imply the AVG-only cutting is useless since the cutting can still be effective if v is substantially larger than the average of all the base cells.

3. Compute iceberg cubes whose measure is SUM of positive and negative values.

Suppose our iceberg cube query is similar to *AvgI* except AVG(price) is replaced with SUM(profit), as shown in query Q_2 of Example 1. That is, its HAVING-clause becomes HAVING SUM(*profit*) $>= v$ AND COUNT($*$) $>= k$.

Notice that SUM(M), when M is either nonnegative or negative, such as profit, is not anti-monotonic. This is different from the case where M is nonnegative, which is anti-monotonic and can be computed by a BUC-like method directly, as shown in [6, 4]. However, when M can also be negative, it is unfortunate that we cannot even use a weaker, auxiliary function $sum^k(c) \geq v$, where $sum^k(c)$ is the top-k sum of a cell c, and the top-k sum is the sum of the first k values in c when all the base cell values in c are sorted in value-descending order. This is because even when top-k sum invalidates $sum^k(c) \geq v$, adding remaining values in a cell may still validate $sum(c) \geq v$.

However, we can use a simple weaker, antimonotonic auxiliary function as follows to handle it,

- $p\_sum(c) \geq v$, if $p\_count(c) \geq k$, where $p\_sum(c)$ is the sum of all the non-negative base cell values in cell c, and $p\_count(c)$ is the number of nonempty non-negative base cells in cell c, and
- $sum^k(c) \geq v$, otherwise (i.e., $p\_count(c) < k$).

Then, the methods discussed in sections 4 and 5, can be easily extended to this case, by keeping two additional counters, $p\_sum(c)$ and $p\_count(c)$, and a small number of bins for negatives.

4. Compute iceberg cubes with measures like *max, min, count,* and *p_sum,* where *p_sum* means the sum of all nonnegative base cell values in a cube.

Since conditions like $count(M) \geq v$, $max(M) \geq v$, $min(M) \leq v$, and $p\_sum(M) \geq v$ generate anti-monotonic cubes, we can use either a BUC-like method (such as [4]) or the H-Cubing method introduced here to compute it without seeking for an auxiliary function. Notice if the condition is changed to $max(M) \leq v$, we can use a weaker, anti-monotonic auxiliary function $min(c) > v$ since if a cell c's minimum base cell value is no greater than v, one cannot find in c or in its descendants whose maximum base cell value can be less than or equal to v. Similarly, a condition $min(M) \geq v$ can be tested by an auxiliary, anti-monotonic function, $max(c) < v$.

5. Compute iceberg cubes having conjunctions of multiple conditions with different measures.

In this case, one can explore combined, stronger anti-monotonic constraints to reduce the portions of iceberg cubes that have to be computed. Since our iceberg cube computation requires some parameters, such as v and k in the case of HAVING AVG($price$) $>= v$ AND COUNT($*$) $>= k$, and those parameters may likely be available only at "query time", one may wonder how the iceberg cube precomputation may help? Our view is as follows. Since the computation of a (background) high-dimensional cube is prohibitively expensive, it is more realistic to precompute one of its corresponding iceberg cubes by setting a set of reasonably low bound parameters, and consider the aggregated cells below such low bound(s) as trivial. For example, one may precompute an iceberg cube corresponding to some minimal average price and minimal count and use the precomputed iceberg cube to support most of *interesting* queries.

6. Finally, one may ask, "can we efficiently compute iceberg cubes with any complex measures?"

Although we have worked out some methods for efficient computation of iceberg cubes with several complex measures, this by no means implies that iceberg cubes with any complex measures can be computed efficiently. It seems there is no general answer to efficiently compute iceberg cubes with holistic measures, such as median, mode, and rank. Even for some complex algebraic measures, such as *standard_deviation* and *variance*, more research is needed to find easy to compute, not too weak, anti-monotonic functions in order to successfully perform efficient computation of such iceberg cubes.

7.2 Related work

Since the introduction of the concept of data cubes [8], efficient computation of data cubes has been a theme of active research, with many interesting approaches proposed, such as [1, 10, 16, 14, 4]. As shown in [14, 4], computation of high dimensional, large data cubes are challenging due to the huge sizes of cuboids that could be generated by multi-dimensional group-bys. Thus, computing iceberg cubes rather than complete cubes, proposed by [4], motivated by iceberg query computation [6], is a promising direction. [4] proposed an efficient cube computation method BUC, which has been shown highly efficient for computing not only iceberg cubes but also complete cubes. However, for computing average iceberg cubes, [4] does not suggest an effective method. Thus, this study extends the scope of computation to iceberg cubes with complex measures, including the ones discussed above.

Our proposal of computing iceberg cubes with complex measure has also been influenced by the previous works on constraint-based mining of association rules, such as [15, 12, 3, 11, 7, 13]. [12] introduced the notion of anti-monotonicity and studied methods for effective push of anti-monotonic constraints into association mining. Unfortunately, the complex measures studied here are not anti-monotonic. A recent study by [13] introduced a new class of constrains, called *convertible constraints*, that includes the constraint "$avg(c) \geq v$" for association mining. However, the method proposed there, i.e., evaluation of average in a value-sorted order, is difficult to realize in a multi-dimensional data cube space. Therefore, we believe our top-k average is a novel solution to this problem, and its effectiveness is demonstrated in our performance study.

The major algorithm proposed here for efficient computation of iceberg cubes with complex measures is Top-k H-Cubing, which is based on a hyper-tree structure, H-tree. The H-tree structure is influenced by the FP-tree structure, proposed in [9]. However, besides some structure differences between the two, a crucial difference is at the computation process: FP-growth mines frequent patterns by recursively constructing and mining conditional (or projected) databases; whereas Top-k H-Cubing uses one H-tree structure in the entire computation, which saves both space and time. Our experiments show that due to recursive construction of conditional FP-trees, the FP-growth method has weaker performance than both Top-k BUC and Top-k H-Cubing at computing iceberg cubes in most cases.

The best algorithm that Top-k H-Cubing has been competing with is Top-k BUC, a revised BUC [4] for computing top-k average. A performance analysis has been reported in section 6. Based on our view, the major strength of Top-k H-Cubing is at (1) compressed database: H-tree structure, (2) pointer adjustment instead of partitioning and tuple sorting, and (3) the exploration of shared precomputation of quant_info.

Finally, we should note that this study did not compare Top-k H-Cubing with the multiway array aggregation method developed by Zhao et al. [16]. This is because, as pointed out in [4], the multiway array aggregation method cannot take advantage of iceberg cube constraints in computation, and it encounters difficulties for computing iceberg cubes with high dimensionality. However, when the cube contains only a small number of dimensions, it could still be a rival of Top-k H-Cubing in performance unless some integrated processing is considered. More study is needed in this direction.

8 Conclusions

In this paper, we have studied issues on efficient computation of iceberg cubes with some popularly encountered complex measures and proposed some efficient computation methods. It contributes to iceberg cube computation in two aspects: (1) a methodology is developed that derives a weaker but anti-monotonic condition for testing and pruning search space, especially, it shows that the top-k average pruning is an effective technique for computing average iceberg cubes; and (2) instead of simple extension of two previously studied methods, Apriori and BUC, to Top-k Apriori and Top-k BUC, an interesting hyper-tree structure, called H-tree, is designed and a new iceberg cubing method, Top-k H-Cubing, is developed. Our performance study shows that Top-k H-Cubing is a promising approach for efficient computation of iceberg cubes.

Although interesting progress has been made for efficient computation of iceberg cubes with some complex measures, as shown in section 7, efficient computation of iceberg cubes with some other complex measures is still an open problem. Moreover, the application of the H-tree structure and its computation method to other OLAP and data mining tasks may deserve further attention.

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. VLDB'96.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. VLDB'94.

[3] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining on large, dense data sets. ICDE'99.

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

[5] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26:65–74, 1997.

[6] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. VLDB'98.

[7] G. Grahne, L. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. ICDE'00.

[8] 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.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. SIGMOD'00.

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

[11] L. V. S. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. SIGMOD'99.

[12] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. SIGMOD'98.

[13] J. Pei and J. Han. Can we push more constraints into frequent pattern mining? KDD'00.

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

[15] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. KDD'97.

[16] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. SIGMOD'97.