

Efficiently Computing the Top N Averages in Iceberg Cubes

Pauline LienHua Chou

Xiuzhen Zhang

School of CS & IT, RMIT University
GPO Box 2476V, Melbourne 3001, Australia
Email: {lchou, zhang}@cs.rmit.edu.au

Abstract

Data cubes are an enabling approach for efficient on-line analytical processing (OLAP) systems. Iceberg cubes are special cubes consisting of the aggregates of multi-dimensional groups that satisfy user-specified thresholds. When there are a relatively large number of dimensions, the number of groups in an iceberg cube is huge. End users cannot fully understand the aggregate results or directly use them to make a decision. Approaches for the efficient computation of the top n groups have been proposed and have been shown to work well on iceberg cubes with simple aggregate functions such as COUNT and SUM. In this paper, we study the efficient computation of the top n groups with the complex aggregate function AVERAGE. As the average of a sub-group does not increase/decrease monotonically with its super-group, AVERAGE constraints cannot be used directly for pruning. We propose a new technique upper-bounding average which is anti-monotonic and can be used for low-cost effective pruning. Based on a tree structure representing groups, search and pruning techniques are developed, and an algorithm is proposed to compute the top n averages efficiently.

1 Introduction

Decision support places different requirements on database technology compared to traditional data processing applications. To this end, data warehouses, which support on-line analytical processing (OLAP) (Chaudhuri & Dayal. 1997), are built separately from operational databases, which support on-line transaction processing (OLTP). While OLTP applications automate routine operations of an organisation, OLAP focuses on queries from knowledge workers — executives, managers, analysts — to make better and faster decisions.

On-Line Analytical Processing (OLAP) Systems enable fast on-line analysis of huge collections of data with multi-dimensions and numeric measures. They provide perspective summarised views to data to facilitate analysis, which supports high level decision making in an organisation. An example Sales relation with City, Month, and CustomerGroup dimensions, and a measure Amount that records the volume of sales, is illustrated in Table 1.

Month, City, and CustomerGroup are the three dimensions and Amount is the numeric measure. OLAP queries are typically aggregations of the numeric measure in terms of different dimensions. For the Sales relation above, queries might be

What was the total sales for January
in Sydney?

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Month	City	CustomerGroup	Amount
Jan	Melbourne	Student	850
Jan	Perth	Student	400
Feb	Sydney	Business	577
Mar	Melbourne	Employee	200
Feb	Perth	Household	305
May	Melbourne	Farmer	183
		• • • •	

Table 1: An Example Sales Relation

Which customer group in Melbourne
had higher average sales, students
or employees?

Note that queries might involve any combinations of the three dimensions. Note also that queries typically involve aggregations over the numeric measure Amount, such as sum, count, or average.

The data cube was proposed by Gray et al. (Gray, Chaudhuri, Bosworth, Layman, Reichart, & Venkatrao 1997) to improve the performance of OLAP systems. A data cube consists of the aggregates over the groups of all combinations of dimensions. The materialisation of the multi-dimensional data can be used to efficiently answer OLAP queries. The CUBE for the sales relation is as follows:

```
SELECT city, month, group, AVG(amount)
FROM Sales
CUBE BY city, month, group
```

CUBE BY is the multi-dimensional extension of GROUP BY in SQL. The fully materialised data cube would consist of the average sales of all groups by any of City, Month, or CustomerGroup. Month takes 12 values. Suppose that City takes 3 values and CustomerGroup takes 4 values. Each dimension also contains the special value "ALL" (denoted as *). Thus the data cube consists of $13 \times 4 \times 5 = 260$ groups. Materialised data cubes can greatly improve the efficiency of OLAP queries. For instance, the second query on sales can be answered using the following two groups:

```
(* , Melbourne, Student, AVG(Amount))
(* , Melbourne, Employee, AVG(Amount))
```

As the number of groups in data cubes grow exponentially with the number of dimensions, when the number of dimensions is high, the number of groups in a cube is large, and it becomes expensive to fully materialise data cubes (Beyer & Ramakrishnan 1999,

Harinarayan, Rajaraman, & Ullman 1996). Iceberg cubes and Iceberg queries (Beyer et al. 1999, Fang, Shivakumar, Garcia-Molina, Motwani, & Ullman 1998) were proposed to overcome such difficulties by materialising only non-trivial groups where the aggregations are above certain user-specified thresholds.

Consider an example iceberg cube for the Sales relation:

```
SELECT city, month, group, AVG(amount)
FROM Sales
CUBE BY city, month, group
HAVING COUNT(*) >= 100 and AVG(amount) >= 500
```

The constraint thresholds are specified in the HAVING clause. Only groups that contain at least 100 tuples are considered meaningful groups. Such groups are called large groups, and the *count threshold* for large groups is 100. In this iceberg cube, only large groups with an average sale of at least 500 are included. With these thresholds, we expect to have a much smaller number of groups in the iceberg cube than in the entire cube.

For iceberg cubes with very high dimensions, the number of groups can still be too large for end users to carry out sensible analysis. In our experiments, on a dataset with 80,000 tuples, 10 dimensions, and cardinality of 100 (all dimensions having the same cardinality), when the support threshold is 100, there are 123,881 groups in the iceberg cube. Decision-makers can hardly draw sensible conclusions from this iceberg cube. Of course we can sort the groups in the iceberg cube into descending order of average sales so that the most significant groups appear at the top, but the computation would be expensive.

Typically, executives and managers are only interested in the most significant groups. In our previous work (Chou & Zhang 2002), the concept of *top-n queries* was proposed, where, given a user-specified n , only the top n groups in an iceberg cube are computed. Coupled with different aggregate functions, we can have top- n functions such as TOP_SUM(amount, n), TOP_COUNT(*, n) and TOP_AVG(amount, n). An example top- n query on the sales iceberg cube with AVERAGE might look like

```
SELECT city, month, group, TOP_AVG(amount, n)
FROM Sales
CUBE BY city, month, group
HAVING COUNT(*) >= 100 and AVG(amount) >= 500
```

In our previous study (Chou & Zhang 2002), the large-group tree (LG-tree) was proposed to keep large groups that pass the count threshold. Effective search and pruning strategies were developed to efficiently compute the top n groups from the LG-tree. However, we addressed the computation of only simple top- n aggregate functions such as TOP_SUM or TOP_COUNT.

Although the concept of TOP_AVG was introduced in our previous work, its efficient computation was not studied. Simple top- n aggregate functions, for example TOP_SUM, are anti-monotonic: given a group g , the sums of sub-groups of g are no greater than the sum of g . So if a group g fails a given threshold s , $\text{TOP_SUM}(g) \geq s$, all of its sub-groups can be pruned. Based on the same reasoning, the heuristics for determining search order for such aggregate functions can easily be decided. Since TOP_AVG is not anti-monotonic, it presents more difficulties for search and pruning.

Experiments showed that on a skewed dataset, our approach is 8 times faster than the naive approach based on the best known iceberg cubing algorithm (Han, Pei, Dong, & Wang 2001) for various n values; and is more than 10 times faster with count

threshold below 0.1%. Also, our approach achieved linear scalability.

In this study, we further design new techniques for efficiently computing TOP_AVG of an iceberg cube. Based on the LG-tree, effective pruning and search strategies are developed for efficiently computing the top n averages in iceberg cubes. Bounding techniques are developed for pruning groups using the average value that lies in a branch of the LG-tree, and a new search strategy is used for computing TOP_AVG so that the most promising branch is searched first. Compared with the most recent work for computing iceberg cubes with AVERAGE (Han et al. 2001), which has been shown to outperform other approaches including the Apriori-based approach (Agrawal & Srikant. 1994) and BUC (Beyer et al. 1999), our pruning strategy is expected to involve less computation and less memory usage.

2 Computing iceberg cubes

Iceberg cubes consist of groups which are sets of tuples having the same grouping dimensional values. A group is denoted by the grouping dimensional values. A group consisting of m dimensional grouping values is called an m -dimensional (m - d) group. For example, (*, Melbourne, *) represents a 1- d group that comprises tuples that have the value “Melbourne” for the “City” dimension and any values for “Month” or “CustomerGroup”.

Sub-groups of an m - d group g have the same m dimensional values as the group and dimensional values for the other dimensions where g takes “ALL”. A sub-group contains a subset of tuples of the super-group. For example, (Jan, Melbourne, *) is a sub-group of (*, Melbourne, *). Among all the tuples that form the group (*, Melbourne, *), a subset of tuples that not only have “Melbourne” for the “City” dimension but also “Jan” for the “Month” dimension form the sub-group (Jan, Melbourne, *). As is mentioned previously, an iceberg cube consists of groups that pass the constraint thresholds specified in the HAVING clause of the iceberg query. This section presents the current techniques for the efficient computation of iceberg cubes.

2.1 Bottom-up cubing

The iceberg cube concept and the bottom-up cubing (BUC) approach were proposed by Beyer and Ramakrishnan (Beyer et al. 1999). To search for groups satisfying specified thresholds, BUC recursively partitions the database according to dimension-values. To speed up computation, the dimensions are considered in the cardinality-descending order. Partitions are searched depth-first.

To achieve pruning, BUC proceeds from the most aggregated “ALL” group (the bottom) to the least aggregated groups (the top) with specific values for each dimension. The basic pruning function is COUNT. The anti-monotonic property of COUNT is used for pruning groups (partitions):

if a group g is not a large group, that is $\text{COUNT} < \delta$, all its sub-groups must also have $\text{COUNT} < \delta$ and so g and all its sub-groups can be pruned from further search.

For instance, for the Sales relation, assuming the three dimensions in cardinality-descending order are City, Month, and CustomerGroup, denoted as C-M-G, the order of BUC searching groups is shown in Figure 1. Note that due to the depth-first search strategy, the CM groups, which are searched under C groups, are checked before the M groups. As a result,

even if a group (m_1) fails the constraint thresholds, its sub-group (c_1, m_1) is still searched and computed.

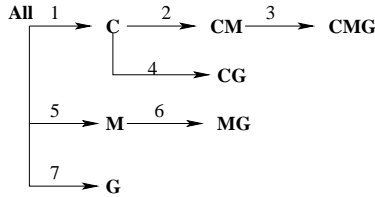


Figure 1: *BUC Search Order*

In addition to COUNT, with aggregate functions such as MIN, MAX, or SUM, BUC makes use of the anti-monotonic property of these aggregate functions to efficiently compute iceberg cubes. However, for a complex, non-anti-monotonic aggregate function such as AVERAGE, BUC can not be directly used.

2.2 Iceberg cubes with AVERAGE

The AVERAGE function is not anti-monotonic: for a group g with $AVG(\text{Amount})=500$, the averages of its sub-groups may be larger than or less than 500. As a result, in computing iceberg cubes with the aggregate function AVERAGE, a constraint such as $AVG(\text{Amount}) \geq 500$ can not be directly used for pruning.

To address this limitation, Han et al. (Han et al. 2001) proposed a weaker but anti-monotonic function *top-k average* for pruning. Given the HAVING clause in an iceberg cube specification:

HAVING COUNT(*) $\geq k$ and $AVG(\text{amount}) \geq v$

The top- k average of a group g is the average of the tuples having the top k largest amount in the group. The top- k average of a group g , denoted as $avg^k(g)$ is anti-monotonic:

If the top- k average of a group is less than v , the top- k average of all its sub-groups will also be less than v .

As the real average of a group is definitely less than its top- k average, if the top- k average of a group is less than v , the group and all its sub-groups can be safely pruned.

Keeping track of the top- k tuples for each group in order to compute the top- k average for pruning is expensive, especially if k is large. To reduce the cost of the top- k average computation, a binning technique is used. Basically, a set of bins for a group, g , is created. For each bin, sum and count values are kept. A tuple in g is added to one of the bins depending on its measure with regard to v . A special bin, the *large bin*, is reserved for tuples with the measure larger than v . The rest of the bins, *small bins*, are defined by a range specified using v . For example, a set of 5 bins can be created for a group with the value ranges defined as $(\infty, v]$, $(v, 0.9v]$, $(0.9v, 0.8v]$, $(0.8v, 0.5v]$, and $(0.5v, -\infty)$.

To calculate the top- k average of the group, the set of bins is checked in value range descending order until the sum of the top k tuples is obtained for the computation. If the count in the large bin is larger than k , the group passes the top- k average constraint, and there is no need to store the small bins. Otherwise, the sum and count values in the next largest bin are used. In the last bin being used where only part of the tuples, let the number be i , are needed to make up the top k tuples, the i tuples are assumed to have the measure with the value of the upper boundary of the bin. The accumulated sum divided by k is treated

as the top- k average of the group. Note that the value may be larger than the actual top- k average if not all tuples in the last bin are used.

2.3 H-cubing

The H-tree is a hyper-tree structure, which comprises a header table and a tree structure. In the header table, each entry records the quantity information count and sum for each dimension-value pair. The tree structure is a prefix tree. The H-tree for the iceberg cube with AVERAGE on the Sales relation is shown in Figure 2.

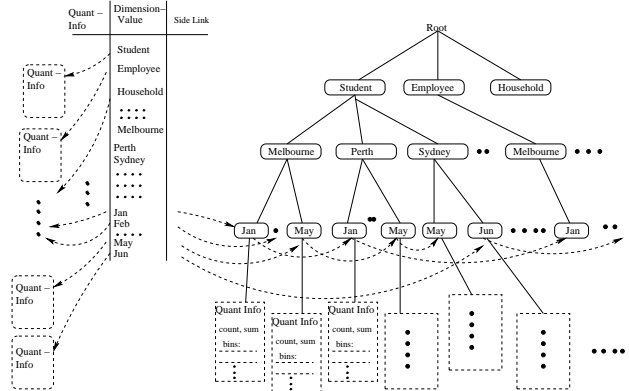


Figure 2: *An example H-tree.*

The database is scanned once to construct the tree and the header table. In the example Sales relation in Figure 1, following the CustomerGroup-City-Month order, the first tuple $t_1=(\text{Jan}, \text{Melbourne}, \text{Student}, 850)$ is inserted into the tree and forms the leftmost path. Quantity information is kept in the leaf node and is used to update the quantity information for the “Jan”, “Melbourne” and “Student” entries of the header table. The second tuple $t_2=(\text{Jan}, \text{Perth}, \text{Student}, 400)$ is similarly inserted into the tree. Note that it shares a path with t_1 for “Student”. Leaf nodes with the same dimension-value are linked together and to the header table by side links. Note that to maximise sharing, nodes are organised in dimension cardinality-descending order. Note also that, in order to use top- k average for pruning, quantity information such as count, sum(amount) and a set of bins need to be kept in both the header table and tree.

H-cubing searches an H-tree for groups that satisfy given count and average thresholds, in the reverse order in which they were constructed. With our example, all groups involving dimension Month are computed and the groups passing the threshold check are output. For example, the groups based on “Jan”, are $(\text{Jan}, *, *)$, $(\text{Jan}, c, *)$, $(\text{Jan}, *, g)$, and (Jan, c, g) . $(\text{Jan}, *, g)$ denotes all groups grouped by January and some CustomerGroup value. H-cubing recursively searches the H-tree depth-first. After computing all groups involving dimension Month, those involving City but not Month and those involving only CustomerGroup are computed accordingly.

Top- k average is used for pruning: If the top- k average of a group fails the constraint threshold, the group and its sub-groups are removed from search. As noted before, to use top- k average for pruning, all quantity information, including the sum, count, and a set of bins are recalculated and updated for each header entry and nodes within each recursion.

Experiments have shown that H-Cubing for iceberg cubes with AVERAGE outperforms in most

cases the BUC with top- k average function for pruning. However, these studies only focus on the computation of entire iceberg cubes. None of them have addressed the problem of oversized iceberg cubes which consists of huge number of groups and are impossible for end-users to understand.

3 Computing the top n groups in iceberg cubes

In our previous work (Chou & Zhang 2002), we proposed to compute only the top n groups in iceberg cubes where n is specified by users. Especially, we studied the efficient computation of the top- n groups in iceberg cubes with simple anti-monotonic functions. A compact data structure and efficient algorithm are proposed for computing the top n groups efficiently. This efficiency is achieved by an appropriate search order, combined with the dynamic adjustment of the constraint threshold. Our discussion focuses on the simple function TOP_SUM.

3.1 Large group tree

Given an iceberg query involving TOP_SUM as follows:

```
SELECT city, month, group, TOP_SUM(amount,80)
FROM Sales
CUBE BY city, month, group
HAVING COUNT(*) >= 50
```

Given the constraint “COUNT(*) \geq 50”, groups that consist of at least 50 tuples are called *large groups*. Only combinations of frequent dimension-values with count of at least the count threshold can produce large groups. Based on this observation, in order to represent the space of possible large groups, we construct the large-group tree (LG-tree), whose nodes are frequent dimension-values. We call such frequent dimension-values *large dimension-values*.

An LG-tree is composed of a header table and a tree structure. Rather than recording all dimension-values of the dataset, an LG-tree only keeps the frequent dimension-values, which are obtained by scanning the dataset once, and each is allocated an entry in the header table.

The dataset is scanned a second time to construct the tree. The frequent dimension-values obtained from the first scan are sorted into count-descending order. In scanning the dataset, the projection of the frequent dimension-values of each tuple are inserted into the tree. A tuple being inserted later shares the same path with previous tuples if its projected dimension-values are the same as previous tuples. The quantity information is accumulated in each node while each dimension-value is being inserted. The nodes carrying the same dimension-values are linked together by side links and the whole list is connected to the corresponding header table entry.

The LG-tree for our example relation is shown in Figure 3. Suppose by scanning the database once, the frequent dimension-values in count-descending order are Melb, Jan, Perth, Student, Staff, Sydney, ... The projection of the frequent dimension-values of the first tuple is Melb, Jan, Student and forms the leftmost path of the tree. The second tuple, whose the only frequent dimension-value is “Jan”, forms the branch Root—Jan and so on. In this process, sum and count are accumulated in each node. Side links are formed to link the same grouping dimension-values. In the leftmost branch of the final tree, there are 20 tuples in group (Jan, Melb, Student) and 2 tuples in group (Jan, Melb, g), where g denotes some CustomerGroup value not

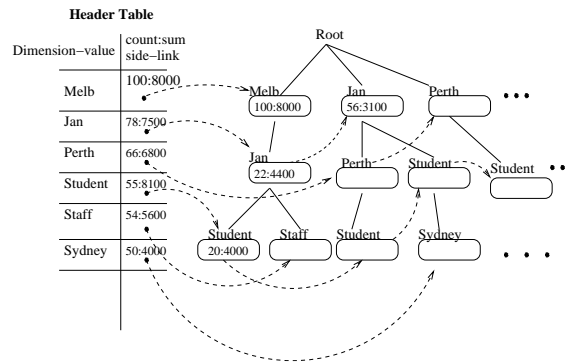


Figure 3: An example LG-Tree

frequent. These 2 tuples share the left most path from the root up to the “Jan” node with the 20 tuples of (Jan, Melb, Student).

The LG-tree represents the space of possible large groups. Large 1-dimensional groups can be directly derived from the header table. For example, the entry for “Student” in the header table represent the 1- d group (*, *, Student), denoted as (Student). Higher-dimensional groups comprise all paths containing the grouping dimension-values. Let us look at the grouping dimension-value “Student”. Following the side links for “Student”, all groups containing “Student” (except the 1- d group (Student)) are formed by traversing the paths from the node “Student” to the root, which are (Jan, *, Student), (*, Melb, Student), (Jan, Melb, Student), (*, Perth, Student), and (Jan, Perth, Student). In this way, we search every possible large group. Note that with the distinct values “Student” and “Jan”, Student-based groups and Jan-based groups have no common sub-groups. The group searched under the 1- d group (Jan, *, *) is (Jan, Melb, *). In this way, the LG-tree represents all possible large groups without repetition.

Our experiments show that compared with the H-tree, the LG-tree usually has fewer nodes. Heuristically, with the dimension-values in count-descending order, tuples share many nodes and the LG-tree compresses the dataset significantly. As only the frequent dimension-values are inserted to the LG-tree, it takes less time to construct the tree. Note that the design of the LG-tree is inspired by the FP-tree structure (Han, Pei, & Yan 2000), which is used in another problem domain called Frequent Pattern Mining. The most important difference between the LG-tree and the H-tree, however, is the way we search and prune the LG-tree, as is shown in the next section.

3.2 TopN-Miner

To search only the top n groups in an LG-tree without computing the entire iceberg cube, we push the constraint as early as possible into computation and use it to prune the search space and speed up search.

The constraint specified in the HAVING-clause of the query is used for computing all groups in an iceberg cube. However, computing the top n groups has a different constraint on those n groups although such a constraint is not explicitly specified in the query. An optimal constraint would be the aggregate of the top n th group. For example, given TOP_SUM(amount, 80) in the SELECT-clause of the top- n query, the aim is to find the top 80 groups with the largest sums. If we know the sum of the top 80th group and use this value as the constraint threshold for pruning, we can get the top 80 groups that are our answers and prune the rest of the groups at once. Any groups with sum

less than this value are definitely not groups of the top 80 sums.

Unfortunately such an optimal constraint cannot be obtained directly. An alternative is to derive a new constraint closer to the optimal during the search process. Each time the constraint threshold becomes closer to the optimal, the pruning becomes more effective, resulting in faster computation.

In our example, there is no initial SUM constraint. As we search for the top-80 groups, we keep up to 80 groups with the largest sums found so far in a list L . The 80th largest sum (the least sum value), let it be $L.\alpha$, becomes the new SUM threshold which is used for pruning in further computation. Each time a group with a larger sum than $L.\alpha$ is found, we push the group into L . At this point, $L.\alpha$ is updated accordingly and becomes closer to the optimal SUM threshold. The search finishes when $L.\alpha$ is equal to the optimal threshold which prunes all groups not of the top 80 sums, and the global top n groups are in L .

Combined with our search strategy, we push the constraint as early as possible into the computation of the top n groups in iceberg cubes. The earlier we push the constraint into computation, the more severe the pruning is and the more quickly the search completes.

3.2.1 Search order

Our search strategy applies heuristics to search for groups with the possible large sum values and push them into L as early as possible so that a stricter constraint can be used sooner for more severe pruning.

A depth-first-based search with lightweight level-wise checking and pruning is identified as the most appropriate strategy. The way the groups are found in an LG-tree and the different characteristics of different datasets are both taken into consideration by our search strategy. While groups are more easily searched depth-first, different datasets have different distributions of the top n groups. On datasets with dimension-values randomly distributed, the top n groups are randomly scattered along different dimensions; whereas on skewed datasets, the top n groups may reside along only a few dimension-values.

Figure 4 uses a prefix tree to demonstrate our search strategy. Suppose that the values are the large dimension-values on the LG-tree. The number indicates the order in which groups are considered for searching the sub-groups. As can be seen, it is depth-first based. The arrows in each triangle illustrate the level-wise checking of $(m+1)$ - d sub-groups of an m - d group. After an m - d group, g , is found, instead of checking a single $(m+1)$ - d sub-group of g , all the $(m+1)$ - d sub-groups are checked at once. The level-wise checking of all the $(m+1)$ - d sub-groups of g is carried out before one of the $(m+1)$ - d sub-groups is next considered for the search for higher dimensional sub-groups.

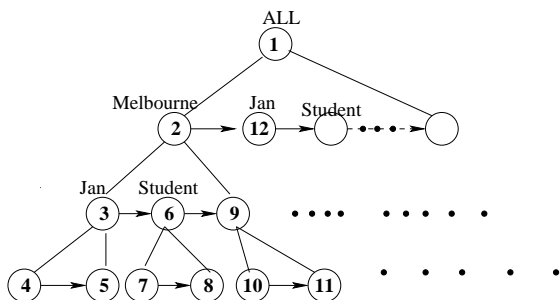


Figure 4: the Search Strategy

It is observed that the sum of a group is greater than or equal to the sum of any of its sub-groups; also, heuristically, 1-dimensional groups have larger sums than their higher-dimensional sub-groups. Based on these observations, we first initialise L with large 0- d and 1- d groups. Among the 1- d groups, we first search the sub-groups of group (Melbourne) (that is (*, Melbourne, *, *, ...)), which has the largest sum value. During the level-wise checking of the sums of the 2- d sub-groups of (Melbourne) such as (Jan, Melbourne), (Melbourne, Student), if there are values greater than $L.\alpha$, L and $L.\alpha$ are updated accordingly. With the updated L and a tighter threshold $L.\alpha$, we then search the sub-groups of (Jan, Melbourne), which is the largest 2-dimensional group of (Melbourne), and so on until we have finished all the sub-groups of (Melbourne).

Note that we do not impose an order on dimension values in which groups are considered for searching for sub-groups. Instead, we dynamically adjust the order in which the groups are considered in SUM descending order. Such an order ensures that groups having sub-groups with larger sums are considered first. It is observed that in an LG-tree, the sub-groups of a group, g , are determined by their positions in the tree structure, so the same sub-groups of g will be retrieved regardless of whether g is considered before or after other groups. Therefore, by considering groups in their SUM descending order, we search for sub-groups with large sums early without missing any groups.

Such search order is efficient for finding the top n groups for the following reasons. First, level-wise checking of $(m+1)$ - d sub-groups of an m - d group, g , is of low overhead since all the $(m+1)$ - d sub-groups of g are already in the sub-header table, which is constructed for deriving sub-groups of g . They are likely to have larger sums than their sub-groups, therefore, they are pushed into L immediately at the level-wise checking process. Second, the sums of the sub-groups can only be at most that of their super-group, so sub-groups with larger sums are likely found under the group with the largest sum. Hence, by considering the groups in sum descending order, higher dimensional groups with large sums are found early, and pushed into L early.

4 Bounding the averages of groups

In (Chou & Zhang 2002), our discussions were based on the computation of the top n groups with anti-monotonic functions such as TOP_SUM. This study further explores efficient computation of the TOP_AVG in iceberg cubes. In this section, we derive a new anti-monotonic condition for effectively pruning iceberg cubes with AVERAGE.

Recall that H-Cubing uses the top- k average as an anti-monotonic condition for pruning iceberg cubes with AVERAGE. In contrast to the top- k average, we propose a more efficient anti-monotonic condition for pruning iceberg cubes with AVERAGE. We call this technique Upper Bound Average (UB-average). The UB-average is space efficient because it does not need additional space such as that used in the binning technique in top- k average. In addition, it is easier to compute than the average of top- k tuples in a group.

The UB-average bounds the averages of a group and its sub-groups, and uses this upper bound for pruning. A group and its sub-groups can be bounded if they are constituted in the way described below: First, a group, g , is formed by multiple disjoint sets of tuples. To assist our explanations, we call such a set of tuples a *bundle*. Second, every sub-group of g , g_{sub} , is formed by some bundles among all the

selected as the v_{upper} of the group. Note that it is larger than the average of the group.

When searching for the sub-group (Jan, Student) of (Student), the 3 bold branches in the LG-tree are considered since these paths contain “Jan” as an ancestor of “Student”. Therefore, (Jan, Student) is constituted by the 3 branches. As can be observed, the 3 branches that constitute the sub-group (Jan, Student) come from the 4 branches that constitute (Student). The largest among these 3 averages 200.0 is selected as the v_{upper} of the sub-group (Jan, Student). Note that it is smaller than the v_{upper} of (Student).

From this example we can observe that the v_{upper} of (Student) is at least the v_{upper} of (Jan, Student). Therefore, given that the v_{upper} of (Student) is 210.0, the averages of all the Student-based sub-groups are at most 210.0. Suppose that a constraint of “HAVING average ≥ 250.0 ” is given, the v_{upper} of (Student) fails that constraint. (Student) and all its sub-groups will also fail the constraint. With such a v_{upper} of (Student), we know that (Student) as well as all its sub-groups do not need to be checked.

4.2 The effectiveness of UB-average

The only information required by the “UB-average” pruning technique is the v_{upper} values of groups. As will be shown in the algorithm, the v_{upper} of groups can be easily obtained at the same time as they are searched. When searching for $(m + 1)$ - d sub-groups of an m - d group, the v_{upper} values of the $(m + 1)$ - d sub-groups can be easily obtained since the paths comprising the $(m + 1)$ - d sub-groups have to be traversed in order to construct the conditional header table for the sub-groups.

The top- k average technique, on the other hand, requires much more information to be stored. In addition to sum and count, a set of bins is required for all the entries in the header table and for all the leaf nodes. When searching for $(m + 1)$ - d sub-groups of an m - d group, quantity information has to be rolled up from child nodes to the parent nodes. In cases where the set of bins in a child node fails the top- k average test, the set of bins also has to be copied to the parent. In addition, the top- k average involves calculating the top- k average from a set of bins in each node while the UB-average involves calculating the average of each node. The former is more complicated than the latter.

Pruning using UB-average should be more effective than the top- k average when k is small or when there is no count constraint, i.e., when $k = 1$. When k is small, the pruning using the top- k average is very coarse since only a small number of tuples are involved. On the other hand, the UB-average is not directly dependent on the k value. The UB-average involves the bundle of tuples that has the largest average for the group in the LG-tree. With the tuples compressed into branches of the LG-tree, the upper bound average of a group can still achieve severe pruning even when k is small. The more compact the LG-tree is, the more effective UB-average is for pruning.

5 Computing the top n averages

An effective pruning technique, an appropriate search order, and the principle of pushing the constraints early together constitute our novel solution for efficiently computing the top n averages in iceberg cubes. The search strategy and the algorithm that incorporates these techniques are presented in this section.

5.1 Searching the LG-tree for the top n averages

To search for the top n averages, we adopt depth-first based search with low overhead level-wise checking. The important difference from the search strategy for simple functions is the key determining the order in which groups are considered for searching the sub-groups. It is essential that such a key accurately indicates which is the most promising group that has sub-groups with larger averages. Different aggregate functions have different properties, therefore, we have designed a key suitable for the AVERAGE function.

Note that exploring a group which has more sub-groups with large averages finds the optimal average threshold faster than exploring a group which has one sub-group of very large average value and the rest of the sub-groups of small averages. This is because sub-groups of mostly large averages update L more frequently and therefore increase $L.\alpha$ quickly. On the other hand, sub-groups of one very large and many small averages have less effect on L and especially $L.\alpha$. The sub-group of a very large average is likely to update L once while the rest of the sub-groups may not even be pushed into L , therefore, not participating in increasing $L.\alpha$. Note that being able to increase $L.\alpha$ rapidly is crucial to the efficiency of our search strategy.

Unlike sum, the average of a group is not a good indicator of where sub-groups with large averages are since a group with a small average could still have sub-groups with large averages.

Recall that a group in an LG-tree is formed by those branches having the grouping dimension-values. We consider the average of the averages of these branches as a better indicator of whether the group has many sub-groups of large averages. This is because a subset of the branches forming the group forms a sub-group. In general, a sub-group, formed by the branches of large averages, is more likely to have a large average than if it is formed by branches of small averages. Therefore, if the averages of branches forming a group are mostly large, its sub-groups, which are made up by some of these branches, are also likely to be of large averages.

To consider all the averages of the branches forming a group, we calculate the average of these averages, and call it the *aoa* of the group. The *aoa* is the sort key for determining the order in which groups are considered for searching for sub-groups. By exploring the sub-groups of a group having a larger *aoa*, we are likely to find sub-groups with large averages early.

Suppose a group, g_1 , has mostly sub-groups with large averages and another group, g_2 , has one sub-group with a very large average and rest with small averages. While the average of g_1 and the average of g_2 could be similar, g_1 tends to have a larger *aoa* value than g_2 . Hence, the *aoa* of a group is a better key than is the average of a group to determine which group is likely to have sub-groups of large averages.

5.2 TopN-AVG-Miner

We present our algorithm *TopN-AVG-Miner* in Figure 6. Its main differences from TopN-Miner in our previous work are the pruning and search strategies for the AVERAGE aggregate function.

First (line 1), an LG-tree using only count threshold is constructed, consisting of a header table H and a tree structure. In addition to count and sum, the v_{upper} and the *aoa* of the 1- d groups are also kept in the header table entries for the purpose of pruning and determining the search order.

The 1- d groups derived from the entries in H so far pass the count threshold only. Therefore in line 2, the

Input: a) relational table T with dimensional attributes and measure e ,
b) top- n query with aggregate function AVERAGE and
count threshold τ , average threshold ν , where n is a parameter.

Output: L , the list of n large groups with the top n average-values

- 1) construct the LG-tree for T ; //let H be the header of the tree.
- 2) remove entries that have $v_{upper} < \nu$ from H
- 3) $L =$ the set of 0- and 1-dimensional groups derived from H that have average $\geq \nu$; //initialise L
- 4) sort H into aoa descending order;
- 5) **foreach** $a_i \in H$ **do**
- 6) construct H_{a_i} , the conditional header table based on a_i ;
- 7) call LG-AVG-Miner(H_{a_i}); // Assume L is global and $L.size = n$.
- 8) sort L into average-descending order and output;

Procedure LG-AVG-Miner(H_C)

// H_C is the Header conditioned on the set of dimension-values C ,

- 1) sort H_C into aoa descending order;
- 2) **foreach** $a_i \in H_C$ **do**
- 3) let g_{a_i} denote the group formed by a_i under C ;
- 4) **if** $g_{a_i}.AVG(e) > L.\alpha$ **then**
- 5) push g_{a_i} into L ; // $L.\alpha$ becomes larger
- 6) **foreach** a_i in H_C **do**
- 7) construct the **conditional** Header $H_{C_{a_i}}$ based on C and a_i ;
- 8) a) follow the side links of a_i to collect all a_i 's ancestor dimension-values
and their accumulated, conditional sum, count, aoa , and v_{upper} based on a_i ;
- 9) b) select only those dimension-values having
 $v_{upper} \geq L.\alpha$ and Count $\geq \tau$ to form $H_{C_{a_i}}$; // pruning
- 10) c) connect the ancestor dimension-values to the corresponding entries in $H_{C_{a_i}}$ with side links

- 8) LG-AVG-Miner($H_{C_{a_i}}$);

Figure 6: Algorithm TopN-AVG-Miner

v_{upper} values of the 1-dimensional groups have to be compared with the user-specified ν . The groups that have their v_{upper} less than ν can be pruned since they and their sub-groups must have averages less than ν .

When initialising L in line 3, only those 1- d groups derived from the remaining entries that have the average values $\geq \nu$ can be put into L . Note that for those groups that have the averages less than ν but the v_{upper} no less than ν , their entries in H have to be kept. Although these groups failed the average test, it is possible that some of their sub-groups have larger averages than ν ; their sub-groups still need to be searched.

Based on the heuristics discussed previously, by sorting the header entries into aoa descending order and then calling LG-AVG-Miner, we search the sub-groups of the group which is the most promising for having many sub-groups with large averages.

Procedure LG-AVG-Miner takes the conditional header table as a parameter and searches the sub-groups of a 1- d group to find the top n groups in an iceberg cube with AVERAGE. The construction of the conditional header table will be explained shortly. From the conditional header table, a set of groups with count $\geq \tau$ and $v_{upper} \geq \nu$ can be directly derived from the entries. In lines 4-5, among these groups, if their average is also greater than ν , they are added to L before the size of L reaches n , during which time $L.\alpha$ remains as ν . At the point when the size of L reaches n , the dynamic pruning strategies can commence. $L.\alpha$ from then on replaces ν as the AVERAGE threshold for pruning sub-groups and its value becomes larger each time a new group of a larger average is pushed into L .

Similar to the way 1- d groups are searched from the LG-tree, we search for the $(m+1)$ - d sub-groups of a m - d group a_i by constructing the conditional header

table based on a_i and deriving $(m+1)$ - d groups from the conditional header table (line 7). The dimension-values under the condition of a_i are the ancestor dimension-values of a_i in the LG-tree. They derive the sub-groups of a_i with $m+1$ dimensions. These dimension-values are collected by following the side-links of a_i , traversing up all the paths comprising a_i , and recording the dimension-values of the ancestor nodes of a_i (in line 7.a). While collecting the dimension-values, their count and sum values are accumulated and the v_{upper} and aoa calculated.

In line 7.a, while traversing the paths, the v_{upper} and aoa of an ancestor dimension-value of a_i can be easily obtained as the by-products of constructing the conditional header table. The v_{upper} of an ancestor dimension-value, d_{a_i} , of a_i is the largest average among the branches comprising a_i and d_{a_i} , while the aoa of d_{a_i} is the average of all the averages among those branches.

In line 7.b, pruning the sub-groups that do not pass the count and upper bound average constraints is achieved by including only those conditional dimension-values passing these two constraints. The groups derived from such conditional header table entries definitely pass the count threshold and potentially have average larger than the current average threshold. The averages of these sub-groups can be easily calculated from the sum and count in the header entries.

By recursively calling LG-AVG-Miner in line 8, all the sub-groups of the initial 1- d groups can be searched.

6 Discussions and Conclusions

In this study, we have the following two contributions to the computation of the top n averages in iceberg cubes. First, we derive a new weaker but anti-monotonic condition, the upper bound average (UB-average), from the non anti-monotonic function AVERAGE. Second, we identify a search strategy suitable for the computation of the top n averages in iceberg cubes. The UB-average pruning technique is simple to apply and requires little space. In addition, our search strategy finds groups with large averages as early as possible. The search strategy combined with the principle of pushing the constraints early into computation makes computing the top n averages in iceberg cubes efficient.

Compared to the top- k average, the UB-average is more space efficient. The UB-average only needs to keep a single v_{upper} for each entry in the header table. In contrast, the top- k average needs one set of bins for each entry in the header table and for every leaf node. Furthermore, the UB-average is easier to compute, while the top- k average requires a more complex processing of the set of bins.

While the value of k can affect the effectiveness of pruning using the top- k average, the effectiveness of pruning using the UB-average is related to the compactness of the LG-tree. In our previous study, it is observed that an LG-tree usually compresses data significantly. Improvement on the compactness of LG-tree can further improve the efficiency of computing iceberg cubes with AVERAGE function as well as the top n averages in iceberg cubes.

The idea of pushing the top- n constraint early into the search process is inspired by constraint-based data mining (Bayardo, Agrawal, & Gunopulos 1999, Lakshmanan, Ng, Han, & Pang 1999, Ng, Lakshmanan, Han, & Pang. 1998, Pei & Han 2000, Srikant, Vu, & Agrawal 1997). The principle of finding a weaker but anti-monotonic function to replace a non anti-monotonic function for pruning the search space is related to the work in data mining (Ng et al. 1998, Han et al. 2001).

While our solution is efficient from the analysis of both space and computation efforts, experiments have to be conducted to confirm the efficiency of our techniques for computing the top n averages in iceberg cubes. We are currently working on the experiments that compare our algorithm with the most current method (H-Cubing) for computing iceberg cubes with AVERAGE.

Acknowledgements

We thank Justin Zobel for his helpful comments.

References

- R. Agrawal & R. Srikant. Fast algorithms for mining association rules. In *Proceedings of VLDB'94*.
- V. Harinarayan, A. Rajaraman, & J. D Ullman. Implementing data cubes efficiently. In *Proceedings of SIGMOD'96*.
- S. Chaudhuri & U. Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD Record*, 26:65–74, (1997).
- R. Srikant, Q. Vu, & R. Agrawal. Mining association rules with item constraints. In *Proceedings of KDD'97*.
- J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, & M. Venkatrao. Data cube: a relational aggregation operator generalising group-by, cross-tab, and sub-totals. In *Data Mining and Knowledge Discovery*, 1(1):29–53, (1997).
- M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, & J. D. Ullman. Computing iceberg queries efficiently. In *Proceedings of VLDB'98*.
- R. Ng, L. V. S. Lakshmanan, J. Han, & A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proceedings of SIGMOD'98*.
- K. Beyer & R. Ramakrishnan Bottom-up computation of sparse and iceberg cubes. In *Proceedings of SIGMOD'99*.
- L. V. S. Lakshmanan, R. Ng, J. Han, & A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proceedings of SIGMOD'99*.
- R. J. Bayardo, R. Agrawal, & D. Gunopulos. Constraint-based rule mining on large, dense data sets. In *Proceedings of ICDE'99*.
- J. Han, J. Pei, & Y. Yan. Mining frequent patterns without candidate generation. In *Proceedings of SIGMOD'00*.
- J. Pei & J. Han. Can we push more constraints into frequent pattern mining? In *Proceedings of KDD'00*.
- J. Han, J. Pei, G. Dong, & K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proceedings of SIGMOD'01*.
- P. Chou & X. Zhang. Efficiently Computing the Top N Groups in Iceberg Cubes. Technical Report TR-02-3, RMIT, Melbourne, Australia, 2002.