Mining The Most General Multidimensional Summarization of “Probable Groups” in Data Warehouses*

Hui Yu[1] Jian Pei[2] Shiwei Tang[1] Dongqing Yang[3]

[1] National Laboratory on Machine Perception, Peking University, Beijing, China, yuhui@db.pku.edu.cn, tsw@pku.edu.cn

[2] Simon Fraser University, Burnaby, BC, Canada, jpei@cs.sfu.ca

[3] Peking University, Beijing, China, dqyang@pku.edu.cn

Abstract

Data summarization is an important data analysis task in data warehousing and online analytic processing. In this paper, we consider a novel type of summarization queries, probable group queries, *such as “What are the groups of patients that have a 50% or more opportunity to get lung cancer than the average?” An aggregate cell satisfying the requirement is called a* probable group. *To make the answer succinct and effective, we propose that only the most general probable groups should be mined. For example, if both groups (smoking, drinking) and (smoking, *) are probable, then the former groups should not be returned. The problem of mining the most general probable groups is challenging since the probable groups can be widely scattered in the cube lattice, and do not present any monotonicity in group containment order. We extend the state-of-the-art BUC algorithm to tackle the problem, and develop techniques and heuristics to speed up the search. An extensive performance study is reported to illustrate the effect of our approach.*

1 Introduction

Data summarization is an important data analysis task in data warehousing and online analytic processing. For example, an insurance company may want to summarize the common features of low-risk customers based on a data warehouse of customers’ claims. High accuracy and good understandability are the major requirements for high quality summarization.

Summarization from large databases, including multidimensional databases and data warehouses, has been studied extensively in previous work. For example, as a major research field in machine learning and data mining, accurate classification has been investigated intensively. A classifier for a target class can be viewed as the summarization. Accurate classifiers with good understandability, such as decision trees [16] and Bayesian networks [6], are often used as summarization of concepts in data analysis. As another example, the attribute-oriented induction [4] method is one of the pioneering database-oriented methods for concept summarization and generalization.

Although previous studies developed effective and efficient methods for summarization, most of them only work for *large* classes. That is, the previous methods implicitly or explicitly assume that the data (e.g., the training data set or the base table in a data warehouse) contains sufficient attributes and enough instances to support the summarization of the target class. However, this assumption may not be honored in some applications.

Example 1 (Motivation) In practice, more often than not a minor class may not be accurately characterized using the available attributes and cases. For example, although it is well known that smoking may lead to lung cancer, fortunately, more than 90% of smokers will not end up getting lung cancer. Generally, lung cancer happens in less than 0.1% of average population. In other words, lung cancer patients form a minor class. The available attributes in consensus data may not be sufficient to characterize the class of lung cancer patients. Therefore, it is often impossible to build an accurate classification model for lung cancer patients on consensus data sets. Instead, it would be more practical to identify the combinations of attributes, such as “smoking” and “family history of lung cancer”, that have a much higher probability to lead to lung cancer than the average cases. For example, the statistics identifying the high-risk groups that have 50% or more opportunities to get lung cancer than the average might be very interesting and helpful in health-informatics research.

As another example, in security-informatics, it is gener-

*This research is supported in part by National Natural Science Foundations of China Grant NSFC 60473072, NSERC Grant RGPIN312194-05, NSF (US) Grant IIS-0308001, a President’s Research Grant and an Endowed Research Fellowship Award in Simon Fraser University. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

ally very hard, if not impossible at all, to construct an accurate model for terrorists, since the class terrorists is a very minor class in population. Instead, it is more practical to identify the suspicious groups and then follow-up investigations can be conducted. ■

From the above example, we obtain the motivating observations as follows. There are real applications where the task is to summarize some minor classes that might not be accurately characterized using the available attributes and instances. In such cases, it is often useful to query the groups of instances that have a much higher probability to belong to the target minor classes. Such groups are called *probable groups*.

In this paper, we tackle the problem of *multidimensional summarization of probable groups in data warehouses*, and make the following contributions.

- *We identify a novel type of data summarization queries – probable group queries*. We illustrate that the probable group summarization queries are useful for summarization of minor classes. We also show that the problem of multidimensional summarization of probable groups in data warehouses is challenging since the probable groups may be widely scattered in the data cube lattice as the search space, and they do not present any monotonicity in group containment order.
- *We propose mining the most general multidimensional summarization*. We show that finding all probable groups can be ineffective and computational costly. Instead, we propose mining the most general probable groups as the succinct summarization.
- *We develop efficient algorithms*. We extend the sate-of-the-art cubing algorithm BUC [3] to compute all the most general probable groups. To make the mining more efficient, we further develop a heuristic dynamic-ordering method with smart techniques to prune unpromising recursive search. The new method is up to 3 times faster than the simple extension of BUC.
- *We report an extensive performance study*. The experimental results strongly suggest that our approach is efficient and scalable.

The rest of the paper is organized as follows. The problem is defined in Section 2. We develop algorithms for the problem in Section 3. An extensive performance study is presented in Section 4. We review related work in Section 5. The paper is concluded in Section 6.

2 Problem Description

In this section, we first introduce the preliminaries. Then, we present the probable group queries. Last, we examine how such queries should be answered effectively.

2.1 Preliminaries

Consider a *base table* $B = (D_1, \ldots, D_n, C)$, where D_1, ..., D_n are the n *dimensions* and C is the attribute of *class labels*. We assume that all dimensions are in categorical domains. For any tuple t, the value of t on attribute A is denoted by $t.A$.

An (*aggregate*) *cell* is a tuple $w = (w_1, \ldots, w_n)$, where $w_i \in D_i \cup \{*\}$ $(1 \leq i \leq n)$. When $w_i = *$, the dimension D_i is generalized in w. That is, $*$ matches any value in a dimension. The *cover* of an aggregate cell w, denoted by $cov(w)$, is the set of tuples in B that have the same values as w in all dimensions that $w_i \neq *$. That is,

$$cov(w) = \{t | (t \in B) \wedge (t.D_i = w.D_i \text{ for any } w.D_i \neq *)\}.$$

A cell w is called a *base cell* if for any dimension D_i, $w.D_i \neq *$. A base cell is a group-by of all dimensions in the base table.

For aggregate cells w_1 and w_2, w_1 is an *ancestor* of w_2 and w_2 is a *descendant* of w_1, denoted by $w_1 \succ w_2$, provided (1) for every dimension D_i such that $w_1.D_i \neq *$, $w_2.D_i = w_1.D_i$; and (2) there exists some dimension D_{i_0} such that $w_1.D_{i_0} = *$ and $w_2.D_{i_0} \neq *$. Particularly, if w_2 is a descendent of w_1 and agrees with w_1 on $(n-1)$ dimensions, then w_1 is called a *parent cell* of w_2, and w_2 is a *child cell* of w_1. It is easy to show the following.

Lemma 1 (Cover containment [12]) *For any cells w_1 and w_2 such that $w_1 \succ w_2$, $cov(w_1) \supseteq cov(w_2)$.* ■

However, the reverse direction of Lemma 1 is not true. That is, generally, we cannot derive $w_1 \succ w_2$ based on the fact $cov(w_1) \supseteq cov(w_2)$ [12].

2.2 Probable Group Queries

For a given *target class* $c \in C$ and an aggregate cell w, the *probability* of c in w, denoted by $prob(w, c)$, is the ratio of tuples in $cov(w)$ that belong to class c. That is,

$$prob(w, c) = \frac{|\{t | (t \in cov(w)) \wedge (t.C = c)\}|}{|cov(w)|}.$$

When the target class c is fixed and clear from context, we omit c and write $prob(w, c)$ as $prob(w)$.

An aggregate cell w is called a *probable group* or a *probable cell* provided that $prob(w, c) \geq min\_prob$, where c is the target class and min_prob is the *minimum probability threshold* specified by a user.

Problem definition 1 (Probable group queries) *Given a base table, a target class and a minimum probability threshold, a* probable group query *is to retrieve the complete set of probable groups.* ■

A	B	C	# tuples	# tuples in P	$prob$
a_1	b_1	c_1	7	1	14.29%
a_1	b_1	c_2	9	2	22.22%
a_1	b_2	c_1	4	1	25.00%
a_1	b_2	c_2	7	3	42.86%
a_1	b_3	c_1	10	2	20.00%
a_1	b_3	c_2	8	3	37.50%
a_2	b_1	c_1	5	0	0.00%
a_2	b_1	c_2	11	1	9.09%
a_2	b_2	c_1	7	2	28.57%
a_2	b_2	c_2	15	3	20.00%
a_2	b_3	c_1	4	0	0.00%
a_2	b_3	c_2	12	3	25.00%

Table 1. The base table as our running example.

As shown in Example 1, probable group queries are useful in summarization of minor classes, such as "*What are the groups of patients that have a* 50% *or more opportunity to get lung cancer than the average?*"

Now, the problem becomes searching all probable groups in a data warehouse. A nice property of data warehouse is that all aggregate cells in data warehouse can be organized in a lattice (called *cube lattice*) by the cell cover containment order [9, 12].

Example 2 (Cube lattice) Consider the base table B in Table 1 as our running example. The table has 3 dimensions, namely $A = \{a_1, a_2\}$, $B = \{b_1, b_2, b_3\}$, and $C = \{c_1, c_2, c_3\}$. The number of tuples in every group-by on dimensions A, B and C is also shown in the table (column "# tuples"). Let class P be the target minor class. The number of tuples of class P in every group-by is also shown in the column "# tuples in P". $prob(w, P)$ is also shown for every base cell w.

The set of all possible aggregate cells has $|A \cup \{*\}| \cdot |B \cup \{*\}| \cdot |C \cup \{*\}| = 3 \times 4 \times 3 = 36$ cells. The aggregate cells form a lattice as shown in Figure 1.

Suppose we are interested in the aggregate cells that have a ratio of 25% or up. Those aggregate cells are highlighted in Figure 1. There are in total 13 probable groups (cells). ■

Probable cells are scattered in the cube lattice, as demonstrated in Figure 1. If the probable cells have some monotonic properties in the cube lattice, the search can be facilitated substantially. Unfortunately, probable cells do not carry such a nice property.

Example 3 (Probable cells have no monotonic property) For aggregate cells $w_1 = (a_1, b_2, c_1)$, $w_2 = (a_1, *, c_1)$, and $w_3 = (a_1, *, *)$ in Figure 1, $w_1 \prec w_2 \prec w_3$. As shown in the figure, w_1 and w_3 are probable cells, but w_2 is not. Therefore, probable cells are not monotonic. That is, a probable cell w does not imply that the ancestors or the descendants of w must be probable cells. ■

2.3 Most General Probable Cells: Succinct Summarization

Although probable cells are not monotonic, as shown in Example 3, fortunately, they have a weak monotonic property as follows.

Lemma 2 (Weak monotonicity) *If w is a probable cell, then at least one child of w must also be a probable cell.*
Proof sketch. Let w' be a child cell of w such that $prob(w')$ is the maximum among all children cells of w. It can be shown that $prob(w) \leq prob(w')$. Since w is a probable cell, w' is also a probable cell. ■

Example 4 (Weak monotonicity) It is easy to verify that, in Figure 1, every probable cell has at least a child that is also a probable cells. In fact, for any probable cell c that is not a base cell, there is a path from some base probable cell to c such that each cell on the path is a probable cell. ■

The weak monotonicity gives us two important hints.

- *All probable cells stem from base probable cells.* In other words, although there can be many probable cells in a data warehouse, the base cells that have much higher ratio of the target class enable the more general aggregate probable cells. They are the "roots" of those probable cells.

- *The most general probable cells summarize the probable cells.* For any probable cell a, if it has some ancestor cell that is also a probable cell, it still can be generalized. A cell is *most general* if every ancestor cell of it is not a probable cell. The set of most general probable cells describe the most general extent of probable cells. Each probable cell is either most general, or is summarized by some most general probable cell.

Based on the above discussion, we can use the set of base probable cells and the set of most general probable cells to succinctly summarize a minor class.

Example 5 (Most general probable cells) In our running example, there are in total 13 probable cells. 5 of them are base probable cells. There are 3 most general probable cells, namely $(a_1, *, *)$, $(*, b_2, *)$ and $(*, b_3, c_2)$. In other words, only $\frac{3}{13} = 23.08\%$ of probable cells are most general, and another $\frac{5}{13} = 38.46\%$ of probable cells are base cells. If only the base probable cells and the most general probable

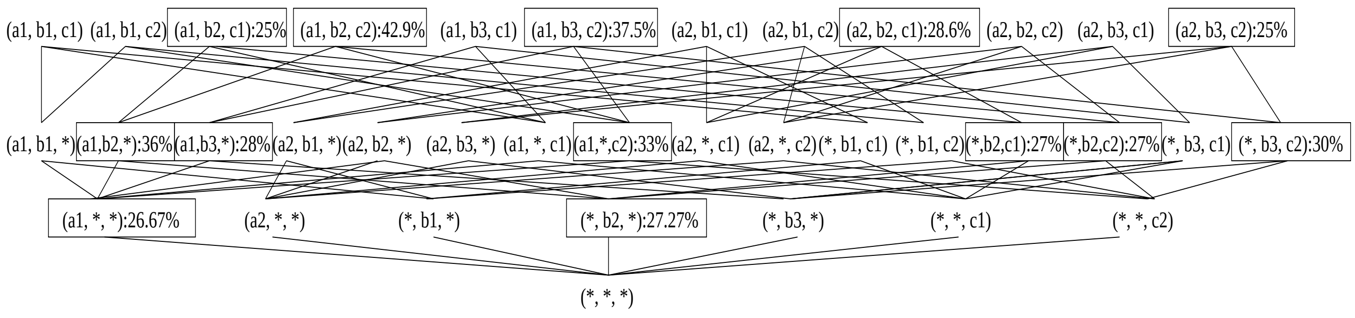

Figure 1. The cube lattice.

cells are used for the succinct summarization, we only need to record 8 probable cells, or $\frac{8}{13} = 61.54\%$ of all probable cells. There is a considerable saving.

As shown in our experimental results, using the base probable cells and the most general probable cells can achieve good saving in summarizing probable cells. ■

Clearly, the set of base probable cells can be computed as the group-by on all dimensions. Since the dimensions are categorical, we can use counting sort[1] to compute them efficient. As will be shown later, computing the set of base probable cells can be a byproduct of computing the set of most general probable cells.

Now, the problem becomes whether we can compute the set of most general probable cells efficiently. In the rest of the paper, we will focus on this issue.

Problem definition 2 (Succinct Summarization) *Given a base table, a target class, and a minimum probability threshold, the problem of* succinct summarization of the target class *is to compute the complete set of base probable cells and the complete set of most general probable cells.* ■

3 Algorithms

In this section, we first review BUC [3], a state-of-the-art algorithm for computing complete data cubes. Then, we discuss how BUC can be extended to mine the set of most general probable cells. We further develop a heuristic algorithm that can be much faster.

3.1 BUC: Bottom-up Cubing

In [3], Beyer and Ramakrishnan developed algorithm BUC, which computes the complete cube for a given base table, i.e., the complete set of aggregate cells. Extensive performance studies [3, 15] showed that BUC is efficient, scalable and moderate in main memory usage.

[1]According to Knuth, counting sort was invented by H.H. Seward in 1954. It is explained in many text books on algorithms, such as [5].

BUC conducts bottom-up computation and can use the monotonic iceberg conditions to prune. To compute a data cube on a base table $T(A, B, C, D)$, BUC first partitions the table according to dimension A, i.e., computing group-bys $(A, *, *, *)$. Then, BUC recursively searches the partition of $cov(a, *, *, *)$, where $a \in A$, and computes the descendant aggregate cells in depth-first search manner, such as $(a, b_1, *, *)$, $(a, b_2, *, *)$, and so on. The computation order is summarized in Figure 2. It also employs counting sort to make partitioning and group-by operations efficient.

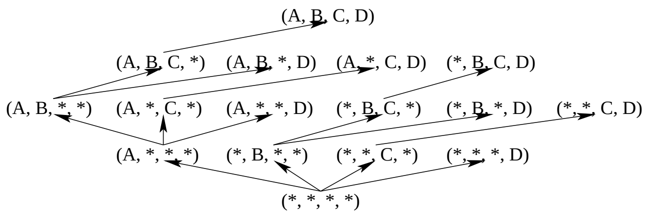

Figure 2. Bottom-up computation in BUC.

BUC can also efficiently incorporate monotonic conditions to compute iceberg cubes. A monotonic condition says that if an aggregate cell fails an iceberg condition, any descendants of it must also fail. If an aggregate cell $(a, *, *, *)$ fails the monotonic iceberg condition, any descendant of it, such as $(a, b, *, *)$, $(a, *, c, *)$ must also fail the condition and thus does not need to be computed in the depth-first search of BUC.

3.2 eBUC: Extending BUC to Mine Most General Probable Cells

Although BUC is efficient to compute the complete data cube, it cannot be directly used to compute the most general probable cells – it cannot use the weak monotonicity of probable cells to prune in a depth-first search. Here, we propose eBUC (for $\underline{e}$xtended $\underline{\text{BUC}}$), an extension of BUC to use the weak monotonicity in the mining.

The central idea of eBUC is the following observation.

Theorem 1 *An aggregate cell w is a probable cell only if w is a base probable cell or it is an ancestor of some base probable cell.*

Proof sketch. The theorem can be proved by induction on the number of $*$-dimensions in w. Lemma 2 can be applied repeatedly in the induction. ■

eBuc conducts depth-first search just like BUC. At the beginning of eBUC, by sorting all tuples in the base table using counting sort, eBUC computes the complete set of base cells as a byproduct. It stores those base cells that are probable.

During the rest of the depth-first search, when a new aggregate cell w is encountered, eBUC "looks ahead". That is, it checks whether w is an ancestor of some base probable cells. If not, then following Theorem 1, w cannot be a probable cell. Moreover, any descendant of w cannot be an ancestor of a base probable cell, either. Thus, the recursive search starting at w cannot find any probable cells and thus can be pruned.

When the search encounters a probable cell w, it does not need to search any descendants of w, since they cannot be most general. w is stored and checked after the search. If w is not a descendant of any other probable cells encountered by the search, then w is one of the most general probable cells.

Example 6 (Extended BUC) Let us run eBUC on the running example (Table 1). The search is shown in Figure 3. Only the cells connect by a directed edge are searched. The isolated cells are not searched.

eBUC starts from the most general cell $(*, *, *)$. It is not a probable cell, but it is an ancestor of some base probable cells. Thus, eBUC searches its children recursively in depth-first manner. The children are sorted in the dimensions order A-B-C, and within each dimension, the alphabetical order is used.

The first child, $(a_1, *, *)$, is probable. Thus, no descendants of $(a_1, *, *)$ are searched.

The second child, $(a_2, *, *)$, is not a probable cell, but it is an ancestor of base probable cells (a_2, b_2, c_1) and (a_2, b_3, c_2). Thus, eBUC recursively searches its children. The first child, $(a_2, b_1, *)$, is not a probable cell, and it is not an ancestor of any base probable cells. Thus, as suggested by Theorem 1, the search of $(a_2, b_1, *)$ as well as its descendants can be pruned. eBUC moves to the sibling of $(a_2, b_1, *)$ and search recursively.

The rest of the search is conducted similarly. Limited by space, we omit the details here.

After the search, eBUC checks all the probable cells encountered. For example, although probable cells (a_2, b_2, c_1) and (a_2, b_3, c_2) are encountered by eBUC, they are not the most general since they are descendants of probable cells $(*, b_2, *)$ and $(*, b_3, c_2)$, respectively, and thus will not be output.

As shown in Figure 3, eBUC can find the complete set of most general probable cells. ■

From Example 6, we can see that the most general probable cells and the weak monotonicity of probable cells can prune the search substantially. In this running example, only 17 of the 36 aggregate cells are searched. In other words, summarization takes only $\frac{17}{36} = 47.22\%$ of the cost of computing the complete cube.

3.3 DYNO: Heuristic Search by Dynamic Ordering

Algorithm eBUC shows good progress on mining the most general probable cells. It can be further improved based on the following two observations.

- In depth-first search, when an aggregate cell has multiple children to be searched, the search from the leftmost child covers the largest number of descendant cells. The search from a child cell always covers more descendant cells than that from its right sibling. If a child cell is probable, then all its descendants do not need to be searched. Thus, if we can order the cells dynamically such that the more promising a cell or its descendants are probable, the more left the cell is put, then sharper pruning is likely accomplished.

- As indicated by Theorem 1, only aggregate cells that are ancestors of some base probable cells should be considered. Thus, when expanding the search to children cells, only the dimension values that appear in some base probable cells that are descendants of the current cell should be used to expand the children of the current cell. All other children of the current cell are not promising.

Based on the above two observations, we develop algorithm DYNO (for DYNamic Ordering). DYNO follows the framework of eBUC and has the major improvements as follows.

In the depth-first search, if the current cell w is not a probable cell but is an ancestor of some base probable cell, then DYNO dynamically generates and orders the children cells.

DYNO does not expand all children cells of w. Instead, DYNO collects all base probable cells that are descendants of w. Only dimension values of those base probable cells are used to assemble children cells of w. The correctness of this improvement follows the second observation above. Moreover, to guarantee the completeness of the search and avoid searching a cell more than once in the depth-first

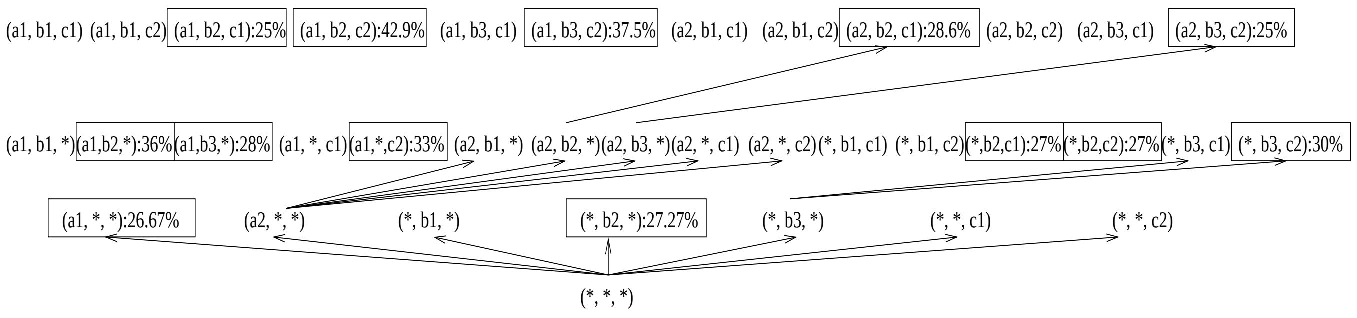

Figure 3. Search using eBUC.

search, DYNO joins w with the right siblings of w to generate its children cells, where the join is defined as follows.

For a pair of sibling cells w_1 and w_2, two cases may arise.

- w_1 and w_2 agree on all dimensions except for one dimension D. That is, $w_1.D \neq w_2.D$ and both do not take $*$ on dimension D. In this case, the join is not defined. In other words, w_1 and w_2 cannot be joined.
- w_1 and w_2 agree on all dimensions except for two dimensions D and D'. That is, $w_1.D = *$, $w_2.D \neq *$, $w_1.D' \neq *$ and $w_2.D' = *$. In this case, the join is defined as w_3 such that w_3 take values as its parent except for dimensions D and D', $w_3.D = w_2.D$ and $w_3.D' = w_1.D'$.

The current cell w may have multiple children. Then, according to the first observation discussed above, we should search them in the order of likelihood that they are probable cells. Heuristically, we can search them in their probability descending order – the higher the probability, the better chance that it or some of its descendants are a probable cell.

We need to show that the above dynamic generation and ordering of children retains the completeness and non-redundancy of depth-first search.

Theorem 2 (Dynamic generation and ordering) *A depth-first search with the dynamic generation and ordering of children cells visits each aggregate cell once and only once if no pruning is taken.*

Proof sketch. The theorem can be proved by induction on the number of non-$*$ dimensions in aggregate cells. For each cell w, it can be shown that w will be generated once and only once. Limited by space, we only show the essential idea here. ■

Example 7 (DYNO) Let us apply algorithm DYNO on our running example. The search is illustrated in Figure 4.

DYNO starts from the most general cell $(*, *, *)$. Since it is not a probable cell, but it is an ancestor of some base probable cells, we need to search its children.

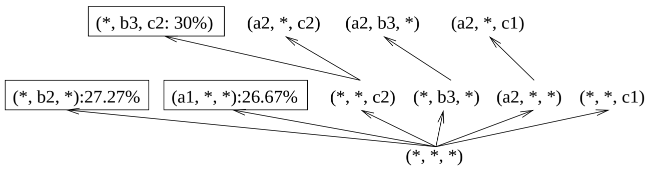

Figure 4. Search using DYNO.

When generating the children cells of $(*, *, *)$, DYNO notices that b_1 never appears in any base probable cell. Thus, any aggregate cells having b_1 cannot be a probable cell. Although $(*, b_1, *)$ is a child of $(*, *, *)$, it is unpromising and thus should not be generated. The children cells of $(*, *, *)$ generated by DYNO are $(*, b_2, *)$, $(a_1, *, *)$, $(*, *, c_2)$, $(*, b_3, *)$, $(a_2, *, *)$, and $(*, *, c_1)$, in the probability descending order.

$(a_1, *, *)$ and $(*, b_2, *)$ are probable cells. They are stored for postprocessing. The descendants of the two cells will not be searched.

DYNO recursively searches cell $(*, *, c_2)$. By joining the right siblings, two children cells are generated, namely $(*, b_3, c_2)$ and $(a_2, *, c_2)$. The mining can be conducted recursively. Limited by space, we omit the details here.

After the search, for each probable cell w encountered in the search, DYNO checks whether w is a descendant of some other encountered probable cells. If not, then cell w is output as a most general probable cell.

It can be verified that DYNO can find the three most general probable cells. ■

From the above example, we can see that DYNO can find the complete set of most general probable cells. Moreover, DYNO searches much fewer cells than eBUC. In this example, DYNO searches 11 cells, while eBUC searches 17 cells. DYNO searches $\frac{6}{17} = 35.29\%$ less cells than eBUC. Our experimental results show that DYNO can search over 50% less cells than eBUC. This is a major saving in the mining. To summarize, algorithm DYNO is shown in Figure 5.

```
Algorithm DYNO
Input: a base table $B$, a target class $c$, and a minimum
    probability threshold $\delta$;
Output: the set of base probable cells and the set of most
    general probable cells;
Method:
1.  sort tuples in $B$, compute and output the base probable
    cells, also compute $prob(*, \dots, *)$;
2.  let $W = \emptyset$;
3.  conduct depth-first search from cell $(*, \dots, *)$, for
    each current cell $w$, do
4.      if $w$ is not an ancestor of any base probable cell
        then return;
5.      if $prob(w) \geq \delta$ then $W = W \cup \{w\}$, return;
6.      generate children of $w$ by joining $w$ with the right
        siblings of $w$, using only the dimension values that
        appear in descendant base probable cells of $w$
7.      compute probability for children cells;
8.      sort the children of $w$ in the probability descending
        order;
9.      search the children recursively in depth-first
        manner;

    // Postprocessing
10. remove cells $w$ from $W$ such that $w$ has an ancestor $w'$
    in $W$;
11. output $W$;
```

Figure 5. Algorithm DYNO.

4 Experimental Results

We conducted extensive experiments using synthetic data sets. The results are consistent. Limited by space, we only reported some results in this section.

All the algorithms are implemented using Microsoft Visual C++ V6.0. The experiments are conducted on a PC with a P4 1.5G Hz CPU and 512 MB main memory. The operating system is Microsoft Windows XP.

By default, a base table has 10 dimensions. The cardinality of each dimension is 100. There are 100 thousand tuples in the base table. Each tuple is a base cell with a population and a probability of the target class. The probability of the target class in base cells follows the Half-Normal Distribution in $[0, 1]$, i.e., a normal distribution with mean 0 and standard deviation $\frac{1}{\theta}$ limited to the domain $[0, 1]$. In the results reported in this section, we set $\theta = 1$.

First of all, it is interesting to examine the change of the number of probable cells and the number of most general probable cells with respect to the probable threshold, which is shown in Figure 6. As the minimum probability threshold goes down, the number of probable cells keeps growing. However, the number of most general probable cells does not monotonically change. When the minimum probability threshold is high, there are only a small number of probable cells, and the number of most general probable cells is also small. As the minimum probability threshold goes down, both the number of probable cells and the number of most general probable cells increase. When the minimum probability threshold is lower than 50% in our experiments, there are many probable cells. They can be summarized by some quite general probable cells. The strong capability of high level aggregate cells to summarize the low level cells brings down the number of most general aggregate cells. In the extreme case, when the most general cell in the cube, $(*, \dots, *)$, is probable, there is only one most general probable cell.

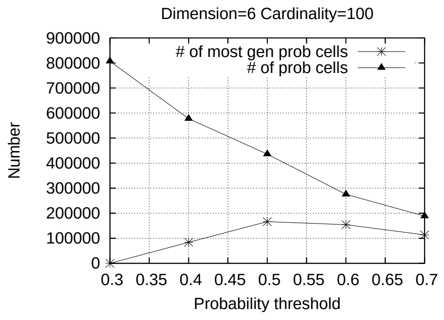

Figure 6. Number of probable cells with respect to minimum probability threshold.

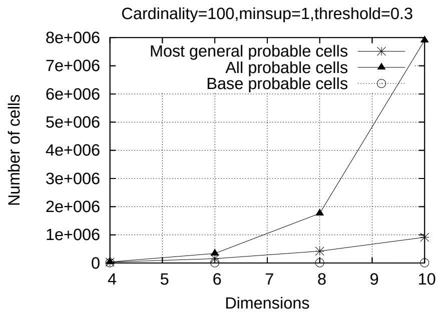

Figure 7. The number of probable cells with respect to dimensionality.

The number of probable cells and the number of most general probable cells also increase as the dimensionality increases, as shown in Figure 7. However, the number of most general probable cells has a much more moderate increase rate.

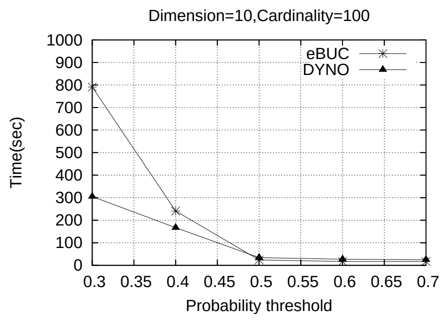

Figure 8. The scalability with respect to minimum probability threshold.

In Figure 8, we tested the scalability of eBUC and DYNO with respect to the probability threshold. When the probability threshold is set high, the number of probable cells and the number of most general probable cells are small. Thus, both algorithms are fast and the difference between the two algorithms is minor. However, when the probability threshold is low, there can be many probable cells. DYNO has a much better scalability than eBUC.

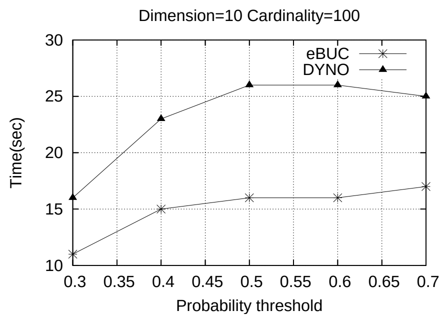

Figure 9. The depth-first search runtime with respect to minimum probability threshold.

The runtime of both DYNO and eBUC can be divided into two parts: the time for depth-first search and the time for postprocessing. An interesting observation is that the depth-first searches in DYNO and eBUC only take a small part in the total runtime. The runtime for depth-first search is shown in Figure 9. As can be seen, when the minimum probability threshold is low and the number of most general probable cells decreases, DYNO and eBUC become more efficient in the depth-first search. In other words, the curves of depth-first search runtime of DYNO and eBUC in Figures 9 are consistent with the curve of number of most general probable cells in Figure 6. In terms of search time per cell, eBUC is shorter than DYNO since DYNO needs to collect more information than eBUC. However, the major advantage of DYNO is that it generates much less candidate cells than eBUC, which makes the postprocessing of DYNO clearly faster.

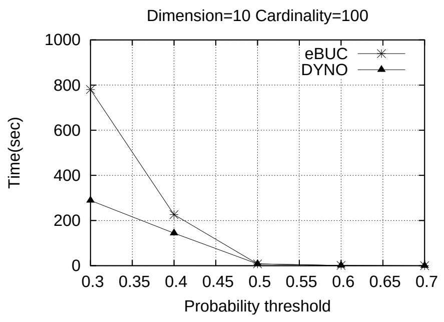

Figure 10. The postprocessing runtime with respect to minimum probability threshold.

Figure 10 shows the postprocessing runtime. Both DYNO and eBUC use the same method in postprocessing to remove the non-most general probable cells. Since the heuristic search in DYNO (dynamic generation and ordering of children cells) can effectively reduce the number of probable cells searched, the postprocessing cost in DYNO is substantially smaller than that in eBUC.

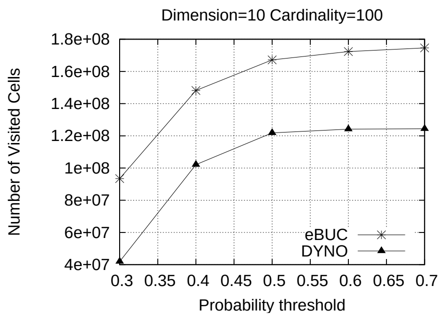

Figure 11. The number of aggregate cells searched with respect to minimum probability threshold.

Figure 11 supports the claim that DYNO visits substantially less aggregate cells in finding the most general probable cells. DYNO also encounters much less probable cells in the depth-first search than eBUC. The numbers of probable cells encountered by DYNO and eBUC, respectively, follow the trends similar to the results in Figure 10. Limited by space, we omit the details here. It shows that the pruning techniques in DYNO are effective.

To test the scalability of our methods, we ranged the dimensionality from 4 to 10. The results are shown in Figure 12. DYNO has a better scalability. Moreover, the results are consistent with the number of most general scalable cells shown in Figure 7.

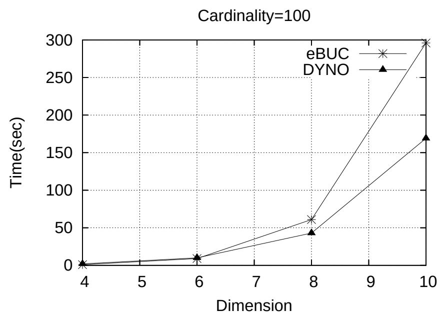

Figure 12. The runtime with respect to dimensionality.

We also tested the runtime of DYNO and eBUC on the number of tuples in the base table. Both are linearly scalable, and DYNO has a better scalability. Limited by space, we omit the details here.

In summary, the extensive experimental results strongly suggest that using the most general aggregate cells can effectively summarize the probable cells. DYNO is an efficient method to compute the most general probable cells.

5 Related Work

The data cube operator [9] is one of the most influential operators in OLAP. Many approaches have been proposed to compute data cubes efficiently from scratch (e.g., [24, 17, 18, 3]). In general, they speed up the cube computation by sharing partitions, sorts, or partial sorts for group-bys with common dimensions.

It is well recognized that the space requirements of data cubes in practice are often huge. Some studies investigate partial materialization of data cubes, e.g., [11, 3]. Example methods to compress data cubes are [19, 20, 12, 13]. Moreover, [1, 2, 21] investigate various approximation methods for data cubes.

There are several major methods on computing (iceberg) cubes. MultiWay [24] is an array-based top-down approach to computing complete data cube. The basic idea is that a high level aggregate cell can be computed from its descendants instead of the base table. To compute a data cube on a base table $T(A, B, C, D)$, MultiWay first scans the base table once and computes group-bys (A, B, C, D), $(*, B, C, D)$, $(*, *, C, D)$, $(*, *, *, D)$ and $(*, *, *, *)$. These group-bys can be computed simultaneously without resorting the tuples in the base table. Once these group-bys are computed, we do not need to scan the base table any more. MultiWay may not be efficient in computing iceberg cubes with monotonic iceberg conditions, since the top-down search cannot use the monotonic iceberg condition to prune.

Fang et al. [7] proposed the concept of iceberg queries and developed some sampling algorithms to answer such queries. An iceber cube is the set of aggregate cells in a cube that satisfy some user-specified condition. Beyer and Ramakrishnan [3] introduced the problem of iceberg cube computation in the spirit of [7] and developed algorithm BUC, which is revisited in Section 3.1. Often, monotonic iceberg conditions are used to prune in the computation of iceberg cubes.

H-cubing [10] uses a hyper-tree data structure called H-tree to compress the base table. Then, the H-tree can be traversed bottom-up to compute iceberg cubes. It also can prune unpromising branches of search using monotonic iceberg conditions. Moreover, a strategy was developed in [10] to use weakened but monotonic conditions to approximate non-monotonic conditions to compute iceberg cubes. The strategies of pushing non-monotonic conditions into bottom-up iceberg cube computation were further improved by Wang et al. [22]. A new strategy, divide-and-approximate, was developed. The general idea is that the weakened but monotonic condition can be made up for each search sub-branch and thus the approximation and pruning power ca be stronger.

In [23], Xin et al. developed Star-Cubing by extending H-tree to Star-Tree and integrating the top-down and bottom-up search strategies. Feng et al. [8] proposed another interesting cubing algorithm, Range Cube, which uses a data structure called range trie to compress data and identify correlation in attribute values. On the other hand, since iceberg cube computation is often expensive in both time and space, parallel and distributed iceberg cube computation has been investigated. For example, Ng et al. [15] studied how to compute iceberg cubes efficiently using PC clusters.

In all the previous studies, either the complete cube or the complete iceberg cube is computed. None of them consider the problem of computing a summarization of the cells that satisfy some user-specified condition. None of them either deal with mining the most general aggregate cells. To the best of our knowledge, this paper is the first one that addresses the issue.

On the other hand, this paper is also related to previous work on concept summarization [4], generalization and learning [14]. However, different from those approaches, we use the most general aggregate cells to summarize probable groups, which have not been discussed in those previous studies.

6 Conclusions

Data summarization is an important data analysis task in data warehousing and online analytic processing. In this paper, we identified a new type of summarization queries, *probable group queries*, and proposed a succinct summarization answer to the queries using the base probable cells and the most general probable cells. The problem of mining the most general probable cells is challenging since the probable cells can be widely scattered in the cube lattice, and do not present any monotonicity in cover containment order. We extended the state-of-the-art BUC algorithm to tackle the problem, and developed techniques and heuristics to speed up the search. An extensive performance study verified that our approach is effective and efficient.

This study raises several interesting problems for future studies. For example, it is interesting to improve the performance of DYNO further, especially reducing the cost of ancestor-descendant checking in the postprocessing. Moreover, summarization and understanding of minor classes are important for data analysis and applications. Theoretical framework as well as practical mining methods should be explored further.

References

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

[2] D. Barbara and X. Wu. Using loglinear models to compress datacube. In *WAIM'2000*, pages 311–322, 2000.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.

[4] Y. Cai, N. Cercone, and J. Han. An attribute-oriented approach for learning classification rules from relational databases. In *Proc. 1990 IEEE Int. Conf. Data Engineering (ICDE'90)*, pages 281–288, Los Angeles, CA, Feb. 1990.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[6] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, 1973.

[7] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 299–310, New York, NY, Aug. 1998.

[8] Y. Feng, D. Agrawal, A. E. Abbadi, and A. Metwally. Range Cube: Efficient cube computation by exploiting data correlation. In *Proc. 2004 Int. Conf. Data Engineering (ICDE'04)*, pages 658–669, Boston, MA, April 2004.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In *Proc. 1996 Int. Conf. Data Engineering (ICDE'96)*, pages 152–159, New Orleans, Louisiana, Feb. 1996.

[10] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01)*, pages 1–12, Santa Barbara, CA, May 2001.

[11] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 205–216, Montreal, Canada, June 1996.

[12] L. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, Hong Kong, China, Aug. 2002.

[13] L.V.S. Lashmanan, J. Pei, and Y. Zhao. QC-Trees: An efficient summary structure for semantic OLAP. In *Proc. 2003 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'03)*, San Diego, California, June 2003.

[14] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[15] Raymond T. Ng, Alan S. Wagner, and Yu Yin. Iceberg-cube computation with PC clusters. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01)*, Santa Barbara, CA, May 2001.

[16] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[17] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 1997 Int. Conf. Very Large Data Bases (VLDB'97)*, pages 116–125, Athens, Greece, Aug. 1997.

[18] Kenneth A. Ross and Kazi A. Zaman. Optimizing selections over datacubes. In *Statistical and Scientific Database Management*, pages 139–152, 2000.

[19] Jayavel Shanmugasundaram, Usama Fayyad, and P. S. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 223–232, San Diego, California, United States, 1999. ACM Press.

[20] Yannis Sismanis, Nick Roussopoulos, Antonios Deligiannakis, and Yannis Kotidis. Dwarf: Shrinking the petacube. In *Proc. 2002 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'02)*, Madison, Wisconsin, June 2002.

[21] J. S. Vitter, M. Wang, and B. R. Iyer. Data cube approximation and historgrams via wavelets. In *Proc. 1998 Int. Conf. Information and Knowledge Management (CIKM'98)*, pages 96–104, Washington DC, Nov. 1998.

[22] K. Wang, Y. Jiang, J. X. Yu, G. Dong, and J. Han. Pushing aggregate constraints by divide-and-approximate. In *Proc. 2003 Int. Conf. Data Engineering (ICDE'03)*, pages 291–302, Bangalore, India, March 2003.

[23] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proc. 2003 Int. Conf. on Very Large Data Bases (VLDB'02)*, pages 476–487, Berlin, Germany, Sept. 2003.

[24] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 159–170, Tucson, Arizona, May 1997.