Submission to IEEE TKDE April 10, 2003

Mining Constrained Gradients in Large Databases [*]

Guozhu Dong[*], Jiawei Han[†], Joyce Lam[‡], Jian Pei[§], Ke Wang[‡], Wei Zou[‖]

[*] Correspondence author:
Department of Computer Science and Engineering
Wright State University
Dayton, OH 45435, U.S.A.
gdong@cs.wright.edu

[†] Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.
hanj@cs.uiuc.edu

[‡] School of Computing Science
Simon Fraser University
B.C., Canada V5A 1S6
{lamd, wangk}@cs.sfu.ca

[§] Department of Computer Science and Engineering
Stata University of New York at Buffalo
Buffalo, NY 14260, U.S.A.
jianpei@cse.buffalo.edu

[‖] Department of Computer Science
Jiangxi Normal University
Nanchang, Jiangxi, 330027, China
zouwei@jxnu.edu.cn

[*]Work supported in part by the Natural Sciences and Engineering Research Council of Canada (grant NSERC-A3723) and the Networks of Centres of Excellence of Canada (grant NCE/IRIS-3). A preliminary version of the paper appeared in Proc. 2001 VLDB Conference under the title "*Mining Multi-Dimensional Constrained Gradients in Data Cubes*".

Abstract

Many data analysis tasks can be viewed as search or mining in a multidimensional space (MDS). In such MDSs, dimensions capture potentially important factors for given applications, and cells represent combinations of values for the factors. To systematically analyze data in MDS, an interesting notion, called "cubegrade" was recently introduced by Imielinski, et al. [IKA02], which focuses on the notable changes in measures in MDS by comparing a cell (which we refer as *probe cell*) with its *gradient cells*, namely its ancestors, descendants and siblings. We call such queries *gradient analysis queries* (GQs). Since an MDS can contain billions of cells, it is important to answer GQs efficiently. In this study, we focus on developing efficient methods for mining GQs constrained by certain (weakly) anti-monotone constraints.

Instead of conducting an independent gradient-cell search once per probe cell, which is inefficient due to much repeated work, we propose an efficient algorithm, *LiveSet-Driven*. This algorithm finds all good gradient-probe cell pairs in one search pass. It utilizes measure-value analysis and dimension-match analysis in a set-oriented manner, to achieve bi-directional pruning between the sets of hopeful probe cells and of hopeful gradient cells. Moreover, it adopts a hyper-tree structure and an H-cubing method to compress data and to maximize sharing of computation. Our performance study shows that this algorithm is efficient and scalable.

In addition to data cubes, we extend our study to another important scenario: mining constrained gradients in *transactional databases* where each item is associated with some measures such as price. Such transactional databases can be viewed as sparse MDSs where items represent dimensions, although they have significantly different characteristics than data cubes. We outline efficient mining methods for this problem in this paper.

Index terms: Data cube, data mining, gradient analysis, iceberg query, anti-monotonicity, dimension-based pruning, constraint-based pruning, complex measures.

1 Introduction

Recently, there have been growing interests in multidimensional analysis of relational databases, transactional databases, and data warehouses. Most of such analysis involve data cube-based summary or transaction-based association analysis. However, in many interesting applications one may want to analyze the *changes* of *measures* in multidimensional space. For example, one may want to ask what are associated with significant *changes* of the *average* house price in the Vancouver area in year 2000 compared against 1999, and the answer could include statements of the form "the average price for those sold to professionals in the West End went down by 20%, while those sold to business people in Metrotown went up by 10%, etc." Expressions such as "professionals in the West End" correspond to cells in data cubes, and describe sectors of the business modeled by the data cube.

The problem of mining *changes* of *sophisticated measures* in a multidimensional space was first proposed by Imielinski, et al. [IKA02] as a *cubegrade* problem, which can be viewed as a generalization of association rules and data cubes. It studies how changes in a set of measures (aggregates) of interest are associated with changes in the underlying characteristics of sectors, where changes in sector characteristics are expressed in terms of dimensions of the cube and are limited to specialization (drill-down), generalization (roll-up), and mutation (a change in one of the cube's dimensions). For example, one may want to ask "what kind of sector characteristics are associated with major changes in average house prices in the Vancouver area in 2000," and the answer will be pairs of sectors, associated with major changes in average house prices, including for example "the sector of professional buyers in the West End area of Vancouver" vs. "the sector of all buyers in the entire area of Vancouver" as a specialization.

The cubegrade query is significantly more expressive than association rules since it captures the trends in data and handles arbitrary measures, not just COUNT, as association rules do. The problem is interesting and has broad applications, such as trend analysis, answering "what-if" questions, discovering exceptions or outliers, etc. However, it also poses serious challenges on both understandability of results and on computational efficiency and scalability, as illustrated below.

1. A data cube may have many dimensions. Even though each dimension may involve only a small number of values, the total number of cells of the data cube may still be quite huge. For example, a data cube with 20 dimensions each containing 99 distinct values has $(99 + 1)^{20} = 10^{40}$ base and high level cells. Even if there is only one nonempty cell in every 10^{10} cells, the cube will

still contain 10^{30} cells—too huge to be precomputed and stored with reasonable resources. In a transactional database, if we consider each item (such as milk or bread) as one independent dimension, as in [IKA02], we may need to handle thousands of dimensions, and the curse of dimensionality will be even worse than that of classical data cubes which usually contain only dozens of dimensions. An effective compromise to this problem is to compute iceberg cubes instead of the complete cubes [BR99]. To this end, we need to introduce a *significance constraint* for pruning the huge number of trivial cells in the answer set.

2. The cubegrade problem needs to compare each cell in the cube with its associated cells generated by specialization, generalization, and mutation. Even when considering only iceberg cubes, it may still generate a very large number of pairs. Since for each analysis task, a user is often interested in examining only a small subset of cells in the cube, it is desirable to enforce certain *probe constraints* to select a subset of cells (called *probe cells*) from all the possible cells as focus points for examination. Using such constraint, one is focused only on *these* cells and their relationships with corresponding siblings, ancestors, and descendants.

3. A user is usually interested in only certain types of changes between the cells (sectors) under comparison. For example, one may be interested in only those cells whose average profit increases by more than 40% compared to that of the probe cells, etc. Such changes can be specified as a threshold in the form of ratio/difference between certain measure values of the cells under comparison. We will call the cell that captures the change from the probe cell *the gradient cell*, and call such constraints the *gradient (interestingness) constraints*.

From this discussion, one can see that to mine interesting gradients in a multidimensional space, it is often necessary to have the following three kinds of constraints: (1) *significance constraint*, which ensures that we produce only cells which have certain statistical significance in the data, such as those containing at least certain number of base cells or at least certain total sales; (2) *probe constraint*, which confines the set of probe cells that our gradient analysis will focus on; and (3) *gradient constraint*, which specifies the user's range of interest on the gradient (i.e., measure change). In this paper, we consider significance constraints that can be specified using thresholds on measures and that are anti-monotone or weakly anti-monotone (see Section 2), and we restrict probe constraints to non-nested SQL queries. Enforcing these constraints may lead to interesting, clearly understandable answers as well as possibility to derive efficient methods for gradient analysis in a multidimensional space. In this context, the problem of multidimensional gradient analysis with such constraints represents a confined but interesting version of the cubegrade problem, which we call the *constrained* (*multidimensional*) *gradient analysis*.

In this paper, we study efficient and scalable methods for constrained gradient analysis in multidimensional space. Our study is focused on mining constrained gradients in data cubes. However, we will also examine how to extend the method to mining constrained gradients in transactional databases.

For mining constrained gradients in data cubes, we first consider a naive approach which computes such gradients by conducting a search, for the gradient cells, once per probe cell[1]. This approach is inefficient because there is a large amount of repeated work for different probe cells. To avoid this problem, we propose an efficient algorithm, called the *LiveSet-Driven* algorithm, which utilizes constraints early on and during computation, for computing pairs of cells. The algorithm first computes the set of significant probe cells, and then processes the potential gradient cells from low dimensional cells to high dimensional ones. The computation for a set of probe cells is bundled together, and the set of live probe cells is used for pruning. We introduce a method to determine the optimal set of probe cells that needs to be used for pruning a potential gradient cell and its descendants, which takes into consideration the dimensional relationship between cells and the gradient constraint with respect to a set of probe cells. Moreover, a compressed hyper-tree structure is used to represent the base table of a data cube, and an H-cubing method is used to achieve "maximal" sharing of computation among different cells. Even though the naive algorithm also prunes as much as possible, the *LiveSet-Driven* algorithm is much better because it uses (i) set-oriented processing, (ii) set-oriented pruning, and (iii) a one-pass search for all probe-gradient pairs. Our performance study shows that the *LiveSet-Driven* algorithm makes good use of constraints and is efficient and scalable for large data sets.

Finally, we extend our scope of study to efficient mining of constrained gradients in transactional databases and outlined a probe-based *FP-growth* method for constrained gradient mining.

The rest of the paper is organized as follows. Section 2 defines the constrained gradient analysis problem and presents an example. Section 3 presents the methods for mining constrained gradients in data cubes, including the LiveSet-Driven algorithm, which covers the techniques for pruning probe cells and gradient cells. Section 4 reports the results of our experiments and performance study. Section

[1]Since the gradient constraint is on *pairs of cells* whereas the significance and probe constraints are on *individual cells*, we believe that the significance and probe constraints combined are usually more restrictive than the gradient constraint. So we only consider approaches that first use the significance and probe constraints to restrict the search space.

5 discusses variations of the method, extends our scope of study to mining constrained gradients in transaction databases, and compares with the related work. Finally, we conclude our study in Section 6.

2 Problem Definition and Assumptions

Let $\mathcal{D}$ be a relational table, called the *base table*, of a given cube. The set of all *attributes* $\mathcal{A}$ in $\mathcal{D}$ are partitioned into two subsets, the *dimensional attributes* DIM and the *measure attributes* M (so $DIM \cup M = \mathcal{A}$ and $DIM \cap M = \emptyset$). The measure attributes functionally depend on the dimensional attributes in $\mathcal{D}$ and are defined in the context of data cube using any of these five SQL aggregate functions: COUNT, SUM, AVG, MAX, and MIN.

A tuple with schema $\mathcal{A}$ in a multi-dimensional space (i.e., in the context of data cube) is called a **cell**. Given three distinct cells c_1, c_2 and c_3, c_1 is an **ancestor** of c_2, and c_2 a **descendant** of c_1 iff on every dimensional attribute, either c_1 and c_2 share the same value, or c_1 has value "$*$" (where "$*$" indicates "all", i.e., aggregated to the highest level on this dimension); c_2 is a **sibling** of c_3, and vice versa, iff c_2 and c_3 have identical values in all dimensions except one dimension in which neither has value "$*$". A cell which has k non-* values is called a **k-d cell**.

A tuple $c \in \mathcal{D}$ is called a **base cell**. A base cell does not have any descendant. A cell c is an **aggregated cell** iff it is an ancestor of some base cell. For each aggregated cell c, its values on the measure attributes are derived from the complete set of descendant base cells of c.

As mentioned in Section 1, the specification of a constrained gradient analysis problem requires three constraints: a *significance constraint* C_{sig}, a *probe constraint* C_{prb}, and a *gradient constraint* C_{grad}. Both C_{sig} and C_{prb} are unary (defined over cells). A cell c is a **significant** cell iff $C_{sig}(c) = true$, and a cell c is a **probe cell** iff c is significant and $C_{prb}(c) = true$. The complete set of probe cells is denoted as $\mathcal{P}$.

Significance constraints are usually defined as threshold conditions on measure attributes. These constraints do not have to be anti-monotonic[2], and can be for example, on a measure defined by the AVG aggregate function. In [HPDW01], methods for deriving *weaker anti-monotonic* constraints[3] from

[2]Anti-monotonicity is very useful for pruning. It states that *if a cell c does not satisfy an (anti-monotonic) significance constraint C_{sig}, none of c's descendants can do so.* For example, the constraint "*count* > 10" is anti-monotone. Anti-monotonicity-based pruning forms the foundation for most algorithms for computing iceberg cubes.

[3]We will call those constraints from which one can derive weaker anti-monotone constraints the *weakly anti-monotone*

non-anti-monotonic constraints and for efficiently computing iceberg cubes were discussed. We will use such weaker anti-monotonic constraints for pruning candidate cells.

We assume that a probe constraint is a one-level SQL query (without nested query), which will "select" a set of user-desired cells. The query can involve the dimensional attributes as well as the measure attributes.

A **gradient constraint** is binary (defined over pairs of cells). It has the form $C_{grad}(c_g, c_p) \equiv (g(c_g, c_p)\ \theta\ v)$, where θ is in $\{<, >, \geq, \leq\}$, v is a constant value, and g is a *gradient function*. $g(c_g, c_p)$ is defined iff c_g is either an ancestor, a descendant, or a sibling of c_p. A gradient cell c_g is **interesting** with respect to a probe cell $c_p \in \mathcal{P}$ iff c_g is significant and $C_{grad}(c_g, c_p) = true$.

In this paper we mainly consider gradient constraints defined using the ratio of two measure values such as "$m(c_g)/m(c_p)\ \theta\ v$", where $m(c)$ is a measure value for a cell c. Most of the results derived for ratio can be easily extended to difference, "$m(c_g) - m(c_p)\ \theta\ v$" (see Section 5).

Problem definition. Given a base table $\mathcal{D}$, a significance constraint C_{sig}, a probe constraint C_{prb}, and a gradient constraint $C_{grad}(c_g, c_p)$, the **constrained gradient analysis** problem is: Find the complete set of all interesting gradient-probe pairs (c_g, c_p) such that $C_{grad}(c_g, c_p) = true$. □

Example 1 (Constrained average gradient) Let the *base table* $\mathcal{D}$ be a sales table with the schema

$$sales(year, city, cust\_grp, prod\_grp, cnt, avg\_price)$$

Attributes *year*, *city*, *cust_grp*, and *prod_grp* are the *dimensional attributes*; and *cnt* and *avg_price* are the *measure attributes*.

c_1	$(00, Vancouver, Business, PC, 300, \$200)$
c_2	$(*, Vancouver, Business, PC, 800, \$1900)$
c_3	$(*, Toronto, Business, PC, 900, \$2350)$
c_4	$(*, *, Business, PC, 8600, \$6850)$

Table 1: A set of base and aggregated cells

Table 1 shows a set of base and aggregated cells. Tuple $c_1 \in \mathcal{D}$ is a *base cell*, while tuple c_2 is an *aggregated cell*. Here, the value of *cnt* in an aggregated cell c is the sum of the corresponding values in the complete set of descendant base cells of c, and the value of *avg_price* of c is the average price of the complete set of descendant base cells of c.

constraints.

Tuple c_3 is a sibling of c_2, c_4 is an ancestor of c_2, and c_1 is a descendent of c_2.

Suppose the significance constraint is $C_{sig} \equiv (cnt \geq 100)$. All cells (including base and aggregated ones) with *cnt* no less than 100 are regarded as *significant*. Suppose the probe constraint is $C_{prb} \equiv (city = \text{“}Vancouver\text{”}, cust\_grp = \text{“}Business\text{”}, prod\_grp = *)$. The set of *probe cells* $\mathcal{P}$ contains the set of aggregated tuples about the sales of the Business customer group in Vancouver, *for every product group*, provided the *cnt* in the tuple is greater than or equal to 100. It is easy to see $c_2 \in \mathcal{P}$.

Let the *gradient constraint* be $C_{grad}(c_g, c_p) \equiv (avg\_price(c_g)/avg\_price(c_p) \geq 1.4)$. The constrained gradient analysis problem specified by the three constraints is: Find all pairs (c_g, c_p), where c_p is a probe cell in $\mathcal{P}$, c_g is a sibling, ancestor, or descendant of c_p, c_g is a significant cell, and c_g's average price is at least 40% more than c_p's. $\Box$

If a data cube is completely materialized, that is, all the aggregated cells are computed and stored without considering constraints, the query posed in Example 1 becomes a relatively simple retrieval of the pairs of computed cells that satisfy the constraints. Unfortunately, the number of aggregated cells is often too huge to be precomputed and stored. Thus we assume only the base table is available, and it is our task to compute from it the gradient-probe pairs efficiently.

To confine our discussion, we first develop efficient methods for computing constrained *average* gradients, as posed in Example 1, and then extend our scope to more general cases in Section 5.

3 Mining Constrained Gradients in Data Cubes

In this section we examine efficient methods for mining constrained gradients in data cubes. First, we outline a relatively rudimentary algorithm, *All-Significant-Pairs*, and analyze its deficiencies. Then we propose a better algorithm, called *LiveSet-Driven*, which uses a set of relevant probe cells, called *LiveSet*, to prune potential gradient cells during iterative exploration. Moreover, an H-tree structure is developed for efficient computation with minimal replication of data. A rough analysis is given in the last subsection to compare the runtime of the two algorithms.

As discussed in Section 1, we believe that it is often the case that the significance and probe constraints combined are more restrictive than the gradient constraint. So we only consider approaches that first use the significance and probe constraints to restrict the search space.

3.1 A Rudimentary Approach: All-Significant-Pairs

Constrained gradients can be mined by a rudimentary algorithm, called **All-Significant-Pairs**. It first computes iceberg cube $\mathcal{P}$ consisting of all significant probe cells from $\mathcal{D}$ using the significance constraint C_{sig} and the probe constraint C_{prb}, and then for each probe cell, $c_p \in \mathcal{P}$, computes the set of gradient cells from $\mathcal{D}$ by using the gradient constraint $C_{grad}(c_g, c_p)$.

Both steps are carried out by using an efficient iceberg method, and they use the constraints to prune the search. One can use efficient iceberg cube computation algorithm, such as BUC [BR99] or H-Cubing [HPDW01] (see brief descriptions in Section 3.4). In our implementation we used the latter method. The computation in the first step should use the significance constraint C_{sig} and the probe constraint C_{prb}.

The computation in the second step uses the gradient constraint $C_{grad}(c_g, c_p)$. Optimization can be explored to prune the search for ancestors and/or descendants of a probe cell c_p based on the anti-monotonic relationships between them. If the gradient measure is an anti-monotonic function, such as count and sum of positive items, one can explore the following property: *if the measure m of a cell c is no greater than τ, none of c's descendants can have the measure m greater than τ; and if the measure m of a cell c is no less than τ, none of c's ancestor can have the measure m less than τ.* If the gradient measure is not an anti-monotonic function, such as average and sum of positive or negative elements, one can explore a weaker but anti-monotonic constraint to prune its ancestors and/or descendants. For example, for average, one can explore the property of top-k average[4] [HPDW01]: *if the top-k average of the base cells in a cell c is no greater than τ, where k is the significance constraint threshold value, then none of c's significant descendants can have the average value greater than τ.* Similarly, one can derive many other interesting properties to facilitate pruning for constraints involving some other complex measures.

Example 2 (All-Significant-Pairs) Let's examine how to perform constrained gradient analysis for the problem specified in Example 1, using the *All-Significant-Pairs* method.

First, we compute all the significant probe cells in the data cube by applying an efficient iceberg cube computation algorithm, such as H-Cubing [HPDW01], for the significance constraint $C_{sig} \equiv (cnt \geq 100)$ and the probe constraint $C_{prb} \equiv (city = \text{“}Vancouver\text{”}, cust\_grp = \text{“}Business\text{”}, prod\_grp = *)$. This

[4]For a multi-set of values, we define its top-k average as the average of the top-k values of the multi-set. For example, the top-3 average of the multi-set $\{2, 4, 5, 5, 8\}$ is 6. The top-k average for a cell is the top-k average of the measure values in the cell.

will yield a set of probe cells, e.g. $c_1 = (00, Vancouver, Business, PC, 300, \$200)$, $c_2 = (*, Vancouver, Business, PC, 800, \$1900)$, and so on. Let the set of significant probe cells be $\mathcal{P}$.

For each probe cell, $c_p \in \mathcal{P}$, we compute the set of gradient cells by using the gradient constraint $C_{grad}(c_g, c_p)$, and performing possible pruning of ancestors and/or descendants of the gradient cell currently under examination. The computation proceeds from top-level down (i.e., first computing high-level cells and then their descendants). If a cell c_g's top-k average value[5] (where $k = 100$, the minimum support threshold, i.e., significance constraint) is no more than $c_p \times 1.4$, then c_g and all of its descendants can be pruned since none of them can satisfy the gradient constraint. $\square$

The algorithm is summarized as follows.

Algorithm 1 (All-Significant-Pairs)

Input: A base relational table $\mathcal{D}$, a significance constraint C_{sig}, a probe constraint C_{prb}, and a gradient constraint C_{grad}.

Output: The complete set of gradient-probe pairs in the data cube derived from $\mathcal{D}$ that satisfy the three constraints.

Method:

1. Apply the iceberg cube computation algorithm H-Cubing, to compute the set $\mathcal{P}$ from $\mathcal{D}$ using significance constraint C_{sig} and the probe constraint C_{prb}.

2. For each probe cell, $c_p \in \mathcal{P}$, compute its ancestor, descendant, and sibling cells based on the cell gradient constraint, $C_{grad}(c_g, c_p)$. The computation is also carried out in a way similar to H-Cubing, and the search for c_p's descendants and ancestors can be pruned if c_p does not satisfy certain (transformed) constraints obtained from $C_{grad}(c_g, c_p)$ and c_p's measures. $\square$

This algorithm suffers from the following major deficiency:

- The search for gradient cells is done in a one-search-loop-per-probe-cell fashion. A huge amount of repeated work is performed for probe cells which are similar. It may involve computing the set of gradient cells $|\mathcal{P}|$ times, where $|\mathcal{P}|$ is the number of probe cells in $\mathcal{P}$, which is costly.

In the subsequent discussion, we will propose a better algorithm to overcome these deficiencies.

[5]The efficient computation of top-k average has been discussed in [HPDW01] and will be detailed also in Section 4.

3.2 The LiveSet-Driven Algorithm

To avoid the waste of resource for computing cells unrelated to the probe cells, it could be more preferable to first compute the set of iceberg probe cells $\mathcal{P}$ from $\mathcal{D}$, using both the probe and significance constraints. The second step is to use the set of derived iceberg probe cells $\mathcal{P}$ to efficiently constrain the search for interesting gradient-probe cell pairs, using the gradient constraint. This is similar to the golden rule of pushing selection deeply in relational query processing. We aim to design such an algorithm, with the additional bonus that, in the second step, the algorithm will not check more significant cells than *All-Significant-Pairs*, it will examine each significant cell only once (i.e., one pass), and it will use the probe cells to constrain the search in an "optimized" way.

To make the computation of the second step efficient, several techniques are developed as outlined below.

1. Using sets of probe cells to constrain the processing: To avoid the costly repetition of computation in the *All-Significant-Pairs* algorithm, we propose to use set-oriented processing and optimization. Roughly speaking, we associate with each gradient cell the set of all possible probe cells that might co-occur in interesting gradient-probe pairs with some descendants of the gradient cell, and use that set to prune future gradient cell search space.

2. Low-to-high dimension growth: The multi-dimensional space should be explored in a progressive and confined manner, using an "iceberg growth approach": Start at lower dimensional cells and proceed to higher dimensional ones. This is advantageous because there are usually a smaller number of lower dimensional cells than that of the higher dimensional ones. The anti-monotonicity property of significance constraints (or their weaker versions) and the (transformed) gradient cell constraints can be used to prune the remaining search space: if a k-d cell fails to satisfy a constraint, so will all of its descendants (higher dimensional cells). All three types of constraints, i.e., the probe, significance and gradient constraints, are used in this iceberg growth process.

3. Dynamic pruning of probe cells during the growth: During the dimension growth process, increasingly more probe cells fail to be associated with the higher dimensional gradient cells due to dimension value mismatch or the relevant measure value being out of the gradient range. Thus, one can prune the set of probe cells associated with the gradient cells in the growth. The search terminates when either no significant gradient cells can be generated or none of the probe cells can proceed further. The pruning of probe cells increases the power to prune gradient cells.

4. Incorporation of compressed data structure, H-tree, and efficient iceberg growth algorithm, H-cubing: For efficient computation of iceberg cubes, we also incorporate a compressed data structure, H-tree, and extend an efficient iceberg growth algorithm, H-cubing. This data structure and algorithm were shown to be highly efficient for computing iceberg cubes with complex measures [HPDW01]; they allow us to do maximal sharing between cells in the computation. This further enhances the efficiency of constrained gradient analysis.

3.2.1 Pruning gradient cells and probe cells using gradient constraints

Suppose $\mathcal{P}$, the set of probe cells, has been computed. The next step in the computation is to determine which cell (as gradient cell) should be associated with which probe cell to produce valid gradient-probe pairs. The computation will start from low dimensions and then proceed to higher dimensions, in a depth-first manner. Information on low dimension gradient cells will be used to prune higher dimension cells.

To study how the pruning should be performed, we need to introduce the concepts of *LiveSet* for probe cells and *potential cell* for gradient cells.

Definition 1 *The* **live set** *of a gradient cell c_g, denoted as $LiveSet(c_g)$, is the set of probe cells c_p such that it is possible that $(c_{g'}, c_p)$ is an interesting gradient-probe pair, for some descendant cell $c_{g'}$ of c_g.*

From this definition it is clear that the smaller *LiveSet* is, the more gradient cells can be pruned.

The determination of *LiveSet* involves the gradient constraint and the matches between dimensions of gradient and probe cells. This section only deals with the former, and the next section extends to deal with the latter.

Interestingly, pruning can be done in both directions between $LiveSet(c_g)$ and c_g:

(a) Obviously, $LiveSet(c_g)$ can be used to determine if c_g and its descendants have the potential to be interesting gradient cells w.r.t. (any probe cell in) $LiveSet(c_g)$; if not, c_g can be pruned.

(b) Information about c_g can also be used to prune probe cells c_p in $LiveSet(c_g)$. This involves checking whether c_g and its descendants have the potential to be interesting gradient cells w.r.t. c_p. If the answer is no, c_p can be pruned from the $LiveSet(c_g)$.

We now make precise the meaning of "having potential to be interesting gradient cells w.r.t. a set of probe cells."

Definition 2 *Let c_g be a gradient cell, C_p a set of probe cells, and C_{grad} the gradient constraint. We say c_g and its descendants have* potential *to be interesting gradient cells w.r.t. C_p if the following is true:*

(1) If the gradient constraint is anti-monotone (such as the sum constraint), then $C_{grad}(c_g, c_p)$ is satisfied for some $c_p \in C_p$.

(2) If the gradient constraint is not anti-monotone, such as $(avg\_price(c_g)/avg\_price(c_f) \geq v)$, then a transformed, weaker constraint can be potentially satisfied for some $c_p \in C_p$, such as $(avg^k\_price(c_g)/avg\_price(c_p) \geq v)$, where avg^k represents top-k average and k is the minimum support threshold (i.e., significance constraint). Observe that the avg^k constraint is a weaker anti-monotonic constraint constructed for the non-anti-monotonic avg constraint.

We say a gradient cell c_g is a **potential cell***, or* **has potential to grow***, if (i) c_g is significant and (ii) c_g or its descendants have* potential *to be interesting gradient cells w.r.t. $LiveSet(c_g)$.*

Observations: Some non-antimonotonic constraint, though itself cannot be used for pruning, can be transformed into a weaker, anti-monotonic constraint for pruning. For (2) above, we use $avg^k\_price(c_g)$ as an upper estimate of $avg\_price(c_{g'})$ for all significant descendant cells $c_{g'}$ of c_g.

We illustrate these with an example.

Example 3 Using the schema of Example 1, suppose $C_{grad}(c_g, c_p) \equiv (avg\_price(c_g)/avg\_price(c_p) \geq 1.4)$. Assume the set of probe cells $\mathcal{P}$ has been derived using some two constraints C_{sig} and C_{prb}. Let c_g be the 1-d cell $(00, *, *, *)$, which is assumed to be significant.

Suppose that initially[6] $LiveSet(c_g)$ is the following subset $\{c_{p_1}, c_{p_2}, c_{p_3}\}$ of $\mathcal{P}$, where c_{p_1} = (00, "*Vancouver*", "*Business*", $*$, 2800, \$1500), c_{p_2} = (99, "*Toronto*", $*$, *PC*, 7900, \$3000), and c_{p_3} = (00, "*Toronto*", "*Education*", *PC*, 450, \$2000).

We will illustrate with two scenarios.

(i) Suppose $avg^k\_price(c_g) = \$2500$. Since $avg^k\_price(c_g)/avg\_price(c_{p_1}) = 2500/1500 > 1.4$, c_g has potential to grow. However, because $2500/3000 < 2500/2000 < 1.4$, c_{p_2} and c_{p_3} can both be pruned from $LiveSet(c_g)$.

[6]The next section will discuss how *LiveSet* is derived.

(ii) Suppose $avg^k\_price(c_g) = \$2000$. Since $avg^k\_price(c_g)/avg\_price(c_p) < 1.4$ for each $c_p \in LiveSet(c_g)$, c_g does not have potential to grow, and can thus be pruned. $\square$

Let's consider how to use a set C_p of probe cells to prune gradient cells, where $avg\_price(c_p)$ is known for every c_p in C_p. Given a gradient cell c_g, clearly it is not efficient to check against all individual probe cells c_p in *LiveSet* whether the condition $avg^k\_price(c_g)/avg\_price(c_p) \geq 1.4$ holds. Fortunately, one can derive an overall gradient cell constraint for set C_p, $C_{gcell}(C_p)$, which specifies a range of measure values (such as average prices) for c_g and which must be satisfied by a gradient cell c_g if c_g *might co-occur in interesting gradient-probe pairs with any probe cell in* C_p. For example, if the minimal avg_price for all $c_p \in C_p$ is \$1200, then the (optimal) gradient cell constraint should be $C_{gcell}(C_p) \equiv (avg\_price(c_g) \geq \$1680)$. Because $avg^k(c_g)$ is an upper estimate of $avg(c')$ for all significant descendant cell of c_g, if $avg^k(c_g)$ cannot satisfy the constraint $avg^k(c_g) \geq \$1680$, then none of its descendants can satisfy it either. So c_g can be pruned by such a gradient constraint analysis.

In general, we have the following:

Property 3.1 (gradient cell constraint for a set of probe cells) If $C_{grad} \equiv (m(c_g)/m(c_p)\ \theta\ v)$, where θ is in $\{<, >, \geq, \leq\}$, v is a constant value, and $m(c_p) > 0$, then the gradient cell constraint corresponding to a set of probe cells C_p is $C_{gcell}(C_p)$, where

$$C_{gcell}(C_p) \equiv \begin{cases} m(c_g)\ \theta\ v \times min\{m(c_p) | c_p \in C_p\} & if\ \theta\ \in \{>, \geq\} \\ m(c_g)\ \theta\ v \times max\{m(c_p) | c_p \in C_p) & if\ \theta\ \in \{<, \leq\} \end{cases} \quad (1)$$

$\square$

This property can be used to derive a gradient cell constraint from a set of probe cells.

3.2.2 Pruning probe cells by dimension matching analysis

In the previous subsection we described how to use the gradient constraint to prune probe cells and gradient cells. In this subsection we describe what probe cells should be associated with a gradient cell, and how to prune the associated probe cells when the processing goes from a gradient cell to a descendant one; both will be from a dimension-matching perspective.

The dimension matching analysis is made possible under the assumption that we are only interested in gradient-probe pairs involving ancestor-descendant, descendant-ancestor, and sibling-sibling pairs.

Let c_g be a gradient cell. Recall that $LiveSet(c_g)$ denotes the set of probe cells c_p such that it is possible that $(c_{g'}, c_p)$ is an interesting gradient-probe pair for some descendant cell $c_{g'}$ of c_g. Hence, from a dimensional perspective, a probe cell c_p can be in $LiveSet(c_g)$ if (i) c_p is an ancestor or descendant of c_g, or c_g itself; or (ii) c_p is a sibling of some descendant of c_g or a sibling of c_g. It turns out that these conditions can be captured by a notion of "matchable," defined next.

Let $c_p = (d_{t1}, d_{t2}, \ldots, d_{tm})$ be a probe cell and $c_g = (d_{g1}, d_{g2}, \ldots, d_{gm})$ be a gradient cell. The number of **solid-mismatches** between the two cells c_p and c_g is the number of dimensions in which both values are not $*$ but are not matched (i.e., of different values). The number of **$*$-mismatches** between c_p and c_g is the number of dimensions in which c_p is $*$ but c_g is not. (Observe that the notion of $*$-mismatches is not symmetric and the cells are *playing certain roles.*) A probe cell c_p is **matchable** with a gradient cell c_g if either c_g and c_p have no solid-mismatch, or they have exact one solid-mismatch but no $*$-mismatch.

We now give an example to illustrate the notion of "matchability."

Example 4 Consider the 4-d probe cell $c_p = (a, b, *, d)$. c_p is matchable with its ancestor gradient cell $c_{g1} = (*, *, *, d)$ since c_{g1} contains neither $*$-mismatch nor solid-mismatch; c_p is matchable with its sibling $c_{g2} = (f, b, *, d)$ since c_{g2} contains only one solid-mismatch but no *-mismatch; c_p is matchable with $c_{g3} = (*, g, *, d)$ since c_{g3} contains one solid-mismatch but no $*$-mismatch (observe that c_{g3} is a sibling of a parent of c_p); c_p is matchable with $c_{g4} = (a, *, c, d)$ since c_{g4} contains no solid-mismatch (observe that c_p and c_{g4} have a common descendant, (a, b, c, d)); and also c_p is matchable with its descendant $c_{g5} = (a, b, c, d)$ since c_{g5} contains only one $*$-mismatch. However, it is not matchable with $c_{g6} = (*, c, e, d)$ since c_{g6} contains one solid-mismatch and one $*$-mismatch. $\square$

Property 3.2 (correctness of dimension analysis) c_p is matchable with c_g iff c_p is c_g, an ancestor of c_g, a descendant of c_g, or it is a sibling of c_g or of some descendant of c_g.

Rationale. For the "only if": Suppose c_p is matchable with c_g. Two cases arise: (a) c_g and c_p have no solid-mismatch. Let c' be obtained by taking the more specific value, for each dimension, from c_g and c_p. (Non-* values are not comparable, and each non-* value is more specific than the * value.) Then c' is a descendant of c_g and c' is a descendant of c_p. Hence c_p is an ancestor of some descendant of c_g. There are special cases here: if $c' = c_g$, then c_p is an ancestor of c_g; if $c' = c_p = c_g$, then c_p is c_g.

(b) c_g and c_p have exactly one solid-mismatch but no $*$-mismatch. Let c' be obtained by taking the more specific value, for each dimension, from c_g and c_p, except that c' takes the value of c_g for the dimension of the solid-mismatch. So c' is a descendant of c_g. Since there is no *-mismatch between c_p and c_g, each of the specific value also occurs in c_p. Clearly c_p and c' have exactly one solid-mismatch, and so c_p is a sibling of c'. Observe that c' can be c_g; in that case c_p is a sibling of c_g.

We omit the details of the "if." The non-trivial cases are illustrated in Example 4. □

We now discuss how dimension analysis is used for pruning *LiveSet* when the processing goes from a gradient cell to a descendant one.

Property 3.3 (relationship between livesets of ancestor-descendant cells) Let c_{g1} and c_{g2} be two gradient cells such that c_{g2} is a descendant of c_{g1}. Then $LiveSet(c_{g2}) \subseteq LiveSet(c_{g1})$.

Rationale. Let c_p be a probe cell such that (c_{g3}, c_p) might exist as an interesting gradient-probe cell pair for some descendant cell c_{g3} of c_{g2}. Since c_{g3} is a descendant of c_{g1} as well, the fact in the last statement implies that c_p is also in $LiveSet(c_{g1})$. □

This property ensures that we can produce the *LiveSet* of a descendant cell from that of the ancestor cell. The way to do that is simply to do a dimension matching analysis, plus a gradient-based pruning. We illustrate the dimension-matching based pruning using the following example:

Example 5 Let $c_{g1} = (*, *, c, *)$ be a gradient cell and let $c_{g2} = (*, b, c, *)$, which is a descendant of c_{g1}. Suppose $LiveSet(c_{g1}) = \{(*, *, *, *), (a, b, c, *), (*, b1, c, *), (a, b1, c, *), (*, b1, c1, *)\}$. Then $LiveSet(c_{g2}) = \{(*, *, *, *), (a, b, c, *), (*, b1, c, *), (a, b1, c, *)\}$, i.e. it is the result of pruning $(*, b1, c1, *)$ from $LiveSet(c_{g1})$. □

Notice that if the expansions of gradient cells follow a particular order (which is usually the case), then more pruning of the probe cells can be done. For example, if the dimensions are expanded from left to right, some descendants of c_g will be processed before c_g is processed (observe that the ancestor-descendant relationship is many-to-many). For instance, for $c_g = (a, *, c1, *)$ and $c_p = (a, b, c2, *)$, the only descendant of c_g which is a sibling of c_p is $(a, b, c1, *)$, which would have been processed earlier than c_g in the depth-first order, and thus c_p will not be counted in the *LiveSet* of c_g. To deal with this issue algorithmically, we can call $c_{g'}$ a *depth-first descendant*[7] of c_g if $c_{g'}$ is a descendant of c_g and

[7]We can provide a syntactic definition of "depth-first descendant". Let $c_p = (p_1, p_2, \ldots, p_m)$ and $c_d = (d_1, d_2, \ldots, d_m)$ be two m-dimensional gradient cells. Roughly speaking, c_d is a depth-first descendant of c_p if c_d is an expansion of c_p on the left, i.e., if they have a common suffix and c_d is the result of instantiating some $*$'s in the remainder of c_p. Formally, c_d is a *depth-first descendant* of c_p iff there exists an i such that $1 < i \leq m$, $(p_1, p_2, \ldots, p_i) = (*, *, \ldots, *)$, $p_{i+1} \neq *$, $(p_{i+1}, \ldots, p_m) = (d_{i+1}, \ldots, d_m)$, and $(d_1, d_2, \ldots, d_i) \neq (*, *, \ldots, *)$.

$c_{g'}$ is processed later than c_g in the depth-first order. In the dimension-based analysis, we just need to further restrict the *LiveSet* of a cell c to those depth-first descendants of c.

In this study, we assume that the set of probe cells, and hence the *LiveSet*, is usually a small set, which can be sorted in value ascending order according to certain measure values (see the next subsection) to facilitate pruning using gradient constraint. In case there is a large set, tree structure or hash table can be adopted for fast accessing.

3.3 The LiveSet-Driven Algorithm

Based on the above discussion, the *LiveSet*-driven algorithm is worked out for computing all the gradient-probe pairs which satisfy all the constraints. We first give an informal description using an example and then a formal algorithm. Efficiency issues regarding the data structure (H-tree) and its manipulation (H-cubing) will be discussed in the next subsection.

Our method starts with the 0-d cell of the cube, carrying the initial set of probe cells, P, as its *LiveSet*, and proceeds to higher dimensional gradient cells. Along the way, it uses the given constraints to prune the gradient cells which cannot satisfy the *LiveSet*, and to prune the cells in the *LiveSet* which cannot pass either gradient constraints or dimensional matching analysis. The processing along any branch terminates when the *LiveSet* becomes empty, or when the gradient cell has no potential to generate any interesting pairs.

Let's examine an example in more detail.

Example 6 (LiveSet-Driven) For the same base table schema $\mathcal{D}$ in Example 1, we examine how to perform constrained gradient analysis by the *LiveSet-Driven* algorithm. Let the gradient constraint be $C_{grad}(c_g, c_p) \equiv (ave\_price(c_g)/avg\_price(c_p) \geq 1.2)$, and the significance constraint be $C_{sig} \equiv (cnt \geq 100)$.

Let the set $\mathcal{P}$ of probe cells be given in Table 2, sorted in *avg_price* ascending order. Notice this order is important since once a probe cell in the table cannot satisfy the gradient constraints, all the cells following it cannot satisfy it either (since they carry an even larger measure value) and thus can all be pruned immediately.

The set of all probe cells $\mathcal{P}$ is the initial *LiveSet* for the 0-d gradient cell $c_0 = (*, *, *, *)$. Since 1500 is the lowest *avg_price* value among all current probe cells, it is taken as the global gradient lower bound. Suppose the top-100 average of the 0-d cell c_0 is 4000 and its count is 50000. Then

$(00, Vancouver, Education, PC, 100, 1500)$
$(99, Toronto, *, PC, 4000, 1800)$
$(*, Montreal, Business, PC, 1500, 8000)$
$(*, Edmonton, *, Ski, 2000, 10000)$
$(*, Whisler, *, Ski, 1000, 10050)$

Table 2: The set of probe cells, $\mathcal{P}$.

c_0 has potential to grow, because $4000 \geq 1.2 \times 1500 = 1800$ and $50000 \geq 100$. Now, the top-100 average of c_0 is used to prune the probe cells to generate a tighter *LiveSet* for c_0: Since the fourth cell $(*, Edmonton, *, Ski, 2000, 10000)$ cannot satisfy the gradient constraint due to $4000 < 1.2 \times 10000$, this cell and all probe cells with avg_price higher than 10000 in the *LiveSet* will be pruned. The actual average value of c_0 will decide which probe cell will be paired with this cell to become an interesting gradient-probe pair.

The computation then proceeds to process 1-d cells, 2-d cells, and so on, in a depth first manner. To avoid repetition and for the sake of clarity, we now show how the processing is done for a typical 3-d cell.

Suppose the first three probe cells are all alive after processing the 2-d gradient cell $c_2 = (00, Toronto, *, *)$, and the processing goes from this 2-d cell to the 3-d cell $c_3 = (00, Toronto, *, PC)$.

Probe cell	# of mismatches
$(00, Vancouver, Education, PC, 100, 1500)$	1
$(99, Toronto, *, PC, 4000, 1800)$	1
$(*, Montreal, Business, PC, 1500, 8000)$	$1, 1*$

Table 3: Number of mismatches in probe cells.

- We first prune the *LiveSet* of c_2 using dimensionality matching with c_3. The number of mismatches of each probe cell w.r.t. c_3 is presented in Table 3, where 1 indicates that there is one solid mismatch, and $1*$ indicates that there is one *-mismatch. Table 3 indicates that the first two probe cells remain alive with respect to the 3-d gradient cell c_3.

- The actual average value of c_3 decides which probe cell should be paired with this cell to become an interesting gradient-probe pair. If $avg\_price(c_3) = 1850$, then c_3 and the first probe cell form an interesting gradient-probe cell pair, but not c_3 and the second.

- This minimum average of the cells in *LiveSet*, 1500, and the top-100 average of c_3, will decide if we will continue processing with the descendants of c_3. If the top-100 average of c_3 is less than $1800 = 1.2 * 1500$, computation stops for this branch. If the top-100 average of c_3 is higher than or equal to $1800 = 1.2 * 1500$, computation continues. Suppose the top-100 average of c_3 is 1900. Then we go back to prune the current *LiveSet* of c_3. Because $1900 < 1800 * 1.2$, we can indeed prune the second probe cell, namely $(99, Toronto, *, PC, 4000, 1800)$, from the *LiveSet*.

In summary, we can see that the processing of a gradient cell c involves these steps: Derive an initial *LiveSet* from the *LiveSet* of the ancestor of the cell c, using dimension matching. The necessary measures and top-k average measures of c are computed, and checked against the *LiveSet* for answers and to decide if descendants of c may require processing. If processing of descendants is needed, then we prune *LiveSet* using the gradient constraint and the top-k average values. □

We now present the LiveSet-Driven algorithm.

Algorithm 2 (LiveSet-Driven)

Input and Output: The same as that of Algorithm 1.

Method:

1. Apply an iceberg cube computation algorithm to compute the set of iceberg probe cells $\mathcal{P}$ from $\mathcal{D}$ using significance constraint C_{sig} and probe constraint C_{prb};

2. Derive gradient cell constraint C_{gcell} for $\mathcal{P}$;

3. Initialize the potential gradient cell to $c = (*, ..., *)$. Initialize $LiveSet(c) = \mathcal{P}$.

4. Use a bottom-up, depth-first iceberg cubing method to find all interesting gradient-probe pairs. In depth-first processing, values in each dimension are ordered, and the dimensions are also ordered.

   for every value in each dimension do{

   1 If c is significant, for each live probe cell c_p in $LiveSet(c)$, output the gradient-probe pair (c, c_p) if the pair passes the gradient cell constraint.

   2 Use the measure (or transformed measure such as top-k) value of c to prune $LiveSet(c)$.

   3 If $LiveSet(c)$ is empty or c has no potential to grow, terminate this branch and backtrack to process the next cell according to the depth-first order.

4 If c has potential to grow, expand it to the next level, according to the depth-first order. If a descendant cell c' of c is processed from this expansion, derive $LiveSet(c')$ from $LiveSet(c)$ using the matchability test.

} □

3.4 H-Cubing: Efficient Data Storage and Manipulation Via H-tree

As shown in [BR99, HPDW01], bottom-up computation in cubes is efficient because it allows us to use low-dimension cells to prune high dimension cells. One important issue for realizing this efficiency is how much data is to be copied around and how much computation is to be shared. In this section, we review the spirit of a hyper-tree (called H-tree) structure and of an H-cubing algorithm, which allows us to use "minimal" copying of data and "maximal" sharing of computation. These tools have been useful for computing iceberg queries [HPDW01] and constrained gradient analysis; we believe that they will be useful for many other types of data cube computations.

The back-bone structure of H-tree is a basic tree, which gives a compressed representation of a base table. It uses some auxiliary structures to store necessary information that facilitates sharing of computation.

Roughly speaking, nodes in the hyper tree are labeled by attribute values and prefix sub-paths in the tree are shared whenever possible. Auxiliary structures include header tables, side-links, and quantitative information (quant-info). Quant-info can be used to help incrementally maintain the information needed for checking expensive constraints, using their weaker versions (such as top-k average). One design of quant-info for the top-k average constraint is to use a small number of "bins" to estimate safe lower bound of the top-k average value (more details can be found in [HPDW01]).

We illustrate the tree using an example with a small base table schema, (*customer-group*, *month*, *city*, *price*). The hyper-tree for the four tuples of $t_1 = (Education, Jan, Toronto, 485)$, $t_2 = (Household, Jan, Toronto, 1200)$, $t_3 = (Education, Jan, Toronto, 1280)$, and $t_4 = (Education, March, Vancouver, 520)$, inserted in this order, is shown in Figure 1. The quant-info shown is designed to deal with top-k average measures.

Definition 3 (H-tree) Given a base table T with attributes $A_1, \ldots, A_m, M$, the H-tree is defined as follows.

1. Attributes $A_1, \ldots, A_m$ are sorted in cardinality-ascending order $R : A_{j_1}, \ldots, A_{j_m}$ (to promote

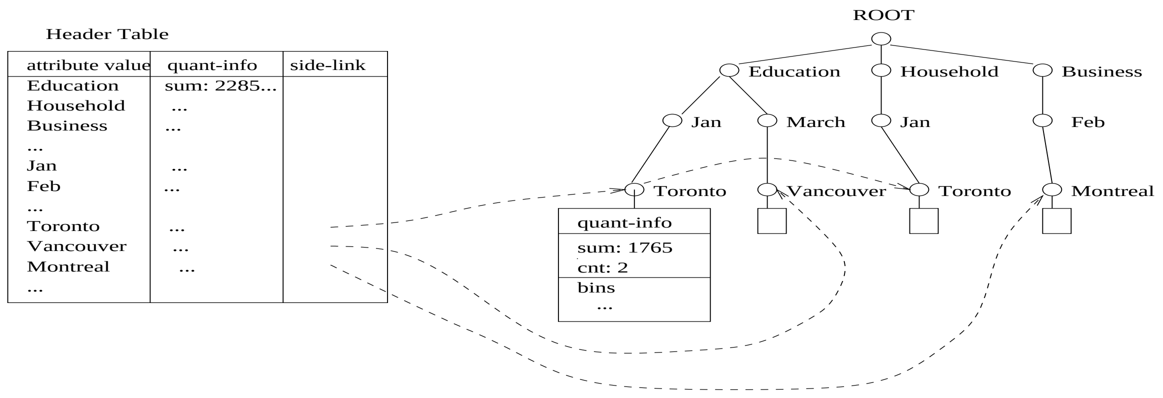

Figure 1: An H-tree.

sharing, since the number of values for an attribute is roughly proportional to the number of nodes in the level for the attribute).

2. H-tree has a root node labeled "*null*". Every other node in the tree is labeled by an attribute value. A quant-info and a side-link can be attached to a node if necessary, as explained above.

3. H-tree has a **header table** H with three fields: attribute-value, quant-info and side-link. Each attribute-value pair has one row in H; we omitted the attributes in Figure 1 for conciseness.

4. For each tuple t in table T, t is inserted into the tree as follows.

   (a) A tuple $t' = (a_{j_1}, \dots, a_{j_m}, p)$ is derived by projecting t on attributes $A_{j_1}, \dots, A_{j_m}, M$.

   (b) A path $ROOT$-a_{j_1}-$\cdots$-a_{j_m} is used to register tuple t'. Its maximal prefix already existent in the tree is used, and only the path or lower sub-path not existing in the tree is created.

   (c) Use p to update quant-info in (1) the leaf node of the path, and (2) the entries of $a_{j_1}, \dots, a_{j_m}$ in the header table H.

5. All leaf nodes with a common label are linked together as a queue by **side-links**. The side-link field of row for $a_{j_m} \in A_{j_m}$ in header table H is the head of the queue for $a_{j_m} \in A_{j_m}$. $\square$

The H-tree has several interesting properties which can facilitate computation with a data cube. In particular, it can be constructed by scanning the database only once, and, because of sharing of paths, its size is usually very small. (In fact, the actual size of the tree is notably smaller, as shown by experimental results in section 6 of [HPDW01].) Moreover, one can easily reconstruct all the information of the original table from the H-tree.

To obtain efficient sharing in carrying out data cube computations, we associate multiple header tables with one basic tree. Each header table will register the information corresponding to the computation for a group of cells (sharing some common structure). For example, in Figure 2, the header table on the left is associated with cells such as $(*, *, Toronto)$, ..., and $(*, *, Montreal)$, whereas the the header table on the right is associated with the cells $(*, Jan, Toronto)$, $(*, Feb, Toronto)$, ...

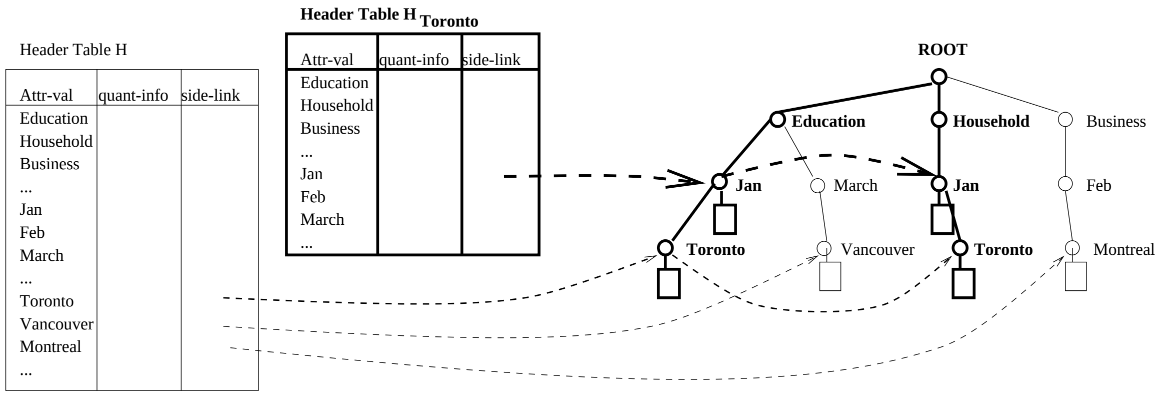

Figure 2: H-tree for $(*, *, Toronto)$

We first illustrate how computing is done using H-tree. Suppose we wish to process the cell $(*, *, Toronto)$. This can be done using the H-tree of Figure 1. The *quant-info* of the H-tree tells us the top-k average and average of cells of the form $(*, *, c)$, for each city c. From the row $Toronto$ in the header table H in H-tree, we get the $avg^k(price)$ and $avg(price)$ for $(*, *, Toronto)$.

We now discuss how "reuse" of computation is achieved in H-cubing with H-tree. We first do this by considering the processing of $(*, Jan, Toronto)$, a descendant of $(*, *, Toronto)$. For this, we will create a new header table, which we call $HT_{Toronto}$ (Figure 2) and the associated side-links. These will contain information about all paths of the tree related to city Toronto.

The *side-link* for $Toronto$ in header table H links all paths related to the city Toronto. By traversing the side-links only once, we can (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. city Toronto, and (3) link all parent nodes, of leaf-nodes labeled $Toronto$, having identical labels. The updated tree is shown in Figure 2.

Observation: Every parent node of a leaf node labeled $Toronto$ in Figure 2 has a copy of quant-info from its $Toronto$ leaf node. In the new header table $H_{Toronto}$, only attribute values for dimension $Customer\_group$ and $Month$ are needed. The quant-info in row Jan collects complete quant-info for sales in January and Toronto.

We now illustrate how computation is reused by considering the processing of $(*, Jan, *)$, a descendant of $(*, *, *)$. This involves a "roll-up of the quant-info" to dimension Month. Every leaf node in H-tree merges its quant-info into that of its parent node. (Before accepting quant-info from its children, a parent node resets its quant-info.) All nodes labeled by a common month, no matter what children they have, should be linked by side-links and also linked to corresponding row in 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* information are collected for such nodes, and no binning is needed, since they pass top-k average checking already. In further quant-info rolling up, parent nodes of nodes marked *top-k_OK* should also be similarly marked.

Even though H-tree compresses a database, we do not assume that an H-tree for an arbitrary database can always be held in main memory. A discussion on how to address the problem of handling large databases is given in [HPDW01].

3.5 A Rough Comparison of the Two Algorithms

Since the execution of the two algorithms depend on the data and the parameter settings, it is hard if not impossible to give closed formulas for their runtime. However, we are able to offer a rough comparison of their runtime as follows.

Observe that both algorithms use the H-cubing algorithm, and the difference in performance is due to the one-pass set-oriented processing and pruning of LiveSet-Driven and the repeated computation of All-Significant-Pairs. Let $\mathcal{P}$ denote the set of all probe cells to be considered. For each probe c_p in $\mathcal{P}$, let $\mathcal{C}(c_p)$ denote the set of all gradient cells that are examined by the All-Significant-Pairs algorithm. Let $\mathcal{C}$ denote the set of all gradient cells that are examined by the LiveSet-Driven algorithm. Then $\mathcal{C}$ should be equal to $\cup_{c_p \in \mathcal{P}} \mathcal{C}(c_p)$. However, note that the All-Significant-Pairs algorithm may examine each gradient cell a number of times; let rp denote the average of these numbers. Moreover, note that the LiveSet-Driven algorithm incurs some overhead in order to do the set-oriented processing and pruning; let $ovhd$ denote the average overhead for a gradient cell. Then we see that the runtime of LiveSet-Driven is roughly $(1 + ovhd)|\mathcal{C}|$, whereas the runtime of All-Significant-Pairs is roughly $rp|\mathcal{C}|$. Hence the speedup by the LiveSet-Driven algorithm is roughly $\frac{rp}{1+ovhd}$. Our experiments show that the speedup is usually approximately 10 fold.

4 Performance Analysis

In this section, we report our experimental results on computing gradients in data cubes. The results show that the LiveSet-Driven algorithm is scalable and much faster than the All-Significant-Pairs algorithm.

As we will see soon, the speed up is roughly proportional to the number of probe cells. As noted earlier, both algorithms use the probe and significance constraints to restrict the set of probe cells. The difference in their execution times is due to the following reasons: The All-Significant-Pairs algorithm does an independent search for each probe cell. As a result, it leads to much "repeated search" for different probe cells. On the other hand, the LiveSet-Driven algorithm bundles the gradient-cell search for all probe cells in one pass, and uses techniques for two-way pruning (i.e. the pruning of probe cells using information about a gradient cell under consideration, and the pruning of gradient cells using the set of probe cells).

All experiments were conducted on a PC with an Intel Pentium III 700MHz CPU and 256M main memory, running Microsoft Windows/NT. All programs were coded in Microsoft Visual C++ 6.0.

The experiments were conducted on synthetic data sets generated using the data generator described in [HPDW01]. Parameters for the generator include: the number n of tuples in the base table, the number m of dimensions, the cardinality of each dimension, m_min and m_max for the range of the measure, a repeat factor τ which dictates the number of tuples to be generated based on some model tuples with uniform distribution, and a noise factor in percentages to represent the fraction of the n tuples to be generated using a random distribution (to distort the repeat factor). The generator uses $rand()$, a function which generates numbers in the range of $[0, 1]$ following uniform distribution. It works by repeatedly adding $r = rand() \times \tau$ number of new tuples $t'_1, \cdots, t'_r$, until we get enough tuples, using steps (1–4) as follows:

(1) let $D_{i_1}, \ldots, D_{i_d}$ be d dimensions randomly picked from $D_1, \ldots, D_m$, where $d = rand() \times m$;

(2) let $a_{i_1}, \ldots, a_{i_d}$ be d values randomly picked from dimensions $D_{i_1}, \ldots, D_{i_d}$, to be used as the "repeating" values;

(3) let $t'_i[D_\ell] = a_\ell$ for $D_\ell \in \{D_{i_1}, \ldots, D_{i_d}\}$, and randomly generate $t'_i[D_\ell]$ for $D_\ell \notin \{D_{i_1}, \ldots, D_{i_d}\}$;

(4) randomly generate measure values of $t'_1, \cdots, t'_r$, with normal distribution with mean $c = m\_min + rand() \times (m\_max - m\_min)$.

We conducted experiments on various synthetic datasets generated by this generator. The results are similar. Limited by space, except for performance with respect to the number of tuples, we report here only results on some typical data sets, with 10 dimensions and between 10,000-20,000 tuples. The cardinality for every dimension is set to 10^8. The measures are in range of [100, 1000]. The noise factor is set to 20% and repeat factor is 200.

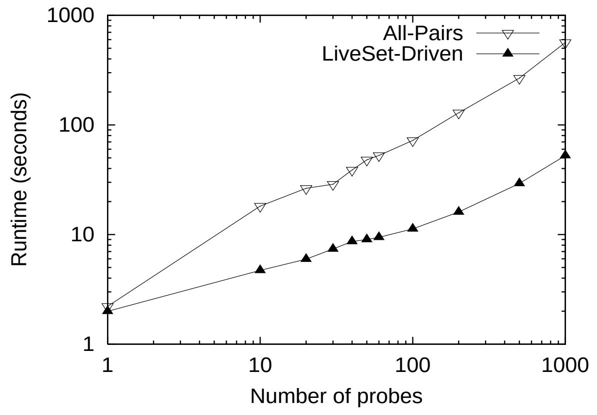

Figure 3: Scalability over number of probe cells

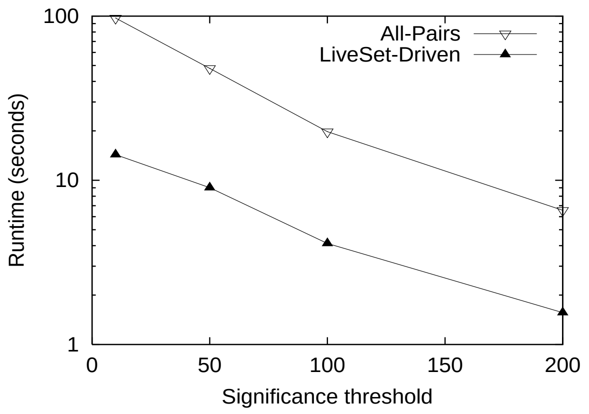

Figure 4: Scalability w.r.t. significance threshold

The first data set we used has 10,000 tuples. We tested the scalability of the algorithms with respect to number of probes in Figure 3, significance threshold in Figure 4, and gradient threshold in Figure 5.

Figure 3 shows the scalability of the two algorithms, All-Significant-Pairs and LiveSet-Driven, with respect to the number of probe cells. We set the significance threshold to 10, the number of bins to 3 for top-k average, and the gradient threshold is 2. The number of probes varies from 1 to 1,000. When the number of probes is small, both algorithms have similar performance. However, as the number of probes grows, the pruning power of LiveSet-Driven algorithm takes effect. In in one pass, it combines the searches for all probe cells and prunes unfruitful searches, and so it keeps the runtime low. In contrast, the All-Significant-Pairs algorithm does not scale well under large number of probes, because it does one independent search for each probe cell.

Figure 4 shows the scalability of both algorithms with respect to the significance threshold. The gradient threshold is set to 1.2, the number of bins to 3 and the number of probes to 50. LiveSet-Driven achieves good scalability by pruning many cells in the search whereas All-Significant-Pairs checks a huge number of pairs of cells.

[8]The smaller the cardinality, the denser the data cube, and thus the larger number of cells satisfy the constraints.

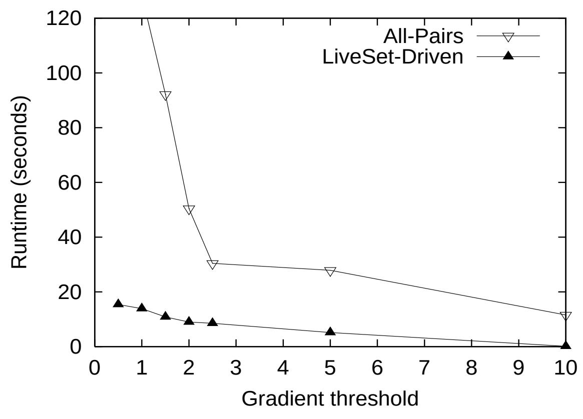

Figure 5: Scalability w.r.t. gradient threshold

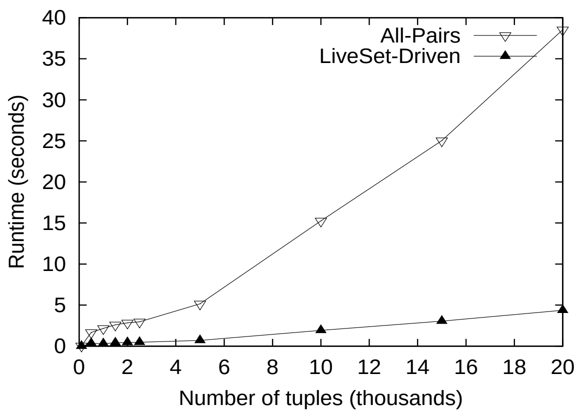

Figure 6: Scalability w.r.t. number of tuples

Figure 5 shows the scalability of All-Significant-Pairs and LiveSet-Driven with respect to various gradient thresholds. We fixed the significance threshold to 10, number of bins to 3 and number of probes to 50. As the gradient threshold goes down, the number of cells that All-Significant-Pairs has to check increases dramatically, and thus its runtime increases dramatically as well.

Figure 6 shows a scaling-up experiment with respect to various number of tuples, varying up to 20,000. We set the significance threshold to 1% of the number of tuples, the gradient threshold to 1.2, the number of bins to 3 and the number of probes to 100. While both algorithms are scalable, LiveSet-Driven naturally is more efficient.

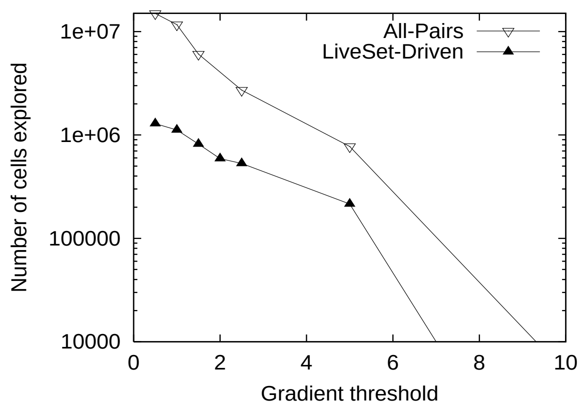

Figure 7: Using gradient in pruning

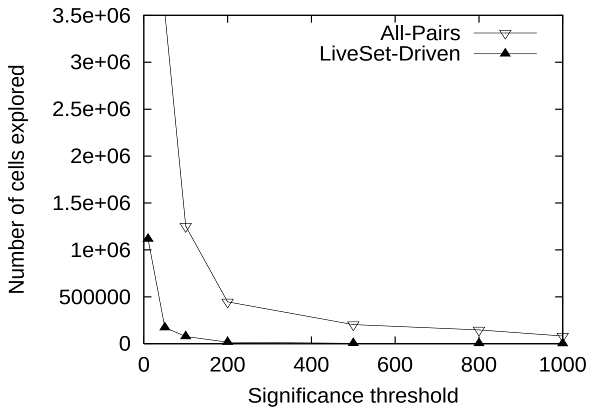

Figure 8: Significance threshold and pruning

We also analyzed the number of cells explored by each algorithm during the mining process on a $10,000$-tuple dataset with 50 probe cells. Figure 7 presents the number of cells that the two algorithms explored with respect to various gradient thresholds. It confirms that LiveSet-Driven achieves better pruning than All-Pairs. As shown in the figure, LiveSet-Driven on average explores only about one tenth of the cells All-Significant-Pairs does. That explains the difference of efficiency and scalability between the two algorithms.

Similar statements can be made about Figure 8, where the significance threshold varies from 10 to $1,000$. LiveSet-Driven explores a substantially smaller set of cells than All-Significant-Pairs does.

5 Discussion

In this section, we examine various alternatives of constraints and gradients for gradient mining in data cubes, extend our scope to mining constrained gradients in transactional databases, and discuss the related work.

5.1 Variations for mining constrained gradients in data cubes

The last two sections presented an efficient method for mining constrained gradients in multi-dimensional space. Here we discuss possible extensions and refinements of the method in various kinds of alternative situations, as follows.

1. **Finding constrained gradients among only ancestors, descendants, or siblings.**

   Algorithm 2 (LiveSet-Driven) searches for three kinds of relationships: *ancestors, descendants*, and *siblings*, at the same time. In some applications, people may be interested in only one or two kinds but not all kinds. How should we modify the algorithm to ensure efficient computation when we want to find only siblings, only ancestors, or only descendants?

   This can be easily addressed by modifying the definition of *LiveSet* for potential gradient cells. For each special case, we will remove more probe cells than the general case in accordance with user-restrictions on what cells are "comparable", and there is no need to change other parts of the algorithm. As a result, the algorithm will be more efficient.

   Similar extensions can be worked out if a user would like to find the constrained gradients only in relevance to *a small subset of dimension combinations*, such as $\{D_i, \ldots, D_j\}$ in a data cube. In this case, starting with the 0-d cell, $(*, \ldots, *)$, the set of (non-*) gradient cells to be considered and tested will be confined to only those in a subset of dimensions $\{D_i, \ldots, D_j\}$.

2. **Finding multi-dimensional gradients constrained by an "interval".**

   Our algorithm searches for multi-dimensional gradients by checking a single gradient constraint, such as $C_{grad}(c_g, c_p) \equiv (g(c_g, c_p)\ \theta\ v)$, where θ is in $\{<, >, \geq, \leq\}$, v is a constant value, and g is a *gradient function*. In many cases, the desired constraint could be an interval, such as $1.4 \leq g(c_g, c_p) \leq 2.5$. In such cases, one can modify the gradient testing part of the algorithm by testing not only the lower bound on the top-k average of the measure (i.e., no less than $1.4 \times avg(c_p)$) but also the upper bound on the bottom-k average of the measure (i.e., no more than $2.5 \times avg(c_p)$). Clearly, the computation for the bottom-k average will be similar to that for the top-k average. Whether it is more efficient to prune the search space using both upper and lower bounds or using only one of them and postponing the evaluation of the other after the constraint evaluation will depend on the gradient constraint values and the data set.

3. **Replacing ratio-based gradients by differences as "gradient" constraint.**

   Although our algorithm handles ratio-based gradients, with some slight modification, one can handle gradients defined with differences. Suppose the new gradient constraint is

   $$C_{grad}^{dif}(c_g, c_p) \equiv (avg\_price(c_g) - avg\_price(c_p) \geq 400).$$

   Let

   $$C_{grad}^{div}(c_g, c_p) \equiv (avg\_price(c_g)/avg\_price(c_p) \geq 1.4).$$

   Basically, we will only need to change the algorithm for C_{grad}^{div} by replacing 1.4 with 400, and $\times$ with $+$, respectively. For example, $avg\_price(c_g) \geq 1.4 \times avg\_price(c_p)$ will be replaced by $avg\_price(c_g) \geq 400 + avg\_price(c_p)$. The algorithm and our previous discussion still hold.

4. **Finding similar (or stable) patterns, i.e., the measures which are similar when some dimension values change**.

   "Similar" (or stable) can be defined in one of the following two ways:

   $$C_{grad}^{sim1}(c_g, c_p) \equiv (|avg\_price(c_g) - avg\_price(c_p)| \leq 50)$$

   $$C_{grad}^{sim2}(c_g, c_p) \equiv (|1 - avg\_price(c_g)/avg\_price(c_p)| \leq 0.1)$$

Observe that similar cells are expected to have similar measure values. Hence the number of similar cells is normally large, and so most of the pairs of similar cells will not be interesting. To find interesting pairs, one should impose some other constraints so that the uninteresting pairs are eliminated. Our algorithm can be adapted to deal with such constrained similar cell queries. (It may be better to compute the non-constrained similar cell query as displayed above as a number of range queries.)

5. **What will happen if we replace avg by sum and count?**

   We have been using the measure "average" in our gradient analysis because it is natural to define interesting gradients as substantial changes on the measure "average". However, if we replace avg by sum and count, an ancestor cell should naturally have much bigger sum or count values than its descendants. In this case, the simple gradient definition, such as $g(c_g, c_p) \geq v$ may not be so interesting.

   In this case, some "normalized" definition of gradients will make more sense. For example, one may compare the cells (siblings, ancestors, or descendants) with only relatively comparable size (i.e., containing almost the same number of cells), or the expected values of sum/count based on the proportional size (which is similar to average). Only those which are substantially larger or smaller than the expected values will be caught as "interesting" cells. In this context, our algorithm can also be made to work with minor modifications.

6. **Other ways to define "comparable cells."**

   Our discussion has been confined to finding large ratios or differences among ancestors, descendants, and 1-d siblings. There could be many other ways to define "comparable cells." For example, one may want to find comparable cells in 2-dimensional mutations, or find groups of comparable cells with user-specified explicit constraints. Some corresponding modifications of the definition and rules for derivation of *LiveSet* will make our algorithm adaptable to these cases.

5.2 Mining constrained gradients in transactional databases

We now examine how to extend our model to mining constrained gradients in transactional databases. In comparison with mining transaction-based association rules [AIS93, AS94], a distinct feature for mining constrained gradients is that its measure is not confined to frequency counting (i.e., support) but extended to complex measures, such as sum of the sales, average sales price, profit, and so on.

Let's examine such an example.

Example 7 (Gradient mining in transactional databases) Suppose a database in *E-City* stores a large set of customer shopping history. Each tuple (representing one customer) contains a set of items bought, together with a sales price of each item. A few of such tuples are shown below.

customer	items_bought
c_1	(*desktop_pc* : \$529), (*digital_camera* : \$129), (*dvd_player* : \$199), (*repair_kit* : \$45)
c_2	(*desktop_pc* : \$1299), (*laptop_pc* : \$2599), (*dvd_player* : \$699), (*video_camera* : \$1295)
...	....
c_{405}	(*washer* : \$559), (*dryer* : \$429), (*color_tv* : \$2099), (*digital_camera* : \$695)

Table 4: A set of "transactions" (customer shopping history) in the E-city database

An example constrained gradients query can be: find situations where the average sales price of one kind of product (such as *digital_camera*) may be substantially higher than that at some other situations. One may find that the following relationships could be interesting: (1) the average price of *digital_cameras* sold is 20% higher than usual when customer also bought (or will buy) *laptop_pcs*, and (2) the average price of *color_tvs* sold is 60% higher when customers also bought or will buy *dvd_players* in comparison with situations where customers also bought or will buy *repair_kit*.

In such a transactional database, the *significance constraint* may correspond to the minimum number of transactions containing the itemset under consideration, the *probe constraint* can be user's interest, which can be single items, such as *digital_camera*, or itemsets, such as $\{$*color_tv*, *dvd_player*$\}$, and the *gradient constraint* can be a ratio such as ≥ 1.2 (which means at least 20% higher of one average sales price than the other). From this point of view, one can see that the (constrained) transaction gradient problem shares a lot of similarities with the (constrained) cube gradient problem discussed above. □

Can we mine constrained gradients in a transaction database using the same method as we developed for mining constrained gradients in data cubes? In principle, our model and algorithm developed for constrained gradients in data cubes should be still applicable to mining transaction-based gradients with complex measures. However, since a transaction database may contain a large number of distinct items which may correspond to a huge number of dimensions, and since a transaction usually contains only a very small portion of possible items in a transaction database (i.e., very sparse), a data cube-based method may not lead to an efficient solution. Moreover, the previously proposed H-tree structure is not appropriate for storing and mining of such gradients due to the sparsity of the data and the large number of dimensions.

To overcome this difficulty, we propose a probe-based *FP-growth* method described below.

1. Scan the transaction database once to find frequent single items and sort these single items in item-frequency descending order to obtain an *f_list*.

2. For each probe itemset p, construct p's *FP-tree* as follows. For each transaction containing p, following the *f_list* ordering, insert every frequent item i other than p into the tree, and update the corresponding node i by (1) incrementing the *count*, and (2) adding the current price to the sum of p's *sales*. (Notice that the sum of p's sales of a new node is first initialized to zero).

3. Call the *FP-growth* algorithm [HPY00], or other similar frequent pattern mining algorithms to calculate count and sum of sales for itemsets in p's *FP-tree* and print those whose both support and gradient are no less than their corresponding thresholds.

   Note: To ensure that we cover all ancestor-descendant pairs of the desired probe itemsets, we also build an *FP-tree* for each subset of each probe itemset.

4. As an optimization, each node may store p's count and sum of sales partitioned according to their price range in a few bins, in the same way as we discussed before. Then top-k optimization can be explored to prune the search, based on the heuristic: *if the top-k average of an itemset s is less than $avg(p) \times gradient\_threshold$, its projected database will not need to be mined.*

The correctness and efficiency of this method can be easily verified in a similar way as we discussed in Section 3, and will be left to interested readers as an exercise.

5.3 Related work

The closest work related to our study on multi-dimensional gradient analysis is that on the cubegrade problem by Imielinski, et al. [IKA02]. A cubegrade query asks for association-type rules that describe changes in measure values associated with changes in dimension descriptions of cuboids. It deals with questions such as "what cube changes are associated with significant measure changes." Cubegrade queries can also have constraints that restrict the attributes in the gradient cells, other than those allowed by roll-up, drill-down and mutation. Our constrained gradient analysis does not have user-defined constraints on gradient cells. However, as shown in our discussion, they can be easily dealt with by adding more power to prune *LiveSet*. Thus adding user-defined constraints will actually lead to more efficient processing.

The main contributions of [IKA02] are the cubegrade framework and the proposed language. It considered a relativized notion of monotonicity (w.r.t. a cube or a constrained cube), the so-called structural monotonicity, which can be tested quite efficiently. The evaluation strategy proposed in that paper uses multiple loops: for each probe cell, search through the entire space for potential gradient cells. It will have a serious efficiency problem if we generalize the notion of "comparable" cells as we discussed above, because the search space per probe cell will be large, and this search will be repeated once per probe cell. While constraints may have been used for pruning, the effect of this pruning on efficiency is not clear in the paper.

In general, the proposed method in [IKA02] is similar to our *all-significant-pairs* approach (Algorithm 1). Based on our performance analysis, our *LiveSet*-Drive method leads to a more efficient solution. This is due to grouped processing of live probe cells and pruning of search space by pushing various kinds of constraints deeply.

There are also a few other studies on efficient exploration of interesting cells in data cubes or interesting rules in multi-dimensional space.

Reference [SAM98] considers discovery-driven exploration of OLAP data cubes. It computes anticipated values for a cell using the neighborhood values of the cell, and a cell is considered an exception if its value is significantly different from its anticipated value. This is rather different from what has been studied in this paper where "interestingness" is defined based on a user-specified gradient ratio in relevance to the cell's ancestors, descendants, and siblings. Therefore, the computational methods adopted in two studies are rather different. The former ([SAM98]) is based on the statistical analysis of neighborhood values of a cell to determine whether it is an exception; whereas the latter (our study) is cube-based computation of constrained gradients. Also, the former is on interactive exploration of *computed* cube cells; whereas the latter is on computing (nonmaterialized) cells (more exactly, pairs of cells) satisfying certain constraints. Both definitions may find their corresponding applications. It is an interesting issue to see whether our computation can be used as a filtering process and feed the results into the statistical analysis of neighborhood cells to reduce the overall processing cost of discovery-driven exploration of OLAP data cubes.

More recently, [SS01] considered the so-called "intelligent rollup" operation on datacubes, which allows an analyst to discover the most specific generalizations of a pair of cells with some interesting properties. The approaches proposed by the authors are different from ours.

[DL99] considers the mining of the so-called emerging patterns, patterns whose frequency change ratio between two datasets is larger than a certain threshold. In a data cube environment, two base tables are first extracted from the base data cube, one with base cells satisfying one property and the other with base cells satisfying another property. Emerging patterns are then the aggregated cells where the measure changes significantly between the two corresponding data cubes. Notice unlike the model constructed in our study, the method developed in [DL99] cannot handle arbitrary measures, such as avg. It is still a research issue on how to efficiently compute such complex gradients in an association mining environment.

[AL99] considers how statistics (a measure) of one group of tuples differs from the same measure of a supergroup. It shows that, by adopting such difference or ratio measure, the number of association rules can be reduced substantially and only the interesting rules are preserved. This shares a similar motivation as our study here. However, our study provides a general mechanism to specify constraints and any kind of measures and/or gradients in relevance to ancestors, descendants and siblings, and thus provides a more general model, as well as an efficient constraint-pushing and computation method. We believe that our method can serve as an efficient preprocessing step for subsequent statistical studies on mined interesting gradients or rules.

Our study is also closely related to (1) data cube and iceberg cube computation methods proposed in previous studies, such as [HRU96, AAD$^+$96, ZDN97, CD97, FSGM$^+$98, GCB$^+$97, RS97, BR99, HPDW01], as well as (2) constraint-based data mining methods, such as [SVA97, NLHP98, GLW00, PHL01]. This study can be considered as (1) an extension of data cube computation to mining interesting gradients, and (2) an extension of constraint-based mining toward mining constrained gradients in data cubes. Thus it is an extension and integration of both mechanisms towards efficient, multi-dimensional, constrained gradient analysis.

6 Conclusions

In this paper, we have studied issues and methods on efficient mining of multi-dimensional, constrained gradients in data cubes. Constrained gradients are substantial changes in a set of measures (aggregates) of interest associated with the changes in the underlying characteristics of cube cells, where changes in characteristics are expressed in terms of the dimensions and are limited to specialization, generalization, and 1-d mutation. To ensure only interesting changes of relevant cells are studied, we show that it is necessary to introduce three kinds of constraints: *significance constraints*, *probe constraints*, and *gradient constraints*.

An efficient algorithm, *LiveSet*-driven, has been developed which explores set-oriented processing and the maximal pushing of the constraints as deeply as possible in the early stage of the mining process to prune the search space. Moreover, we also adopt a compressed hyper-tree structure to represent the base table of a data cube, and to achieve "maximal" sharing of computation among different cells. Our performance study shows that this method is efficient and scalable. It outperforms another method which relies on the iceberg cube computation of all-significant-pairs.

Furthermore, we have briefly introduced the mining of constrained gradients in transaction databases, as well as a few alternatives in gradient mining. The integration of constrained gradient mining with discovery-driven exploration of data cubes [SAM98] is an interesting issue for future research.

References

[AAD$^+$96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 506–521, Bombay, India, Sept. 1996.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'93)*, pages 207–216, Washington, DC, May 1993.

[AL99] Y. Aumann and Y. Lindell. A statistical theory for quantitative association rules. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, San Diego, CA, Aug. 1999.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, Sept. 1994.

[BR99] 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.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26:65–74, 1997.

[DL99] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 43–52, San Diego, CA, Aug. 1999.

[FSGM$^+$98] 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.

[GCB$^+$97] 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.

[GLW00] G. Grahne, L. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proc. 2000 Int. Conf. Data Engineering (ICDE'00)*, pages 512–521, San Diego, CA, Feb. 2000.

[HPDW01] 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.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, Dallas, TX, May 2000.

[HRU96] 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.

[IKA02] T. Imielinski, L. Khachiyan, and A. Abdulghani. Cubegrades: Generalizing association rules. *Data Mining and Knowledge Discovery*, 6:219–258, 2002.

[NLHP98] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 13–24, Seattle, WA, June 1998.

[PHL01] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 433–332, Heidelberg, Germany, April 2001.

[RS97] 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.

[SAM98] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *Proc. Int. Conf. of Extending Database Technology (EDBT'98)*, pages 168–182, Valencia, Spain, Mar. 1998.

[SS01] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional OLAP data. In *Proc. 2001 Int. Conf. on Very Large Data Bases (VLDB'01)*, pages 531–540, Rome, Italy, Sept. 2001.

[SVA97] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 67–73, Newport Beach, CA, Aug. 1997.

[ZDN97] 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.