Pushing Aggregate Constraints by Divide-and-Approximate [*]

Ke Wang
Simon Fraser University
wangk@cs.sfu.ca

Yuelong Jiang
Simon Fraser University
yjiang@cs.sfu.ca

Jeffrey Xu Yu
Chinese University of Hong Kong
yu@se.cuhk.edu.hk

Guozhu Dong
Wright State University
gdong@cs.wright.edu

Jiawei Han
University of Illinois at Urbana-Champaign
hanj@cs.uicu.edu

Abstract

Iceberg-cube mining *is to compute the GROUP BY partitions, for all GROUP BY dimension lists, that satisfy a given aggregate constraint. Previous works have pushed* anti-monotone *constraints into iceberg-cube mining. However, many useful constraints are not anti-monotone. In this paper, we propose a novel strategy for pushing general aggregate constraints, called* Divide-and-Approximate. *This strategy divides the search space and approximates the constraint in subspaces by a pushable constraint. As the strategy is recursively applied, the approximation approaches the given constraint and the pruning tights up. We show that all constraints defined by SQL aggregates, arithmetic operators and comparison operators can be pushed by Divide-and-Approximate. We present an efficient implementation for an important subclass and evaluate it on both synthetic and real life databases.*

1 Introduction

Decision support systems, which rapidly gain competitive advantage for businesses, make heavy use of aggregations for identifying trends. The *iceberg query*, introduced in [5], performs an aggregate function over a specified dimension list and then eliminates aggregate values below some specified threshold. The prototypical iceberg query based on a relation $R(target1, \ldots, targetk, rest)$ and a threshold T is as follows:

```
SELECT target1, ..., targetk, count(rest)
FROM R
GROUP BY target1, ..., targetk
HAVING count(rest)>=T
```

The tuples are partitioned according to the GROUP BY dimension list and one result row is produced for each GROUP BY partition with $count(rest)$ above the threshold T. In the *iceberg-cube mining* [3], the user does not have to specify a GROUP BY list, and the result will be computed for *all* GROUP BY lists. On a relation R(Product, Store, Year, rest), for example, it will perform the aggregate function for all eight GROUP BY lists over Product, Store, Year. Each GROUP BY partition is called a *cell* in the cube. The iceberg-cube mining is to find all above-threshold cells.

The drawback of performing one iceberg query for each GROUP BY list is that the work in different queries is not shared. Computing the full cube often is unrealistically expensive [3]. A promising approach to iceberg-cube mining is to "push" the constraint so that only likely satisfying cells are examined. Previous works have pushed *anti-monotone* constraints [3, 1]: if a cell fails the constraint, so does every super-cell (for a longer GROUP BY list). The well-known anti-monotone constraint is $count(rest) \geq T$, because a super-cell cannot have a larger count than a sub-cell. A *monotone* constraint, e.g., $count(rest) \leq T$, allows pruning in the opposite direction: if a cell fails, so does every sub-cell.

1.1 The problem

In this paper, we study the "push" approach for general aggregate constraints of the form $f(v)\theta\sigma$, where f is defined by SQL-like aggregates and arithmetic operators $+, -, \times, /$, and θ is a comparison operator and σ is a real. The basis of this study comes from the following observations about anti-monotone constraints. Similar observations apply to monotone constraints.

Anti-monotonicity is too loose as a pruning strategy. The anti-monotonicity implies an exponential lower bound

[*]The work was supported in part by the Natural Sciences and Engineering Research Council of Canada, and in part by the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK4229/01E)

on the result size because *all* 2^k sub-cells of a satisfying cell of length k are satisfying. In real life applications, this size often imposes a big problem. For example, to find the cells that distinguish the "responder" class from the "non-responder" class in the KDD-CUP-98 direct marketing dataset [7], the support threshold must be much lower than 5%, the percentage of "responder" tuples. In this case, the size k and the number of satisfying cells are very large. This size problem also makes it impossible for a human user to comprehend the mined result.

Anti-monotonicity is too restricted as an interestingness criterion. In practice, many useful constraints are not anti-monotone. For example, $sum(v) \geq \sigma$ and $avg(v) \geq \sigma$ specify the minimum "net profit" requirement and minimum average "net profit" requirement, respectively, where a positive measure value corresponds to "profit" and a negative measure value corresponds to "cost". Another example is the maximum variance constraint, $var(v) \leq \sigma$, which specifies the minimum homogeneity requirement on the measure involved. This constraint is particularly useful for extracting rules to predict the measure value (e.g., profit) because the variance corresponds to the prediction error.

Since the absence of anti-monotonicity and monotonicity is the norm in the real life, techniques for pushing constraints under such conditions will contribute significantly to real life applications. Unfortunately, pushing aggregate constraints without anti-monotonicity or monotonicity presents a significant challenge: even if a cell fails, its super-cells or sub-cells still need to be considered. This is why most previous works essentially require anti-monotonicity or monotonicity to push a constraint.

1.2 The proposed approach

We investigate two questions. First, if a constraint is not anti-monotone or monotone, can we push the constraint into the iceberg-cube mining? Second, is there a pushing methodology that is constraint independent, i.e., not specific to a particular constraint? Two thoughts underpin our approach.

Divide-and-Approximate. If the given constraint is not anti-monotone, the next best thing to do is to approximate the constraint by some weaker, anti-monotone constraint, called an *approximator*. If a cell fails the approximator, all super-cells must also fail the approximator and the given constraint. Clearly, cells that do satisfy the approximator may still fail the given constraint. Therefore, the effectiveness of this approach crucially depends on finding a strong approximator to minimize "false positives". We address this issue by dividing the search space into subspaces and identifying approximators individually in each subspace. As the "weaker, anti-monotone" condition is required only in a smaller subspace each time, a stronger approximator may be found in a subspace. By recursively applying this *Divide-and-Approximate* strategy, the approximators approach the given constraint.

Separate monotonicity. The Divide-and-Approximate strategy is applicable to all constraints $f(v)\theta\sigma$ with f containing two groups of aggregates denoted A^+ and A^- such that, as a cell v grows, f monotonically increases via the aggregates in A^+ and monotonically decreases via the aggregates in A^-. For such constraints, we can obtain an approximator using the maximum cell for A^+ if θ is $\geq$, or using the minimum cell for A^- if θ is $\leq$. This class is called *separable constraints*.

To apply Divide-and-Approximate strategy, several questions must be answered: What constraints are separable? How do we partition the search space and extract an approximator in a subspace? How do we prune unpromising spaces without enumerating the cells in them? Can Divide-and-Approximate strategy be implemented efficiently? We answer these questions in the rest of this paper.

2 Related Work

The iceberg-cube mining problem was considered in [3], but only for anti-monotone constraints. The idea of pushing a weaker constraint was previously suggested in [8], but the key question of finding such constraints, especially effective ones, was left unanswered there. In a recent work [6], the "top-k average" technique was proposed to push the minimum average constraint. The effectiveness of "top-k average" technique relies on a large minimum support k. Unlike [6], the work proposed here is applicable, with or without a minimum support, to a general class of constraints, rather than specific to a particular constraint.

This work is related to constrained association mining in transactional databases [8, 9]. [8] considers constraints of anti-monotonicity, monotonicity, and succinctness. We consider constraints without such properties. [8] uses "item-based" aggregates where the measure is associated with items (i.e., dimension values in our setting). In contrast, we adopt "tuple-based" aggregates, similar to SQL aggregates. This difference essentially affects the applicability of pushing techniques. For example, by adopting "item-based" aggregates, [9] is able to push $avg(v) \geq \sigma$ through the item order induced by the associated measure, but that technique is not applicable to "tuple-based" aggregates because no such order exists.

Some other constraints are pushed in [2, 10, 11], using constraint specific techniques, but they are quite different from SQL-like aggregate constraints considered here.

3 Iceberg-Cube Mining

3.1 The problem statement

The database is a relational table with one or more *dimensions* D_i and one or more *measures* M_i. A *cell* is a set of values d_i with at most one value from each dimension. We write a cell as $d_{i_1} \dots d_{i_k}$. A cell c corresponds to the GROUP BY partition, denoted $SAT(c)$, which contains all the tuples that contain all the values in c. c' is a *super-cell* of c, or c is a *sub-cell* of c', if c' contains all the values in c. As in SQL, $avg(c)$, $min(c)$, $max(c)$, $sum(c)$ compute the average, minimum, maximum, sum of some specified measure of the tuples in $SAT(c)$, $count(c)$ computes the number of tuples in $SAT(c)$. We consider three other aggregates (more can be added): $ssum(c)$ computes the square sum of the measure in $SAT(c)$, and $psum(c)$ and $nsum(c)$ compute the positive sum and unsigned negative sum of the measure in $SAT(c)$.

Definition 3.1 A *constraint* has the form $f(v)\theta\sigma$, where $f(v)$ is a function of cell-valued variable v defined by aggregates, arithmetic operators $+, -, \times, /$ and constants, θ is either $\geq$ or $\leq$, and σ is a real. A cell c *satisfies* a constraint $\mathcal{C}$ if $\mathcal{C}$ holds with v instantiated by cell c; otherwise, c *fails* a constraint $\mathcal{C}$. $CUBE(\mathcal{C})$ denotes the set of cells that satisfy $\mathcal{C}$. A constraint $\mathcal{C}$ is *weaker than* a constraint $\mathcal{C}'$ if $CUBE(\mathcal{C}') \subseteq CUBE(\mathcal{C})$.

Example 3.1 $count(v \wedge D_i = d_i)/count(v) \geq \sigma$, where D_i and d_i are fixed, specifies all *association rules* $v \rightarrow D_i = d_i$ above the minimum confidence σ [1]. $count(v \wedge D_i = d_i)/(count(v) - count(v \wedge D_i = d_i)) \geq \sigma$ specifies emerging patterns v in the two partitions specified by $D_i = d_i$ and $D_i \neq d_i$ [4]. We can specify the maximum variance constraint $var(v) \leq \sigma$ by

$$var(v) = \frac{\Sigma_{t\in SAT(v)}(M[t]-avg(v))^2}{count(v)},$$

where $M[t]$ denotes the measure value of tuple t. By rewriting and substituting $ssum(v)$ and $sum(v)$ for $\Sigma_{t\in SAT(v)}M[t]^2$ and $\Sigma_{t\in SAT(v)}M[t]$, we have

$$var(v) = \frac{ssum(v)-2sum(v)avg(v)+avg(v)^2count(v)}{count(v)}.$$

Definition 3.2 Given a database and a constraint $\mathcal{C}$, the *iceberg-cube mining* problem is to find $CUBE(\mathcal{C})$.

3.2 Monotonicity of constraints

We use "a-monotone" for "anti-monotone", "m-monotone" for "monotone", and "τ-monotone" for either, where τ is either a or m. This naming allows us to state results in a uniform way.

constraint	meaning
$count(v) \geq \sigma$	minimum count
$psum(v) \geq \sigma$	minimum positive sum
$nsum(v) \geq \sigma$	minimum negative sum
$ssum(v) \geq \sigma$	minimum square sum
$max(v) \geq \sigma$	minimum maximum
$min(v) \leq \sigma$	maximum minimum
$max(v) - min(v) \geq \sigma$	minimum diameter

Table 1. Some a-monotone constraints

Definition 3.3 A constraint $\mathcal{C}$ is *a-monotone* if whenever a cell is not in $CUBE(\mathcal{C})$, neither is any super-cell. A constraint $\mathcal{C}$ is *m-monotone* if whenever a cell is in $CUBE(\mathcal{C})$, so is every super-cell.

Table 1 lists some known a-monotone constraints. By replacing $\geq$ with $\leq$, or replacing $\leq$ with $\geq$, a-monotone constraints become m-monotone, and m-monotone constraints become a-monotone.

Definition 3.4 A function $x(y)$ is *a-monotone* wrt y if x decreases as y grows (for cell-valued y) or increases (for real-valued y). A function $x(y)$ is *m-monotone* wrt y if x increases as y grows (for cell-valued y) or increases (for real-valued y).

For example, $psum(v) - nsum(v)$ is m-monotone wrt $psum(v)$ and is a-monotone wrt $nsum(v)$, but is neither wrt v. Notice that the terms "a-monotone" and "m-monotone" are overloaded for both constraints and functions.

Observation 3.1 Let $\overline{a}$ denote m and $\overline{m}$ denote a. Let σ be a threshold value. (1) $f(v) \geq \sigma$ is τ-monotone if and only if $f(v)$ is τ-monotone wrt v. (2) $f(v) \leq \sigma$ is τ-monotone if and only if $f(v)$ is $\overline{\tau}$-monotone wrt v.

4 The New Strategy

4.1 The Divide-and-Approximate strategy

If a constraint is neither a-monotone nor m-monotone, it makes sense to approximate the constraint by pushing a weaker, a-monotone constraint, called an *approximator*: if a cell c fails the approximator, all super-cells of c must fail the a-monotone approximator, and thus, fail the (stronger) given constraint. However, a cell that satisfies the approximator may also fail the given constraint, i.e., a "false positive". If this happens often, the benefit of pushing the approximator diminishes. A key issue is to identify a strongest possible approximator to minimize "false positives". To address this issue, we *divide* the search space into subspaces and *approximate* the constraint independently in each subspace. Now since the "weaker, a-monotone" condition is required in a smaller subspace, a stronger approximator can be found for each subspace. As the space is divided recursively, such approximators approach the given constraint. This strategy is called *Divide-and-Approximate*.

Let us consider $sum(v) \geq \sigma$ (which is neither anti-monotone nor monotone) in the space $S = \{c \mid c \text{ is a sub-cell of } d_1 \ldots d_p\}$, where $d_1 \ldots d_p$ is a fixed cell and each d_i is a value from dimension D_i. First, we rewrite $sum(v) \geq \sigma$ into $psum(v) - nsum(v) \geq \sigma$. The first approximation is to ignore the "cost" $nsum$, i.e., consider $psum(v) \geq \sigma$. If the "cost" is high, many cells that have above-threshold "profit" $psum$ still do not have above-threshold "net profit" $psum(v) - nsum(v)$. A better attempt is to consider $psum(v) - nsum(d_1 \ldots d_p) \geq \sigma$, where $nsum(d_1 \ldots d_p)$ gives the minimum $nsum$ in S. This constraint may still be too weak if the minimum underestimates the "cost" too much. An even better attempt is to divide the space S into p subspaces S_i, $i = 1, \ldots, p$, where S_i contains all the sub-cells of $d_i \ldots d_p$, and use the (stronger) approximator $psum(v) - nsum(d_i \ldots d_p) \geq \sigma$ in each S_i.

Before discussing an actual implementation, the first question we need to answer is what constraints allow this above strategy and how approximators are extracted from such constraints.

4.2 Separate monotonicity

One way to obtain an approximator is to separate the aggregates in a constraint $f(v) \geq \sigma$ into two groups: as cell v grows, one group denoted A^+ monotonically increases the value of f, and the other group denoted A^- monotonically decreases the value of f. If this separation is possible, we can obtain an m-monotone approximator by instantiating v in A^- to the minimum cell, or obtain an a-monotone approximator by instantiating v in A^+ to the maximum cell. A similar consideration applies to a constraint $f(v) \leq \sigma$.

Example 4.1 Consider the constraint, $avg(v) \geq \sigma$, rewritten as

$$psum(v)/count1(v) - nsum(v)/count2(v) \geq \sigma.$$

We have renamed the two occurrences of $count(v)$. $avg(v)$ is a-monotone wrt $A^+ = \{nsum(v), count1(v)\}$ by holding $psum(v)$ and $count2(v)$ at constant, and is m-monotone wrt $A^- = \{psum(v), count2(v)\}$ by holding $nsum(v)$ and $count1(v)$ at constant. Moreover, as v grows, $avg(v)$ increases via A^+, by composing two a-monotone functions, i.e., f wrt A^+ and A^+ wrt v. Similarly, as v grows, $avg(v)$ decreases via A^-. Therefore, if we instantiate v in A^+ to the maximum cell $\overline{c}$, we have a-monotone approximator:

$$psum(v)/count1(\overline{c}) - nsum(\overline{c})/count2(v) \geq \sigma.$$

If we instantiate v in A^- to the minimum cell $\underline{c}$, we have m-monotone approximator:

$$psum(\underline{c})/count1(v) - nsum(v)/count2(\underline{c}) \geq \sigma.$$

In this example, all aggregates are either a-monotone or m-monotone, and all aggregates evaluate to a single sign. Imagine what would happen if $count1$ can be both positive and negative: we can no longer determine A^+ and A^- because the role of aggregates in increasing or decreasing the value of f now depend on the sign of $count1$.

Below, we show that all aggregates considered so far can be rewritten in terms of sign-preserved τ-monotone aggregates (τ is either a or m). An expression is *sign-preserved* if it never changes sign. For example, $psum$, $nsum$, $count$ are sign-preserved, but sum, avg, max and min are not. We have rewritten sum and avg in terms of sign-preserved τ-monotone aggregates, i.e., $psum - nsume$ and $(psum - nsume)/count$. Now, we rewrite max and min in terms of the following new sign-preserved τ-monotone aggregates:

- $P(v)$: return real 1 if some tuple in $SAT(v)$ has a non-negative measure value; 0 otherwise. $P(v)$ is a-monotone.
- $N(v)$: return real 1 if some tuple in $SAT(v)$ has a non-positive measure value (including 0); 0 otherwise. $N(v)$ is a-monotone.
- $pmax(v)$: return the maximum non-negative measure value in $SAT(v)$. If there is no such value, return 0. $pmax(v)$ is a-monotone.
- $pmin(v)$: return the minimum non-negative measure value in $SAT(v)$. If there is no such value, return $MAXVALUE$ (i.e., the maximum machine value). $pmin(v)$ is m-monotone.
- $nmax(v)$: return the maximum $|M|$ where M is a non-positive measure value in $SAT(v)$. If there is no such value, return 0. $nmax(v)$ is a-monotone.
- $nmin(v)$: return the minimum $|M|$ where M is a non-positive measure value in $SAT(v)$. If there is no such value, return $MAXVALUE$. $nmin(v)$ is m-monotone.

It is easy to see that

$$max = P \times pmax - (1 - P) \times nmin,$$
$$min = N \times nmax + (1 - N) \times pmin.$$

Up to this point, all SQL aggregates are defined by sign-preserved τ-monotone aggregates. A main result is stated below.

Theorem 4.1 Given any function f defined by sign-preserved τ-monotone aggregates, arithmetic operators and constants, there is a rewriting of f and a partition $S_1, \ldots, S_k$ of space such that every operand of $\times$ and $/$ in the rewritten f is sign-preserved in each subspace S_i, where $k \leq 2^p$ and p is the number of denominators in f that are not sign-preserved.

In Theorem 4.1, we say that the rewritten f is *($\times$,/)-sign-preserved* in S_i. This property ensures that the role of each aggregate in terms of increasing or decreasing the value of f is known, therefore, A^+ and A^- can be determined, in S_i. The procedure for obtaining the rewritten f and the partition in Theorem 4.1 is to eliminate $+$ and $-$ operators from the operands of $\times$ and $/$, by rewriting $(E_1+E_2)\times E$ into $E_1 \times E+E_2\times E$ and rewriting $(E_1+E_2)/E$ into $E_1/E+E_2/E$, and for each denominator E in f that is not sign-preserved in a subspace S_i created (initially the whole space), split S_i into two subspaces corresponding to the positive values and negative values of the denominator. Our implementation in Section 5 does not need to find the partition $S_1, \ldots, S_k$ in Theorem 4.1. The purpose of Theorem 4.1 is to provide the notion of ($\times$,/)-sign-preserved subspaces for talking about the monotonicity of $f(v)$ wrt building aggregates, as in the definition below.

Definition 4.1 Assume that f is ($\times$,/)-sign-preserved in S_i. Let A^+ and A^- be two groups of aggregates in $f(v)$ such that: if $agg(v)$ in A^+ is τ-monotone wrt v, f is τ-monotone wrt $agg(v)$ by holding all other aggregates at constant in S_i; if $agg(v)$ in A^- is τ-monotone wrt v, f is $\overline{\tau}$-monotone wrt $agg(v)$ by holding all other aggregates at constant in S_i. We write this as $f(A^+; A^-)$ in S_i, and we say that f or $f\theta\sigma$ is *separable* in S_i.

Condition 1 says that A^+ contains those aggregates $agg(v)$ that have the same monotonicity wrt v as f wrt $agg(v)$ in S_i. Therefore, by holding all other aggregates at constant, $f(v)$ is a function of v through composing two functions of the *same* monotonicity, thus, is m-monotone wrt v. Condition 2 is a statement about composing two functions of the *complement* monotonicity.

Let A^+/c denote A^+ with all occurrences of v instantiated to c. Similarly, A^-/c. Let $\overline{c}$ denote the maximum cell and $\underline{c}$ denote the minimum cell in S_i.

Corollary 4.1 Assume $f(A^+;A^-)$ in S_i. Let σ be a threshold value. (1) $f(A^+;A^-/\underline{c}) \geq \sigma$ and $f(A^+/\underline{c};A^-) \leq \sigma$ are m-monotone approximators in S_i. (2) $f(A^+/\overline{c};A^-) \geq \sigma$ and $f(A^+;A^-/\overline{c}) \leq \sigma$ are a-monotone approximators in S_i.

Remarks: The classic m-monotone constraints are a special case of Corollary 4.1(1) where $A^-/\underline{c}$ or $A^+/\underline{c}$ is empty and every denominator in f is sign-preserved, therefore, S_i is the whole space. The classic a-monotone constraints are a special case of Corollary 4.1(2) where $A^+/\overline{c}$ or $A^-/\overline{c}$ is empty and every denominator in f is sign-preserved.

4.3 Arithmetic closure of separability

We show that any arithmetic operation preserves the separability of constraints in Definition 4.1. Therefore, by repeatedly applying this property, we can compute A^+ and A^- for a separable constraint.

Theorem 4.2 Assume that f is ($\times$,/)-sign-preserved in S_i. Suppose that $f_1(A_1^+;A_1^-)$ and $f_2(A_2^+;A_2^-)$ are two sub-expressions of f. $(A^+;A^-)$ for a larger sub-expression of f built by f_1 and f_2 is computed as follows:

1. $-f_1$: $A^+ = A_1^-$ and $A^- = A_1^+$.
2. f_1+f_2: $A^+ = A_1^+ \cup A_2^+$ and $A^- = A_1^- \cup A_2^-$.
3. f_1-f_2: $A^+ = A_1^+ \cup A_2^-$ and $A^- = A_1^- \cup A_2^+$.
4. $f_1 \times f_2$: If the sign of (f_1,f_2) in S_i is $(+,+)$, $A^+ = A_1^+ \cup A_2^+$ and $A^- = A_1^- \cup A_2^-$. If the sign is $(-,-)$, we consider $(-f_1)\times(-f_2)$, which reduces to case 1 and the $(+,+)$ sign. Similarly, if the sign is $(+,-)$, we consider $f_1 \times (-f_2)$, and if the sign is $(-,+)$, we consider $(-f_1)\times f_2$.
5. f_1/f_2: If the sign of (f_1,f_2) in S_i $(+,+)$, $A^+ = A_1^+ \cup A_2^-$ and $A^- = A_1^- \cup A_2^+$. Similar to case 4, other signs of (f_1,f_2) can be reduced to case 1 and the $(+,+)$ sign.

Notice that (f_1,f_2) must have one of the above four signs in S_i because f is ($\times$,/)-sign-preserved in S_i. The following corollary follows from Theorem 4.1 and Theorem 4.2.

Corollary 4.2 All functions constructed by SQL aggregates, arithmetic operators and constants are separable (wrt a partition of space).

For example, all constraints but 8, 9 and 11 in Table 2 are separable in the whole space. Constraints 8, 9 and 11 are separable in each of S_1 and S_2, where S_1 and S_2 are subspaces corresponding to positive values and negative values of the denominator of $/$.

Example 4.2 Let us extract $(A^+;A^-)$ for $var(v)$. See Example 3.1 for the definition of $var(v)$. We rewrite $var(v)$, as in Theorem 4.1, so that all operands of $\times$ and $/$ are sign-preserved:

Separable constraints
1. $sum(v)\theta\sigma$
2. $avg(v)\theta\sigma$
3. $var(v)\theta\sigma$
4. $count(v \wedge D_i = d_i) - count(v \wedge D_i = d'_i)\theta\sigma$
5. $count(v \wedge D_i = d_i)/count(v)\theta\sigma$
6. $count(v \wedge D_i = d_i)/count(v \wedge D_i = d'_i)\theta\sigma$
7. $sum(v \wedge D_i = d_i) - sum(v \wedge D_i = d'_i)\theta\sigma$
8. $sum(v \wedge D_i = d_i)/sum(v \wedge D_i = d'_i)\theta\sigma$
9. $avg(v)/max(v)\theta\sigma$
10. $max(v) - avg(v)\theta\sigma$
11. $avg(v)/min(v)\theta\sigma$
12. $min(v) - avg(v)\theta\sigma$

Table 2. Some separable constraints (σ, D_i, d_i, d'_i are fixed)

	$(A^+; A^-)$
$ssum(v)/count1(v)-$	$(\{count1\}; \{ssum\})$
$psum1(v)^2/count2(v)^2-$	$(\{count2\}; \{psum1\})$
$nsum1(v)^2/count3(v)^2+$	$(\{count3\}; \{nsum1\})$
$2psum2(v)nsum2(v)/count4(v)^2$	$(\{count4\}; \{psum2, nsum2\})$

We have renamed occurrences of aggregates. Since all denominators of / are sign-preserved, there is no need for space partitioning, and $var(v)$ is ($\times$,/)-sign-preserved in the whole space. The $(A^+; A^-)$ for each term is given on the right. Applying cases 2 and 3 in Theorem 4.2 several times, we have

$$A^+ = \{count1, psum1, nsum1, count4\},$$
$$A^- = \{ssum, count2, count3, psum2, nsum2\}$$

5 Implementation

We present an efficient implementation, called D&A, of Divide-and-Approximate strategy for a subclass of separable constraints. A constraint is *denominator-monotone* if every denominator in the constraint is either a-monotone or m-monotone. For example, all constraints but No. 8 in Table 2 are denominator-monotone. We consider a constraint $\mathcal{C}$ that is separable and denominator-monotone. To bring out the main ideas, we first ignore the minimum support and we consider only a-monotone approximators.

5.1 The search strategy

We use the *lexicographic tree* to enumerate GROUP BY lists. A node exists in the lexicographic tree corresponding to a GROUP BY list $D_1 \ldots D_k$, in some lexicographic order of dimensions D_i. This node represents all the cells on dimensions $D_1 \ldots D_k$. The root corresponds to the *null* GROUP BY list and has one child for each dimension D_i, in the lexicographic order. If a node $u = D_1 \ldots D_{k-1}D_k$ has q siblings on its right in order:

$$D_1 \ldots D_{k-1}D_{k+1}, \ldots, D_1 \ldots D_{k-1}D_{k+q},$$

u has q child nodes in that order

$$D_1 \ldots D_kD_{k+1}, \ldots, D_1 \ldots D_kD_{k+q}.$$

$tree(u)$ denotes the subtree rooted at node u. $tail(u)$ denotes the set of all dimensions appearing in $tree(u)$.

If we ignore the constraint $\mathcal{C}$, our algorithm follows the BottomUpCUBE computation (BUC for short) in [3] for enumerating cells. BUC partitions the database in the depth-first order of the lexicographic tree, as given by the sequence numbers in Figure 1. BUC first produces the empty cell at the root. Next, it partitions on node A, producing partitions a_1 to a_i. The a_1 partition is aggregated and produces a cell for a_1. Next, it partitions the a_1 partition on dimension B, producing partitions a_1b_1 to a_1b_j at node AB. It aggregates the a_1b_1 partition and writes a cell for a_1b_1. It then processes $a_1b_1c_1$ partition at ABC, $a_1b_1c_1d_1$ partition at node $ABCD$, and $a_1b_1c_1d_1e_1$ at node $ABCDE$ in that order. Once the $a_1b_1c_1d_1$ partition is completely processed, BUC proceeds to the $a_1b_1c_1d_2$ partition, and so on. BUC stops partitioning if a partition does not satisfy a minimum support.

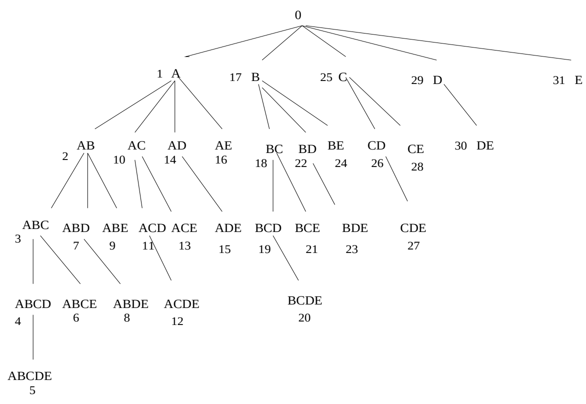


Figure 1. The lexicographic tree for A, B, C, D, E

5.2 The pruning strategy

Now, let us consider the constraint $\mathcal{C}$. First, we associate a bit-vector $b_1 \ldots b_k$ with each cell, where b_i corresponds to a denominator in $\mathcal{C}$ that is not sign-preserved, and $b_i = 1$ if and only if the denominator evaluates to a negative value on the cell. Therefore, in the subspace consisting of the cells that have the same bit-vector, all denominators are sign-preserved, and A^+ and A^- can be extracted following Theorem 4.2. We make use of the following property of denominator-monotone constraints: if a sub-cell c and a super-cell c' have the same bit-vector, all cells c'' such that $c \subseteq c'' \subseteq c'$ have the same bit-vector as c. The proof is straightforward.

Suppose that, in the above BUC-like computation, we reach a leaf node u_0 and find some cell p at u_0 fails $\mathcal{C}$. Based on the bit-vector associated with p, we can extract A^+ and A^- for $\mathcal{C}$ following Theorem 4.2. We define a new constraint $\mathcal{C}_p$ as

$$f(A^+/p; A^-)\theta\sigma \text{ if } \theta \text{ is } \geq, \text{ or } f(A^+; A^-/p)\theta\sigma \text{ if } \theta \text{ is } \leq.$$

From Corollary 4.1, $\mathcal{C}_p$ is an a-monotone approximator in the subspace in which p is the maximum cell and all cells have the same bit-vector as p.

To exploit this approximator, consider an ancestor u_k of u_0 such that u_0 is the left-most leaf in $tree(u_k)$. Let $tree(u_k, p)$ denote the set of sub-cells of p corresponding to the nodes in $tree(u_k)$:

$$tree(u_k, p) = \{p[u] \mid u \text{ is a node in } tree(u_k)\},$$

where $p[u]$ denotes p projected onto the dimensions at a node u. p is the maximum cell and $p[u_k]$ is the minimum cell in $tree(u_k, p)$. Suppose that $p[u_k]$ has the same bit-vector as p. The denominator-monotonicity of $\mathcal{C}$ implies that all the cells in $tree(u_k, p)$ have the same bit-vector as p. Therefore, $\mathcal{C}_p$ is an a-monotone approximator in $tree(u_k, p)$: if $p[u_k]$ fails $\mathcal{C}_p$, so do all cells in $tree(u_k, p)$. Notice that this does not authorize the pruning of the *whole subtree* below the partition $p[u_k]$ because there may exist some sub-partition p' at some node u in $tree(u_k)$ such that $p'[u_k] = p[u_k]$ but $p'[u] \neq p[u]$.

The pruning implementation. To leverage the above pruning, on backtracking to an ancestor node u_k from a leaf node u_0 in the BUC computation, we check if (i) u_0 is on the left-most path in $tree(u_k)$, (ii) $p[u_k]$ has the same bit-vector as $p[u_0]$, and (iii) $p[u_k]$ fails $\mathcal{C}_p$. If all conditions hold, for every unexplored child w_j of u_k, we prune all the tuples that match p on $tail(w_j)$ because such tuples generate only cells in $tree(u_k, p)$, which we know fail $\mathcal{C}_p$. The pruning of these tuples may affect the aggregates of the cells in $tree(u_k, p)$, but *only* such cells. For each partition produced (by unpruned tuples), we still need to test if it satisfies the constraint $\mathcal{C}$. However, for those partitions (or cells) in $tree(u_k, p)$, we just mark them as failed (as they are indeed) without testing the constraint.

Example 5.1 Suppose that we reach a cell p at the leaf node $u_0 = ABCDE$ in Figure 1 and suppose that p fails $\mathcal{C}$. On backtracking to node $ABCD$ and then to node ABC, in the BUC computation, suppose that $p[ABC]$ has the same bit-vector as p and fails $\mathcal{C}_p$. Before producing the partitions at (the only) child $ABCE$ like in BUC, we prune all the tuples t such that $t[ABCE] = p[ABCE]$, by not partitioning them, because such tuples generate only cells in $tree(ABC, p)$. A pruned tuple will not be examined in a later partitioning. Next, on backtracking to node AB, if $p[AB]$ also has the same bit-vector as p and fails $\mathcal{C}_p$, we prune all tuples t such that $t[ABDE] = p[ABDE]$ from $tree(ABD)$, where $ABDE$ is the set of dimensions in $tree(ABD)$, and prune all tuples t such that $t[ABE] = p[ABE]$ from $tree(ABE)$.

This example conveys several interesting points. First, by pruning the tuples that generate only failed cells, we deal with a much smaller set of tuples in a subtree and stop going down a subtree earlier if there is no tuple left. Second, if we can identify the highest node u_k such that $p[u_k]$ has the same bit-vector as p and fails $\mathcal{C}_p$, we can prune tuples from largest subtrees. Third, this strategy can be applied recursively in any subtree at any level. For example, in subtree $tree(B)$, an a-monotone approximator $\mathcal{C}_p$ can be extracted from a failed cell p at leaf node $BCDE$ and then used in the tree. Let us formalize these ideas.

Definition 5.1 Consider a leaf node u_0, a cell p at u_0, and the left-most path $u_k, \ldots, u_0$ in $tree(u_k)$, $k \geq 0$. p is a *pruning anchor* wrt (u_k, u_0) if (i) p fails $\mathcal{C}$, (ii) $p[u_k]$ has the same bit-vector as p and fails $\mathcal{C}_p$, and (iii) u_k is the highest possible node satisfying (ii).

If p is a pruning anchor wrt (u_k, u_0), we say that p is *anchored* at u_k, $tree(u_k, p)$ is the *pruning scope* of p, and the tuples in the partition for p are *generating tuples* of p.

Theorem 5.1 For a pruning anchor p, all cells in the pruning scope of p fail $\mathcal{C}_p$ and $\mathcal{C}$.

Theorem 5.1 and our pruning implementation ensure that only failed cells are short of the testing against the constraint $\mathcal{C}$.

6 Interaction with Minimum Support

If the user also specifies a minimum support, in addition to the constraint $\mathcal{C}$, it would be more effective to push both constraints. However, there is a non-trivial interaction between the two because now a pruning anchor is required to be frequent (i.e., satisfying the minimum support). Consider Figure 1. Suppose that on the left-most path in $tree(A)$, a cell $abcd$ is frequent but cell $abcde$ is not. We should stop at $abcd$ because further partitioning generates only infrequent cells. Now, even if $abcd$ is anchored at node A, we cannot prune the sub-cells of $abcd$ in $tree(A, abcd)$

because non-sub-cells like ae, abe, ace, ade in the tree "depend on" such sub-cells. The fact that E occurs in every leaf node makes it impossible to prune any sub-cells of $abcd$ without affecting those cells involving E.

The rollback tree. We propose a new enumeration tree, called the *rollback tree*, to solve this problem. At each non-root node u in the rollback tree, we generate the *first* child of u using the *last* sibling of u, and generate the ith child node ($i > 1$) using the $(i-1)$th sibling of u. Notice that the rollback tree does not use a lexicographic order of dimensions. Similar to Section 5, let $RBtree(u)$ denote the subtree at a node u, and $RBtree(u, p)$ denote the set of projected cells of p onto the nodes in $RBtree(u)$.

The rollback tree in Figure 2 explains the rationale of "rolling back" the last sibling to generate the first child. Consider node $u = A$. The first child AB of u is generated using the last sibling B of u; the second child AE of u is generated using the first sibling E of u; and so on. The child nodes of other nodes are generated similarly. This tree generates exactly the same left-most path as the lexicographic tree with A, B, C, D, E at the first level in Figure 1, with the important difference that the last dimension E on the left-most path, E, does *not* occur in several subtrees: $RBtree(AC)$, $RBtree(AD)$, $RBtree(ABD)$, $RBtree(B)$, $RBtree(C)$, and $RBtree(D)$. Therefore, a pruning anchor $abcd$ at $ABCD$ can be used to prune its sub-cells in these subtrees, as in Section 5.

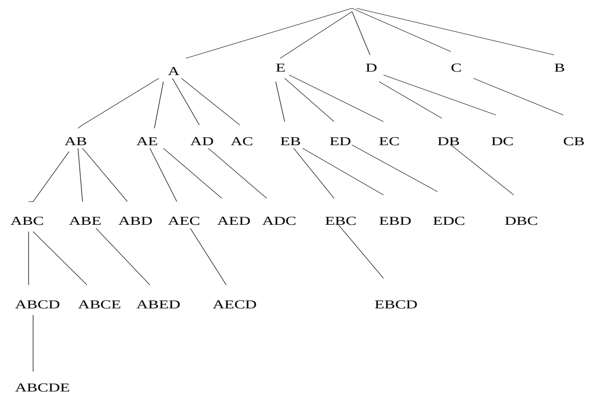


Figure 2. The rollback tree for A, B, C, D, E

We now modify the notions of pruning anchors and pruning scope to adapt to the presence of minimum support. As in Section 5, $\mathcal{C}_p$ denotes the a-monotone approximator defined for p.

Definition 6.1 Consider a (possibly non-leaf) node u_0, a cell p at u_0, and the left-most path $u_k, \ldots, u_0$ in $RBtree(u_k)$, $k \geq 0$. p is a *pruning anchor* wrt (u_k, u_0) if (i) p satisfies the minimum support and fails $\mathcal{C}$, but no super-cell of p on the left-most path in $RBtree(u_0)$ does, (ii) $p[u_k]$ has the same bit-vector as p and fails $\mathcal{C}_p$, (iii) u_k is the highest possible node satisfying (ii).

If p is a pruning anchor wrt (u_k, u_0), the *pruning scope* of p consists of all subtrees $RBtree(w^i, p)$, for $k \geq i \geq 1$, where w^i are the last $i-1$ child nodes of u_i. Essentially, w^i are those nodes such that subtrees $RBtree(w^i)$ contain only the dimensions appearing at node u_0. With these modifications, Theorem 5.1 remains valid in the presence of minimum support.

Example 6.1 Consider the rollback tree in Figure 2. Suppose that $abcd$ is a pruning anchor wrt $(A, ABCD)$, where $u_3 = A, u_2 = AB, u_1 = ABC, u_0 = ABCD$. The pruning scope of $abcd$ consists of the following subtrees: $RBtree(AD, abcd)$ and $RBtree(AC, abcd)$ where AD and AC are the last 2 child nodes of u_3, and $RBtree(ABD, abcd)$ where ABD is the last child node of u_2. If ebc is a pruning anchor wrt (E, EBC), where $u_2 = E, u_1 = EB, u_0 = EBC$, the pruning scope of ebc is $RBtree(EC, ebc)$ where EC is the last child node of u_2.

7 Experiments

This section evaluates the performance of D&A. We choose constraints of the prototypical form $agg_1(x) - agg_2(x) \geq \sigma$ for our case study, where agg_1 and agg_2 are non-negative a-monotone aggregates. We compare D&A with BUC and BUC+. BUC pushes only the minimum support constraint. BUC+ also pushes the positive term constraint $agg_1(x) \geq \sigma$, in addition to the minimum support constraint. We consider two criteria of performance, *execution time* and *tuple examination*. The tuple examination is the number of examinations of tuples. An effective pruning should reduce the tuple examination. All algorithms were written in C and run on a PC with Windows 2000, CPU clock of 1G and memory of 512M. We conducted two sets of experiments, one on synthetic datasets to cover a wide range of data characteristics, one on a real life dataset.

7.1 Experiments on synthetic datasets

We use the minimum sum constraint, $psum(x) - nsum(x) \geq \sigma$, in this experiment. Our objective is to study the effectiveness of pushing the negative term $nsum(x)$ for various data characteristics.

Datasets. We generate the synthetic dataset by iteratively adding groups of new tuples following the parameters in Figure 3. In one iteration, we add a group of $r = rand() \times \beta$ new tuples $t_1, \ldots, t_r$ that repeat the values on d randomly determined dimensions. $rand()$ generates a number uniformly distributed in range $[0, 1]$. d follows

Parameter	Meaning
n	number of tuples
m	number of dimensions
$card[i]$	cardinality of the ith dimension
$[0, Mmax]$	normally distributed positive measure
$[-Mmin, 0]$	normally distributed negative measure
α	splitting factor of measure
β	repeat factor
γ	Poisson mean of repeat dimension #

Figure 3. The parameters of the data generator

the Poisson distribution of mean γ [1]. γ and β dictate the count of frequent cells. To simulate the sharing of values between groups, a fraction, 0.5 in our experiments, of the d repeat dimensions takes values from those of the previous group. The measure of tuples follows two normal distributions, one for positive measure with the range $[0, Mmax]$, and one for negative measure with the range $[-Mmin, 0]$. For each group, we toss a $\alpha/(1-\alpha)$-weighed coin to choose one of these distributions, where α is in the range [0,1]. The measures for the tuples $t_1, \ldots, t_r$ in one group are then generated following the chosen normal distribution.

If parameters are not varied in an experiment, the following default values are used: $n = 100K$, $m = 15$, $card[i] = 10$, $Mmax = Mmin = 10$, $\alpha = 0.5$, $\beta = 1000$, $\gamma = 10$, $minsup = 0.5\%$, and $\sigma = 300$. The effect of varying $Mmax$ and $Mmin$ can be simulated by varying σ.

Varying constraint strength. Figure 4(a,a') plots the execution time (on the left) and tuple examination (on the right) for varied minimum support. As the minimum support is reduced, BUC slows down quickly whereas BUC+ improves over BUC by picking up the pruning from the positive term constraint. D&A improves even further by pushing stronger approximators. A smaller minimum support has mixed effects on D&A: it generates weaker approximators (negative) that cover larger subspaces (positive). Figure 5(b,b') plots the curves for varied threshold σ. As expected, D&A and BUC+ benefit from a larger σ, but D&A takes a lead because of the negative term pruning. Figure 4(c,c') plots the curves in the absence of minimum support, where BUC was omitted because it takes more than 5,000 seconds!

Varying data characteristics. Figure 4(d,d') and (e,e') show the results for varied repeat factor β and Poisson mean γ. For a more "dense" dataset, i.e., a larger Poisson mean or a larger repeat factor, D&A shows a more substantial speed up over the other two algorithms. For the default threshold $\sigma = 300$, BUC+ is almost identical to BUC because the positive term constraint is too weak. But for a larger σ, BUC+ can do better than BUC, as shown in (b,b').

[1]We didn't use uniform distribution because it generates the data distribution that too much depends on the initialization of the random seed.

Scalability. In Figure 5(f), we vary the number of dimensions m from 15 to 21 and keep the Poison mean γ at $2/3$ of m and the other parameters at the default setting. In Figure 5(g), we vary the database size n from 200K to 1,000K while keeping the other parameters at the default setting. As the dimensionality or database size increases, D&A shows a better scalability.

7.2 Experiments on real life datasets

Dataset. The second set of experiments was conducted on a real life dataset. Most real life datasets used by data mining research are for the classification problem, and have a very small size that is not representative of the iceberg-cube mining applications. After examination of various alternatives, we choose the KDD-CUP-98 dataset [7] (the learning set only). This dataset was collected from the result of 1997 Paralyzed Veterans of America fund raising mailing campaign. We identify two columns as our measures:

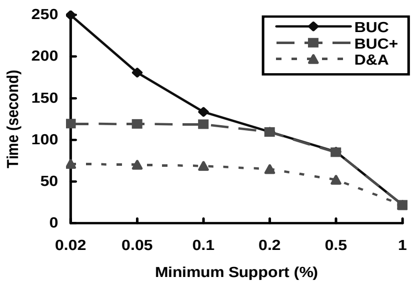
- 97NK: the donation amount in 1997. There are 4,843 nonzero tuples with maximum of \$200, minimum of \$1 and mean of \$15.62.
- 95NK: the donation amount in 1995. There are 23,317 nonzero tuples, with maximum of \$200, minimum of \$1 and mean of \$13.25.

We define the constraint to be $psum_1(x) - psum_2(x) \geq \sigma$, where $psum_1$ denotes the sum of the 97NK measure and $psum_2$ denotes the sum of the 95NK measure. This constraint specifies all cells (i.e., donor's characteristics) that improve the gift amount by at least σ.

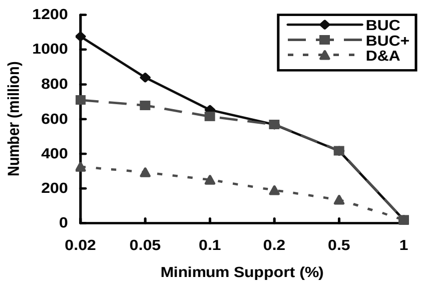
We made two changes to the dataset. First, we remove all tuples in which both 97NK and 95NK measures are zero because they do not affect our constraint. The number of remaining tuples is 26,600. Second, we selected 16 likely relevant dimensions. The original dataset has 481 dimensions, most of which are not related to the donation amount. The 16 dimensions we selected are:

```
RECINHSE(2):       In house file flag
RECP3(2):          P3 file flag
RECPGVG(2):        Planned giving file flag
RECSWEEP(2):       Sweepstakes file flag
MDMAUD(5,4,5,2):   The major donor matrix
                   code
DOMAIN(6,5):       Domain/Cluster code.
CLUSTER(54):       Code indicating which
                   cluster group the donor
                   falls into
HOMEOWNR(3):       Home owner flag
NUMCHLD(8):        Number oF children
```

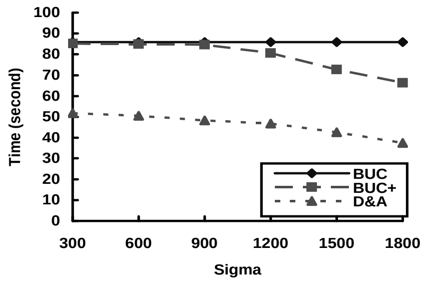
a



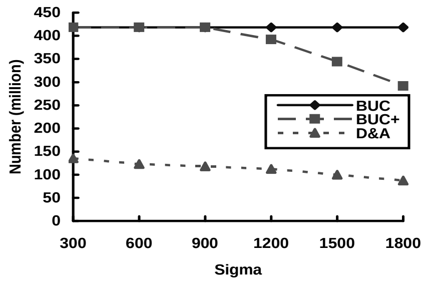
a'



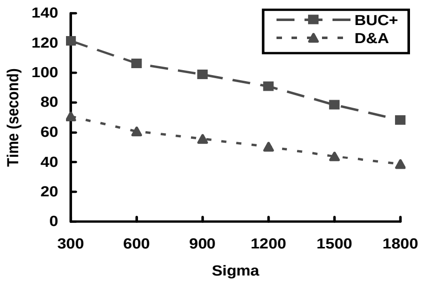
b



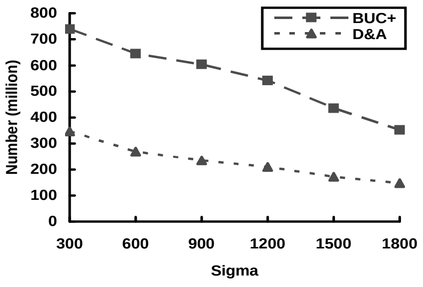
b'



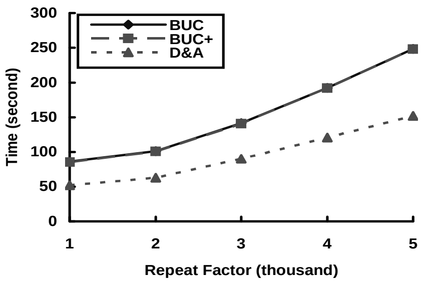
c



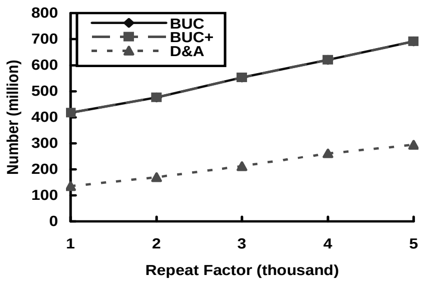
c'



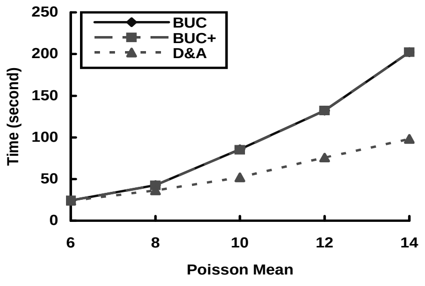
d



d'



e



e'

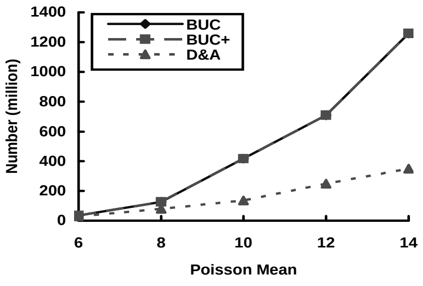
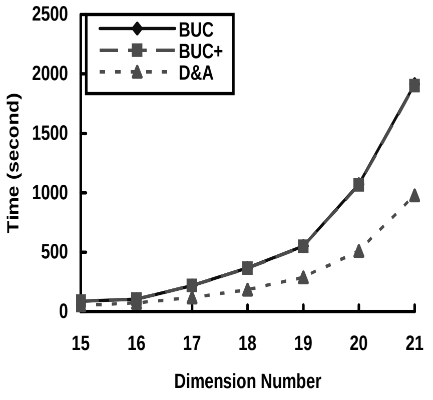


Figure 4. Experiments on synthetic datasets

f



g

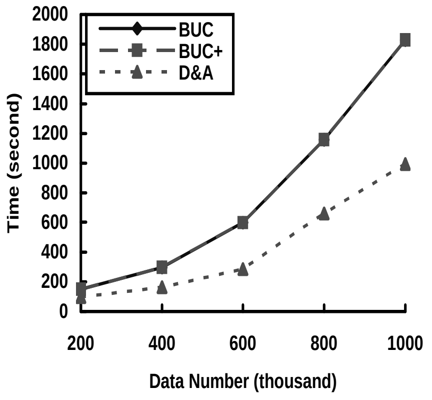
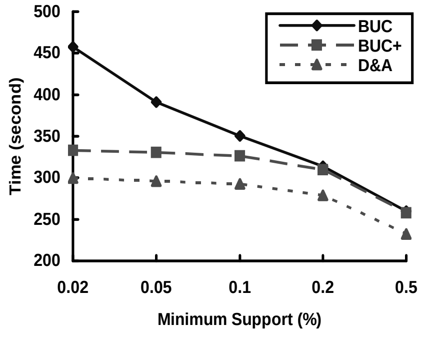
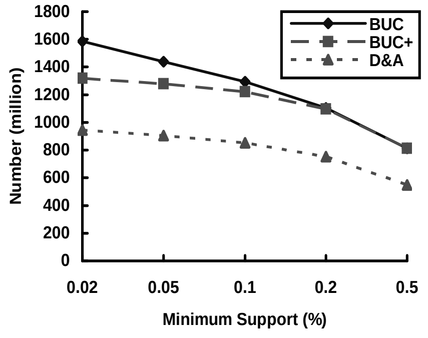


Figure 5. Scalability experiment on synthetic datasets

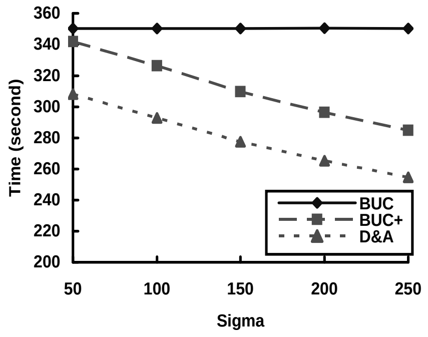
a



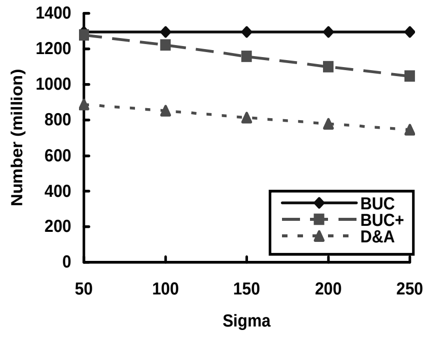
a'



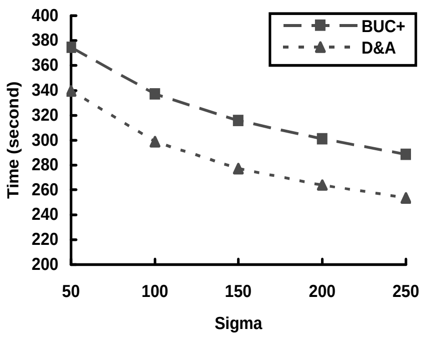
b



b'



c



c'

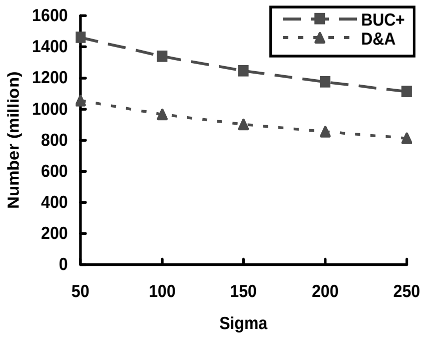


Figure 6. Experiments on KDD-CUP-98 dataset

```
INCOME(8):        Household income
GENDER(7):        Gender
WEALTH1(11):      Wealth rating
```

The cardinality of each dimension is given in (). (Indeed, there are 7 values for GENDER in the data.) MDMAUD and DOMAIN have two or more sub-dimensions, each being treated as a new dimension in our framework.

Varying constraint strength. Figure 6(a,a') plots the execution time and tuple examination for varied minimum support, with the threshold σ fixed at 100. Figure 6(b,b') plots the curves for varied threshold σ, with the minimum support fixed at 0.1%. Figure 6(c,c') plots the curves for varied threshold σ in the absence of minimum support. Again, BUC is omitted here because it takes substantially longer time (about 700 seconds).

Compared to the synthetic dataset, the improvement of D&A over BUC+ is less on this dataset. A close analysis of the data explains why. As described earlier, only 4,843 of the 26,600 tuples have nonzero 97NK donation. In this case, $psum_1$ tends to be low and the constraint $psum_1(x) \geq \sigma$ used by BUC+ is somehow sufficient to prune a large portion of space. A similar behavior is expected if only a few tuples have had nonzero 95NK donation, in which case the pruning role of $psum_2(x)$ used by D&A would diminish. The data characteristics that favors D&A over BUC+ is when many cells have both a large $psum_1$ and a large $psum_2$. In this case, the negative term $psum_2$ could prune many "false positives" that are able to pass $psum_1 \geq \sigma$.

In summary, D&A is better than BUC+ in all cases tested, which is better than BUC, especially for a weak minimum support constraint. D&A is never worse than BUC+ because the case where the negative term becomes less effective is also the case where the overhead of D&A diminishes. These are the case when very few pruning anchors are generated or when pruning anchors are not pushed up high in the search tree; in either cases, D&A incurs little overhead. In conclusion, D&A takes the new pruning opportunity of Divide-and-Approximate strategy when there is any, at an overhead justified by the benefit, and degenerates somehow to BUC+ when there is no such opportunity.

8 Conclusion

Previous works on constrained data mining have mainly centered around identifying "well-behaved" constraints with respect to constraint pushing. In this paper, we considered (1) general aggregate constraints, rather than only "well-behaved" constraints, (2) SQL-like tuple-based aggregates, rather than item-based aggregates, and (3) constraint independent techniques, rather than constraint specific techniques. We proposed a new pushing strategy called *Divide-and-Approximate*. This strategy combines two interesting ideas, "divide-and-conquer" and "approximate push", to produce strongest pushable constraints, which is really the central issue of the approach. We showed that many previously non-pushable, but useful, constraints are pushable by Divide-and-Approximate strategy. We implemented and evaluated this strategy on both synthetic and real life datasets. We believe that the idea of Divide-and-Approximate contributes a novel piece to constrained data mining.

References

[1] R. Agrawal, T. Imilienski, and A. Swami. Mining association rules between sets of items in large datasets. In *SIGMOD*, pages 207–216, 1993.

[2] R. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large dense databases. In *ICDE*, 1999.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD*, pages 359–370, 1999.

[4] G. Dong and J. Li. Efficient mining of emerging patterns: discovering trends and differen ces. In *SIGKDD*, pages 43–52, 1999.

[5] M. Fang, N. Shivakumar, H. Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *VLDB*, pages 299–310, 1998.

[6] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD*, 2001.

[7] KDD98. The kdd-cup-98 dataset. In *http://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html*. KDD, August 1998.

[8] R. Ng, L. V. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associatio ns rules. In *SIGMOD*, pages 13–24, 1998.

[9] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *ICDE*, 2001.

[10] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *KDD*, pages 67–73, 1997.

[11] K. Wang, Y. He, and J. Han. Pushing support constraints into frequent itemset mining. In *VLDB*, 2000.