Divide-and-Approximate: A Novel Constraint Push Strategy for Iceberg Cube Mining

Ke Wang, Yuelong Jiang, Jeffrey Xu Yu, Guozhu Dong, Jiawei Han

The work was supported in part by the Natural Sciences and Engineering Research Council of Canada, Networks of Centres of Excellence/Institute for Robotic and Intelligent Systems, and the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK4229/01E)

Ke Wang (wangk@cs.sfu.ca) and Yuelong Jiang (yjiang@cs.sfu.ca) are associated with Simon Fraser University. Jeffrey Xu Yu (yu@se.cuhk.edu.hk) is associated with the Chinese University of Hong Kong. Guozhu Dong (gdong@cs.wright.edu) is associated with Wright State University. Jiawei Han (hanj@cs.uiuc.edu) is associated with University of Illinois at Urbana-Champaign.

Abstract

The *iceberg cube mining* computes all cells v, corresponding to GROUP BY partitions, that satisfy a given constraint on aggregated behaviors of the tuples in a GROUP BY partition. The number of cells often is so large that the result cannot be realistically searched without pushing the constraint into the search. Previous works have pushed *anti-monotone* and *monotone* constraints. However, many useful constraints are neither anti-monotone nor monotone. We consider a general class of *aggregate constraints* of the form $f(v)\theta\sigma$, where f is an arithmetic function of SQL-like aggregates and θ is one of $<, \leq, \geq, >$. We propose a novel pushing technique, called *Divide-and-Approximate*, to push such constraints. The idea is to recursively *divide* the search space and *approximate* the given constraint using anti-monotone or monotone constraints in subspaces. This technique applies to a class called separable constraints, which properly contains all constraints built by an arithmetic function f of all SQL aggregates.

Index Terms

Aggregate constraint, constrained data mining, data cube, iceberg cube mining, iceberg query.

I. INTRODUCTION

Decision support systems, which rapidly gain competitive advantage for businesses, make heavy use of aggregations for identifying trends. The *iceberg query*, introduced in [8], 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 $R(target1, \cdots, targetk, rest)$ and a threshold T is as follows:

```
SELECT target1, ..., targetk, count(rest)
FROM R
WHERE ...
GROUP BY target1, ..., targetk
```

HAVING count(rest)≥T

This query partitions the tuples according to the GROUP BY list and produces one row for each partition with $count(rest)$ above the threshold T. In *iceberg cube mining*, the user specifies a constraint in the HAVING clause, but not the GROUP BY list, and wants to find the result for *all* GROUP BY lists. A *cell* specifies one GROUP BY partition. On a relation R(Product, Store, Year, rest), for example, the cell $\{Toyota, Vancouver\}$ specifies a partition for the GROUP BY list "Product, Store". $\{Toyota, Vancouver, 2000\}$ and $\{Toyota\}$ are a s*uper-cell* and *sub-cell* of $\{Toyota, Vancouver\}$, respectively. Iceberg cube mining aims to compute all the cells for the eight GROUP BY lists over Product, Store, Year, returning those satisfying the constraint in the HAVING clause.

Performing one iceberg query per GROUP BY list does not share the work in different queries. Computing the full cube then discarding unsatisfying cells suffers from the fact that the full cube is too large to be realistically computed. Materializing "views" for efficient computation is useful only if all the constraints are known in advance. A promising approach is "pushing" a given constraint so that only likely satisfying cells are computed. Previous works have pushed *anti-monotone* constraints [5], [2] and *monotone* constraints [13]. In an anti-monotone constraint, if a cell fails the constraint, so does every super-cell; in a monotone constraint, if a cell satisfies the constraint, so does every super-cell. These properties provide a natural pruning opportunity.

However, anti-monotonicity or monotonicity like these are undesirable for two reasons. On one hand, anti-monotonicity and monotonicity are too loose as a pruning strategy. Both properties impose an exponential lower bound on the result size because all super-cells of a failed or satisfying cell also fail or satisfy. A result of such size is neither efficient to compute nor easy to be comprehended by for a human user. On the other hand, both properties are too restricted

as an interestingness criterion. For example, $sum(v) \geq \sigma$, $avg(v) \geq \sigma$ and $var(v) \leq \sigma$ are neither anti-monotone nor monotone, but are useful for extracting patterns capturing minimum (average) profit with a small variance.

We consider the problem of pushing *aggregate constraints* of the form $f(v)\theta\sigma$ in iceberg cube mining. f is an arithmetic function of SQL-like aggregates, θ is a comparison operator, σ is a threshold, and v is a cell-valued variable. As we will show, $var(v) \leq \sigma$ is in this form, where $var(v)$ computes the variance of the measure for the tuples that match the cell v. Pushing an aggregate constraint presents a significant challenge because even if a cell fails or satisfies the constraint, its super-cells still need to be examined. We like to answer two questions. First, if a constraint $f(v)\theta\sigma$ is not anti-monotone or monotone, can it be pushed into iceberg cube mining? Second, is there a principled method that is independent of the specific form of f? This independence is essential because the user-specified f is unknown in advance. Two thoughts underpin our study.

Divide-and-Approximate. If the given constraint $\mathcal{C}$ is neither anti-monotone nor monotone, we can "approximate" it by some weaker or stronger constraint $\mathcal{C}'$ that has such monotonicities. For example, we can approximate $\mathcal{C}$ by a weaker anti-monotone constraint $\mathcal{C}'$: if a cell fails $\mathcal{C}'$, all its super-cells fail $\mathcal{C}'$, therefore, fail the stronger $\mathcal{C}$. Note that cells satisfying $\mathcal{C}'$ may still fail $\mathcal{C}$. The effectiveness thus depends on finding strongest $\mathcal{C}'$ to minimize such false positives. To address this issue, we divide the search space into subspaces and seek for individual approximation in each subspace. By recursively applying this strategy to subspaces, the approximation in a subspace approaches the given constraint. This strategy is called *Divide-and-Approximate*.

Separable monotonicities. The above strategy applies to a class called *separable constraints*. In a separable constraint, $f(v)\theta\sigma$, the occurrences of aggregates in f can be separated into two

groups, A^+ and A^-, that affect f in the opposite way: as a cell v grows, f monotonically increases via those in A^+ and monotonically decreases via those in A^-. For example, let $psum$ and $nsum$ be the sum of positive and negative measures, $A^- = \{psum(v)\}$ and $A^+ = \{nsum(v)\}$ for $psum(v) - nsum(v) \geq \sigma$. Therefore, by holding variables v at the maximum cell or the minimum cell for either A^+ or A^-, we are able to construct four types of approximation: weaker anti-monotone, weaker monotone, stronger anti-monotone, and stronger monotone, to prune the search of failed cells, the search of satisfying cells, or both. The details will be presented shortly. In the case that only the minimum support is given, pruning satisfying sub-cells amounts to mining maximal frequent cells in the literature [3], [6].

We review related work in Section II and define the problem in Section III. In Section IV, we present the Divide-and-Approximate strategy and show that it applies to separable constraints. In Section V and Section VI, we present an efficient implementation for the four types of approximations. We evaluate the proposed approach in Section VII. Section VIII extends this approach to Boolean combinations of aggregate constraints. We then conclude the paper.

II. Related Work

Most works on data cubes focus on efficient computation of full cube [18], [1], view materialization [10], range queries where a constraint occurs in the WHERE clause [11]. These results cannot be applied because an aggregate constraint is specified for a cell through the HAVING clause and is unknown at the time of view materialization. The full cube is often too large compared to the result satisfying the aggregate constraint.

This study is related to the works on constrained data mining [5], [13], [9], [14], [4], [16], [15], [17]. Those techniques are specific to pre-determined constraints, namely, item constraints [15], minimum confidence/improvement [4], succinct constraints [13], convertible constraints [14],

minimum average [9], and support constraints [17]. We consider all constraints specified by the whole language of SQL-like aggregates and arithmetic operators (extended to Boolean operators), and seek for a specification-independent push strategy. Further, aggregates in traditional rule mining are "extensional" where the values being aggregated are associated with the items in v. We consider "intensional" aggregates where the values being aggregated are associated with the tuples that match the items in v. Techniques for the former, such as for the extensional $avg(v)$ in [14], are not always applicable to the latter.

III. Iceberg Cube Mining

A database is a relational table R with some columns called *dimensions* D_i and some columns called *measures* M_i. A *cell* is a set of values, $d_{i_1} \cdots d_{i_k}$, over some GROUP BY list $D_{i_1} \cdots D_{i_k}$, and defines the GROUP BY partition consisting of the tuples matching $d_{i_1} \cdots d_{i_k}$. $SAT(c)$ denotes the GROUP BY partition defined by a cell c. For example, $c = \{Toyota, Vancouver, 2003\}$ is a cell on the GROUP BY list "Product, Store, Year", and $SAT(c)$ is the set of tuples containing all the values in c. $c = \{Toyota, Vancouver, 2003\}$ is a super-cell of $c' = \{Toyota, Vancouver\}$, in which case $SAT(c)$ must be a subset of $SAT(c')$. $avg(c)$, $min(c)$, $max(c)$, $sum(c)$ compute the average, minimum, maximum, sum of some measure of the tuples in $SAT(c)$, and $count(c)$ computes the number of tuples in $SAT(c)$. $ssum(c), psum(c), nsum(c)$ compute the sum of square, positive sum and (unsigned) negative sum, respectively. v/c means holding the variable v at the cell c.

Definition 3.1 (Constraints): A *(aggregate) constraint* $\mathcal{C}$ has the form $f(v)\theta\sigma$. $f(v)$ is a function of cell-valued variable v, defined by aggregates, arithmetic operators $+, -, \times, /$, and constants. θ is one of $<, \leq, \geq, >$. σ is a real. A cell c *satisfies* a constraint $\mathcal{C}$ if applying v/c to $\mathcal{C}$ evaluates to true; otherwise, c *fails* $\mathcal{C}$. $CUBE(\mathcal{C})$ denotes the set of cells that satisfy $\mathcal{C}$. $\mathcal{C}$ is

weaker than $\mathcal{C}'$ if $CUBE(\mathcal{C}') \subseteq CUBE(\mathcal{C})$. ■

Example 3.1: Let d_i, d_i' be values on dimension D_i and let v be a cell-valued variable. $v \cup \{d_i\}$ (resp. $v \cup \{d_i'\}$) denotes the variable for the cells obtained by unioning the dimension values in v and d_i. $count(v \cup \{d_i\})/count(v) \geq \sigma$ specifies *association rules*, $v \rightarrow d_i$, above the minimum confidence σ [2]. $count(v \cup \{d_i\})/count(v \cup \{d_i'\}) \geq \sigma$ specifies *emerging patterns* v with respect to the two partitions specified by two cells d_i and d_i' [7]. $var(v) \leq \sigma$ specifies the maximum variance constraint, where

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

$M[t]$ denotes the measure of tuple t. By rewriting and substituting, we have

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

In all examples, an optional minimum support can be specified separately. ■

Definition 3.2 (Iceberg cube mining): Given a database R, a constraint $\mathcal{C}$, and a minimum support $minsup$ the *iceberg cube mining* problem is to find $CUBE(\mathcal{C}) \wedge CUBE(count(v)/|R| \geq minsup)$, i.e., all frequent cells that satisfy $\mathcal{C}$ ($|R|$ denotes the number of tuples in R). ■

We treat the minimum support differently because it is optional and is anti-monotone.

Below, the terms "a-monotone"/"m-monotone" refer to "anti-monotone"/"monotone", respectively, and "β-monotone" refers to either. $\overline{\beta}$ denotes the "complement" of β, i.e., $\overline{a} = m$ and $\overline{m} = a$.

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

Definition 3.4 (Monotonicity of functions): A function $x(y)$ is *a-monotone* wrt y if x decreases whenever y grows (for cell-valued y) or increases (for real-valued y). A function $x(y)$

is m-*monotone* wrt y if x increases whenever y grows (for cell-valued y) or increases (for real-valued y). ■

$psum(v) - nsum(v)$ is m-monotone wrt $psum(v)$, a-monotone wrt $nsum(v)$, and is neither wrt v. The terms "a-monotone" and "m-monotone" are overloaded for both constraints and functions, and are differentiated from the subjects involved.

Observation 3.1: (i) $f(v) \geq \sigma$ is β-monotone if and only if $f(v)$ is β-monotone wrt v. (ii) $f(v) \leq \sigma$ is β-monotone if and only if $f(v)$ is $\overline{\beta}$-monotone wrt v. ■

A similar observation holds for $f(v) > \sigma$ and $f(v) < \sigma$.

IV. The Proposed Approach

A. Divide-and-Approximate

If the given constraint is neither a-monotone nor m-monotone, we can push some a-monotone or m-monotone approximation, called an *approximator*. There are four types of approximators: weaker a-monotone approximators, stronger a-monotone approximators, weaker m-monotone approximators, and stronger m-monotone approximators, called wa-*approximators*, sa-*approximators*, wm-*approximators* and sm-*approximators*, respectively. We use $\alpha\beta$ for these approximators, $s\beta$ for stronger approximators, $w\beta$ for weaker approximators, αa for a-monotone approximators, and αm for m-monotone approximators.

If a cell c fails a wa-approximator, we can prune the search of super-cells of c because they fail the given constraint. If a cell c fails a wm-approximator we can prune the search of (failed) sub-cells of c. If a cell c satisfies sa-approximator, we can prune the search of sub-cells of c because they satisfy the given constraint and can be generated directly from c. If a cell c satisfies a sm-approximator, we can prune the search of (satisfying) super-cells of c. However, a satisfying cell

of a $w\beta$-approximator may still fail the given constraint, and a failed cell of a $s\beta$-approximator may still satisfy the given constraint. Minimizing such "false positives" and "false negatives" depends on finding strongest $w\beta$-approximators or weakest $s\beta$-approximators. To address this requirement, we seek for local approximators in subspaces. Below, we explain this strategy using wa-approximators for $sum(v) \geq \sigma$ in the space $S = \{c \mid c \text{ is a sub-cell of } d_1 \cdots d_p\}$, where $d_1 \cdots d_p$ is a fixed cell.

First, we rewrite $sum(v) \geq \sigma$ into $psum(v) - nsum(v) \geq \sigma$ and regard $psum$ as the "profit" and $nsum$ as the "cost". *Ignoring* the "cost" entirely gives the first wa-approximator, $psum(v) \geq \sigma$. *Under-estimating* the "cost" by the minimum for any cell gives a stronger wa-approximator, i.e., $psum(v) - nsum(d_1 \cdots d_p) \geq \sigma$. That is, if it is so hopeless to pass the threshold even with the minimum cost, there is no need to consider any super-cell of v in S. A still better attempt is to divide S into subspaces $S_1 = \{d_1 c\}$ and $S_0 = \{c\}$, where c is a sub-cell of $d_2 \cdots d_p$, and use $psum(v) - nsum(d_1 d_2 \cdots d_p) \geq \sigma$ in S_1 and $psum(v) - nsum(d_2 \cdots d_p) \geq \sigma$ in S_0. The latter is stronger than the former. We can apply this strategy recursively to S_0 and S_1 to obtain increasingly stronger wa-approximators in subspaces. We call this strategy *Divide-and-Approximate*.

B. Separable constraints

To obtain an approximator for $f(v)\theta\sigma$, the key is to separate the aggregates in $f(v)$ into two groups, A^+ and A^-, such that as a cell v grows, A^+ increases the value of f, and A^- decreases the value of f. We then can obtain an approximator by holding the variable v in one of A^+ and A^- at the maximum cell or the minimum cell. Below, A^+/c and A^-/c mean holding the variable v in A^+ and A^- at the cell c.

Example 4.1: Consider $avg(v) \geq \sigma$, or written $psum(v)/count1(v) - nsum(v)/count2(v) \geq$

σ. The two occurrences of $count$ are renamed because they have different memberships in A^+ and A^-. Note that all aggregates now are a-monotone wrt v. Let $A^+ = \{nsum(v), count1(v)\}$ and $A^- = \{psum(v), count2(v)\}$. avg is a-monotone wrt each aggregate in A^+ and is m-monotone wrt each aggregate in A^-. Therefore, as v grows, avg increases via A^+ by composing two a-monotone functions, i.e., avg wrt A^+ and A^+ wrt v, and avg decreases via A^- by composing one m-monotone function with one a-monotone function, i.e., avg wrt A^- and A^- wrt v. Let $\underline{c}$ and $\overline{c}$ be the minimum and maximum cells. Applying $A^+/\overline{c}$ gives the wa-approximator:

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

and applying $A^-/\underline{c}$ gives the sm-approximator:

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

■

To separate the aggregates into A^+ and A^-, a requirement is that every aggregate be β-monotone and *sign-preserved*, i.e., never change the sign. Imagine what if $count1$ could have changed the sign: its membership in A^+ or A^- would depend on the sign. Below, we rewrite an aggregate constraint and partition the space to meet these requirements. First of all, $psum$, $nsum$, $count$ are β-monotone and sign-preserved, and sum and avg can be rewritten into such aggregates, i.e., $sum = psum - nsume$ and $avg = (psum - nsume)/count$. max and min can be rewritten into β-monotone and sign-preserved aggregates: $max = pos \times pmax - (1 - pos) \times nmin$ and $min = neg \times nmax + (1 - neg) \times pmin$, where

- $pos(v)$: return 1 if some tuple in $SAT(v)$ has a non-negative measure (including 0), return 0 otherwise.
- $neg(v)$: return 1 if some tuple in $SAT(v)$ has a non-positive measure (including 0), return 0 otherwise.

- $pmax(v)$: return the maximum non-negative measure in $SAT(v)$, return 0 if all measures in $SAT(v)$ are negative.
- $pmin(v)$: return the minimum non-negative measure in $SAT(v)$, return 0 if all measures in $SAT(v)$ are negative.
- $nmax(v)$: return the maximum $|M|$ where M is a non-positive measure in $SAT(v)$, return 0 if all measures in $SAT(v)$ are positive.
- $nmin(v)$: return the minimum $|M|$ where M is a non-positive measure in $SAT(v)$, return 0 if all measures in $SAT(v)$ are positive.

Note that these new aggregates are β-monotone and sign-preserved.

Consider an arithmetic function f of sign-preserved β-monotone aggregates. Suppose that f contains k denominators $Z_1, \cdots, Z_k$ that are not sign-preserved. A *sign-space* consists of all cells c that agree on the sign of Z_i, $1 \leq i \leq k$. We denote a sign-space by a bitmap $b_1 \cdots b_k$ where b_i represents the sign of Z_i, i.e., 1 for "-" and 0 for "+". Conceptually, the whole space can be partitioned into 2^k sign-spaces, corresponding to the 2^k bitmaps, such that in each sign-space no denominator changes the sign. Below is the main result we like to establish.

Theorem 4.1: Consider an arithmetic function f of sign-preserved β-monotone aggregates. There is a rewriting f' of f such that in each sign-space, every operand of $\times$ and $/$ in f' is sign-preserved.

Proof: In a sign-space, no denominator of $/$ changes the sign. If an operand of $\times$ changes the sign, it must be an expression of $+$ and $-$ because each aggregate is sign-preserved. We can then distribute $\times$ over $+$ and $-$ in the expression. This distribution is repeated as long as an operand of $\times$ changes the sign. ■

We say that f' in Theorem 4.1 is $(\times, /)$-*sign-preserved* (wrt sign-spaces). In a sign-space, since no operand of $\times$ and $/$ in f' changes the sign, each aggregate either increases or decreases f', but not both, as v grows. In other words, f' is either m-monotone or a-monotone wrt each aggregate in f', while fixing the other aggregates. Therefore, in each sign-space the A^+/A^- membership of an aggregate in f' is well defined.

Definition 4.1 (Separable constraints): $f\theta\sigma$ is a *separable constraint* if f is an arithmetic function of sign-preserved β-monotone aggregates. ■

In light of Theorem 4.1, we assume that a separable constraint $f\theta\sigma$ is $(\times, /)$-sign-preserved.

Definition 4.2 (A^+ and A^-): Consider a separable constraint $f\theta\sigma$ and some sign-space. Let A^+ and A^- be the partition of aggregates (occurrences) in f, denoted by $f(A^+; A^-)$, such that i) $agg(v)$ is in A^+ if $agg(v)$ is β-monotone wrt v and if f is β-monotone wrt $agg(v)$ in the sign-space by fixing other aggregates, ii) $agg(v)$ is in A^- if $agg(v)$ is β-monotone wrt v and if f is $\overline{\beta}$-monotone wrt $agg(v)$ in the sign-space by fixing other aggregates. ■

In other words, A^+ contains the aggregates $agg(v)$ whose monotonicity wrt v is the same as f wrt $agg(v)$. If we hold A^- at constant, $f(v)$ becomes composing two functions of the same monotonicity, thus, m-monotone wrt v. A^- contains the aggregates $agg(v)$ whose monotonicity wrt v is the complement of f wrt $agg(v)$. If we hold A^+ at constant, $f(v)$ becomes composing two functions of the complement monotonicity, thus, a-monotone wrt v.

Corollary 4.1: The following classes are separable constraints, with each (except the first) generalizing the previous one. (i) All constraints built by arithmetic functions of SQL aggregates $count$, sum, avg, max and min. (ii) All constraints built by arithmetic functions of $count$, $psum$, $nsum$, pos, neg, $pmax$, $pmin$, $nmax$, $nmin$. (iii) All constraints built by arithmetic functions of sign-preserved β-monotone aggregates. ■

1. $sum(v)\theta\sigma$	8. $sum(v \cup \{d_i\})/sum(v \cup \{d'_i\})\theta\sigma$
2. $avg(v)\theta\sigma$	9. $avg(v)/max(v)\theta\sigma$
3. $var(v)\theta\sigma$	10. $avg(v)/min(v)\theta\sigma$
4. $count(v \cup \{d_i\}) - count(v \cup \{d'_i\})\theta\sigma$	11. $avg(v \cup \{d_i\})/avg(v)\theta\sigma$
5. $count(v \cup \{d_i\})/count(v)\theta\sigma$	12. $avg(v \cup \{d_i\})/avg(v \cup \{d'_i\})\theta\sigma$
6. $count(v \cup \{d_i\})/count(v \cup \{d'_i\})\theta\sigma$	13. $max(v) - avg(v)\theta\sigma$
7. $sum(v \cup \{d_i\}) - sum(v \cup \{d'_i\})\theta\sigma$	14. $min(v) - avg(v)\theta\sigma$

TABLE I

SOME SEPARABLE CONSTRAINTS (d_i, d'_i ARE CONSTANTS)

The above corollary conveys three points. First, separable constraints include most constraints arising from real life. Second, the single strategy of Divide-and-Approximate provides a uniform way to deal with all separable constraints. Third, the notion of separable constraints is open to the arithmetic function f and sign-preserved β-monotone aggregates in f. This flexibility is essential in real life where constraints are specified by the user and are not known in advance.

The following theorem tells how to compute A^+ and A^- for $f\theta\sigma$, denoted $f(A^+; A^-)\theta\sigma$, in a given sign-space.

Theorem 4.2: Consider $f_1(A_1^+; A_1^-)$ and $f_2(A_2^+; A_2^-)$. $(A^+; A^-)$ for a function 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) is $(+, +)$, $A^+ = A_1^+ \cup A_2^+$ and $A^- = A_1^- \cup A_2^-$. If the sign is $(-, -)$, consider $(-f_1) \times (-f_2)$, thus, reduced to (1) and $(+, +)$ sign. If the sign is

	(m) m-monotone	(a) a-monotone
(w) weaker	wm-approximator obtained by $A^-/\underline{c}$ prune failed $\langle \underline{c}, \overline{c} \rangle$	wa-approximator obtained by $A^+/\underline{c}$ prune failed $\langle \underline{c}, \overline{c} \rangle$
(s) stronger	sm-approximator obtained by $A^-/\underline{c}$ prune satisfying $\langle \underline{c}, \overline{c} \rangle$	sa-approximator obtained by $A^+/\overline{c}$ prune satisfying $\langle \underline{c}, \overline{c} \rangle$

TABLE II

APPROXIMATORS FOR $f(v) \leq \sigma$

	(m) m-monotone	(a) a-monotone
(w) weaker	wm-approximator obtained by $A^-/\underline{c}$ prune failed $\langle \underline{c}, \overline{c} \rangle$	wa-approximator obtained by $A^+/\overline{c}$ prune failed $\langle \underline{c}, \overline{c} \rangle$
(s) stronger	sm-approximator obtained by $A^-/\overline{c}$ prune satisfying $\langle \underline{c}, \overline{c} \rangle$	sa-approximator obtained by $A^+/\underline{c}$ prune satisfying $\langle \underline{c}, \overline{c} \rangle$

TABLE III

APPROXIMATORS FOR $f(v) \geq \sigma$

$(+,-)$, consider $f_1 \times (-f_2)$, and if the sign is $(-,+)$, consider $(-f_1) \times f_2$.

5) f_1/f_2: If the sign of (f_1, f_2) is $(+,+)$, $A^+ = A_1^+ \cup A_2^-$ and $A^- = A_1^- \cup A_2^+$. Similar to 4), other signs of (f_1, f_2) can be reduced to 1) and $(+,+)$ sign. ■

C. Approximators

Consider a sign-space. Let $\langle \underline{c}, \overline{c} \rangle$ denote the set of cells with $\underline{c}$ as the minimum cell and $\overline{c}$ as the maximum cell. Following Observation 3.1 and Definition 4.2, Tables II and III summarize the construction of $\alpha\beta$-approximators. These constructions remain unchanged by replacing $\geq$ with $>$ and replacing $\leq$ with $<$. "Pruning satisfying $\langle \underline{c}, \overline{c} \rangle$" means outputting the minimum $\underline{c}$ and maximum $\overline{c}$ without testing the constraint for every cell bounded by them. To use these approximators for pruning, we need to identify a sign-space and minimum/maximum cells $\underline{c}$ and $\overline{c}$ in the sign-space, and the space $\langle \underline{c}, \overline{c} \rangle$ without enumerating its cells. We consider these implementation issues in Section V.

V. The Implementation

A. Strongly separable constraints

The effectiveness of $\alpha\beta$-approximators depends on having a large $\langle \underline{c}, \overline{c} \rangle$ within a sign-space, i.e., a "connected" sign-space.

Definition 5.1: A constraint is *sign-space connected* if every denominator is either sign-preserved or β-monotone wrt v. A constraint is *strongly separable* if it is both separable and sign-space connected. ■

In a strongly separable constraint, every denominator changes the sign *at most once* as the cell v grows. In Table I, except for 8, 11, 12, all constraints are strongly separable. If avg is non-negative, 8, 11 and 12 are strongly separable. Let $sign(c)$ denote the the bitmap that identifies the sign-space of a cell c.

Theorem 5.1 (Inward monotonicity): Consider a strongly separable constraint. (i) For every cell c in $\langle \underline{c}, \overline{c} \rangle$, $sign(c) = sign(\underline{c})$. (ii) If $\underline{c}$ and $\overline{c}$ fail an $\alpha\beta$-approximator, so do all cells in $\langle \underline{c}, \overline{c} \rangle$. (iii) If $\underline{c}$ and $\overline{c}$ satisfy an $\alpha\beta$-approximator, so do all cells in $\langle \underline{c}, \overline{c} \rangle$.

Proof: (i) follows because the sign changes at most once as a cell v grows. (ii) and (iii) follow because $\underline{c}$ and $\overline{c}$ agree on whether to satisfy a $\alpha\beta$-approximator that is either a-monotone or m-monotone in $\langle \underline{c}, \overline{c} \rangle$. ■

In other words, knowing that a minimum $\underline{c}$ and a maximum $\overline{c}$ fail (or satisfy) the constraint is sufficient to know that all cells between them fail (or satisfy) the constraint. By identifying such $\underline{c}$ and $\overline{c}$, we can prune the work of generating the partitions for all cells between them.

B. Approximators originating at leaf nodes

In this section, we construct wa-approximators for a strongly separable constraint $f \geq \sigma$, $f(A^+/\overline{c}; A^-)\theta\sigma$, where $\overline{c}$ is the maximum cell that fails $f \geq \sigma$. See the upper-right corner in Table III. First, we describe the search space.

The lexicographic tree. A node in the *lexicographic tree* corresponds to a GROUP BY list $D_1 \cdots D_k$, $k \geq 0$, in the lexicographic order. The root corresponds to the null GROUP BY list and has one child for each dimension D_i, in the lexicographic order. For a non-root node $u = D_1 \cdots D_{k-1} D_k$ with q siblings on its right, $D_1 \cdots D_{k-1} D_{k+i}$, $1 \leq i \leq q$, the ith child of u, $1 \leq i \leq q$, is generated by the extra dimension at the ith sibling of u, i.e., $D_1 \cdots D_k D_{k+i}$ (ith child). $tree(u)$ denotes the subtree rooted at node u and $tail(u)$ denotes the set of dimensions in $tree(u)$. Note that $tail(u)$ is represented by the leaf node on the left-most path in $tree(u)$.

The *depth-first search* is illustrated by the sequence number next to each node in Figure 1. First, we examine the empty cell at the root. Next, we produce partitions a_1 to a_i. Next, we produce partitions $a_1 b_1, \cdots$ at node AB, $a_1 b_1 c_1, \cdots$ at node ABC, $a_1 b_1 c_1 d_1, \cdots$ at node $ABCD$, and $a_1 b_1 c_1 d_1 e_1, \cdots$ at node $ABCDE$, in that order. After completing $a_1 b_1 c_1 d_1$, we "backtrack" to node $ABCD$ to process other partitions at the node in a similar manner, "backtrack" to node ABC to partition on dimension E. After completing the $a_1 b_1 c_1$ partition, we proceed to $a_1 b_1 c_2, a_1 b_1 c_3, \cdots$. We then "backtrack" to node AB to process $a_1 b_2, a_1 b_3, \cdots$, and "backtrack" to A to process $a_2, a_3, \cdots$, and finally "backtrack" to the root to process other child nodes of the root. This search was used in the *Bottom-Up Computation (BUC)* [5] to find frequent cells, where partitioning is stopped if a cell becomes infrequent.

Constructing wa-approximators. Consider a strongly separable $\mathcal{C}$: $f(v) \geq \sigma$. Suppose that we reach a leaf node u_0 and find a cell p at u_0 fails $\mathcal{C}$. Following Table III, we construct

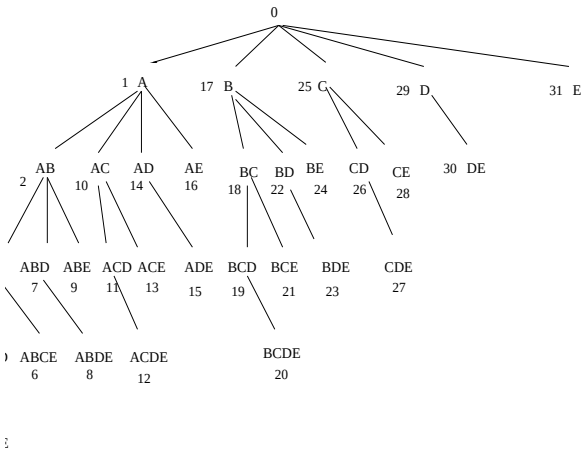

Fig. 1. The lexicographic tree for A, B, C, D, E

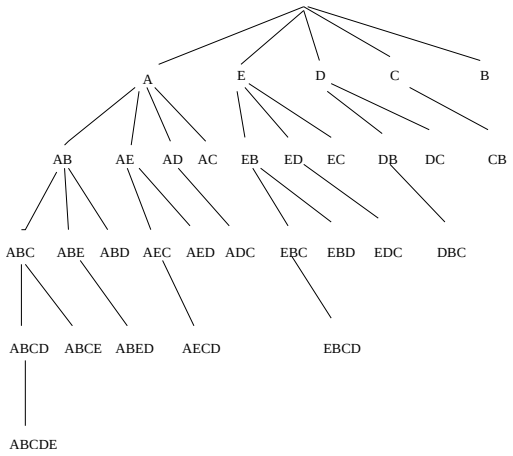

Fig. 2. The rollback tree for A, B, C, D, E

the wa-approximator in the sign-space $sign(p)$: $\mathcal{C}_p : f(A^+/p; A^-) \geq \sigma$. Consider an ancestor u_k of u_0 such that u_0 is on the left-most path in $tree(u_k)$ and $sign(p[u_k]) = sign(p)$. Define $tree(u_k, p) = \{p[u] \mid u \text{ is a node in } tree(u_k)\}$, where $p[u]$ is the projection of cell p onto the dimensions at the node u. Note that p and $p[u_k]$ are the maximum cell and the minimum cell in $tree(u_k, p)$, respectively. From Theorem 5.1, if $p[u_k]$ fails $\mathcal{C}_p$, all cells in $tree(u_k, p)$ fail $\mathcal{C}_p$ (thus $\mathcal{C}$).

To leverage the above pruning, we push p to u_k to mark that all cells in $tree(u_k, p)$ fail $\mathcal{C}_p$. Particularly, on backtracking from the *first* child u_{k-1} to the parent u_k, for each p pushed to u_{k-1}, we check if $sign(p[u_k]) = sign(p)$ and if $p[u_k]$ fails $\mathcal{C}_p$. If both conditions hold, we push p to u_k. To exploit each p pushed to u_k, for each remaining child w_j of u_k, we prune all tuples that match p over $tail(w_j)$, because such tuples generate *only* cells in $tree(u_k, p)$, all of which fail $\mathcal{C}_p$. This new form of partitioning is formalized below.

The filtered-partitioning. A *filter* at u_k refers to a cell pushed to u_k. The *filtered-partitioning* for a child w_j of u_k refers to partitioning all the tuples at u_k except those that match any filter

at u_k over $tail(w_j)$. By not partitioning such tuples, affected are only those cells in $tree(u_k, p)$, which are known to fail $\mathcal{C}_p$. Note that it does not work to prune "all" partitioning below $p[u_k]$ because there may exist some partition p' at some node u in $tree(u_k)$ such that p' is not in $tree(u_k, p)$, i.e., $p'[u_k] = p[u_k]$ but $p'[u] \neq p[u]$. To tell if a cell in $tree(u_k)$ is in $tree(u_k, p)$, we also partition the filters pushed to u_k, just like partitioning regular tuples. Such partitions are called *auxiliary partitions*.

Theorem 5.2: A cell in $tree(u_k)$ is in $tree(u_k, p)$ for some filter p if and only if the corresponding auxiliary partition is non-empty.

Proof: For a cell c in $tree(u_k)$, if its auxiliary partition is non-empty, for every filter p in the auxiliary partition, c is a sub-cell of p, so in $tree(u_k, p)$. On the other hand, if a cell c is in $tree(u_k, p)$, for some filter p at u_k, p is a super-cell of c, so belongs to the auxiliary partition of c. ■

Example 5.1: Consider the constraint $\mathcal{C}$: $sum(v) \geq \sigma$, or written as $psum(v) - nsum(v) \geq \sigma$. $A^+ = \{nsum(v)\}$ and $A^- = \{psum(v)\}$ because as v grows, sum increases via $nsum(v)$ and decreases via $psum(v)$. In Figure 1, suppose that we reach a cell p at the leaf node $u_0 = ABCDE$ and p fails $\mathcal{C}$. The wa-approximator $\mathcal{C}_p$ is $psum(v) - nsum(p) \geq \sigma$. Note that $nsum(p)$ is an under-estimate of $nsum(v)$ for any cell v at a node in $tree(u_k)$ such that u_0 is on the left-most leaf in $tree(u_k)$.

On backtracking to the node ABC, suppose that $p[ABC]$ is in $sign(p)$ and fails $\mathcal{C}_p$. At the child $ABCE$, the filtered-partitioning will not partition any tuple t such that $t[ABCE] = p[ABCE]$ because they generate only cells in $tree(ABC, p)$. Subsequently, these tuples are not examined in any lower partitioning. On backtracking to the node AB, if $p[AB]$ is in $sign(p)$ and fails $\mathcal{C}_p$, at the child ABD the filtered-partitioning will not partition any tuple t such that $t[ABDE] =$

$p[ABDE]$, where $ABDE = tree(ABD)$, and at the child ABE, the filtered-partitioning will not partition any tuple t such that $t[ABE] = p[ABE]$. Note that, if $p[AB]$ satisfies $\mathcal{C}(p)$, all higher level sub-cells, i.e., $p[A]$ and the empty cell, must satisfy $\mathcal{C}(p)$. ■

Remarks. The effectiveness of filtered-partitioning depends on a filter p being pushed up a left-most path to a high ancestor u_k so that filtered-partitioning can be performed in a large subtree below u_k. This occurs under the following conditions: the threshold σ is so large that the under-estimate $nsum(p)$ does not help to pass it, there are many negative measure values, $nsum(p)$ is a good approximation of $nsum(p[u_k])$. The last condition occurs when the values in $p[u_k]$ are correlated to those in $p - p[u_k]$, or when the tuples matching $p[u_k]$ but not p have close-to-zero negative values.

C. Approximators originating at any nodes

So far, a filter is generated by partitioning all the way to a leaf node. If a minimum support is specified, it makes sense to restrict filters to frequent cells. Consider Figure 1. Suppose that the cell $p = abcd$ at $ABCD$ is frequent, but the cell $abcde$ at node $ABCDE$ is not. Now, even if we can push p to $u_k = A$, we cannot prune the cells in $tree(u_k, p)$, i.e., ac, ad, acd, abd, because cells not in $tree(u_k, p)$, i.e., $ace, ade, acde, abde$, "depend on" the cells in $tree(u_k, p)$. The fact that the dimension E occurs in every leaf node presents the worst scenario for pruning cells not involving E. This difficulty stems from the "sequential growth" of the lexicographic tree where the ith child of a node is grown by the ith sibling. We propose a novel "rollback growth" to address this problem.

The rollback tree. Suppose that u has q siblings on its right, $D_1 \cdots D_{k-1} D_{k+i}$, $1 \leq i \leq q$. For $1 \leq i \leq q$, the ith child of u is generated using the $(i-1)$th sibling (with 0 treated as q): $D_1 \cdots D_k D_{k+i-1}$. $RBtree(u)$ denotes the subtree at a node u. $RBtree(u, p)$ denotes the set of

projected cells of p onto the nodes in $RBtree(u)$. As before, $tail(u)$ denotes the dimensions in $RBtreee(u)$. Note that the rollback tree assumes no fixed order of dimensions.

Consider Figure 2. The first child AB of $u = A$ 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 last dimension E on the left-most path $ABCDE$ now occurs in the second child of the nodes on this path (i.e., $ABCE, ABE, AE, E$), the second last dimension D on the left-most path $ABCDE$ occurs in the third child of the nodes on this path (i.e., ABD, AD, D), and so on. As a result, E does not occur in the following subtrees: $RBtree(AC)$, $RBtree(AD)$, $RBtree(ABD)$, $RBtree(B)$, $RBtree(C)$, and $RBtree(D)$. Therefore, we can use a cell $p = abcd$ at the node $ABCD$ to prune the sub-cells of p in these subtrees. These subtrees are defined by the notion of filtering scope.

Definition 5.2 (The filtering scope): Consider a (possibly non-leaf) node u_0, a cell p at u_0, and the left-most path $u_k, \cdots, u_0$ in $RBtree(u_k)$, $k \geq 0$. p is a ***filter*** generated at u_0 and ***anchored*** at u_k if (i) p is frequent and fails $\mathcal{C}$, (ii) no partition of p at the first child of u_0 satisfies (i), (iii) u_k is the highest possible node such that $sign(p[u_k]) = sign(p)$ and fails $\mathcal{C}_p$. The ***filtering scope*** of p consists of $RBtree(w^i, p)$, for $k \geq i \geq 1$, where w^i are the last $i-1$ child nodes of u_i. The tuples in the partition for p are ***generating tuples*** of p. ■

Intuitively, w^i are such child nodes of u_i that $tail(w^i)$ contains only the dimensions at the node u_0. This ensures that all cells in $RBtree(w^i, p)$ are sub-cells of p and pruning them has no effect on any cell that is not a sub-cell of p. (ii) ensures the maximality of p. (iii) ensures the maximality of the filtering scope of p.

Example 5.2: Consider the rollback tree in Figure 2. Suppose that $p = abcd$ is a filter generated at node $ABCD$ and anchored at node A. We have $u_3 = A, u_2 = AB, u_1 = ABC, u_0 = ABCD$.

The filtering scope of p consists of $RBtree(AD, p)$ and $RBtree(AC, p)$, where AD and AC are the last 2 child nodes of u_3, and $RBtree(ABD, p)$, where ABD is the last child node of u_2. If $p = ebc$ is a filter generated at node EBC and anchored at node E, $u_2 = E, u_1 = EB, u_0 = EBC$, and the filtering scope of p is $RBtree(EC, p)$, where EC is the last child node of u_2. $p = ebc$ is not a filter generated at EBC and anchored at the root because EBC is not on the left-most path in $RBtree(root)$. ■

Theorem 5.3: Let p be a filter generated at u_0 and anchored at u_k. (i) The filtering scope of p is a subspace of $\langle p[u_k], p\rangle$. (ii) All cells in the filtering scope of p fail $\mathcal{C}_p$.

Proof: (i) follows from the above discussion. (ii) follows from Theorem 5.1 and (i). ■

D. The algorithm

Following the above discussions, we modify BUC for our purpose as follows. (i) We use the rollback tree instead of the lexicographic tree. (ii) On backtracking from the first child u_i to the parent u_{i+1}, we push a filter p at the child to the parent if $p[u_{i+1}]$ fails $\mathcal{C}_p$ and if $sign(p[u_{i+1}]) = sign(p)$. A filter p at u_{i+1} is stored as $\langle p, i+1\rangle$. (iii) For the jth child w_j of u_{i+1}, where $j > 1$, we apply Definition 5.2 to determine the filters for the filtered-partitioning at w_j. The jth child w_j from the left is the rth child from the right, where $r = Num\_child(u_{i+1}) - j + 1$. So the filters for filtered-partitioning at w_j have the form $\langle p, r+1\rangle$, where p is a filter pushed to u_{i+1}. (iv) After processing all child nodes of u_{i+1}, if no filter is pushed to u_{i+1} (to ensure the maximality in Definition 5.2(ii)) and if the current partition p at u_{i+1} fails $\mathcal{C}$, we generate a new filter p at u_{i+1}. (v) At each node, we partition filters to produce auxiliary partitions, which are used to test if a cell is in any pruning scope.

For any two filters at the same node, their generating tuples are disjoint because neither filter is a super-cell of another (Definition 5.2(ii)). Since each (frequent) filter has at least $minsup \times |R|$

generating tuples, at most $1/minsup$ filters are pushed to a node in the rollback tree. Therefore, there are at most $l \times 1/minsup$ filters on a partitioning path of length l. This bound is independent of the database size $|R|$, which is highly desirable for the scalability on very large databases. If partitioning is implemented as "moving" instead of "copying", this bound remains unchanged after partitioning filters. For example, with $minsup = 0.1\%$, we have at most $1,000 \times l$ filters on a path of length l.

VI. Extension to Other Approximators

A $w\beta$-approximator is effective when many cells fail the given constraint, i.e., the constraint is tight. A $s\beta$-approximator is effective when many cells satisfy the given constraint, i.e., the constraint is loose. Below, we consider implementation for other approximators of $f \geq \sigma$. A similar consideration applies to the comparators $\leq, >, <$.

wm-approximators. A wm-approximator is obtained by $A^-/\underline{c}$ and is used to prune failed $\langle \underline{c}, \bar{c} \rangle$ (Table III). $\underline{c}$ is the highest frequent cell p' that fails $\mathcal{C}$ at some node u_k. We construct the wm-approximator $\mathcal{C}_{p'}$ following Table III, and go down from p' following the left-most path, identify $\bar{c}$ as the lowest frequent cell p that fails $\mathcal{C}_{p'}$ but satisfies $sign(p') = sign(p)$. Note that $p' = p[u_k]$. From Theorem 5.1, all the cells in $\langle p[u_k], p \rangle$ fail $\mathcal{C}_{p'}$. Upon backtracking, like for wa-approximators, we push the filter p up to the node u_k, for the filtered-partitioning in the filtering scope of p. The filtering scope of p is defined as in Definition 5.2, with "$\mathcal{C}_p$" replaced with "$\mathcal{C}_{p[u_k]}$".

sm-approximators. A sm-approximator is obtained by $A^-/\bar{c}$ and is used to prune satisfying $\langle \underline{c}, \bar{c} \rangle$ (Table III). We construct the sm-approximator $\mathcal{C}_p$ as in Table III. In Definition 5.2, replace "fails" with "satisfies". Theorem 5.1 implies that all the cells in the filtering scope of p satisfy $\mathcal{C}_p$.

sa**-approximators**. A sa-approximator is obtained by $A^+/\underline{c}$ and is used to prune satisfying $\langle \underline{c}, \overline{c} \rangle$ (Table III). We look for the highest frequent cell p', at some u_k on the left-most path that satisfies $\mathcal{C}$, constructing the sa-approximator $\mathcal{C}_{p'}$, and look for the lowest frequent cell p on the left-most path that satisfies $\mathcal{C}_{p'}$ and $sign(p) = sign(p')$. In Definition 5.2, we replace "fails $\mathcal{C}$" with "satisfies $\mathcal{C}$" and replace "fails $\mathcal{C}_p$" with "satisfies $\mathcal{C}_{p[u_k]}$". Theorem 5.1 implies that all the cells in $\langle p[u_k], p \rangle$, thus, in the filtering scope of p, satisfy $\mathcal{C}_{p[u_k]}$. The rest is similar to the case of sm-approximators.

Combinations of approximators. Pushing both a $w\beta$-approximators and an $s\beta$-approximators prunes both failed and satisfying cells, whereas pushing both a wm-approximator and a wa-approximator prunes failed cells by either approximator. This can be done by maintaining a separate set of filters for each approximator. The bound on filters for k approximators is k times the bound in Subsection V-D. Such combinations are beneficial if the subspaces pruned by different approximators are largely non-overlapping. The perfect non-overlapping is guaranteed by the combination of $w\beta$-approximators and $s\beta$-approximators because the former prunes failed cells and the latter prunes satisfying cells.

VII. Experiments

We empirically evaluated the Divide-and-Approximate approach or DnA in short. The DnA family refers to the algorithms by pushing wa-approximators, sm-approximators, wm-approximators and sa-approximators, denoted by WA, SM, WM and SA, and combinations of two approximators, denoted by WA/SM, WA/SA, WM/SM and WM/SA. We will explain why we do not consider combinations of more than two approximators. We considered two constraints, $sum \geq \sigma$ and $avg \geq \sigma$, where sum is rewritten into $psum(x) - nsum(x)$, with or without the minimum support. These constraints capture a minimum requirement on two types of growth, i.e., difference

Parameter	Meaning	Default setting
n	number of tuples	100,000
m	number of dimensions	15
$card[i]$	cardinality of the ith dimension	10
$[0, Mmax]$	normally distributed positive measure	$[0, 10]$
$[-Mmin, 0]$	normally distributed negative measure	$[-10, 0]$
α	split factor of measure	0.5
β	repeat factor of groups	1,000
γ	Poisson mean of repeat dimension #	10
σ	minimum sum	300
$minsup$	minimum support	0.5%

TABLE IV

THE PARAMETERS OF THE DATA GENERATOR

and ratio.

We compared DnA with BUC and BUC+. BUC pushes only the minimum support (when it is specified). BUC+ pushes the minimum support and the weaker a-monotone $psum \geq \sigma$. All these algorithms are based on the depth-first search, which minimizes the difference contributed by factors other than the proposed pruning. We considered two performance criteria, *execution time* and *tuple examination*. The tuple examination refers to the number of times a tuple or filter is examined during partitioning. The partitioning operation was implemented by a linear sorting algorithm called CountingSort in [5]. All algorithms were implemented in C and tested on a PC with Windows 2000, CPU clock of 1G and memory of 512M.

A. Experiments on synthetic data sets

As pointed out in Section V-B, the effectiveness of $\alpha\beta$-approximators depends on the distribution of positive and negative measure values, the threshold σ and the correlation of dimension

values. Synthetic data sets were generated to simulate a wide range of such characteristics. We iteratively added groups of new tuples using the parameters in Table IV. In each iteration, we add a group of $r = rand() \times \beta$ new tuples $t_1, \cdots, t_r$ that repeat the values on d randomly determined dimensions. $rand()$ generates a number uniformly distributed in the range $[0, 1]$. d follows the Poisson distribution of the mean γ. 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. For each tuple in a group $t_1, \cdots, t_r$, we toss a $\alpha/(1-\alpha)$-weighted coin to choose the normal distribution for the negative measure or the normal distribution for the positive measure. [1]

The search of the full cube requires $2^{15} \times 100,000 = 3,276,800,000$ tuple examinations, at 0% minimum support, and BUC took about 9,000 seconds. For the trivial "true" $\mathcal{C}$, every cell satisfies $\mathcal{C}$, and so WM and WA are inapplicable. SM and SA pruned the search of the cells in $\langle \underline{c}, \overline{c} \rangle$ (see Table III), where $\underline{c}$ is the empty cell and $\overline{c}$ is a maximal frequent cell. In this case SM and SA degenerated into mining maximal frequent cells. Figure 3 compared SM and SA with BUC for different minimum supports while fixing other parameters at the default setting. Hence, our strategies provided additional pruning beyond the classic a-monotonicity based pruning.

1) $sum \geq \sigma$*:* Figure 4 and 5 show the results for $sum \geq \sigma$.

The effect of minimum support. Figure 4(a,a') plots the execution time on the left and tuple examination on the right. Refer to Table IV for default settings. The first observation is that, as the minimum support was reduced, BUC slowed down quickly, whereas BUC+ and the DnA family picked up the pruning via the constraint $psum \geq \sigma$ and the approximator. Particularly, as the minimum support was reduced, eventually to 0% (not shown here), the time of BUC

[1]The range [a,b] for the normal distribution has 95% confidence interval.

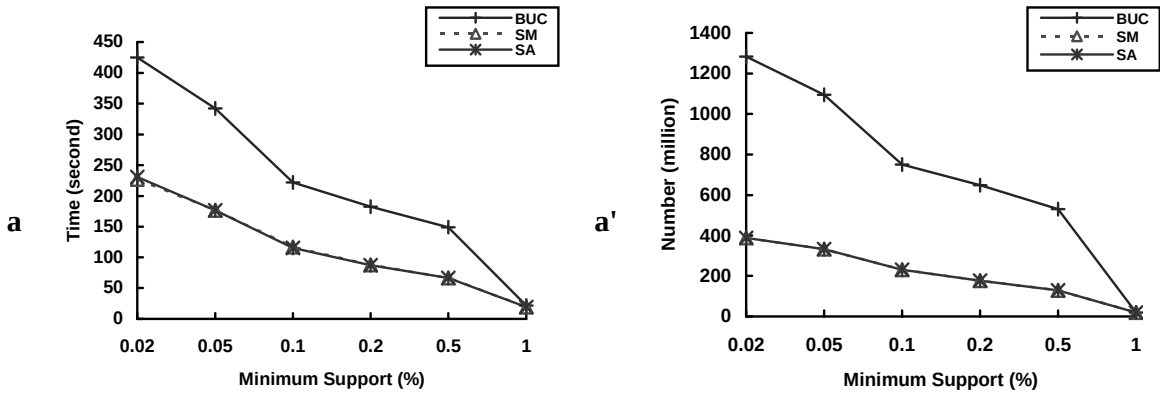

Fig. 3. Minimum support only

quickly increased, eventually to 9,000 seconds, whereas the time of other algorithms remained similar to that at the minimum support of 0.02%. This showed that the constraint pushing beyond minimum support is important in dealing with explosion of computation.

In this experiment, $w\beta$-approximators, i.e., WA and WM, performed better than $s\beta$-approximators, i.e., SA and SM. Recall that $w\beta$-approximators prune failed cells, whereas $s\beta$-approximators prune (the search of) satisfying cells (Table III). For the default threshold $\sigma = 300$ and default ranges $[0, 10]$ and $[-10, 0]$ of the positive and negative measures, it is easier to fail a $w\beta$-approximator than to satisfy a $s\beta$-approximator. As a result, pruning failed cells is more effective than pruning satisfying cells.

The effect of minimum sum. Figure 4(b,b') plots the performance over a range of minimum sum σ. WM and WA benefited from a larger σ, whereas SM and SA benefited from a smaller σ, because a larger σ helps generate failed filters and a smaller σ helps generate satisfying filters. With the default minimum support of 0.5%, BUC+ is not better than BUC because the minimum support constraint is stronger than $psum \geq \sigma$. However, as in Figure 6(a,a'), for a smaller minimum support, BUC+ benefited from the positive term constraint.

The effect of correlation. Figure 4(c,c') and (d,d') show the performance for a range of repeat factor β and Poisson mean γ, respectively. For a "dense" data set with a larger β or a larger γ, all algorithms took a longer time. WM and WA performed better than SM and SA for the default setting of $\sigma = 300$. The converse was observed for a smaller σ in Figure 4(b,b') where the existence of many satisfying cells made pruning such cells more effective.

The scalability. In Figure 5(e), we varied the number of dimensions m from 15 to 21 and kept the Poison mean γ at $2/3$ of m and other parameters at the default setting. In Figure 5(f), we varied the database size n from 200K to 1,000K and kept the repeat factor β at 1% of n and other parameters at the default setting. WM and WA showed a better scalability than other algorithms. But for a smaller σ, SM and SA were more scalable (not shown here).

The effect of split factor. Figure 5(g) shows the performance over a range of split factor α, with other parameters at their default settings. A larger split factor generated more tuples with a negative measure. This makes it easier to generate more filters required by WM and WA. In this aspect, a large split factor is similar to a large minimum sum.

The effect of combining approximators. Figure 5(h) shows that combining "heterogeneous" approximators, i.e., one $w\beta$-approximator and one $s\beta$-approximator, inherited the benefit of both. As the split factor varied, one approximator became more effective, whereas the other became less effective (see Figure 5(g)). Therefore, the pruning is effective in the whole range of split factor. To the contrary, in a "homogeneous" combination of two $w\beta$-approximators or two $s\beta$-approximators, each approximator made the other approximator redundant because they reached the peak performance under a similar condition, i.e., either both prune failed cells or both prune satisfying cells. We will not further consider combinations of three or more types of approximators (such as WA/SA/SM) because such combinations always contained

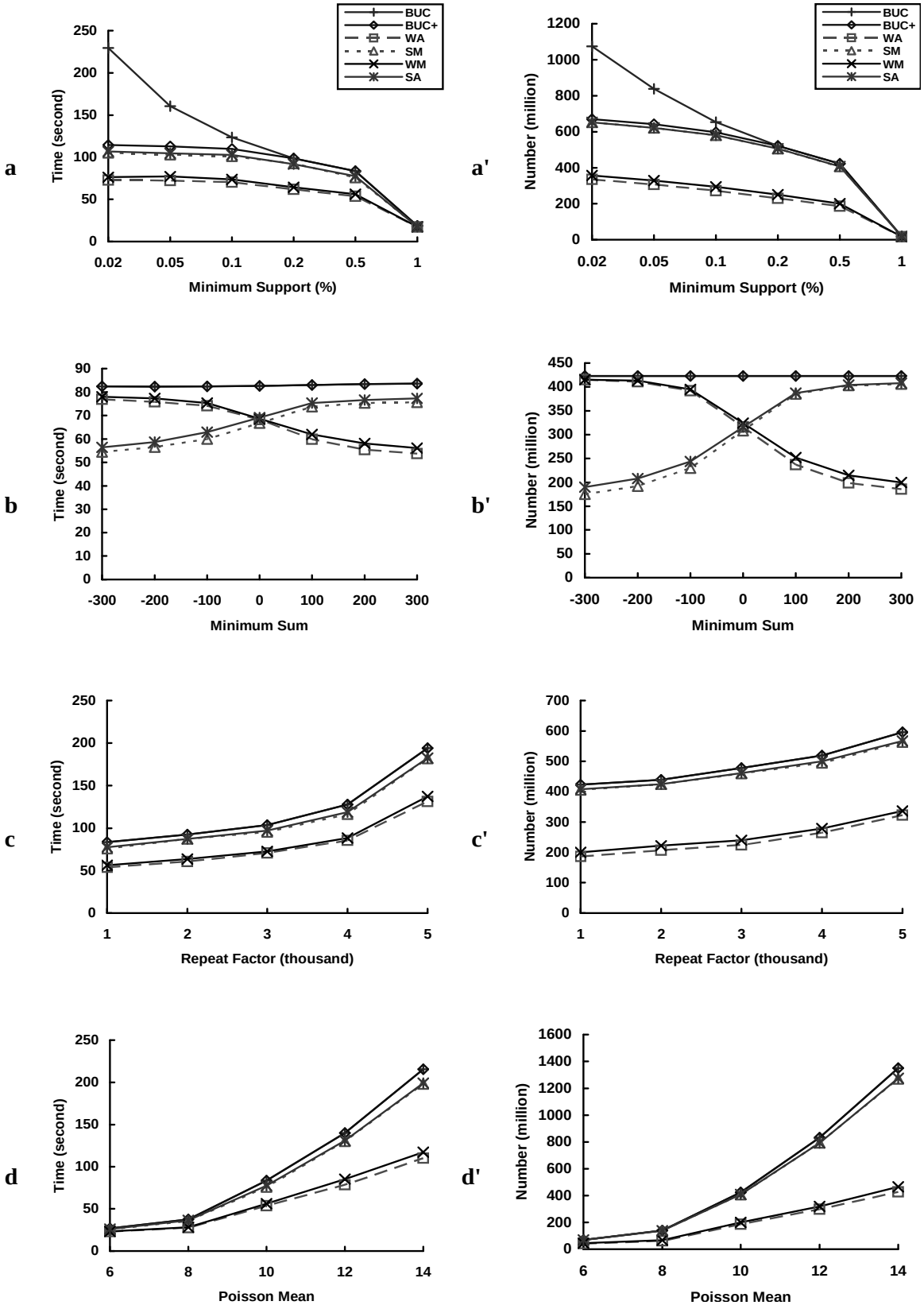

Fig. 4. $sum \geq \sigma$

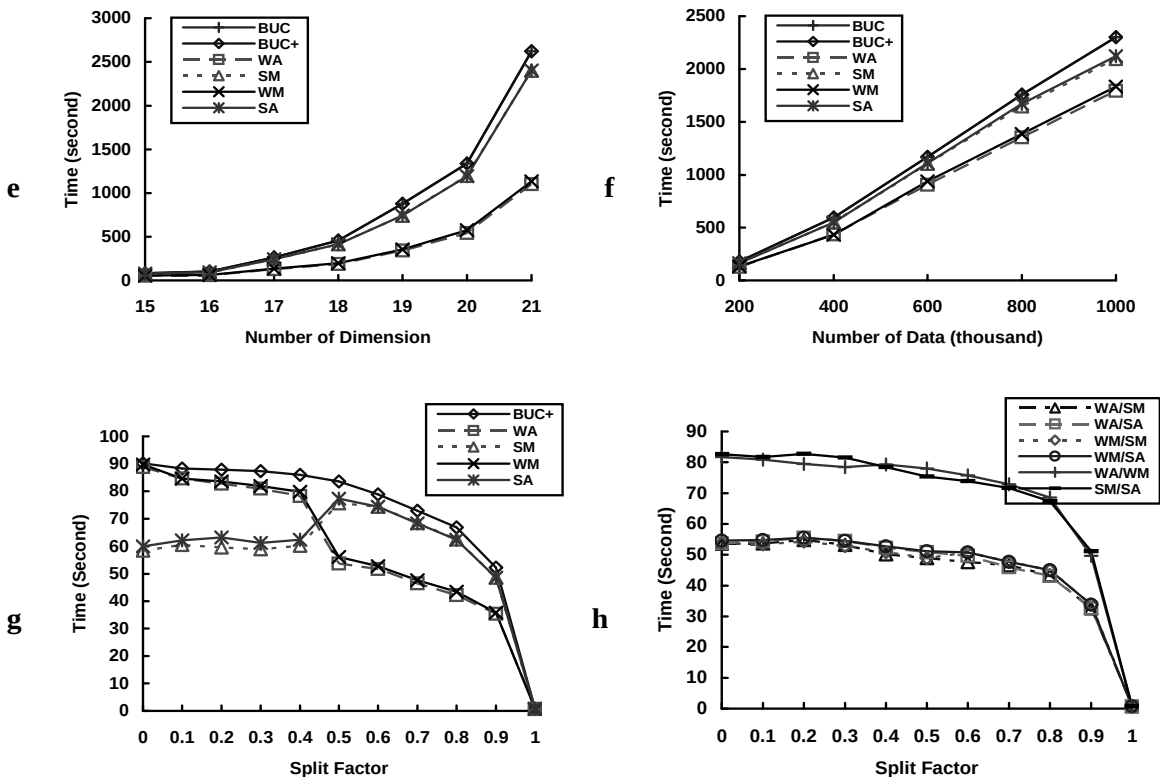

Fig. 5. $sum \geq \sigma$ continued

"homogeneous" approximators.

2) $avg \geq \sigma$*:* The data set in this experiment is exactly same as for $sum \geq \sigma$, except that all measure values are positive. The default minimum average σ is 6, which is 20% higher than the mean 5. The performance was shown in Figure 6, which was quite similar to that for $sum \geq \sigma$. This shows that the pruning is effective for minimum requirements on both types of growth.

B. Experiments on real life data sets

We also experimented on the learning set of the KDD-CUP-98 data set [12]. We chose two measures, 97NK, which represents the donation amount in 1997, and 95NK, which represents the donation amount in 1995. 4,843 tuples have a nonzero value on 97NK, with the range

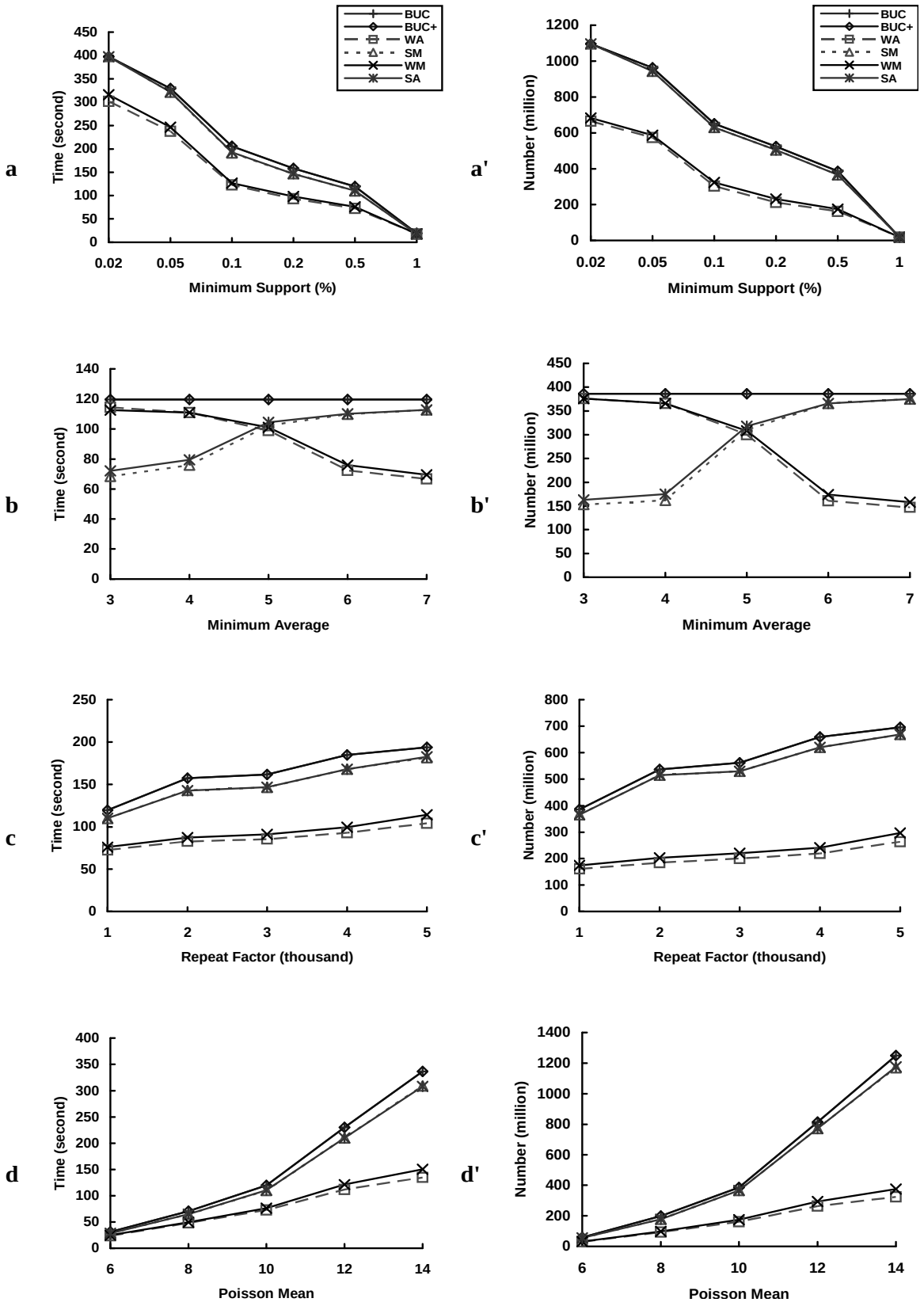

Fig. 6. $avg \geq \sigma$

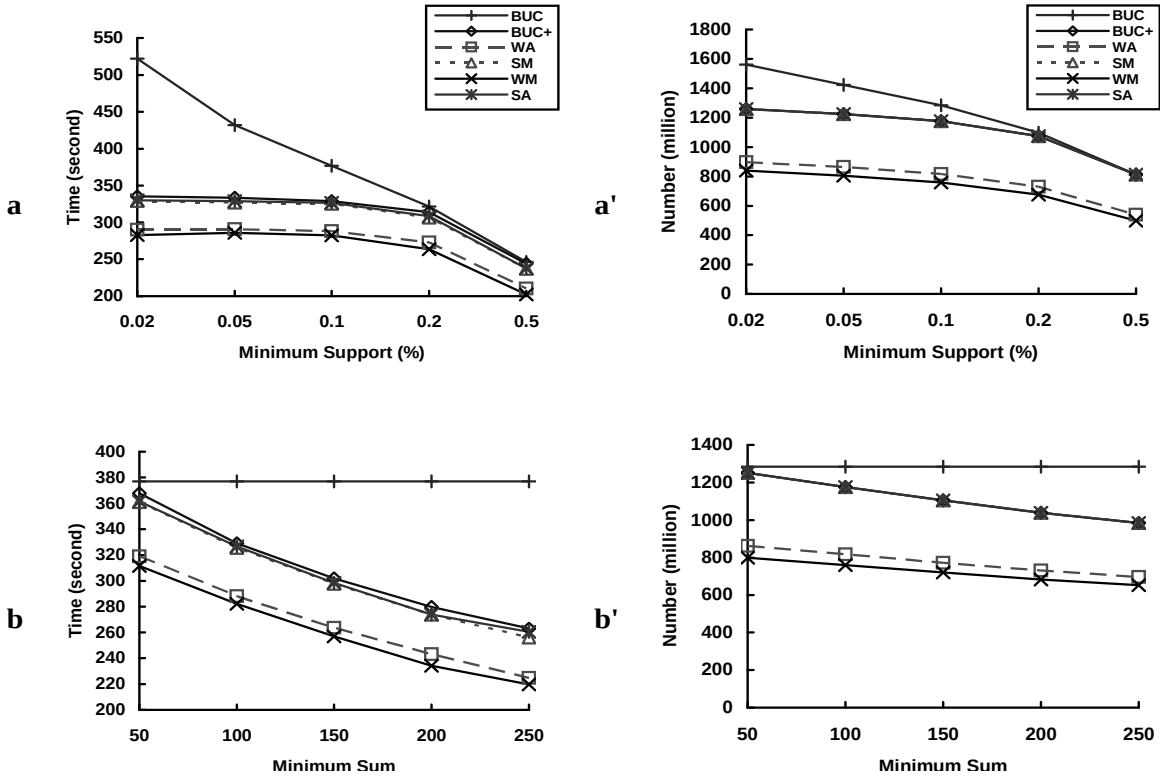

Fig. 7. Experiments on the KDD-CUP-98 data set

$[1, 200]$ and the mean 15.62. 23,317 tuples have a nonzero value on 95NK, with the range of $[1, 200]$ and the mean 13.25. We chose the constraint $sum_1(x) - sum_2(x) \geq \sigma$, where sum_1 computes the sum of 97NK and sum_2 computes the sum of 95NK. This constraint specifies donor's characteristics that improve the donation amount by at least σ. The original data set has 95,412 tuples. After removing all tuples having zero value on both 97NK and 95NK, we have 26,600 remaining tuples. The original data set has 481 dimensions, most of which are not related to the donation amount. We selected the following likely relevant 16 dimensions:

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

The cardinality of each dimension is given in (). MDMAUD and DOMAIN have two or more sub-dimensions, each of which is treated as a dimension.

The full search space at 0% minimum support is $2^{16} \times 26,600 = 1,743,257,600$ tuple examinations. Figure 7(a,a') showed the performance of all algorithms for a range of minimum support, with the minimum sum fixed at 100. Figure 7(b,b') showed the performance for a range of minimum sum, with the minimum support fixed at 0.1%. Compared to the synthetic data set, the improvement of WA and WM over BUC+ was less on this data set. With only 4,843 out of 26,600 tuples having nonzero 97NK donation, sum_1 tends to be small and $sum_1(x) \geq \sigma$ used by BUC+ is somehow sufficient for pruning. SM and SA have a similar performance to BUC+ because this data set did not produce so many satisfying cells to make pruning such cells a big benefit. In fact, most of the 23,317 tuples with nonzero 95NK donation have zero 97NK donation because only 4,843 tuples have nonzero 97NK donation. This situation is similar to a large split factor in Figure 5(g) where more negative measures were generated than positive measures.

C. Summary

The DnA family outperformed BUC+, which outperformed BUC, especially for a small minimum support. Within the DnA family, WM and WA are effective when there are many

failing cells because of a tight constraint. SM and SA are effective when there are many satisfying cells because of a loose constraint. The "heterogeneous" combinations, i.e., WA/SM, WA/SA, WM/SM and WM/SA. could supplement the pruning strength in each case. The "homogeneous" combinations, i.e., WA/WM and SM/SA tend to add more overhead than benefits, due to overlapping of pruning.

VIII. Extension to Boolean Constraints

Often, some Boolean combination of aggregate constraints must be satisfied for interesting cells. A *Boolean constraint* is an expression of aggregate constraints, built using $\neg$ (negation), $\wedge$ (conjunction) and $\vee$ (disjunction). We consider a Boolean constraint in the *conjunctive normal form*, $\mathcal{D}_1 \vee \cdots \vee \mathcal{D}_k$, where each $\mathcal{D}_i = \mathcal{C}_{i1} \wedge \cdots \wedge \mathcal{C}_{iq}$ is a conjunction of one or more aggregate constraints $\mathcal{C}_{ij}$. An example is $(avg(v) \geq \sigma_1) \wedge (var(v) \leq \sigma_2)$, which specifies the cells forming homogeneous and profitable subpopulations by maximum variance and minimum average, respectively. To extend our approach to Boolean constraints, no change is needed in the notion of "weaker than" (Definition 3.1) and various monotonicities of constraints (Definition 3.3). Therefore, the notion of $\alpha\beta$-approximators remains unchanged. Below, we extend the notion of separable constraints.

Definition 8.1: A Boolean constraint $\mathcal{D}_1 \vee \cdots \vee \mathcal{D}_k$ is *separable* (*strongly separable*) if for every $\mathcal{D}_i = \mathcal{C}_{i1} \wedge \cdots \wedge \mathcal{C}_{iq}$, every aggregate constraint $\mathcal{C}_{ij}$ is separable (strongly separable). ■

A sign-space corresponds to one assignment of "+" and "-" signs to each denominator in $\mathcal{C}$ that is not sign-preserved. For a separable Boolean constraint $\mathcal{C} = \mathcal{D}_1 \vee \cdots \vee \mathcal{D}_k$, where $\mathcal{D}_i = \mathcal{C}_{i1} \wedge \cdots \wedge \mathcal{C}_{iq}$, we can obtain the $(\times, /)$-sign-preserved form by applying Theorem 4.1 to each $\mathcal{C}_{ij}$. $(A^+; A^-)$ for each $\mathcal{C}_{ij}$ is determined by Theorem 4.2.

Theorem 8.1: Consider a sign-space of $\mathcal{C}$. Let $\mathcal{C}'_{ij}$ be the $\alpha\beta$-approximator for $\mathcal{C}_{ij}$ constructed

as in Tables II and III. Let $\mathcal{C}'$ be $\mathcal{C}$ with every $\mathcal{C}_{ij}$ replaced with $\mathcal{C}'_{ij}$. Then $\mathcal{C}'$ is a $\alpha\beta$-approximator of $\mathcal{C}$ in the sign-space.

Proof: Let op be $\wedge$ or $\vee$. The theorem follows because (1) if x and y are β-monotone, so is $x\ op\ y$, (2) if x is weaker (stronger) than x' and if y is weaker (stronger) than y', $x\ op\ y$ is weaker (stronger) than $x'\ op\ y'$. ■

Section IV, V, VI are now applicable to Boolean constraints, by constructing $\alpha\beta$-approximators using Theorem 8.1. An interesting question is how this extension affects the effectiveness of Divide-and-Approximate. The study in Section VII provides some insights. Since negation and disjunction tend to relax the constraint, they make pruning satisfying cells more effective. SM and SA would perform better in this case. In contrast, conjunction tightens up the condition, making pruning failed cells more effective. WM and WA would perform better in this case. If both negation/disjunction and conjunction occur, we recommend the "heterogeneous" combinations WM/SM, WA/SA, WA/SM and WM/SA.

IX. Conclusion

Pushing aggregate constraints into iceberg cube mining presents a significant challenge, due to the lack of the "well-behaved" anti-monotonicity or monotonicity. We presented a novel strategy called *Divide-and-Approximate* to address this challenge, by combining two well-known ideas, "divide-and-conquer" and "approximate". This strategy does not depend on the specific form of the f function in the constraint, therefore, is applicable when the constraint is unknown in advance. Experiments showed promising results.

Acknowledgements. We wish to thank the reviewers for helpful comments.

REFERENCES

[1] S. Agarwal, R. Agarwal, and P. M. D. et al. On the computation of multidimensional aggregates. In *VLDB*, 1996.

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

[3] R. Bayardo. Efficient mining long patterns from databases. In *SIGMOD*, pages 85–93, 1998.

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

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

[6] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, 2001.

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

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

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

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[11] C. T. Ho, R. Agrawal, and R. Srihant. Range queries in data cubes. In *SIGMOD*, 1997.

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

[13] 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.

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

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

[16] K. Wang, Y. He, D. Cheung, and F. Chin. Mining confident rules without support requirement. In *CIKM*. ACM, 2001.

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

[18] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD*, 1997.