Journal of Computer and System Sciences **58**, 1–12 (1999)
Article ID jcss.1998.1595, available online at http://www.idealibrary.com on IDEAL®

Mining Optimized Association Rules for Numeric Attributes*

Takeshi Fukuda† and Yasuhiko Morimoto‡

Tokyo Research Laboratory, IBM Research, 1623–14, Shimo-tsuruma, Yamato City, Kanagawa 242, Japan
E-mail: †fukudat@trl.ibm.co.jp; ‡morimoto@trl.ibm.co.jp

Shinichi Morishita

Institute of Medical Science, University of Tokyo, 4-6-1, Shirogane-dai, Minato Ward, Tokyo 108, Japan
E-mail: moris@ims.u-tokyo.ac.jp

and

Takeshi Tokuyama

Tokyo Research Laboratory, IBM Research, 1623-14 Shimo-tsuruma, Yamato City, Kanagawa 242, Japan
E-mail: ttoku@trl.ibm.co.jp

Received November 13, 1996; revised April 14, 1998

Given a huge database, we address the problem of finding association rules for numeric attributes, such as

$$(\mathit{Balance} \in I) \Rightarrow (\mathit{CardLoan} = \mathit{yes}),$$

which implies that bank customers whose balances fall in a range I are likely to use card loan with a probability greater than p. The above rule is interesting only if the range I has some special feature with respect to the interrelation between *Balance* and *CardLoan*. It is required that the number of customers whose balances are contained in I (called the *support* of I) is sufficient and also that the probability p of the condition $\mathit{CardLoan} = \mathit{yes}$ being met (called the *confidence ratio*) be much higher than the average probability of the condition over all the data.

Our goal is to realize a system that finds such appropriate ranges automatically. We mainly focus on computing two *optimized ranges*: one that maximizes the support on the condition that the confidence ratio is at least a given threshold value, and another that maximizes the confidence ratio on the condition that the support is at least a given threshold number.

Using techniques from computational geometry, we present novel algorithms that compute the optimized ranges in linear time if the data are sorted. Since sorting data with respect to each numeric attribute is expensive in the case of huge databases that occupy much more space than the main memory, we instead apply randomized bucketing as the preprocessing method and thus obtain an efficient rule-finding system.

Tests show that our implementation is fast not only in theory but also in practice. The efficiency of our algorithm enables us to compute optimized rules for all combinations of hundreds of numeric and Boolean attributes in a reasonable time. © 1999 Academic Press

* This work is based on the extended abstract of "Mining Optimized Association Rules for Numeric Attributes" in the Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (Montreal, Quebec, Canada, June), ACM, New York, 1996, 182–191.

1. INTRODUCTION

Recent progress in technologies for data input through such media as bar-coded labels, credit cards, OCRs, and cash dispensers has made it easier for finance and retail organizations to collect massive amounts of data and to store them on disk at a low cost. Such organizations are interested in extracting from these huge databases unknown information that inspires new marketing strategies. Current database systems are the primary means of realizing this aim, but in database and AI communities, there has been a growing interest in the efficient discovery of interesting rules, which is beyond the power of current database functions [1–4, 6, 8, 12–15, 17, 19].

1.1. Association Rules

Given a database universal relation, we consider the association rule that if a tuple meets a condition C_1, then it also satisfies another condition C_2 with a certain probability (called a *confidence* in this paper). We will denote such an association rule (or rule, for short) between the presumptive condition C_1 and the objective condition C_2 by $C_1 \Rightarrow C_2$.[1]

We call a rule *exact* if its confidence is 100%. For the purpose of discovering exact or almost exact rules, Piatetsky-Shapiro [14] presents the KID3 algorithm. Important rules in scientific databases are likely to be exact. On the other hand, business databases, such as customer databases and transaction databases, tend to reflect the uncontrolled real

[1] We use the symbol "⇒" in order to distinguish the relationship from logical implication, which is usually denoted by "→."

0022-0000/99 $30.00
Copyright © 1999 by Academic Press
All rights of reproduction in any form reserved.

world, and the confidence of an interesting rule is usually much less than 100%.

Thus, for commercial databases, we should consider a broader class of rules whose confidences are greater than a specified minimum threshold, such as 30%. We call such rules *confident*. Agrawal *et al.* [3] study ways of discovering all confident rules. They focus on rules with conditions that are conjunctions of $(A = yes)$, where A is a Boolean attribute, and present an efficient algorithm. They have applied the algorithm to basket-data-type retail transactions to derive interesting associations between items, such as

$$(Pizza = yes) \wedge (Coke = yes) \Rightarrow (Potato = yes).$$

Improved versions of the algorithm have also been reported [4, 13].

1.2. Optimized Association Rules

In addition to Boolean attributes, databases in the real world usually have numeric attributes, such as age and the balance of account in a database of bank customers. Thus, it is also an important issue to find association rules for numeric attributes. In this paper, we focus on finding a simple rule of the form

$$(Balance \in [v_1, v_2]) \Rightarrow (CardLoan = yes),$$

which states that customers whose balances fall in the range between v_1 and v_2 are likely to take out credit card loans. If an instance of the range is given, the confidence of this rule can be computed with ease. In practice, however, we want to find a range that yields a confident rule. Such a range is called a *confident* range. Unfortunately, a confident range is not always unique, and for instance, we may find a confident range that contains only a very small number of customers.

Let the *support* of a range be the ratio of the number of tuples in the range to the number of all tuples. A range is called *ample* if its support is no less than a given fixed threshold. We want to find a rule associated with a range that is both ample and confident.

In particular, we would like to find the confident range with maximum support, and we call the associated rule an *optimized support* rule. This range captures the largest cluster of customers that are likely to take out credit card loans with a probability no less than the given minimum confidence threshold. Here, we refer to a data set associated with a range of the numeric attribute value as a *cluster*, for short.

Instead of the optimized support rule, it is also interesting to find the ample range that maximizes the confidence. We call the associated rule an *optimized confidence* rule. This range gives us a cluster of more than, for instance, 10% of customers that tend to use card loan with the highest confidence factor. If we want to promote credit card loans by sending direct mail to a fixed number of new customers within a limited budget, this rule gives us useful information on the target customers.

1.3. Main Results

There are trivial ways of computing optimized support rules and optimized confidence rules in $O(N^2)$ time, where N is the number of all tuples. In this paper, we give a non-trivial linear time algorithm for each optimized rule, on the assumption that the data are sorted with respect to the numeric attribute. Each algorithm uses some computational geometry techniques to achieve linear time complexity.

Given the sorted data, out algorithms are asymptotically optimal. Sorting the database, however, could create a serious problem if the database is much larger than the main memory, because sorting data for each numeric attribute would take an enormous amount of time.

To handle giant databases that cannot fit in the main memory, we need to find another way of computing optimized rules. For this purpose, we present algorithms for approximating optimized rules, using randomized algorithms [11]. The essence of these algorithms is that we generate thousands of almost equi-depth buckets[2] and then combine some of those buckets to create approximately optimized ranges. In order to obtain such almost equi-depth buckets, we first create a sample of data that fits into the main memory, thus ensuring the efficiency with which the sample is sorted. Then, we sort the sample and divide it into equi-depth buckets. We show that the buckets in the sample also give almost equi-depth buckets in the original data with high probability.

Tests show that our implementation is fast not only in theory but also in practice. Even for a small case, our algorithm is faster than a naive quadratic-time algorithm by an order of magnitude. The efficiency of our algorithm makes it possible to compute a complete set of optimized rules for all combinations of hundreds of numeric and Boolean attributes in a reasonable time. We present some performance results.

1.4. Extensions of Optimized Association Rules

We have been focusing on rather simple rules of the form $(A \in [v_1, v_2]) \Rightarrow C$, but our algorithms can be straightforwardly extended to generate rules of the form $(A \in [v_1, v_2]) \wedge C_1 \Rightarrow C_2$, where C_1 and C_2 are Boolean statements that do not contain any uninstantiated ranges on numeric attributes.

[2] Buckets are called (almost) *equi-depth* if tuples are (almost) uniformly distributed into buckets.

Another interesting application of the algorithms for computing optimized association rules is to generate efficiently a range in a numeric attribute that maximizes the average of values in another attribute. For example, bankers are interested in customers whose saving account balances are very high. They therefore would like to know the range I of ages that maximizes the average of saving account balances of customers in I under the condition that I contains an ample number (no less than a given threshold) of customers. We will discuss this problem in Section 5.

It would also be valuable to extend our framework to rules with two numeric attributes in the presumptive condition, and to find the region in the two-dimensional space of these attributes that represents a nice association rule between these two numeric attributes and the conclusion. For instance, we would like to find a rule such as

$$(Age, Balance) \in X \Rightarrow (CardLoan = yes),$$

where X is a rectangle or a connected region in two-dimensional space of *Age* and *Balance*. Optimized rules can also be naturally defined in this extension. While the problem of finding the optimal arbitrary connected region is NP-hard, the authors present practical solutions for the cases where the regions are rectangular, x-monotone, and rectilinear-convex in related papers [7, 20].

1.5. Related Work

Some other work has been done on handling numeric attributes. Piatetsky-Shapiro [14] studies how to sort the values of a numeric attribute, divide the sorted values into approximately equi-depth ranges, and use only those fixed ranges to derive *exact* rules. Other ranges except for the fixed ones are not considered in his framework. Our method is not only capable of outputting optimized ranges, but is also more convenient than Piatetsky-Shapiro's method, since we need not make candidate ranges beforehand. Recently Srikant and Agrawal [18] have improved Piatetsky-Shapiro's method by considering combinations of some consecutive ranges. A combined range could be the whole range of the numeric attribute, which produces a trivial rule. To avoid this, Srikant and Agrawal present an efficient way of computing a combined range whose size is at most a threshold given by the user. Although Srikant and Agrawal's approach does not output optimized ranges, it can generate not only ranges but also interesting rectangular regions and hypercubes.

The association rule is a fundamental tool to construct efficient decision trees. If we consider a decision tree on a database containing numeric attributes, we need a method to compute good association rules. ID3 [17], CART [6], CDP [3], and SLIQ [9] perform binary partitioning of numeric attributes repeatedly until each range contains data of one specific group (or several groups, in some cases) with high probability, while IC [1] uses k decomposition. Our optimized association rule is a powerful substitute to those known methods, and our subsequent paper [10] gives such a construction of efficient decision trees.

2. PRELIMINARIES

2.1. Association Rules

DEFINITION 2.1. Let R be a relation. To describe conditions on tuples in R, we use *primitive* conditions. For a Boolean attribute A, $A = yes$ and $A = no$ are primitive conditions. For a numeric attribute A, $A = v$ and $A \in [v_1, v_2]$ are primitive conditions. Let t be a tuple in R, and let $t[A]$ denote the t's value for the attribute A. t *meets* $A = v$ if $t[A]$ is equal to v. t meets $A \in [v_1, v_2]$ if $t[A]$ belongs to the interval $[v_1, v_2]$. In order to describe more complicated conditions, we use also conjunctions of primitive conditions.

EXAMPLE 2.1. Consider a relation for retail transactions. Each attribute of the relation is a Boolean one whose domain is $\{yes, no\}$, and represents an item, such as *Coke* or *Pizza*. $(Coke = yes) \wedge (Pizza = yes)$ is a condition, and a tuple that meets the condition represents a customer who purchased a Coke and a Pizza.

EXAMPLE 2.2. Consider a relation for data on a bank's customers. Suppose that each tuple contains the balance of account and services (card loan or automatic withdrawal, say) for one customer. An example of a condition is

$$(Balance \in [15821, 26264]) \wedge (Cardloan = yes).$$

DEFINITION 2.2. The *support* of condition C is defined as the percentage of tuples that meet condition C, and is denoted by $support(C)$.

For instance, in the retail relation given in Example 2.1, if $support(Coke = yes) = 10\%$, then 10% of customers purchase a Coke.

DEFINITION 2.3. Let C_1 and C_2 be conditions on tuples. An *association rule* (or *rule*, for short) has the form $C_1 \Rightarrow C_2$. The *confidence* of rule $C_1 \Rightarrow C_2$ is defined as $support(C_1 \wedge C_2)/support(C_1)$, which we will denote by $conf(C_1 \Rightarrow C_2)$.

For instance, in the bank relation in Example 2.2, suppose that the confidence of the rule

$$(Balance \in [15821, 26264]) \Rightarrow (Cardloan = yes)$$

is 50%; then 50% of customers whose balances fall in the range use credit card loans.

2.2. Optimized Association Rules

Throughout this paper, we focus on mining association rules of the form $(A \in [v_1, v_2]) \Rightarrow C$. Suppose that A and C are fixed.

DEFINITION 2.4. A rule is *confident* if its confidence is not less than the given minimum confidence threshold. Among confident rules, an *optimized support rule* maximizes $support(A \in [v_1, v_2])$. A rule is *ample* if $support(A \in [v_1, v_2])$ is not less than the given minimum support threshold. Among ample rules an *optimized confidence rule* maximizes the confidence.

EXAMPLE 2.3. Consider rules of the form

$$(Balance \in [v_1, v_2]) \Rightarrow (CardLoan = yes).$$

Suppose that 50% is given as the minimum confidence threshold. We may have many instances of ranges that yield confident rules, such as

Range	[1000, 10000]	[5000, 5500]	[500, 7000]
Support	20%	2%	15%
Confidence	50%	55%	52%

Among those ranges, [1000, 10000] is a candidate range for an optimized support rule. Next, given 10% as the minimum support threshold, we may also have many ample rules, such as

Range	[1000, 5000]	[2000, 4000]	[3000, 8000]
Support	13%	10%	11%
Confidence	65%	50%	52%

The reader might feel it strange that although [1000, 5000] is a superset of [2000, 4000], the confidence of the rule of the former range is greater than that of the latter range, but observation will confirm that corresponding situations could really occur.

2.3. Buckets

DEFINITION 2.5. Let t be a tuple of the given relation R, and let $t[A]$ denote the value of the attribute A of t. *Buckets* of the domain of A are a sequence of disjoint ranges

$$B_1, B_2, ..., B_M$$
$$(B_i = [x_i, y_i] \text{ and } x_i \leqslant y_i < x_{i+1})$$

such that the value of A for all tuples is covered by the buckets; namely, for an arbitrary tuple $t \in R$, there exists a bucket B_j that contains $t[A]$. We say that a bucket B_i is *finest* if $B_i = [x, x]$ for a value x.

EXAMPLE 2.4. If A represents age and the domain of A is non-negative integers bounded by 120, we can make 121 finest buckets $[i, i]$ for each $i = 0, 1, ..., 120$. When A shows the balances of millions of customers in a bank, the domain of A may range from \$0 to \$$10^{10}$. In this case the number of finest buckets may amount to millions.

Linking consecutive buckets $B_s, B_{s+1}, ..., B_t$ creates a range $[x_s, y_t]$. Observe that if all buckets are finest, the combination of consecutive finest buckets gives the range of an optimized association rule. Given a large number (thousands, say) of buckets that may not be finest, an approximation of the range of an optimized rule can be obtained by joining consecutive buckets. Thus as ranges of rules we only use those that consist of consecutive buckets.

DEFINITION 2.6. We call the number of tuples in $\{t \in R \mid t[A] \in B_i\}$ the *size* of B_i and denote this by u_i. We assume that each bucket B_i contains at least one tuple, that is, $u_i \geqslant 1$. B_i's are called *equi-depth* if the size of any B_i is the same. Let v_i denote the number of tuples in $\{t \in R \mid t[A] \in B_i, t \text{ meets } C\}$, and let N be the number of all tuples. $(\sum_{i=s}^{t} v_i)/(\sum_{i=s}^{t} u_i)$ gives the confidence of rule $(A \in [x_s, y_t]) \Rightarrow C$, and the support of $A \in [x_s, y_t]$ is $(\sum_{i=s}^{t} u_i)/N$.

To compute u_i and v_i, for each tuple t, we need to determine the bucket that $t[A]$ belongs to. One natural way of doing this is to scan each tuple once and locate the bucket to which the tuple belongs by using a hash function or by building an ordered binary tree of finest buckets. This technique works fast for a huge database if the number of finest buckets is small (recall the case of age in Example 2.4), but it may run very slowly if it has to handle millions of finest buckets, owing to the limited size of the main memory. Another natural method is to sort the given relation over A and divide the sorted data into finest buckets, but it takes an enormous amount of time to sort a giant database that is much larger than the main memory.

The above discussion shows that the most difficult case is that in which the number of finest buckets is large and the size of the given database is huge. One such example may be the balances of millions of customers in a bank. In this case, for the sake of efficiency, we should avoid sorting a huge database and reduce the number of buckets to be considered. Thus our approach is to generate a small number (say thousands) of buckets which may not be finest, instead of making millions of finest buckets. We will make almost equi-depth buckets so that we can make good approximations of optimized rules. In the next section we present a way of making almost equi-depth buckets without sorting the data.

3. MAKING EQUI-DEPTH BUCKETS

We present a way of dividing N data into M buckets almost evenly.

3.1. Algorithm

Since we must avoid sorting data, as mentioned in the previous section, we use the following approximation algorithm:

ALGORITHM 3.1.

1. Make an S-sized random sample from N data.
2. Sort the sample in $O(S \log S)$ time.
3. Scan the sorted sample and set the $i(S/M)$th smallest sample to p_i for each $i = 1, ..., M-1$. Let p_0 be $-\infty$ and p_M be $+\infty$.
4. For each tuple x in the original N data, find i such that $p_{i-1} < x \leqslant p_i$ and assign x to the ith bucket. This check can be done in $O(\log M)$ time by using the binary search tree for the buckets. Thus, for all i, the size u_i of B_i can be computed in $O(N \log M)$ time.

The complexity of this algorithm is

$$O(\max(S \log S, N \log M)).$$

In practice, $S \ll N$, and hence the complexity is $O(N \log M)$. We evaluated the performance of this algorithm with very large sets of data (containing up to ten million tuples) and found that the computation time grows almost linearly in proportion to the data size. See Subsection 6.1.

3.2. Sample Size

How many samples are enough to generate almost equi-depth buckets? Let S be the sample size and I be an interval that contains N/M original data. Let X denote the number of sample points that belongs to I. Since we pick each sample point independently and uniformly at random with replacement from the original data, the probability that a sample point belongs to I is $1/M$. Hence, X follows a binomial distribution, $B(S, 1/M)$, and therefore, we can compute the following probability for δ and M by using the tail probability of the binomial distribution:

$$p_e = \Pr(|X - S/M| \geqslant \delta S/M).$$

Note that p_e does not depend on the number of tuples, N. Figure 1 shows the relationship between p_e and S/M for $\delta = 0.5$ and $M = \{5, 10, 10000\}$. For every value of M, p_e goes down sharply when $S/M < 40$. It becomes smaller than 0.3% when $S/M = 40$, and it does not decrease much when $S/M > 40$. Thus, in our implementation we use $40 \cdot M$ as S.

In practice, a value of at most $M = 10^4$ is precise enough to allow us to derive approximate rules, and a (4×10^5)-sized sample fits into the main memory. Subsection 6.1 presents performance results for Algorithm 3.1 and a naive method using Quick Sort.

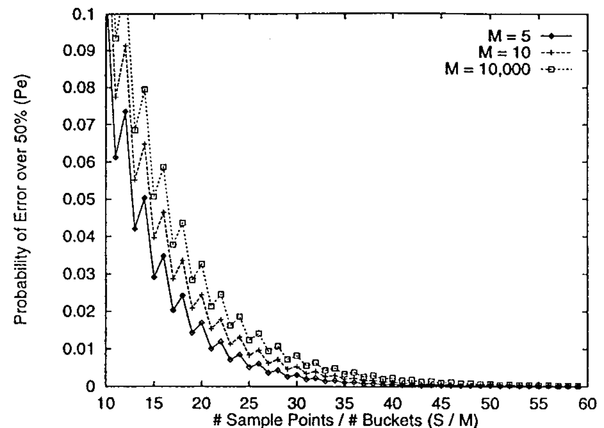

FIG. 1. Sample size and probability of the error being over 50%.

3.3. Parallel Bucketing

The most time-consuming part of Algorithm 3.1 is Step 4, which scans the entire database to find buckets for all tuples. Because we want to know only the size of each bucket, we can easily perform Step 4 in parallel:

ALGORITHM 3.2.

1. Randomly distribute the tuples in the database to processor elements (PEs) almost evenly.
2. At a coordinating PE, execute Steps 1, 2, and 3 of Algorithm 3.1.
3. At each PE, perform Step 4 of Algorithm 3.1; namely, scan the divided data and count the number of data in each bucket.
4. Gather the results from all PEs to the coordinating PE and compute their sum.

No communication is necessary during the counting process, and hence we expect that this algorithm will be scalable to the number of PEs.

3.4. Number of Buckets

If all buckets are finest, every possible range can be expressed by connecting some subsequence of the buckets.

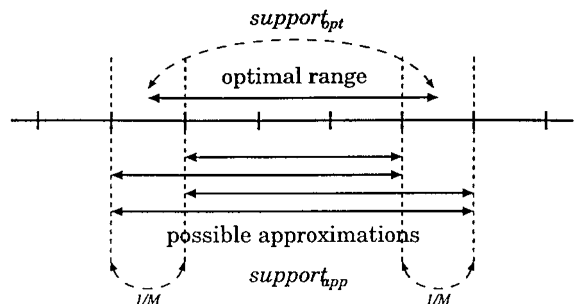

FIG. 2. Approximation by buckets.

TABLE I

Error Range of Approximation Depending on the Number of Buckets

No. of buckets	$support_{app}$	$conf_{app}$
10	$10.0\% \cdots 50.0\%$	$42.0\% \cdots 100\%$
50	$26.0\% \cdots 34.0\%$	$59.2\% \cdots 80.8\%$
100	$28.0\% \cdots 32.0\%$	$65.6\% \cdots 75.0\%$
500	$29.6\% \cdots 30.4\%$	$69.1\% \cdots 70.9\%$
1,000	$29.8\% \cdots 30.2\%$	$69.5\% \cdots 70.5\%$

Note. $support_{opt} = 30\%$, $conf_{opt} = 70\%$

Otherwise, the optimal range that we want to compute may not be generated from the buckets.

Let us consider the possible error range caused by the granularity of buckets. Assume that we have M equi-depth buckets.[3] As shown in Fig. 2, the optimal range will be replaced by one of four possible approximate ranges. Let $support_{opt}$ and $conf_{opt}$ be the support and confidence of the optimal range, and let $support_{app}$, and $conf_{app}$ be those of an approximate range. Since the support of each bucket is $1/M$, we can bind the error of an approximation as

$$\frac{|support_{app} - support_{opt}|}{support_{opt}} \leqslant \frac{2}{M\, support_{opt}}$$

$$\frac{|conf_{app} - conf_{opt}|}{conf_{opt}} \leqslant \frac{2}{M\, support_{opt} - 2}.$$

For example, when the support of the optimal range is 30% and its confidence is 70%, Table I shows how the support and confidence of an approximate range depend on the number of buckets. Observe that the approximation may contain significant error when the number of buckets is small. To make the error negligible, the number of buckets should be much larger than $1/support_{opt}$.

4. ALGORITHMS

As explained in Subsections 2.3 and 3.4, if all buckets are finest or if plenty of buckets are given, we can focus on rules whose ranges are combinations of consecutive buckets, and therefore we give algorithms for computing optimized rules among such rules. Precisely, given a sequence of buckets $B_1, B_2, ..., B_M$, such that $B_i = [x_i, y_i]$ and $x_i \leqslant y_i < x_{i+1}$, we focus on rules of the form

$$(A \in [x_s, y_t]) \Rightarrow C,$$

where $[x_s, y_t]$ is a combination of consecutive buckets $B_s, B_{s+1}, ..., B_t$.

Definition 4.1. Since any range $[x_s, y_t]$ is specified by a pair of indexes $s \leqslant t$, for simplicity, we denote $support(A \in [x_s, y_t])$ by $support(s, t)$ and denote $conf((A \in [x_s, y_t]) \Rightarrow C)$ by $conf(s, t)$ throughout this section.

4.1. Optimized Confidence Rules

Definition 4.2. Let $B_1, ..., B_M$ be buckets. Let N denote the number of all tuples. Let u_i denote the number of tuples in $\{t \in R \mid t[A] \in B_i\}$. We assume that $u_i \geqslant 1$. Let v_i denote a real number associated with B_i. Consider the sequence of points $Q_k = (\sum_{i=1}^{k} u_i, \sum_{i=1}^{k} v_i)$ for $k = 1, ..., M$, and let Q_0 be $(0, 0)$. Let m and n be non-negative integers such that $m < n$. Observe that the x-coordinate of Q_n minus the x-coordinate of Q_m is equal to $N \times support(m+1, n)$.

We call $s \leqslant t$ an *ample* pair if $support(s, t)$, which is $\sum_{i=s}^{t} u_i/N$, is no less than the given minimum support threshold. We call m and n an *optimal slope pair*, if $m+1$ and n are an ample pair that maximizes the slope of $Q_m Q_n$. If more than one pair has the same maximum slope, select a pair that maximizes $support(m+1, n)$.

In the special case that v_i is the number of tuples in $\{t \in R \mid t[A] \in B_i, t \text{ meets } C\}$, the slope of the line $Q_m Q_n$ gives $conf(m+1, n)$. Thus, if m and n are an optimal slope pair, $(A \in [x_{m+1}, y_n]) \Rightarrow C$ is an optimized confidence rule. We will therefore present an algorithm for computing an optimal slope pair.

To compute an optimal slope pair, we use a technique of handling convex hulls, for which we introduce some special terms.

Definition 4.3. Let S be a set of distinct points. A *convex polygon* of S has the property that any line connecting any two points of S must itself lie entirely inside the polygon. The *convex hull* of S is the smallest convex polygon of S. Let v_{min} be the node in S with the minimum x-coordinate, and let v_{max} be the node in S with the maximum x-coordinate. Observe that v_{min} and v_{max} are on the convex hull of S. From v_{min} we can visit nodes on the convex hull of S in clockwise (counterclockwise) order until we hit v_{max}, and we call the set of nodes visited the *upper* (*lower*) hull of S.

Let U_m denote the upper hull of $\{Q_m, ..., Q_M\}$, and let $r(m)$ be

$$\min\{i \mid m+1 \leqslant i \text{ is an ample pair}\}.$$

Now consider the tangent of Q_m and $U_{r(m)}$, and suppose that the tangent touches $U_{r(m)}$ at Q_t as illustrated in Fig. 3. Q_t is called the *terminating* point of the tangent (if the tangent touches more than one node of $U_{r(m)}$, select the node with the maximum x-coordinate as Q_t).

[3] Using equi-depth buckets minimizes the possible error of approximations for any fixed number of buckets, since other bucketing methods will produce a larger bucket than $1/M$.

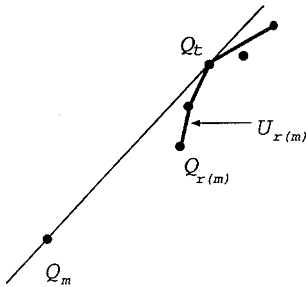

FIG. 3. The inner tangent of Q_m and $U_{r(m)}$.

It is easy to see that if $m \leqslant n$ is an optimal slope pair, Q_n is the terminating point of the tangent of Q_m and $U_{r(m)}$. Thus, we need to find the tangent of Q_m and $U_{r(m)}$ with the maximum slope among all m. To this end, we will present an algorithm whose computational complexity is linear in the number of buckets.

Online Maintenance of Convex Hulls

We present Algorithm 4.1, which constructs a data structure that represents *the convex hull tree* of $Q_0, ..., Q_M$, such as illustrated in Fig. 4. While various implementations are possible [16], we use stacks S and $D_i (i = 0, ..., M)$ in the data structure. We use S to store the sequence of nodes of the convex hull that we are focusing on, and we use D_i to store a branch of the convex hull tree, namely, the nodes that belong to U_{i+1}, but do not belong to U_i. Algorithm 4.1 consists of a preparatory phase and a restoration phase. Given a sequence of nodes $Q_0, ..., Q_M$ that is the sorted list with respect to the x-coordinate value, the preparatory phase sets each branch of the convex hull tree to D_i, for $i = M-1, ..., 0$. The restoration phase makes $U_{r(m)}$ on S, for each $m = 0, ..., M-1$, using D_i's.

ALGORITHM 4.1. Suppose that we are given a sequence of nodes $Q_0, Q_1, ..., Q_M$ that is the sorted list with respect to the x-coordinate value. Let S and $D_i (i = 1, ..., M)$ be empty.

Preparatory Phase: For each $i = M, ..., 0$, we perform the following step so that, after the execution of each step, the top-to-bottom order of nodes in S corresponds to the clockwise order of nodes on U_i (the upper hull of $\{Q_i, ..., Q_M\}$), which enables us to access the neighbors of each node Q on U_i by looking at the next node and the previous node of Q in S.

> Initially, when $i = M$, push Q_M onto S, which trivially makes S store U_M. Otherwise, visit each node Q on U_{i+1} from Q_{i+1} in clockwise order (visit each node in S in top-to-bottom order), and find the Q_iQ with the maximum slope. This search is done by performing the following procedure:
>
> > Clockwise Search: If the slope of Q_i and the top node of S is less than or equal to the slope of Q_i and the node that is second from the top of S, the top node is no longer a node on U_i; in this case, pop the top node from S, push it onto D_i, and repeat the check. Otherwise, the slope of Q_i and the top node is maximum; in this case, push Q_i onto S.

D_i's are used for recording the nodes deleted at each step. Since at most $M-1$ nodes are popped from S in the above check, the check is executed at most $2(M-1)$ times, and hence the time and space complexities of clockwise search are $O(M)$. After the termination, S stores U_0.

Restortation Phase: Assume that S stores U_0. Make S contain U_1 by popping the top node Q_0 from S and pushing back all nodes of D_0 onto S. Set 1 to i. We use i to search for $r(m)$ for each $m = 0, ..., M-1$. Now for each $m = 0..., M-1$, perform the following procedure so that S store $U_{r(m)}$:

> While $(m+1, i)$ is not an ample pair, pop the top node Q_i from S, push back all nodes of D_i in top-to-bottom order onto S, which makes S store U_{i+1}, and increment i. If $i > M$, we stop the restoration phase.

Observe that, after the execution of the above while statement, i contains $r(m)$, and hence S stores $U_{r(m)}$. Since throughout the execution, at most M nodes are popped from S, and at most M nodes are pushed back from D_i's to S, the overall computation time is $O(M)$.

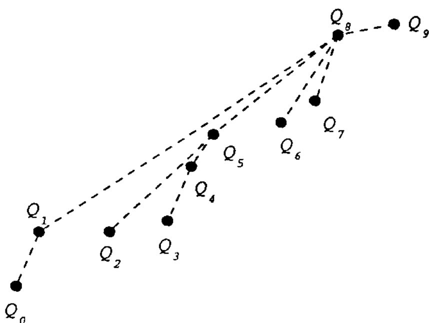

FIG. 4. Upper hulls.

EXAMPLE 4.1. Consider nodes $Q_0, Q_1, ..., Q_9$ in Fig. 4. The dotted line from Q_i to Q_9 shows the upper hull of $\{Q_i, ..., Q_9\}$. Let us apply the preparatory phase of Algorithm 4.1 to $\{Q_0, ..., Q_9\}$. Each column of the upper table in Fig. 5 illustrates the content of S for each $i = 9, ..., 0$. Observe that each column contains the upper convex hull of $\{Q_i, ..., Q_9\}$. Each column of the lower table in Fig. 5 shows D_i for $i = 9, ..., 0$. We can also see how the restoration phase works by observing the columns from $i = 0$ to 9.

							Q_3			
						Q_4	Q_4	Q_2		Q_0
			Q_7	Q_6	Q_5	Q_5	Q_5	Q_5	Q_1	Q_1
		Q_8	Q_8	Q_8	Q_8	Q_8	Q_8	Q_8	Q_8	Q_8
	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9
$i =$	9	8	7	6	5	4	3	2	1	0

							Q_4	Q_5	
			Q_7	Q_6			Q_3	Q_2	
D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

FIG. 5. Computing upper hulls.

FIG. 6. Leaving L untouched.

Computing Tangents

Now we are in a position to present the algorithm for computing the tangent of Q_m and $U_{r(m)}$ with the maximum slope among all m.

ALGORITHM 4.2. Given $Q_0, Q_1, ..., Q_M$, perform the preparatory phase of Algorithm 4.1, so that in what follows, we can obtain $U_{r(m)}$ for each $m = 0, ..., M$ by executing the restoration phase of Algorithm 4.1. We use variable L to store the tangent of Q_m and $U_{r(m)}$ for some m.

Base Step: Compute $U_{r(0)}$ by using the restoration phase of Algorithm 4.1. Find the terminating point of the tangent of Q_0 and $U_{r(0)}$ by clockwise search; that is, visit each node Q on $U_{r(0)}$ from $Q_{r(0)}$ in clockwise order, find the $Q_0 Q$ with the maximum slope, and set the tangent to L.

Inductive Step: For each $m = 1, ..., M$, while $U_{r(m)}$ is not empty, compute $U_{r(m)}$ by using the restoration phase of Algorithm 4.1, assume that L stores the tangent of Q_k and $U_{r(k)}$ for some $k < m$, and perform one of the following steps:

- If Q_m is above or on L, the slope of the tangent of Q_m and $U_{r(m)}$ is not greater than that of L. Figure 6 illustrates an example of this case. We do not compute the tangent of Q_m and $U_{r(m)}$, and leave L untouched.

- Otherwise, let Q_t be the terminating point of L, compute the tangent of Q_m and $U_{r(m)}$ by one of the following steps, and set the tangent to L:

  — If L does not touch $U_{r(m)}$ (Q_t is on the left-hand side of $Q_{r(m)}$), find the terminating point of Q_m and $U_{r(m)}$ by clockwise search; that is, visit each node Q on $U_{r(m)}$ from $Q_{r(m)}$ in clockwise order, and find the $Q_m Q$ with the maximum slope. Figure 7 illustrates this case.

  Among all nodes on both $U_{r(k)}$ and $U_{r(m)}$, let X denote the node with the minimum x-coordinate. The above clockwise search only scans edges from $Q_{r(m)}$ to at most X; otherwise, $U_{r(k)}$ cannot be convex, since the terminating point is above $Q_m X$, and at least one node on the left-hand side of X on $U_{r(k)}$ is also above or on $Q_m X$. Since edges between $Q_{r(m)}$ and X are hidden inside $U_{r(k)}$, those edges have never been scanned in this algorithm.

  — Otherwise, L touches $U_{r(m)}$ at Q_t. Find the terminating point of Q_m and $U_{r(m)}$ by counterclockwise search; that is, visit each node Q on $U_{r(m)}$ from Q_t in counterclockwise order, and find the $Q_m Q$ with the maximum slope. Figure 8 illustrates this case.

  Note that edges between $Q_{r(m)}$ and Q_t have never been scanned before in this algorithm.

Final Step: Among all tangents that have been set to L, select the ones with the maximum slope.

FIG. 7. Clockwise search.

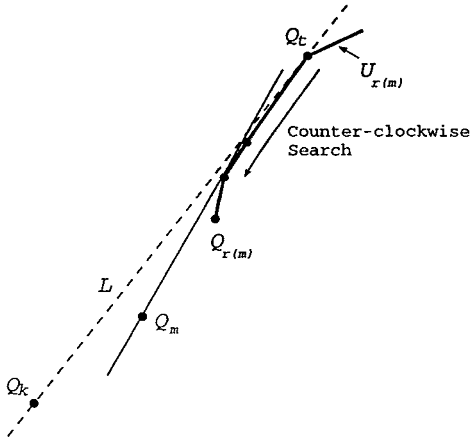

FIG. 8. Counterclockwise search.

THEOREM 4.1. *Algorithm* 4.2 *computes all optimal slope pairs in* $O(M)$ *time.*

Proof. Let S denote the set of edges on $U_{r(m)}$ for all m. Both the clockwise search and the counterclockwise search in the algorithm scan each edge in S at most once. Since the number of edges in S is at most $M-1$, the algorithm computes tangents with the maximum slope in $O(M)$ time. ∎

4.2. Optimized Support Rules

DEFINITION 4.4. Let $B_1, ..., B_M$ be buckets such that $B_i = [x_i, y_i]$ and $x_i \leqslant y_i < x_{i+1}$. Let N denote the number of all tuples. Let u_i denote the number of tuples in $\{t \in R \mid t[A] \in B_i\}$. Let v_i denote a real number associated with B_i. Let s and t be non-negative integers such that $s \leqslant t$. Let $avg(s, t)$ denote $(\sum_{i=s}^{t} v_i)/(\sum_{i=s}^{t} u_i)$. Let θ be a minimum threshold for $avg(s, t)$. We call (s, t) an *optimal support pair* if $avg(s, t) \geqslant \theta$, and (s, t) maximizes $support(s, t)$ $(=\sum_{i=s}^{t} u_i/N)$.

In the special case that v_i is the number of tuples in $\{t \in R \mid t[A] \in B_i, t \text{ meets } C\}$, $avg(s, t)$ is equal to $conf(s, t)$. Suppose that θ is the minimum confidence threshold. If (s, t) is an optimal support pair, $(A \in [x_s, y_t]) \Rightarrow C$ is an optimized support rule.

We will now present an algorithm for computing an optimal support pair.

DEFINITION 4.5. Let us call *s effective* if $avg(j, s-1) < \theta$ for every $j < s$.

LEMMA 4.1. *If* $s \leqslant t$ *is an optimal support pair, s is effective.*

Proof. Otherwise, there exists j such that

$$avg(j, s-1) \geqslant \theta.$$

Since $s \leqslant t$ is an optimal support pair, $avg(s, t) \geqslant \theta$, and hence $avg(j, t) \geqslant \theta$, which contradicts the optimality of s and t. ∎

From the above lemma, we will find all effective indices and choose an optimal support pair. Let w be $\max_{j>s} \sum_{i=j}^{s-1} (v_i - \theta u_i)$ for each index s. Then, note that s is effective iff $w < 0$. The following algorithm computes w for all indices in $O(M)$ by scanning buckets forward and gives the set of all effective indices.

ALGORITHM 4.3.

```
1 is effective;
w := 0;
for each s := 2 to M begin
    w := v_{s-1} − θu_{s-1} + max{0, w};
    if (w < 0) then s is effective
end.
```

Let $top(s)$ denote the largest index t, such that $s \leqslant t$ and $avg(s, t) \geqslant \theta$. The final step is to choose a value of s that maximizes $\sum_{i=s}^{top(s)} u_i$.

LEMMA 4.2. *If* $s < s'$ *are effective,* $top(s) \leqslant top(s')$.

Proof. Since s' is effective, $avg(s, s'-1) < \theta$. From the definition of $top(s)$, $avg(s, top(s)) \geqslant \theta$. Then, it follows that $avg(s', top(s)) \geqslant \theta$, which implies that $top(s) \leqslant top(s')$. ∎

Thanks to this property, we only need to scan backward through the list of effective indices $(s(1), ..., s(q))$ and the list of all indices $(1, ..., M)$ alternately to find $top(s(i))$. We can do this by means of the following algorithm:

ALGORITHM 4.4.

```
i := M;
for each j from q to 1 begin
    while (avg(s(j), i) < θ) do begin
        i := i − 1;
    end;
    top(s(j)) := i
end.
```

In the above algorithm, we precompute a cumulative table $F(j) = \sum_{i=1}^{j} v_i - \theta \sum_{i=1}^{j} u_i$. Since

$$avg(s(j), i) < \theta \quad \text{iff} \quad F(i) - F(s(j)-1) < 0,$$

we can check $avg(s(j), i) < \theta$ in a constant time ($F(0)$ is defined as 0). Thus both Algorithms 4.3 and 4.4 run in $O(M)$ time, and we have:

THEOREM 4.2. *All optimal support pairs can be computed in* $O(M)$ *time.*

In the algorithm literature, Bentley [5] introduced a linear-time algorithm (Kadane's algorithm), which unfortunately does not work for finding the optimized rules. Kadene's algorithm computes a range I that maximizes $\sum_{i \in I} x_i$ against an array of real numbers x_i. If every x_i is non-negative, trivially $[1, M]$ gives the solution, so we assume that some elements are negative. Let $a(j)$ denote $\max\{\sum_{i \in [s, t]} x_i \mid s \leqslant t \leqslant j\}$; then the interval of $a(M)$ is our answer. To compute $a(j)$ we introduce another auxiliary data item $b(j)$ that denotes $\max\{\sum_{i \in [s, j]} x_i \mid s \leqslant j\}$. Then, the following relations hold:

$$b(j+1) = \max\{0, b(j)\} + x_{j+1}$$
$$a(j+1) = \max\{b(j+1), a(j)\}.$$

Set 0 to $b(0)$. Then, a simple dynamic programming gives a linear-time solution.

For a confidence threshold θ, call $\sum_{i \in I}(v_i - \theta u_i)$ the *gain* of range I. If $v_i - \theta u_i$ is set to x_i, Kadane's algorithm computes the range that maximizes the gain. Unfortunately, it is not equivalent to the range of the optimized support rule, since there may be a larger confident range $I' \supset I$.

4.3. Generalization of Optimized Rules

Our algorithms can be straightforwardly extended to compute rules of the general form

$$(A \in [v_1, v_2]) \wedge C_1 \Rightarrow C_2,$$

where C_1 and C_2 are Boolean statements that do not contain any variables on numeric attributes. Let u_i be the size of $\{t \in R \mid t[A] \in B_i, t \text{ meets } C_1\}$, let v_i be the size of $\{t \in R \mid t[A] \in B_i, t \text{ meets } C_1 \text{ and } C_2\}$, and apply our algorithms to this case.

5. OPTIMIZED RANGES FOR AVERAGE OPERATOR

In this section, we present an application of the algorithm for computing optimized slope pairs and the algorithm for finding optimized support pairs, which are given in Section 4.

In decision-support-type query processing, range queries are often accompanied by aggregates. For instance, bankers want to characterize excellent customers whose saving account balances are relatively high. In order to characterize such customers, the database user may guess that the range from 1000 to 3000 is promising and issue the following query:

```
select  avg(SavingAccount)
from    BankCustomers
where   1000 < CheckingAccount < 3000
```

To discover a satisfactory range with a high average, however, the user might have to guess and generate many queries for various ranges. Instead, we want to obtain the optimized range that maximizes the average of `SavingAccount` among all ranges in `CheckingAccount` that contain an ample percentage of customers, say no less than 10%. In what follows, we formalize this problem and present an efficient way of computing optimized ranges by using the algorithms for generating optimized association rules.

Definition 5.1. Let R be a relation. Let A denote a numeric attributes of R. Suppose that $[v_1, v_2]$ is a range in A. The *support* of range $[v_1, v_2]$—denoted by $support([v_1, v_2])$—is defined as the percentage of tuples for which the values of A fall within the range.

Let B be another numeric attribute to which the average operator is applied. We call B the target numeric attribute. The sum_B of range $[v_1, v_2]$—denoted by $sum_B([v_1, v_2])$—is defined as the summation of the values of B for all tuples in which the value of A falls within $[v_1, v_2]$. Let N denote the number of tuples in R. The avg_B of range $[v_1, v_2]$ denoted by $avg_B([v_1, v_2])$—is $sum_B([v_1, v_2])/(N \times support([v_1, v_2]))$.

Example 5.1. Consider a `BankCustomers` database with numeric attributes `CheckingAccount` and `SavingAccount`. Let $[v_1, v_2]$ be a range in the domain of `CheckingAccount`.

$avg_{\texttt{SavingAccount}}([v_1, v_2])$ is the average of saving account balances of tuples whose checking account balances are in $[v_1, v_2]$.

There is a trade-off between maximizing $support(I)$ and maximizing $avg_B(I)$ for a range I. We therefore give a minimum threshold for either of $support(I)$ or $avg_B(I)$ and compute the range that maximizes the value of the other.

Definition 5.2. Suppose that a minimum threshold is given for the support of a range. Among ranges that meet this constraint, the *maximum* average range I maximizes $avg_B(I)$.

Example 5.2. Consider our running example. Suppose that 10% is given as the minimum threshold for the support of a range. The maximum average range I maximizes $avg_{\texttt{SavingAccount}}(I)$ among all ranges in `CheckingAccount` whose support is no less than 10%.

Definition 5.3. Suppose that we are given a minimum average threshold for $avg_B(I)$ of a range I that is greater than the average of all data. Among all ranges that meet this constraint, the *maximum* support range I maximizes $support(I)$.

If the threshold is no greater than the average of all data, it is trivial that the longest range, namely the domain of B, presents the maximum support range.

EXAMPLE 5.3. In our running database `Bank Customers`, bankers may want to use 10,000, which is greater than the average of all data, as the minimum threshold for the average of saving account balances, and they want to have the maximum support range in `CheckingAccount` that maximizes the number of customers whose checking account balances are in the range.

As in the case of computing the optimized confidence/support rules, we divide the domain of B into buckets $B_1, ..., B_M$, and we only consider ranges that are combinations of consecutive buckets $B_s, ..., B_t$.

Let B_i be $[x_i, y_i]$ such that $x_i \leqslant y_i < x_{i+1}$. Let u_i be the number of tuples in $\{t \in R \mid t[A] \in B_i\}$. Let $support(s, t)$ be $\sum_{i=s}^{t} u_i$. Let v_i denote $\sum_{\{t \in R \mid t[A] \in B_i\}} t[B]$. Let $avg(s, t)$ denote $(\sum_{i=s}^{t} v_i)/(\sum_{i=s}^{t} u_i)$.

Given a minimum support threshold for $support(s, t)$, Algorithm 4.2 computes an optimal slope pair $s \leqslant t$ in $O(M)$ time, and therefore $[x_s, y_t]$ is the maximum average range, since we focus on ranges combined by consecutive buckets.

Also, given a minimum average threshold for $avg(s, t)$, Algorithm 4.4 generates an optimal support pair $s \leqslant t$ in $O(M)$ time, and hence $[x_s, y_t]$ is the maximum support range.

6. PERFORMANCE RESULTS

The proposed algorithms have been implemented in C++. We evaluated their performance on an IBM Power Series 850 with a 133-MHz PowerPC 604 and 96 MB of main memory, and running AIX 4.1.

6.1. Making Buckets

We randomly generated test data with eight numeric attributes and eight Boolean attributes, that is, with 72 bytes per tuple. The test data resided in the AIX file system on a 3.5″ 1.2-GB IDE drive. As a test case, we divided the test data into 1000 buckets with respect to each numeric attribute and counted the number of tuples in every bucket for each Boolean attribute. We compared the performance of our bucketing algorithm, Algorithm 3.1, with those of two other methods. One of these methods, which we call *Naive Sort*, sorts data for each numeric attribute by using Quick Sort. The other one, which we call *Vertical Split Sort*, first splits data vertically to generate a smaller table with tuple identifier and each numeric attribute, and then sorts the temporary table.

Figure 9 shows the execution times for numbers of tuples ranging from 5×10^5 to 5×10^6. For large data sets with more than one million tuples, Algorithm 3.1 outperforms the naive method by more than an order of magnitude, and it also beats Vertical Split Sort by a factor of 2 to 4. Furthermore, the execution time of Algorithm 3.1 grows almost linearly in proportion to the data size.

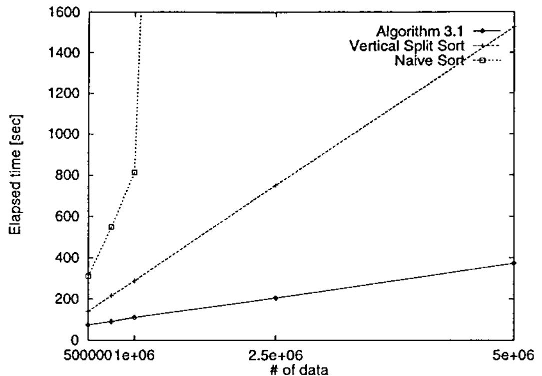

FIG. 9. Performance of bucketing algorithms.

6.2. Finding Optimized Rules

We evaluated the performance of the algorithms for finding optimized rules, comparing the results with those of a naive method that computes, in quadratic time, the confidence and support of all ranges in order to find an optimal one.

Figures 10 and 11 respectively show the execution times for finding optimized confidence rules with a 5% support

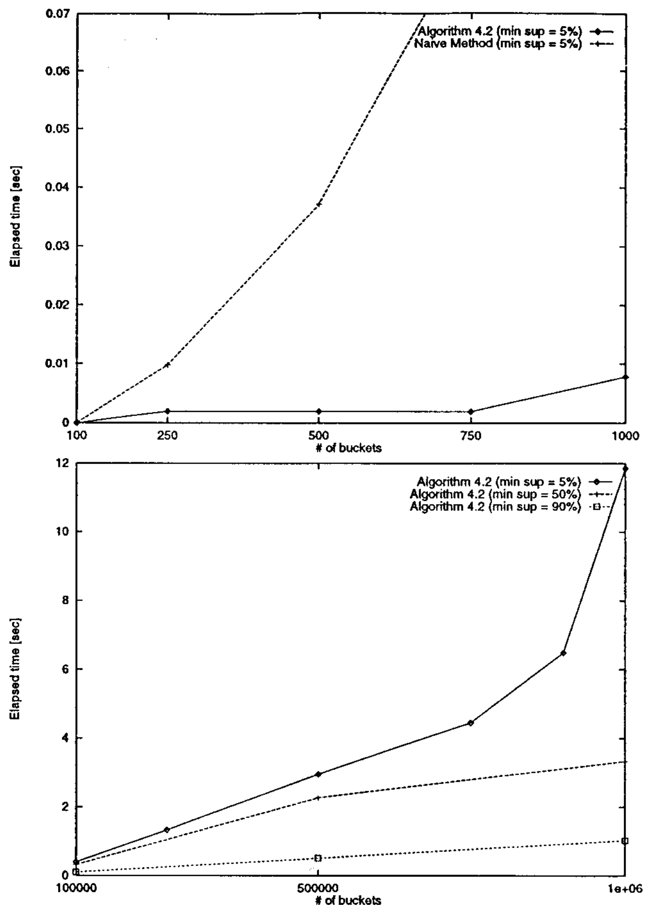

FIG. 10. Finding optimized confidence rules.

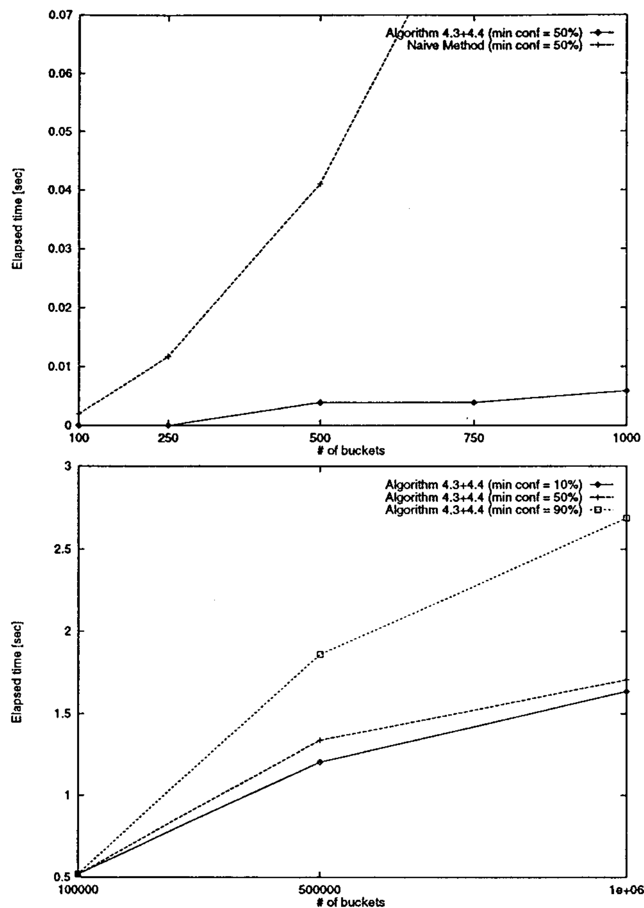

FIG. 11. Finding optimized support rules.

threshold and optimized support rules with a 50% confidence threshold for buckets whose sizes are from 100 to 1,000,000. Our algorithm for finding optimized confidence rules beats the naive method by more than an order of magnitude for more than 500 buckets. For finding optimized support rules, our algorithm is also faster than the naive method by more than an order of magnitude for data sets of more than 100 buckets. Even for small data sets, both of our algorithms outperform the naive ones. The execution time of both algorithms increases linearly in proportion to the number of buckets. In the case for finding optimized confidence rules with minimum support of 5%, the execution time turns to increase rapidly when the number of buckets is more than 800,000 (the second graph of Fig. 10). This is because the algorithm uses more memory than the test machine has. We observed that the operating system reported frequent page faults, which were enforced by the extensive memory accesses for the small minimum support.

REFERENCES

1. R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami, An interval classifier for database mining applications, *in* "Proceedings of the 18th VLDB Conference," pp. 560–573, 1992.
2 R. Agrawal, T. Imielinski, and A. Swami, Database mining: A performance perspective, *IEEE Trans. Knowledge Data Engrg.* **5** (1993), 914–925.
3. R. Agrawal, T. Imielinski, and A. Swami, Mining association rules between sets of items in large databases, *in* "Proceedings of the ACM SIGMOD Conference on Management of Data," pp. 207–216, May 1993.
4. R. Agrawal, and R. Srikant, Fast algorithms for mining association rules, *in* "Proceedings of the 20th VLDB Conference," pp. 487–499, 1994.
5. J. Bentley, Programming pearls, *Comm. ACM* **27** (1984), 865–871.
6. L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, "Classification and Regression Trees," Wadsworth, Belmont, CA, 1984.
7. T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama, Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization, *in* "Proceedings of the ACM SIGMOD Conference on Management of Data," pp. 13–23, June 1996.
8. J. Han, Y. Cai, and N. Cercone, Knowledge discovery in database: An attribute-oriented approach, *in* "Proceedings of the 18th VLDB Conference," pp. 547–559, 1992.
9. M. Mehta, R. Agrawal, and J. Rissanen, Sliq: A fast scalable classifier for data mining, *in* "Proceedings of the Fifth International Conference on Extending Database Technology," 1996.
10. Y. Morimoto, T. Fukuda, S. Morishita, and T. Tokuyama, Implementation and evaluation of decision trees with range and region splitting, *Constraints* **2** (1997), 401–427.
11. R. Motwani and P. Raghavan, "Randomized Algorithms," Cambridge Univ. Press, Cambridge, UK, 1995.
12. R. T. Ng and J. Han, Efficient and effective clustering methods for spatial data mining, *in* "Proceedings of the 20th VLDB Conference," pp. 144–155, 1994.
13. J. S. Park, M.-S. Chen, and P. S. Yu, An effective hash-based algorithm for mining association rules, *in* "Proceedings of the ACM SIGMOD Conference on Management of Data," pp. 175–186, May 1995.
14. G. Piatetsky-Shapiro, Discovery, analysis, and presentation of strong rules, *in* "Knowledge Discovery in Databases," pp. 229–248, 1991.
15. G. Piatetsky-Shapiro and W. J. Frawley, Eds., "Knowledge Discovery in Databases," AAAI Press, Menlo Park, CA, 1991.
16. F. P. Preparata and M. I. Shamos, "Computational Geometry, an Introduction," Springer-Verlag, Berlin/New York, 1985.
17. J. R. Quinlan, "C4.5: Programs for Machine Learning," Morgan Kaufmann, San Mateo, CA, 1993.
18. R. Srikant and R. Agrawal, Mining quantitative association rules in large relational tables, *in* "Proceedings of the ACM SIGMOD Conference on Management of Data," June 1996.
19. M. Stonebraker, R. Agrawal, U. Dayal, E. J. Neuhold, and A. Reuter, DBMS research at a crossroads: The vienna update, *in* "Proceedings of the 19th VLDB Conference," pp. 688–692, 1993.
20. K. Yoda, T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama, Computing optimized rectilinear regions for association rules, *in* "Proceedings of the Third International Conference on Knowledge Discovery and Data Mining," pp. 96–103, Aug. 1997.