Mining Optimized Support Rules for Numeric Attributes*

Rajeev Rastogi

Bell Laboratories
Murray Hill, NJ 07974
rastogi@research.bell-labs.com

Kyuseok Shim

Bell Laboratories
Murray Hill, NJ 07974
shim@research.bell-labs.com

Abstract

In this paper, we generalize the optimized support association rule problem by permitting rules to contain disjunctions over uninstantiated numeric attributes. For rules containing a single numeric attribute, we present a dynamic programming algorithm for computing optimized association rules. Furthermore, we propose a bucketing technique for reducing the input size, and a divide and conquer strategy that improves the performance significantly without sacrificing optimality. Our experimental results for a single numeric attribute indicate that our bucketing and divide and conquer enhancements are very effective in reducing the execution times and memory requirements of our dynamic programming algorithm. Furthermore, they show that our algorithms scale up almost linearly with the attribute's domain size as well as the number of disjunctions.

1 Introduction

Association rules, introduced in [AIS93], provide a useful mechanism for discovering correlations among the underlying data. In its most general form, an association rule can be viewed as being defined over attributes of a relation, and has the form $C_1 \rightarrow C_2$, where C_1 and C_2 are conjunctions of conditions, and each condition is either $A_i = v_i$ or $A_i \in [l_i, u_i]$ (v_i, l_i and u_i are values from the domain of the attribute A_i). Each rule has an associated *support* and *confidence*. Let the *support* of a condition C_i be the ratio of the number of tuples satisfying C_i and the number of tuples in the relation. The support of a rule of the form $C_1 \rightarrow C_2$ is then the same as the support of $C_1 \wedge C_2$, while its confidence is the ratio of the supports of conditions $C_1 \wedge C_2$ and C_1. The association rules problem is that of computing all association rules that satisfy user-specified minimum support and minimum confidence constraints, and schemes for this can be found in [AIS93, AS94, SON95].

For example, consider a relation in a telecom service provider database that contains call detail information. The attributes of the relation are date, time, src_city, src_country, dst_city, dst_country and duration. A single tuple in the relation thus captures information about the two endpoints of each call, as well as the temporal elements of the call. The association rule (src_city = NY) $\rightarrow$ (dst_country = France) would satisfy the user-specified minimum support and minimum confidence of 0.05 and 0.3, respectively, if at least 5% of total calls are from NY to France, and at least 30% of the calls that originated from NY are to France.

The *optimized association rules* problem, motivated by applications in marketing and advertising, was introduced in [FMMT96b]. An association rule R has the form $(A_1 \in [l_1, u_1]) \wedge C_1 \rightarrow C_2$, where A_1 is a numeric attribute, l_1 and u_1 are uninstantiated variables, and C_1 and C_2 contain only instantiated conditions (that is, the conditions do not contain uninstantiated variables). The authors then propose algorithms for determining values for the uninstantiated variables l_1 and u_1 for each of the following cases:

- Confidence of R is maximized and the support of the condition $(A_1 \in [l_1, u_1]) \wedge C_1$ is at least the user-specified minimum support (referred to as the *optimized confidence* rule).
- Support of the condition $(A_1 \in [l_1, u_1]) \wedge C_1$ is maximized and confidence of R is at least the user-specified minimum confidence (referred to as the *optimized support* rule).

Optimized association rules are useful for unraveling ranges for numeric attributes where certain trends or correlations are strong (that is, have high support or confidence). For example, suppose the telecom service provider mentioned earlier was interested in offering a promotion to NY customers who make calls to France. In this case, the timing of the promotion

*This work is part of the Serendip data mining project at Bell Labs. URL: http://www.bell-labs.com/project/serendip/.

may be critical – for its success, it would be advantageous to offer it close to a period of consecutive days in which the maximum number of calls from NY are made and a certain minimum percentage of those calls from NY are to France. The framework developed in [FMMT96b] can be used to determine such periods. Consider, for example, the association rule (date $\in [l_1, u_1]$) $\wedge$ src_city = NY $\rightarrow$ dst_country = France. With a minimum confidence of 0.5, the optimized support rule results in the period during which at least 50% of the calls from NY are to France, and the number of calls originating in NY is maximum.

A limitation of the optimized association rules dealt with in [FMMT96b] is that only a single optimal interval for a single numeric attribute can be determined. However, in a number of applications, a single interval may be an inadequate description of local trends in the underlying data. For example, suppose the telecom service provider is interested in doing upto k promotions for customers in NY calling France. For this purpose, we need a mechanism to identify upto k periods during which a sizeable number of calls from NY to France are made. If association rules were permitted to contain disjunctions of uninstantiated conditions, then we could determine the optimal k (or fewer) periods by finding optimal instantiations for the rule: (date $\in [l_1, u_1]) \vee \cdots \vee$ (date $\in [l_k, u_k]$) $\wedge$ src_city = NY $\rightarrow$ dst_country = France. This information can be used by the telecom service provider to determine the most suitable periods for offering discounts on international long distance calls to France.

In this paper, we generalize the optimized association rules problem for support, described in [FMMT96b]. We allow association rules to contain upto k disjunctions over one uninstantiated numeric attribute. We present a *dynamic programming* algorithm for computing the optimized association rule and whose complexity is $O(n^2k)$, where n is the number of values in the domain of the uninstantiated attribute. We also present two optimizations that significantly improve the computational complexity of our algorithm. The first is a bucketing algorithm that coalesces contiguous values – all of which have confidence greater than minConf. The second is a divide and conquer strategy that enables us to split the original problem into multiple smaller subproblems and then combine the solutions for the subproblems. This not only drastically reduces the computation to be performed but also reduces storage requirements, thus permitting our algorithm to execute in main-memory even for large databases. Our experimental results indicate that the two optimizations enable our dynamic programming algorithm to scale almost linearly with both the domain size n and the number of disjunctions k.

Proofs of theorems presented in the paper can be found in [RS97].

2 Related Work

Association rules for a set of transactions in which each transaction is a set of items bought by a customer, were first studied in [AIS93]. These association rules for sales transaction data have the form $X \rightarrow Y$, where X and Y are disjoint sets of items. Efficient algorithms for computing them can be found in [AS94, PCY95, SON95].

The optimized association rule problem was introduced in [FMMT96b]. The authors permit association rules to contain a single uninstantiated condition $A_1 \in [l_1, u_1]$ on the left hand side, and propose schemes to determine values for variables l_1 and u_1 such that the confidence or support of the rule is maximized. In [FMMT96a], the authors extend the results in [FMMT96b] to the case in which rules contain two uninstantiated numeric attributes on the left hand side. However, their schemes only compute a single optimal region.

In [RS98], we extended the optimized support and confidence problems to compute the k optimal regions, and showed that the problems are NP-hard even for the case of one uninstantiated numeric attribute. We proposed search algorithms that employed *branch and bound* pruning techniques to compute k optimal regions. The optimized support problem described in [RS98] required the confidence over all the optimal regions, considered together, to be greater than a certain minimum threshold. Thus, the confidence of an optimal region could fall below the threshold and this was the reason for its intractability. In this paper, we redefine the optimized support problem such that each optimal region is required to have the minimum confidence. This makes the problem tractable for the one attribute case.

In [ORS98], the authors study the problem of discovering *cyclic association rules*, that is, association rules that display regular cyclic variation over time.

3 Preliminaries

In this section, we define the optimized association rule problem addressed in the paper. The data is assumed to be stored in a relation defined over categorical and numeric attributes. Association rules are built from *atomic* conditions each of which has the form $A_i = v_i$ (A_i could be either categorical or numeric), and $A_i \in [l_i, u_i]$ (only if A_i is numeric). For the atomic condition $A_i \in [l_i, u_i]$, if l_i and u_i are values

from the domain of A_i, the condition is referred to as *instantiated*; otherwise, if they are variables, we refer to the condition as *uninstantiated*.

Atomic conditions can be combined using operators $\wedge$ or $\vee$ to yield more complex conditions. Instantiated association rules, that we study in this paper, have the form $C_1 \rightarrow C_2$, where C_1 and C_2 are arbitrary instantiated conditions. Let the support for an instantiated condition C, denoted by $sup(C)$, be the ratio of the number of tuples satisfying the condition C and the total number of tuples in the relation. Then, for the association rule R: $C_1 \rightarrow C_2$, $sup(R)$ is defined as $sup(C_1)$ and $conf(R)$ is defined as $\frac{sup(C_1 \wedge C_2)}{sup(C_1)}$. Note that our definition of $sup(R)$ is different from the definition in [AIS93] where $sup(R)$ was defined to be $sup(C_1 \wedge C_2)$. Instead, we have adopted the definition used in [FMMT96b] and [FMMT96a]. Also, let minConf denote the user-specified minimum confidence.

The optimized association rule problem requires optimal instantiations to be computed for an uninstantiated association rule that has the form: $U \wedge C_1 \rightarrow C_2$, where U is an uninstantiated atomic condition involving a numeric attribute, and C_1 and C_2 are arbitrary instantiated conditions. For simplicity, we assume that the domain of the uninstantiated numeric attribute is $\{1, 2, \ldots, n\}$. For the one attribute case, the region $[l_1, u_1]$ is simply the interval $[l_1, u_1]$ for the attribute. Suppose, for a region $R = [l_1, u_1]$, we define $conf(R)$ and $sup(R)$ to be $conf$ and sup, respectively, for the rule $A_1 \in [l_1, u_1] \wedge C_1 \rightarrow C_2$. In addition, for a set of non-overlapping regions, $S = \{R_1, R_2, \ldots, R_j\}$, $R_i = [l_i, u_i]$, we define $conf(S)$ and $sup(S)$ to be the $conf$ and sup, respectively, of the rule $\vee_{i=1}^{j} A_1 \in [l_i, u_i] \wedge C_1 \rightarrow C_2$. Note that, since $R_1, \ldots, R_j$ are non-overlapping regions, the following hold for set S.

$$sup(S) = sup(R_1) + \cdots + sup(R_j)$$

$$conf(S) = \frac{sup(R_1) \cdot conf(R_1) + \cdots + sup(R_j) \cdot conf(R_j)}{sup(R_1) + \cdots + sup(R_j)}$$

Having defined the above notation for association rules, we present below, the formulations of the optimized association rule problems for support.

> Given k, determine a set S containing at most k regions such that for each region $R_i \in S$, $conf(R_i) \geq$ minConf and $sup(S)$ is maximized.

We refer to the set S as the optimized support set. In the remainder of the paper, we shall assume that the support and confidence for every point in a region are available – these can be computed by performing a single pass over the relation. The points, along with their supports and confidences, thus constitute the input to our algorithms. Thus, the input size is n.

4 Computing Optimized Support Sets

In this section, we tackle the problem of computing optimized support sets when association rules contain a single uninstantiated numeric attribute. Thus, the uninstantiated rule has the form: $(A_1 \in [l_1, u_1]) \wedge C_1 \rightarrow C_2$, where A_1 is the uninstantiated numeric attribute. We propose a dynamic programming algorithm for computing optimized support sets in Section 4.2. But first, in Section 4.1, we present preprocessing algorithms for collapsing certain contiguous ranges of values in the domain of the attribute into a single bucket, thus reducing the size of the input n. We also present a divide and conquer optimization in Section 4.3 to improve the computational complexity and memory requirements of our dynamic programming algorithm.

4.1 Bucketing

For the one attribute case, each region is an interval and since the domain size is n, the number of possible intervals is $O(n^2)$. Now, suppose we could split the range $1, 2, \ldots, n$ into b buckets, where $b < n$, and map every value in A_1's domain into one of the b buckets to which it belongs. Then the new domain of A_1 becomes $\{1, 2, \ldots, b\}$ and the number of intervals to be considered becomes $O(b^2)$ – which could be much smaller, thus reducing the time and space complexity of our algorithms. Note that the reduction in space complexity also results in reduced memory requirements for our algorithms.

There are two considerations that we must take into account when assigning values in A_1's domain to buckets. The first is that the bucketing procedure must not adversely impact the optimality of the optimized set – that is, the optimized set computed on the buckets must be identical to the one computed using the raw domain values. In order to ensure this, the bucketing procedure must be such that no interval in the optimized set computed prior to bucketing contains a subset of the values assigned to a bucket.

The second consideration has to do with the complexity of the bucketing procedure. The input to the bucketing algorithm is the support and confidence for each of the n values in A_1's domain, sorted in increasing order of the values. We would prefer for the time complexity to be linear in n and the procedure to perform a single pass over the input data. This way, even if n is large or if the information on supports and confidences for the n points does not fit in memory, the b buckets can still be computed efficiently in a single scan of the data.

In the following, we present bucketing algorithms that meet both of the above-mentioned requirements – that is, they do not compromise the optimality of the optimized sets and have time complexity $O(n)$. The output of the algorithms is the b buckets with their supports and confidences, and this becomes the input to our dynamic programming algorithm in Section 4.2.

For optimized support sets, contiguous values in the domain of the uninstantiated attribute for which the confidence is greater than or equal to minConf are collapsed into a single bucket by our bucketing algorithm. Each other value j whose confidence is less than minConf is assigned to a separate bucket – the bucket contains only the value j. In other words, if values in interval $[i, j]$ are assigned to a bucket then either

1. for all $i \leq l \leq j$, confidence of $[l, l]$ is at least minConf, or

2. $i = j$ and confidence of $[i, i]$ is less than minConf.

For example, let the domain of A_1 be $\{1, 2, \ldots, 6\}$ and confidences of 1, 2, 5 and 6 be greater than or equal to minConf, while confidences of 3 and 4 be less than minConf. Then, our bucketing scheme generates 4 buckets, the first containing values 1 and 2, the second and third containing values 3 and 4, respectively, and the fourth containing values 5 and 6. It is straightforward to observe that assigning values to buckets can be achieved by performing a single pass over the input data.

Furthermore, the following theorem can be used to show that the above bucketing scheme preserves the optimality of the computed optimized sets.

Theorem 4.1: *Let S be an optimized support set. Then, for any interval $[i, j]$ in S, it is the case that $conf([i-1, i-1]) <$ minConf and $conf([j+1, j+1]) <$ minConf.* ∎

From the above theorem, it follows that for any set of contiguous values, each of which has confidence at least minConf, either all of the values are contained in the optimized set or none of them are. The reason for this is that if the optimized set contained some of the values and not all the values, then for some interval $[i, j]$ in the optimized set, either $i - 1$ or $j + 1$ would be one of the values with confidence at least minConf – thus violating Theorem 4.1. Thus, each interval in the optimized set computed prior to bucketing, either contains all the values assigned to a bucket or none of them. As a result, the optimized set can be computed using the buckets instead of the initial domain values.

4.2 Optimized Support Algorithm

In this subsection, we present a dynamic programming algorithm for the optimized support problem. The input to the algorithm is the b buckets generated by our bucketing scheme in Section 4.1 along with their confidences and supports. The problem is to determine a set of at most k (non-overlapping) intervals such that the confidence of each interval is greater than or equal to minConf and support of the set is maximized.

4.2.1 Intuition

Suppose optSet$[i, j, l]$ denotes the optimized set for interval $[i, j]$ containing at most l (non-overlapping) intervals. Thus, for every interval $[p, q]$ in the set, 1) confidence is at least minConf, and 2) $i \leq p \leq q \leq j$. It is fairly straightforward to observe that optSet$[i, i, l]$ for all $l = 1, \ldots, k$ is $[i, i]$ if $conf([i, i]) \geq$ minConf. Otherwise, it is $\emptyset$. Similarly, for $i < j$, if $conf([i, j]) \geq$ minConf, then optSet$[i, j, l]$ for all $l = 1, \ldots, k$ is $[i, j]$. On the other hand, if $conf([i, j]) <$ minConf, then optSet for the interval bounded by i and j can be computed from the optSet values for its subintervals.

Theorem 4.2: *If, for $i < j$, $conf([i, j]) <$ minConf, then the optimized set* optSet$[i, j, l]$ *is*

- $l = 1$*:* optSet$[i, j-1, 1]$, *if* $sup($optSet$[i, j-1, 1]) > sup($optSet$[i+1, j, 1])$. optSet$[i+1, j, 1]$, *otherwise.*
- $l > 1$*:* optSet$[i, r, 1] \cup$ optSet$[r+1, j, l-1]$, *where $i \leq r < j$ is such that $sup($*optSet$[i, r, 1] \cup$ optSet$[r+1, j, l-1])$ *is maximum.*

∎

The above theorem enables us to use dynamic programming in order compute optSet$[1, b, k]$.

4.2.2 Dynamic Programming Algorithm

The dynamic programming algorithm for computing optSet$[i, j, l]$ is as shown in Figure 1. Before the algorithm is invoked, we assume that $conf([i, j])$ and $sup([i, j])$ is precomputed for every interval $[i, j]$. The array optSet is used to store the optimized sets for the various intervals, and the entries of the array are initially set to $\emptyset$.

Procedure optSup1D first checks if optSet$[i, j, l]$ has already been computed (by invoking the function *computed* in Step 1). If so, then the optimized set stored in optSet$[i, j, l]$ is returned. Otherwise, the algorithm calculates optSet$[i, j, l]$ using the results of Theorem 4.2. It first checks the confidence of $[i, j]$ to see

```
procedure optSup1D(i, j, l)
begin
1.  if computed(optSet[i, j, l]) = true
2.     return optSet[i, j, l]
3.  if conf([i, j]) ≥ minConf {
4.     optSet[i, j, l] := {[i, j]}
5.     return optSet[i, j, l]
6.  }
7.  tmpSet := ∅
8.  if i < j {
9.     if l = 1
10.       tmpSet := maxSupSet(optSup1D(i, j − 1, 1),
               optSup1D(i + 1, j, 1))
11.    else
12.       for r:= i to j − 1 do
13.          tmpSet := maxSupSet(tmpSet,
               optSup1D(i, r, 1) ∪ optSup1D(r + 1, j, l − 1))
14. }
15. optSet[i, j, l] := maxSupSet(optSet[i, j, l], tmpSet)
16. return optSet[i, j, l]
end
```

Figure 1: Algorithm for computing optimized support set

whether it is at least minConf. If this is the case, then optSet$[i, j, l]$ is the interval $[i, j]$ itself since it has the maximum possible support in $[i, j]$. However, if the confidence of $[i, j]$ is less than minConf, then the two cases are when $l = 1$ and $l > 1$. For $l = 1$, the interval we are interested must be in either $[i, j-1]$ or $[i+1, j]$. The function *maxSupSet* in Step 10 takes two or more optSets as arguments and returns the set with the maximum support. On the other hand, if $l > 1$, then one of the intervals we are interested in must lie in $[i, r]$ and the other $l-1$ must lie in $[r+1, j]$ for some $i \leq r < j$. Thus, optSet$[i, j, l]$ is set to the union of the optSets for the pair of intervals with the maximum support. Note that if optSup1D was initially invoked with parameters 1, b and k, then when Step 12 of the algorithm is executed, the value of j is always b.

In [RS97], we show that the time and space complexity of our dynamic programming algorithm is $O(b^2k)$ and $O(b^2 + bk)$, respectively.

4.3 Divide and Conquer

The dynamic programming algorithm for computing optimized support sets presented in the previous section had time complexity $O(b^2k)$ and space complexity $O(b^2)$, where b is the number of input buckets. In this section, we propose a divide and conquer algorithm that partitions the range consisting of b buckets into subranges and uses the dynamic programming algorithm in order to compute optimized sets for each subrange. It then combines the optimized sets for the various subranges to derive the optimized set for the entire range of buckets. Since the input to the dynamic programming algorithm is a subrange whose size is smaller than that of the entire range, the divide and conquer approach reduces the time and space complexity of computing the optimized set for the entire range. The result is reduced execution times and memory requirements, thus allowing our algorithms to execute in main-memory. Our experiments, in Section 5, indicate that with the divide and conquer optimization, execution times and storage needs of our algorithms increase linearly (as opposed to quadratically) with the number of input buckets.

In the following subsections, we first present the intuition underlying our divide and conquer approach. We then present a scheme with linear time complexity for splitting the range of buckets into subranges – this makes the scheme practical even for large b and for the case in which the support and confidence information for the b buckets does not fit in main-memory. Finally, we show how the results for the subranges can be combined to yield the optimized support set.

4.3.1 Intuition

We first describe the intuition underlying the optimization. Suppose for a bucket p, $1 \leq p \leq b$, it is the case that for every interval $[i, j]$ containing p, we have $conf([i, j]) <$ minConf. Then, since every interval in the optimized support set must have confidence at least minConf, we can conclude that p does not occur in the optimized set. Consequently, every interval in the optimized set must either be to the left of p or to the right of p (none of the intervals can span p). We can thus independently compute optSet$[1, p-1, l]$ and optSet$[p+1, b, l]$, for all $1 \leq l \leq k$, and then set optSet$[1, b, k]$ to be optSet$[1, p-1, l] \cup$ optSet$[p+1, b, k-l]$ for the value of l between 0 and k that results in maximum support (optSet$[1, p-1, 0]$ and optSet$[p+1, b, 0]$ are both trivially $\emptyset$). Since the optimized set must have $0 \leq r \leq k$ intervals on the left and at most $k-r$ intervals on the right of p, by considering all values of l between 0 and k for optSet$[1, p-1, l] \cup$ optSet$[p+1, b, k-l]$, optSet$[1, b, k]$ is guaranteed to be an optimized set.

A generalization of the above idea is to first compute *partition points* – a partition point is a bucket with the property that every interval containing it has confidence less than minConf. Thus, no interval in

```
procedure computePartition()
begin
1.  partPoints := ∅
2.  earliest := b + 1
3.  j := b
4.  for  i := numEffective downto 1 do
5.     while j ≥ effective[i] do
6.        if conf([effective[i], j] ≥ minConf){
7.           earliest := effective[i]
8.           break (out of while-loop)
9.        } else {
10.          if j < earliest
11.             partPoints := partPoints ∪ {j}
12.          j := j − 1
13.       }
14. return partPoints
end
```

Figure 2: Algorithm for generating partition points

the optimized set can contain a partition point. If $p_1, \ldots, p_{m-1}$ are the partition points in increasing order, then these partition interval $[1, b]$ into m intervals – $[1, p_1 - 1]$, ..., $[p_{i-1} + 1, p_i - 1]$, ..., $[p_{m-1} + 1, b]$. The m partitioned intervals have the property that every interval in the optimized set is wholly contained in a single partition. The reason for this is that between any two adjacent partitions, there is a partition point. optSet$[1, b, k]$ can then be computed by computing for every partition $[i, j]$ and for $0 \leq l \leq k$, optSet$[i, j, l]$, and then choosing the combination of optSets for the various partitions, that has the maximum support.

4.3.2 Computing Partition Points

In Figure 2, we present an $O(b)$ algorithm for computing the partition points $p_1, \ldots, p_{m-1}$. The key idea underlying the algorithm is as follows. For every bucket j, we first compute the largest interval $[i, j]$ ending at j and with confidence at least minConf. A point p_i is a partition point if for all $j > p_i$, the largest interval ending at j does not contain p_i and no interval ending at p_i has confidence that exceeds minConf. The largest interval with confidence minConf for all buckets can be computed in linear time using recent results from [FMMT96b]. In [FMMT96b], the authors introduce the notion of *effective points* – a bucket s is effective if for all $i < s$, $conf([i, s-1]) <$ minConf. It is fairly straightforward to observe that if $[s, j]$ is the largest interval with confidence exceeding minConf and ending at j, then s must be an effective point. Also, for an effective point s, if $[s, j]$ has confidence less than minConf, then for every other $i < s$, $conf([i, j]) <$ minConf.

We are now in a position to describe how procedure computePartition (see Figure 2) utilizes the effective points in order to compute the partition points. In [FMMT96b], the authors show how effective points can be computed in linear time in a single forward pass over the b buckets. We do not repeat this here and assume that there are numEffective effective points that are stored in increasing order in the array effective. The above-mentioned properties of effective points make them useful for efficiently computing the largest interval ending at bucket j and with confidence at least minConf – only effective points preceding j need to be scanned in reverse order until one is encountered, say s, for which $conf([s, j])$ decreases below minConf.

Procedure computePartition simultaneously scans both the input buckets as well as the effective points in the reverse order. The variable j keeps track of the current bucket being scanned while effective[i] is the effective point currently under consideration. Finally, the variable earliest stores the earliest effective point such that there exists an interval with confidence minConf begining at earliest and ending at a bucket greater than or equal to j. Thus, if for bucket j, $j <$ earliest, then j is a partitioning point (see steps 10 and 11) since no interval ending at or after j has confidence minConf.

When scanning the buckets and effective points in reverse order, for bucket j, only effective points preceding it are candidates for the longest interval with confidence minConf and ending at j. Furthermore, if $conf([\text{effective}[i], j]) \geq$ minConf (see Step 6), then we next consider the effective point immediately before it (by decrementing i by 1) for the longest interval ending at j. Also, earliest is set to effective[i]. On the other hand, if $conf([\text{effective}[i], j]) <$ minConf, then we simply consider the bucket preceding j (by decrementing j in Step 12) since the earliest bucket for the longest interval with confidence minConf ending at j cannot be before earliest (if there are effective points between effective[i] and j, earliest stores the effective point immediately following effective[i]).

The algorithm performs two passes over the data – one forward pass to compute the effective points and a reverse pass during which the partition points are computed. Thus, the algorithm is efficient and can be used even if the data is too large to fit in main-memory.

```
procedure divideConquerSup()
begin
1.  for i := 1 to m do {
2.      optSetPart[i, 0] := ∅
3.      for l := 1 to k do
4.          optSetPart[i, l] := optSup1D(lower(i), upper(i), l)
5.  }
6.  for i := 0 to k do
7.      optSet[i] := optSetPart[1, i]
8.  for i := 2 to m do
9.      for j := k downto 1 do
10.         for p := 0 to j do {
11.             tmpSet := optSet[j − p] ∪ optSetPart[i, p]
12.             if sup(tmpSet) > sup(optSet[j]) then
13.                 optSet[j] := tmpSet
14.         }
15. return optSet[k]
end
```

Figure 3: Divide and conquer algorithm for optimized support

4.3.3 Divide and Conquer Algorithm

Once the m partitions are generated (from the partitioning points), then procedure divideConquerSup (see Figure 3) can be used to compute the optimized support set. Let $lower(i)$ and $upper(i)$ be the boundary buckets for partition i, and let b_i be the number of buckets in partition i. First, in steps 1 through 5, the optimized set for each partition containing at most 0, ..., k intervals is computed using optSup1D. Procedure optSup1D(i, j, l) returns the optimized support set whose size is at most l for the interval consisting of buckets i through j. The optimized set of size l for partition i is stored in optSetPart$[i, l]$. Even though, in the for-loop in steps 3-4, optSup1D is invoked k times with the number of intervals ranging from 1 to k, the complexity of the for-loop is still $O(b_i^2 k)$ because the optimized sets for the various intervals in partition i (that are computed during an invocation of optSup1D) can be stored and shared between the k invocations of optSup1D for the partition. Thus, for an arbitrary interval in the partition and a maximum size l for the optimized set, the optimized set is computed only once (the first time it is required).

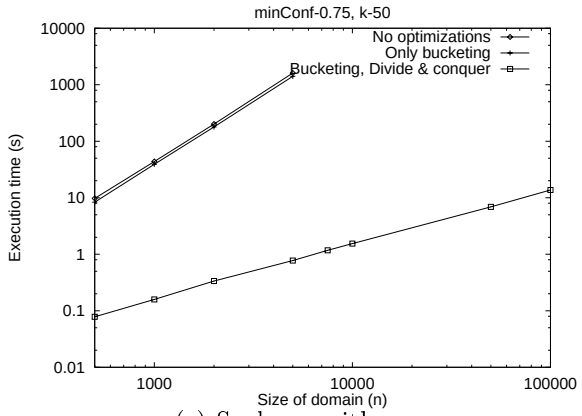
The optimized sets computed for the various partitions are then merged in steps 6–14. The merging process is carried out in successive steps – in step i, the optimized set for partition i is merged with the result of the merge of partitions 1, ..., $i-1$ (that is stored in optSet). Thus, at the end of step i, optSet$[j]$ stores the optimized set containing at most j intervals belonging partitions $1, \ldots, i$. We need to compute optSet$[j]$ for all $1 \leq j \leq k$ since the optimized set (containing at most k intervals) for the first $i+1$ partitions is obtained by combining, during step $i+1$, optSet$[j]$ and optSetPart$[i+1, k-j]$ for the value of j that causes optSet$[k]$ to be maximized. The complexity of each step of the merge is thus $O(k^2)$ since we need to compute optSet for all values between 1 and k. Since the number of partitions is m, the complexity of steps 6–14 becomes $O(mk^2)$. Thus, the overall complexity of the algorithm becomes $O((b_1^2 + \ldots + b_m^2)k + mk^2)$ where $b \geq b_1 + \ldots + b_m$. If b_{max} denotes the largest value among $b_1, \ldots, b_m$, then the complexity becomes $O(b_{max}^2 mk + mk^2)$. In our experiments, we found that for a large number of cases, $b_{max} << b$ and since $k << b$, the divide and conquer results in substantial reductions in the computational complexity of our dynamic programming algorithm (whose original complexity is $O(b^2 k)$). In addition, it also reduces the storage and memory requirements of our dynamic programming algorithm from $O(b^2)$ to $O(b_{max}^2)$.

5 Experimental Results

In this section, we study the performance of our algorithms for computing optimized support sets for the one attribute case. In particular, we show that the bucketing and divide and conquer optmizations make our dynamic programming algorithm highly scaleable. For instance, we can tackle domains of sizes as high as 100,000 in a few seconds. Furthermore, mining as many as 250 intervals for 100,000 domain values can be achieved in a matter of a few minutes. We also study the sensitivity of the above two optimizations to the minimum confidence threshold.

In our experiments, the data file is read only once at the beginning of each algorithm in order to compute the support and confidence for every point. The time for this, in most cases, constitutes a tiny fraction of the total execution time of our algorithms. Thus, we do not include the time spent on reading the data file in our results. Furthermore, note that the performance of our algorithms does not depend on the number of tuples in the data file – it is more sensitive to the size of the attribute's domain n and the number of intervals k. Our experiments were performed on a Sun Ultra-2/200 machine with 512 MB of RAM and running Solaris 2.5.

Synthetic Datasets: The association rule that we experimented with, has the form $U \wedge C_1 \rightarrow C_2$ where U contains 1 uninstantiated attribute (see Section 3) whose domain consists of integers ranging from 1 to n.



(a) Scale-up with n

n	b	m	b_{max}
500	195	80	6
1000	354	140	8
2000	738	293	8
5000	1829	743	10
7500	2759	1123	8
10000	3711	1518	12
50000	16321	6847	10
100000	32266	13512	12

(b) Values of b, m and b_{max} for different input sizes

Figure 4: Varying input size

Every domain value (that is, point in one-dimensional space) is assigned a randomly generated confidence between 0 and 1 with uniform distribution. Each value is also assigned a randomly generated support between 0 and $\frac{2}{n}$ with uniform distribution; thus, the average support for a value is $\frac{1}{n}$.

5.1 Bucketing and Divide and Conquer

In this subsection, we study the improvements in execution times that result due to the bucketing and divide and conquer optimizations. In Figure 4(a), we plot the performance of three variants of our algorithm as the domain size is increased from 500 to 100,000 – (1) with no optimizations, (2) with only bucketing, and (3) with both bucketing and divide and conquer. We use a log scale to represent values along both axes. Also, in our experiments, we fix the number of intervals k at 50 and use a minimum confidence threshold of 0.75.

For optimized support, we found that with no optimizations, our dynamic programming algorithm took times excessive of 30 minutes for as few as 5000 values in the domains of the attributes. On the other hand, with both bucketing and divide and conquer, our algorithm took less than 15 seconds for domain sizes as high as 100,000. From the graphs, it follows that the major portion of the performance improvement results due to divide and conquer. Even though bucketing does reduce input size, these reductions are fairly small for optimized support (about 5-6%). Divide and conquer, on the other hand, partitions the original problem of size b into m subproblems of size at most b_{max} and has complexity $O(b_{max}^2 mk + mk^2)$. For $n = 100,000$ and the optimized support case, divide and conquer splits the $b = 95000$ buckets into $m = 13000$ partitions each of whose size is less than $b_{max} = 20$. Obviously, since $b_{max} << b$, $k << b$ and $m < b$, it follows that the computational complexity of divide and conquer is much smaller than $O(b^2 k)$, the complexity with only bucketing. The effectiveness of bucketing and divide and conquer depend on minConf values. We discuss this in more detail in Section 5.3.

5.2 Scale-up with n

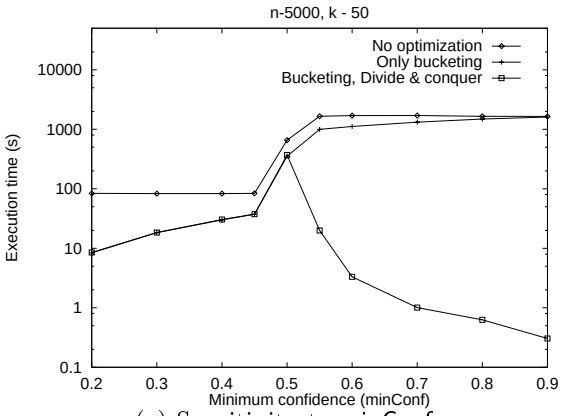
Figure 4(a) also shows how the algorithms with and without optimizations scale with the input size n. Due to the quadratic complexity of our algorithm, we found that as n doubles, without optimizations, the execution time increases almost four-fold. This is in line with what we expected. The same holds when we use only the bucketing optimization.

With the divide and conquer strategy, the complexity of our algorithm is $O(b_{max}^2 mk + mk^2)$, where m is the number of partitions and b_{max} is the size of the largest partition. Thus, if with increasing n, b_{max} stays approximately the same and m increases linearly with n, then we can expect the divide and conquer algorithm to exhibit a linear scale-up with increasing n. This is exactly what we observe for the optimized support case. The table in Figure 4(b) contains the values of m and b_{max} for increasing values of n. For optimized support, b_{max} stays in a narrow range between 15 and 20. Furthermore, m increases almost linearly with respect to n from 80 (for $n = 500$) to 13500 (for $n = 100,000$). Thus, as n increases, the cost of computing the optimized sets for each partition stays approximately the same, and the algorithm only incurs a linear increase in the cost for computing optimized sets for all the partitions and then combining them.

5.3 Sensitivity to minConf

We next study the senstivity of our bucketing and divide and conquer optimizations to minimum confidence values. In Figure 5, for $n = 5000$ and $k = 50$, we plot execution times for our algorithms as minConf is varied from 0.2 to 0.9.

Figure 5(a) plots the performance of our algorithms with and without optimizations for computing optimized support sets. First, we find that without any optimizations, the performance of our dynamic program-



(a) Sensitivity to minConf

minConf	b	m	b_{max}
0.2	1620	1	1619
0.3	2166	1	2165
0.4	2489	1	2488
0.45	2553	1	2552
0.5	2539	1	2538
0.55	2451	136	172
0.6	2355	395	70
0.7	2063	708	14
0.8	1537	686	8
0.9	829	399	2

(b) Values of b, m and b_{max} for different confidences

Figure 5: Varying minConf

ming algorithm becomes worse as confidence increases. The reason for this is that, for optimized support, if the confidence of an interval is minConf or higher, the optimized set for the interval is the interval itself, and further recursive calls for the interval are not needed. When minConf is small, there are a large number of such intervals with confidence at least minConf.

We next turn our attention to the bucketing and divide and conquer optimizations. In the table in Figure 5(b), for a given confidence, we present the number of buckets generated by the bucketing algorithm b, the number of partitions generated by the divide and conquer algorithm m, and the max size of a partition b_{max}. From the table, we can make the following three observations – (1) as the confidence increases, the number of buckets input to our algorithm increases, (2) for values of minConf that are lower than 0.5, the divide and conquer optimization has no effect, and (3) as the confidence increases beyond 0.5, for increasing confidence values, the max partition size decreases. The reason for Point (1) is that the bucketing algorithm coalesces contiguous values with confidence more than minConf and at lower minConf values, there are a larger number

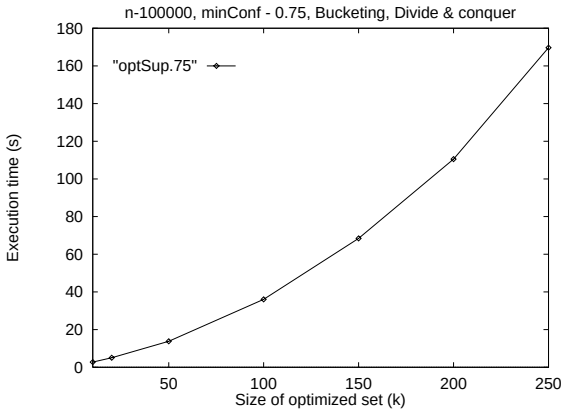


Figure 6: Scale-up with k

of such values. Points (2) an (3) can be attributed to the fact that the number of partition points increases as the confidence increases (and the number is 0 when minConf is less than 0.5). The above three points completely explain the behaviour of our optimizations in Figure 5(a). With only bucketing, at lower minConf values, due to the smaller number of buckets and consequently the smaller input sizes, the performance of our algorithm with bucketing is much better than without bucketing. However, as the confidence increases, the reductions in input size become smaller and smaller, and thus the bucketing optimization has very little effect for large confidence values. The performance of our algorithm with both bucketing and divide and conquer enhancements is more interesting. First, since for confidence values below 0.5, divide and conquer does not generate any partitions, the performance of our algorithm with and without divide and conquer is the same. However, beyond 0.5, divide and conquer kicks in and the execution times decrease as confidence increases – the smaller sizes of partitions is primarily responsible for this.

5.4 Scale-up with k

In order to determine, how our algorithm with both the bucketing and divide and conquer optimizations scales for increasing values of k, we varied k between 10 and 250 with a domain size of 100,000 and minConf = 0.75. The results of our experiments for the optimized support case are as shown in Figure 6. From the graphs, it follows that the execution times increase slightly more than linearly as k is increased. The reason for this is that the complexity of our divide and conquer algorithm is $O(b_{max}^2 mk + mk^2)$. The first term, which is the complexity of computing optmized sets for the m partitions, increases linearly with k. However, the second term, which is the cost of combin-

ing the results for the partitions to get the final optimized set has complexity that is quadratic in k. Thus, we find that the increase in execution times for our algorithms is somewhere between linear and quadratic for increasing k. On an average, we found that computation times increase about 3-fold every time k doubles. Note that for values of k much larger than b_{max}, a majority of the time is spent in combining the results for the various partitions. Thus, subsequent increases in the value of k could result in quadratic increases in execution times when the divide and conquer optimization is employed.

6 Concluding Remarks

In this paper, we generalized the optimized support association rule problem by permitting rules to contain upto k disjunctions over one uninstantiated numeric attribute. We presented a *dynamic programming* algorithm for computing the optimized association rule and whose complexity is $O(n^2k)$, where n is the number of values in the domain of the uninstantiated attribute. We also presented two optimizations that significantly improve the computation time and memory requirements of our algorithm. The first is a bucketing algorithm that coalesces contiguous values – all of which have confidence either greater than the minimum specified confidence or less than the minimum confidence. The second is a divide and conquer strategy that enables us to split the original problem into multiple smaller subproblems and then combine the solutions for the subproblems. We experimentally showed that the two optimizations enable our dynamic programming algorithms to execute in main-memory for large values of the domain size n. With the optimizations, our algorithms scale almost linearly with both the domain size n and the number of disjunctions k.

Acknowledgements: We would like to thank Narain Gehani, Hank Korth and Avi Silberschatz for their encouragement. Without the support of Yesook Shim, it would have been impossible to complete this work.

References

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. of the VLDB Conference*, Santiago, Chile, September 1994.

[FMMT96a] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takesh Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1996.

[FMMT96b] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takesh Tokuyama. Mining optimized association rules for numeric attributes. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1996.

[ORS98] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Int'l Conference on Data Engineering*, Orlando, 1998.

[PCY95] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proc. of the ACM-SIGMOD Conference on Management of Data*, San Jose, California, May 1995.

[RS97] R. Rastogi and K. Shim. Mining optimized association rules for numeric attributes. Technical Report 0112370-971110-25, Bell Laboratories, Murray Hill, 1997.

[RS98] R. Rastogi and K. Shim. Mining optimized association rule for categorical and numeric attributes. In *Int'l Conference on Data Engineering*, Orlando, 1998.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the VLDB Conference*, Zurich, Switzerland, September 1995.