Mining Confident Rules Without Support Requirement [*]

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

Yu He
Hewlett-Packard Singapore
yu_he@hp.com

David W. Cheung
Francis Y. L. Chin
University of Hong Kong
dcheung,chin@csis.hku.hk

ABSTRACT

An open problem is to find all rules that satisfy a minimum confidence but not necessarily a minimum support. Without the support requirement, the classic support-based pruning strategy is inapplicable. The problem demands a confidence-based pruning strategy. In particular, the following monotonicity of confidence, called the *universal-existential upward closure*, holds: if a rule of size k is confident (for the given minimum confidence), for *every* other attribute not in the rule, *some* specialization of size $k+1$ using the attribute must be confident. Like the support-based pruning, the bottleneck is at the memory that often is too small to store the candidates required for search. We implement this strategy on disk and study its performance.

1. INTRODUCTION

The problem of mining *association rules* [2, 3] is to find all rules $X \rightarrow Y$ between itemsets X and Y, from a given collection of transactions, that have the user-specified minimum support and minimum confidence. A high support ensures the statistical significance and a high confidence ensures the predictability. The classic support-based pruning strategy, such as Apriori [2, 3], is to push the support requirement into the search by exploiting the *downward closure* property: if an itemset $\{i_1, \ldots, i_k\}$ is frequent, i.e., above the minimum support, so is every subset of size $k-1$. Thus, an itemset $\{i_1, \ldots, i_k\}$ needs to be examined only if all its subsets of size $k-1$ are frequent. The confidence requirement is entirely ignored until frequent itemsets are screened for rules of high confidence.

The above approach suffers from an unnecessary bottleneck if there are many frequent itemsets but a few confident rules, which occurs when the minimum support is low and the minimum confidence is high, an interesting case pointed out recently [5, 6]. In fact, with a high minimum support, discovered rules often are obvious and well known, and rules of low support but high confidence, which usually provides new insights, are not discovered. For example, a recommendation rule is expected to have high confidence for a high hit rate, but each rule applies to a small number of customers because of their non-uniform tastes. The document classification provides another example for confident rules with low support [9] where most topics have alternative characteristics, each of which applies to a small number of documents. Our experience is that very few datasets have a "clean" structure that is captured by a small number of rules of high support. Much often, the structure is less clean and is described by many rules, each of which captures a small portion of the structure.

With low or no minimum support, the classic support-based pruning strategy becomes inapplicable. What is needed is a confidence-based pruning strategy by pushing the confidence requirement into the search. This change, from support-based pruning to confidence-based pruning, tends out to be drastic. Unlike support, confidence does not have the downward closure property: shorter rules $Age \geq 35 \rightarrow Salary = High$ and $Gender = M \rightarrow Salary = High$ could have lower confidence than longer rule $Age \geq 35 \wedge Gender = M \rightarrow Salary = High$. The confidence does not have the *upward closure* property either: longer rule $Age \geq 35 \wedge Gender = M \rightarrow Salary = Low$ could have lower confidence than shorter rules $Age \geq 35 \rightarrow Salary = Low$ and $Gender = M \rightarrow Salary = Low$.

Recently, [8] pointed out the following pruning property of confidence: if a rule of size k, $A_{i_1} = a_{i_1} \wedge \ldots \wedge A_{i_k} = a_{i_k} \rightarrow C = c$, is confident (for the given minimum confidence), some specialization of size $k+1$, $A_{i_1} = a_{i_1} \wedge \ldots \wedge A_{i_k} = a_{i_k} \wedge A_{i_{k+1}} = a_{i_{k+1}} \rightarrow C = c$, must be confident. Therefore, to generate candidate confident rules of size k, we only need to examine confident rules of size $k+1$. For a large database, the bottleneck of this approach is at the memory that is often too small to hold all candidates/rules. Indeed, it is not uncommon that candidates/rules take more space than the input database. Thus, an unsolved problem is to minimize the I/O cost where rules/candidates are kept on disk. A major challenge is that candidates/rules often are not clustered on disk in the way they are requested. In this paper, we present a clustering scheme and an access method to solve this problem. Our goal is to minimize the dominating I/O cost as in a typical database environment.

[*]Research was supported in part by research grants from the Natural Science and Engineering Research Council of Canada

The rest of the paper is organized as follows. In Section 2, we present an overview of our approach, which covers the problem studied, the confidence-based pruning strategy, and a conceptual algorithm. In Section 3, we present a disk-based implementation. In Section 4, we evaluate the implementation. In Section 5, we review related work. We conclude this paper in Section 6.

2. THE CONFIDENCE-BASED PRUNING

We assume that the database is a relational table T over m *non-class attributes* $A_1, \ldots, A_m$ and one *class attribute* C. All attributes are *categorical*. A tuple in T has the form $< a_1, \ldots, a_m, c >$, where a_i are values of A_i, and c is a value of C, called a *class*. A rule, or a k-rule, has the form $A_{i_1} = a_{i_1} \wedge \ldots \wedge A_{i_k} = a_{i_k} \rightarrow C = c$, with each attribute occurring at most once. By assuming that each value a_i is prefixed with its attribute A_i, we can simply write a rule as $a_{i_1}, \ldots, a_{i_k} \rightarrow c$ without mentioning attributes. x often denotes one or more values. A tuple t and a rule $x \rightarrow c$ *match* if t contains all the values in x. A rule of the form $x, a_i \rightarrow c$ is called a A_i*-specialization* of rule $x \rightarrow c$ if a_i is a value of A_i. $|T|$ denotes the number of tuples in T, and $num(x)$ denotes the number of tuples in T that contain all the values in x. The *support* of rule $x \rightarrow c$, denoted $sup(x \rightarrow c)$, is $num(x,c)/|T|$. The *confidence* of rule $x \rightarrow c$, denoted $conf(x \rightarrow c)$, is $num(x,c)/num(x)$. Given a minimum confidence $minconf$, a rule is *confident* if $conf(x \rightarrow c) \geq minconf$.

Definition 2.1 (Mining confident rules). The problem of *mining confident rules* is to find all confident rules for a given minimum confidence. □

Importantly, we drop the usual minimum support requirement adopted in most association rule mining algorithms. Thus, the support-based pruning, such as [2, 3], is not applicable to the confident rule mining problem. [8] observed a *confidence-based pruning* strategy that pushes the confidence requirement into the search of rules. The following rules illustrate the idea:

r1: $Age.young \rightarrow Buy.yes$
r2: $Age.young, Gender.M \rightarrow Buy.yes$
r3: $Age.young, Gender.F \rightarrow Buy.yes.$

r2 and r3 are two specializations of r1, by having the additional conditions $Gender.M$ and $Gender.F$. These conditions are exclusive and exhaustive in the sense that exactly one will hold for each tuple. Therefore, if one condition has a negative impact on confidence, the other must have a positive impact, and vice versa. Put differently, one of r2 and r3 must have as much confidence as the original rule r1, or equivalently, we can prune r1 if none of r2 and r3 is confident for the given minimum confidence. The same argument applies to attribute *Education* with two values *high* and *low*: if $r1$ is confident, at least one of $Age.young$, $Education.high \rightarrow Buy.yes$ and $Age.young$, $Education.low \rightarrow Buy.yes$ must be confident. This observation was stated as the following upward closure property.

Theorem 2.1. For every attribute A_i not occurring in a rule $x \rightarrow c$, (i) some A_i-specialization of $x \rightarrow c$ has at least the confidence of $x \rightarrow c$, (ii) if $x \rightarrow c$ is confident, so is some A_i-specialization of $x \rightarrow c$. This property is called the *universal-existential upward closure*.

The universal-existential upward closure suggests the following level-wise generation of confident rules.

The level-wise candidate generation: Assume that all confident k-rules are generated, starting with $k = m$, the number of non-class attributes. We generate a candidate $(k-1)$-rule $x \rightarrow c$ only if for every attribute A_i not occurring in $x \rightarrow c$, some A_i-specialization of $x \rightarrow c$ is confident. In other words, a $(k-1)$-rule is pruned if for some attribute A_i, no A_i-specialization of the rule (of size k) was found confident.

We can implement this generation in *relational algebra* supported by any database system as follows. Let $Rule_k$ and $Cand_k$ be the set of confident k-rules and candidate k-rules. Let $Rule_k(X,C)$ and $Cand_k(X,C)$ be the set of rules in $Rule_k$ and the set of rules in $Cand_k$ with attributes X on the left-hand-side. We represent rules $a_{i_1}, \ldots, a_{i_k} \rightarrow c$ by tuples $< a_{i_1}, \ldots, a_{i_k}, c >$ on attributes $A_{i_1}, \ldots, A_{i_k}, C$, and $Rule_k(X,C)$ and $Cand_k(X,C)$ by relational tables over attributes X, C. With this notation, Theorem 2.1(ii) gives the following relational computation of $Cand$.

Corollary 2.1. Let $Cand_{k-1}(X,C) = \cap_{A_i} \pi_{X,C} Rule_k(X, A_i, C)$, where $\pi_{X,C}$ denotes the projection onto the attributes X and C, and A_i ranges over all non-class attributes not in X. Then $Cand_k(X,C) \supseteq Rule_k(X,C)$. □

Corollary 2.1 identifies two kinds of candidate pruning.

Projection-based pruning. Every candidate of size $k-1$ (except longest ones) must come from *some* confident A_i-specialization, i.e., a projection of a confident k-rule.

Intersection-based pruning. Every candidate of size $k-1$ (except longest ones) must come from a confident A_i-specialization for *every* attribute A_i not yet in the candidate. There are $m-k+1$ such attributes A_i for a candidate of size $k-1$, so each candidate of size $k-1$ comes from intersecting the projection of $m-k+1$ confident $(k+1)$-rules. Note that this pruning assumes the projection-based pruning.

Figure 1 gives an overview of the level-wise generation. The search starts with the seed $Rule_m$ containing all tuples in T that represent confident rules (line 2). In iteration k, starting with $k = m$, we generate $Cand_{k-1}$ based on Corollary 2.1 (line 4), compute the confidence of candidates in $Cand_{k-1}$ in one pass of T (line 5), and collect confident $(k-1)$-rules (line 6). To compute the confidence of candidates, we need to scan the tuples in T, and for each tuple t, we update $num(x)$ and $num(x,c)$ for all matching candidates $x \rightarrow c$ in $Cand_{k-1}$: if t contains class c, we increment both $num(x)$ and $num(x,c)$; otherwise, we increment $num(x)$. At the end of the scan, the confidence of $x \rightarrow c$ is $num(x,c)/num(x)$.

```
Input: table $T$ over $A_1, \ldots, A_m, C$ and $minconf$;
Output: all confident rules;
Method:
1 $k = m$;
2 $Rule_m$ = all confident $m$-rules;
3 while $k > 1$ and $Rule_k$ is not empty do
4   generate $Cand_{k-1}$ from $Rule_k$ (based on Corollary 2.1);
5   compute the confidence of candidates in one pass of $T$;
6   $Rule_{k-1}$ =all confident candidates in $Cand_{k-1}$;
7   $k - -$;
8 return all $Rule_k$;
```

Figure 1: The overview of mining confident rules

This algorithm works fine if T, $Rule_k$, $Cand_{k-1}$ all fit in memory. However, we frequently observed that rules/candidates generated are many times larger than the input database, in which case this assumption no longer holds. In the rest of the paper, we consider a disk-based implementation of the confidence-based pruning strategy, where T, $Rule_k$, $Cand_{k-1}$ are stored on disk. Our goal is to minimize the I/O cost, i.e., the number of disk pages accessed. The major challenge is that these data structures are not clustered in the way they are requested, and a straightforward implementation yields excessive I/O. Without a careful clustering and access method, the extra I/O generated for performing the confidence-based pruning can wipe out the benefit of the pruning. In the next section, we propose a clustering scheme and an access method with the goal of minimizing the I/O cost.

3. THE DISK-BASED IMPLEMENTATION

In this section, we implement the algorithm in Figure 1 assuming that T, $Cand_{k-1}$, and $Rule_k$ are stored on disk. Our goal is to minimize the disk access. We focus on Step 2, 4 and 5. Let us first analyze some common requirements of these steps. In Step 2, we need to search for all tuples that agree on all non-class attribute values. In Step 4, we need to search for the rules in $Rule_k$ that agree on $k-1$ non-class values and disagree on the remaining non-class attribute. In Step 5, we need to search for the candidates that match a given tuple in T. A common operation of all is to retrieve tuples or rules by non-class values. Since tuples and rules are not clustered on disk for such retrievals, a straightforward implementation will result in excessive repeated disk access. We like to cluster tuples and rules on disk so that those that are "likely to share similar values" are retrieved together. Two tuples or rules are likely to share values if they belong to the same bucket determined by a hash-partitioning on those values. Let us consider such a partitioning scheme.

Consider a hash function h_i for attribute A_i with the range $[0..q_i]$. $q_i = 0$ if no partitioning is chosen for A_i. h_i hashes the tuples in the database T into $(q_1 + 1) \times \ldots \times (q_m + 1)$ buckets, called *T-buckets.* A T-bucket denoted by bucket id $[b_1, \ldots, b_m]$ contains all the tuples $< a_1, \ldots, a_m, c >$ in T such that $h_i(a_i) = b_i$ for $1 \le i \le m$. Similarly, the same hash functions partition the candidates in $Cand_{k-1}$ and rules in $Rule_k$ into *C-buckets* and *R-buckets.* Note that hash values for different attributes should not be mixed up. This can be enforced by representing each hash value b_i as $A_i.b_i$, where A_i is the attribute for b_i. We assume that there is a mapping from bucket ids to disk addresses of buckets. We say that a C-bucket $[b'_{i_1}, \ldots, b'_{i_k}]$ *matches* a T-bucket $[b_1, \ldots, b_m]$ if $\{b'_{i_1}, \ldots, b'_{i_k}\}$ is a subset of $\{b_1, \ldots, b_m\}$.

The major advantage of hash-partitioning is that certain prunings can be performed at the *bucket id level* before accessing actual tuples and candidates, and only when this fails are tuples and candidate accessed. These prunings come from the properties of hash-partitioning, projection-based pruning, intersection-based pruning discussed in Section 2. Let us consider such prunings.

Match-based pruning for bucket ids. A candidate in a C-bucket of $Cand_{k-1}$ matches a tuple in a T-bucket only if the C-bucket matches the T-bucket. Thus, a T-bucket is relevant to computing the confidence for candidates in a C-bucket *only if* the C-bucket matches the T-bucket. This condition can be checked by examining the bucket ids involved.

Projection-based pruning for bucket ids. A C-bucket $[b_1, \ldots, b_{k-1}]$ of $Cand_{k-1}(X, C)$ (except longest ones) is non-empty *only if* some R-bucket $[b_1, \ldots, b_{k-1}, b'_i]$ is non-empty, where b'_i is for some attribute A_i not in X. Thus, we need to generate C-bucket $[b_1, \ldots, b_{k-1}]$ only if some R-bucket $[b_1, \ldots, b_{k-1}, b'_i]$ is non-empty. Since we never keep empty buckets, this condition can be checked efficiently.

Intersection-based pruning bucket ids. A C-bucket $[b_1, \ldots, b_{k-1}]$ of $Cand_{k-1}(X, C)$ (except longest ones) is non-empty only if, for every attribute A_i not in X, some R-bucket $[b_1, \ldots, b_{k-1}, b'_i]$ is non-empty, where each b'_i is for A_i. Thus, we need to generate C-bucket $[b_1, \ldots, b_{k-1}]$ only if this condition is satisfied. Again, this can be checked efficiently.

We can represent the "matching" relationship between C-buckets and T-buckets by the hypergraph H_{k-1} defined below.

Hypergraph H_{k-1}: A vertex corresponds to a T-bucket and a hyperedge corresponds to a (non-empty) C-bucket of $Cand_{k-1}$. A hyperedge contains a vertex if and only if the corresponding C-bucket matches the corresponding vertex. The terms "vertex" and "T-bucket" are interchangeable, and so are "hyperedge" and "C-bucket". We keep H_{k-1} in memory because it contains only bucket ids, not buckets themselves.

3.1 Computing the seed $Rule_m$

Conceptually, a grouping operation on T by all non-class attributes can find all the rules with the same left-hand-side. For each group of size n, the confidence of each rule in the group is $1/n$. Since we have partitioned T into T-buckets, we only need to perform the grouping operation on each T-bucket. Thus, we read each T-bucket into memory, one at a time, perform the in-memory group-by, and write the confident m-rules as a R-bucket of $Rule_m$.

3.2 Generating candidates $Cand_{k-1}$

We generate the C-buckets of $Cand_{k-1}$ from the R-buckets of $Rule_k$ in two steps.

```
Step 1:
1. for each bucket id $[b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_k]$ of $Rule_k$ do
2.     add tuples $< b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_k, A_i >$ to $W_{k-1}$,
       $1 \leq i \leq k$, $A_i$ being the attribute of $b_i$;
3.     group the tuples in $W_{k-1}$ by the first $k-1$ fields;
4.     prune all groups of size less than $m-k+1$;
Step 2:
5. for each group with the first $k-1$ fields $b_1, \ldots, b_{k-1}$ do
6.     intersect the unions represented by the tuples in the group;
7.     if the result is not empty then
8.         add the result as bucket $[b_1, \ldots, b_{k-1}]$ of $Cand_{k-1}$;
```

Figure 2: Generating candidates in $Cand_{k-1}$

Step 1. This step prunes empty buckets of $Cand_{k-1}$ based on the intersection-based pruning for bucket ids discussed earlier. Thus, no I/O is involved. For each (non-empty) bucket id $[b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots, b_k]$ of $Rule_k$, we create k tuples $< b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_k, A_i >$, $1 \leq i \leq k$, where A_i is the attribute of b_i. Let W_{k-1} denote this set of tuples. We group the tuples in W_{k-1} by the first $k-1$ fields and prune the groups of size less than $m-k+1$. For each surviving group (of size $m-k+1$) with the first $k-1$ fields $b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_k$, we add $[b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_k]$ as a bucket id of $Cand_{k-1}$.

Step 2. This step generates the actual bucket for each bucket id of $Cand_{k-1}$ determined in Step 1. Assume that a bucket id $[b_1, \ldots, b_{k-1}]$ determined in Step 1 comes from the group:

$$< b_1, \ldots, b_{k-1}, A_{i_1} >, \ldots, < b_1, \ldots, b_{k-1}, A_{i_{m-k+1}} >.$$

Each $< b_1, \ldots, b_{k-1}, A_{i_j} >$ represents the union of all the R-buckets $[b_1, \ldots, b_{k-1}, A_{i_j}.*]$ of $Rule_k$, where $A_{i_j}.*$ denotes any hash value on A_{i_j}. The C-bucket $[b_1, \ldots, b_{k-1}]$ is computed by intersecting all these unions over the first $k-1$ fields. If the intersection result is not empty, we write it as C-bucket $[b_1, \ldots, b_{k-1}]$ onto disk.

The algorithm for generating $Cand_{k-1}$ is given in Figure 2.

3.3 Computing the confidence of candidates

To compute the confidence for candidates in $Cand_{k-1}$, we read a block of several T-buckets at a time, subject to the memory space allocated for holding T-buckets), and for the T-buckets currently in memory, we access all the matching C-buckets. We make use of the match-based pruning for bucket ids to determine which C-buckets are accessed. In H_{k-1}, these C-buckets correspond to the hyperedges that contain one or more vertices corresponding to the T-buckets in memory.

In details, we partition the memory into two buffers, M_1 and M_2. M_1 is used for reading several T-buckets, called a *T-block*. M_2 is used for holding C-buckets that are read. A larger M_2 reduces the repeated I/O access by keeping more C-buckets in memory, but also reduces the sharing of accessed C-buckets because fewer T-buckets can be read in one T-block. A read C-bucket is kept in M_2 if there is free space in M_2. When a C-bucket is requested, a disk read is performed only if the C-bucket is not in M_2. A replacement policy can be adopted for replacing C-buckets in M_2. For simplicity, we replace a C-bucket in M_2 only if it is no longer needed. In our algorithm, this condition arises when the hyperedge for the C-bucket becomes empty.

A C-bucket *matches* a T-block if it matches some T-bucket in the T-block. For each T-block, we need to access all matching C-buckets for updating the confidence information. Therefore, a C-bucket on disk will be read as many times as the number of matching T-blocks. The key to reducing the I/O cost is to group T-buckets into T-blocks in such a way that the disk read of C-buckets is minimized. (The disk read of T-buckets is always the same.) Let us formalize this optimization problem.

Definition 3.1. Given a set of T-buckets, a set of C-buckets, and sizes M_1 and M_2, an *optimal blocking* is a sequence $< B_1, \ldots, B_t >$ of T-blocks such that $size(B_i) \leq M_1$ and the I/O cost of accessing C-buckets is minimized, where size(B_i) denotes the total number of tuples in the T-buckets contained in B_i. □

Finding an optimal blocking is feasible only for applications of a trivial size. In practice, it suffices to find an approximate solution, but efficiently. One approximation is to ignore the ordering of T-blocks, thereby, simplifying the above problem into the *hypergraph partitioning* of H_{k-1}: find a partitioning $\{B_1, \ldots, B_t\}$ of the set of vertices such that $size(B_i) \leq M_1$ and if E is the set of hyperedges that contain a vertex from different B_i, $\sum_{e \in E} disk(e)$ is minimized, where $disk(e)$ denotes the I/O cost for reading the C-bucket corresponding to e. $disk(e) = 0$ if the C-bucket is M_2. The hypergraph partitioning minimizes the total size of the C-buckets that are read more than once. To account for the number of times that a C-bucket is read, we can replace $\sum_{e \in E} disk(e)$ with $\sum_{e \in E} times(e) * disk(e)$, where $times(e)$ is the number of B_i such that $B_i \cap e \neq \emptyset$. However, even the hypergraph partitioning is NP-hard. For a body of heuristic algorithms, see [7].

We are interested in simple heuristics for determining the next T-block to read. We consider two such heuristics.

Heuristic I: *The more T-buckets match a C-bucket, the higher priority such T-buckets should be included in the next T-block.* The number of T-buckets that match the same C-bucket is the maximum number of times that the C-bucket will be read in the worst case; by reading the T-buckets that maximize this number in the next T-block, we hope to eliminate this worst case.

Heuristic II. *The more C-buckets matches a T-bucket, the higher priority this T-bucket should be included in the next T-block.* The rationale is to maximize the number of C-buckets matched by a T-block. By maximizing this number, we hope to eliminate the worst case that these C-buckets are read repeatedly for different T-blocks.

4. EXPERIMENTS

We have conducted experiments to evaluate the proposed algorithm. We evaluated: 1) the effectiveness of pruning strategies; 2) the effectiveness of partitioning scheme, blocking heuristics, buffer allocation; 3) the scalability of with respect to the size and dimension of databases.

We compared our algorithm with Dense-Miner [4]. To our knowledge, Dense-Miner is the only algorithm that can be used for finding general confident rules without a support requirement. [5, 6] finds only confident 1-rules where the left-hand-side contains a single value, whereas we find confident k-rules for all size k, which are substantially more than confident 1-rules. As such, these two cases are not really comparable. Our experiments were conducted on PIII 500 PC with 512MB memory and Windows NT Server 4.0.

4.1 The experimental setup

We selected the synthetic databases generated by the generator in [1]. This choice gives us the flexibility of controlling the size and dimension of the database. The default attributes of this database are shown in Table 1 plus one class attribute (not shown). The first three columns were copied from [1]. Attributes *elevel*, *car* and *zipcode* are categorical and the other attributes are non-categorical. Each tuple has a class value generated using one of the classification functions documented in the source code. We used the classification function that determines the class value of each person using the intervals of three attributes, namely, *age*, *salary*, and *loan*. In all experiments, except for the scalability study, we set the number of tuples at 100K and the number of classes at 10. We discretized non-categorical values using the *equal-width interval partitioning* and replaced non-categorical values by their corresponding intervals. The numbers of intervals for non-categorical attributes are shown in the last column in Table 1.

4.2 The effect of pruning strategies

We compare several search spaces defined below. "Proj+Inter" refers the number of candidates generated by applying both the projection-based pruning and the intersection-based pruning. "Proj" refers the number of candidates generated by applying only the projection-based pruning. "Dense-Miner" refers to the number of candidates generated by Dense-Miner. "Confident rules" refers to the number of confident rules. The difference between "Proj+Inter" and 'Proj" represents the effectiveness of the intersection-based pruning. The difference between "Proj+Inter" or 'Proj" and 'Dense-Miner" represents the effectiveness of our pruning strategies compared to Dense-Miner. The difference between "Proj+Inter" or 'Proj" and "Confident rules" represents the tightness of our pruning strategies. We do not have "Inter" because the intersection-based pruning must use the projection-based pruning.

Figure 3(a) shows the search space vs various minimum confidence, and Figure 3(b) shows the search space for each iteration at minimum confidence of 80%. The iteration No. is named by the size of rules. Thus, our algorithm proceeds from iteration No. 9 to iteration No. 1, whereas Dense-Miner proceeds in the opposite direction. For Dense-Miner, we have to stop the algorithm after 4 hours running, which only reached iteration No. 5. This explains why the curve

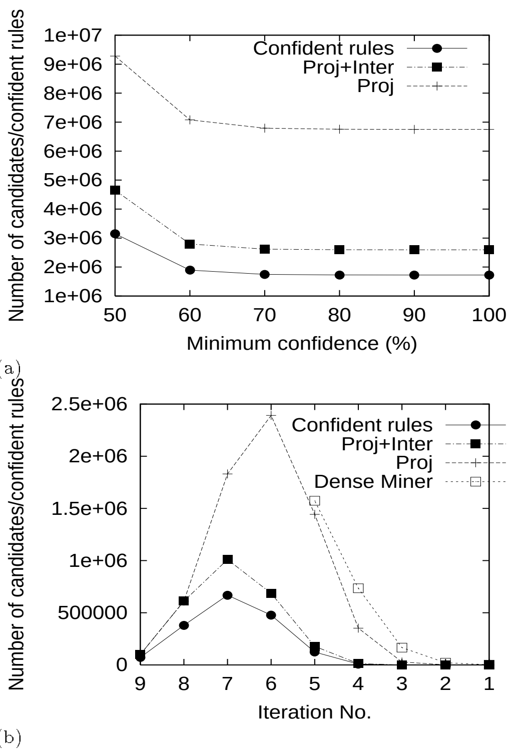


Figure 3: The effect of pruning strategies

"Dense-Miner" is not shown in Figure 3(a). There are several findings from this experiment.

First, the iteration-by-iteration comparison in Figure 3(b) shows that Dense-Miner generated far more candidates than "Proj+Inter". For example, at minimum confidence of 80%, Dense-Miner runs out of memory after 14 minutes when it just started iteration No. 5. From Figure 3(b), most confident rules have a large size and are generated between iteration No. 9 to 5. In such cases, Dense Miner will have to examine all the shorter rules, confident or not, that have confident specializations. In contrast, our algorithm examines only confident rules for generating candidates. Thus, the downward candidate generation based on the universal-existential upward closure of confidence is indeed an effective strategy.

Second, the big difference between "Proj" and "Proj+Inter" shows that the intersection pruning is highly effective. Without the intersection pruning, the number of candidates generated is close to that of Dense-Miner (for iteration No 1 to 5). In early iterations, say 9 to 7, there are more possible rules but fewer projections participating in the intersection pruning. This explains the quick increase in the number of candidates in early iterations. In fact, the intersection pruning takes effect only after the first 2 iterations: at iteration No. 9, no candidate is pruned by the intersection pruning because there is no A_i-specialization; at iteration No. 8, no candidate is pruned by the intersection pruning because only one projection participates in the intersection. For these iterations, only the projection pruning is in effect, thus, "Proj" and "Proj+Inter" coincide. However, after the first two iterations, the intersection pruning takes a strong

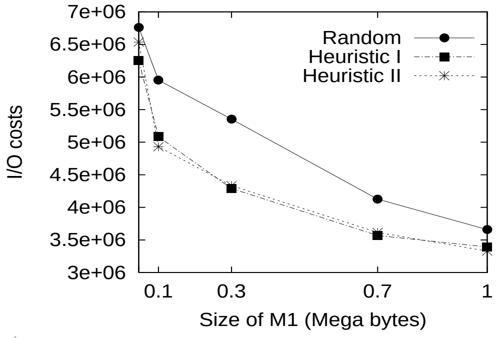
Attribute	Description	Domain	Number of attribute values
salary	salary	uniformly distributed from 20,000 to 150,000	3
commission	commission	$salary \geq 75{,}000 \Longrightarrow commission = 0$ else uniformly distributed from 10,000 to 75,000	8
age	age	uniformly distributed from 20 to 80	4
elevel	education level	uniformly chosen from 0 to 4	5
car	make of the car	uniformly chosen from 1 to 20	20
zipcode	zip code of the town	uniformly chosed from 9 available zipcodes	9
hvalue	value of the house	uniformly distributed from $0.5k100{,}000$ to $1.5k100{,}000$, where $k \in \{0 \dots 9\}$ depends on *zipcode*	5
hyears	years house owned	uniformly distributed from 1 to 30	3
loan	total loan amount	uniformly distributed from 0 to 500,000	2

Table 1: Description of attributes.

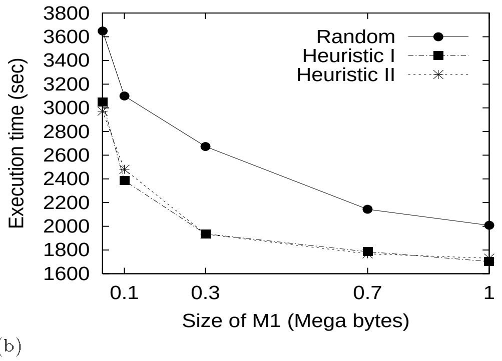
effect by pruning a large portion of candidates.

Third, the comparison of "Proj+Inter" and "Confident rules" shows that the ratio of the number of candidates generated over the number of confident rules is about 3/2. In other words, 2 out of every 3 candidates generated are actually confident rules! This shows that our search space is indeed rather tight, thanks to the effective pruning strategies.

In the following, we study the efficiency of our algorithm in terms of the I/O cost and the execution time, and the scalability of the algorithm for large databases. The I/O cost refers to the number of candidates accessed from disk.



(a)



(b)

Figure 4: The effect of buffer allocation

4.3 The effect of buffer allocation

We study how the buffer allocation of M_1 and M_2, for a fixed memory size, affects the algorithm. Recall that M_1 holds one T-block and M_2 holds some C-buckets during computing the confidence of candidates, and both hold some R-buckets during generating candidates. In this experiment, the memory size is fixed at 1MB, and the size of M_1 is varied from the maximum size of T-buckets, which is 0.046MB, to 1MB; the rest is used for M_2. The minimum confidence is set at 80% and the number of partitions at 100 (see Table 2 for the detail of partitioning). The size of the database is 100K tuples, which takes up 2.2MB. The space required by C-buckets in a single iteration can be as large as 21MB. Larger databases will be considered in the study of scalability shortly. Figure 4(a)(b) shows the I/O cost and the execution time. "Random" refers to no use of blocking heuristic, and "Heuristic I" and "Heuristic II" refer to the blocking heuristics discussed in Section 3.

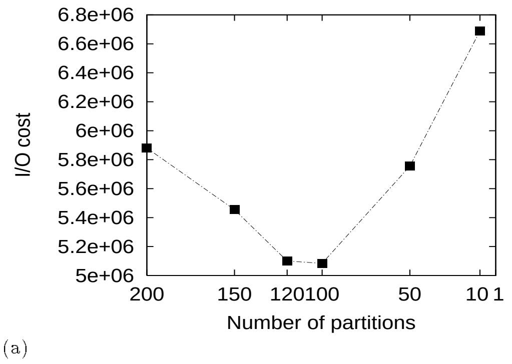
First, Figure 4(a) shows that, as the size of M_1 increases, the I/O cost of accessing the candidates decreases. With a larger M_1, more T-buckets can be read in one T-block and so more I/O access to C-buckets can be shared by such T-buckets. Put differently, a larger M_1 leaves more room to the blocking heuristic to minimize the I/O access to C-blocks. In addition, a larger M_1 means fewer T-blocks, therefore, fewer scans of candidates in general. On the other hand, a larger M_1 also means a smaller M_2, thereby, more C-buckets on disk. Overall, however, the I/O cost is reduced by having a larger M_1. In fact, with the total size of C-buckets being much larger than the memory size (i.e., 21.97MB in the maximum case), the impact of buffering C-buckets to reduce the I/O cost is very limited. In such a case, reducing the number of scans of C-buckets by allocating more space to M_1 outweighs the benefit of buffering C-buckets.

4.4 The effect of blocking heuristics

Figure 4(a)(b) also shows the effectiveness of Heuristics I and II in terms of reducing the I/O cost and execution time. "Random" means that the a T-block is randomly determined without using any heuristic and that the average of 5 random selections is used. The experiments show that both heuristics are highly effective in reducing the I/O and execution time, compared to the random blocking. For a small M_1, each T-block contains fewer T-buckets, thus, the sharing of C-buckets among the T-buckets in memory is limited. As the size of M_1 increases, both heuristics have more

Total # of buckets	# of buckets per attribute		
	car	*zipcode*	*commission*
1	1	1	1
10	10	1	1
50	10	5	1
100	10	5	2
120	10	4	3
150	10	5	3
200	10	5	4

Table 2: The partitioning of attributes



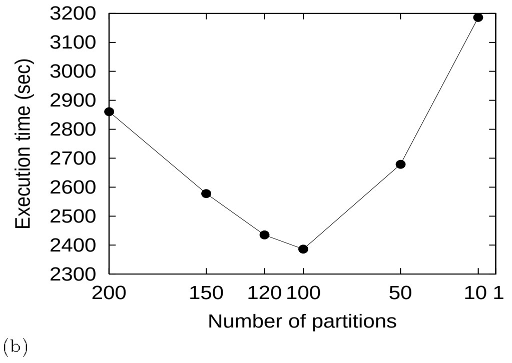
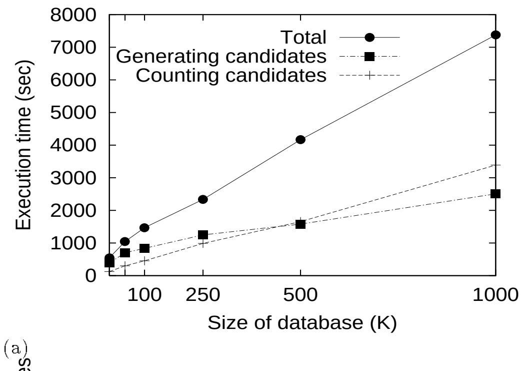


Figure 5: The effect of partitioning

room to decide what T-buckets should be read as a T-block for sharing the I/O access of C-buckets. For example, as M_1 varies from 0.046MB to 0.3MB, the average number of T-buckets in a T-block increases from 1.7 to 10 for Heuristic I. This effect, however, decreases as the size of M_1 gets close to the maximum size, i.e., 1MB. This is because, for very large T-blocks, the likelihood of sharing the same C-buckets is also decreased.

4.5 The effect of partitioning

In this experiment, we study how the hash-partitioning affects the algorithm. We use the minimum confidence of 80%, the memory size of 1MB, 10% of which is allocated to M_1, and Heuristic I. The hash function is *attr_val mod #bucket*. The three attributes with the most number of values are hashed, see Table 2 for the detail. Figure 5(a)(b) shows the I/O cost and execution time for the different partitionings. Both the I/O cost and execution time have a "V" shape as the number of buckets is varied from 200 to 10. For under-partitioning, i.e., a small number of buckets, reading a bucket involves more I/O. If there is no partitioning, for example, the whole candidate set $Cand_{k-1}$ has to been read for each T-block. It is less obvious why over-partitioning also increases the I/O and execution time. We observed that, when the database is over-partitioned, some "related T-buckets" tend to be read in different T-blocks, which in turn increases the cost of reading candidates. For example, suppose that only *commission*, not *zipcode*, is important for high confidence. It is possible that two T-buckets $[commission.c, zipcode.z1]$ and $[commission.c, zipcode.z2]$, which agree on *commission* but disagree on *zipcode*, are read in two different T-blocks. If this happens, the C-bucket $[commission.c]$ will be read once for each of these T-buckets. If we do not hash on *zipcode*, these T-buckets become one T-bucket and the above C-bucket will be read only once.



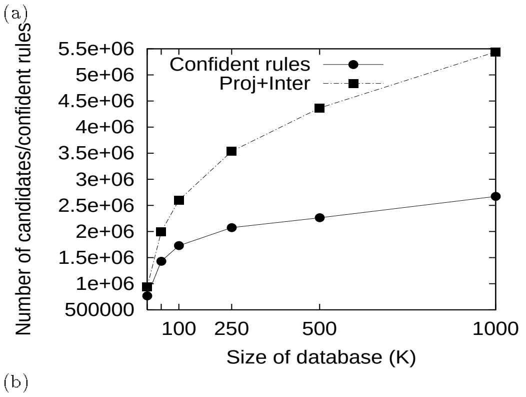
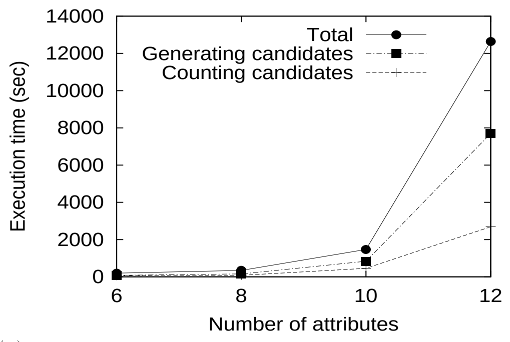


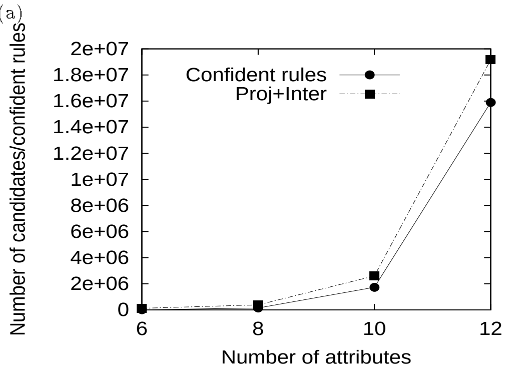
Figure 6: The scalability with the database size

4.6 The scalability

We set the minimum confidence at 80%, the memory size at 10MB, and M_1 at 10% of the memory. Figure 6(a)(b) shows the scalability with respect to the database size, which is varied from 10K to 1000K. As the database size increases, the execution time of the algorithm linearly increases. Figure 7(a)(b) shows the scalability with respect to the database dimensionality, with the database size fixed at 100K. We varied the number of dimensions from 6 to 12. The database of 8 dimensions is generated by removing the two attributes that have the smallest number of values, i.e., *hyears* and *loan*. The database of 6 dimensions is generated by removing the next two attributes that have the smallest number of values among the remaining attributes, i.e., *age* and *salary*. The database of 12 dimensions is generated by adding two attributes with uniformly distributed values chosen from 0 to 100,000 and discretizing them into two intervals. Each



(a)



(b)

Figure 7: The scalability with the database dimensionality

database is partitioned on attributes *car*, *zipcode*, *commission*, with the number of buckets being 10, 5, 2, respectively. The experiment shows that, as the dimension increases, there is a quick growth in both the number of candidates/confident rules and the execution time. The time for generating candidates dominates the growth.

5. RELATED WORK

Most work on mining association rules makes use of the support requirement to prune rules of small support [2, 3]. The problem of mining association rules without support requirement was recently considered in [5, 6] and [8]. [5, 6] considered tuples of 0/1 binary values and a very low 1-to-0 ratio, e.g., 1% or less (similar to a transaction in [2, 3]). More importantly, [5, 6] restricts rules to the form $a \to b$ where a and b are single values. In contrast, we considered attributes of arbitrary domains and rules of multiple values on the left-hand side. (We observed that most confident rules do contain more than one value on the left-hand side.) The explosion of the number of such rules presents a new challenge and demands a new method to deal with. In fact, if the counting method in [5, 6] is used, counters must be maintained for combinations of any number of columns, which is prohibitively large. We dealt with this problem by exploring a confidence-based level-wise pruning. The confidence-based pruning was proposed in [8]. However, [8] did not address the issue that memory may not hold the candidates/rules required to perform the pruning. In fact, the main topic of [8] is building classifiers using confident rules (without support requirement), not finding such rules for large databases. Experiments show that a straightforward disk-based implementation results in excessive I/O.

Dense_Miner [4] applies all of minimum support, minimum confidence, and minimum improvement to constraint the search space. Our experiments show that without a minimum support, Dense_Miner generates too many candidates. Also, the effectiveness of Dense_Miner critically depends on the tightness of the estimated bound, which in turn depends on whether items can be ordered so that unpromising rules are forced into the same portion of the enumeration tree.

6. CONCLUSION

Mining confident rules without support requirement was previously identified as an important problem. With only the confidence requirement available, the widely used support-based pruning strategy does not apply. We exploit a certain monotonicity of confidence, called the *universal-existential upward closure*, so that only confident rules of larger size need to be examined for generating confident rules of smaller size. This property yields a level-wise candidate generation with a confidence-based pruning. The main topic of this paper is to implement this pruning strategy in a disk-based environment. We addressed several performance related issues, namely, data partitioning and data blocking. Experiments show that the new pruning method often yields a very tight search space, in the sense that out of every three candidates generated, two are actually confident. This translates into a superior performance compared to existing methods.

7. REFERENCES

[1] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *VLDB*, pages 560–573, Sept 1992.

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

[3] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. In *VLDB*, pages 487–499, Sept 1994.

[4] R. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *ICDE*, pages 188–197, Feb 1999.

[5] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding interesting associations without support pruning. In *ICDE*, pages 489–499, Feb 2000.

[6] S. Fujiwara, J. D. Ullman, and R. Motwani. Dynamic miss-counting algorithms: finding implication and similarity rules with confidence pruning. In *ICDE*, Feb 2000.

[7] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. In *The 34th Design Automation Conference*, 1998.

[8] K. Wang, S. Zhou, and Y. He. Growing decision trees on support-less association rules. In *KDD*, pages 265–269, August 2000.

[9] K. Wang, S. Zhou, and S. Liew. Building hierarchical classifiers using class proximity. In *VLDB*, pages 363–374, Sept 1999.