Efficient Computation of Frequent Itemsets in A Subcollection of Multiple Set Families

Hong Shen[†*]   Weifa Liang[‡]

† School of Computing and Information Technology
Griffith University
Nathan, QLD 4111, AUSTRALIA `hong@cit.gu.edu.au`

‡ Department of Computer Science and Electrical Engineering
University of Queensland
St. Loucia, QLD 4072, AUSTRALIA `wliang@it.uq.edu.au`

March 22, 1998

Abstract

Many applications need to deal with the additive and multiplicative subcollections over a group of set families (databases). This paper presents two efficient algorithms for computing the frequent itemsets in these two types of subcollections respectively. Let T be a given subcollection of set families of total size m whose elements are drawn from a domain of size n. We show that if T is an additive subcollection we can compute all frequent itemsets in T in $O(m2^n/(pn)+\log p)$ time on an EREW PRAM with $1 \leq p \leq m2^n/n$ processors, at a cost of maintaining the occurrences of all itemsets in each individual set family. If T is a multiplicative subcollection, we can compute all itemsets in T in $O(mk/p+\min\{\frac{m'}{p}2^n, n3^n \log m'/p\})$ time on an EREW PRAM with $1 \leq p \leq \min\{m, 2^n\}$ processors, where $m' = \min\{m, 2^n\}$. These present improvements over direct computation of the frequent itemsets on the subcollection concerned.

1 Introduction

Given a family of sets T, e.g. a transaction database, each containing a set of items, a fundamental problem in data mining is to find all frequent (synonyms: large, interesting) itemsets in T with a support not smaller than a predefined *minimal support* (threshold), where the *support* of an itemset is the ratio of its frequency of occurrence in T to the size of T [2]. Centralized on the *apriori* approach [1], there have been various algorithms and parallel implementations proposed for this problem and its variants [8, 12, 5, 3, 9, 11, 6, 7, 4].

In many applications, we often need to compute the frequent itemsets *collectively* across a subcollection of several set families constructed in a well-defined way. For instance, community service in a large campus may need to know the consumption figure of a particular group of food

*This work was partially supported by The Australian Research Council under its Small Research Grant Scheme (1998).

across all shops on the campus by individual students. This requires an *additive* subcollection of the transaction databases of these shops. The traditional set *intersection* and *union*, with duplicates being counted, are two special cases of additive subcollection: intersection is the case when that group contains only one item, and union when the group covers all items in the whole database. On the other hand, a multi-category grade maintenance system in a university student administration system may require to list all students who have passed one course in each category satisfying a predefined course structure (patterns). This requires a *multiplicative* subcollection on all grade databases of individual courses. The standard relational-join in databases is a special case when the course structure covers all combinations of the courses from different categories. More formally, we have the following definition for these two types of subcollections.

Let $T_0, T_1, \ldots, T_{k-1}$ be k given set families, and R be a k-ary relation (e.g. k-parameter equation). Denote by $\oplus$ and $\otimes$ the operators for additive and multiplicative subcollections respectively. We define the additive subcollection $T_\oplus$ and multiplicative subcollection $T_\otimes$ on $T_0, T_1, \ldots, T_{k-1}$ as follows:

$$\begin{aligned} T_\oplus &= T_0 \oplus T_1 \oplus \cdots \oplus T_{k-1} = \{t_i \mid t_i \in T_i, 0 \le i \le k-1, R(t_0, t_1, \ldots, t_{k-1}) holds\}. & (1) \\ T_\otimes &= T_0 \otimes T_1 \otimes \cdots \otimes T_{k-1} = \{\langle t^0, t^1, \ldots, t^{k-1}\rangle \mid t^i \in T_i, 0 \le i \le k-1, \\ &\qquad R(t^0, t^1, \ldots, t^{k-1}) holds\}. & (2) \end{aligned}$$

In $T_\otimes$, each element is a k-tuple (sets) satisfying relation R. For any element $t_i = \langle t_i^0, t_i^1, \ldots, t_i^{k-1}\rangle$ in $T_\otimes$, we define $t \subseteq t_i$ iff $t \subseteq t_i^j$ for all $0 \le j \le k-1$.

In this paper, we consider an interesting problem of computing the frequent itemsets across a well-defined subcollection of additive and multiplicative types on a group of set families. We show how to compute these frequent itemsets efficiently by applying the relevant bit-vector operations. We organize the paper as follows. As the main technical body of the paper the next two sections present algorithms for computing the frequent itemsets in additive and multiplicative subcollections respectively. We conclude the paper in Section 4 with some open problems for future research.

2 Algorithm for the additive subcollection

As shown in [10], efficient parallel solution to the problem of finding all frequent itemsets in T_i requires to first compute the frequencies of all itemsets and then "filter" them according to the values of their frequencies. We now show how to apply this algorithm to the problem we are addressing and to produce efficient solution to our problem.

Let $T_\Sigma = T_0 \parallel T_1 \parallel \ldots \parallel T_{k-1}$, where "$\parallel$" is the operator of simple set concatenation. Assume that all elements of T_i are drawn from (itemset) *domain* I, i.e. $I = \cup_i T_i$, and let U contain all subsets generatable on I. Clearly U covers all possible subsets in any T_i and $|U| = 2^n - 1$. In our approach, we spend an extra space A of $|U||T|/d$ words, where d is the machine word-length, to record all frequent itemsets' occurrences in T. A is organized as an $|U| \times |T|$ bit-array, where $A[i][j] = 1$ if the ith itemset of U occurs in the jth element (set) of T, and $A[i][j] = 0$ otherwise. Computing A can be viewed as a precomputation which is invoked only once. We also assume that $T_\oplus = T_0 \oplus T_1 \oplus \ldots, \oplus T_{k-1}$ is given as input in the form of a bit-vector V of $|T|$ bits, where $V[i] = 1$ if $T[i] \in T_\oplus$ and $V[i] = 0$ otherwise. In most practical applications, $T_\oplus$ can be easily computed without incurring much cost because each T_i is usually stored in certain way of classification in the hard disk. The basic idea of our approach is to use these bit-vectors to reduce the computation.

Algorithm AddFrequentSets (A, V, L)
{*Input A and V, output L containing all frequent itemsets in $T_0 \oplus T_1 \oplus \ldots, \oplus T_{k-1}$.*}

for $i = 0$ **to** $|U| - 1$ **do**
1. $A'[i][0..|T|-1] := A[i][0..|T|-1] \wedge V[0..|T|-1]$;
2. Compute the number of 1's in $A'[i][0..|T|-1]$ and store it in $C[i]$;
3. **if** $C[i]/|T_\oplus| \geq \delta$ **then** $L := L \cup \{I_i\}$.

{*L is initialized to $\emptyset$.*}

We now analyze the time complexity of the algorithm. We assume that we are given a machine with word length d that can perform basic arithmetic and logic operations and also extracting the number of 1's in a single word in one step (constant time). The latter assumption is reasonable because extracting the number of 1's in a word should not be more difficult than performing an arithmetic/logic operation (e.g. mulitplication). Step 1 of the algorithm requires $O(|U||T|/d)$ time since the logic AND operator"$\wedge$" is carried out word-by-word throughout all $|U||T|/d$ words in A. Step 2 requires also $O(|U||T|/d)$ time for word-by-word extracting the number of 1's and adding each row's together. Step 3 takes only constant time. Therefore the whole algorithm requires $O(|U||T|/d)$ time.

We say that a computation model is *conservative* if it has a word of $\Theta(\log N)$ bits for processing data of magnitude N. Clearly this word length is the minimum for processing such data as it is required to store each datum in a single word and process it in a single step. Since $|U| = 2^n - 1$ and $|T| = m$, in our case a conservative machine should have $\max\{\log|U|, \log|T|\} \geq n$ bits. Throughout this paper, unless otherwise stated, all the computation models are conservative. Thus we have

Lemma 1 *For a group T of set families of total size m drawn from an domain of size n, an additive subcollection of T, by spending an extra $m2^n/n$ space to maintain all subset occurrences in T we can compute all frequent itemsets in a given additive subcollection of T in $O(m2^n/n)$ sequential time.*

If we are given an EREW PRAM with $1 \leq p \leq m2^n/n$ processors, it is easy to see that Steps 1 and 3 in the algorithm can be completed in $m2^n/(pn)$ time, while Step 2 requires $O(m2^n/(pn) + \log p)$ by parallel summation. This results in the following theorem:

Theorem 1 *Given an EREW PRAM with $1 \leq p \leq m2^n/n$ processors, if all subset occurrences in T are known, we can compute all frequent itemsets in a given additive subcollection of a group of set families of total size m drawn from a domain of size n in $O(m2^n/(pn) + \log p)$ time.*

The following corollary is immediate:

Corollary 1 *Algorithm AddFrequentSets can be completed in $O(n)$ time on an EREW PRAM with $m2^n/n^2$ processors.*

From [1] we know that directly computing all frequent itemsets in $T_\oplus$ rather than taking advantage of the frequencies of all itemsets requires $O(m2^n)$ sequential time and $O(m2^n/p + n\log m)$ parallel time using p processors for any p. So our algorithm is more efficient than this direct approach.

Consider the case when all sets in T are distinct, which algorithm AddFrequentSets can easily accommodate by deleting all duplicate elements and attaching a counter of duplicates to each element in T. In this case $m \leq 2^n - 1$, where $m = 2^n - 1$ when T contains all possible subsets of U. In this worst scenario, we can narrow the upper bound of time complexity of the apriori further:

Lemma 2 *When all sets in T are distinct, the apriori requires time $O(4^n/\sqrt{n})$ in the worst case, where n is the domain size from which all elements of T's are drawn.*

Proof

By [1], we know that the *apriori* approach works as follows. For $I = \bigcup T_i$ place all subsets of I in a lattice of $|I|+1$ levels with source $\emptyset$ at level 0 and sink I at level $|I|$, where nodes at level i represent all subsets containing i elements — i-sets, and there is an edge from node x at level i to node y at level $i+1$ if and only if $x \subset y$; starting from the source level-by-level compute the supports from T for all subsets at each level and delete those whose supports are smaller than δ together with all branches rooted at them (pruning).

Clearly, in the above scheme, when generating a new node at level i we must examine it with all nodes at level $i-1$ for subset inclusion so that all edges associated with this node can be added correctly. In the worst case when T contains all 2^n subsets of U (including the extra empty-set at the source), level i must contain all $\binom{i}{n}$ distinct i-sets constructed from I for all $0 \leq i \leq n$. Thus the total number of examinations for subset inclusion is in the lattice, that is, the possible maximal number of edges in the lattice, is given by the following equation:

$$\sum_{i=0}^{n-1} \binom{i}{n}\binom{i+1}{n} < \sum_{i=0}^{n} \binom{i}{n}^2 = \binom{n}{2n}. \tag{3}$$

Applying the Stirling's formula, we can approximate this number to $(\frac{2}{\pi n})^{1/2} 2^{2n}$. This results in the lemma. □

In this case, our algorithm requires $O(4^n/(pn) + \log p)$ time, which is clearly still more efficient than directly using apriori.

3 Algorithm for the multiplicative subcollection

In this section we consider the problem of computing all frequent itemsets in a multiplicative subcollection of set families, and present an efficient algorithm for it.

Let $T_\otimes = T_0 \otimes T_1 \otimes \ldots \otimes T_{k-1} = \{t_0, t_1, \ldots, t_{m-1}\}$, where $t_i = \langle t_i^0, t_i^1, \ldots, t_i^{k-1}\rangle$ and $t_i^j \in T_j$, $0 \leq i \leq m-1$, $0 \leq j \leq k-1$. $T_\otimes$ can be generated by invoking some standard database query operation, *join* for instance. As stated in Section 1, we define $t \subseteq t_i$ iff $t \subseteq t_i^j$ for all $0 \leq j \leq k-1$.

Assume that $T_\otimes$ is given by the user. A straightforward solution to our problem is to apply an existing algorithm, e.g. the *apriori* [1] or subset statistics [10], to find all frequent itemsets on $T_\otimes$ directly. This would require in the worst case $O(2^n km) = O(k2^{(k+1)n})$ time by [1], since m can be as large as 2^{kn}. This is obviously too expensive even for very small k, and therefore is not applicable in practice. We shall present an efficient algorithm that runs in a time which is almost same as required for computing all frequent itemset on a single set family when m is not excessively large.

Our algorithm is based on two observations. First, with the definition of set inclusion on t_i, an itemset occurs in t_i iff it occurs in the intersection of all t_i's component-sets t_i^j. So examining the occurrence of an itemset in t_i is concluded to examining its occurrence in the corresponding intersection-set. Second, no matter how large m is, all the component-sets of t_i for all i have the same domain D of size n, so there can be at most 2^n different intersection-sets among all t_i's. Therefore we need only to consider at most 2^n such sets while taking the number of duplicates of each set into account when counting the occurrences of each set.

We use an $m \times k \times n$ bit-array T to represent $T_\otimes$, where $T[i][0..k-1][0..n-1]$ represents t_i, $T[i][j][0..n-1]$ represents t_i^j. $T[i][j][h] = 1$ indicates that t_i^j contains the hth element in the domain – $D[h]$, $T[i][j][h] = 0$ indicates $D[h] \notin t_i^j$. Our algorithm is described as follows:

Algorithm MultiFrequentSets (T, D, L)
{*Input T representing $T_0 \otimes T_1 \otimes \ldots, \otimes T_{k-1}$ and itemset domain D, output L containing all frequent itemsets in T.*}

1. **for** $i = 0$ **to** $m-1$ **do**
   for $j = 0$ **to** $k-1$ **do**
   $T'[i][0..n-1] := T'[i][0..n-1] \wedge T[i][j][0..n-1]$;
   {*Compute $T'[i][0..n-1]$ representing $t'_i = t_i^0 \cap t_i^1 \cap \cdots \cap t_i^{k-1}$, where $T'[i][0..n-1]$ is initialized to 1, $0 \le i \le m-1$.*}

2. **for** $i = 0$ **to** $m-1$ **do**
   Distribute $T'[i][0..n-1]$ to the bucket with index of value $T'[i][0..n-1]$;
   {*There are 2^n buckets, each corresponding a subset on D.*}

3. **for** $i = 0$ **to** $2^n - 1$ **do**
   if the ith bucket is not empty **then**
   Collect the contents of this bucket to T'' and keep the number of duplicates (size of the bucket) in dup_i;
   {*T'' is an $m' \times n$ bit array containing all distinct sets of T', $m' \le \{m, 2^n\}$.*}

4. **for** $i = 0$ **to** $m' - 1$ **do**
   $D'[0..n-1] := D'[0..n-1] \vee T''[i][0..n-1]$;

5. $D'[0..n'-1] := D'[0..n-1]$ after deleting all 0 bits in $D'[0..n-1]$;
   {*D' initialized to 0, is the itemset domain of T'', $n' \le n$.}

6. Compute all frequent itemsets of T'' on D', where an itemset t's occurrence in any set $T''[i][0..n-1]$ is dup_i instead of 1 as usual counting in [1], if $t \subseteq T'[i][0..n-1]$.

The correctness of the algorithm can be seen from the comments interspersed with the algorithm. We proceeds with analysis of its time complexity. Assume that we are given a conservative machine with word length of n bits. Step 1 of the algorithm takes $O(mk)$ time. Steps 2 and 3 require time $O(m)$ and $O(m+2^n)$ respectively. Steps 4 and 5 require time $O(m') =$

$O(\min\{m, 2^n\})$ and $O(n)$ respectively. Step 6 can be done either in $O(m'2^{n'}) = O(m'2^n)$ time using the algorithm of [1], or in $O(n'3^{n'}\log m') = O(n3^n \log m')$ time using the result of [10], where $m' \le \min\{m, 2^n\}$. So we have

Lemma 3 *Given a multiplicative subcollection $T_\otimes$ of size m of k set families on a domain of size n, all frequent itemsets in $T_\otimes$ can be computed in $O(mk + \min\{m'2^n, n3^n \log m'\})$ time sequentially, where $m' = \min\{m, 2^n\}$.*

In the parallel environment, assume that we are given an EREW PRAM with $1 \le p \le \min\{m, 2^n\}$ processors. Then Step 1 can be done in $O(mk/p)$ time. Steps 2 and 3 need $O(1)$ and $O(\log m\lceil 2^n/m\rceil)$ time respectively. Steps 4 and 5 take $O(m'/p + \log m')$ and $O(\log n)$ time respectively. Step 6 requires $O(\frac{m'}{p}\sum_{i=0}^{n'-1}\binom{n'}{i}) = O(\frac{m'}{p}2^{n'})$ time by [1], since at level $i+1$ of the lattice [10] each processor holding a block of m'/p $(i+1)$-sets of T'' needs to examine each of these sets with all $\binom{n'}{i}$ i-sets at level i for subset inclusion. This step can also be done in $O(n'3^{n'}\log m'/p)$ time by [10]. Hence we have the following theorem:

Theorem 2 *Given a multiplicative subcollection $T_\otimes$ of size m of k set families on a domain of size n, we can compute all frequent itemsets in $T_\otimes$ in $O(mk/p + \min\{\frac{m'}{p}2^n, n3^n \log m'/p\})$ time on an EREW PRAM with $1 \le p \le \min\{m, 2^n\}$ processors, where $m' = \min\{m, 2^n\}$.*

4 Concluding remarks

Given a group of set families (databases), two types of subcollections of them encountered in practical applications, the additive subcollection and multiplicative subcollection, have been studied. This paper addresses the problem of how to compute all frequent itemsets whose occurrences in these two types of subcollections exceed a predefined threshold. We have proposed two efficient algorithms. The first algorithm computes all frequent itemset in additive subcollection that uses bit-vector techniques and takes advantage of the occurrences of all itemsets in each individual set family. The second algorithm computes all frequent itemset in the multiplicative subcollection by first computing the intersection of all component-sets of each set and then computing the frequent itemsets on the resulting set of intersection sets after necessary simplification. Both algorithms are considerably more efficient than their counterparts of straightforward approaches to solving these two problems respectively, in both sequential and parallel environments.

Future research includes studies on efficient algorithms for finding all frequent itemsets in other types of collections and combinations of set families.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In Jorgeesh Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 487–499, Los Altos, CA 94022, USA, 1995. Morgan Kaufmann Publishers.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):207–216, June 1993.

[3] Luc Dehaspe and Luc De Raedt. Mining association rules in multiple relations. In Nada Lavrač and Sašo Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *LNAI*, pages 125–132, Berlin, September17–20 1997. Springer.

[4] A. A. Freitas and S. H. Lavington. Parallel data mining for very large relational databases. *Lecture Notes in Computer Science*, 1067:158–, 1996.

[5] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Toluyama. Data mining using two-dimensional optimized association rules: scheme, algorithms, and visualization. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):13–, 1996.

[6] Eui-Hong Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):277–288, 1997.

[7] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, University of Maryland, College Park, August 1995.

[8] Jong Soo Park, Ming-Syan Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):175–186, June 1995.

[9] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB '95: proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, Sept. 11–15, 1995*, pages 432–444, Los Altos, CA 94022, USA, 1995. Morgan Kaufmann Publishers.

[10] H. Shen. Fast parallel subset statistics and its applications in data mining. Technical report, 1998.

[11] R. Srikant and R. Agrawal. Mining generalized association rules. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB '95: proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, Sept. 11–15, 1995*, pages 407–419, Los Altos, CA 94022, USA, 1995. Morgan Kaufmann Publishers.

[12] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):1–, 1996.