Calculating a New Data Mining Algorithm for Market Basket Analysis

Zhenjiang Hu[1] Wei-Ngan Chin[2] Masato Takeichi[1]

[1]Department of Information Engineering
University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113, Japan
Email: {hu,takeichi}@ipl.t.u-tokyo.ac.jp

[2]Department of Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
Email: chinwn@comp.nus.edu.sg

Abstract

The general goal of data mining is to extract interesting correlated information from large collection of data. A key computationally-intensive subproblem of data mining involves finding frequent sets in order to help mine association rules for market basket analysis. Given a bag of sets and a probability, the frequent set problem is to determine which subsets occur in the bag with some minimum probability. This paper provides a convincing application of program calculation in the derivation of a new and fast algorithm for this practical problem. Beginning with a simple but inefficient specification expressed in a functional language, the new algorithm is calculated in a systematic manner from the specification by applying a sequence of known calculation techniques.

Keywords: Program Transformation/Calculation, Functional Programming, Frequent Set Problem, Data Mining, Algorithm Derivation.

1 Introduction

Program derivation has enjoyed considerable interests over the past two decades. Early work concentrated on deriving programs in imperative languages, such as Dijkstra's Guarded Command Language, but nowadays functional languages are increasing popular as they offer a number of advantages over imperative ones.

- Functional languages are so abstract that they can express the specifications of problems in a more concise way than imperative languages, resulting in programs that are shorter and easier to understand.
- Functional programs can be constructed, manipulated, and reasoned about, like any other kind of mathematics, using more or less familiar known algebraic laws.
- Functional languages can often be used to express both clear specification and its efficient solution, so the derivation can be carried out within a single formalism. In contrast, the derivation for imperative languages often rely on a separate (predicate) calculus for capturing both specification and program properties.

Such derivation in a single formalism is often called *program calculation* [Bir89, BdM96], as opposed to simply program derivation. Many attempts have been made to apply the program calculation for the derivation of various kinds of efficient programs [Jeu93], and for the construction of optimization passes of compilers [GLJ93, OHIT97]. However, people are still expecting more *convincing* and *practical* applications where program calculation can give a better result, while other approaches could falter.

This paper aims to illustrate a practical application of program calculation, by deriving a new algorithm to solve the problem for finding frequent sets - an important building block for derivation of association rules [AIS93, AS94, Mue95, MT96, ZPOL97a, Zak99] which is important for market basket analysis. In this problem, we are given a set of items and a large collection of transactions which are essentially subsets of these items. The task is to find all sets of items that occur in the transactions frequently enough - exceeding a given threshold. More concrete explanation of the problem can be found in Section 2.

The most well-known classical algorithm for finding frequent set is the Apriori algorithm [AIS93, AS94] (from which many improved versions have been proposed) which relies on the property that a set can only be frequent if and only if all of its subsets are frequent. This algorithm builds the frequent sets in a level-wise fashion. Firstly, it counts all the 1-item sets (sets with a single item), and identifies those counts which exceed the threshold, as frequent 1-item sets. Then it combines these to form candidate (potentially frequent) 2-item sets, counts them in order to determine the frequent 2-item sets. The counting process will re-traverse the entire database and needs to check each transaction against the candidate 2-item sets. It continues by combining the frequent 2-item sets to form candidate 3-item sets, counting them before determining which are the frequent 3-item sets, and so forth. The Apriori algorithm stops when there are no more frequent n-set found.

Two important factors, which govern the performance of this algorithm, are the number of passes made over the transactions, and the efficiency of each of these passes.

- The database that records all transactions is likely to be very large, so it is often beneficial for as much information to be discovered from each pass, so as to reduce the total number of passes [BMUT97].
- In each pass, we hope that counting can be done efficiently and less candidates are generated for later check. This has led to the studies of different pruning algorithms as in [Toi96, LK98].

Two essential questions arise; what is the least number of passes for finding all frequent sets, and could we generate candidates that are so necessary that they will not be pruned later? Current researches in data mining, as far as we are aware, have not adequately address both these issues in a *formal* way. Instead, most of them have been focusing on the improvement of the Apriori algorithm. Two nice survey papers can be found in [Mue95, Zak99].

We shall show that program calculation indeed provides us with a nice framework to examine into these practical issues for data mining application. In this framework, we can start with a straightforward functional program that solve the problem. This initial program may be terribly inefficient or practically infeasible. We then try to improve it by applying a sequence of (standard) calculations such as fusion, tabulation, and accumulation, to reduce the number of passes and to avoid generating unnecessary candidates. As will be shown later in this paper, our program calculation can yield a new algorithm for finding frequent sets, which uses only a single pass of the database, while generating only necessary candidates during execution. Furthermore, the new algorithm is guaranteed to be correct with respect to the initial straightforward program due to our use of correctness-preserving calculation.

The rest of this paper is organized as follows. We begin by giving a straightforward functional program for finding frequent sets in Section 2. We then go to apply the known calculation techniques of fusion, accumulation, base-case filter promotion technique and tabulation to the initial functional program to derive an efficient program in Section 3. Discussion on the features and implementation issues of our derived algorithm are given in Section 4. Section 5 concludes the paper.

2 Specification

Within the area of data mining, the problem of deriving associations from data has received considerable attention [AIS93, AS94, Mue95, Toi96, BMUT97, ZPOL97a, ZPOL97b, Zak99], and is often referred to as the "market-basket" problem. One common formulation of this problem is finding association rules which are based on *support* and *confidence*. The support of an itemset (a set of items) I is the fraction of transactions that the itemset occurs in (is a subset of). An itemset is called *frequent* if its support exceeds a given threshold σ. An association rule is written as $I \rightarrow J$ where I and J are itemsets. The confidence of the rule is the fraction of the transaction I that also contains J. For the association rule $I \rightarrow J$ to hold, $I \cup J$ must be frequent and the confidence of rule must exceed a given confidence threshold, γ. Two important steps for mining association rules are thus:

- Find frequent itemsets for a given support threshold, σ.
- Construct rules that exceed the confidence threshold, γ, from the frequent itemsets.

Of these two steps, finding frequent sets is the more computationally-intensive subproblem, and have received the lion share of data mining community's attention. Let us now formalize a specification for this important subproblem.

Suppose that a shop has recorded the set of objects (items) purchased by each customer on each visit (transaction). The problem of finding frequent sets is to compute all subsets of objects that appear frequently in customers' visits with respect to a specific threshold. As an example, suppose a shop has the following object set:

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

and the shop recorded the following customers' visits:

visit 1:	$\{1, 2, 3, 4, 7\}$
visit 2:	$\{1, 2, 5, 6\}$
visit 3:	$\{2, 9\}$
visit 4:	$\{1, 2, 8\}$
visit 5:	$\{5, 7\}$

We can see that 1 and 2 appear together in three out of the five visits. Therefore we say that the subset $\{1, 2\}$ has frequency ratio of 0.6. If we set the frequency ratio threshold to be 0.3, then we know that the sets of

$$\{1\},\ \{2\},\ \{5\},\ \{7\} \text{ and } \{1, 2\}$$

pass this threshold, and thus they should be returned as the result of our frequent set computation.

To simplify our presentation, we impose some assumption on the three inputs, namely object set os, customers' visits vss, and threshold $least$. We shall represent the objects of interest using an ordered list of integers without duplicated elements, e.g.,

$$os = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$$

and represent customers' purchasing visits by a list of the sublists of os, e.g.,

$$vss = [[1, 2, 3, 4, 7], [1, 2, 5, 6], [2, 9], [1, 2, 8], [5, 7]].$$

Furthermore, for threshold, we will use an integer, e.g.,

$$least = 3$$

to denote the *least* number of appearances in the customers' visits, rather than using a probability ratio.

Now we can solve the frequent set problem straightforwardly by the following pseudo Haskell program[1]

$$\begin{array}{lcl} fs & :: & [Int] \rightarrow [[Int]] \rightarrow Int \rightarrow \{[Int]\} \\ fs\ os\ vss\ least & = & (fsp\ vss\ least) \triangleleft (subs\ os). \end{array}$$

It consists of two passes that can be read as follows.

1. First, we use *subs* to enumerate all the sublists of the object list *os*, where *subs* can be defined by

$$\begin{array}{lcl} subs & :: & [a] \rightarrow \{[a]\} \\ subs\ [\,] & = & \{[\,]\} \\ subs\ (x:xs) & = & subs\ xs \cup (x\,:) * subs\ xs. \end{array}$$

   We use the infix $*$ to denote the map function on sets: $f * s$ means to apply the function f to each element of the set s. Similar to the *map* function on lists [Bir89], it satisfies the so-called *map-distributivity* property (we use $\circ$ to denote function composition):

$$(f*) \circ (g*) = (f \circ g) * .$$

2. Then, we use the predicate *fsp* to filter the generated sublists to keep only those that appear frequently (exceeding the threshold *least*) in customers' visits *vss*. Such *fsp* can be easily defined by

$$\begin{array}{lcl} fsp & :: & [[Int]] \rightarrow Int \rightarrow [Int] \rightarrow Bool \\ fsp\ vss\ least\ ys & = & \#((ys \subseteq) \triangleleft vss) \geq least \end{array}$$

   Note that for ease of program manipulation, we use the shorten notation: # to denote the function *length* which computes the number of elements of a set, and $\triangleleft$ to denote the filter operator on set: $p \triangleleft s$ produces a new set whose elements are all from the set s but satisfy the predicate p. The filter operator enjoys the *filter-map* property (that is commonly used in program derivation e.g. [Bir84]):

$$(p\triangleleft) \circ ((x\,:)*) = ((x\,:)*) \circ ((p \circ (x\,:))\triangleleft) \tag{1}$$

   and the *filter-pipeline* property:

$$(p\triangleleft) \circ (q\triangleleft) = (\lambda x.(p\ x \wedge q\ x)) \triangleleft . \tag{2}$$

   In addition, $xs \subseteq ys$ is true if xs is a sublist (i.e., subset) of ys, and false otherwise:

$$\begin{array}{lcl} [\,] \subseteq ys & = & True \\ (x:xs) \subseteq ys & = & xs \subseteq ys \wedge x \in ys. \end{array}$$

So much for our specification program which is simple, straightforward, and easy to understand. No attention has been paid to efficiency or to implementation details. In fact, this initial functional program is practically infeasible for all but the very small object set, because the search space of potential frequent sets consists of $2^{\#os}$ sublists.

3 Derivation

We shall demonstrate how the exponential search space of our initial concise program can be reduced dramatically via program calculation. Specifically, we will derive an *efficient* program for finding frequent sets from the specification

$$fs\ os\ vss\ least \quad = \quad (fsp\ vss\ least) \triangleleft (subs\ os)$$

[1] We assume that the readers are familiar with the Haskell language [Bir98] in this paper. In addition, we say that our Haskell programs are "pseudo" in the sense that they include some additional notations for sets.

by using the known calculation techniques of fusion [Chi92], generalization (accumulation) [Bir84, HIT99], base-case filter promotion [Chi90], and tabulation [Bir80, CH95].

Our derivation strategy is rather standard for program optimization and can be summarized as follows. First, we eliminate as much unnecessary intermediate data structures as possible by fusion of composition of recursive functions into a single recursion function. Then, we reuse necessary intermediate data structures by generalization, and we filter out unnecessary recursive calls by base-case filter promotion transformation. Finally, we reuse partial results to produce the whole result by tabulation transformation.

3.1 Fusion

Fusion [Chi92] is used to merge two passes (from nested recursive calls) into a single one, by eliminating intermediate the data structure passing between the two passes. Notice that our fs has two passes, and the intermediate data structure is huge containing all the sublists of os. We shall apply the fusion calculation to eliminate this huge intermediate data structure by the following calculation via an induction on os.

$$
\begin{array}{ll}
 & fs\ [\,]\ vss\ least \\
= & \quad \{ \text{ def. of } fs \ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ [\,]) \\
= & \quad \{ \text{ def. of } subs \ \} \\
 & (fsp\ vss\ least) \triangleleft \{[\,]\} \\
= & \quad \{ \ \text{def. of } \triangleleft \text{ and } fsp \ \ \} \\
 & \text{if } \#(([\,] \subseteq) \triangleleft vss) \geq least \text{ then } \{[\,]\} \text{ else } \{\,\} \\
= & \quad \{ \ \text{def. of } \subseteq \ \ \} \\
 & \text{if } \#((\lambda ys.True) \triangleleft vss) \geq least \text{ then } \{[\,]\} \text{ else } \{\,\} \\
= & \quad \{ \ \text{simplification} \ \ \} \\
 & \text{if } \#vss \geq least \text{ then } \{[\,]\} \text{ else } \{\,\}
\end{array}
$$

And

$$
\begin{array}{ll}
 & fs\ (o:os)\ vss\ least \\
= & \quad \{ \text{ def. of } fs \ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ (o:os)) \\
= & \quad \{ \text{ def. of } subs \ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ os \cup (o:) * (subs\ os)) \\
= & \quad \{ \ \text{def. of } \triangleleft \ \ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ os) \ \cup \\
 & (fsp\ vss\ least) \triangleleft ((o:) * (subs\ os)) \\
= & \quad \{ \ \text{eq. (1)} \ \ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ os) \ \cup \\
 & (o:) * ((fsp\ vss\ least \circ (o:)) \triangleleft (subs\ os)) \\
= & \quad \{ \ \text{eq. (3) later} \ \ \} \\
 & (fsp\ vss\ least) \triangleleft (subs\ os) \ \cup \\
 & (o:) * ((fsp\ ((o \in) \triangleleft vss)\ least) \triangleleft (subs\ os))
\end{array}
$$

To complete the above calculation, we need to show that

$$fsp\ vss\ least \circ (o:) \ = \ fsp\ ((o \in) \triangleleft vss)\ least. \tag{3}$$

This can be easily proved by the following calculation.

$$
\begin{array}{ll}
& fsp\ vss\ least \circ (o :) \\
= & \quad \{ \text{ def. of } fsp \ \} \\
& (\lambda ys.(\#((ys \subseteq) \triangleleft vss) \geq least)) \circ (o :) \\
= & \quad \{ \text{ function composition } \} \\
& \lambda ys.(\#(((o : ys) \subseteq) \triangleleft vss) \geq least) \\
= & \quad \{ \text{ def. of } \subseteq \ \} \\
& \lambda ys.(\#((\lambda xs.(ys \subseteq\ xs\ \wedge\ o \in\ xs)) \triangleleft vss) \geq least) \\
= & \quad \{ \text{ eq. (2) } \} \\
& \lambda ys.(\#((ys \subseteq) \triangleleft ((o \in) \triangleleft vss)) \geq least) \\
= & \quad \{ \text{ def. of } fsp \ \} \\
& fsp\ ((o \in) \triangleleft vss)\ least
\end{array}
$$

To summarize, we have obtained the following program, in which the intermediate result used to connect the two passes have been eliminated; many unnecessary subsets produced by *subs* have been removed.

$$
\begin{array}{lcl}
fs\ []\ vss\ least & = & \text{if } \#vss \geq least \text{ then } \{[]\} \text{ else } \{\} \\
fs\ (o : os)\ vss\ least & = & fs\ os\ vss\ least\ \cup \\
& & \underline{(o :)*}(fs\ os\ ((o \in) \triangleleft vss)\ least)
\end{array}
$$

The benefits of our fusion optimization can be compared to the technique of pass-bundling [Mue95] which is used to eliminate some unnecessary candidates that end up infrequent in the partitioned parallel association rule algorithm. Compared to that in [Mue95], our study is more formal and general.

3.2 Generalization/Accumulation

Notice that the underlined part in the above program for insert o to every element of a list will be rather expensive if the the list consists of a large number of elements. Fortunately, this could be improved by introducing an accumulating parameter in much the same spirit as [Bir84, HIT99]. To this end, we generalize fs to fs', by introducing an accumulating parameter as follows.

$$fs'\ os\ vss\ least\ r = (r +\!\!+\ ) * (fs\ os\ vss\ least)$$

And clearly we have

$$fs\ os\ vss\ least = fs'\ os\ vss\ least\ [].$$

Calculating the definition for fs' is easy by induction on os, and thus we omit the detailed derivation. The end result is as follows.

$$
\begin{array}{lcl}
fs'\ []\ vss\ least\ r & = & \text{if } \#vss \geq least \text{ then } \{r\} \text{ else } \{\} \\
fs'\ (o : os)\ vss\ least\ r & = & fs'\ os\ vss\ least\ r\ \cup \\
& & fs'\ os\ ((o \in) \triangleleft vss)\ least\ (r +\!\!+ [o])
\end{array}
$$

The accumulation transformation has successfully turned an expensive map operator of $(o :)*$ into a simple operation that just appends o to r. In addition, we have got a nice side-effect from the accumulation transformation in that fs' is defined in an *almost* tail recursive form, in the sense that each recursive call produces independent part of the resulting list. This kind of recursive form is used by the base-case filter promotion technique of [Chi90].

3.3 Base-case Filter Promotion

From the second equation (inductive case) of fs', we can see that computation of

$$fs'\ os\ vss\ least\ r$$

still needs $2^{\#os}$ recursive calls to $(fs'\ []\ \ldots)$ after recursive expansion. In fact, not all these recursive calls are necessary for computing the final result, because the first equation (base case) of fs' shows that those recursive calls of $fs'\ []\ vss\ least\ r$ will not contribute to the final result if

$$\#vss < least.$$

The base-case filter promotion [Chi90] says that the base case condition could be promoted to be a condition for the recursive calls, which is very helpful in pruning unnecessary recursive calls. Applying the base-case filter promotion calculation gives the following program:

$$\begin{array}{lcl}
fs'\ []\ vss\ least\ r & = & \text{if } \#vss \geq least \text{ then } \{r\} \text{ else } \{\,\} \\
fs'\ (o:os)\ vss\ least\ r & = & (\text{if } \#vss \geq least \\
 & & \quad \text{then } fs'\ os\ vss\ least\ r \text{ else } \{\,\})\ \cup \\
 & & (\text{if } \#((o \in) \triangleleft vss) \geq least \\
 & & \quad \text{then } fs'\ os\ ((o \in) \triangleleft vss)\ least\ (r +\!\!+ [o]) \\
 & & \quad \text{else } \{\,\})
\end{array}$$

and accordingly fs changes to

$$fs\ os\ vss\ least = \text{if } \#vss \geq least \text{ then } fs'\ os\ vss\ least\ []\ \text{else } \{\,\}.$$

Now propagating the condition of $\#vss \geq least$ backwards from the initial call of fs' to its recursive calls, we obtain

$$\begin{array}{lcl}
fs'\ []\ vss\ least\ r & = & \{r\} \\
fs'\ (o:os)\ vss\ least\ r & = & fs'\ os\ vss\ least\ r\ \cup \\
 & & (\text{if } \#((o \in) \triangleleft vss) \geq least \\
 & & \quad \text{then } fs'\ os\ ((o \in) \triangleleft vss)\ least\ (r +\!\!+ [o]) \\
 & & \quad \text{else } \{\,\})
\end{array}$$

in which any recursive call $fs'\ os\ vss\ least\ r$ that does not meet the condition of $\#vss \geq least$ would be selectively pruned.

3.4 Tabulation

Although much improvement has been achieved through fusion, accumulation, and base-case filter promotion, there still remains a source of serious inefficiency because the inductive parameter os is traversed multiple times by fs'. We want to share some computation among all recursive calls to fs', by using the tabulation calculation [Bir80, CH95].

3.4.1 Tree Structure with Invariants

The purpose of our tabulation calculation is to exploit the relationship among recursive calls to fs' so that their computation could be shared. The difficulty in such tabulation is to determine which values should be tabulated and how these values are organized.

Taking a close look at the derived definition for fs':

$$\begin{array}{lcl}
fs'\ (o:os)\ \underline{vss}\ least\ \underline{r} & = & fs'\ os\ \underline{vss}\ least\ \underline{r}\ \cup \\
 & & (\text{if } \#((o \in) \triangleleft vss) \geq least \\
 & & \quad \text{then } fs'\ os\ \underline{((o \in) \triangleleft vss)}\ least\ \underline{(r +\!\!+ [o])} \\
 & & \quad \text{else } \{\,\})
\end{array} \quad (4)$$

we can see dependency of the second and the fourth arguments of fs' among the left and the right recursive calls to fs', as indicated by the underlined parts. Moreover these two arguments will be used to produce the final result, according to the base case definition of fs'. This hints us to keep (memoize) all *necessary* intermediate results of the second and the fourth parameters:

$$(r_1, vss_1), (r_2, vss_2), \ldots.$$

According to base-case filter promotion, each element (r_i, vss_i) should meet the invariant $\mathcal{I}_1$:

$$\#vss_i \geq least.$$

We could store all these pairs using a list. But a closer look at the second equation of fs' reveals that along with the extension of r, the corresponding number of vss decreases with each filtering. More precisely, for any two intermediate results of (r_i, vss_i) and (r_j, vss_j), $r_i \subseteq r_j$ implies $vss_j \subseteq vss_i$. An ideal way to make this relation explicit is to use a lattice (sort of graph), but graph manipulation would complicate the later derivation. Instead, we choose the following tree data structure for memoization:

$$Tree \quad = \quad Node\ ([Int], [[Int]])\ [Tree]$$

where each node, tagged with a pair storing (r_i, vss_i), can have any number of children. Two tree invariants are introduced to express the above relation:

1. The invariant $\mathcal{I}_2$ for parent-children: If the node (r_i, vss_i) is an ancestor of the node (r_j, vss_j), then $r_i \subset r_j$ and $vss_j \subseteq vss_i$.
2. The invariant $\mathcal{I}_3$ for siblings: For any node $I : (r_i, vss_i)$ having a child $C_i : (r_{c_i}, vss_{c_i})$, there must exist a node $J : (r_j, vss_j)$ which is a right sibling of I, such that $r_j \subset r_{c_i}$.

The invariant $\mathcal{I}_3$ is desired to allow trees to express shareness in lattice graphs while retaining the property that there is no same r appearing in different tree nodes. For example, in tree (a) of Figure 3, since $\{[1,2], \ldots\}$ is a child of $\{[1], \ldots\}$, we expected a sibling $\{[2], \ldots\}$ that will also be a frequent set in the tree. This invariant relationship can be used for further pruning.

3.4.2 Tabulation Transformation

To apply the tabulation calculation to fs', we define tab on tabulation tree to glue all possible recursive calls to fs' in Equation (4):

$$\begin{aligned} tab\ os\ least\ (Node\ (r, vss)\ ts) \quad &= \quad fs'\ os\ vss\ least\ r\ \cup \\ &\quad\quad flattenMap\ (tab\ os\ least)\ ts \end{aligned} \tag{5}$$

where list x to get a set as the result, and then merge all resulting sets. It is defined by $flattenMap\ f\ =\ foldr\ (\cup)\ \{\}\ \circ\ f*$. For the sake of later calculation, we note below several simple algebraic properties about $flattenMap$ and tab.

$$flattenMap\ (f \circ g) \quad = \quad flattenMap\ f\ \circ\ g* \tag{6}$$
$$flattenMap\ f\ (xs +\!\!+ [x]) \quad = \quad f\ x \cup flattenMap\ f\ xs \tag{7}$$
$$fs'\ os\ vss\ least\ r \quad = \quad tab\ os\ least\ (Node\ (r, vss)\ []). \tag{8}$$

In addition, tab enjoys the property:

$$tab\ (o : os)\ least\ t = tab\ os\ least\ t \tag{9}$$

provided that for any node (r, vss) in t, $\#((o \in) \triangleleft vss) < least$ holds. This follows directly from the definition of fs'.

Now we hope to synthesize a new definition that defines tab inductively on os while os is traversed only once (it is now traversed by both fs' and tab). To this end, we make use of the trick in [Bir84], and define tab by

$$tab\ []\ least\ t \quad = \quad select\ least\ t \tag{10}$$
$$tab\ (o : os)\ least\ t \quad = \quad tab\ os\ least\ (add\ o\ least\ t). \tag{11}$$

In the case where os is empty, tab uses $select$ to compute the result from the tree t; otherwise, tab uses add to update the tree t. Here $select$ and add are two newly introduced functions that are to be calculated.

We can synthesize *select* by the following calculation.

$$
\begin{array}{ll}
 & \mathit{tab}\ [\,]\ \mathit{least}\ (\mathit{Node}\ (r, \mathit{vss})\ \mathit{ts}) \\
= & \quad \{\ \text{eq. (5), we also know } \#\mathit{vss} \geq \mathit{least} \text{ from inv. } \mathcal{I}_1\ \} \\
 & \mathit{fs}'\ [\,]\ \mathit{vss}\ \mathit{least}\ r \cup \mathit{flattenMap}\ (\mathit{tab}\ [\,]\ \mathit{least})\ \mathit{ts} \\
= & \quad \{\ \text{def. of } \mathit{fs}',\ \#\mathit{vss} \geq \mathit{least} \text{ (above)}\ \} \\
 & \{r\}\ \cup\ \mathit{flattenMap}\ (\mathit{tab}\ [\,]\ \mathit{least})\ \mathit{ts} \\
= & \quad \{\ \text{eq. (10)}\ \} \\
 & \{r\}\ \cup\ \mathit{flattenMap}\ (\mathit{select}\ \mathit{least})\ \mathit{ts}
\end{array}
$$

We then match it with Equation (10) and get:

$$
\mathit{select}\ \mathit{least}\ (\mathit{Node}\ (r, \mathit{vss})\ \mathit{ts}) \quad = \quad \{r\}\ \cup\ \mathit{flattenMap}\ (\mathit{select}\ \mathit{least})\ \mathit{ts}.
$$

The definition of *add* can be inferred in a similar fashion.

$$
\begin{array}{ll}
 & \mathit{tab}\ (o : \mathit{os})\ \mathit{least}\ (\mathit{Node}\ (r, \mathit{vss})\ \mathit{ts}) \\
= & \quad \{\ \text{eq. (5)}\ \} \\
 & \mathit{fs}'\ (o : \mathit{os})\ \mathit{vss}\ \mathit{least}\ r\ \cup\ \mathit{flattenMap}\ (\mathit{tab}\ (o : \mathit{os})\ \mathit{least})\ \mathit{ts} \\
= & \quad \{\ \text{def. of } \mathit{fs}',\ \mathit{if} \text{ property}\ \} \\
 & \text{if } \#((o \in) \triangleleft \mathit{vss}) \geq \mathit{least} \\
 & \quad \text{then } \mathit{fs}'\ \mathit{os}\ \mathit{vss}\ \mathit{least}\ r\ \cup\ \mathit{fs}'\ \mathit{os}\ ((o \in) \triangleleft \mathit{vss})\ \mathit{least}\ (r \mathbin{+\!\!+} [o])\ \cup \\
 & \qquad\quad \mathit{flattenMap}\ (\mathit{tab}\ (o : \mathit{os})\ \mathit{least})\ \mathit{ts} \\
 & \quad \text{else } \mathit{fs}'\ \mathit{os}\ \mathit{vss}\ \mathit{least}\ r\ \cup\ \mathit{flattenMap}\ (\mathit{tab}\ (o : \mathit{os})\ \mathit{least})\ \mathit{ts} \\
= & \quad \{\ \text{eqs. (8) and (9)}\ \} \\
 & \text{if } \#((o \in) \triangleleft \mathit{vss}) \geq \mathit{least} \\
 & \quad \text{then } \mathit{fs}'\ \mathit{os}\ \mathit{vss}\ \mathit{least}\ r\ \cup\ \mathit{tab}\ \mathit{os}\ \mathit{least}\ (\mathit{Node}\ (r \mathbin{+\!\!+} [o], (o \in) \triangleleft \mathit{vss})\ [\,])\ \cup \\
 & \qquad\quad \mathit{flattenMap}\ (\mathit{tab}\ (o : \mathit{os})\ \mathit{least})\ \mathit{ts} \\
 & \quad \text{else } \mathit{fs}'\ \mathit{os}\ \mathit{vss}\ \mathit{least}\ r\ \cup\ \mathit{flattenMap}\ (\mathit{tab}\ \mathit{os}\ \mathit{least})\ \mathit{ts} \\
= & \quad \{\ \text{eqs. (11) and (6)}\ \} \\
 & \text{if } \#((o \in) \triangleleft \mathit{vss}) \geq \mathit{least} \\
 & \quad \text{then } \mathit{fs}'\ \mathit{os}\ \mathit{vss}\ \mathit{least}\ r\ \cup\ \mathit{tab}\ \mathit{os}\ \mathit{least}\ (\mathit{Node}\ (r \mathbin{+\!\!+} [o], (o \in) \triangleleft \mathit{vss})\ [\,])\ \cup \\
 & \qquad\quad \mathit{flattenMap}\ (\mathit{tab}\ \mathit{os}\ \mathit{least})\ ((\mathit{add}\ o\ \mathit{least}) * \mathit{ts}) \\
 & \quad \text{else } \mathit{fs}'\ \mathit{os}\ \mathit{vss}\ \mathit{least}\ r\ \cup\ \mathit{flattenMap}\ (\mathit{tab}\ \mathit{os}\ \mathit{least})\ \mathit{ts} \\
= & \quad \{\ \text{eq. (7)}\ \} \\
 & \text{if } \#((o \in) \triangleleft \mathit{vss}) \geq \mathit{least} \\
 & \quad \text{then } \mathit{fs}'\ \mathit{os}\ \mathit{vss}\ \mathit{least}\ r\ \cup \\
 & \qquad\quad \mathit{flattenMap}\ (\mathit{tab}\ \mathit{os}\ \mathit{least})\ ((\mathit{add}\ o\ \mathit{least}) * \mathit{ts} \mathbin{+\!\!+} [\mathit{Node}\ (r \mathbin{+\!\!+} [o], (o \in) \triangleleft \mathit{vss})\ [\,]]) \\
 & \quad \text{else } \mathit{fs}'\ \mathit{os}\ \mathit{vss}\ \mathit{least}\ r\ \cup\ \mathit{flattenMap}\ (\mathit{tab}\ \mathit{os}\ \mathit{least})\ \mathit{ts} \\
= & \quad \{\ \text{eq. (5), keeping invariants}\ \} \\
 & \text{if } \#((o \in) \triangleleft \mathit{vss}) \geq \mathit{least} \\
 & \quad \text{then } \mathit{tab}\ \mathit{os}\ \mathit{least}\ (\mathit{Node}\ (r, \mathit{vss})\ ((\mathit{add}\ o\ \mathit{least}) * \mathit{ts} \mathbin{+\!\!+} [\mathit{Node}\ (r \mathbin{+\!\!+} [o], (o \in) \triangleleft \mathit{vss})\ [\,]]) \\
 & \quad \text{else } \mathit{tab}\ \mathit{os}\ \mathit{least}\ (\mathit{Node}\ (r, \mathit{vss})\ \mathit{ts}) \\
= & \quad \{\ \mathit{if} \text{ property}\ \} \\
 & \mathit{tab}\ \mathit{os}\ \mathit{least} \\
 & \quad (\text{if } \#((o \in) \triangleleft \mathit{vss}) \geq \mathit{least} \\
 & \quad\ \ \text{then } \mathit{Node}\ (r, \mathit{vss})\ ((\mathit{add}\ o\ \mathit{least}) * \mathit{ts} \mathbin{+\!\!+} [\mathit{Node}\ (r \mathbin{+\!\!+} [o], (o \in) \triangleleft \mathit{vss})\ [\,]]) \\
 & \quad\ \ \text{else } \mathit{Node}\ (r, \mathit{vss})\ \mathit{ts})
\end{array}
$$

Therefore, we obtain:

$$
\begin{array}{l}
\mathit{add}\ o\ \mathit{least}\ (\mathit{Node}\ (r, \mathit{vss})\ \mathit{ts}) = \\
\quad \text{if } \#((o \in) \triangleleft \mathit{vss}) \geq \mathit{least} \\
\quad \text{then } \mathit{Node}\ (r, \mathit{vss})\ (\underline{(\mathit{add}\ o\ \mathit{least}) * \mathit{ts}} \mathbin{+\!\!+} [\mathit{Node}\ (r \mathbin{+\!\!+} [o], (o \in) \triangleleft \mathit{vss})\ [\,]]) \\
\quad \text{else } \mathit{Node}\ (r, \mathit{vss})\ \mathit{ts}.
\end{array}
$$

Comparing the two programs before and after tabulation calculation, we can see that the latter is more efficient in that it shares the computation for checking the invariant $\mathcal{I}_1$ conditions; when an object o is added to the tree, it checks from the root and if it fails at a node, it does not check its descendants due to the invariant $\mathcal{I}_2$.

Further improvement can be made on the underlined part above by making use of the invariant $\mathcal{I}_3$. Let t_i and t_j be two sibling trees where t_i is on the left of t_j (i.e., $i < j$), let $I : (r_i, vss_i)$ and $J : (r_j, vss_j)$ be the root nodes of t_i and t_j respectively, and let $C_i : (r_{c_i}, vss_{c_i})$ be I's child. With the assumption of $r_j \subset r_{c_i}$, the invariant $\mathcal{I}_3$ implies that if an object (item) o can be added to C_i (to make a larger frequent set), it must be able to be added to J. In other words, if o cannot be added to J, it cannot be added to C_i. Therefore, rather than applying *add o least* to each sibling tree independently, we would be better to do it from right to left. We thus introduce a new function:

$$add'\ o\ least\ ts = (add\ o\ least) * ts \tag{12}$$

and we take account of the invariant $\mathcal{I}_3$ to derive an efficient definition for *add'* by the following calculation.

$$\begin{array}{ll}
& add'\ o\ least\ ((Node\ (r, vss)\ ts) : ts') \\
= & \quad \{\ \text{eq. (12)}\ \} \\
& (add\ o\ least) * ((Node\ (r, vss)\ ts) : ts') \\
= & \quad \{\ *\ \} \\
& add\ o\ least\ (Node\ (r, vss)\ ts) : (add\ o\ least) * ts' \\
= & \quad \{\ \text{eq. (12)}\ \} \\
& add\ o\ least\ (Node\ (r, vss)\ ts) : add'\ o\ least\ ts' \\
= & \quad \{\ \text{def. of } add\ \} \\
& \text{if } \#((o \in) \triangleleft vss) \geq least \\
& \text{then } Node\ (r, vss)\ ((add\ o\ least) * ts +\!\!+ [Node\ (r +\!\!+ [o], (o \in) \triangleleft vss)\ [\,]]) : add'\ o\ least\ ts' \\
& \text{else } Node\ (r, vss)\ ts : add'\ o\ least\ ts' \\
= & \quad \{\ \text{inv. } \mathcal{I}_3\ \} \\
& \text{if } \#((o \in) \triangleleft vss) \geq least \\
& \quad \text{then } Node\ (r, vss)\ (pAdd * ts +\!\!+ [Node\ (r +\!\!+ [o], (o \in) \triangleleft vss)\ [\,]]) : add'\ o\ least\ ts' \\
& \quad \text{else } Node\ (r, vss)\ ts : add'\ o\ least\ ts' \\
& \text{where } pAdd\ tree@(Node\ (r, vss)\ ts) \\
& \qquad\qquad\quad \mid or\ [r' \subseteq r \mid (r', vss') \leftarrow root * ts',\ \#((o \in) \triangleleft vss') < least]\ =\ tree \\
& \qquad\qquad\quad \mid \text{otherwise}\ =\ add\ o\ least\ tree \\
= & \quad \{\ \text{inv. } \mathcal{I}_1 \text{ implies } r' \subseteq r \equiv last\ r' = last\ r\ \ \} \\
& \text{if } \#((o \in) \triangleleft vss) \geq least \\
& \quad \text{then } Node\ (r, vss)\ (pAdd * ts +\!\!+ [Node\ (r +\!\!+ [o], (o \in) \triangleleft vss)\ [\,]]) : add'\ o\ least\ ts' \\
& \quad \text{else } Node\ (r, vss)\ ts : add'\ o\ least\ ts' \\
& \text{where } pAdd\ tree@(Node\ (r, vss)\ ts) \\
& \qquad\qquad\quad \mid or\ [last\ r' = last\ r \mid (r', vss') \leftarrow root * ts',\ \#((o \in) \triangleleft vss') < least]\ =\ tree \\
& \qquad\qquad\quad \mid \text{otherwise}\ =\ add\ o\ least\ tree
\end{array}$$

Here *root* returns the root of a tree and *last* returns the last element of a list. Notice the important role of lazy evaluation for computing the above expression, where any subexpression will not be computed until its result is necessary.

So much for the derivation. Now putting all together, we get the final algorithm in Figure 1. We leave an informal explanation of the algorithm later in Section 4.1.

4 Features of the New Algorithm

We shall clarify several features of the derived algorithm, namely correctness, simplicity, efficiency and inherited parallelism, and highlight how to adapt the algorithm to practical use.

4.1 Correctness

The correctness of the derived algorithm follows directly from the basic property of program calculation. Our derived algorithm is *correct* with respect to the initial straightforward specification,

$$
\begin{array}{lcl}
fs\ os\ vss\ least & = & fs'\ os\ vss\ least\ [\,] \\
fs'\ os\ vss\ least\ r & = & tab\ os\ least\ (Node\ (r, vss)\ [\,]) \\
tab\ [\,]\ least\ t & = & select\ least\ t \\
tab\ (o : os)\ least\ t & = & tab\ os\ least\ (add\ o\ least\ t)
\end{array}
$$

where

$select\ least\ (Node\ (r, vss)\ ts) = \{r\} \ \cup\ flattenMap\ (select\ least)\ ts$

$add\ o\ least\ t = add'\ o\ least\ [t]$
$add'\ o\ least\ ((Node\ (r, vss)\ ts) : ts') =$
  if $\#((o \in) \triangleleft vss) \geq least$
    then $Node\ (r, vss)\ (pAdd * ts +\!\!+ [Node\ (r +\!\!+ [o], (o \in) \triangleleft vss)\ [\,]]) : add'\ o\ least\ ts'$
    else $Node\ (r, vss)\ ts : add'\ o\ least\ ts'$
  where $pAdd\ tree@(Node\ (r, vss)\ ts)$
    $\mid or\ [last\ r' = last\ r \mid (r', vss') \leftarrow root * ts',\ \#((o \in) \triangleleft vss') < least] \ = \ tree$
    $\mid$ otherwise $=$ $add\ o\ least\ tree$

Figure 1: Our Algorithm for Finding Frequent Sets

because the whole derivation is done in a semantics-preserving manner. In contrast, the correctness of existing algorithms, well summarized in [Mue95, Toi96, Zak99], are often proved in an ad-hoc manner.

Figure 1 summarizes the algorithm formally described in Haskell. It is of high level, and efficient implementation of the algorithm requires more refinement particularly on the design of suitable data structures (Section 4.5). As seen in Figure 1, the most costly but important computation in our algorithm is $(o \in) \triangleleft vss$, so to implement this computation efficiently, we may impose restriction that the database is organized item by item rather than transaction by transaction.

To help readers to see more clearly the new features of our algorithm in this section, we briefly explain the derived algorithm in an informal way, using the example in Section 2. For efficient implementation of $(o \in) \triangleleft vss$, we assume that the input to our algorithm may is a vertical representation of the customers' transactions (visits):

$$
\begin{array}{ll}
\text{item } 1: & \{1,2,4\} \\
\text{item } 2: & \{1,2,3,4\} \\
\text{item } 3: & \{1\} \\
\text{item } 4: & \{1\} \\
\text{item } 5: & \{2,5\} \\
\text{item } 6: & \{2\} \\
\text{item } 7: & \{1,5\} \\
\text{item } 8: & \{4\} \\
\text{item } 9: & \{3\} \\
\text{item } 10: & \{\,\} \\
\text{item } 11: & \{\,\}
\end{array}
$$

which will be scanned item by item. Here, associated with each item is a set of visits (visit numbers) during which customers purchased the item.

The tabulation tree is basically for saving intermediate resulting frequent sets. It starts from a tree with a single node tagged with $(\{\,\}, \{1,2,3,4,5,6,7,8,9,10,11\})$. Each node in the tree is tagged with a pair, whose first component is a frequent set and whose second component is a set of visits where the frequent set appears.

The tree is then updated sequentially by attempting to add item 1 till 11 to the tree *from the root*, corresponding to the *add* function in Figure 1. This process is shown in Figure 2.

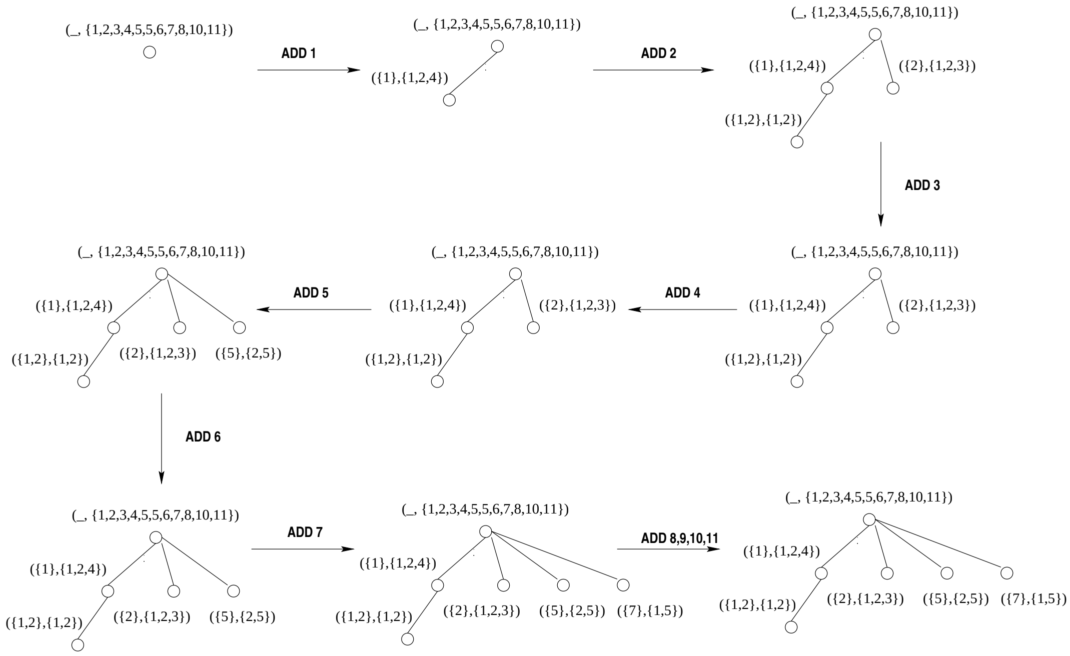

Figure 2: Sequence of the Tree Updating for Adding Item 1 to 11

Item 1 is tested for addition from the root of the tree, and the set intersection operation on $\{1,2,3,4,5,6,7,8,9,10,11\}$ and $\{1,2,4\}$ gives $\{1,2,4\}$, indicating that it can be placed as a child of the root. Item 2 is next added as the *rightmost* child of the root as item 1 does, and it is further tested for being added to the *left* sibling trees and is finally placed as a child of the node $(\{1\},\{1,2,4\})$. Note that scanning sibling trees leftwards is important in reducing searching space in our algorithm. Item 3 is not frequent, so adding operation stops leading to no change of the tree. Same thing happens for item 4. Item 5 is placed as the rightmost child of the root, but it fails to be further added to the left sibling trees. Other items are added similarly. The important point of this algorithm is that if an item cannot be added to a node, say N, (to form a larger frequent set) then it will neither be added to any child of N, as in the case of adding item 3, nor will it be added to any of N's left sibling nodes' children with larger frequent set than that of N, as in the case of adding item 5 where from the fact that it cannot be added to the node $(\{2\},\{1,2,3\})$ we know that it cannot be added to $(\{1,2\},\{1,2\})$, a child of its left sibling node $(\{1\},\{1,2,3\})$.

After a single scan of all items, our frequent sets appear as the first components at the tree nodes: $\{1\}$, $\{2\}$, $\{5\}$, $\{7\}$ and $\{1,2\}$.

4.2 Simplicity

Our derived algorithm is surprisingly *simple*, compared to many existing algorithms which pass over the database many times and use complicated and costly manipulation (generating and pruning) of candidates of frequent sets [AIS93, AS94, Mue95, Toi96, BMUT97]. A major difference is that our algorithm traverses the database vertically (i.e., item by item) with a single pass while most of the traditional algorithms like the Apriori Algorithm traverse the database horizontally (i.e., transaction by transaction) with multiple passes.

It is interesting to see that newer data mining algorithms by Zaki [Zaki99] and his colleagues [ZPOL97,ZPOL97b] have largely adopted a similar idea of traversing database vertically. These algorithms use a vertical database format to efficiently determine the support of any k itemset by simply intersecting transaction-id-lists of the lexicographically first two $(k-1)$-length subsets that share common prefix to reduce searching space. However, their approach still needs more than

Table 1: Experiment on Three Algorithms

	total time (secs)	memory cells (mega bytes)
Our Initial Specification	131.2	484.1
An Apriori Algorithm	10.88	72.0
Our Final Algorithm	0.44	2.5

three database scans [Zak99]. Comparatively, our algorithm adopts a simpler strategy to reduce searching space by organizing the intermediate frequent sets in a tree with specific constraints on sibling nodes. As a nice consequence, our algorithm requires just a single data base scan.

In fact, the vertical traversal of database comes naturally from our final algorithm derived from the initial straightforward specification where we were not at all concerned with efficiency and implementation details. If the database is organized transaction by transaction, we can preprocess the database to fit our algorithm by transposing it through a single pass. This preprocessing can be done in an efficient way even for a huge database saved in external storage (see [Knu97]). As such preprocessing need only be done once for a given transaction database, we can easily amortize its costs over many data mining runs for the discovery of interesting information/rules.

4.3 Efficiency

To see how efficient our algorithm is in practice, we shall not give a formal study of the cost. Such a study needs to take account of both the distribution as well as the size of data sample. Rather we use a *simple* experiment to compare our algorithm with an existing improved Apriori algorithm [MT96], one of the best algorithms used in the data mining community.

We start by considering the case of a small database which can be put in the memory. We tested three algorithms in Haskell: our initial specification program, the functional coding of the existing improved Apriori algorithm in [HLG+99], and our final derived program. Our initial and final algorithms can be directly coded in Haskell by representing sets using lists.

The input sample data was extracted from the Richard Forsyth's zoological database, which is available in the UCI Repository of Machine Learning Databases [BM98]. It contains 17 objects (corresponding to 17 boolean attributes in the database) and 101 transactions (corresponding to 101 instances). We set the threshold to be 20 (20% of frequency)[2] , and did experiment with Glasgow Haskell Compiler and its profiling mechanism. The experimental result is given in Table 1. It shows that our final algorithm has been dramatically improved as compared to our initial specification, and that it is also much more efficient than the functional coding of an existing algorithm (about 20 times faster but using just 1/30 of memory cells). Among other reasons, calculating frequent sets by computing with set intersection rather than counting through database traversal seems to have played an important role here (As it is, the derived algorithm assumes that we have sufficient memory for the tabulation tree.)

More experiments and discussion on the comparison between our algorithm and the above improved Apriori algorithm in functional community can be found in [HLG+99].

What if the database is so huge that only part of database can be read into memory at one time? Except for the preprocessing of the database to match our algorithm (to be done just once as discussed above), our algorithm can deal with the partitioning of database with an idea that is similarly used in [ZPOL97b, Zak99]. If the database has N objects (items), our algorithm allows it to be partitioned into N smaller sections. We only require that each of these sections be read into memory, one at a time, which poses no problem practically.

[2] We could set a smaller threshold and our algorithm showed even better timing.

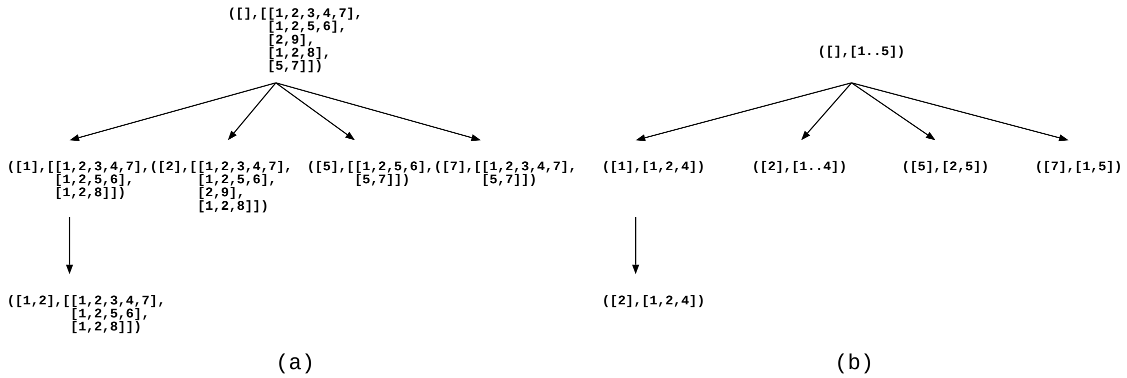

Figure 3: Tree Structure for Tabulation

4.4 Parallelism

As shown in [Zak99], many sequential data mining algorithms have their corresponding parallel versions. Similarly, our algorithm can be easily parallelized, as briefly explained below.

Suppose that we have objects from 1 to N, and M processors of $P_1, \ldots, P_M$. We can decompose the objects into M groups, say 1 to N/M, $N/M + 1$ to $2N/M$, $\ldots$, and use P_i to compute the tabulation tree for items from $(i-1)N/M + 1$ to iN/M. Certainty all the processors can do this in parallel. After that, we may propagate information of the single-item frequent sets from processor P_{i+1} to P_i for all i, to see whether these single-item frequent sets in P_{i+1} could be merged with frequent sets computed in P_i.

Note that this parallel algorithm could be *formally* obtained from the sequential program *tab* in Figure 1 by parallelization calculation [HTC98], which is omitted here.

4.5 Implementation

The derived algorithm can be used practically to win over the existing algorithms, because of the single traversal of database and much less use of the costly operation for checking of subset relationship. To be able to compare our results more convincingly with those in data mining field, we are mapping the algorithm to a C program and testing it on the popular benchmark of sample database. The detailed results will be summarized in another paper. Here, we only highlight one practical consideration.

A crucial aspect in practical implementation of the derived algorithm is the design of an efficient data structure to represent the tabulation tree to keep memory usage down. In fact, we can refine the current structure of tabulation tree to use less space. Notice that each node of the tabulation tree is attached with a pair (r, vss) where r represents a frequent set, and vss represents all the visits that contain r. Naive implementation would take much space. To be concrete, consider the example given in the beginning of Section 2. After traversing all objects from 1 to 11, we get the tree (a) in Figure 3. The vss part in each node consumes much space. In fact, it is not necessary to store the detailed visit content in each node. Instead, it is sufficient to store a list of indices to the visits, as shown in tree (b) in Figure 3. Practically, the number of indices in each node is not so big except for the root where we use the range notation to represent it cheaply, and this would become smaller with each step down from parent to its children.

There are several other ways to reduce the size of the tabulation tree. (1) In preprocessing phase, we may sort the objects of the database by decreasing frequency, which should allow subrange notation of indices to be used in a maximized fashion. (2) If the size of vss is within twice of the threshold *least* at a particular node, we may keep negative information at the children nodes, as these lists would be shorter than the threshold. (3) As nodes for 1-itemset take the most memory and these should be kept off-line in virtual memory and be paged in when required.

5 Conclusion

In this paper, we have addressed a practical application of program calculation of functional programs by formally deriving a new and efficient algorithm. We have chosen an important subproblem of finding frequent sets as our target. This problem is of practical interest, and have been extensively researched by the data mining community in the last six years. Many researchers have devoted much time and energy to discover clever and fast algorithms. By program calculation of functional programs, we have successfully obtained a new algorithm that is also practically fast.

Our derivation of a new frequent set algorithm did not depend on new tricks. Instead, it is carried out using a sequence of standard calculation techniques (for optimization) such as fusion, accumulation, filter promotion and tabulation. These calculation techniques are quite well-known in the functional programming community.

This work is a continuation of our effort to apply calculational transformation techniques [THT98] to the development of efficient programs [OHIT97, HITT97, HTC98]. Our previous work put emphasis on mechanical implementation of the transformation techniques, while this paper shows that this calculation strategy is also very helpful for guiding programmers/researchers in the development of new algorithms. Our derived frequent set algorithm compares very favorably with a state-of-the-art algorithm used in practice by the data mining community.

One interesting future work, we believe, is to study how to efficiently implement the derived algorithm in a practical language, say C. This is very important, not only for verifying the algorithm on practical databases, but also for making the algorithm be really useful. In fact, much engineering work should be necessary to bridge the gap between algorithm and implementation.

Acknowledgments

This paper owes much to the thoughtful and inspiring discussions with David Skillicorn, who argued that program calculation should be useful in derivation of data mining algorithms. He kindly explained to the first author the problem as well as some existing algorithms. We would also like to thank Christoph Armin Herrmann who gave us his functional coding of an existing (improved) Apriori algorithm, and help test our Haskell code with his HDC system.

References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *1993 International Conference on Management of Data (SIGMOD'93)*, pages 207–216, May 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *International Conference on Very Large Data Base (VLDB)*, pages 487–499, Santiago, Chile, 1994.

[BdM96] R.S. Bird and O. de Moor. *Algebras of Programming.* Prentice Hall, 1996.

[Bir80] R. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.

[Bir84] R. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir89] R. Bird. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

[Bir98] R.S. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[BM98] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[BMUT97] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *1997 International Conference on Management of Data (SIGMOD'97)*, pages 255–264, AZ, USA, 1997. ACM Press.

[CH95] W.N. Chin and M. Hagiya. A transformation method for dynamic-sized tabulation. *Acta Informatica*, 32:93–115, 1995.

[Chi90] W.N. Chin. *Automatic Methods for Program Transformation*. Phd thesis, Department of Computing, Imperial College of Science, Technology and Medicone, University of London, May 1990.

[Chi92] W.N. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

[GLJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[HIT99] Z. Hu, H. Iwasaki, and M. Takeichi. Caculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

[HLG+99] C. Herrmann, C. Lengauer, R. Gunz, J. Laitenberger, and C. Schaller. A compiler for HDC. Technical Report MIP-9907, Fakultat fur Mathematik und Informatik, Universitat Passau, May 1999.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation*. Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[Knu97] D. Knuth. *The Art of Computer Programming: Volume 3 / Sorting and Searching*. Addison-Wesley, Longman, 1997. Second Edition.

[LK98] D. Lin and Z. Kedem. Princer Search: A new algorithm for discovering the maximum frequent set. In *VI Intl. Conference on Extending Database Technology*, Valencia, Spain, March 1998.

[MT96] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In *2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 189 – 194, Portland, Oregon, August 1996. AAAI Press.

[Mue95] A Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report Technical Report CS-TR-3515, University of Maryland, College Park, MD, 1995.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106, Le Bischenberg, France, February 1997. Chapman&Hall.

[THT98] A. Takano, Z. Hu, and M. Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 30(3), December 1998. Special issues for 1998 Symposium on Partial Evaluation.

[Toi96] H. Toivonen. *Discovery of Frequent Patterns in Large Data Collections.* Ph.D thesis, Department of Computer Science, University of Helsinki, 1996.

[Zak99] M.J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, December 1999.

[ZPOL97a] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd International Conference on Knowledge Discovery and Data Mining*, pages 283–296, Menlo Park, Califoria, 1997. AAAI Press.

[ZPOL97b] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 1(4):343–373, December 1997.