World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:1, No:1, 2007

Mining Frequent Patterns with Functional Programming

Nittaya Kerdprasop, and Kittisak Kerdprasop

***Abstract*—**Frequent patterns are patterns such as sets of features or items that appear in data frequently. Finding such frequent patterns has become an important data mining task because it reveals associations, correlations, and many other interesting relationships hidden in a dataset. Most of the proposed frequent pattern mining algorithms have been implemented with imperative programming languages such as C, C++, Java. The imperative paradigm is significantly inefficient when itemset is large and the frequent pattern is long. We suggest a high-level declarative style of programming using a functional language. Our supposition is that the problem of frequent pattern discovery can be efficiently and concisely implemented via a functional paradigm since pattern matching is a fundamental feature supported by most functional languages. Our frequent pattern mining implementation using the Haskell language confirms our hypothesis about conciseness of the program. The performance studies on speed and memory usage support our intuition on efficiency of functional language.

***Keywords*—**Association, frequent pattern mining, functional programming, pattern matching.

I. Introduction

FREQUENT pattern mining is the discovery of relationships or correlations between items in a dataset. A set of market basket transactions [1], [2] is a common dataset used in frequent pattern analysis. A dataset is typically in a table format. Each row is a transaction, identified by a transaction identifier or a *TID*. A transaction contains a set of items bought by a customer. A set of transactions might be organized in either an enumerated (dense), or a sparse binary vector format [3], [7]. In either format a dataset can be processed horizontally or vertically. Fig. 1 illustrates the data organization formats of a simple market basket dataset.

In a horizontally enumerated data organization (fig. 1a), each transaction contains only items positively associated with a customer purchase. It is a simplistic representation of market basket data because it ignores other information such as the quantity of purchased items or the profit of item sold. A horizontally enumerated format is sometimes referred to as a *TidLists* dataset organization. In a vertical organization of items bought enumeration (Fig. 1b), each column stores an ordered list of *TIDs* of the transactions that contain an item. This format of a dataset occupies that same space as the horizontally enumerated format.

Figs. 1c and 1d represent a binary vector format. A value in each vector cell is 1 if the item is present in a transaction and 0 otherwise. A binary vector format is referred to as a *TidSets* dataset organization.

TI	**Items**
D1	{Cereal, Milk}
2	{Beer,Cereal,Diaper,Egg}
3	{Beer, Diaper, Milk}
4	{Beer,Cereal,Diaper,Milk}
5	{Diaper, Milk}

(a) Horizontally enumerated format

TID	**Item IDs**				
	B	C	D	E	M
2	1	2	2	1	
3	2	3		3	
4	4	4		4	
		5		5	

(b) Vertically enumerated format

TID	**Items IDs**				
	B	C	D	E	M
1	0	1	0	0	1
2	1	1	1	1	0
3	1	0	1	0	1
4	1	1	1	0	1
5	0	0	1	0	1

(c) Horizontal binary vector

TID	**Items IDs**				
	B	C	D	E	M
1	0	1	0	0	1
2	1	1	1	1	0
3	1	0	1	0	1
4	1	1	1	0	1
5	0	0	1	0	1

(d) Vertical binary vector

Fig. 1 Organization of a market basket dataset

Recent attention has been given to the influence of data organization on the performance of the process of frequent pattern discovery. Shenoy et al.[7] described the advantages of the vertical organization over the horizontal organization. They also introduced the VIPER algorithm that uses a combination of horizontal and vertical formats to reduce the space. Zaki and Gouda [10] presented a vertical data representation called *Diffset* that only keeps track of the differences in the *TidLists* of a candidate pattern from its generating frequent patterns. A vertical vector organization has been proven an efficient layout for the problem of frequent pattern discovery, but it suffers from the memory usage. We thus propose to switch the paradigm towards the algorithm implementation from conventional imperative to a declarative style of lazy functional programming. Our performance studies have confirmed the improvement on speed and memory usage.

Manuscript received November 29, 2006. This work was supported in part by the research fund from Suranaree University of Technology and the grant from the National Research Council of Thailand.

Nittaya Kerdprasop is with the School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (phone: +66-44-224432; fax: +66-44-224602; e-mail: nittaya@ sut.ac.th, nittaya.k@gmail.com).

Kittisak Kerdprasop is with the School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (e-mail: kerdpras@sut.ac.th).

II. Search Space of Frequent Pattern Mining

In frequent pattern mining, we are interested in analyzing connections among items. A collection of zero or more items is called an itemset. For example, the first transaction in Fig. 1 contains the itemset {Cereal, Milk}. Since this set contains two items, it is called a 2-itemset. An itemset can be an empty set, a 1-itemset, a 2-itemset, and so on. Fig. 2 shows all combinations of distinct itemsets from the set of items {B, C, D, E, M}, where B = Beer, C = Cereal, D = Diaper, E = Egg, and M = Milk.

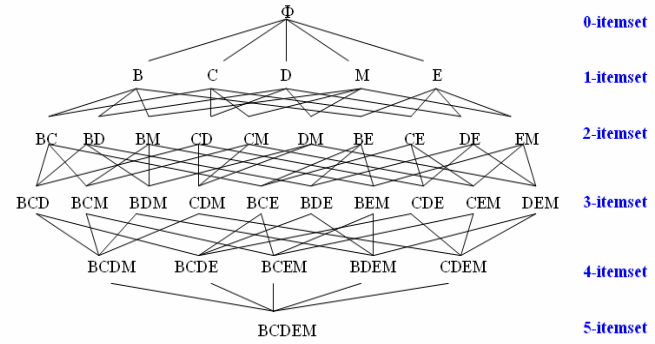


Fig. 2 A lattice of distinct itemsets

The discovery of interesting relationships hidden in large datasets is the objective of frequent pattern mining. The uncovered relationships can be represented in the form of association rules. An association rule is an inference of the form $X \rightarrow Y$, where X and Y are non-empty disjoint itemsets. To form association rules, we consider only valid itemsets. An itemset is valid if it really occurs in a transaction. For instance, from a dataset shown in Fig. 1 an itemset {Egg, Milk} is invalid because none of the customers buy both eggs and milk.

The identification of all valid itemsets is computational expensive. It can be seen from Fig. 2 that a dataset of I items has 2^I distinct itemsets. To reduce the search space, the measurements of *support* and *confidence* are used to constrain the mining process. The constraint *support* forces the mining process to discover only relationships that occur frequently, while *confidence* constrains the reliability of the inference made by a rule. The support count for an itemset Z, denoted as $\sigma(Z)$, is the number of transactions that contain a particular itemset Z. As an example, consider a dataset in Fig. 1, there are three transactions (*TID* 2, 3, 4) contain the item Beer, thus σ(Beer) = 3. Given the definition of support count, the metrics support and confidence of the association rule $X \rightarrow Y$ can be defined as follows [4], [8].

$$\textit{Support}, \; s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N},$$

where N is the number of all transactions.

$$\textit{Confidence}, \; c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}.$$

Given a dataset as shown in Fig. 1, an example of association rule is the statement that "customers who buy beer also buy diaper, with 60% supporting transactions and 100% confidence." An itemset is called a *frequent itemset* if its support is greater than or equal user-specified support threshold (called *minSup*). An association rule generated from frequent itemset with the confidence greater than or equal a confidence threshold (called *minConf*) is considered a valid association rule. With the pre-specified *minSup* and *minConf* metrics, the problem of association rule discovery can be stated as follows: Given a set of transactions, find all the rules having support ≥ *minSup* and confidence ≥ *minConf*. This problem can be decomposed into two subtasks:

(1) Frequent itemset generation: find all itemsets that satisfy the *minSup* threshold.

(2) Rule generation: generate from frequent itemsets all high confidence rules.

It is the *minSup* constraint that helps reducing the computational complexity of frequent itemset generation. Suppose we specify *minSup* = 2/5 = 40% on a set of transactions shown in Fig. 1; the item {Egg} is infrequent. As a result, all supersets of {Egg} are also infrequent. All infrequent itemsets can then be pruned to reduce the search space (see Fig. 3).

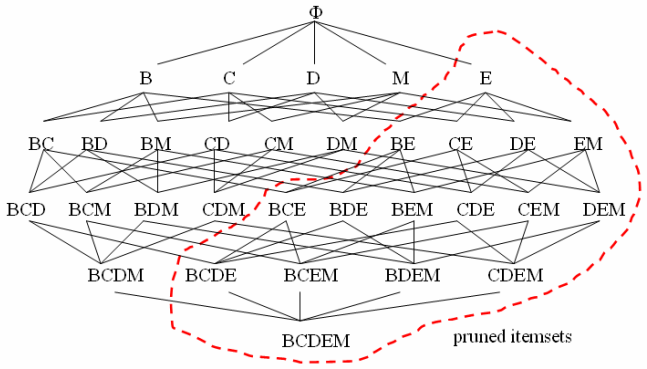


Fig. 3 A pruning of all itemsets that contain an infrequent item E

Frequent itemsets are actually patterns that appear in a dataset frequently. Finding frequent patterns has become an important data mining task. We propose that frequent patterns can be mined efficiently using a high-level programming language such as Haskell that provides a full support for pattern matching functionality.

III. Pattern Matching with Haskell

A problem in frequent pattern discovery is to determine how often a candidate pattern occurs. In association mining, a pattern is a set of items co-occurrence across a dataset. Given a candidate pattern, the task of pattern matching is to search for its frequency looking for the patterns that are frequent enough. The outcome of this search is frequent itemsets that

suggest strong co-occurrence relationships between items in the dataset.

The search for patterns of interest can be efficiently programmed using the Haskell language. Haskell has evolved as a strongly typed, lazy, pure functional language since 1987 [5], [6], [9]. The language is named after the mathematician Haskell B. Curry whose work on lambda calculus provides the basis for most functional languages. A program in functional languages is made up of a series of function definitions. The evaluation of a program is simply the evaluation of functions. Haskell is a pure functional language because functions in Haskell have no side effect, i.e. given the same arguments; the function always produces the same result. As an example, we can define a simple function to square an integer as follows:

```
square :: Int -> Int      -- type declaration
square x = x * x          -- function definition
```

The first line of the definition declares the type of the thing being defined; Haskell is a strongly typed language. This states that square is a function taking one integer argument (the first Int) and returning an integer value (the second Int). The arrow symbol denotes mapping from an argument to a result and the symbol "::" can be read "has type". The statement or phrase following the symbol "--" is a comment. The second line gives the definition of function square, i.e. given an integer x, the function returns the value of x*x. To apply the function, we provide the function an actual argument such as square 5 and the result 25 can be expected.

Pattern matching is one of the most powerful features of Haskell. Defining functions by specifying argument patterns is a common practice in programming with Haskell. As an illustration, consider the following example:

```
fib :: Int -> Int      -- a function takes one Int
                       -- and returns an Int

fib 0 = 0              -- pattern 1: argument is 0
fib 1 = 1              -- pattern 2: argument is 1
fib n = fib (n-2) + fib (n-1)
-- pattern 3: argument is Int other than 0 and 1
```

The function fib returns the n^{th} number in the Fibonacci sequence. The left hand sides of function definitions contain patterns such as 0, 1, n. When applying a function these patterns are matched against actual parameters. If the match succeeds, the right hand side is evaluated to produce a result. If it fails, the next definition is tried. If all matches fail, an error is returned.

Pattern matching is a language feature commonly used with a list data structure. For instance, [1, 2, 3] is a list containing three integers. It can also be written as 1:2:3:[],where [] represents an empty list and ":" is a list constructor. The following example defines length function to count the number of elements in a list.

```
length :: [Int] -> Int
        -- This function takes a list of Int as its
        -- argument and returns the number of
        -- elements in the list
length [ ] = 0
        -- pattern 1: length of an empty list is 0
length (x:xs) = 1 + length xs
        -- pattern 2: length of a list whose first
        -- element is called x and remainder is
        -- called xs is 1 plus the length of xs
```

The pattern [] is defined to match the case of an empty list argument. The pattern x:xs will successfully match a list with at least one element, i.e. xs can be a list of zero or more elements.

IV. Implementation

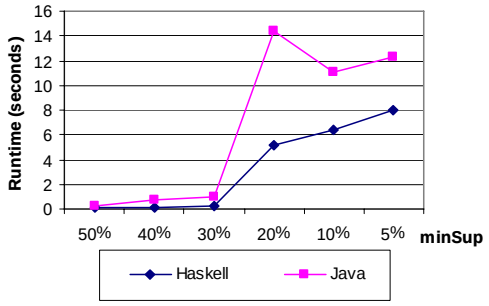
We implement Apriori algorithm [1], [2] using Haskell language as shown in Fig. 4. Each item is represented by the item identifier which is an integer. Thus, an itemset is denoted as a set of Int declared in the first line of our Haskell code. The function sumi is defined to count the number of occurrence of each itemset. Functions listC and listC' perform the task of enumerating candidate frequent itemsets. Only itemsets that satisfy the *minSup* threshold are reported from the functions listL and listL' as frequent itemsets. It can be seen that the discovery of frequent itemsets using Haskell functional language takes only 20 lines of code.

```
itemSet :: [Set Int]
itemSet =[Set.singleton x | x<-[1..9]]
sumi::Set Int->[Set Int]->Int
sumi s [] =0
sumi s (y:ys) |(Set.isSubsetOf s y)= 1+(sumi s ys)
              |otherwise = (sumi s ys)
listC ::Int->[(Set Int,Int)]
listC 1=[let n=(sumi s dataB) in (s,n)  |s<-itemSet]
listC n=[let n=(sumi s dataB) in (s,n) |s<-
                                  Set.toList(listC' n)]
listC' :: Int->Set(Set Int)
listC' 2=Set.fromList
    [(Set.union x y) |x<-(listL' 1),y<-(listL' 1),x/=y]
listC' n=Set.fromList
   [(Set.union x y) |x<-(listL' (n-1)), y<-(listL' (n-1)),
     x/=y, (Set.size(Set.union x y))==n]
listL ::Int->[(Set Int,Int)]
listL n=[(x,y)|(x,y)<-listC n, y>=minSup]
listL'::Int->[Set Int]
listL' n =[x|(x,_)<-listL n]
```

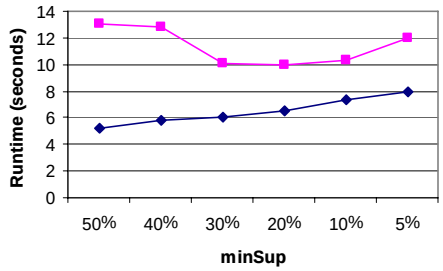
Fig. 4 Frequent itemsets discovery implemented with Haskell

V. Performance Studies

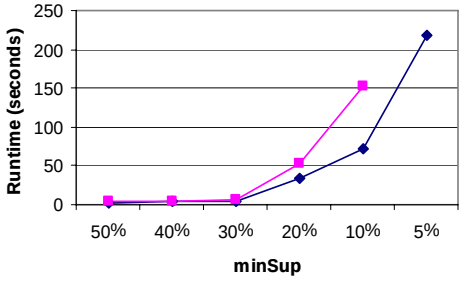
We comparatively study the performance of our implementations of frequent itemset discovery using Haskell versus Java. All experimentations have been performed on a 796 MHz AMD Athlon notebook with 512 MB RAM and 40 GB HD. We select four datasets downloaded from UC Irvine Machine Learning Database Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html) to test the speed of Haskell and Java programs.



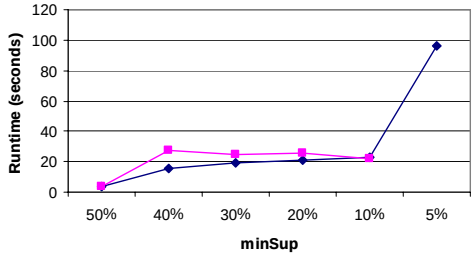
(a) Vote dataset



(b) Chess dataset



(c) DNA dataset



(d) Mushroom dataset

Fig. 5 Experimental results from two programming paradigms

TABLE I
Dataset Characteristics

Dataset	File size	# Transactions	[a]# Items
Vote	13.2 KB	300	17
Chess	237 KB	2,130	37
DNA	252 KB	2,000	61
Mushroom	916 KB	5,416	23

The details of selected datasets are summarized in table 1. The frequent itemset discovery pro-grams have been tested on each dataset with varied *minSup* values. Performance comparisons of Haskell and Java implementations on four datasets are graphically shown in Fig. 5.

It can be noticed from the experimental results that runtime increases as the minimum support (*minSup*) threshold gets lower. This is due to the fact that at a low level of *minSup* the number of frequent itemsets generated is significantly high. The implementation of frequent itemset discovery using Haskell outperforms that of Java on every dataset. A good performance can be clearly seen when *minSup* gets lower than 30%.

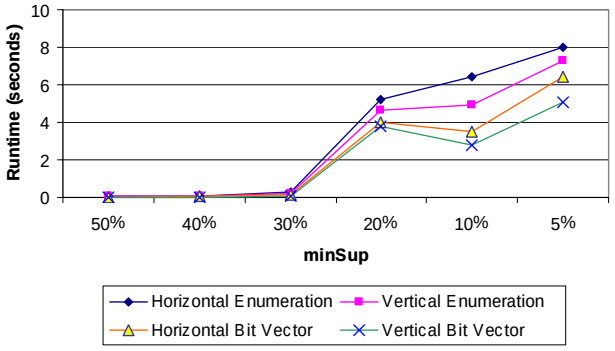
On large datasets with many items such as DNA and Mushroom, the program implemented with Java has a problem of insufficient memory and cannot run to completion at a 5% *minSup* threshold. This problem does not exist in the Haskell implementation.

The experimental results shown in Fig. 5 obtain from the datasets represented in a horizontally enumerated format. We also study the impact of different data formats on program running time and memory usage of a Haskell implementation. To observe the running time we implement the following code.

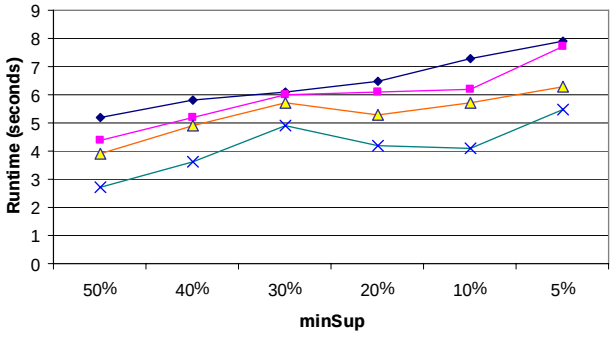
```
benchmark action = do
    prev <- getCPUTime
    action
    current <- getCPUTime
    let
      secs = fromIntegral (current-prev) / 1e12
    putStrln$ "Uses:  " ++ show secs
                ++ "  seconds "
```

The results of running time and memory usage using different styles of data representation are shown in Figs. 6 and 7, respectively. It can be noticed from the experimental results that on a speed comparison the vertical binary vector format is the fastest, the horizontal binary vector comes second following by the vertically enumerated organization. The horizontally enumerated format is the slowest one.

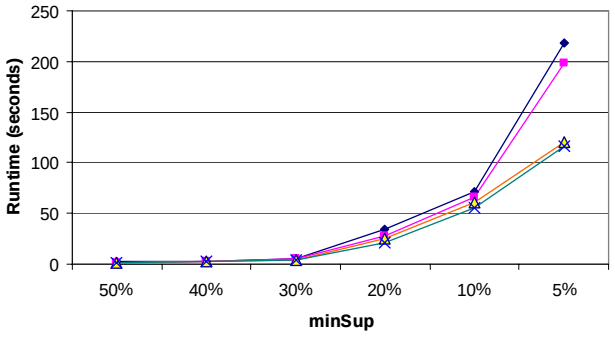
On the memory usage comparison the ordering is vice versa. Dataset represented with an enumeration format takes less storage area, while a binary vector format consumes more memory. The horizontal layout slightly outperforms the vertical layout in terms of memory usage during the process of finding frequent itemsets from the generated candidate sets.



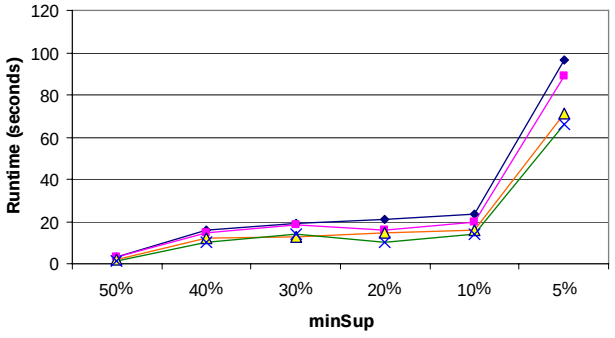
(a) Vote dataset



(b) Chess dataset

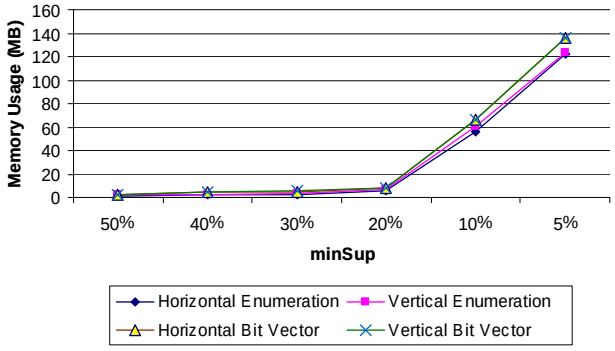


(c) DNA dataset

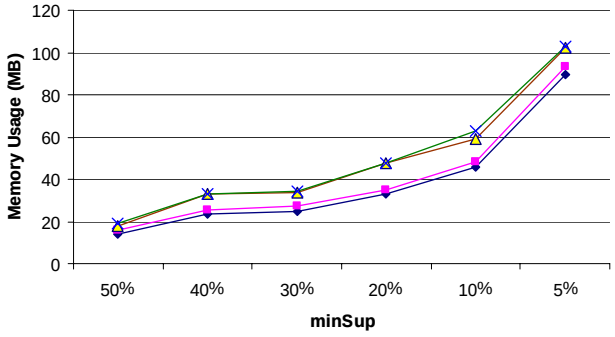


(d) Mushroom dataset

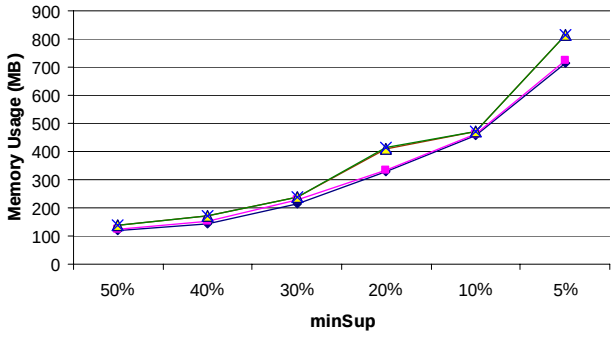
Fig. 6 The effect of data organization on speed



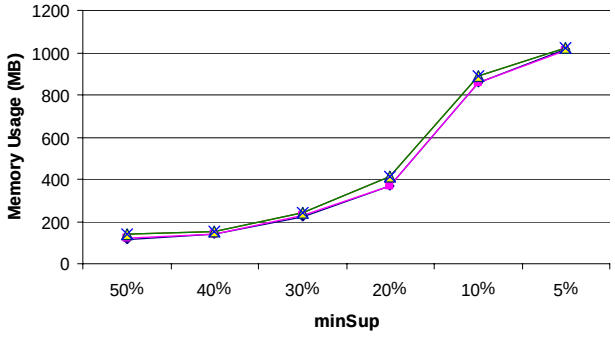
(a) Vote dataset



(b) Chess dataset



(c) DNA dataset



(d) Mushroom dataset

Fig. 7 The effect of data organization on memory usage

VI. Conclusion and Discussion

Association mining is one major problem in the area of data mining. The problem concerns finding frequent patterns hidden in a dataset. Frequent patterns are patterns such as set of items that appear in data frequently. Finding such frequent patterns has become an important data mining task because it reveals associations, correlations, and many other interesting relationships among items in the dataset.

The idea to mine association rules was first proposed in 1993 by R. Agrawal, T. Imielinski, and A. Swami and the well known Apriori algorithm was proposed by R. Agrawal and A. Swami in 1994. Since then many variations of Apriori have been proposed. Most algorithms are implemented with imperative programming languages such as C, C++, Java. We, on the other hand, suggest that the problem of frequent pattern discovery can be efficiently and concisely implemented with functional languages. Our supposition is that pattern matching is a fundamental feature supported by functional languages. The implementation of Apriori algorithm using Haskell confirms our hypothesis about conciseness of the program. The performance studies also support our intuition on efficiency because Haskell implementation outperforms the Java implementation in terms of speed and memory usage in every dataset.

This preliminary study supports our belief regarding functional programming paradigm towards frequent itemsets mining. We focus our future research on the design of data organization to optimize the speed and storage requirement. We also consider the extension of Apriori in the course of concurrency to improve its performance. With the power of Haskell, this is a very promising extension

Acknowledgment

This work was supported in part by grants from National Research Council of Thailand (NRCT) and the Thailand Research Fund (TRF). Kittisak Kerdprasop is a director of Data Engineering and Knowledge Discovery (DEKD) research unit in which Nittaya Kerdprasop is also a member and a researcher of this research unit. DEKD is fully supported by Suranaree University of Technology.

References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1993, pp. 207–216.

[2] R. Agrawal and R. Srikant, "Fast algorithm for mining association rules," in *Proc. Int. Conf. Very Large Data Bases*, 1994, pp. 487–499.

[3] A. Ceglar and J. Roddick, "Association mining," *ACM Computing Surveys*, vol. 38, no.2, 2006.

[4] J. Han and M. Kamber, *Data Mining: Concepts and Techniques* (2nd ed.), Morgan Kaufmann, 2006.

[5] P. Hudak, J. Fasel, and J. Peterson, "A gentle introduction to Haskell," Yale University, *Technical Report Yale U/DCS/RR-901*, 1996.

[6] P. Jones and J. Hughes (eds.), *Standard Libraries for the Haskell 98 Programming Languages*. Available: http://www.haskell.org/library/.

[7] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah, "Turbo-charging vertical mining of large databases," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2000, pp. 22–33.

[8] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, Addison Wesley, 2005.

[9] S. Thompson, *Haskell: The Craft of Functional Programming* (2nd ed.), Addison Wesley, 1999.

[10] M. Zaki and K. Gouda, "Fast vertical mining using diffsets," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2003, pp. 326–335.

Nittaya Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology, Thailand. She received her B.S. from Mahidol University, Thailand, in 1985, M.S. in computer science from the Prince of Songkla University, Thailand, in 1991 and Ph.D. in computer science from Nova Southeastern University, USA, in 1999. She is a member of ACM and IEEE Computer Society. Her research of interest includes Knowledge Discovery in Databases, AI, Logic Programming, Deductive and Active Databases.

Kittisak Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology, Thailand. He received his bachelor degree in Mathematics from Srinakarinwirot University, Thailand, in 1986, master degree in computer science from the Prince of Songkla University, Thailand, in 1991 and doctoral degree in computer science from Nova Southeastern University, USA, in 1999. His current research includes Data mining, Artificial Intelligence, Functional Programming, Computational Statistics.