Comparison of Apriori, FP-growth and Dynamic FP- growth for Frequent Patterns

War War Cho, Nwe Nwe
University of Computer Studies, Hpa-an
yinyinlayyoo.2010@gmail.com

Abstract

Frequent pattern mining is one of the active research themes in data mining. It is an important role in all data mining tasks such as clustering, classification, prediction and association analysis. Frequent pattern is the most time consuming process due to a massive number of patterns generated. Frequent patterns are generated by using association rule mining algorithms that use candidate generation and association rules such as Apriori algorithm, and the algorithms without candidate set generation and FP-tree such as FP-growth and DynFP-growth algorithms. In this paper, this system used computer sales items for generating frequent patterns by applying Apriori, FP-Growth and DynFP-Growth algorithms. The frequent patterns are used for comparing performance results with run time and scalability. The scalability and run time of DynFP-Growth algorithm is faster than Apriori and FP-Growth algorithms.

Keyword: frequent pattern mining, association mining algorithms, performance improvements

1. Introduction

Mining frequent patterns [16] or itemsets are a fundamental and essential problem in many data mining applications. These applications include the discovery of association rules, strong rules, correlations, sequential rules, episodes, multidimensional patterns, and many other important discovery tasks. Algorithm for extracting and reconstructing of all association rules shows the results of experiments carried out on real datasets and it shows the usefulness of each approach. Databases, has been adopted for a field of research dealing with the automatic discovery of implicit information of knowledge within databases [9]. The implicit information within databases, and mainly the interesting association relationships among sets of objects, that lead to association rules, may disclose useful patterns for decision support, financial forecast, marketing policies, even medical diagnosis and many other applications. As shown in [10], mining association rules may require iterative scanning of large databases [1- 5, 11, 12] which is costly in processing. The system consists of get transactions from the database, extracting frequent itemsets using Apriori algorithm, FP-Growth algorithm and DynFP-growth algorithm. The system creates a transaction file which consists of amount of sales data by scanning the data from the database. The transaction file includes item Names. Then, algorithms are used for mining the frequent pattern with the minimum support count. This system produces frequent patterns. Finally, the system used the frequent patterns for comparing the result with run time and scalability.

A major step forward in improving the performances of these algorithms was made by the introduction of a novel, compact data structure, referred to as frequent pattern tree, or FP-tree [11], and the associated mining algorithm, FP-growth.

The main difference between the two approaches is that the Apriori-like techniques are based on bottom-up generation of frequent itemset combinations and the FPtree based ones are partition-based, divide-and-conquer methods. After conducting several performance studies developed an improved FP-tree based technique, named Dynamic FP-tree [6]. The developed method clearly indicates a performance gain mainly when applied on real world sized databases.

The remaining of this paper is organized as follows. The related work is described in section 2. The main aspects of Apriori, FP-growth and DynFP-growth algorithms and frequent itemsets generation are presented in section 3. Section 4 described the system design. The paper is concluded in section 5.

2. Related work

The Apriori [15] is a basic algorithm for finding frequent patterns. It has been described by several variations for improving efficiency and scalability. This algorithm is almost suffered inherently from two problems; multiple database scans that are costly and generating lots of candidates.

The frequent pattern tree or FP-tree as a prefix-based tree structure, and an algorithm called FP-growth. The FP-tree stores only the frequent items in a header table which is sorted in descending order. The highly compact nature of FP-tree enhances the performance of the FP-growth. The FP-tree construction requires two database scans.

After conducting several performance studies developed an improved FP-tree based technique, named Dynamic FP-tree [16]. The developed method clearly indicates a performance gain mainly when applied on real world sized databases.

3. Mining Frequent Pattern Using Algorithms

3.1. The Apriori Algorithm

The Apriori algorithm used the data as shown in Table1 for finding all frequent itemsets. The first pass of the algorithm simply counts [7] item occurrences to determine the large 1-itemsets. A subsequent pass, say pass *k*, consists of two phases. First, the large itemsets L_{k-1} found in the (*k*-1)th pass are used to generate the candidate itemsets C_k, using the Apriori candidate generation function (apriori-gen) described below. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, an efficient determination if the candidates in C_k that are contained in a given transaction *t* is needed. A hash-tree data structure [11] is used for this purpose. The Apriori algorithm is:

Input: Database D of transaction s; min-sup
Output: L, frequent itemsets in D.

L_1 = {*large 1-itemsets*};
for ($k = 2$; $L_{k-1} \neq \emptyset$; k++)
begin
C_k = *apriori-gen*(L_{k-1}); //*New candidates*
forall *transactions* $t \in D$
begin
C_t = *subset*(C_k, t);
//*Candidates contained in t*
forall *candidates* $c \in C_t$ ***do***
c.count++;
end
$L_k = \{ c \in C_k \mid c.\text{count} \geq \text{minsup} \}$
End *Answer* = $\cup_k L_k$;

The *apriori-gen* function takes as argument L_{k-1}, the set of all large (*k*-1)-itemsets. It returns a superset of the set of all large *k*-itemsets and is described in [1]. This system used computer sales center data items which are described in Table 1.

Table 1 Transactional Database

TID	Itemsets
T1	I1,I2,I3
T2	I1,I2,I4
T3	I2,I4
T4	I1,I2,I5
T5	I2,I3
T6	I2,I3
T7	I1,I3
T8	I1,I3
T9	I1,I2,I3,I5
T10	I1,I2

I1=Monitor
I2=CPU
I3=Motherboard
I4=Hard disk
I5=Memory
I6=Printer
I7=Speaker
I8=Casing
I9=UPS
I10=DVDR/W

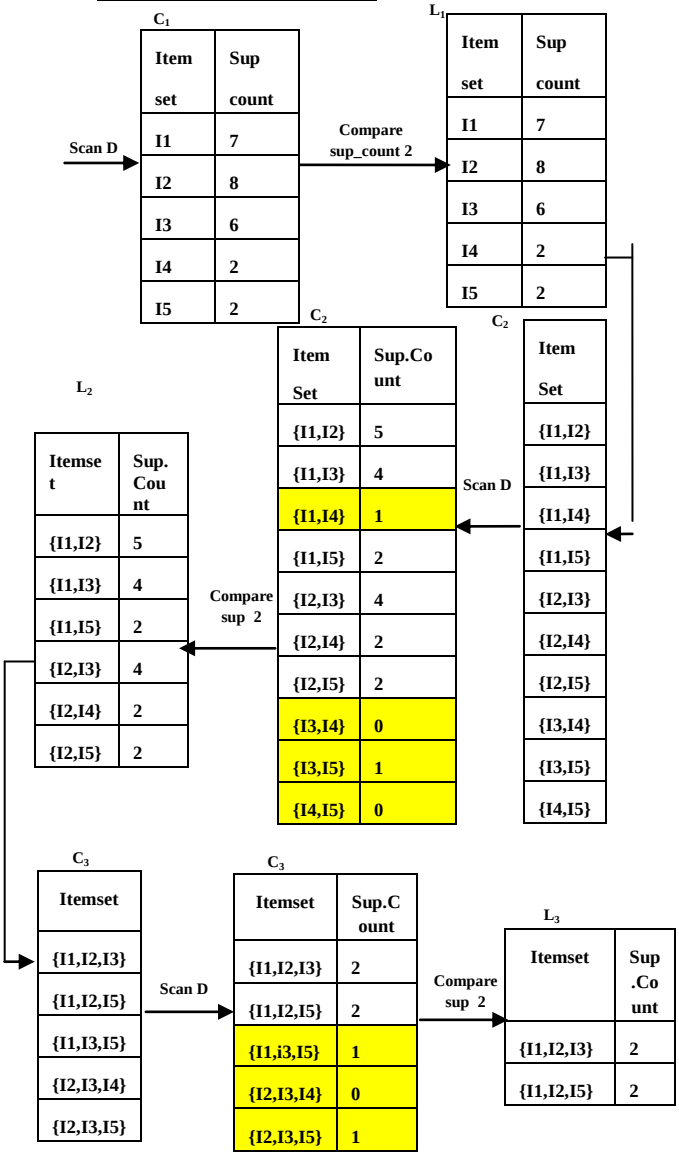

Figure 1 Generation of Frequent itemsets where minimum support count is 2

The Apriori algorithm uses a level-wise approach for generating association rules, where each level corresponds to the number of items that belong to the rule consequent described in Figure 1. Frequent itemsets do no mean association rule. One more step is required to convert these frequent itemsets into rules. If the minimum confidence threshold is 70%, then the rules are generated. Since there are generated that are strong because confidence is greater than the minimum confidence.

- R1: I1 →I2^I5, Confidence = 2/7 = 28%
  R1 is rejected.
- R2: I2 →I1^I5, Confidence = 2/8= 25%
  R2 is rejected.
- R3: I5 →I1^I2, Confidence = 2/2 = 100%
  R3 is selected.
- R4: I2^I5→ I1, Confidence = 2/2 = 100%
  R4 is selected.
- R5: I1^I5 → I2, Confidence = 2/2 = 100%
  R5 is selected.
- R6: I1^I2 → I5, Confidence = 2/5= 40%
  R6 is rejected.

3.2. The FP-growth Algorithm

As shown in [12], the main bottleneck of the Apriori-like methods is at the candidate set generation and test. This problem was dealt with by introducing a novel, compact data structure, called frequent pattern tree, or FP-tree then based on this structure an FP-tree-based pattern fragment growth method was developed, FP-growth. The algorithm is as shown in below.

Algorithm: (FP-Growth)
Input: *D* – transaction database; *s* – minimum support threshold.
Output: The complete set of frequent patterns.
Method: **call** FP-growth(*FP-tree*, *null*).
1. scan *D* to discover frequent items and their counts
2. create the root of *FP-tree* labeled as *null*
3. scan *D* and add each transaction to *FP-tree* (omitting non-frequent items)
4. call *FP-growth*(*FP-tree*, *null*)
procedure *FP-growth*(*FP-tree*, ⟨⟩ {
if *FP-tree* contains a single path *P*
then for each combination β of nodes in *P* **do** generate frequent itemset β∪α
with *support*(β∪α,*D*)= min support of nodes inβ*;*
else for each a_i in header table of *FP-tree* **do** {
generate frequent itemset β = a_i∪α with *support*(β,*D*) = *support*(a_i,*D*); construct β's conditional pattern base and β's conditional *FP-tree*$_\beta$; **if** *FP-tree*$_\beta \neq \phi$ **then** *FP-growth*(*FP-tree*$_\beta$,β);
}

The function insert-tree ([p/P], T) is performed as follows. It T has a child N such that N. item-name = p.item-name, then increment N's count by 1; else create a new node N, with its count initialized to 1, its parent link linked to T, and its node-link linked to the nodes with some (P, N) recursively. The result of FP growth is expressed in Figure 2.

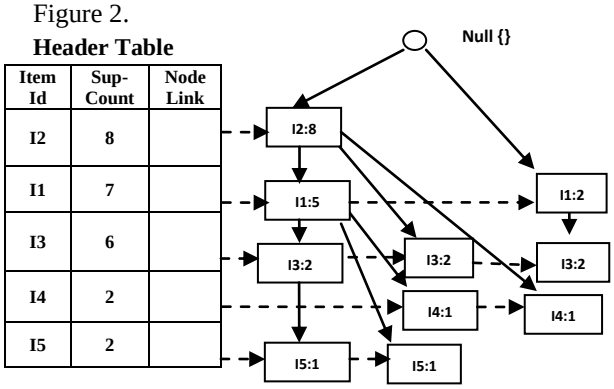

Figure 2 The FP-Tree with Header Table

3.3. The DynFP-growth Algorithm

The dynamic FP-tree reordering algorithm is a "promotion" to a higher order of at least one item is detected. The dynamic reordering one doesn't have to rebuild the FP-tree even if the actual database is updated. This approach can provide a very quick response to any queries even on databases. Because the dynamic reordering process, [3] proposed a modification of the original structures, by replacing the singly linked list with a doubly linked list for linking the tree nodes to the header and adding a master-table to the same header. The algorithm is as shown in below.

Algorithm (Dynamic FP-tree construction)
Input: A transactional database *DB* and a minimum support threshold ξ .
Output: Its frequent pattern tree, **FP-tree**
Method: The **FP-tree** is constructed in the following steps:

The *reorder* () function is performed as follows:
1. Gather the "promoted" items into a *reorderList* ordered according to their support (descending) and lexicographical order.
2. Call *checkpoint* () to update the insertion order into the FP-tree.

3. For each item from *reorderList* go through the list of linked nodes and for each of these nodes call *moveUp* (*node*) to place that node into the correct position in the FP-tree, according to the *header*'s *master-table*. The *moveUp* (*node*) function is defined as:

1. Repeat the steps (a. to g.) until the *node* and its current *parent* are in the *properOrder*

a. Take the *node*'s *parent*'s parent (*pparent*)

b. If *parent* has the same support as the *node*, remove the *parent* from its parent's *childNodes* and assign it to *newNode*

c. Else perform the following actions:

i. Create a *newNode* with the same item as the *parent*, but having the *node*'s support.

ii. Link it into the *parent*'s list of nodes with the same item.

iii. Adjust the support of the *parent*, by subtracting the *node*'s support

iv. Remove the *node* from the *childNodes* of the *parent*

d. Replace the *childNodes* into the *newNode* with the *childNodes* from the *node* and update the *parent* link of the *childNodes* with their new parent (*newNode*).

e. Set the parent link of the *node* to *pparent* (the original *parent*'s parent), initialize its *childNodes* with the *newNode*, and set the *newNode*'s parent to *node*.

f. (*optional step*) If there is already an *existingNode* for the *node*'s item in the *pparent*'s *childNodes*, then call *merge*(*existingNode*, *node*), and continue with the *existingNode* as the current *node*.

g. Otherwise insert the *node* into the *childNodes* of *pparent*.

The reorder items are I2:1, I3:1. DynFP-growth does not depend on support but only on the database size, this is because the tree construction technique does not need the support information. The tree will contain all the database transactions and depending on the required support. The results in Figure 3 will contain only the itemsets that have their frequency greater than the required support.

Item Id	Sup-Count	Node Link
I2	9	
I1	7	
3	7	
I4	2	
I5	2	

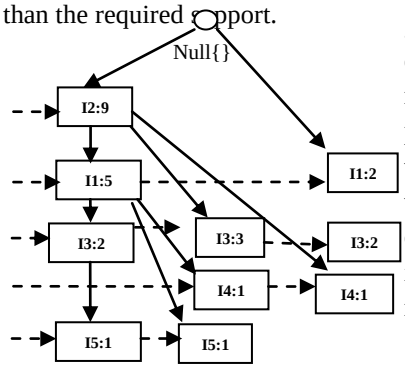

Figure 3 The DynFP-Tree with Header Table

4. System Design

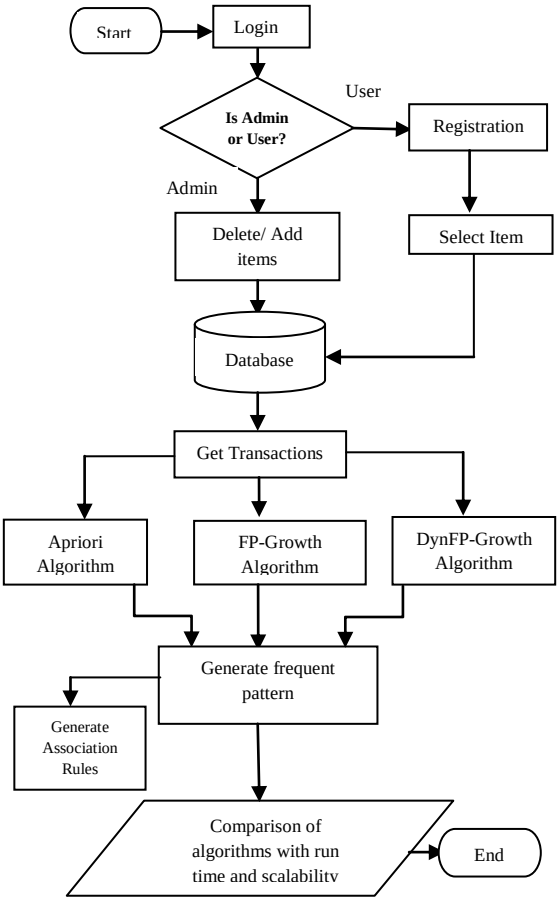

Figure 4 System Flow Diagram

The input of this system is transactional records from store database and these records are extracted frequent patterns by using Apriori, FP-Growth and DynFP-Growth algorithms. Then the resulted frequent itemsets are generated as association rules by using Apriori algorithm. It is produced candidate generation. Then compare the frequent patterns with minimum support count.

If support count is greater than or equal to minimum support for each item, the system determines the support count for candidate generation of pair items. If pair item's support count is less than the support count, the itemsets remove from the system and produce the frequent patterns. When there is not more itemsets in the transaction file, the desire confidence is greater than or equal to minimum confidence. If the confidence is less than minimum confidence for itemsets, it removed these itemsets and produce rules.

The mining of the FP-tree is started from each frequent length 1 pattern and create its conditional pattern base and then construct its conditional FP-tree, and perform mining recursively on such a tree.

The DynFP-Growth algorithm reording the already FP-tree and perform promoted itemset on the FP-tree [13]. Then the resulted frequent patterns from each algorithm are used for comparing with run time and scalability.

4.1. Experimental Result

The performance and scalability of the Apriori, FP-Growth and DynFP-Growth algorithms generated data sets with 150 transactions, and support counts between 1% and 4% are used. Any transaction may contain more than one frequent itemset [14]. The numbers of items in a transaction are numerous. The generated data sets depend on the number of items in a transaction and number of items in a frequent itemset, etc. The parameters to generate the test data sets are defined in Table 1.

Table 2 Sample Itemsets

|D|	Number of transactions
|T|	Average size of the transactions
|L|	Number of maximal potentially large itemsets
N	Number of items

Table 2 describes the test data sets. It is generated for a number of items N = 10 and a maximum number of frequent itemsets |L| = 300. The average size of the transaction |T| is 10.

The run time results in millisecond are presented in Table 3 by applying Apriori, FP-growth and DynFP-growth algorithms for support count of 3%.

Table 3 Run Time of the system

Transaction (K)	Run Time Complexity (millisecond)		
	Apriori	FP-growth	DynFP-growth
10	1210.00	605.00	363.00
20	3630.00	1815.00	1089.00
30	7260.00	3630.00	2178.00
150	87846.00	43923.00	26353.80

In Figure 5 describes the graph of run time results. According to these result, DynFP-growth is more efficient than other both Apriori and FP-growth algorithms.

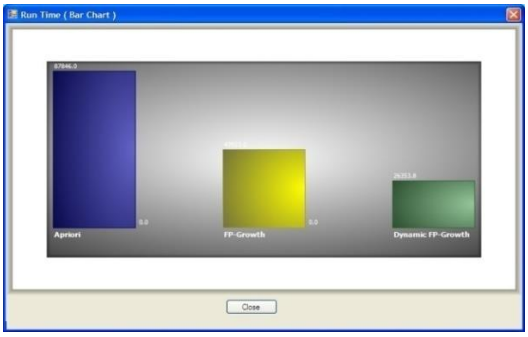

Figure 5 Run Time Comparisons between Apriori and FP-Growth and DynFP-Growth

The scalability can be defined as the ability of a system to keep its performance when the system size is scaled up [8]. The scalability function is

$$\psi(p,p') = \frac{p'W}{pW'} \qquad (4.1)$$

Where p=initial number of processor
p'= scaled number of processors
W= initial problem size
W'= scaled problem size
ψ =scalability

This system used 150 transactions for scalability testing and one processor. This system used 150 transactions for scalability testing and one processor. There is 1 to 4 percent support count (%) used for scalability run time with processor. These results are presented in Table 4.

Table 4 The scalability result of Apriori and FP-Growth and DynFP-Growth algorithms

Support count (%)	scalability (millisecond)		
	Apriori	FP-growth	DynFP-growth
1	93.00	90.70	86.50
2	90.00	86.80	83.70
3	87.00	84.40	80.20
4	84.00	80.00	77.10

The best performance is obtained by the DynFP-growth algorithm according to scalability function. This system used one processor and 150 transactions. The size of the system has 3MB for scalability testing. The execution time of the Apriori algorithm with support count 1% is three times longer than the execution time of the FP-growth algorithm and up to five times longer than DynFP-growth. The execution time of the Apriori, FP-Growth and DynFP-Growth are different values of the support count on a data set with 150 transactions. The Apriori algorithm has a lower scalability than the FP-growth and DynFP-growth

algorithms at support count of 4%.These result are presented in Table 4.

In Figure 6 describes the graph of scalability results .According to these result DynFP- growth's scalability is better than Apriori and FP- growth algorithm.

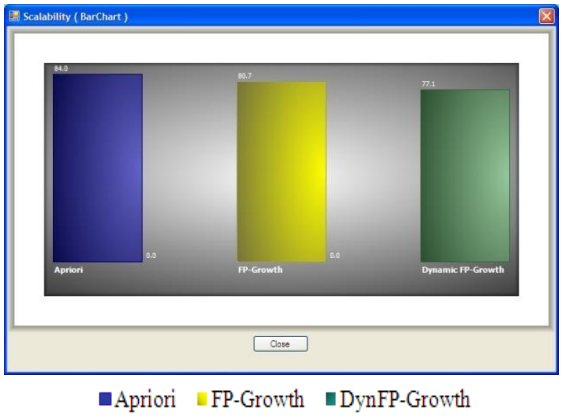

Figure 6 Scalability of the algorithms

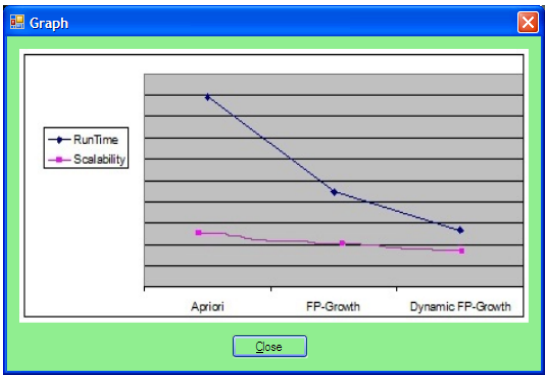

Figure 7 Graph of the compare algorithms

This system's compare result described in Figure 7. This graph expressed the run time and scalability result of the system. The run time of DynFP-growth algorithm is more speedily than other Apriori and FP-growth algorithms. The scalability of DynFP-growth algorithm is faster than other Apriori and FP-growth algorithms.

5. Conclusion

In this paper, the DynFP-growth algorithm behaves better than the Apriori and FP-growth algorithms according to the experimental data presented. The FP-growth algorithm needs at most two scans of the database, while the number of database scans for the candidate generation of Apriori increases with the dimension of the candidate itemsets. DynFP-growth algorithm does not need to multiple database scan. It scans only one database at a time for update on FP-tree. The performance of the FP-growth and DynFP-growth algorithms are not influenced by the support count. In this paper, the experimental results of frequent patterns are run time and scalability. So, DynFP-growth algorithm's scalability and run time is faster than Apriori and FP-growth algorithms.

References

[1] R. Agrawal, R. Srikant. "Fast algorithms for mining association rules in large databases".

[2] A. Ashoka Savasere, B. Navathe. "An Efficient Algorithm for Mining Association Rules in Large Databases."

[3] C. Cornelia Győrödi, Robert Győrödi, T. Cofeey & S. Holban – "Mining association rules using Dynamic FP-trees".

[4] L. Cristofor, "Mining Rules in Single-Table and Multiple-Table Databases".

[5] M. H. Dunham. "Data Mining. Introductory and Advanced Topics". Prentice Hall, 2003.

[6] U.M. Fayyad, et al.: "From Data Mining to Knowledge Discovery: An Overview", 1996

[7] G. Grahne, & Zhu, J. "Fast Algorithm for frequent Itemset Mining Using FP-Trees".

[8] A. Gupta and V. Kumar, "Scalability of Parallel Algorithms for Matrix Multiplication".

[9] C. Győrödi, R. Győrödi. "Mining Association Rules in Large Databases". Oradea, Romania, 2002.

[10] R. Győrödi, C. Győrödi. "Architectures of Data Mining Systems". Oradea, Romania, 2002.

[11] C. Győrödi, T. Cofeey & S. Holban, Robert Győröd– "Mining association rules using Dynamic P-trees", 2003.

[12] J. Han, J. Pei, Y. Yin. "Mining Frequent Patterns without Candidate Generation". 2000.

[13] J. Han, M. Kamber, "Data Mining Concepts and Techniques", San Francisco

[14] C. Silverstein, Brin, S., Motwani, R., and Ullman, J. 1998. Scalable techniques for mining causal structures. 1998

[15] R. Srikant, and Agrawal, R. 1997, "Mining association rules with item constraints", 1997

[16] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for Fast Discovery of Association Rules," 1997.