International Journal of Soft Computing 2 (1): 21-36, 2007

Efficient Association Rules for Data Mining

[1]C.M. Velu, [1]M. Ramakrishnan, [2]V. Somu, [1]P. Loganathan and [1]P. Vivekanandan
[1]Department of Mathematics, Anna University, Chennai-600 025, India
[2]Ramanujam Computing Centre, Anna University, Chennai-600 025, India

Abstract: Frequent Item Sets (FIS) play an essential role in many Data Mining (DM) tasks. We want to find interesting patterns from databases (DBs), such as Association Rules (ARs), correlations, classifiers, clusters and many more. The motivation for searching Ars to examine customer's buying behavior. ARs describe how often items are dependent on each other to purchase together. For example, an AR beer 100% ⇒ chips 80% states that four of five customers that bought beer also bought chips. Such rules can be useful for decisions concerning product pricing, promotions, store layout and many others. Since their introduction in 1993 by Argawal *et al.*, the FIS and AR mining problems have received a great deal of attention. During the past decade, hundreds of papers have been published to solve these mining problems more efficiently. In this study, we explain the basic FIS and compare various AR algorithms to extract required information from DBs. We describe the main techniques used to solve these problems.

Key words: Transaction database, frequent itemet, clusters, association rules, minimal support threshold, support monotonicity, correlation, etc.

PROBLEM DESCRIPTION

Let I be a set of items. A set $X = \{i_1,.. I_k\} \subseteq I$ is called an item set. A transaction over I is a couple I = (tid, I) where tid, is the transaction identifier and I is an item set. A transaction I = (tid, I) is said to support an item set(IS), $X \subseteq I$, if $X \subseteq I$. A transaction database(TDB), D over I is a set of transactions over I. The cover of an item set X in D consists of the set of transaction identifiers in D that support X: cover (X, D):= {tid | (tid, I)∈D,X ⊆ I}. The support of an item set X in D is the number of transactions in the cover of X in D: support (X,D): = |cover(X,D)|. The FIS of X in D is the probability of X occurring in a transaction T∈ D: frequency (X, D): = P(X) =support(X,D)/ |D|. Note that |D| = support({},D). An item set is called frequent if its support is not less than a given absolute minimal support threshold (MST) σ_{abs}, with $0 \le \sigma_{abs} \le |D|$. When working with frequencies of ISs instead of their supports, we use a relative minimal frequency threshold(MFT), σ_{rel}, with $0 \le \sigma_{rel} \le 1$. Obviously, $\sigma_{abs} = [\sigma_{rel} \bullet |D|]$. In this paper, we will only work with the absolute MST for item sets. Table 1. shows notations used in this study.

Definition 1: Let D be a TDB over a set of items I and a MST, σ. The collection of FIS in D with respect to σ is denoted by F(D, σ) := {X ⊆ I | support(X,D) ≥}, if D,σ and γ are clear from the context.

Table 1: Notation used in this study

Symbol	Description
D	Transaction database
I	Set of items appearing in D
K	Length of an items set, of algorithm step counter
m	Size of I, i.e. number of distinct items in D
m_k	Number of distinct items at step k
n	Number of transactions
n_k	Number of transactions at step k
t	A genetic transaction
s	Minimum support threshold
F, F_k	Setoff frequent item sets, of length k
C, C_k	Set of candidate item sets, of length k
σ	Minimum threshold
σ, vα	Confidence thresholds
Bd(F)	Negative bolder

Problem 1: Given a set of items I, a transaction DB, D over I and MST, σ, find F(D,σ). An AR is an expression of the form X⇒Y, where X and Y are item sets and X∩Y = {}. Such a rule expresses the association that if a transaction contains all items in X, then that transaction also contains all items in Y. X is called the body and Y is called the head. The support of an AR X ⇒ Y in D, is the support of X∪Y in D and similarly, the frequency of the rule is the frequency of X∪Y. The confidence of an AR X⇒Y in D is the conditional probability of having Y contained in a transaction, given that X is contained in that transaction: confidence (X⇒Y,D) := P(Y|X) =support (X∪Y,D)/support(X,D). The rule is called confident if P (Y|X) exceeds a given MCT γ, with $0 \le \Upsilon \le 1$.

Corresponding Author: C.M. Velu, Department of Mathematics, Anna University, Chennai-600 025, India

Definition:2 Let D be a TDB over a set of items I, a MST and a MCT σ. The collection of frequent and confident ARs w.r.t. σ and is denoted by : R (D,σ,γ):= {X⇒ Y | X, Y⊆ I,X∩Y = {}, X ∪Y∈ F(D, σ), confidence(X⇒Y,D) ≥ ϒ}, or simply R if D, σ and are clear from the context.

Problem 2: (ARM) : Given a set of items I, a TDB D over I and MST and confidence thresholds σ and ϒ, find R (D,σ,ϒ). Besides the set of all ARs, we are also interested in the support and confidence of each of these rules. If we are given the support and confidence thresholds σ and ϒ, then every frequent item set X also represents the trivial rule X ⇒ {} which holds with 100% confidence. Also note that for every FIS I, all rules X⇒Y, with X∪Y = I, hold with at least σ_{rel} confidence. Hence, the MCT must be higher than the MFT to be of any effect.

Example 1: Consider the DB shown Table 2.

Table 1: An example TDB D. Table 3 shows all FIS in D with respect to a MST of 1. Table 4 shows all frequent and confident ARs with a support threshold of 1 and a confidence threshold of 50%.

Item set mining (ISM): The task of discovering all FISs is quite challenging, because, DBs could be massive, containing millions of transactions, that the task of support counting is a tough problem.

Search space (SS): The SS of all item sets contains exactly 2|I| different item sets. If I is large enough, then the naive approach to generate and count the supports of all item sets over the DB can't be achieved within a reasonable period of time. For example, in many applications, I contains thousands of items and then, the number of item sets is more than the number of atoms in the universe (≈1079). Instead, we could generate only those item sets that occur at least once in the TDB. For large transactions, this number could be too large. We could optimize and generate only those subsets. This technique has been studied by Amir *et al.*[1] and has proven to pay off for very sparse TDBs. It occupies more memory space. Therefore, several solutions have been proposed to perform more directed and optimized search. During the search, several collections of CISs are generated.

Definition 3: CIS: Given a TDB D, a MST σ, and an algorithm that computes F(D, σ), an item set I is called a candidate, based on frequency. Obviously, the size of a collection of CISs fits within memory. In the best case, only the FISs are generated and counted. 1. Several

Table 2: Over the set of items I = {beer, chips, pizza, wine}

Tid	X
100	{beer, chips, wine}
200	{beer, chips}
300	{pizza, wine}
400	{chips, pizza}

Table 3: Itemsets and their support in D

Item set	Cover	Support	Frequency %
{}	{100,200,300,400}	4	100
{beer}	{100, 200}	2	50
{chips}	{100, 200,400}	3	75
{pizza}	{300,400}	2	50
{wine}	{100, 300}	2	50
{beer, chips}	{100, 200}	2	50
{beer, wine}	{100}	1	25
{chips, pizza}	{400}	1	25
{chips, wine}	{100}	1	25
{pizza, wine}	{300}	1	25
{beer, chips, wine}	{100}	1	25

Table 4: ARs and their support and confidence in D

Rule	Support	Frequency %	Confidence %
{beer}-{chips}	2	50	100
{beer}-{wine}	1	25	50
{chips}-{beer}	2	50	66
{pizza}-{chips}	1	25	50
{pizza}-{wine}	1	25	50
{wine}-{beer}	1	25	50
{wine}-{chips}	1	25	50
{wine}-{pizza}	1	25	50
{beer, chips}-{wine}	1	25	50
{beer, wine}-{chips}	1	25	100
{chips, wine}-{beer}	1	25	100
{beer}-{chips, wine}	1	25	50
{wine}-{beer, chips}	1	25	50

efficient algorithms have been proposed to find only the positive border of all FISs, but if we want to know the supports of all item sets in the collection, we still need to count them. Hence it still poses several interesting open problems[2-4].

Proposition 1: (Support monotonicity) Given a TDB D over I, let X, Y ⊆ I be two item sets. Then, X ⊆Y, support(Y)≤ support(X).

Proof: This follows immediately from cover(Y)≤cover(X). Hence, if an item set is infrequent, all of its supersets must be infrequent. The monotonicity property is also called as downward closure property. The SS of all item sets can be represented by a subset-lattice, with the empty item set at the top and the set containing all items at the bottom. The collection of FISs F(D,σ) can be represented by the collection of maximal frequent item sets, or the collection of minimal infrequent item sets, w.r.t set inclusion.

Definition 4: (Border) Let F be a downward closed collection of subsets of I. The Border Bd(F) consists of those item sets X⊆ I such that all subsets of X are in F and no superset of X is in F:

Bd(F):= {X⊆ T|∀ Y⊂ X : Y∈ F∧∀Z⊃X : Z ∉ F }, These item sets in Bd(F) that are in F are called the positive border $Bd^+(F)$:

$Bd^+(F)$:= {X⊆ T|∀ Y⊆ X : Y∈ F∧∀Z⊃X : Z ∉ F }, those item sets in Bd(F) that are not in F are called the negative border $Bd^-(F)$:

$Bd^-(F)$:= {X⊆ T|∀ Y⊇ X : Y∈ F∧∀Z⊃X : Z ∉ F },

The lattice for the FISs for Example 1, together with its borders, is shown.

Theorem 2: Mannila[4] Let D be a TDB over I and σ is a MST. Finding the collection F(D, σ) requires that at least all item sets in the negative border Bd(F) are evaluated. Note that the number of item sets in the positive or negative border of any given closed collection of item sets over I can still be large, but it is bounded by (|x|/(|x|/2)). In combinatorics, this upper bound is well known as Sperner's theorem. If the number of FISs for a given DB is large, it becomes infeasible to generate all of them. A FIS of size k includes the existence of at least 2^k - 1 item sets. Several proposals have been made to generate all FISs for a given TDB[5-7].

DATABASE (DB)

To compute the supports of CISs, presented in a DB can be represented by a binary 2D matrix in which every row represents an individual transaction and the columns represent the items in I. Such a matrix can be implemented horizontal or vertical layout, which is most commonly used. In vertical data layout, the DB consists of a set of items, each followed by its cover[8]. Table 5. a shows both layouts for the DB from Example 1. For both layouts, it is possible to use the exact bit-strings from the binary matrix[9].

Table 5: Horizontal and vertical DB layout of D

	Beer	Wine	Chips	Pizza		Beer	Wine	Chips	Pizza
100	1	1	1	0	100	1	1	1	0
200	1	0	1	0	200	1	0	1	0
300	0	1	0	1	300	0	1	0	1
400	0	0	1	1	400	0	0	1	1

To count the support of an item set X using the horizontal DB layout, we need to scan the DB completely and test for every transaction T, whether X⊆T. The number of transactions in a DB is correlated to the MST in the DB. The vertical DB layout has the major advantage that the support of an item set X can be computed by intersecting the covers of any two subsets Y,Z⊆ X, ∋ Y∪Z = X, proved in Proposition 4.

ASSOCIATION RULE MINING (ARM)

ARM is one of the most popular DM task, application of knowledge extraction by this kind of analysis. The extraction of ARs from a DB is typically composed by two phases. First, it is necessary to find the frequent patterns. Once such patterns are determined, the actual ARs can be derived in the form of logical

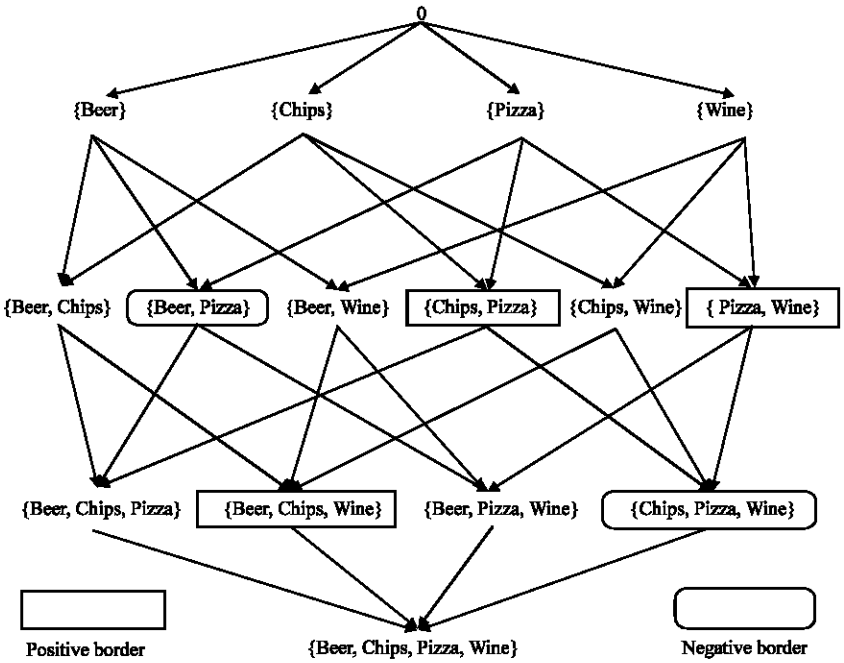

Fig. 1: The latice for the itemsets of example 1 and its horder

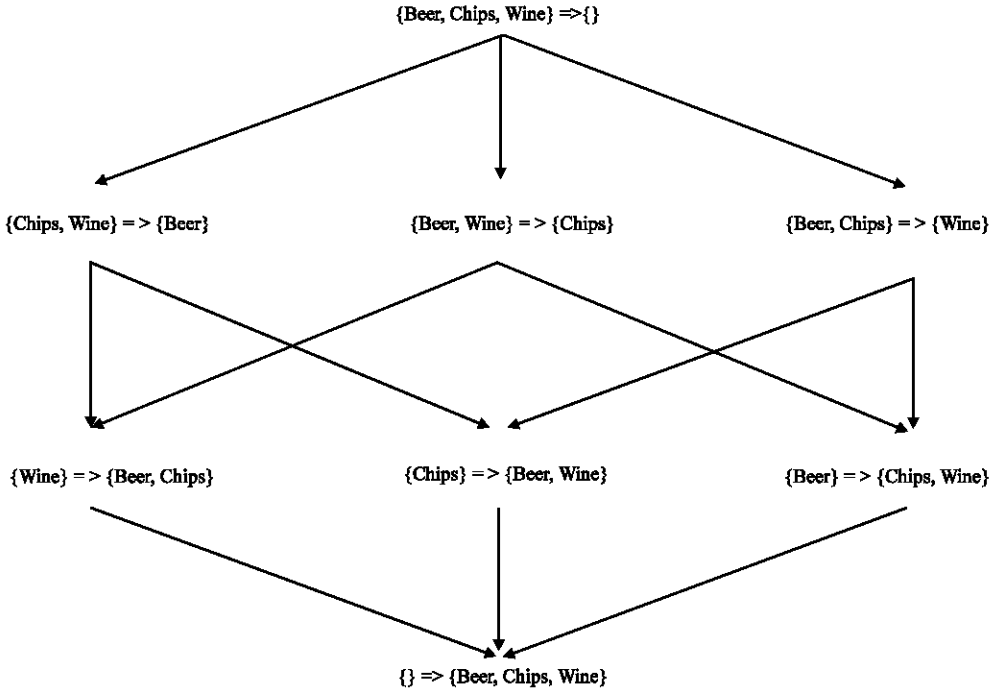

Fig. 2: An example of a lattice representing a collection of Ars for {beer, chips, wine}

implications: X ⇒ Y which reads whenever X occurs in a record, most likely Y also will occur. The computationally intensive part of ARM is the determination of frequent patterns, as confirmed by the considerable amount of efforts devoted to the design of algorithms. DCI (Direct Count and Intersect), algorithm shows significant performance improvement over previous approaches.

Proposition 3: Let X,Y,Z ⊆ T be three item sets, ∋ X ∩Y = { }. Then, Confidence (X \ Z ⇒ Y U Z) ≤ confidence (X ⇒ Y).

Proof: Since, X U Y ⊆ X U Y U Z and X \ Z ⊆ X, we have, Support(X U Y U Z) / support(X \ Z) ≤ support(X U Y) / support(X)

If a certain head of an AR over an item set I causes the rule to be unconfident, all of the head's supersets must result in confident rules. For a given frequent item set I, the SS of all possible ARs X⇒Y, ∋ X ∪Y = I, can be represented by a subset-lattice with respect to the head of a rule, with an empty head at the top and with all items in the head at the bottom. Figure 2 shows such a lattice for the item set {beer, chips, wine}.

EXAMPLE DATA SETS

Table 6 shows the number of items, number of transactions in each data set, minimum, maximum and average length of the transactions. Additionally, Table 7 shows for each data set the lowest MST, number of frequent items, item sets, size of the longest FIS that was found.

Table 6: Data set characteristics

Data set	#Items	#Transactions	Min T	Max T	Avg T
T40I10D100K	942	100000	4	77	39
Mushroom	119	8124	23	23	23
BMS-	497	59602	1	267	2
Webview-1	13103	41373	1	52	9
Basket					

Table 7: Data set characteristics

Data set	σ	F_1	F	Max{k| $\|F_k\|>0$}
T40I10D100k	700	804	550126	18
Mushroom	600	60	945309	16
BMS-Webview-1	36	368	461521	15
Basket	5	8051	285758	11

The apriori algorithm: AIS algorithm generates all FISs and CARs proposed by Agrawal *et al.*[10]. The algorithm was improved and called as Apriori by Agrawal *et al.*, by exploiting the monotonicity property of the support of item sets and the CARs[2,11].

Item set mining (breadth first search algorithm BFS): The item set mining phase of the Apriori algorithm is given in Algorithm 1. We use the notation X[i], to represent the i^{th} item in X. The k-prefix of an item set X is the k-item set {X[1],. . . X[k]}.

Algorithm 1 : Apriori-Itemset mining

Input: D, σ
Output: F(D, σ)
1: C1:={{I} |i∈I}
2: k:=1
3: while $C_k \neq \{\}$ do
4: //compute the supports of all candidate item sets
5: for all transactions (tid,I)∈D do
6: for all candidate item sets $X \in C_k$ do
7: if $X \subseteq I$ then
8: X. support++
9: end if
10: end for
11: end for
12: // Extract all frequent item sets
13: F_k :={X|X supports>= σ}
14: //Generate new candidate item sets
15: for all X, $Y \in F_k$, X[I]=Y[I] for $1 \leq I \leq k-1$,and X[k]<Y[k] do
16: I =X∪{Y[k]}
17: if ∀J⊂I, |J|=k :J ∈F_k then
18: C_{k+1}:=C_{k+1}∪
19: end if
20: end for
21: k++
22: end while

The algorithm performs a BFS through the SS of all item sets by iteratively generating CIS C_{k+1} of size k+1, starting with k = 0 (line 1). More specifically, C_1 consists of all items in I and at a certain level k, all item sets of size k + 1 in $Bd(F_k)$ are generated. This is done in two steps. First, F_k is joined with itself. The union X ∪Y of item sets, F_k is generated if they have the same k - 1-prefix (lines 20-21). In the prune step, X ∪Y is only inserted into C_{k+1} if all of its k-subsets occur in F (lines 22-24). To count the supports of all K-items of CISs, is scanned as one transaction at a time and incremented (lines 6-12). All item sets that turn out to be frequent are inserted into F_k (lines 14-18).

ARM algorithm: As per Algorithm 2, we can generate all FAR and CARs. First, all FISs are generated using Algorithm 1. Then, every FIS I is divided into a candidate head Y and a body X = I/Y. This process starts with Y = {}, which always holds with 100% confidence (line 4). The algorithm iteratively generates candidate heads C_{k+1} of size k + 1, starting with k = 0 (line 5). A head is a candidate if all of its subsets are to represent CRs. To compute the confidence of a candidate head Y, the support of I and X is retrieved from F. All heads that result in CRs are inserted into H_k (line 9). At the end, all CRs are inserted into R (line 20). Computing the CR requires the support of at most 2 item sets. If the number of item sets and ARs is not too large, then the time to find all such rules, takes only the time needed to find all frequent such sets. The computation of all frequent and ARs becomes straightforward, when item sets and its ARs are known. To compute the confidence of an AR X ⇒Y, with X⊆Y = I, we need to find the supports of I and X, which can be retrieved from the collection of FIS.

Data structures (DS): The candidate generation and the support counting process requires an efficient DS.

Algorithm 2 Apriori-ARM

Input: D, σ,γ
Output: R(D, σ,γ)
1: Compute F(D, σ)
2: R: ={}
3: for all I∈F do
4: R: =R∪I⇒{}
5: C1: = {{I} | I ∈ I};
6: k: =1;
7: while $C_k \neq \{\}$ do
8: // Extract all heads of confident association rules
9: H_k : ={X∈C_k| confidence(I \ X⇒ X,D)≥ϒ}
10: // Generate new candidate heads
11: for all X, Y∈H_k, X[I]=Y[I] for $1 \leq i \leq k-1$ and X[k]<Y[k] do
12: I=X∪{Y[k]}
13: if∀J⊂I, | J |=K:J∈H_k then
14: C_{k+1}: =C_{k+1}∪I
15: end if
16: end for
17: k++
18: end while
19: //cumulate all association rules
20: R: =R∪{I\X⇒X|X∈H1∪…∪H_k}
21: end for

Hash-tree (HT): To find all k-subsets of item set, all FISs in F_k are stored in a hash table. CISs are stored in a HT[12]. A node of the HT contains a list of item sets or a hash table. In an interior node, each bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth d points to nodes at depth d + 1 Item sets are stored in leaves. When we add a k-item set X during the candidate generation process, we start from the root and go down the tree until we reach a leaf. At an interior node at depth d, we decide which branch to follow by applying a hash function to the X[d] item of the item set and following the pointer in the corresponding bucket. All nodes are initially created as leaf nodes. When the number of item sets in a

leaf node at depth d exceeds a specified threshold, the leaf node is converted into an interior node, only if k > d. To find the CISs that are contained in a transaction T, we start from the root node. If we are at a leaf, we find which of the item sets in the leaf are contained in T and increment their support. If we are at an interior node and we have reached it by hashing the item i, we hash on each item that comes after i in T and recursively apply this procedure to the node in the corresponding bucket. For the root node, we hash on every item in T.

Trie data structure (TDS): In TDS[12-14], every k-item set has a node associated with it, as does its k - 1-prefix. All the 1-itemsets are attached to the root node and their branches are labeled by the item. Every other k-item set is attached to its k - 1-prefix. Every node stores the last item in the item set it represents, its support and its branches. The branches of a node can be implemented using several DSs such as a hash table and a BST. To find the CISs that are contained in a transaction T, we start at the root node. To process a transaction for a node of the trie,

- Follow the branch corresponding to the first item in the transaction and process the remainder of the transaction recursively for that branch and
- Discard the first item of the transaction and process it recursively for the node itself. The join step of the candidate generation is simple using a trie, since all item sets of size k with the same k - 1-prefix are represented by the branches of the same node. We copy all siblings of the node that represents X as branches of that node. All FISs mining algorithms presented in this paper are implemented using this trie data structure.

Optimizations: The performance is mainly dependent on the number of CISs that occur in each transaction.

AprioriTid, AprioriHybrid: Agrawal *et al.*[5,12] proposed two algorithms, AprioriTid and AprioriHybrid. The AprioriTid algorithm reduces the time for the support counting procedure by replacing every transaction in the DB by the set of CIS that occur in that transaction. This is done repeatedly at every iteration k. This is given in Algorithm 3. It performs slower than Apriori in early iterations. This is mainly due to the additional overhead that is created when C_k does not fit into main memory. If a transaction does not contain any CISs, then C_k will not have an entry for this transaction. Hence, the number of entries in C_k may be smaller than the number of transactions in the DB. At later iterations, each entry may be smaller than the corresponding transaction. In early iterations, each entry may be larger than its corresponding transaction. Hence, another algorithm, AprioriHybrid, has been proposed[5,12] that combines the Apriori and AprioriTid algorithms. This hybrid algorithm uses Apriori for the initial iterations and switches to AprioriTid when it is expected that the set C_k fits into main memory. Since the size of C_k is proportional with the number of CISs

Algorithm 3: Apriori Tid

Input: D,σ
Output: F(D,σ)
1: compute F_1 of all frequent items
2: $C^{'}_1$: D(with all items not in F_1 removed)
3: k:=2
4: while $F_{k-1} \neq \{\}$ do
5: compute C_k of all candidate k-items
6: $C^{'}_k$: ={}
7: // compute the supports of all candidate item sets
8: for all transactions (tid,T)$\in C^{'}_k$ do
9: C_T: ={}
10: for all X$\in C_k$ do
11: if {X[1],……,X[k-1]} $\in$T$\wedge${X[k-2],X[k-2], X[k]} $\in$T then
12: C_T: =$C_T \cup$ {X}
13: X. support++
14: end if
15: end for
16: if $C_T \neq \{\}$ then
17: $C^{'}_k$: =$C^{'}_k \cup$ {(tid, C_T)}
18: end if
19: end for
20: Extract F_k of all frequent k-itemsets
21: k++
22: end while

Counting candidate 2-itemsets:

DHP algorithm (Direct hashing and pruning): Park *et al.* proposed an optimization algorithm to reduce the number of CISs. During the k^{th} iteration, DHP already gathers information about CISs of size k + 1 in such a way that all (k + 1)-subsets of each transaction after some pruning are hashed to a hash table. Each bucket in the hash table consists of a counter to represent how many item sets have been hashed to that bucket so far. During the support counting phase of iteration k, every transaction is trimmed in the following way. If a transaction contains a FIS of size k + 1, any item contained in that k+1 itemset will appear in at least k times of the CIS in C^k. Instead of using the hash-tree to store and count all candidate 2-itemsets, a triangular array C is created, in which the support counter of a candidate 2-itemset {i, j} is stored at location C[i][j]. Using this

array, the support counting procedure reduces to a simple two level for-loop over each transaction.

Alogorithm 3.a : DHP

```
Input: Database
Output: frequent k-itemset
/* Database=set of transactions;
Items=set of items;
Transactions=<TID, {x∈Items}>;
F1 is a set of frequent l-itemsets*/
F1=ϕ;
/* H2 is the hash table for 2-itemsets
Read the transactions and count the
occurrences of each item and generate H2 */
for each transaction t∈Database do begin
for each item x in t do
x.count++;
for each 2-itemset y in t do
H2.add(y);
End
// Form the set of frequent l-itemsets
for each item i∈items do
if i.count/|Database|≥min sup
then F1 = F1 ∪ I;
end
/* Remove the hash values without the
minimum support*/
H2.prune(min sup);
/*Find Fk, the set of frequent k-itemsets, where k≥2*/
for each (k:=2; Fk-1≠ϕ;k++) do begin
// Ck is the set of candidate k-items
Ck=ϕ;
/* Fk-1* Fk-1 is a natural join of
Fk-1 and Fk-1 on the first k-2 items
Hk is the hash table for k-itemsets*/
For each x∈{ Fk-1* Fk-1}do
If Hk.hassupport(x)
Then Ck= Ck ∪x;
End
/* Scan the transaction to count candidate k-itemsets and
generate Hk+1 */
for each transaction t∈Database do begin
for each k-itemset x in t do
if x∈ Ck
thenx.count++;
for each (k+1)-itemset y in t do
if ¬∃z|z=k-subset of y
^¬ Hk.hashsupport(z)
then Hk+1. add(y);
end
// Fk is the set of frequent k-itemsets
Fk=ϕ;
For each x∈ Ck do
if x.count/| Database|≥min sup
then Fk=Fk ∪x;
end
/*Remove the hash values without the minimum support
from Hk+1*/
Hk+1.prume(min sup);
end
answer=∪kFk;
```

Perfect hashing and pruning algorithm (PHP): In the sale transaction database domain, an example association rule may be that 90% of transactions that purchase bread and butter also purchase milk. The following is a formal statement of association rule mining for transaction database[11,12]: let $I=\{i_1,i_2,\ldots,i_m\}$ be a set of items and D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. Each transaction has a unique transaction identifier called its TID. We say that a transaction T contains X if $X \subseteq T$, where X is a set of some items in I. An association rule is an implication of the form $X \Rightarrow Y$, where X and Y are sets of some items in I such that they are disjoint. The rule $X \Rightarrow Y$ holds in the database D with confidence c, if c% of transactions in D that contain X also contain Y. the rule $X \Rightarrow Y$ has support s in the transaction set D, if s% of transactions in D contain $X \cup Y$. Given the database D, the problem of mining association rules involves the generation of all association rules that have support and confidence greater than or equal to the user-specified minimum support and minimum confidence.

The discovery of association rules for a given dataset D, involves two main steps[5]: The first step is to find each set of items, called as itemsets, such that the co-occurrence rate of these items is above the minimum support and these itemsets are called as large itemsets or frequent itemsets. The size of an itemset represents the number of items in that set. If the size of an itemset is equal to k, then this itemset is called as the k- itemset. The second step is to find association rules from the frequent itemsets that are generated in the first step. The second step of the generation of association rules is straightforward. In that step, for every frequent itemset f, all nonempty subsets of f are found. Then for every such subset a, a rule of the form $a \Rightarrow (f-a)$ is generated if the ratio of support (f-a) to support(a) is greater than or equal to the minimum confidence.

The proposed algorithm uses perfect hashing for the hash table generated at each pass and also reduces the size of DB by pruning the transactions. In first pass a hash table is created. Each distinct item in the DB is mapped to different location in the hash table. After the

first pass, the hash table contains the exact number of occurrences of each item in the DB. In the next pass the algorithm prunes the DB by discarding the transactions. Also, trims the items that are not frequent from the transactions. The Pseudo code is given below:

Algorithm 3.b : FPHP

Input: Database
Output: Frequent k-itemset
/* Database =set of transactions;
Items = set of items;
Transaction=<TID, {x∈Items}>;
F1 is a set of frequent l-itemsets*/
$F_1 = \phi$;
/* H_1 is the hash table for I-itemsets
Read the transactions and count the occurrences
of each items and generate H_1 */
For each transaction t∈Database do begin
For each item x in t do
H_1.add(x);
End;
//Form the set of frequent l-itemset
for each itemset y in H_1 do
if H_1.hassupport(y)
then $F_1 = F_1 \cup y$
end
/*Remove the hash values without the minimum support*/
H_1.prune(min sup);
D_1 =Database;
// D_k is the pruned database
/* find F_k the set of frequent k-itemsets,
where k≥2 and prune the database*/
k=2;
repeat
$D_k = \phi$;
$F_k = \phi$;
For each transaction t∈ D_{k-1} do begin
// w is k-1 subset of items in t
if ∀w|w∉ F_{k-1}
then skip t;
else
items=ϕ;
for each k-itemset y in t do
if ¬∃z|z=k-1 subset of y
^¬ H_{k-1}.hassupport(z)
then H_k.add(y);
items=items∪y;
end
$D_k = D_k \cup t$ //such that t contains
//items only in the set items
end
for each itemset y in H_k do
if H_k.hassupport(y)
then $F_k = F_k \cup y$
end
/*Remove the hash values without the minimum support from H_k */
H_k.prune(min sup);
K++;
Until $F_{k-1} = \phi$;
Answer = $\cup_k F_k$;

Efficient algorithm: We propose the following optimization algorithm. When all single items are counted, resulting in the set of all frequent items F1, we do not generate any candidate 2-itemset. Instead, we start scanning the DB and remove from each transaction all the items that are not frequent, on the fly. Then, for each trimmed transaction, we increase the support of all candidate 2-itemsets 20 contained in that transaction. If the candidate 2-itemset does not yet exists, we generate the CIS and initialize its support to 1. Only those candidate 2-itemsets that occur at least once in the DB are generated. For example, this technique was specially useful for the basket data set. In that data set there exist $({}^{8\,051}C_2)$ frequent items. Hence, Apriori would generate 8,051 = 32,405,275 candidate 2-itemsets. Using this technique, this number was drastically reduced to 1,708,203. Support lower bounding apart from the monotonicity property, it is possible to derive information on an item set, given the support of all of its subsets. The technique gives a lower bound on the support of an item set apart from the monotonicity property.

Proposition 4: Let X, Y, Z ⊆ I be itemsets. Support(X U Y U Z) ≥ support (X U Y) + support (X U Z) – support (X)

Proof:

Support (X U Y U Z)
= | cover (X U Y) ∩cover (X U Z)|
= | cover (X U Y) \ (cover (X U Y) \ cover (X U Z))|
≥ | cover (X U Y) \ (cover (X) \ cover (X U Z))|
≥ | cover (X U Y)| – | { cover (X) \ cover (X U Z))|
= | cover (X U Y)| – (| cover (X) | – | cover (X U Z)|)
= support (X U Y) + support (X U Z) – support (X)

The lower bound can be used in the following way. Every time a candidate k + 1-itemset is generated by joining two of its subsets of size k. Suppose the CIS X∈{i_1, i_2} is generated by joining X∈{i_1} and X∈{i_2}, we simply add up the supports of these two itemsets and subtract the support of X. If this lower bound is higher

than the MST, then we already know that it is frequent and hence, we can already generate CISs of larger sizes for which this lower bound can again be computed. We still need to count the exact supports of all these itemsets. Using the efficient support counting mechanism, this optimization could result in significant performance improvements. We can also exploit a special case of Proposition 4 more specifically.

Corollary 5. Let X, Y,Z ∈ I be itemsets. support (X∪Y) = support (X)⇒support (X∪Y∪Z) = support (X∪Z). This specific property was later exploited by Pasquier *et al.* in order to find a concise representation of all FISs[15]. Suppose, we have generated and counted the support of the FIS X∪{i}, its support is equal to the support of X. The supports of every superset X∪ {i}∪Y is equal to the support of X ∪Y and hence, we do not have to generate all such supersets, but, only keep the information that every superset of X∪{i} is also represented by a superset of X. This rules shows significant improvement in performance.

Combining passes: To combine as many iterations as possible in the Apriory algorithm, only few candidate patterns can be generated. We provide several upper bounds on the number of CISs that can be generated after certain iteration.

Dynamic itemset counting (DIC) algorithm: In Apriori, at each iteration the DCI in Algorithm 4, builds the set F_k of the frequent k-itemsets on the basis of the set of candidates C_k. However, DCI adopts a hybrid approach to determine the support of the candidates. During the first iterations, it exploits a counting-based technique and dataset pruning (line 7), i.e., items which will not appear in longer patterns are removed from the dataset. The pruned dataset fits into the main memory, DCI starts using an optimized intersection-based technique to access the in-core dataset (line 14 and 16).

Input D,σ

1: F_1=first_scan(D,s, and D_2);
//find frequent items and remove non-frequents from D
2: F_2=scond_scan(D_2,s, and D_3);
3 : k=2; //find frequent pairs from pruned dataset
4: // until pruned dataset is bigger than memory…
5: while(Dk+1.size()>memory_available())do
6 : k++; // keep the horizontzl format
7: F_k=horizontal_iter(D_k,s,k, and D_k+1);
8: end while // switch ot verticle format
10 : dense=D_k+1.is_dense();
//measure dataset density
11 : while(F_k=0)do
12 : k++;
13 : if (dense) then
14: F_k=verticle_iter_dense(D_1,s,k);
15 : else // optimize dense dataset
16 : F_k=vertical_iter_sparse(D_k,s,k, and D_k+1);
17 : end if // optimize sparse dataset
18 : end while

Algorithm 4 : DCI

The DIC algorithm, proposed by Brin *et al.* tries to reduce the number of passes over the database by dividing the DB into intervals of a specific size. DCI deals with dataset peculiarities by dynamically choosing between distinct optimization heuristics according to the dataset density (line 13). During its initial counting-based phase, DCI exploits an out-of-core, horizontal DB with variable length records. By exploiting effective pruning techniques inspired by the DHP algorithm, DCI trims the transaction. A pruned dataset D_{k+1} is thus written to disk at each iteration k and employed at the next iteration. Let m_k be the number of distinct items included in the pruned dataset D_k and n_k the number of remaining transactions. Due to dataset pruning $m_{k+1} \geq m_k$ and $n_{k+1} \geq n_k$ always hold. As soon as the pruned dataset becomes small enough to fit into the main memory, DCI adaptively changes its behavior, builds a vertical layout DB in-core and starts adopting an intersection based approach to determine frequent sets. Note, however, that DCI continues to have a level-wise behavior. At each iteration, DCI generates the candidate set C_k by finding all the pairs of (k-1) itemsets that are included in F_{k-1} and share a common (k - 2) prefix. Since F_{k-1} is lexicographically ordered, pairs occur in close positions and candidate generation is performed with high spatial and temporal locality. Only during the DCI counting-phase, C_k is further pruned by checking whether all the other subsets of a candidate are included in F_{k-1}. Conversely, during the intersection-based phase, since our intersection method is able to quickly determine the support of a candidate itemset, we found much more profitable to avoid this further check Fig. 3.

Dynamic type selectio: The optimization is concerned with the amount of memory used to represent itemsets and their counters. Since such structures are extensively accessed during the execution of the algorithm, is it profitable to have such data occupying as little memory as possible. This not only allows to reduce the spatial complexity of the algorithm, but also permits low level processor optimizations to be effective at run time. During the first scan of the dataset, global properties are collected like the total number of distinct frequent items or the maximum transaction size and the support of the most

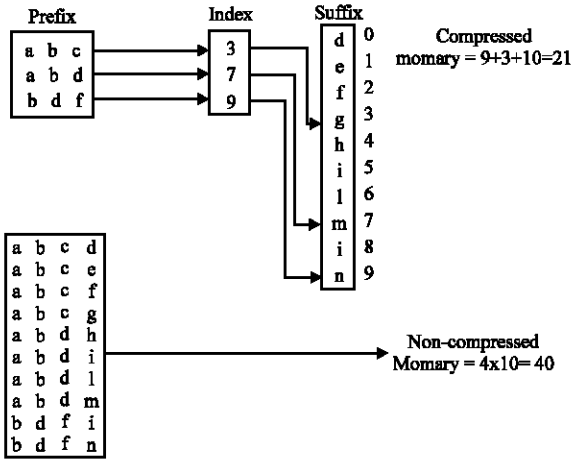

Fig. 3: Compressed data structure used for itemset

frequent item. Once this information is available, we remap items to ordered and contiguous integer identifiers. This allows us to decide the best data type to represent such identifiers and their counters.

Dense vs. sparse optimization: We adopt two different strategies according to whether a dataset is recognized as dense or not. We describe in more detail the heuristic used to determine a dataset density.

Sparse datasets: Sparse datasets originate bit-vectors containing long runs of 0's. To speedup computation, while we compute the intersection of the bit-vectors relative to the first two items c1 and c2 of a generic candidate itemset c = {c1, c2,. . . , ck} å C_k, we also identify and maintain information about the runs of 0's appearing in the resulting bit-vector stored in cache. The further intersections that are needed to determine the support of c will skip these runs of 0's, so that only vector segments which may contain 1's are actually intersected. The same information is reused many times. Note that such technique is a fast approximation a dataset projection, since the vertical dataset is dynamically reduced by removing transactions that do not support a given 2-item prefix. This cannot support larger itemsets sharing the same prefix. Moreover, sparse datasets does the possibility of furthr pruning vertical datasets as computation progresses. The benefits of pruning regard the reduction in the length of the bit-vectors and thus in the cost of intersections. Note that a transaction, can be removed from the vertical dataset when it does not contain any of the itemsets included in F_k. This check can simply be done by oring the intersection bit-vectors computed for all the frequent k-itemsets. However, we observed that dataset pruning is expensive, since vectors must be compacted at the level of single bits. Hence, DCI prunes the dataset only if turns out to be profitable, i.e. if we can obtain a large reduction in the vector length and the number of vectors to be compacted is small with respect to the cardinality of C_k.

Dense datasets: If the dataset is dense, we expect to deal with strong correlations among the most frequent items. This not only means that the bit-vectors associated with the most frequent items contain long runs of 1's, they turn out to be very similar. The heuristic technique adopted by DCI works as follows: Reorder the columns of the vertical dataset, in order to move identical segments of the bit-vectors associated with the most frequent items to the first consecutive positions; since, each candidate is likely to include several of the most frequent items, we avoid repeated intersections of identical segments. This technique may save a lot of work because

- The intersection of identical vector segments is done once,
- The identical segments are usually very large and
- Long candidate itemsets presumably contains several of these most frequent items.

CARMA algorithm (Continuous association rule mining algorithm): Proposed by Hidbert uses interval size to 1. CISs are generated on the fly from every transaction. By reading a transaction, it increments the support of all CISs. CARMA generates more CISs than DIC or Apriori. CARMA allows the user to change the MST during the execution of the algorithm. When the DB has been processed once, CARMA generates a superset of all FIS relative to some threshold. To determine the exact supports of all generated item sets, a second scan of the DB is required.

Sampling algorithm: Proposed by Toivonen, performs two scans through the DB by picking a random sample from the DB. It finds and verifies all frequent patterns in that sample. The sampling method does not produce all frequent patterns. Missing patterns can be found by generating all remaining frequent patterns, verifying their supports during a second pass through the DB. The probability of such a failure can be kept small by decreasing the MST.

Partitioning algorithm: Proposed by Savasere *et al.* uses DB is partitioned into several disjoint parts and the algorithm generates for every part all item sets that are relatively frequent within that part, using Algorithm 5. The parts of the DB are fit into main memory. The algorithm merges all relatively FISs of every part together. This results in a superset of all FISs. Supports of all itemsets

are computed during a second scan through the DB. Every part is read into main memory using the vertical database layout and the support of every itemset is computed by intersecting the covers of all items occurring in that itemset. The exact Partition algorithm is given in Algorithm 6.

Algorithm 5 Partition- Local Itemset Mining

Input: D,σ
Output: F (D,σ)
1: Compute F_1 and store with every frequent item its cover
2: k: =2
3: while $F_{k-1} \neq \{\}$ do
4: F_k: ={}
5: for all X,Y∈ F_{k-1}, X[I]=Y[I] for 1≤i≤k-2 and X[k-1]<Y[k-1] do
6: I={X[I],……,X[k-1],Y[k-1]}
7: if ∀ J⊂ I: J∈F_{k-1} then
8: I. cover: =X.cover ∩Y.cover
9: if | I.cover| ≥σ then
10: F_k: =F_k ∪ I
11: end if
12: end if
13: end for
14:k++
15: end while

Algorithm 6 Partition

Output: F (D,σ)
1: Partition D in D_1, ……, D_n
2: // Find all local frequent item sets
3: for 1 ≤ p ≤ n do
4: Compute C^p:=F(D_p, [σrel. |D_p|])
5: end for
6: // Merge all local frequent item sets
7: C_{global}: =$U_{1\leq p\leq n}$ C^p
8: //Compute actual support of all item sets
9: for 1≤p≤n do
10: Generate cover of each item in D_p
11: for all I∈C_{global} do
12: I. support : = I. support + | I [1]. Cover ∩ … ∩ I [| I |]. Cover |
13: end for
14: end for
15: // Extract all global frequent item sets
16: F: ={ I∈ C_{global} | I. support ≥ σ}

The algorithm is highly dependent on the heterogeneity of the DB and can generate too many local FISs, resulting significant decrease in performance.

Depth-first search (DFS) algorithms: It is possible to reduce this total size by generating collections of CISs in a DFS. The first algorithm proposed to generate all FISs in a DFS manner is the Eclat algorithm by Zaki[8,14]. Later, DFS algorithms have been proposed[1] of which the FP-growth algorithm by Han *et al.*[1] is the most well known. Given a TDB D and a MST σ, denote the set of all frequent k-itemsets with the same k - 1-prefix. All FISs can be computed during an initial scan over the DB, after which infrequent items will be ignored.

[Eclat]. Zaki's Eclat algorithm: It is based on a clever dataset partitioning method that relies on a lattice-theoretic approach for decomposing the SS. Each subproblem obtained from this SS decomposition is concerned with finding all the FISs that share a common prefix. On the basis of a given common prefix it is possible to determine a partition of the dataset, which will be composed of only those transactions, which constitute the support of the prefix. By recursively applying the Eclat's SS decomposition we can thus obtain subproblems which can entirely fit in main memory. Recently Zaki proposed an enhancement to Eclat (dEclat), whose main innovation regards a novel vertical data representation, which only keeps track of differences in the tids of a candidate pattern from its generating frequent patterns. Eclat uses the vertical DB layout and uses the intersection based approach to compute the support of an itemset.

Algorithm 7 Eclat

Input: D,σ,I⊆ I
Output: F[I] (D, σ)
1: F[I]:={}
2: for all i∈I occurring in D do
3: F[I]:=F[I]∪{I∪{I}}
4: // Create D^{ii}
5: D^i: ={}
6: for all j∈I occurring in D such that j>I do
7:C:=cover ({I})∩cover ({j})
8: if |C| ≥σ then
9: D^i : =D^i ∪ {(j, C)}
10: end if
11: end for
12: //Depth-first recursion
13: Compute F[I ∪ {I}] (D^i, σ)
14: F[I]:=F[I] ∪F[I ∪ { I }]
15: end for

Note that a CIS is now represented by each set I U {i, j} of which the support is computed at line 6 of the algorithm. Since the algorithm doesn't fully exploit the monotonicity property, but generates larger CISs based on two of its subsets. Eclat join step from Apriori. To

reorder all items in the DB in support of ascending order to reduce the number of CISs. Such reordering can be performed in step 10 and 11. Here, counting the supports of all itemsets is performed efficiently. The total size of all covers in main memory is much less. In the DFS, the covers of at most all k-itemsets with k-1-prefix are stored in main memory.

To compute the support of I, we simply need to subtract the size of the diffset from the support of its k - 1-prefix. The diffset of an itemset I U {i, j}, given the two diffsets of its subsets IU{i} and IU{j}, with i < j, is computed as follows: diffset(I U {i, j}) := diffset(IU {j}) \ diffset(I U {i}). This technique has shown best improvements, designated as dEclat. Using this DFS, it is possible to optimize Apriori algorithm. Another optimization proposed by Hipp et al. combines Apriori and Eclat into a single Hybrid. The algorithm starts generating FISs in a BFS manner using Apriori and switches after a certain iteration to a DFS strategy using Eclat. The exact switching point must be given by the user. Eclat generates every possible 2-itemset. If the TDB contains a large transactions of frequent items, such that Apriori needs to generate all its subsets of size 2, Eclat outperforms Apriori.

FP-tree (Frequent pattern) algorithm: FP growth builds in memory a compact representation of the dataset, where repeated patterns are represented only once. The DS used to store the dataset is called FP-tree. This algorithm recursively identifies tree paths which share a common prefix and projects the tree accordingly. These paths are intersected by considering the associated counters. FP-growth works very well for dense datasets, to construct very compressed FP-trees. FP-growth is more effective than Apriori algorithms, which need to store huge number of candidates for subset-counting. Unfortunately, FP-growth does not perform well on sparse datasets. It uses a combination of the vertical and horizontal database layout to store. It stores the actual transactions from the DB in a trie structure and every item has a linked list going through all transactions. This new DS is denoted by FP-tree[1].

Example 1: Assume we are given a TDB and a MST of 2. First, the supports of all items is computed, all infrequent items are removed from the DB. An example of preprocessed DB is shown below :

The FP-tree is shown in Fig. 4. The FP-tree, supports of all frequent items can be found in the header table. The FP-tree is just like the vertical and horizontal DB of TDB for the generation of FISs. It uses some additional steps to maintain the FP-tree structure during the recursion steps, while Eclat only needs to maintain the covers of all generated itemsets. First, FP-growth computes all frequent items for D_i at lines 6-10, which is of course different in

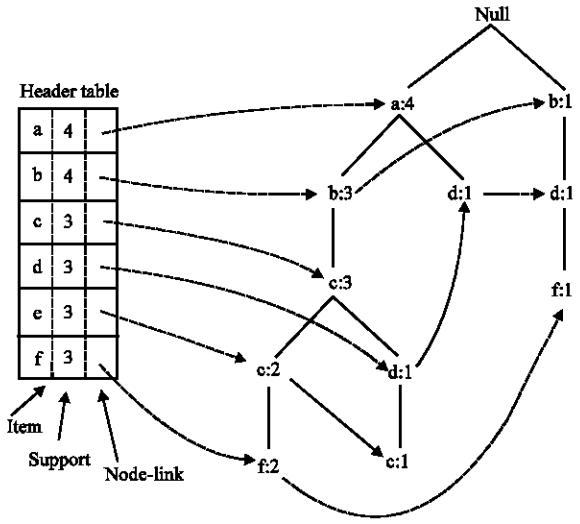

Fig. 4: An example of an FP-tree

every recursion step. This can be efficiently done by simply following the linked list starting from the entry of i in the header table. Then at every node in the FP-tree it follows its path up to the root node and increments the support of each item it passes by its count. Then, at lines 11-13, the FP-tree for the i-projected DB is built for those transactions in which i occurs, intersected with the set of all frequent items in D greater than i. These transactions can be efficiently found by following the nodelinks starting from the entry of item i in the header table and following the path from every such node up to the root of the FP-tree.

Algorithm 8 :Fp growth

Input: D, σ, $I \subseteq I$
Output: F [I] (D, σ)
1: F[I]: ={}
2: for all i$\in$I occurring in D do
3: F[I]:=F[I] $\cup$ {I$\cup${I}}
4: // Create $D^{i]}$
5: D^{i}:={}
6: H: ={}
7: for all j$\in$I occurring in D such that j>I do
8: if support (I $\cup$ {I, j}) $\geq\sigma$ then
9:H:=H$\cup${ j }
10: end if
11: end for
12: for all (tid, X)$\in$D with i$\in$X do
13: D^{ii}= D^{i} $\cup$ {(tid,X $\cap$H)}
14: end for
15: // Depth-first recursion
16: compute F [I $\cup$ { I }] (D^{i}, σ)
17: F [I]:= F [I]$\cup$ F [I $\cup$ { I }]
18: end for

If this node has count n, then the transaction is added n times. We can dynamically add a counter initialized to 1 for every item that occurs on each path in the FP-tree that is traversed. Suppose that the FP tree consists of a single path, then, we can stop the recursion. FP-growth is able to detect this one recursion step ahead of Eclat. At every recursion step, an item j occurring in D_i represents the itemset IU{i, j}. For every frequent item I occurring in D, the algorithm recursively finds all frequent 1-itemsets in the i-projected DB D_i.

The main advantage FP-growth, each linked list, representing the cover of that item, is stored in a compressed form. Unfortunately, to accomplish this gain, it needs to maintain a complex DS and perform a lot of dereferencing, while Eclat only has to perform simple and fast intersections. In Eclat, the cover of an item can be implemented using an array of transaction identifiers. In FP-growth, the cover of an item is compressed using the linked list starting from its node-link in the header table, but, every node in this linked list needs to store its label, a counter, a pointer to the next node, a pointer to its branches and a pointer to its parent. The size of an FP-tree is atmost 20%. Table 9 shows for all four used data sets the size of the total length of all arrays in Eclat (||D||), the total number of nodes in FP-growth (|FP-tree|) and the corresponding compression rate of the FP-tree. We show the size of the DSs in bytes and the corresponding compression of the FP-tree. FP-growth becomes an actual compression of the DB is the mushroom data set.

Opportunistic projection (OP) algorithm: Another projection-based algorithm, has been proposed. OP overcomes the limits of FP-growth, using a tree-based representation of projected transactions for dense datasets and a new array based representation of the same transactions for sparse datasets. OP Algorithm proposes heuristic method to opportunely switch between DFS, BFS of the frequent set tree. The choice of using a BFS depends on the size of the dataset to mine. For large datasets, we use BFS, for small DS, we use DFS approach.

The Breadth First Search (BFS): We create the upper portion of FIS Tree in three steps. First, Create Counting Vector(v). We attach counting vectors to all nodes at the current level k to accumulate local supports for the item of each sibling node that is after the node attached according to the imposed ordering. For example, possible items local to the Projected Transaction set(PTS) of the node (a,3) in figure1 are b, c, f, m and p, which are the

Table 8: An example preprocessed TDB

Tid	X
100	{a, b, c,, d, e, f}
200	{a, b, c, d, e}
300	{a, d}
400	{b, d, f}
500	{a, b, c, e, f}

Table 9: Memory usage of eclat versus FP-growth

Data set	||D||	|FP-tree|	Cover/FP-tree
T40I10D100K	3912459:15283K	3514917:68650K	89%:174%
Mushroom	174332:680K	16354:319K	9% :46%
BMS-Webview-	148209:578K	55410:1082K	37%:186%
1	399838:1561K	294311:5748K	73%:368%
Basket			

items of siblings that follow the node (a, 3). Therefore, a length 5 counting vector is attached to accumulate the supports for items b, c, f, m and p.

Second, Project And Count (t,D'). We project the transaction t along the path from the root to nodes at the current level k and accumulate counting vectors. If a transaction can be projected to a level k node and contribute to its counting vector, it may also be projected to level k+1, therefore record it in D'. Otherwise it can be removed from further consideration. This results in the reduction of the number of transactions level by level.

Third, Generation Children(v). We create children for each node at the current level k for its local frequent items whose element in the counting vector has a value over the support threshold. If the node v has no child, it is removed at that time and its parent will be deleted also if v is the only child of its parent and so on.

The BFS is a recursive procedure. We use the available free memory as parameter to control BFS process.

Algorithm 9 : OP

```
Opportune project (Database: D)
Begin
Create a null root for FIS tree T;
D' = Breadth first(T,D)
v = the null root of T;
Guided depth first(v,D');
End
Breadth first(FIST:T,CurrentLevel:L,Database:D)
Begin
For each node v at level L of T do
Create counting vector(v);
D'={}
For each transaction t in D do
Project and count(t,D');
For each node v at level L of T do
Generate children(v);
```

```
If D' cannot be represented by TVLA and TIF
Then BreadthFirst(T,L+1,D');
Else return(D');
End
GuidedDepthFirst(CurrentFIRSTNode:p,PTS:D)
Begin
ILp=TraverseAndCount(T,p);
Dp=Represent(D,p)
For each frequent entry e in ILp by particular ordering do
Begin
C=GetChild(p,e);
GuidedDepthFirst(c,Dp);
End
End
```

Algorithm 9. OpportunisticProjection

The guided DFS: Suppose the BFS procedure stops at level k, then, only paths with length of k are maintained on the FIS Tree whose lower portion will be generated by GuidedDepthFirst as follows.

First, TraverseAndCount(D,p) scans all transactions in D that support p, namely D_p and get ILp which either be local frequent items list created at that time if D is on the disk or in the form of Threaded varied length arrays (TVLA), or be represented in parent item lis(IL)t if D is in the form of TIF.

Second, Represent(D,p). If D is on the disk or in the form of TVLA, create a Threaded Transaction Forest(TTF) for D_p if the density of D_p is estimated to be greater than a given value, otherwise create a filtered TVLA. If D is in the form of TTF, represent D_p by a pseudo TTF and make a filtered copy if necessary.

Third, GetChild(p,e), for node p, either retrieve a child c that is labeled by the same items as that of e if the child is already created by BreadthFirst procedure, otherwise create the child at that time.

The GuidedDepthFirst procedure is more efficient than unguided one in that it avoids re-creating paths that end at upper portion created by the BreadthFirst procedure.

Experimental evaluation: We implemented the Apriori implementation using the online candidate 2-itemset generation optimization. We implemented the Eclat, Hybrid and FP-growth algorithms. All experiments reported in this paper were performed on a 2.8 GHZ HCL system with 1 GB main memory in C++. Figure 5 shows the performance. The first interesting behavior can be observed in the experiments for the basket data. Eclat performs much worse than all other algorithms. This behavior has been predicted, since the number of frequent items in the basket data set is very large, hence, a huge

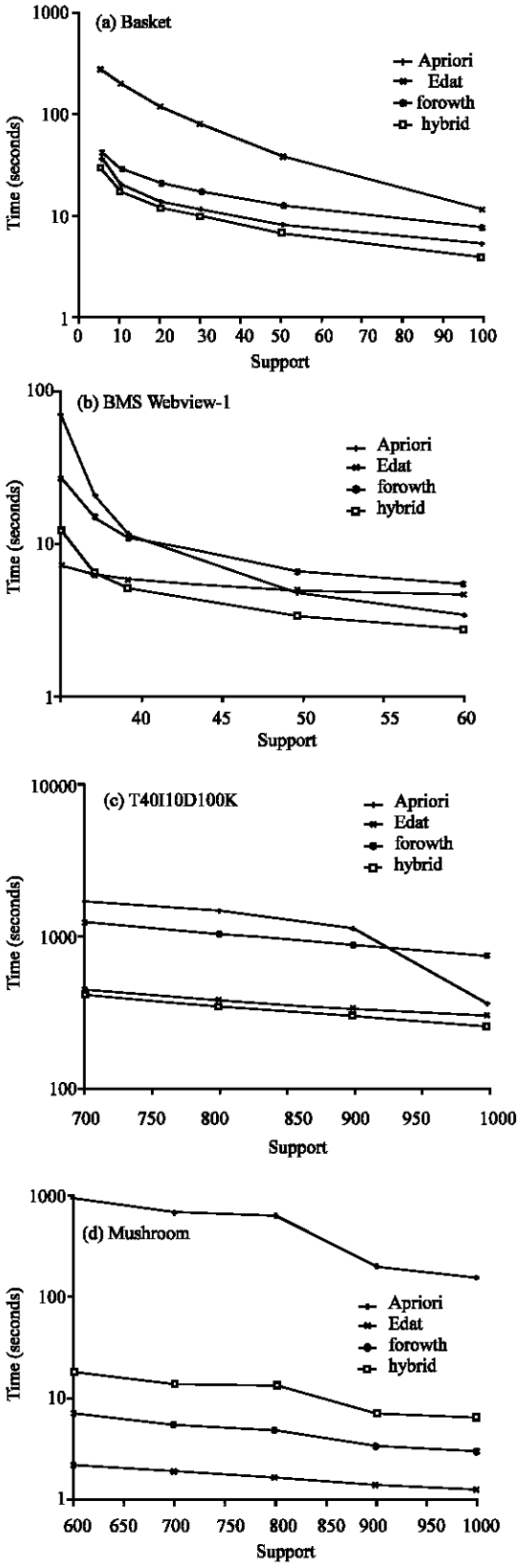

Fig. 5: FIS mining performance

amount of candidate 2-itemsets is generated. The other algorithms all use dynamic candidate generation of 2-itemsets resulting in much better performance results. The Hybrid algorithm performed best when Apriori was switched to Eclat after the second iteration.Apriori performs better than FP-growth for the basket data set. This result is due to the overhead created by the maintenance of the FP-tree structure. For all MSTs higher than 40, the differences in performance are negligible. For MSTs, Eclat clearly outperforms all other algorithms, because, for large transactions for which the subset generation procedure for counting the supports of all CISs consumes most of the time. For example, counting the supports of all 7-itemsets takes 10 seconds of which 9 seconds were used for these 34 transactions. Hybrid algorithm performed best in this case.

The performance of DCI were compared with those achieved under the same testing conditions by three of the most effcient FSC algorithms. We used FP-growth, OP and dEclat, in the implementation. Reports shows the total execution times obtained running FP-growth, dEclat, OP and DCI on various datasets as a function of the support thresholds. We can see that DCI performances are very similar to those obtained by OP. In some cases DCI slightly outperforms OP, in other cases the opposite holds. In all the tests conducted, DCI instead outperforms both dEclat and FP-growth. Finally, due to its adaptiveness, DCI can effectively mine huge datasets. The performance differences of Eclat and FP-growth are negligible and are again mainly due to the differences in initialization and destruction. Because of the small size of the database, both run extremely fast. Hybrid algorithm doesn't perform well. In this situation.

CONCLUSION

Lot of people have compared several algorithms to solve the FIS mining problem as efficiently as possible. We experienced that different implementations of algorithms yield different results. We observe that, different compilers and different machines sometimes showed different behavior for the same algorithms. Also different kind of data sets on which the algorithms were tested showed remarkable differences in the performance of such algorithms. An interesting example of this is presented by Zheng *et al.*[16] in their article on the real world performance of association rule algorithms[16] in which five well-known association rule mining algorithms are compared. In this paper, we have presented an analysis of a lot of algorithms which made a significant contribution to improve the efficiency of FIS. We have shown that as long as the DB fits in main memory, the Hybrid algorithm, as a combination of an optimized version of Apriori and Eclat is far the most efficient algorithm. For dense DBs, the Eclat algorithm is better. If the DB does not fit into memory, the best algorithm depends on the density of the DB. For sparse DBs the Hybrid algorithm seems the best choice if the switch from Apriori to Eclat is made as soon as the DB fits into main memory. For dense DBs, we envisage that the partition algorithm, using Eclat to compute all local FISs, performs best.

REFERENCES

1. Han, J., J. Pei, Y. Yin and R. Mao, 2003. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery.
2. Gunopulos, D., R. Khardon, H. Mannila and H. Toivone, 1997. Data Mining, Hypergraph Transversals and Machine Learning. In Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp: 209-216. ACM Press, 1997.
3. Mannila, H., 2002. Global and Local Methods in Data Mining: Basic Techniques and Open Problems. In Widmayer, P., F.T. Ruiz, R. Morales, M. Hennessy, S. Eidenbenz and R. Conejo, Eds., Proceedings of the 29th International Colloquium on Automata, Languages and Programming, Volume 2380 of Lecture Notes in Computer Science, pp: 57-68. Springer, 2002.
4. Mannila, V. and H. Toivonen, 1997. Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery, 1: 241-258. November 1997.
5. Bykowski, A. and C. Rigotti, 2001. A Condensed Representation to Find Frequent Patterns. In Proceedings of the Twentieth 20th ACM SIGACT-SIGMODSIGART Symposium on Principles of Database Systems, pp: 267-273. ACM Press.
6. Mannila, H., 1997. Inductive Databases and Condensed Representations for Data Mining. In Maluszynski, J. Ed., Proceedings of the 1997 International Symposium on Logic Programming, pp: 21-30. MIT Press, 1997.
7. Pasquier, N., Y. Bastide, R. Taouil and L. Lakhal, 1999. Discovering Frequent Closed Item Sets for Association Rules. In Beeri C. and P. Buneman, Eds., Proceedings of the 7th International Conference on Database Theory, Volume 1540 of Lecture Notes in Computer Science, pp: 398-416. Springer, 1999.
8. Zaki, M.J. , 2000. Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering, 12: 372-390.

9. Orlando, S., P. Palmerini, R. Perego and F. Silvestri, 2002. Adaptive and Resource-Aware Mining of Frequent Sets. In Kumar, V., S. Tsumoto, P.S.Yu and N. Zhong, Eds., Proceedings of the 2002 IEEE International Conference on Data Mining. IEEE Computer Society.
10. Agrawal, R., T. Imielinski and A.N. Swami, 1993. Mining Association Rules Between Sets of Items in Large Databases. In P. Buneman and S. Jajodia, Eds., Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, of SIGMOD Record, pp: 207-216. ACM Press, http://www.almaden.
11. Agrawal, R. and R. Srikant, 1994. Fast Algorithms for Mining Association rules. In Bocca, J.B. , M. Jarke and C. Zaniolo, Eds., Proceedings 20th International Conference on Very Large Data Bases, pp: 487-499. Morgan Kaufmann, 1994.
12. Borgelt, C. and R. Kruse, 2002. Induction of Association Rules: Apriori Implementation. In¨Ardle W.H and B.R¨onz, Eds., Proceedings of the 15th Conference on Computational Statistics, pp: 395-400, http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html, 2002. Physica-Verlag.
13. Brin, S., R. Motwani, J.D. Ullman and S. Tsur, 1997. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Vol. 26(2) of SIGMOD Record, pp: 255-264. ACM Press.
14. Zaki, M.J., 2001. Fast vertical mining using di_sets. Technical Report 01-1, Rensselaer Polytechnic Institute, Troy, New York.
15. Mannila, H., H. Toivonen and A.I. Verkamo, 1994. Efficient Algorithms for Discovering Association Rules. In Fayyad, U.M. and R. Uthurusamy, Eds., Proceedings of the AAAI Workshop on Knowledge Discovery in Databases, pp: 181-192. AAAI Press, 1994.
16. Zheng, Z., R. Kohavi and L. Mason, 2001. Real World Performance of Association Rule Algorithms. In Provost, F. and R. Srikant, Eds., Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp: 401-406. ACM Press, 2001.