DATA MINING AND DATA WAREHOUSING

18CS6H1

Module-03

Association Analysis

Association Analysis :-

→ Many business enterprises accumulate large quantities of data from their day-to-day operations.

→ Table(1) illustrates an example of such data, commonly known as market basket transactions.

(*) Each row in this table corresponds to a transaction, which contains a unique identifier labeleled TID.

(*) Set of items bought by a given customer.

→ Retailers are interested in analyzing the data to learn about the purch-asing behaviour of their customer.

TID	Items
1	{Bread, Milk}
2	{Bread, Diapers, Beer, Eggs}
3	{Milk, Diapers, Beer, Cola}
4	{Bread, Milk, Diapers, Beer}
5	{Bread, Milk, Diapers, Cola}

Table(1) : An example of market basket trasactions.

→ Association analysis is useful for discovering interesting relationships hidden in large data sets.

→ The uncovered relationships can be represented in the form of associati-on rules or sets of frequent items.

Example:- the following rule can be extracted from the data set shown in table(1):

$$\{Diapers\} \longrightarrow \{Beer\}$$

→ The rule suggests that a strong relationship exists between the sale of diapers and beer because many customers who buy diapers also buy beer.

→ Retailers can use this type of rules to help them identify new opportunities for cross-selling their products to the customers.

PROBLEM DEFINITION :-

The terminologies used in Association analysis :

① Binary Representation :-

→ Market basket data can be represented in a binary format as shown in table (2).

→ Each row corresponds to a transaction and each column corresponds to an item.

→ An item can be treated as a binary variable whose value is one if the item is present and zero otherwise.

→ An item is an asymmetric binary variable.

TID	Bread	Milk	Diapers	Beer	Eggs	Cola
1	1	1	0	0	0	0
2	1	0	1	1	1	0
3	0	1	1	1	0	1
4	1	1	1	1	0	0
5	1	1	1	0	0	1

Table (2): A binary 0/1 representation of market basket data.

② Itemset and Support Count :-

→ Let $I = \{i_1, i_2, \ldots, i_d\}$ be the set of all items in a market basket data and $T = \{t_1, t_2, \ldots, t_N\}$ be the set of all transactions.

→ Each transaction t_i contains a subset of items choosen from I.

→ In association analysis, a collection of zero or more items is termed an itemset.

→ If an itemset contains k items, it is called a k-itemset.

Ex:- {Beer, Diapers, Milk} is an example of 3-itemset.

→ The null (or empty) set is an itemset that does not contain any items.

→ The transaction width is defined as the number of items present in a transaction. A transaction t_j is said to contain an itemset X if X is a subset of t_j.

→ Support count refers to the number of transactions that contain a particular itemset.

Mathematically, support count $\sigma(X)$ for an itemset X can be stated as follows: $\sigma(X) = |\{t_i | X \subseteq t_i, t_i \in T\}|$,

where the symbol $|\cdot|$ denote the no of elements in a set.

③ Association Rule :-

→ An association rule is an implication expression of the form $X \rightarrow Y$, where X and Y are disjoint itemsets, i.e., $X \cap Y = \phi$.

→ The strength of an association rule can be measured in terms of its support and confidence.

→ Support determines how often a rule is applicable to a given data set, while confidence determines how frequently items in Y appear in transactions that contain X.

→ The formal definitions of these metrics are

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

$$\text{confidence}, \; c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$$

Ex :- Consider rule $\{\text{Milk, Diapers}\} \rightarrow \{\text{Beers}\}$

$$s = \frac{\sigma(\text{Milk, Diapers, Beers})}{N} = \frac{2}{5} = 0.4$$

$$c = \frac{\sigma(\text{Milk, Diapers, Beers})}{\sigma(\text{Milk, diapers})} = \frac{2}{3} = 0.67$$

Use of support and Confidence :-

→ Support is an important measure because a rule that has very low support may occur simply by chance.

→ Support often used to eliminate uninteresting rules.

→ Support can be used to exploit the efficient discovery of association rules.

→ Confidence, on the otherhand, measures the reliability of the inference made by a rule.

→ For a given rule $X \rightarrow Y$, the higher the confidence, the more likely it is for Y to be present in transactions that contain X.

→ Confidence also provides an estimate of the conditional property of Y given X.

Formulation of Association Rule Mining Problem :-

The association rule mining problem can be formally stated as follows :

Definition :- Given a set of transactions T, find all the rules having
(i) support ≥ minsup threshold
(ii) confidence ≥ minconf threshold
where minsup and minconf are the corresponding support and confidence thresholds.

→ A brute-force approach for mining association rule is to compute the support and confidence for every possible rule.
→ This approach is prohibitively expensive because there are exponen-tially many rules that can be extracted from a data set.
→ The total no of possible rules extracted from a data set that contains d items is,

$$R = 3^d - 2^{d+1} + 1$$

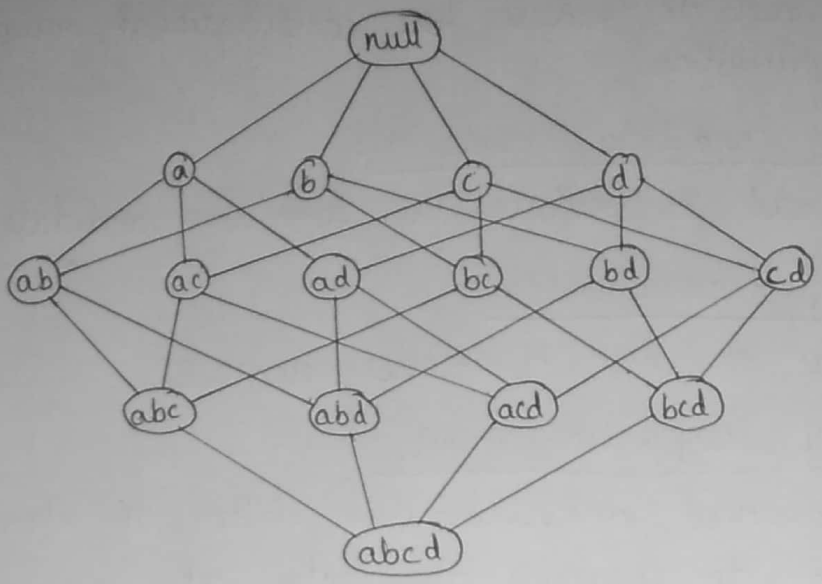
A common strategy adopted by many association rule mining algorithms is to decompose the problem into two major subtasks :

① **Frequent Itemset Generation**, whose objective is to find all the item-sets that satisfy the minsup threshold. These itemsets are called frequent itemsets.

② **Rule Generation**, whose objective is to extract all the high-confi-dence rules from the frequent itemsets found in the previous step. These rules are called strong rules.

FREQUENT ITEMSET GENERATION

→ A lattice structure can be used to enumerate the list of all possible itemsets.
→ Figure (1) shows an itemset lattice for $I = \{a, b, c, d, e\}$.
→ In general, a data set that contains k items can potentially generate up $2^k - 1$ frequent itemsets, excluding the null set.
→ Because k can be very large in many practical applications, the search space of itemsets that need to be explored is exponenti-ally large.



Fig(1): An itemset lattice

→ A Brute-force approach for finding frequent itemsets is to determine the support count for every candidate itemset in the lattice structure.

Ex:- Support for {Bread, Milk} is incremented three times because the itemset is contained in transactions 1, 4 and 5.

Brute force approach :-

(*) Each itemset in the lattice is a candidate frequent itemset.

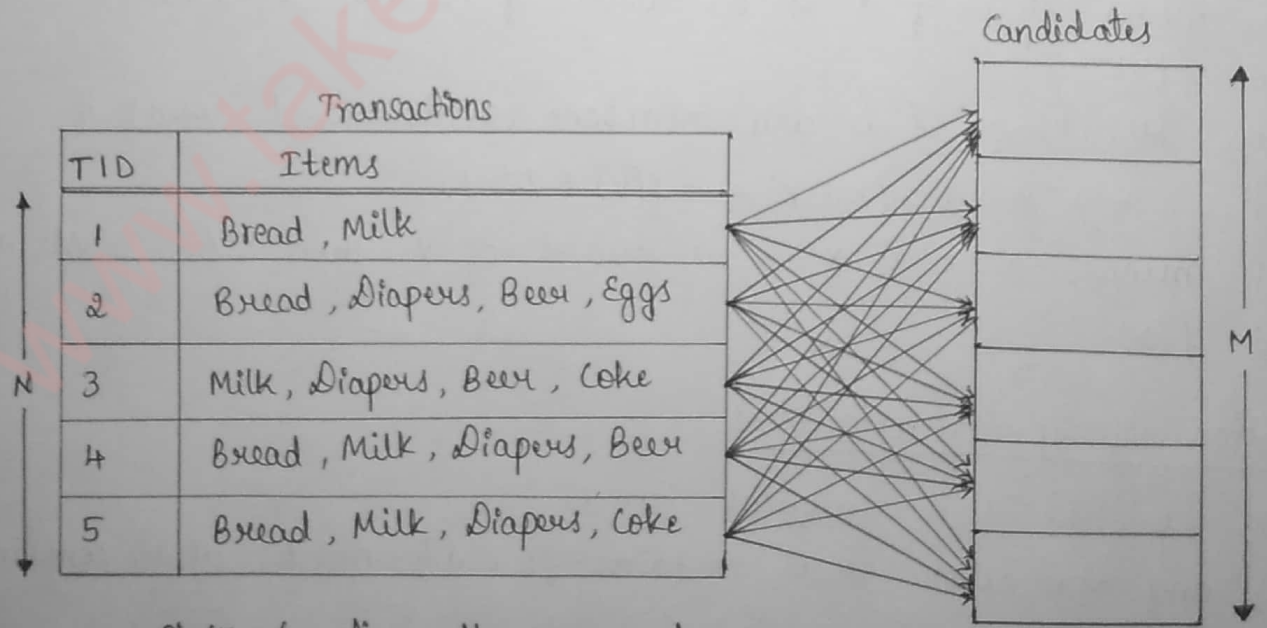
(*) Count the support for each candidate by scanning the database.

(*) Match each transaction against every candidate.

(*) Complexity ≈ $O(NM\omega)$ ⇒ Expensive since $M = 2^k - 1$.

$N \rightarrow$ no of transactions.

$\omega \rightarrow$ max transaction width.



Fig(2): Counting the support of candidate itemsets.

There are several ways to reduce the computational complexity of frequent itemset generation.

① Reduce the number of candidate itemsets (M):

Apriori principle is used to reduce the ~~cand~~ ~~com~~ candidate itemsets.

② Reduce the number of transactions (N):

Reduce size of N as the size of itemset increases.

③ Reduce the number of comparisons (NM):

By using more advanced datastructures, either to store the candidate itemset or to compress the data set.

The Apriori Principle :-

Theorem : (Apriori principle): If an itemset is frequent, then all of its subsets must also be frequent.

→ Apriori principle holds due to the anti-monotone property of the support measure.

Definition: (Monotonicity Property):

Let I be a set of items, and $J = 2^I$ be the power set of I. A measure f is monotone (or upward closed) if

$$\forall X, Y \in J : (X \subseteq Y) \longrightarrow f(X) \leq f(Y),$$

which means that if X is a subset of Y, then f(X) must not exceed f(Y).

On the other hand, f is anti-monotone (or downward closed) if

$$\forall X, Y \in J : (X \subseteq Y) \longrightarrow f(Y) \leq f(X),$$

which means that if X is a subset of Y, then f(Y) must not exceed f(X).

Illustration of Apriori principle :

→ Suppose {b, c, d} is a frequent itemset.

→ Clearly any transaction that contains {b, c, d} must also contain its subsets {b, c}, {b, d}, {c, d}, {b}, {c}, {d}.

→ As a result, if {b, c, d} is frequent, then all subsets of {b, c, d} [shaded itemset in fig(3)] must also be frequent.

→ Conversely, if {a} is infrequent, then all supersets of {a} are infrequent.

→ As illustrated in fig (4), the entire subgraph containing the supersets of {a} can be pruned immediately once {a} is found to be infrequent.

→ This strategy of trimming the exponential search space based on the support measure is known as support based pruning.

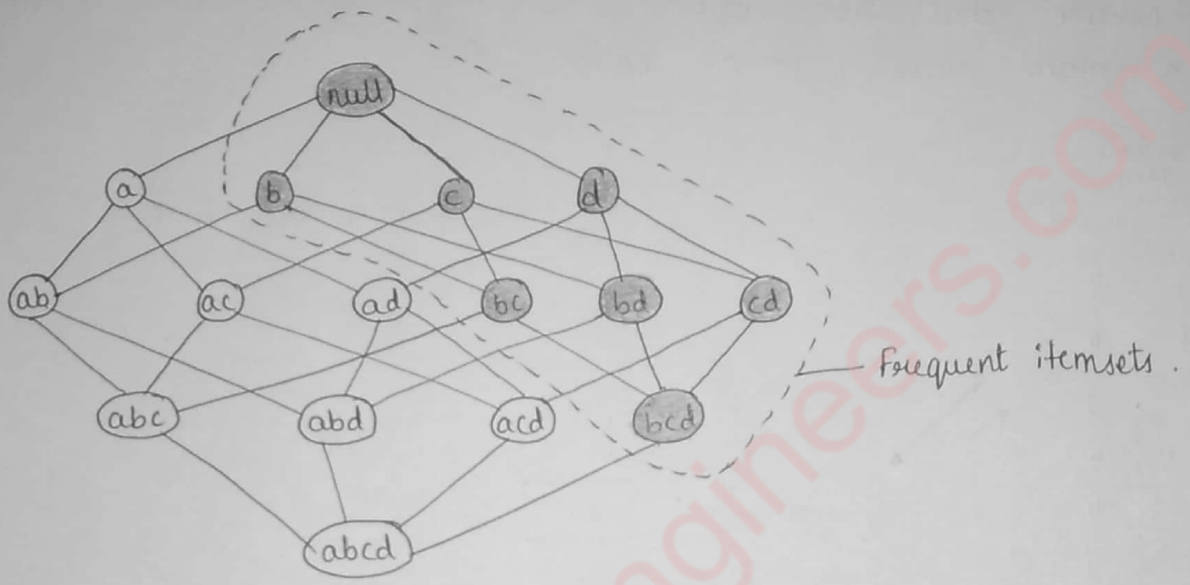


Fig (3): An illustration of Apriori principle. If {b, c, d} is frequent, then all subsets of the itemset are frequent.

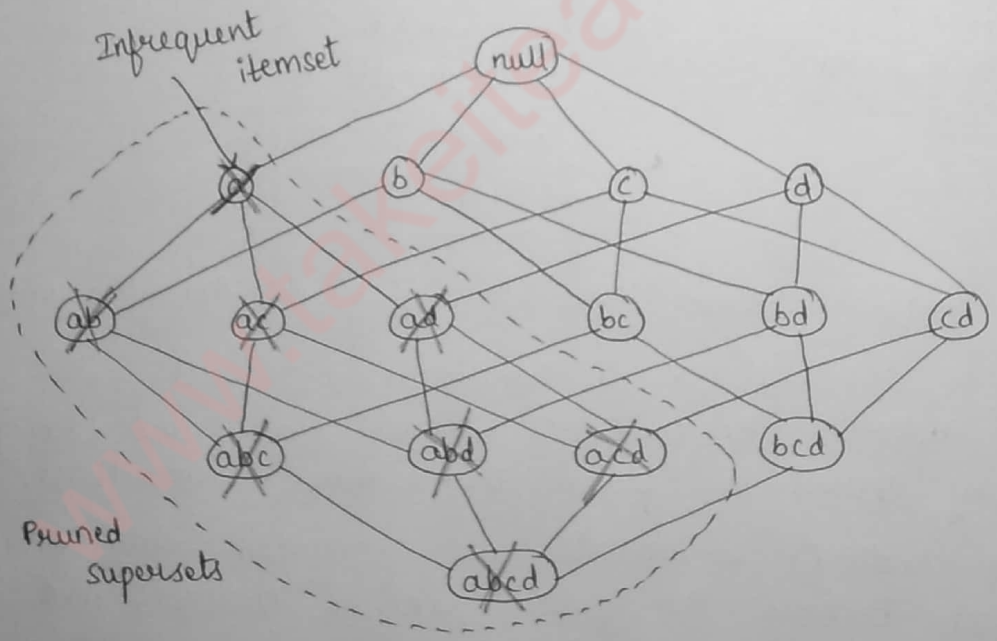


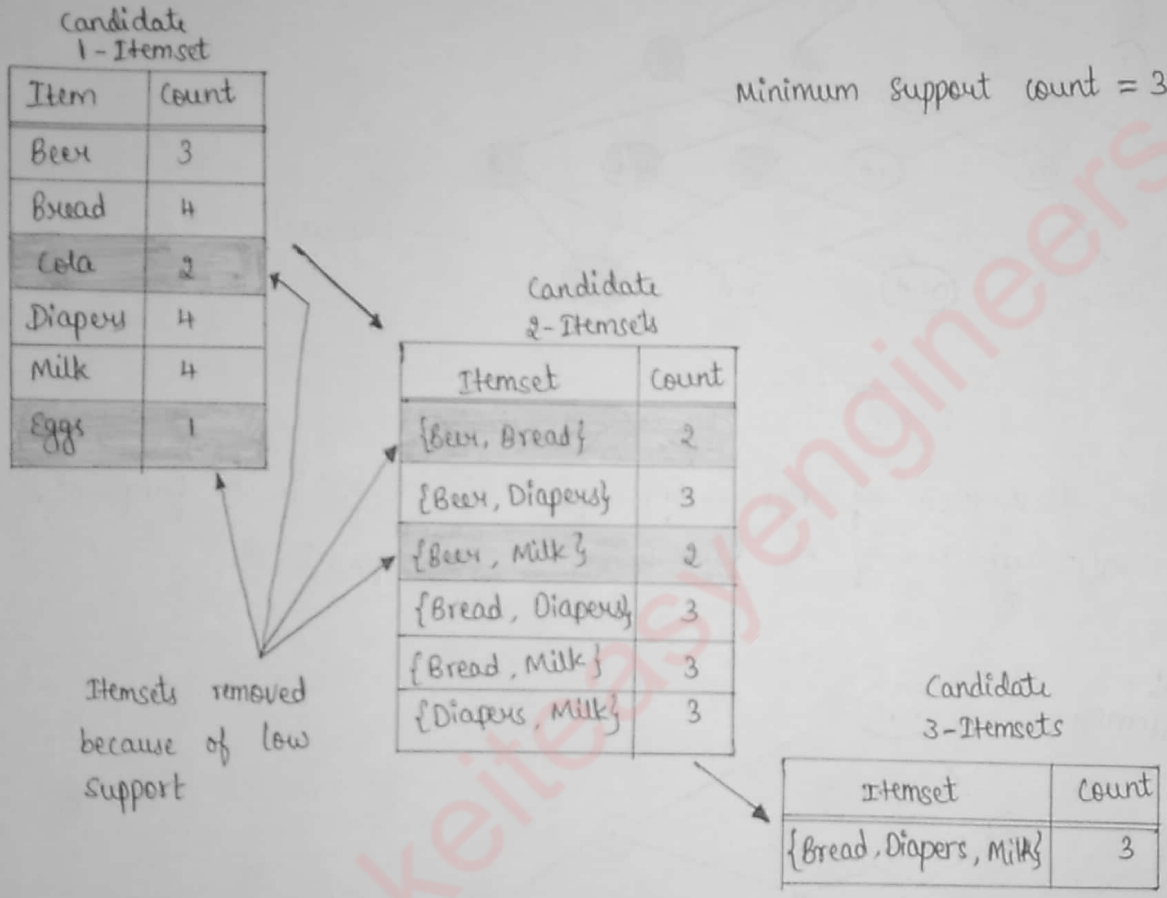
Fig (4): An illustration of support based pruning. If {a} is infrequent, then all suspects of {a} are infrequent.

Frequent Itemset Generation in the Apriori Algorithm

→ Apriori is the first association rule mining algorithm that pioneered the use of support-based pruning to systematically control the exponential growth of candidate itemsets.

→ Fig(5) provides a high-level illustration of the frequent itemset generation part of the Apriori algorithm for the transactions shown in table(1).

→ We assume that the support threshold is 60%, which is equivalent to a minimum support count equal to 3.



Fig(5): Illustration of frequent itemset generation using the Apriori algorithm.

→ Initially, every item is considered as a candidate 1-itemset.

→ After counting their supports, the candidate itemsets {cola} and {Eggs} are discarded because they appear in fewer than three transactions.

→ In the next iteration, candidate 2-itemsets are generated using only the frequent 1-itemsets because the Apriori principle ensures that all supersets of the infrequent 1-itemsets must be infrequent.

→ Because there are only four frequent 1-itemsets, the number of candidate 2-itemsets generated by the algorithm is ~~$(4/2) = 6$~~ $6 \cdot \binom{4}{2} = 6$

→ Two of these six candidates, {Beer, Bread} & {Beer, Milk}, are subsequently found to be infrequent after computing their support values.

→ The remaining four candidates are frequent, & thus will be used to generate candidate 3-itemsets.

→ Without support-based pruning, there are $\binom{6}{3} = 20$ candidates 3-itemset that can be formed using the six items given in this example.

→ With the Apriori principle, we only need to keep candidate 3-itemsets whose subsets are frequent.

→ The only candidate that has this property is {Bread, Diapers, Milk}.

→ The effectiveness of the Apriori pruning strategy can be shown by counting the number of candidate itemsets generated.

(*) A brute-force approach of enumerating all itemsets [up to size 3] as candidates will produce

$$\binom{6}{1} + \binom{6}{2} + \binom{6}{3} = 6 + 15 - 20 = 41 \quad \text{candidates.}$$

(*) Apriori principle produces

$$\binom{6}{1} + \binom{4}{2} + 1 = 6 + 6 + 1 = 13 \text{ candidates, which represents 68\% reduction in the no of candidate itemsets even in this example.}$$

→ The pseudocode for the frequent itemset generation part of Apriori algorithm is shown in Algorithm (1).

→ Let C_k denote the set of candidate k-itemsets & F_k denote the set of frequent k-itemsets:

Algorithm 1 : Frequent itemset generation of the Apriori algorithm.

1: $k = 1$.
2: $F_k = \{i \mid i \in I \wedge \sigma(\{i\}) \geq N \times \text{minsup}\}$. {Find all frequent 1-itemset}
3: repeat
4: $k = k + 1$.
5: $C_k = \text{apriori-gen}(F_{k-1})$. {Generate candidate itemsets}
6: for each transaction $t \in T$ do
7: $C_t = \text{subset}(C_k, t)$. {Identify all candidates that belongs to t}
8: for each candidate itemset $c \in C_t$ do
9: $\sigma(c) = \sigma(c) + 1$. {Increment support count}
10: end for
11: end for
12: $F_k = \{c \mid c \in C_k \wedge \sigma(c) \geq N \times \text{minsup}\}$. {Extract the frequent k-itemsets}
13: until $F_k = \phi$
14: Result = $\cup F_k$.

→ The algorithm initially makes a single pass over the data set to determine the support of each item. Upon completion of this step, the set of all frequent 1-itemsets, F_1, will be known [step 1 and 2].

→ Next, the algorithm will iteratively generate new candidate k-itemsets using the frequent (k-1)-itemsets found in the previous iteration [step 5]. Candidate generation is implemented using a function called apriorigen.

→ To count the support of the candidates, the algorithm needs to make an additional pass over the data set [steps 6-10]. The subset function is used to determine all the candidate itemsets in C_k that are contained in each transaction t.

→ After counting their supports, the algorithm eliminates all candidate itemsets whose support counts are less than minsup [step 12].

→ The algorithm terminates when there are no new frequent itemsets generated, i.e., $F_k = \phi$ [step 13].

*] The frequent itemset generation part of the Apriori algorithm has two important characteristics.

→ It is a level-wise algorithm, i.e., it traverses the itemset lattice one level at a time, from frequent 1-itemsets to the maximum size of frequent itemsets.

→ It employs a generate-and-test strategy for finding frequent itemsets. At each iteration, new candidate itemsets are generated from the frequent itemsets found in the previous ~~items~~ iteration.

(*) The support for each candidate is then counted and tested against the minsup threshold. The total no of iterations needed by the algorithm is $k_{max} + 1$, where k_{max} is the maximum size of the frequent itemsets.

Candidate Generation and Pruning :-

The apriori-gn function shown in step 5 of algorithm 1 generates candidate itemsets by performing the following two operations:

① Candidate Generation: This operation generates new candidate k-itemsets based on the frequent (k-1)-itemsets found in the previous iteration.

② Candidate Pruning: This operation eliminates some of the candidate k-itemsets using the support-based pruning strategy.

→ To illustrate the candidate pruning operation, consider a candidate k-itemset, $X = \{i_1, i_2, \ldots, i_k\}$. The algorithm must determine whether all of its proper subsets, $X - \{i_j\}$ $(\forall j = 1, 2, \ldots, k)$, are frequent.

→ If one of them is infrequent, then X is immediately pruned. The complexity of this operation is $O(k)$ for each candidate k-itemset.

Requirements for an effective candidate generation procedure :-

① It should avoid generating too many unnecessary candidates. A candidate itemset is unnecessary if at least one of its subsets is infrequent. Such a candidate is guaranteed to be infrequent according to the antimonotone property of support.

② It must ensure that the candidate set is complete, i.e., no frequent itemsets are left out by the candidate generation procedure. To ensure completeness, the set of candidate itemsets must subsume the set of all frequent itemsets, i.e., $\forall k : F_k \subseteq C_k$.

③ It should not generate the same candidate itemset more than once. For ex, the candidate itemset $\{a, b, c, d\}$ can be generated in many ways - by merging $\{a, b, c\}$ with $\{d\}$, $\{b, d\}$ with $\{a, c\}$, $\{c\}$ with $\{a, b, d\}$, etc. Generation of duplicate candidates leads to wasted computations & thus should be avoided for efficiency reasons.

Several Candidate generation procedures :-

① Brute-Force Method

② $F_{k-1} \times F_1$ Method

③ $F_{k-1} \times F_{k-1}$ Method

① Brute-Force Method :-

→ The Brute-Force method considers every k-itemset as a potential candidate and then applies the candidate pruning step to remove any unnecessary candidates.

→ The number of candidate item-sets generated at level k is equal to $\binom{d}{k}$, where d is the total no of items.

→ Overall complexity of this method is $O\left(\sum_{k=1}^{d} k \times \binom{d}{k}\right) = O\left(d \cdot 2^{d-1}\right)$.

② ~~$F_{k-1} \times F_1$ Method :-~~

Items

Item
Beer
Bread
Cola
Diapers
Milk
Eggs

→

Candidate Generation

Itemset
{Beer, Bread, Cola}
{Beer, Bread, Diapers}
{Beer, Bread, Milk}
{Beer, Bread, Eggs}
{Beer, Cola, Diapers}
{Beer, Cola, Milk}
{Beer, Cola, Eggs}
{Beer, Diapers, Milk}
{Beer, Diapers, Eggs}
{Beer, Milk, Eggs}
{Bread, Cola, Diapers}
{Bread, Cola, Milk}
{Bread, Cola, Eggs}
{Bread, Diapers, Milk}
{Bread, Diapers, Eggs}
{Cola, Bread, Milk, Eggs}
{Cola, Diapers, Milk}
{Cola, Diapers, Eggs}
{Cola, Milk, Eggs}
{Diapers, Milk, Eggs}

→

Candidate Pruning

Itemset
{Bread, Diapers, Milk}

Fig(6): A brute-force method for generating candidate 3-itemsets.

② $F_{k-1} \times F_1$ Method :-

→ An alternative method for candidate generation is to extend each frequent $(k-1)$ itemset with other frequent items.

→ Fig(7) illustrates how a frequent 2-itemsets such as {Beer, Diapers} can be argumented with a frequent item such as Bread to produce a candidate 3-itemset {Beer, Diapers, Bread}.

Frequent 2-Itemset

Itemset
{Beer, Diapers}
{Bread, Diapers}
{Bread, Milk}
{Diapers, Milk}

Frequent 1-Itemset

Item
Beer
Bread
Diapers
Milk

Candidate Generation

Itemset
{Beer, Diapers, Bread}
{Beer, Diapers, Milk}
{Bread, Diapers, Milk}
{Bread, Milk, Beer}

→

Candidate Pruning

Itemset
{Bread, Diapers, Milk}

Fig(7): Generating and pruning candidate k-itemsets by merging a frequent $(k-1)$-itemset with a frequent item.

→ This method will produce $O(|F_{k-1}| \times |F_1|)$ candidate k-itemsets, where $|F_j|$ is the no of frequent j-itemsets.

→ The overall complexity is $O\left(\sum_k k|F_{k-1}||F_1|\right)$

(3) $F_{k-1} \times F_{k-1}$ Method :-

→ The candidate generation procedure in the apriori-gen function merges a pair of frequent (k-1)-itemsets only if their first k-2 items are identical.

→ Let $A = \{a_1, a_2, \ldots, a_{k-1}\}$ and $B = \{b_1, b_2, \ldots, b_{k-1}\}$ be a pair of frequent (k-1)-itemsets.

→ A and B are merged if they satisfy the following conditions:

$a_i = b_i$ (for $i = 1, 2, \ldots, k-2$) and $a_{k-1} \neq b_{k-1}$.

→ In fig (8), the frequent itemsets {Bread, Diapers} & {Bread, Milk} are merged to form a candidate 3-itemset {Bread, Diapers, Milk}.

→ The algorithm does not have to merge {Beer, Diapers} with {Diapers, Milk} is a viable candidate because the first item in both itemsets is different.

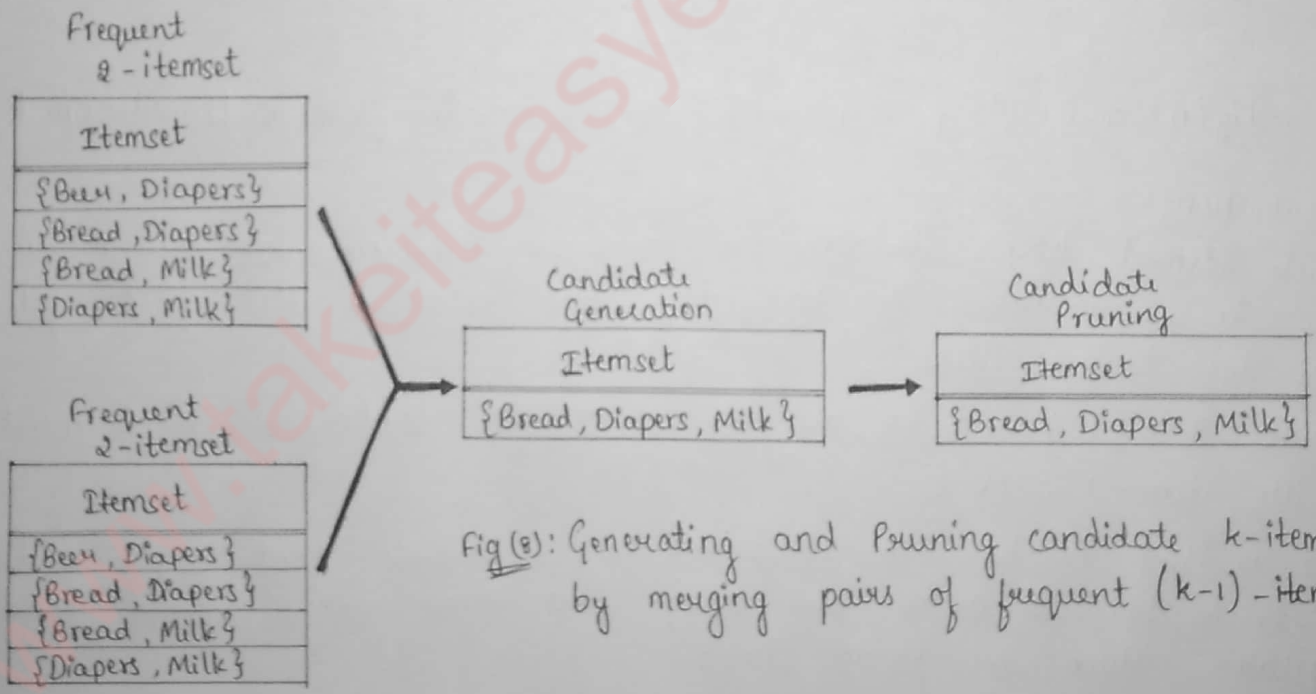


Fig (8): Generating and Pruning candidate k-itemsets by merging pairs of frequent (k-1)-itemsets.

Support Counting :-

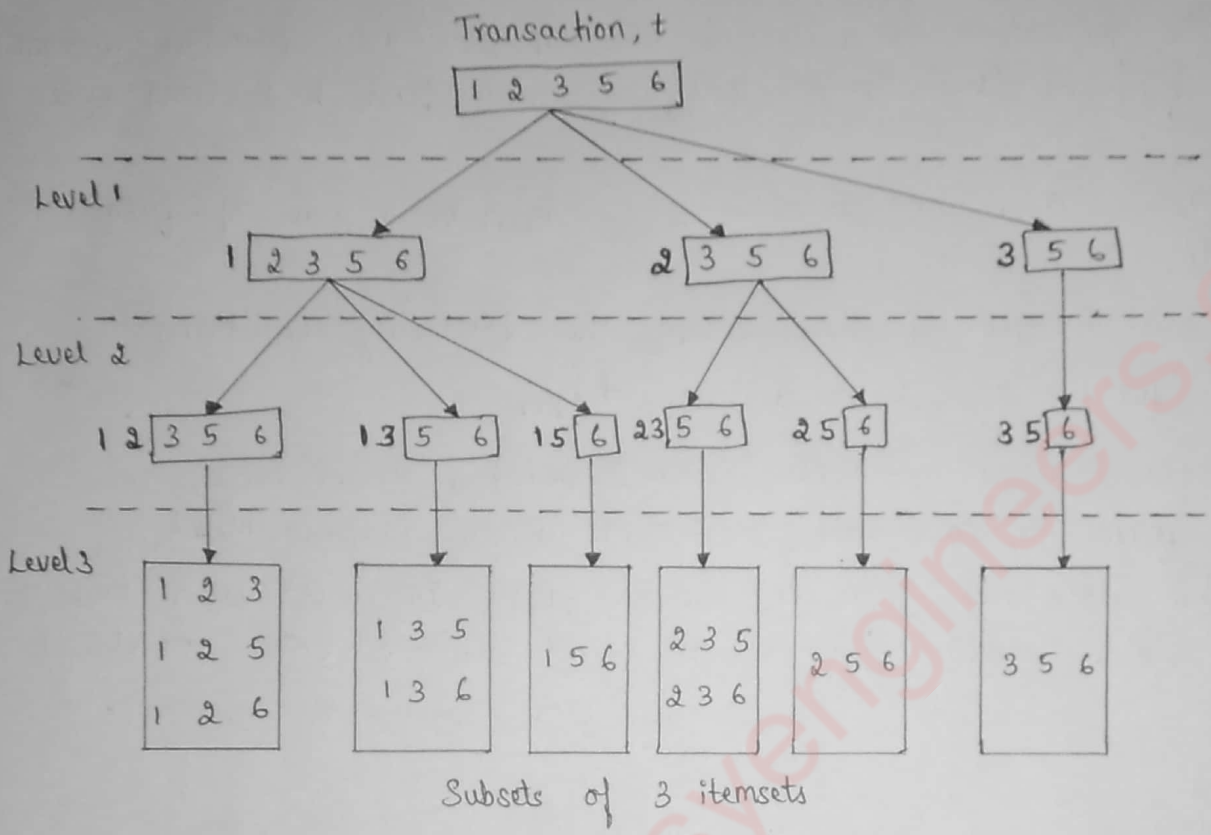
(*) Support Counting is the process of determining the frequency of occurrence for every candidate itemset that survives the candidate pruning step of the apriori-gen function.

Illustration:-

→ Consider a transaction t that contains five items, {1,2,3,5,6}.

→ There are $\binom{5}{3} = 10$ itemsets of size 3 contained in this transactions.

→ Fig(a) shows a systematic way for enumerating the 3-itemsets contained in t.



Fig(a): Enumerating subsets of three itemsets from a transaction t.

Assumptions:-

→ Each itemset keeps its item in increasing lexicographic order, an itemset can be enumerated by specifying the smallest item first, followed by the larger items.

→ Given t = {1,2,3,5,6}, all the 3-itemsets contained in t must begin with item 1, 2, or 3.

→ It is not possible to construct a 3-itemsets that begins with items 5 or 6 because there are only two items in 't' whose labels are greater than or equal to 5.

→ 1 [2 3 5 6] represents a 3-itemsets that begins with item 1, followed by two more item chosen from the set {2,3,5,6}.

→ Level 1 represents the no of ways to select the first item.
Level 2 represents the no of ways to select the second item.
Ex:- 1 2 [3 5 6] corresponds to itemsets that begin with prefix (1,2) & are followed by items 3,5 or 6.

→ level 3 represents the complete set of 3-itemsets contained in t.

Ex:- 3-itemsets that begin with prefix {1,2} are {1,2,3}, {1,2,5} and {1,2,6} while those that begin with prefix {2 3} are {2,3,5} and {2,3,6}

→ Next, we must determines whether each enumerated 3-itemsets corresp-onds to an existing candidate itemset.

→ If it matches one of the candidates, then the support count the corresponding candidate is incremented.

→ This matching operation can be performed efficiently using a hash tree structure.

Support Counting using a Hash Tree :-

→ In the Apriori algorithm, candidate itemsets are partitioned into different buckets & stored in a hash tree.

→ During support counting, itemsets contained in each transaction are also hashed into their appropriate buckets.

→ Instead of comparing each itemset in the transaction with every candidate itemset, it is matched only against candidate itemsets that belong to the same bucket, as shown in fig(10).

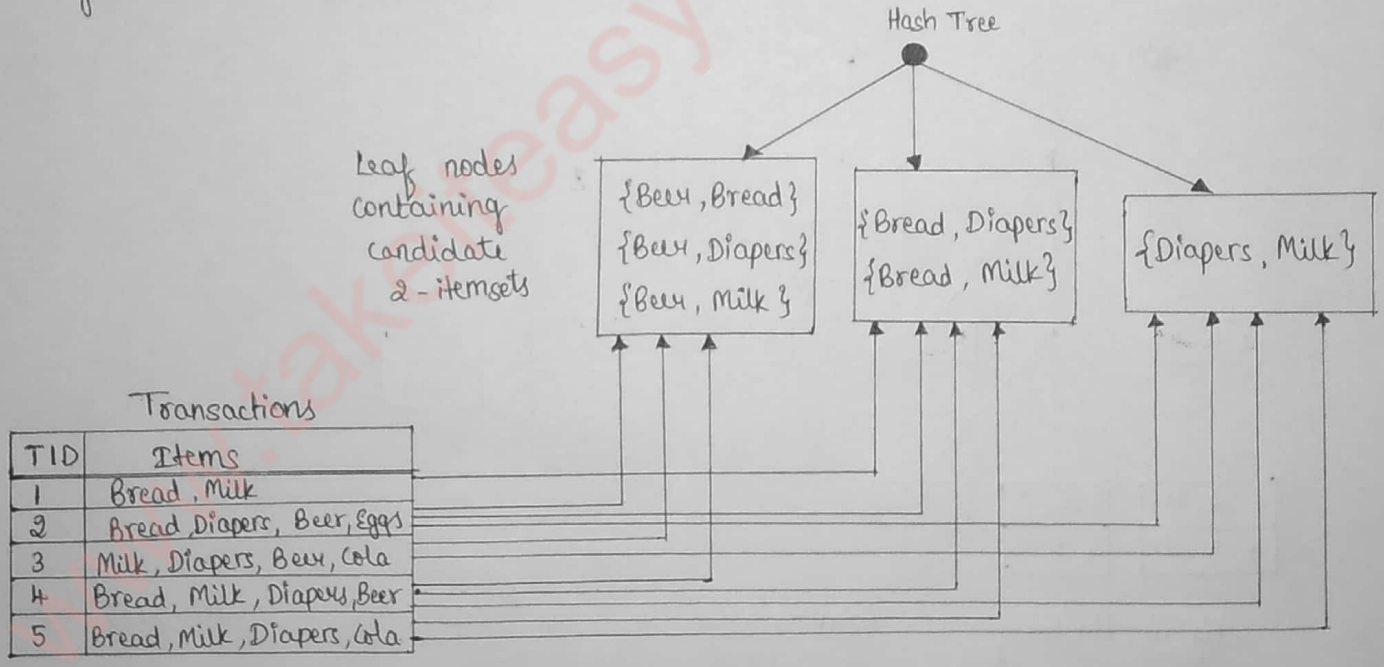
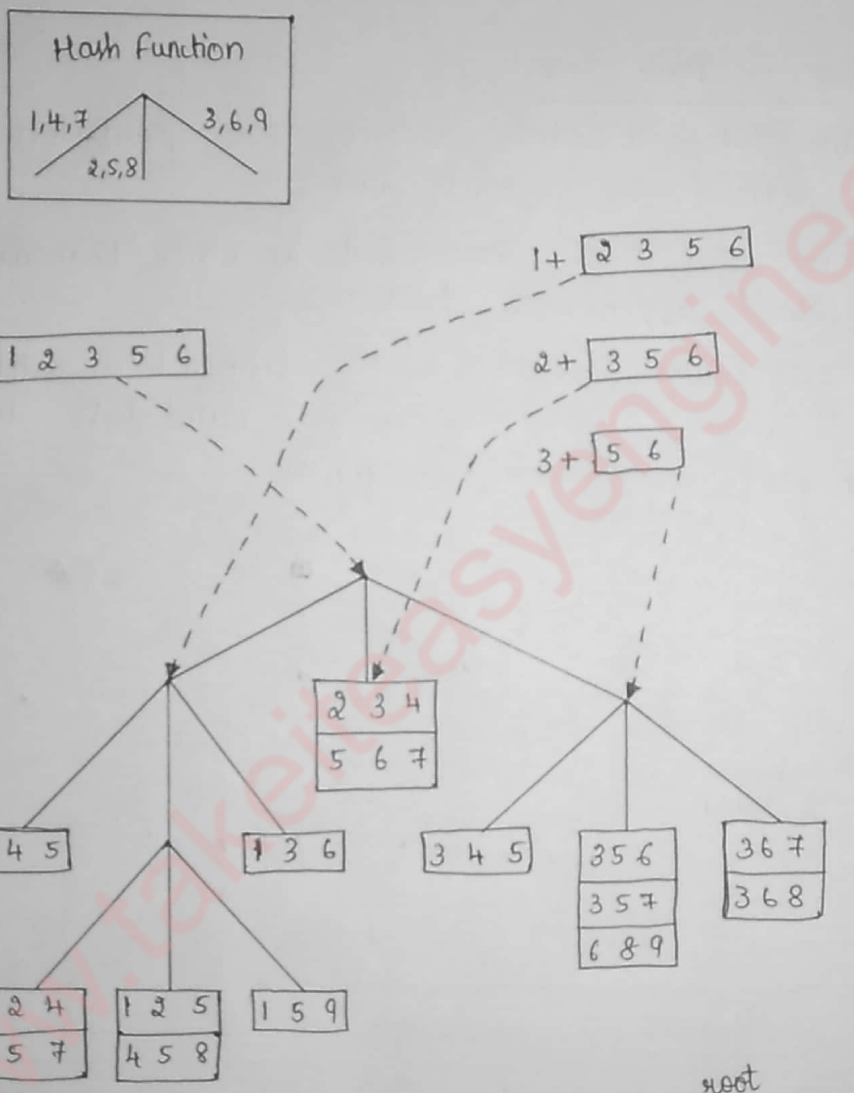


Fig (10): Counting the support of itemsets using hash structure.

(*) Fig (11), shows an example of hash tree structure.

→ Each internal node of the tree uses the following hash function $h(p) = p \bmod 3$, to determine which branch of the current node should be followed next.

→ For example, items 1, 4 & 7 are hashed to the same branch because they have same remainder after dividing no by 3.
→ All candidate itemsets are stored at the leaf nodes of the hash tree.
→ The hash tree shown in fig(ii) contains 15 candidate 3-itemsets, distrib-uted across 9 leaf nodes.
→ Consider a transaction, $t = \{1, 2, 3, 5, 6\}$. To update the support counts of the candidate itemsets, the hash tree must be traversed in such a way that all the leaf nodes containing candidate 3-itemsets belong-ing to t must be visited at least once.



Fig(ii): Hashing a transaction at the root node of a hash tree

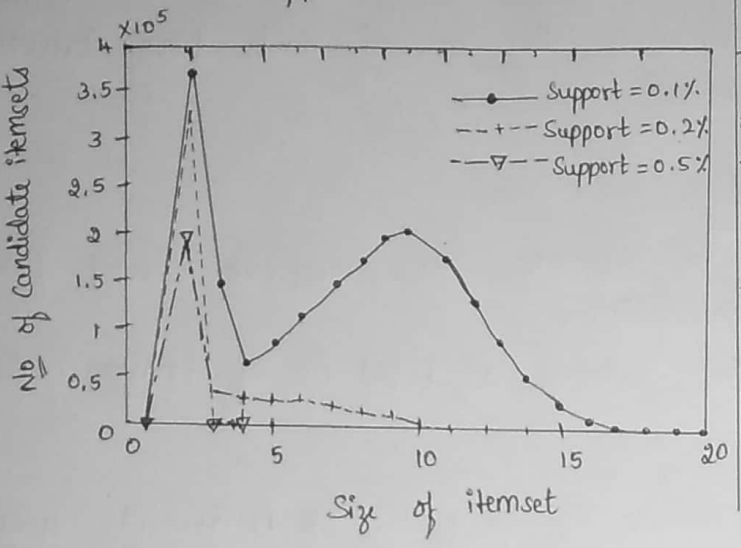
Computational Complexity :

The computational complexity of the Apriori algorithm can be affected by the following factors.

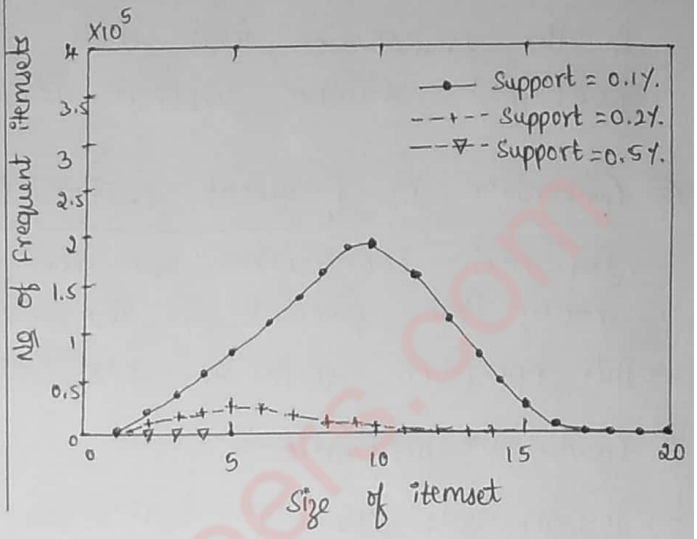
① Support Threshold :- Lowering the support threshold often results in more itemsets being declared as frequent.

→ This has an adverse effect on the computational complexity of the algorithm because more candidate itemsets must be generated and counted, as shown in fig (12).

→ The maximum size of frequent itemsets also ~~to~~ tends to increase with lower support threshold.



(a) No of candidate itemsets



(b) No of frequent itemsets

Fig(12): Effect of support threshold on the no of candidate & frequent itemsets.

② Number of items (Dimensionality) :-

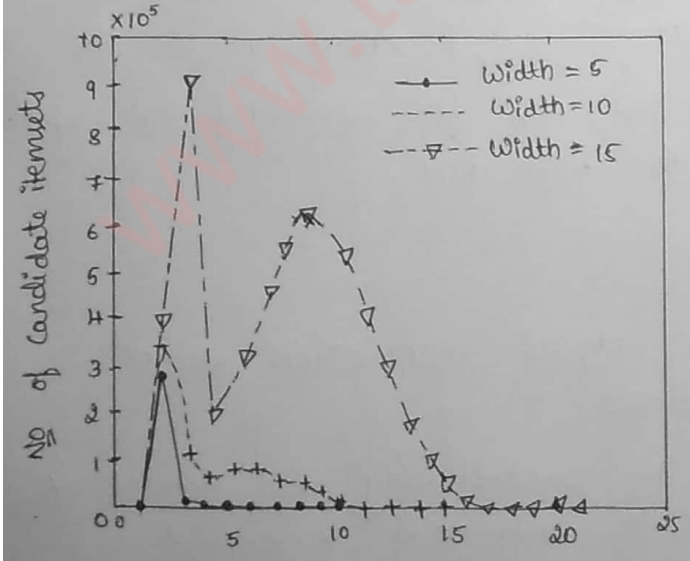
→ As the no of items increases, more space will be needed to store the support counts of items.

→ This results in more computation and I/O costs.

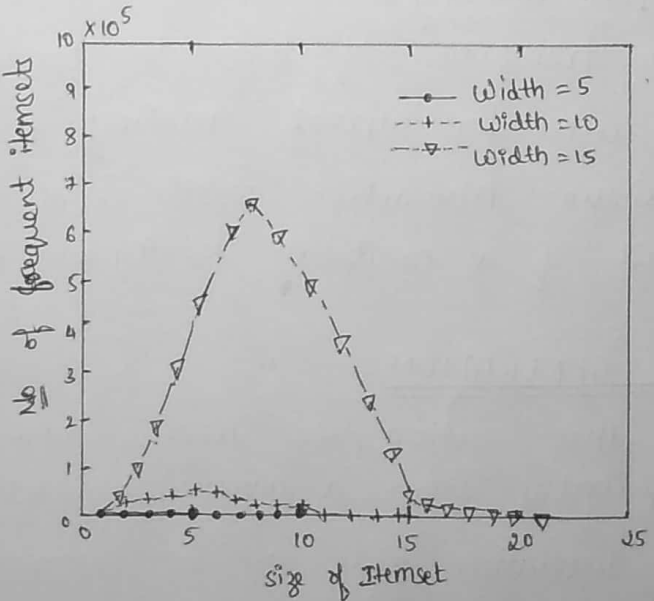
③ Number of Transactions :-

→ Larger the no of transactions, run time of Apriori algorithm increases.

④ Average Transaction width :-



(a) No of candidate itemsets



(b) No of Frequent Itemsets

Fig(13): Effect of avg transaction width on the no of candidate & frequent itemsets.

→ First, the maximum size of frequent itemsets tends to increased as the average transaction width increases. As a result, more candidate itemsets must be examined during candidate generation & ~~candidate~~ support counting as shown in fig(13).

→ Second, as the transaction width increases, more itemsets are contained in the transaction. This will increase the no of hash tree traversals performed during support counting.

⑤ Generation of frequent 1-itemsets :-

→ For each transactions, we need to update the support count for every item present in the transaction.

→ This operation requires $O(N_w)$ time where N - total no of transactions.

⑥ Candidate generation :-

→ To generate candidate k-itemsets, pairs of frequent (k-1) itemsets are merged to determine whether they have at least k-2 items in common. Each merging operation requires at most k-2 equally comparisons.

→ The overall cost of merging frequent itemsets is

$$\sum_{k=2}^{w} (k-2)|C_k| < \text{Cost of merging} < \sum_{k=2}^{w} (k-2)|F_{k-1}|^2.$$

→ During candidate pruning, we need to verify that the k-2 subsets of every candidate k-itemset are frequent.

→ Since the cost for looking up a candidate in a hash tree is $O(k)$, the candidate pruning step requires $O\left(\sum_{k=2}^{w} k(k-2)|C_k|\right)$ time.

⑦ Support Counting :-

→ Each transaction of length $|t|$ produces $\binom{|t|}{k}$ itemsets of size k.

→ The cost for support counting is $O\left(N \sum_{k} \binom{w}{k} \alpha_k\right)$, where w is the maximum transaction width & α_k is the cost for updating the support count of a candidate k-itemset in the hash tree.

RULE GENERATION :

→ In this section, we discuss how to extract association rules efficiently from a given frequent itemset.

→ An association rule can be extracted by partitioning the itemset Y into two non-empty subsets, X and Y-X such that $X \rightarrow Y-X$ satisfies the confidence threshold.

Ex:- Let $X = \{1, 2, 3\}$ be a frequent itemset.
There are six candidate association rules that can be generated from X: $\{1,2\} \rightarrow \{3\}$, $\{1,3\} \rightarrow \{2\}$, $\{2,3\} \rightarrow \{1\}$, $\{1\} \rightarrow \{2,3\}$, $\{2\} \rightarrow \{1,3\}$, and $\{3\} \rightarrow \{1,2\}$.
As each of their support is identical to the support for X, the rules must satisfy the support threshold.

Confidence-Based Pruning :-

Theorem: If a rule $X \rightarrow Y - X$ does not satisfy the confidence threshold, then any rule $X' \rightarrow Y - X'$, where X' is a subset of X, must not satisfy the confidence threshold as well.

→ To prove this theorem, consider the following two rules: $X' \rightarrow Y - X'$ and $X \rightarrow Y - X$, where $X' \subset X$. The confidence of the rules are $\sigma(Y)/\sigma(X')$ and $\sigma(Y)/\sigma(X)$, respectively. Since X' is a subset of X, $\sigma(X') \geq \sigma(X)$. Therefore, the former rule cannot have a higher confidence than the latter rule.

Rule Generation in Apriori Algorithm :-

→ The Apriori algorithm uses a level-wise approach for generating association rules, where each level corresponds to the no of items that belong to the rule consequent.

→ Initially, all the high-confidence rules that have only one item in the rule consequent are extracted.

→ These rules are then used to generate new candidate rules.
Ex:- if $\{a,c,d\} \rightarrow \{b\}$ and $\{a\ b\ d\} \rightarrow \{c\}$ are high-confidence rules, then the candidate rule $\{a\ d\} \rightarrow \{b\ c\}$ is generated by merging the consequents of both rule.

→ Fig (14) shows a lattice structure for the association rules of generated from the frequent itemset $\{a, b, c, d\}$.

→ If any node in the lattice has low confidence, then according to the theorem entire subgraph spanned by the node can be pruned immediately.

Ex:- Suppose the confidence for $\{b\,c\,d\} \rightarrow \{a\}$ is low. All the rules containing item a in its consequent, including $\{c\,d\} \rightarrow \{a\}$, $\{b\,d\} \rightarrow \{a\,c\}$, $\{b\,c\} \rightarrow \{a\,d\}$ & $\{d\} \rightarrow \{a\,b\,c\}$ can be discarded.

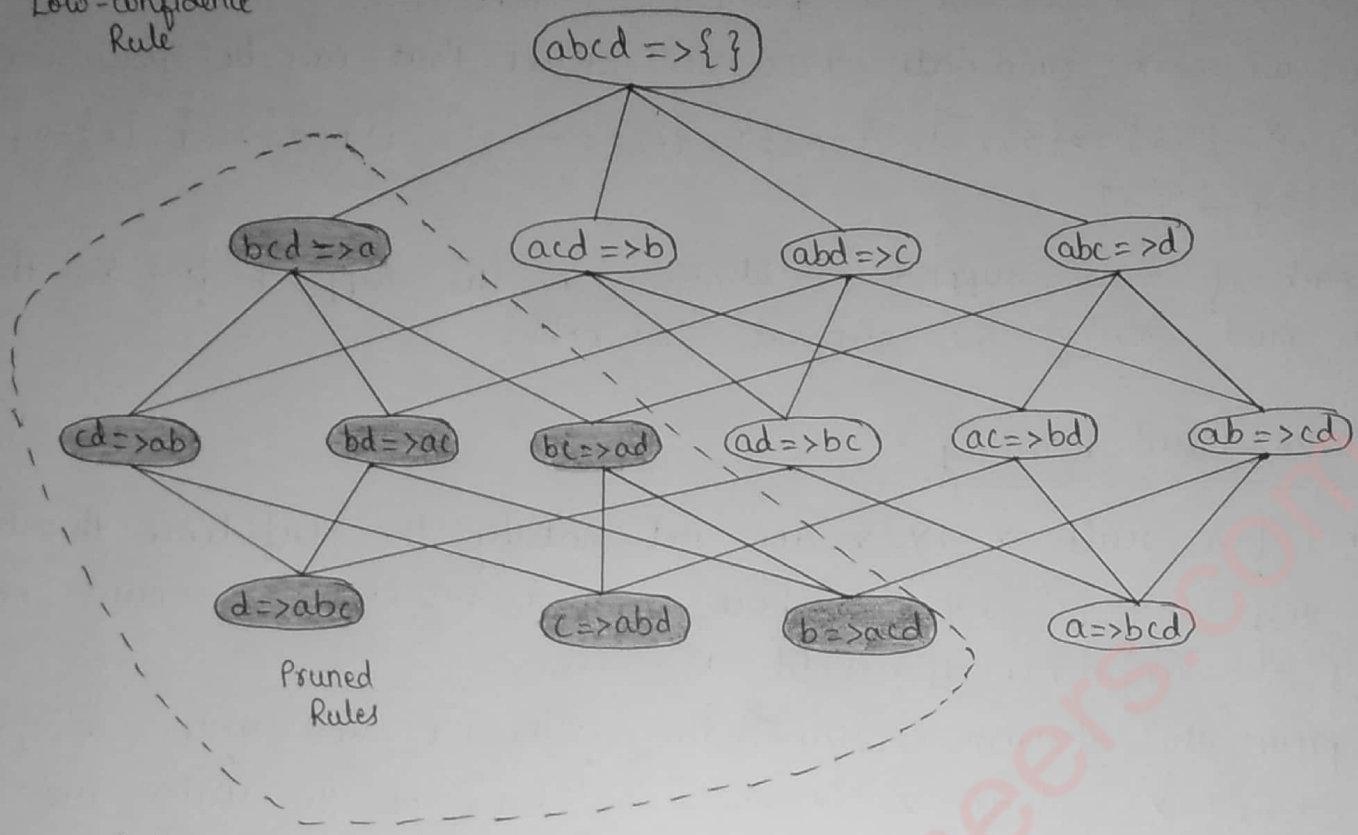


Fig (14): Pruning of association rules using the confidence measure.

→ A pseudocode for the rule generation step is shown in algorithm (2) & (3).

Algorithm 2: Rule generation of the Apriori algorithm.

1: for each frequent k-itemset f_k, $k \geq 2$ do
2: $H_1 = \{i \mid i \in f_k\}$ {1-item consequents of the rule}.
3: call ap-genrules $\{f_k, H_1\}$
4: end for

Algorithm 3: Procedure ap-genrules (f_k, H_m).

1: $k = |f_k|$ {size of frequent itemset.}
2: $m = |H_m|$ {size of rule consequent.}
3: if $k > m+1$ then
4: H_{m+1} = apriori-gen (H_m).
5: for each $h_{m+1} \in H_{m+1}$ do
6: conf $= \sigma(f_k) / \sigma(f_k - h_{m+1})$.
7: if conf $\geq$ minconf then
8: output the rule $(f_k - h_{m+1}) \rightarrow h_{m+1}$.
9: else
10: delete h_{m+1} from H_{m+1}.
11: end if
12: end for
13: call ap-genrules (f_k, H_{m+1})
14: end if

ALTERNATIVE METHODS FOR GENERATING FREQUENT ITEMSETS :

Traversal of Itemset Lattice :-

→ A search for frequent itemsets can be conceptually viewed as a traversal on the itemset lattice.

→ The search strategy employed by an algorithm dictates how the lattice structure is traversed during the frequent itemset generation process.

An overview of few search strategies is presented here:

① General-to-specific versus Specific-to-general

② Equivalence classes

③ Breadth first versus Depth first

① General-to-specific versus specific-to-general :-

→ General-to-specific :

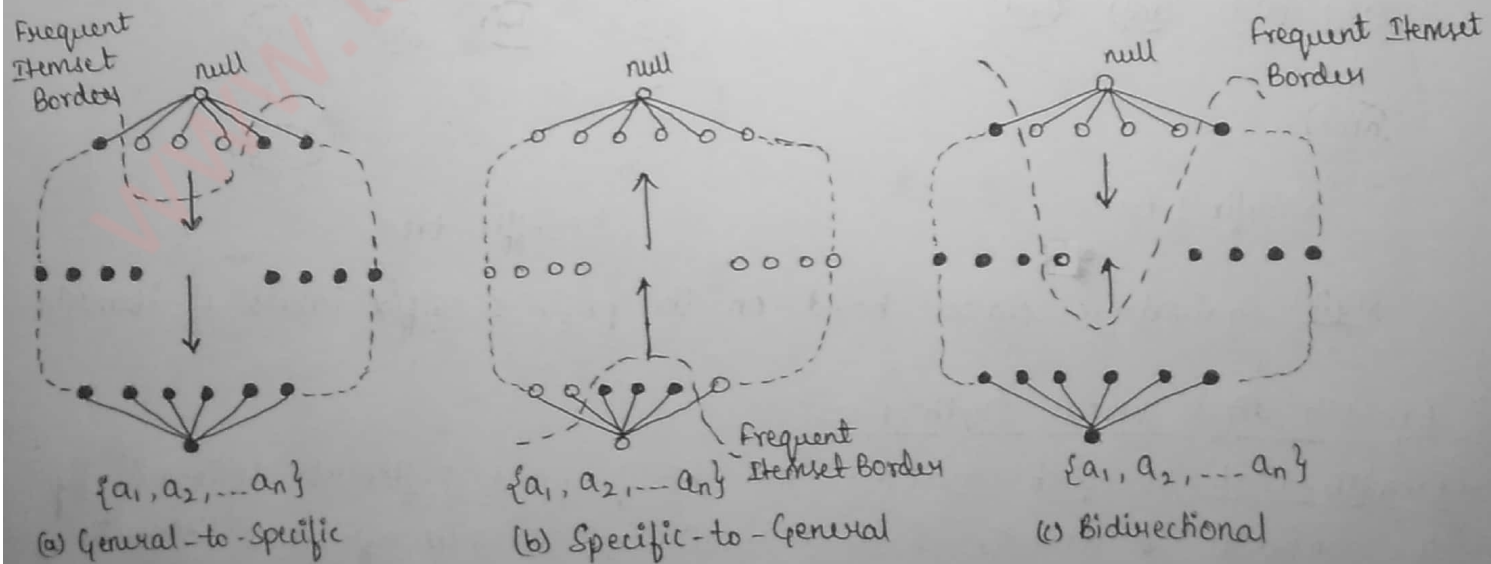
(*) The Apriori algorithm uses a general-to-specific search strategy, where pairs of frequent (k-1) itemsets are merged to obtain candidate k-itemsets.

(*) It is effective, provided the maximum length of a frequent itemset is not too long. This strategy is shown in fig. 15(a).

→ Specific-to-general :

(*) Search strategy looks for more specific frequent itemsets first, before finding the more general frequent itemsets.

(*) This strategy is useful to discover maximal frequent itemsets in dense transactions, where the frequent itemset border is located near the bottom of lattice as shown in fig. 15(b).



Fig(15): General-to-Specific, Specific-to-General and bidirectional search.

→ Another approach is to combine both general-to-specific and specific-to-general search strategies.

→ Bidirectional:

(a) This requires more space to store the candidate itemsets, but it can help to rapidly identify the frequent itemset border. The configuration is shown in fig 15(c).

② Equivalence classes:-

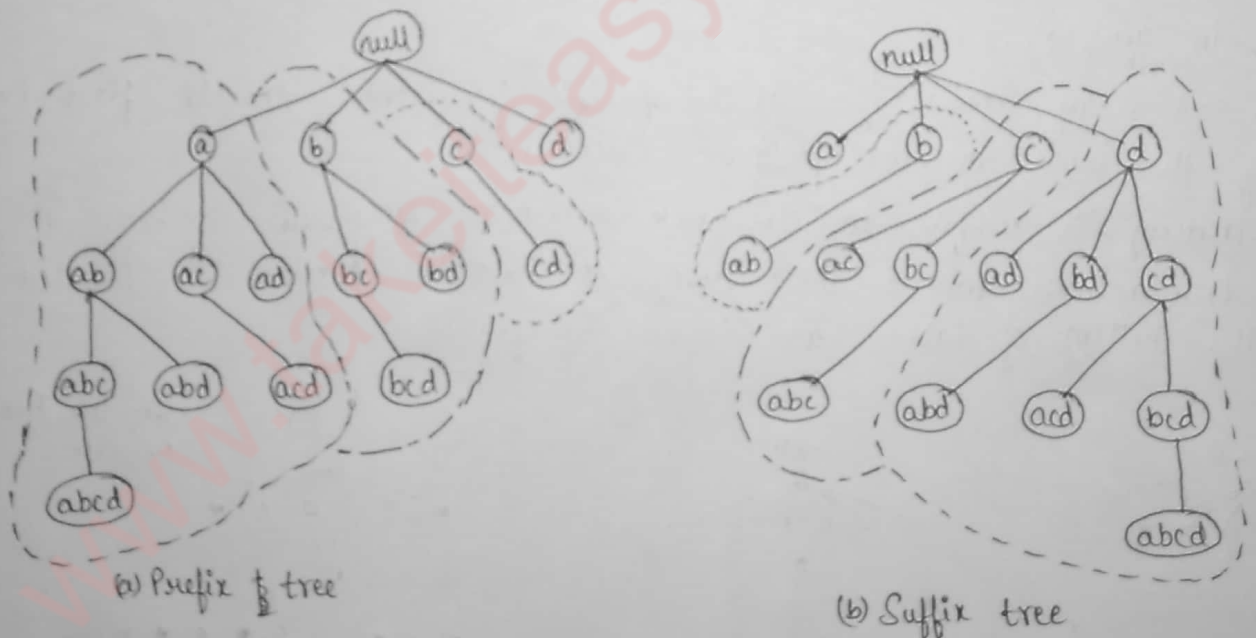
→ Another way to envision the traversal is to first partition the lattice into disjoint group of nodes [or equivalence classes].

→ A frequent itemset generation algorithm searches for frequent itemsets within a particular equivalence class first before moving to another equivalence class.

→ Equivalence class can also be defined according to the prefix or suffix labels of an itemset.

→ In this case, two itemsets belong to the same equivalence class if they share a common prefix or suffix of length k.

→ Both prefix-based and suffix-based equivalence classes can be demonstrated using tree-like structure as shown in figure (16).



(a) Prefix tree

(b) Suffix tree

Fig(16): Equivalence classes based on the prefix & suffix labels of itemsets

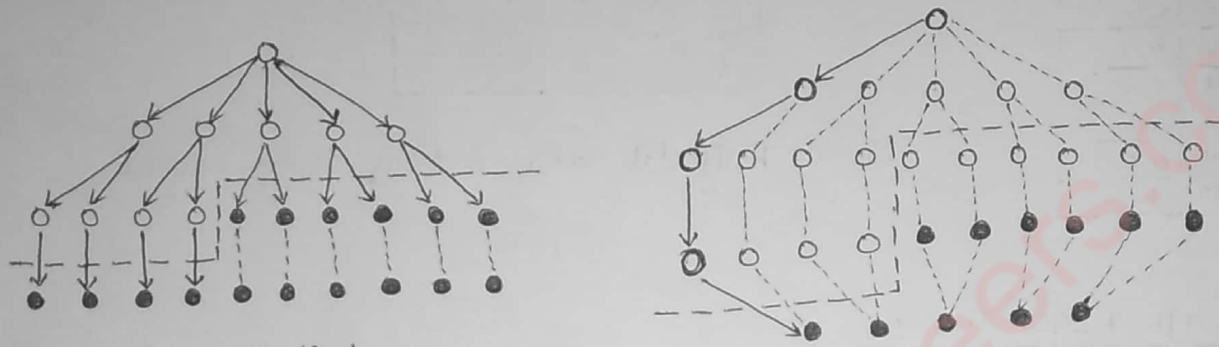
③ Breadth-First versus Depth-First:-

→ Breadth-First: It first discovers all the frequent 1-itemsets, followed by the frequent 2-itemsets & so on, until no new frequent itemsets are generated as shown in fig 17(a).

→ Depth - First : The algorithm can start from, say node a in fig(18) & count its support to determine whether it is frequent.

(*) If it is frequent, the algorithm progressively expands the next level of nodes i.e., ab, abc and so on until an infrequent node is reached say, abcd.

(*) It then back tracks to another branch say abce and continues the search from there. DFS is used by algorithm to find maximal frequent itemsets.



(a) Breadth first

(b) Depth first.

Fig(17): Breadth-first and depth-first traversals.

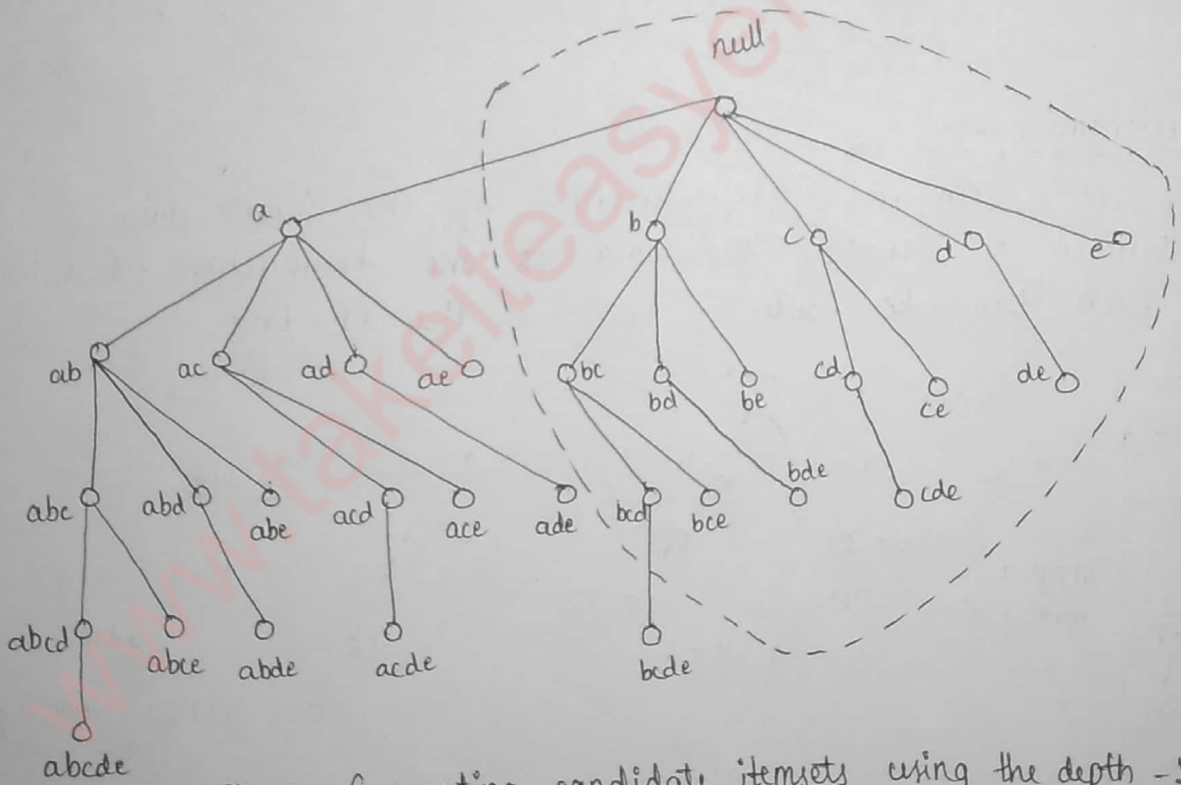


Fig 18:- Generating candidate itemsets using the depth-first approach

Representation of Transaction Data Set :-

→ Fig (19): shows two different ways of representing market basket transaction.

→ The representation on the left is called a horizontal data layout, which is adopted by many association rule mining algorithms including Apriori.

→ Another possibility is to store the list of transaction identifier (TID list) associated with each item. Such a representation is known as vertical data layout.

→ ~~One~~

Horizontal Data Layout

TID	Items
1	a, b, e
2	b, c, d
3	c, e
4	a, c, d
5	a, b, c, d
6	a, e
7	a, b
8	a, b, c
9	a, c, d
10	b

Vertical Data Layout

a	b	c	d	e
1	1	2	2	1
4	2	3	4	3
5	5	4	5	6
6	7	8	9	
7	8	9		
8	10			
9				

Fig (19): Horizontal and Vertical Data Layout.

FP-GROWTH ALGORITHM:

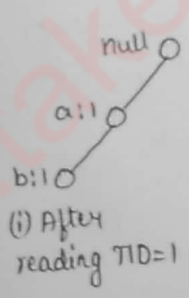
FP-growth that takes a radically different approach to discovering frequent itemsets, it encodes the dataset using a compact data structure called an FP-tree and extracts frequent itemsets directly from this structure.

FP-Tree Representation:-

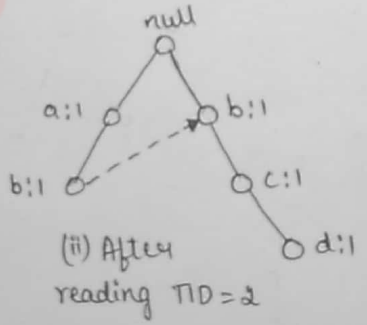
→ An FP-tree is a compressed representation of the input data.
→ It is constructed by reading the data set one transaction at a time & mapping each transaction onto a path in the FP-tree.

Transaction Data set

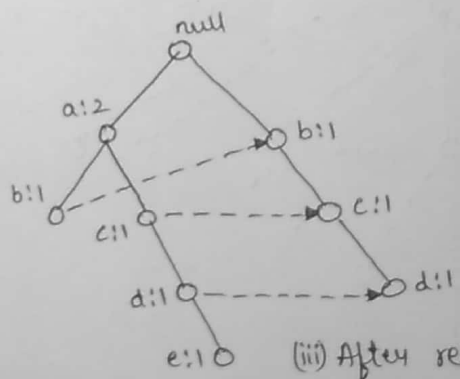
TID	Items
1	{a, b}
2	{b, c, d}
3	{a, c, d, e}
4	{a, d, e}
5	{a, b, c}
6	{a, b, c, d}
7	{a}
8	{a, b, c}
9	{a, b, d}
10	{b, c, e}



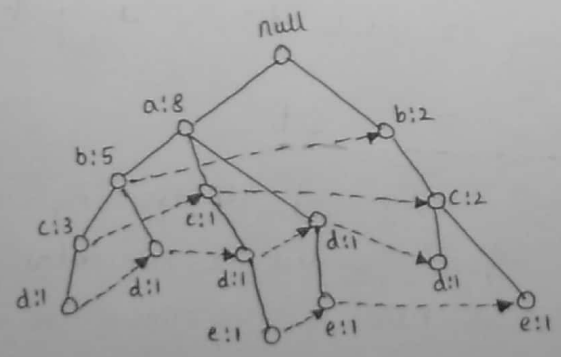
(i) After reading TID=1



(ii) After reading TID=2



(iii) After reading TID=3



(iv) After reading TID=10

Fig(20): Construction of an FP-tree

→ Fig (20), shows a data set that contains 10 transactions & 5 items & also the structures of the FP-tree after reading the first three transactions are also depicted in the diagram.

→ Each node in the ~~th~~ tree contains the label of an item along with a counter that shows the no of transactions mapped onto the given path.

(*) Initially, the FP-tree contains only the root node represented by the null symbol.

→ The FP-tree is subsequently extended in following way:

① The data set is scanned once to determine the support count of each item. Infrequent items are discarded, while the frequent items are sorted in decreasing support counts. For the data set shown in fig (20) a is the most frequent item, followed by b, c, d, and e.

② The algorithm makes a second pass over the data to construct the FP-tree. After reading the first transaction, {a, b}, the nodes labell-ed as a and b are created. A path is then formed from null → a → b to encode the transaction. Every node along the path has a frequency count of 1.

③ After reading the second transaction, {b, c, d}, a new set of nodes is created for items b, c, and d. A path is then formed to repr-esente the transaction by connecting the nodes null → b → c → d. Every node along this path also has a frequency count equal to one. Although the first two transactions have an item in common, which is b, their paths are disjoint because the transactions do not share a common prefix.

④ The third transaction, {a, c, d, e}, shares a common prefix item with the first transaction. As a result, the path for the third transaction, null → a → c → d → e, overlaps with the path for the first transaction, null → a → b. Because of their overlapping path, the frequency count for node a is incremented to two, while the frequency counts for the newly created nodes, c, d, and e, are equal to one.

⑤ This process continues until every transaction has been mapped onto one of the paths given in the FP-tree. The resulting FP-tree after reading all the transactions is shown at the bottom of fig (20).

Frequent itemset Generation in FP-Growth Algorithm :-

→ FP-growth is an algorithm that generates frequent itemsets from an FP-tree by exploring the tree in a bottom-up fashion.

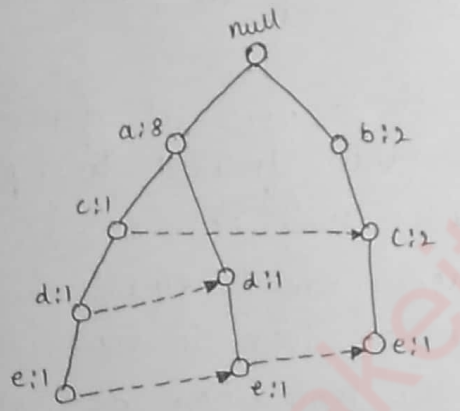
Ex :- Fig (20) the algorithm looks for frequent itemsets ending in e first, followed by d, c, b and finally a.

→ Since every transaction is mapped onto a path in FP-tree, we can derive the frequent itemsets ending with a particular item say, e, by examining only the paths containing node e.

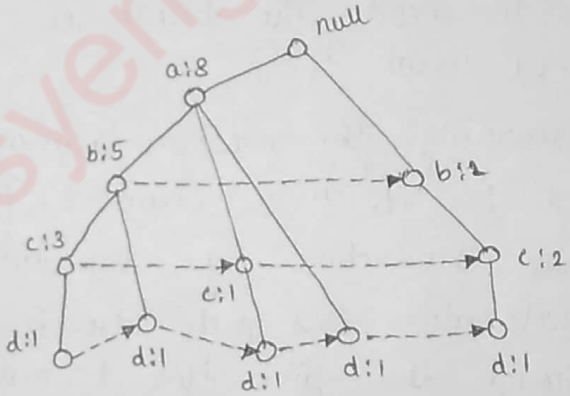
→ These paths can be accessed rapidly using the pointers associated with node e. The extracted paths are shown in fig (21)(a).

→ After finding the frequent itemsets ending in e, the algorithm proceeds to look for frequent itemsets ending in d by processing the paths associated with node d. The corresponding paths are shown in fig 21(b).

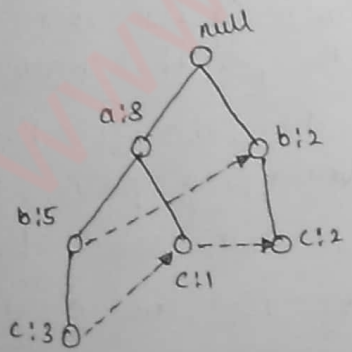
→ This process continues, until all the paths associated with nodes c, d & finally a are processed. The paths for these items are shown in fig 21 (c), (d) & (e).



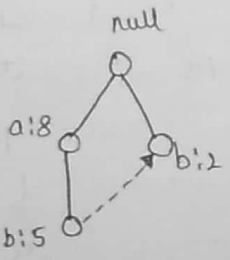
(a) paths containing node e



(b) Paths containing node d



(c) Paths containing node c



(d) Paths containing node b

(e) Paths containing node a

Fig 21 : Decomposing the frequent itemset generation problem into multiple subproblems, where each subproblem involves finding frequent itemsets ending in e, d, c, b and a.

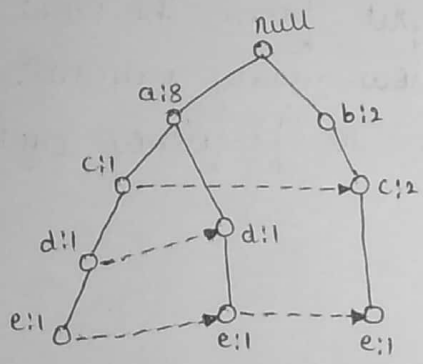
Consider the task of finding frequent itemsets ending with e.

→ The first step is to gather all the paths containing node e. These initial paths are called prefix paths & are shown in fig 22 (a).

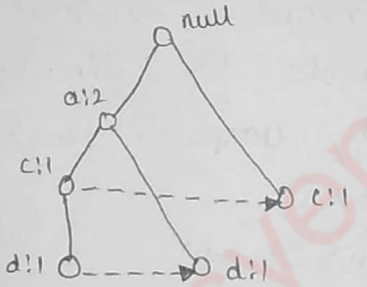
→ From the prefix paths shown in fig 22 (a), the support count for e is obtained by adding the support counts associated with node e. Assuming that the maximum support count is 2, {e} is declared a frequent itemset because its support count is 3.

→ Because {e} is frequent, the algorithm has to solve the sub problems of finding frequent itemsets ending in de, ce, be & ae. Before solving these subproblems, it must first convert the prefix paths into a conditional FP-tree.

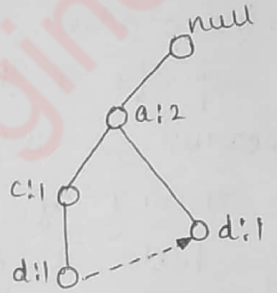
→ FP-growth uses conditional FP-tree for e to solve the subproblems of finding frequent itemsets ending in de, ce and ae.



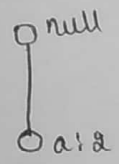
(a) Prefix paths ending in e



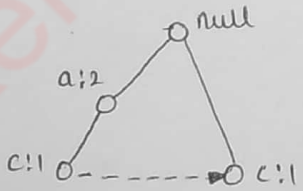
(b) Conditional FP-tree for e



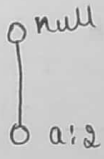
(c) Prefix paths ending in de.



(d) Conditional FP-tree for de



(e) Prefix paths ending in ce



(f) Prefix paths ending in ae

Fig (22) : Example of applying the FP-growth algorithm to find frequent itemsets ending in e.

EVALUATION OF ASSOCIATION PATTERNS :

The Association Analysis Algorithms have the potential to generate a large no of patterns, we could easily end up with thousands or even millions of patterns, many of which might not be interesting.

→ "Objective interestingness measure" that uses statistics derived from data to determine whether a pattern is interesting indeed.

→ Different objective measures define different association patterns with different properties & applications.

→ The different types of Association patterns evaluation criteria are:

① Support - confidence Framework
② Interest Factor
③ Correlation Analysis
④ IS Measure

① Support - Confidence Framework :-

→ It is a data - driven approach for evaluating the quality of association patterns.

→ It is domain - independent & requires minimal input from the users, other than to specify a threshold for filtering low-quality patterns.

→ An objective measure is usually computed based on the frequency counts tabulated in a contingency table.

→ The contingency table is shown below:

	B	$\bar{B}$	
A	f_{11}	f_{10}	f_{1+}
$\bar{A}$	f_{01}	f_{00}	f_{0+}
	f_{+1}	f_{+0}	N

Limitations of Support - confidence Framework :

Existing association rule mining formulation relies on the support & confidence measures to eliminate uninteresting patterns.

→ The drawback of support was elimination of potentially interesting patterns & the drawback of confidence is more stable.

② Interest Factor :-

→ The High - confidence rules can sometimes be misleading because the confidence measure ignores the support of itemset appearing in the rule consequent.

→ One way to address this problem is by applying metric known as "LIFT":

$$\text{Lift} = \frac{c(A \rightarrow B)}{s(B)}$$

which computes the ratio b/w rules confidence & the support of the itemset in rule consequent.

Limitations of interest factor :-

An example from the text mining domain, it is reasonable to assume that the association b/w a pair of words depends on the no of documents that contain both words.

③ Correlation Analysis :-

→ Correlation analysis is statistical Based technique for analyzing relationship between a pair of variables.

→ For continous variables, correlation is defined using "pearson's correlation co-efficient", for the Binary variables is given by,

$$\phi = \frac{F_{11}F_{00} - F_{01}F_{10}}{F_{1+}F_{+1}F_{0+}F_{+0}}$$

Limitations of correlation Analysis :-

→ The drawback of using correlation can be seen from the word association.

→ Because the ϕ-coefficient gives equal importance to both co-presence & co-absence of items in a transaction.

④ IS - Measure :-

→ IS is an alternative measure that has been proposed for handling asymmetric binary variables.

→ The IS measure is defined as follows:

$$IS(A,B) = \sqrt{I(A,B) \times s(A,B)} = \frac{s(A,B)}{\sqrt{s(A)s(B)}}$$

→ The IS measure is large where the interest factor & support of the pattern are large.

Limitations of IS-Measure :-

→ Shares a similar problem as confidence measure that the value of the measure can be quite large even for uncorrelated and negatively correlated patterns.