UNIT –V

ASSOSIATION ANALYSIS

Introduction

- Association analysis, which is suitable for extracting interesting relationships hidden in large transaction data sets.
- The extracted relationships are represented in the form of association rules that can be used to predict the presence of certain items in a transaction based on the presence of other items.
- For example,

{ Bread } → { Milk }

The rule suggests that many customers who buy bread also tend to buy milk .

- An example of grocery store data or commonly known as market-basket transactions. Each row (transaction) contains a unique identifier labeled as Tid and a set of items bought by a given customer.

Tid	Items
1	{**Bread, Milk**}
2	{**Bread**, Diaper, Beer, Eggs}
3	{**Milk**, Diaper, Beer, Coke}
4	{**Bread, Milk,** Diaper, Beer}
5	{**Bread, Milk,** Diaper, Coke}

An example of market-basket transactions.

- **Example of Association Rules**

{Diaper} → {Beer},

{Milk, Bread} → {Eggs,Coke},

{Beer, Bread} → {Milk},

Implication means co-occurrence, not causality!

5.1 Problem Definition

- Market-basket data can be represented in a binary format as shown in below Table , where each row corresponds to a transaction and each column corresponds to an item.

TID	Bread	Milk	Diaper	Beer	Eggs	Coke
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

A binary 0/1 representation of market-basket data.

- An item can be treated as a binary variable whose value is one if the item is present in a transaction and zero otherwise.
- Presence of an item in a transaction is often considered to be more important than its absence.
- The number of items present in a transaction also defines the transaction width.
- Such representation is perhaps a very simplistic view of real market-basket data because it ignores certain important aspects of the data such as the quantity of items sold or the price paid for the items.

Itemset and Support Count

- Let I = {i1, i2, ・・・, id} be the set of all items.
- An itemset is defined as a collection of zero or more items.
- If an itemset contains k items, it is called a k-itemset.
- {Beer,Diaper,Milk}, for instance, is an example of a 3-itemset while the null set, { }, is an itemset that does not contain any items.
- Let T = {t1, t2, ・・・, tN} denote the set of all transactions, where each transaction is a subset of items chosen from I.
- A transaction t is said to contain an itemset c if c is a subset of t.

Support count

- An important property of an itemset is its support count, which is defined as the number of transactions that contain the particular itemset.
- E.g. σ({Milk, Bread,Diaper}) = 2

Support

- Fraction of transactions that contain an itemset .
- E.g. s({Milk, Bread, Diaper}) = 2/5

Frequent Itemset

- An itemset whose support is greater than or equal to a *minsup* threshold.

Association Rule

- An association rule is an implication expression of the form $X \rightarrow Y$, where X and Y are disjoint itemsets, i.e., $X \cap Y = \emptyset$.
- The strength of an association rule is often measured in terms of the support and confidence metrics.

- Support determines how frequently a rule is satisfied in the entire data set and is defined as the fraction of all transactions that contain X ∪ Y.
- Confidence determines how frequently items in Y appear in transactions that contain X.
- The formal definitions of these metrics are given below.

$$\text{support, } s(X \longrightarrow Y) = \frac{\sigma(X \cup Y)}{N} \text{ and}$$

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

Tid	Items
1	{**Bread, Milk**}
2	{**Bread**, Diaper, Beer, Eggs}
3	{**Milk**, Diaper, Beer, Coke}
4	{**Bread, Milk,** Diaper, Beer}
5	{**Bread, Milk,** Diaper, Coke}

Formulation of Association Rule Mining Problem

- **(Association Rule Discovery)** Given a set of transactions T, find all rules having support ≥ minsup and confidence ≥ minconf, where minsup and minconf are the corresponding support and confidence thresholds.

Brute-force approach:

- Mining association rules is to enumerate all possible rule combinations .
- This approach is prohibitively expensive since there are exponentially many rules that can be extracted from a transaction data set.
- More specifically, for a data set containing d items, the total number of possible rules is

  $R = 3^d - 2^{d+1} + 1$
- Then compute their support and confidence values for all possible rules .
- Prune rules that fail the *minsup* and *minconf* thresholds .
- Suppose consider itemset {Milk, Diaper, Beer} . For this we are getting rules and their support & confidence values are

  {Milk,Diaper} → {Beer} (s=0.4, c=0.67)

  {Milk,Beer} → {Diaper} (s=0.4, c=1.0)

  {Diaper,Beer} → {Milk} (s=0.4, c=0.67)

  {Beer} → {Milk,Diaper} (s=0.4, c=0.67)

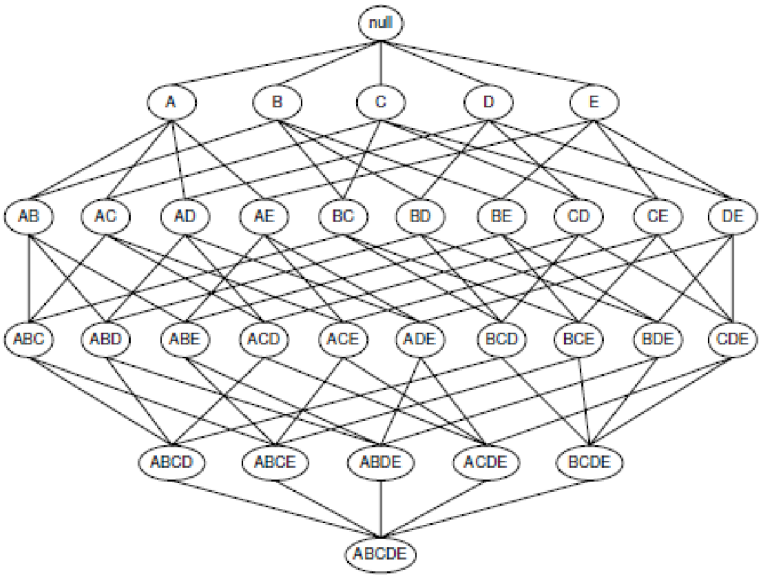
  {Diaper} → {Milk,Beer} (s=0.4, c=0.5)

  {Milk} → {Diaper,Beer} (s=0.4, c=0.5)

- All the above rules are binary partitions of the same itemset: {Milk, Diaper, Beer}
- Rules originating from the same itemset have identical support but can have different confidence .
- Thus, we may decouple the support and confidence requirements.
- A common strategy adopted by many association rule mining algorithms is to decompose the problem into two major subtasks:
  1. ***Frequent Itemset Generation.*** Find all itemsets that satisfy the minsup threshold. These itemsets are called frequent itemsets.
  2. ***Rule Generation.*** Extract high confidence association rules from the frequent itemsets found in the previous step. These rules are called strong rules.

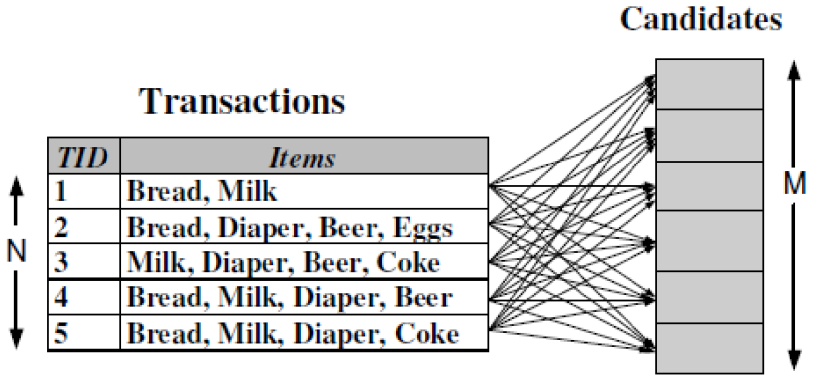
5.2 Frequent Itemset Generation

- A lattice structure can be used to enumerate the list of possible itemsets.
- For example, below Figure illustrates all itemsets derivable from the set {A,B,C,D,E}.



The Itemset Lattice

- In general, a data set that contains d items may generate up to $2^d - 1$ possible itemsets, excluding the null set.
- Each itemset in the lattice is a candidate frequent itemset .
- Count the support of each candidate by scanning the Database .
- Some of these itemsets may be frequent, depending on the choice of support threshold.



Counting the support of candidate itemsets.

- If the candidate is contained within a transaction, its support count will be incremented.
- For example, the support for {Bread,Milk} is incremented three times since the itemset is contained within transactions 1, 4, and 5.
- Such a brute force approach can be very expensive because it requires O(NMw) matching operations, where N is the number of transactions, M is the number of candidate itemsets, and w is the maximum transaction width.
- There are a number of ways to reduce the computational complexity of frequent itemset generation.
  1. Reduce the number of candidate itemsets (M). The Apriori principle, to be described in the next section, is an effective way to eliminate some of the candidate itemsets before counting their actual support values.
  2. Reduce the number of candidate matching operations. Instead of matching each candidate itemset against every transaction, we can reduce the amount of comparisons by using advanced data structures to store the candidate itemsets or to compress the transaction data set.

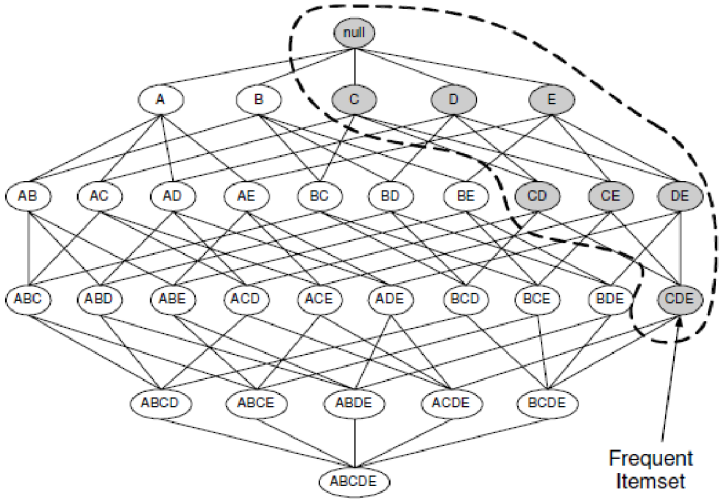
5.2.1 Reducing Number of Candidates by Apriori principle:

Apriori principle:

- If an itemset is frequent, then all of its subsets must also be frequent.
- Apriori principle holds due to the following property of the support measure:
- Support of an itemset never exceeds the support of its subsets . This is known as the anti-monotone property of support

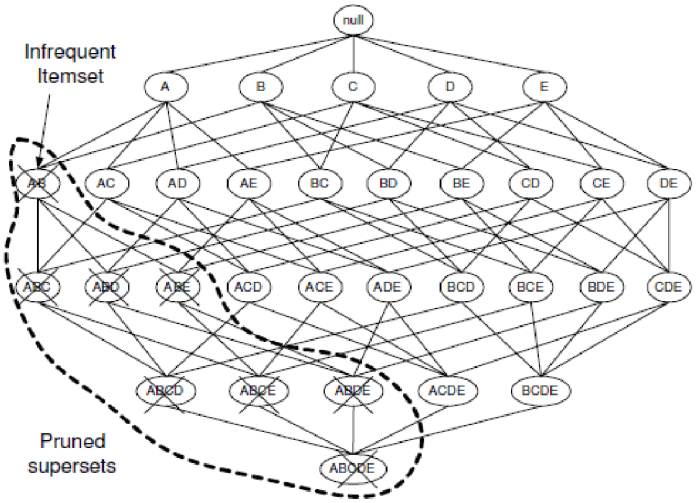
$$\forall X, Y : (X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

- If an itemset such as {C,D,E} is found to be frequent, then the Apriori principle suggests that all of its subsets must also be frequent.



An illustration of the Apriori principle. If {C,D,E} is frequent, then all subsets of this itemset are frequent.

- Conversely, if an itemset such as {A,B} is infrequent, then all of its supersets must be infrequent too.
- For example once {A,B} is found to be infrequent the entire subgraph containing supersets of {A,B} can be pruned immediately.
- This strategy of trimming the exponential search space based on the support measure is known as support-based pruning.



An illustration of support-based pruning. If {A,B} is infrequent, then all supersets of {A,B} are eliminated.

5.2.2 Frequent itemset Generation using Apriori Algorithm

- Apriori is the first algorithm that pioneered the use of support-based pruning to systematically control the exponential growth of candidate itemsets.
- Below Figure provides a high level illustration of the Apriori algorithm for the market-basket transactions .

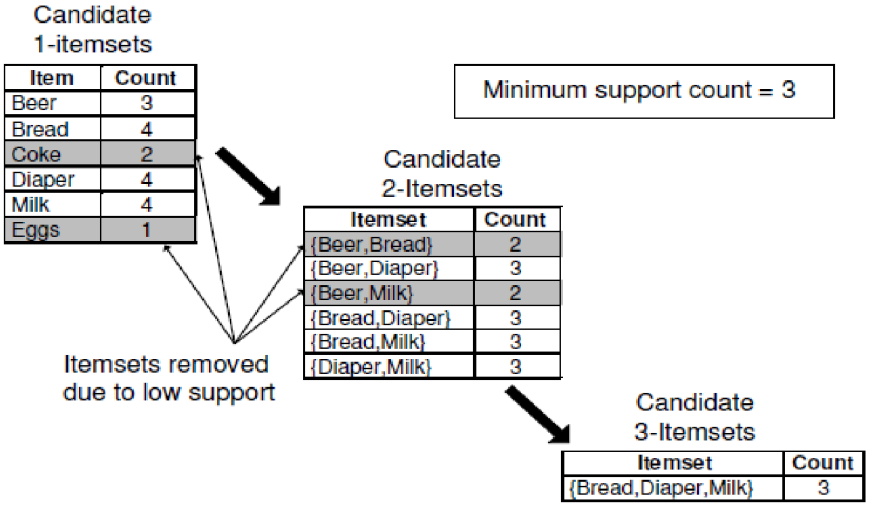


Illustration of Apriori algorithm

- Method:
  1. Let k=1
  2. Generate frequent itemsets of length 1 .
  3. Repeat until no new frequent itemsets are identified
     a. Generate length (k+1) candidate itemsets from length k frequent itemsets .
     b. Prune candidate itemsets containing subsets of length k that are infrequent .
     c. Count the support of each candidate by scanning the DB .
     d. Eliminate candidates that are infrequent, leaving only those that are frequent .
- We assume the minimum support count to be equal to 3 (i.e., support threshold = 3/5 = 60%). Initially, each item is considered as a candidate 1-itemset.
- The candidate itemsets {Coke} and {Eggs} are discarded because they are present in less than 3 transactions.
- In the next iteration candidate 2-itemsets are generated using only the frequent -1 itemsets.
- There are four frequent- 1 itemset, the number of candidate 2-itemsets generated is equal to ${}^4C_2 = 6$.
- Two of these six candidates, {Beer,Bread} and {Beer,Milk}, are found to be infrequent upon computing their actual support counts.

- The remaining four candidates are frequent, and thus, will be used to generate candidate 3-itemsets.
- The effectiveness of the Apriori pruning strategy can be seen by looking at the number of candidate itemsets considered for support counting.
- A brute-force strategy of enumerating all itemsets as candidates will produce

$$\binom{6}{1} + \binom{6}{2} + \binom{6}{3} = 6 + 15 + 20 = 41$$

candidates.

- With the Apriori principle, this number decreases to

$$\binom{6}{1} + \binom{4}{2} + 1 = 6 + 6 + 1 = 13$$

candidates, which represents a 68% reduction in the number of candidate itemsets even in this simple example.

- The pseudo code for the Apriori algorithm is

1: $k = 1$.
2: $F_k = \{ i \mid i \in I \wedge \frac{\sigma(\{i\})}{N} \geq minsup \}$. {Find all frequent 1-itemsets}
3: **repeat**
4: $k = k + 1$.
5: C_k = apriori-gen(F_{k-1}). {Generate candidate itemsets}
6: **for** each transaction $t \in T$ **do**
7: C_t = subset(C_k, t). {Identify all candidates that belong 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: $F_k = \{ c \mid c \in C_k \wedge \frac{\sigma(c)}{N} \geq minsup \}$. {Extract the frequent k-itemsets}
13: **until** $F_k = \emptyset$
14: Result = $\bigcup F_k$.

- Initially, the algorithm makes a single pass over the transaction data set to count the support of each item. Upon completion of this step, the set of all frequent 1-itemsets, F1, will be known (steps 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 genaration is implemenred using a function called apriori-gen .
- The subset function is used to determine all the candidate itemsets in C_k that are aontained in each transaction t.

- To count the support for each candidates, the algorithm needs to make an additional pass over the data set (steps 6 -10)
- Eliminating candidate itemsets whose support count is less than the minsup threshold (step 12).
- The algorithm terminates when there are no new frequent itemsets generated, i.e., Fk = ∅ (step 13).

5.2.3 Generating and Pruning Candidate Itemsets

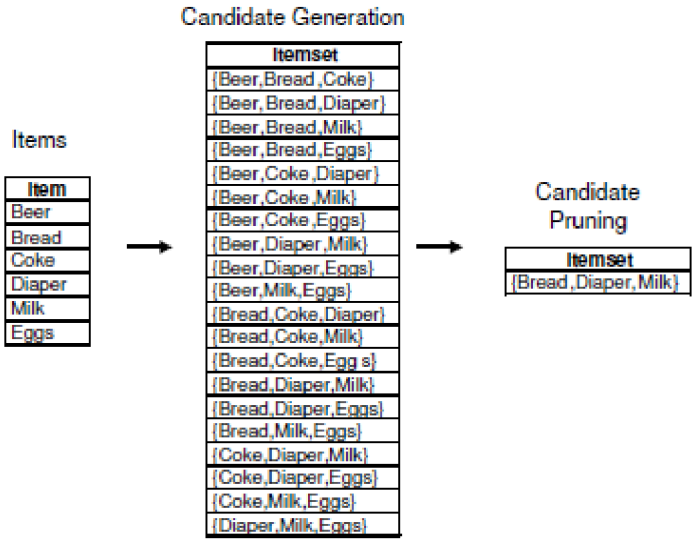
- The apriori-gen function given in Step 5 of Algorithm generates candidate itemsets by performing the following two operations:

  Candidate Generation: This operation generates new candidate k-itemsets from frequent itemsets of size k − 1.

  Candidate Pruning: This operation prunes all candidate k-itemsets for which any of their subsets are infrequent.

Brute-force Method

- The brute- force method considers every k-itemset as a potential candidate and then apply the candidate pruning step to remove any unnecessary candidates.



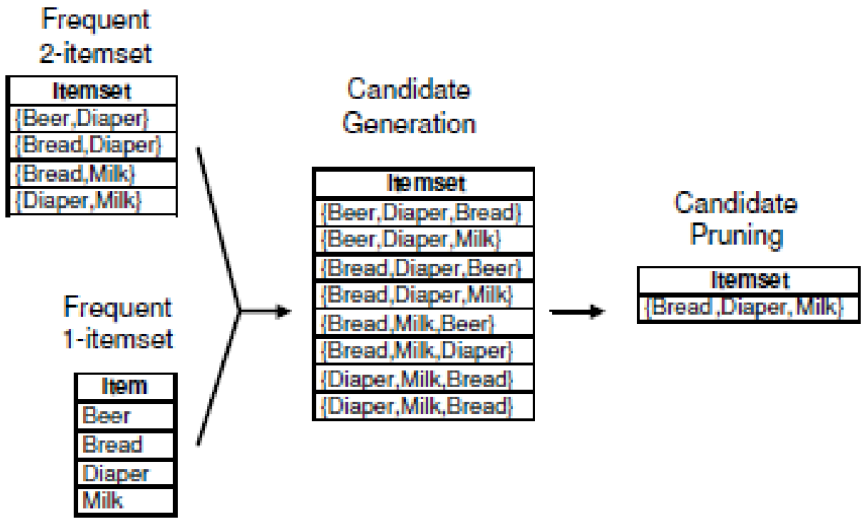
Brute force method for generating candidate 3-itemsets

- With this approach, the number of candidate itemsets generated at level k is equal to $\binom{d}{k}$, where d is the total number of items.
- While candidate generation is rather trivial, the candidate pruning step becomes extremely expensive due to the large number of candidates that must be examined.
- Given that the amount of computation needed to determine whether a candidate k-itemset should be pruned is O(k), the overall complexity of this method is

$O[\sum_k k \times \binom{d}{k}]$.

Fk−1×F1 Method

- An alternative method for candidate generation is to extend each frequent (k−1) itemset with other frequent items.
- For example,the below Figure illustrates how a frequent 2-itemset such as {Beer,Diaper} can be augmented with a frequent item such as Bread to produce a candidate 3-itemset {Beer,Diaper,Bread}.



Generating and pruning candidate k-itemsets by merging a frequent (k − 1)-itemset with a frequent item.

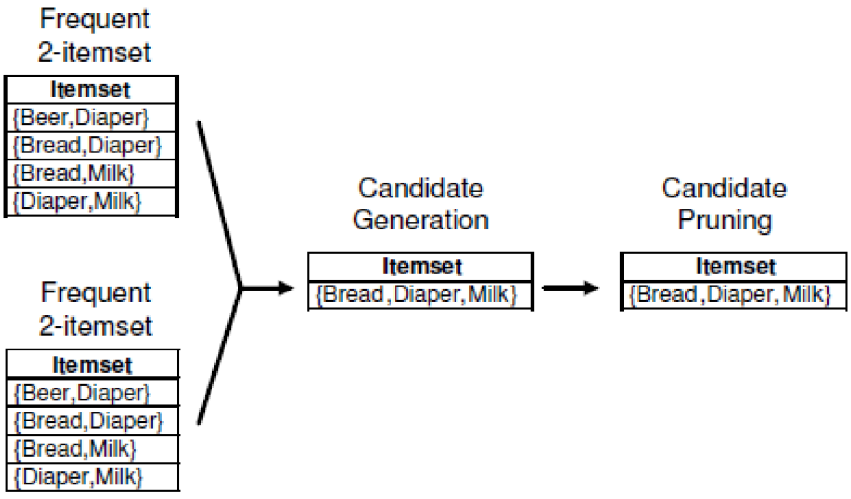
- This method will produce O(|Fk−1| × |F1|) candidate itemsets, where |Fj | is the number of frequent j-itemsets.
- The overall complexity of this step is $O(\sum_k k|F_{k-1}||F_1|)$.
- The procedure is complete because every frequent k-itemset is comprised of a frequent (k−1)-itemset and another frequent item.
- This approach, however, does not prevent the same candidate itemset from being generated more than once.
- For instance, {Bread,Diaper,Milk} can be generated by merging {Bread, Diaper} with {Milk}, {Bread,Milk} with {Diaper}, or {Diaper,Milk} with {Bread}.

- The generation of duplicate candidates can be avoided if items in a frequent itemset are kept in a lexicographic order (dictionary order) and each frequent itemset is extended with items that appear later in the ordering.
- For example, the itemset {Bread, Diaper} can be augmented with {Milk} since "Bread" and "Diaper" precedes "Milk" in alphabetical order.

Fk−1×Fk−1 Method

- The candidate generation procedure in Apriori merges a pair of frequent (k−1)-itemsets only if their first k−2 items are identical.
- More specifically, the pair f1 = {a1, a2, ・・・, ak−1} and f2 = {b1, b2, ・・・, bk−1}, are merged if they satisfy the following conditions.

  ai = bi (for i = 1, 2, ・・・, k − 2) and ak−1 _= bk−1.
- For example, in Figure 6.8, the frequent itemsets {Bread, Diaper} and {Bread,Milk} are merged to form a candidate 3-itemset {Bread,Diaper,Milk}.



Generating and pruning candidate k-itemsets by merging pairs of frequent (k − 1)-itemsets.

- It is not necessary to merge {Beer,Diaper} with {Diaper,Milk} because the first item in both itemsets are different.
- If {Beer,Diaper,Milk} is a viable candidate, it would have been obtained by merging {Beer,Diaper} with {Beer,Milk} instead.

- This example illustrates both the completeness of the candidate generation procedure and the advantages of using lexicographic ordering to prevent the generation of duplicate candidate itemsets.
- However, since each merging operation involves only a pair of frequent (k−1)-itemsets, candidate pruning is still needed in general to ensure that the remaining k − 2 subsets of each candidate are also frequent.

5.3 Rule Generation

- Each frequent k-itemset, f, can produce up to 2^k-2 association rules, ignoring rules that have empty antecedent or consequent ($\emptyset \longrightarrow f$ or $f \longrightarrow \emptyset$).
- An association rule can be extracted by partitioning the itemset f into two non-empty subsets, l and f − l, such that l =⇒ f − l satisfies the confidence threshold.
- Note that all such rules must have already met the support threshold because they are generated from a frequent itemset.
- **Example 6.2** Suppose f = {1, 2, 3} is a frequent itemset. There are six possible rules(2^3-2=6) that can be generated from this frequent itemset: {1, 2} =⇒ {3}, {1, 3} =⇒ {2}, {2, 3} =⇒ {1}, {1} =⇒ {2, 3}, {2} =⇒ {1, 3} and {3} =⇒ {1, 2}.
- As the support for the rules are identical to the support for the itemset {1, 2, 3}, all the rules must satisfy the minimum support condition.
- The only remaining step during rule generation is to compute the confidence value for each rule.
- Computing the confidence of an association rule does not require additional scans over the transaction data set.
- For example, consider the rule {1, 2} =⇒ {3}, which is generated from the frequent itemset f = {1, 2, 3}. The confidence for this rule is σ({1, 2, 3})/σ({1, 2}). Because {1, 2, 3} is frequent, the anti-monotone property of support ensures that {1, 2} must be frequent too.
- **Theorem 6.2** If a rule l =⇒ f − l does not satisfy the confidence threshold, then any rule l^1 =⇒ f – l^1, where l^1 is a subset of l, must not satisfy the confidence threshold as well.

5.3.1 Confidence – Based Pruning

- The Apriori algorithm uses a level-by-level approach for generating association rules, where each level corresponds to the number of items that belong to the rule consequent.

- Initially, all high-confidence rules that have only a single item in the rule consequent are extracted.
- At the next level, the algorithm uses rules extracted from the previous level to generate new candidate rules.
- A pseudocode for the rule generation step is shown in Algorithms

1: **for** each frequent k-itemset f_k, $k \geq 2$ **do**

2: H1 = {i | i ∈ f_k} {1-item consequents of the rule}

3: **call** ap-genrules(f_k,H_1.)

4: **end for**

- Procedure ap-genrules(fk, Hm)

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(Hm).

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 ≥ minconf **then**

8: **output** the rule $(f_k - h_{m+1}) \Longrightarrow h_{m+1}$.

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**

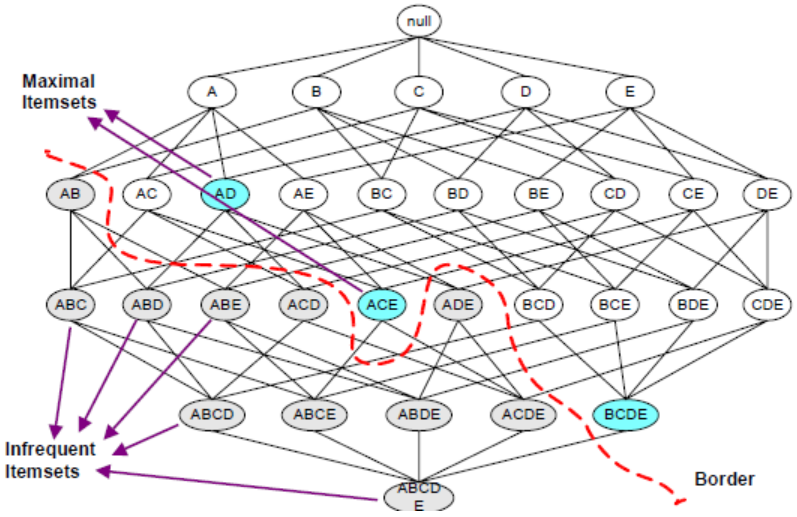
5.4 Compact Representation of Frequent Itemsets

- In practice, the number of frequent itemsets produced from a transaction data set can be very large.
- It will be useful to identify a small representative set of itemsets from which all other frequent itemsets can be derived.
- Two such representation are presented in this section in the form of maximal and closed frequent itemsets.

1. Maximal Frequent Itemsets
2. Closed Frequent Itemsets

5.4.1 Maximal Frequent Itemsets

- A maximal frequent itemset is
- defined as a frequent itemset for which none of its immediate supersets are frequent.



Maximal frequent itemset.

- Consider the itemset lattice shown in above Figure .
- The itemsets in the lattice are divided into two groups, those that are frequent versus those that are infrequent.
- A frequent itemset border, which is represented by a dashed line, is also illustrated in the diagram.
- Every itemset located above the border is frequent while those located below the border (i.e., the shaded nodes) are infrequent.
- Among the itemsets residing near the border, {A,D}, {A,C,E}, and {B,C,D,E} are considered to be maximal frequent itemsets because their immediate supersets are infrequent.
- An itemset such as {A,D} is maximal frequent because all of its immediate supersets, {A,B,D}, {A,C,D}, and {A,D,E} are infrequent.
- In contrast, {A,C} is non-maximal because one of its immediate supersets, {A,C,E}, is frequent.
- Maximal frequent itemsets provide effectively a compact representation of frequent itemsets.
- In other words, it is the smallest set of itemsets from which all other frequent itemsets can be derived.
- For example, the frequent itemsets shown in Figure 6.17 can be divided into two groups:
    - Frequent itemsets that begin with item A and may contain items C, D, or E. This group includes itemsets such as {A}, {A,C}, {A,D}, {A,E}, and {A,C,E}.

- Frequent itemsets that begin with item B, C, D, or E. This group includes itemsets such as {B}, {B,C}, {C,D},{B,C,D,E}, etc.

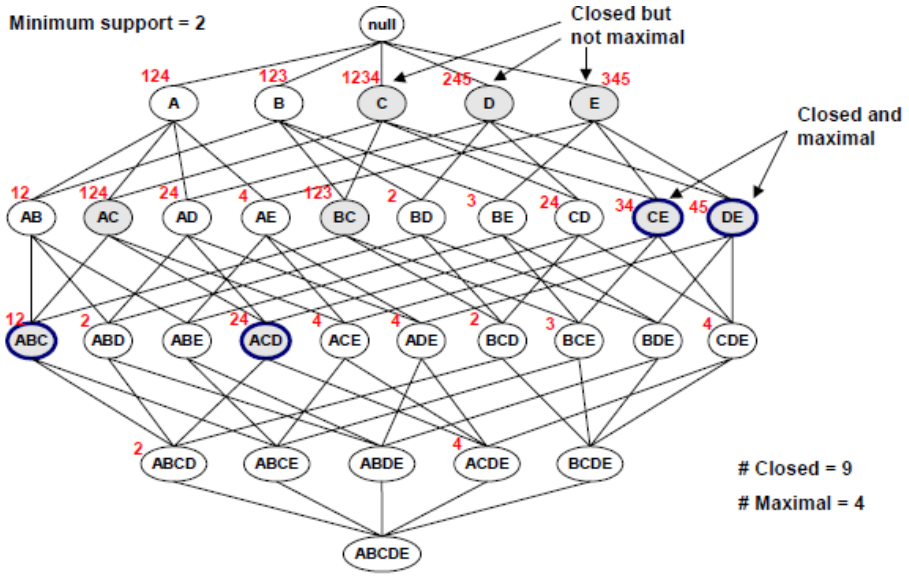
- Frequent itemsets that belong to the first group are subsets of either {A,C,E} or {A,D} while those in the second group are subsets of {B,C,D,E}.
- Hence, the maximal frequent itemsets {A,C,E}, {A,D}, and {B,C,D,E} provide a compact representation of the frequent itemsets shown in above Figure.
- Maximal frequent itemset provides a valuable representation for data sets that can produce very long frequent itemsets as there are exponentially many frequent itemsets in such data.

5.4.2 Closed Frequent Itemsets

- Closed itemsets provide a minimal representation of itemsets without losing their support information.
- **Closed Itemsets :** An itemset X is closed if none of its immediate supersets have exactly the same support count as X.

  or

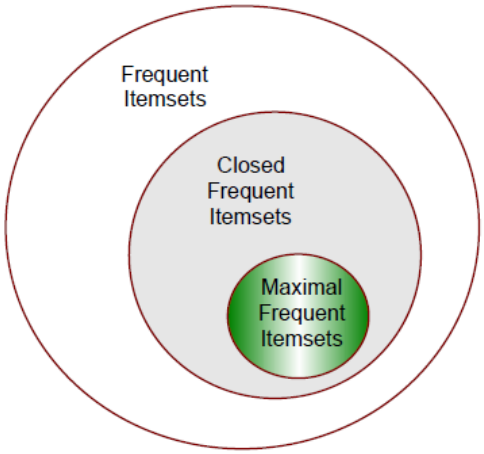
  Put another way, X is not closed if at least one of its immediate supersets has the same support count as X.
- Examples of closed itemsets are shown in below Figure .



TID	Items
1	ABC
2	ABCD
3	BCE
4	ACDE
5	DE

An illustrative example of the closed frequent itemsets (with minimum support count equals to 40%).

- To better illustrate the support count of each itemset, we have associated each node (itemset) in the lattice with a list of their corresponding transaction ids.
- For example, since the node {B,C} is associated with transaction ids 1, 2, and 3, its support count is equal to three.
- From the transactions given in this diagram, notice that every transaction that contains B also contains C.
- Consequently, the support for {B} is identical to {B,C} and {B} should not be considered as a closed itemset.
- Similarly, since C occurs in every transaction that contains both A and D, the itemset {A,D} is not
- closed.
- On the other hand, {B,C} is a closed itemset because it does not have the same support count as any one of its supersets.
- **Closed Frequent Itemsets): A**n itemset X is a closed frequent itemset if it is closed and its support is greater than or equal to minsup.
- In the previous example, assuming that the support threshold is 40%, {B,C} is a closed frequent itemset because its support is 60%.
- The rest of the closed frequent itemsets are indicated by the shaded nodes.
- Algorithms are available to explicitly extract closed frequent itemsets from a given data set.
- Closed frequent itemsets can be used to determine the support counts for all nonclosed frequent itemsets.



Relationships among frequent itemsets, maximal frequent itemsets, and closed frequent itemsets.

5.5 Alternative Methods for Frequent Itemset Generation

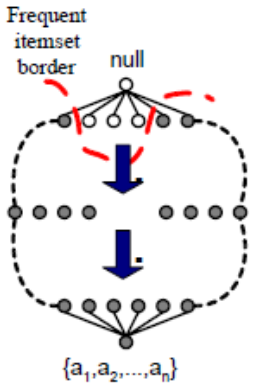
- Apriori is one of the earliest algorithms to have successfully addressed the combinatorial explosion of frequent itemset generation.
- It achieves this by applying the Apriori principle to prune the exponential search space.
- Several alternative methods have been developed to overcome these limitations and improve upon the efficiency of Apriori algorithm.

5.5.1 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 different algorithms dictates how the lattice structure is traversed during the frequent itemset generation process.
- Obviously, some search strategies work better than others, depending on the configuration of frequent itemsets in the lattice.

General-to-Specific versus Specific-to-General:

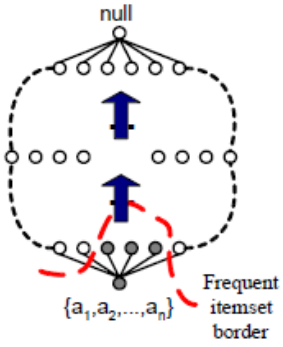
- The Apriori algorithm uses a general-to-specific search strategy, where pairs of frequent itemsets of size k − 1 are merged together to obtain the more specific frequent itemsets of size k.
- During the mining process, the Apriori principle is applied to prune all supersets of infrequent itemsets.
- This general-to-specific search, coupled with support-based pruning, is an effective strategy provided that the length of the maximal frequent itemset is not too long.
- The configuration of frequent itemsets that works best with this strategy is shown in below Figure where the darker nodes represent infrequent itemsets.



General-to-specific

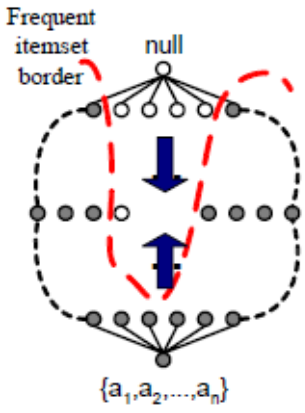
- Alternatively, a specific-to-general search strategy finds the more specific frequent itemsets first before seeking the less specific frequent itemsets.

- This strategy is useful for discovering maximal frequent itemsets in dense transaction data sets, where the frequent itemset border is located near the bottom of the lattice, as shown in below Figure .



Specific-to-general

- During the mining process, the Apriori principle is applied to prune all subsets of maximal frequent itemsets.
- Specifically, if a candidate k-itemset is maximal frequent, we do not have to examine any of its subsets of size k − 1.
- On the other hand, if it is infrequent, we need to check all of its k – 1 subsets in the next iteration.
- Yet another approach is to combine both general-to-specific and specific-to-general search strategies.
- This bidirectional approach may require more space for storing candidate itemsets, but it can help to rapidly identify the frequent itemset border, given the configuration shown in below Figure .

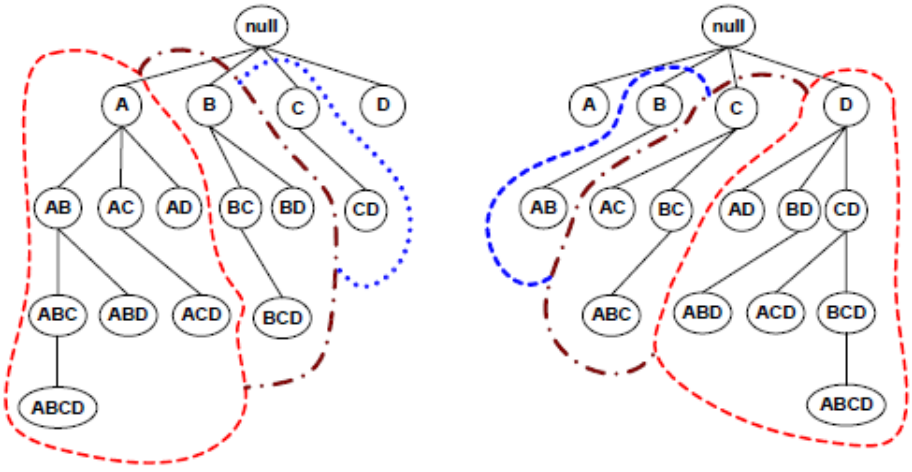


Bidirectional

Equivalent classes:

- Another way of traversal is to first partition the lattice into disjoint groups of nodes (or equivalent classes).

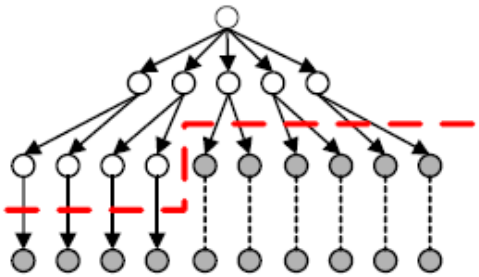
- A frequent itemset generation algorithm seeks for frequent itemsets within a particular equivalent class first before continuing its search to another equivalent class.
- As an example, the level-wise strategy used in Apriori can be considered as partitioning the lattice on the basis of itemset sizes, i.e., the algorithm discovers all frequent 1-itemsets first before proceeding to larger-sized itemsets.
- Equivalent classes 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.
- In the prefix-based approach, the algorithm may search for frequent itemsets starting with the prefix A before looking for those starting with prefix B, C, and so on.
- Both prefix-based and suffix-based equivalent classes can be demonstrated using a set enumeration tree, as shown in Figure .



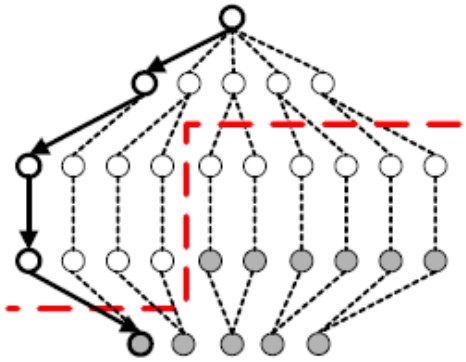
Prefix Tree Suffix tree

Breadth-first versus Depth-first:

- The Apriori algorithm traverses the lattice in a level-wise (breadth-first) manner, as shown in Figure .



Breadth first



Depth first

- It first discovers all the size-1 frequent itemsets at level 1, followed by all the size-2 frequent itemsets at level 2, and so on, until no frequent itemsets are generated at a particular level.
- The itemset lattice can also be traversed in a depth-first manner.
- One may start from, say node {A}, and count its support to determine whether it is frequent.
- If so, we can keep expanding it to the next level of nodes, i.e., {A,B}, {A,B,C}, and so on, until we reach an infrequent node, say {A,B,C,D}.
- We then backtrack to another branch, say {A,B,C,E}, and continue our search from there.
- This approach is often used by algorithms designed to efficiently discover maximal frequent itemsets.
- By using the depth-first approach, we may arrive at the frequent itemset border more quickly than using a breadthfirst approach.
- Once a maximal frequent itemset is found, substantial pruning can be performed on its subsets.
- For example, if an itemset such as {B,C,D,E} is maximal frequent, then the rest of the nodes in the subtrees rooted at B, C, D, and E can be pruned because they are not maximal frequent.
- On the other hand, if {A,B,C} is maximal frequent, only subsets of this itemset (e.g., {A,C} and {B,C}) are not maximal frequent.
- The depth-first approach also allows a different kind of pruning based on the support of itemsets.
- To illustrate, suppose the support for {A,B,C} is identical to the support for its parent, {A,B}.
- In this case, the entire subtree rooted at {A,B} can be pruned because it cannot produce a maximal frequent itemset.

5.5.2 Representation of Transaction Data Set:

- There are many ways to represent a transaction data set.

- The representation may affect the I/O costs incurred when computing the support of candidate itemsets.

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				

Horizontal and vertical data format.

- 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 identifiers (tid-list) for each item. Such representation is known as the vertical data layout.
- The support of each candidate itemset can be counted by intersecting the tid-lists of their subsets.
- The length of the tid-lists would shrink as we progress to larger sized itemsets.
- One problem with this approach is that the initial size of the tid-lists could be too large to fit into main memory, thus requiring rather sophisticated data compression techniques.

5.6 FP-growth Algorithm

- Recently, an interesting algorithm called FP-growth was proposed that takes a radically different approach to discover frequent itemsets.
- The algorithm does not subscribe to the generate-and-count paradigm of Apriori.
- Instead, it encodes the data set Using a compressed representation of the database using an FP-tree
- Once an FP-tree has been constructed, it uses a recursive divide-and-conquer approach to mine the frequent itemsets

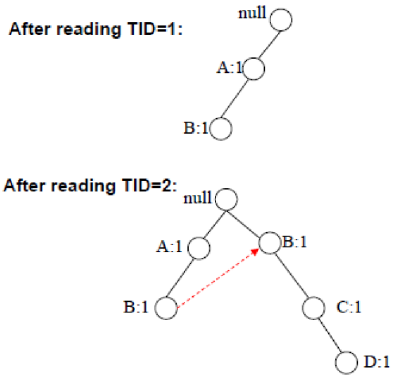
5.6.1 FP-tree Construction

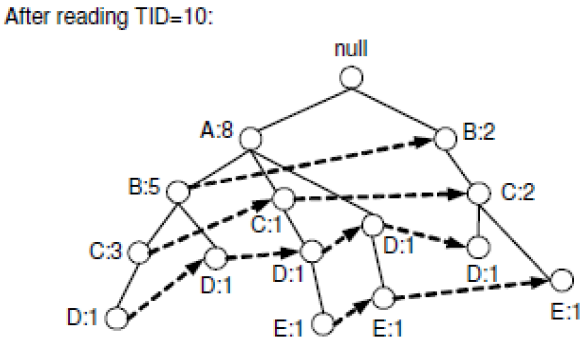
- An FP-tree is a compressed representation of a data set.
- It is constructed by reading the transactions of a data set and mapping each of them onto a path in the FPtree.
- As different transactions may share several items in common, the paths may be overlapping.

- The degree of overlapping among the paths would determine the amount of compression achieved by the FP-tree.
- In many cases, the size of an FP-tree may be small enough to fit into main memory, thus allowing us to extract frequent itemsets directly from the tree instead of making multiple passes over the data.

Example :

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	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

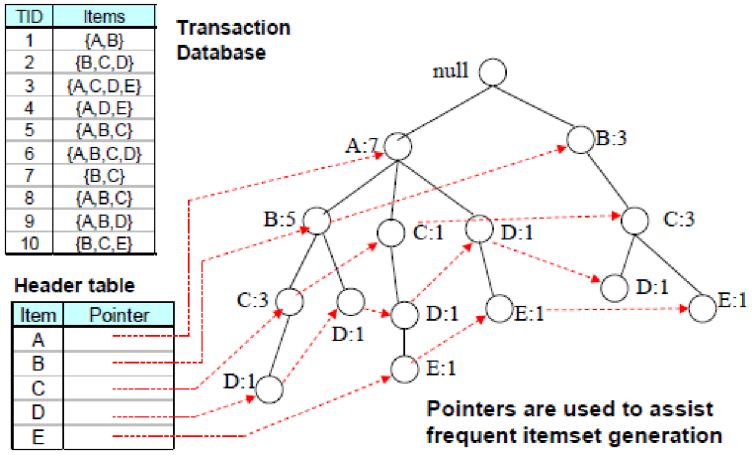




Construction of an FP-tree.

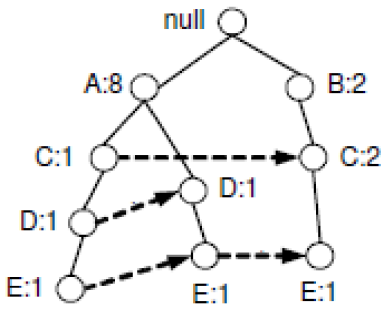
- Each node in the tree (except for the root) encodes information about the item label along with the number of transactions mapped onto the given path.
- If many of the transactions contain similar items, then the size of the induced FP-tree is considerably smaller than the size of the data set.
- The best-case scenario would be that the data set contains the same set of items for all transactions.
- The resulting FP-tree contains only a single branch of nodes.
- The worse-case scenario happens when each transaction contains a unique set of items.

- During tree construction, the FP-tree structure also stores an access mechanism for reaching every individual occurrence of each frequent item used to construct the tree.
- In the above example, there are five such linked lists, one for each item A, B, C, D, and E.
- The linked lists will be used for fast access of the corresponding paths during frequent itemset generation.



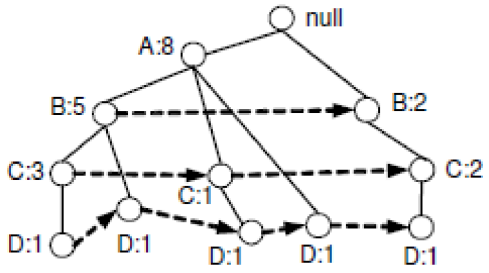
5.6.2 Generating Frequent Itemsets from an FP-tree

- The algorithm used for generating frequent itemsets from an FP-tree is known as FP-growth. FP-growth examines the FP-tree in a bottom-up fashion.
- For example, given the FP-tree the algorithm looks for frequent itemsets ending in E first, before finding frequent itemsets ending in D, followed by C, B, and finally A.
- Furthermore, since every transaction is mapped onto a path in the FP-tree, frequent itemsets ending with a particular item, say E, can be derived by examining only the paths involving node E.
- These paths can be accessed rapidly using the linked list associated with item E. The corresponding paths are shown in below Figure .



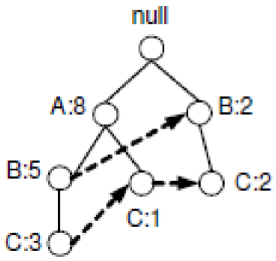
Paths containing node E

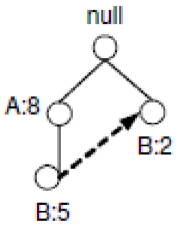
- Assuming the support threshold is 20%, the FP-growth algorithm will process these paths and generates the following frequent itemsets: {E}, {D,E}, {A,D,E}, {C,E}, {A,C,E}, and {A,E}.
- The details of how exactly the paths are processed will be explained later.
- Having discovered the frequent itemsets ending in E, the algorithm proceeds to look for frequent itemsets ending in D by following the linked list associated with item D.
- The corresponding paths to be processed by the FP-growth algorithm are shown in Figure 6.28(b).



Paths containing node D

- After processing these paths, the following frequent itemsets are generated: {D}, {C,D}, {B,C,D}, {A,C,D}, {A,B,C,D}, {B,D}, {A,B,D}, and {A,D}.
- This process will continue as FP-growth seeks for frequent itemsets ending in C, B, and finally A. The corresponding paths for these items are shown in Figures





Paths containing node C *Paths containing node B* *Paths containing node A*

- Note that this strategy of generating frequent itemsets ending with a particular item label can be considered as an implementation of the suffix-based equivalent class approach.
- How does FP-growth discover all the frequent itemsets ending with a particular suffix? The answer is, FP-growth employs a divide-and-conquer strategy to split the problem into smaller subproblems.
- For example, suppose we are interested in finding all the frequent itemsets ending in E.
- To do this, we may perform the following tasks:

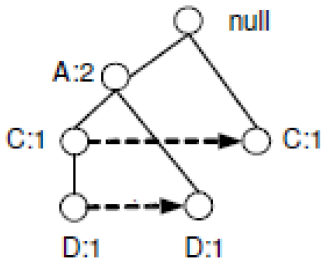
Task 1: Check whether the itemset {E} is frequent.

Task 2: If it is hrequent, we consider the subproblem of finding itemsets ending in DE followed by CE, BE and AE. . This can be achieved by examining all the paths that contain the subpath D −→ E, C □ E, B□ E and A□ E respectively .

Task 3: Each of these subproblems are further decomposed into smaller subproblems .

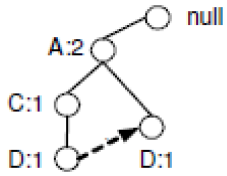
Task 4: By merging the solution set for each subproblem, we would solve the problem of finding frequent itemsets ending in E.

- This divide-and conquer approach is essentially the key strategy used by the FP-growth algorithm.
- The process continues until all the subproblems involve only a single item (Task 1).
- If the support count for this item is greater than the support threshold, then the item label is appended to the current suffix of the itemset.
- The transformation from a prefix path to a conditional FP-tree is carried out in the following way:
  - Update the frequency counts of all the nodes along the prefix path.
  - The count must be updated because the initial prefix path may include transactions that do not contain the item E.
  - Since we are only interested in itemsets ending in E, the count of each item along the prefix path must be adjusted so that they are identical to the count for node E.
  - For example, the right-most prefix path null −→ B:2 −→ C:2 will be updated to null −→ B:1 −→ C:1 because there is only one transaction containing items B, C, and E (while the other transaction mapped onto the same path contains {B,C} but not E, and thus should be ignored).
- Truncate the prefix paths by removing the nodes for E.
- The paths can be truncated because the subproblems of finding frequent itemsets ending in DE, CE, BE or AE no longer need information from the node E.
- After the frequency counts along the prefix paths have been updated, some items may not be frequent anymore and thus can be safely ignored from further analysis (as far as our new subproblem is concerned).
- For example, item B is discarded because the support for B in the subproblem of finding frequent itemsets ending in E (10%) is less than the support threshold (20%).
- The conditional FP-tree for E is shown in below Figure



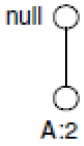
Conditional FP-tree for E

- The tree looks quite different than the original prefix paths because the frequency counts have been updated while items B and E are eliminated.
- FP-growth uses this conditional FP-tree to solve the subproblems of finding frequent itemsets ending in DE, CE, and AE.
- To determine the frequent itemsets ending in DE, the prefix paths for DE is gathered from the conditional FP-tree for E.



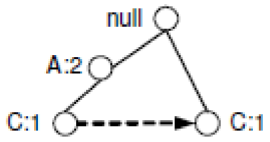
Prefix paths ending in DE

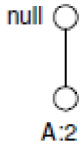
- These prefix paths are rapidly accessed using the linked list for D, starting from the node on the left-most path of the conditional FP-tree.
- The frequency counts associated with the nodes D are added to obtain the support count for DE.
- Since the support is greater than the support threshold, {D,E} is declared as a frequent itemset.
- Next, the algorithm will proceed to decompose the problem of finding frequent itemsets ending in DE into smaller subproblems.
- To do this, it must first construct the conditional FP-tree for DE.
- After updating the support counts and removing the infrequent item (C), the conditional FP-tree for DE is shown in Figure .



Conditional FP-tree for DE

- Since the subproblem contains only a single item, A, whose support count is greater than minsup, the algorithm returns the label A, which will be appended to the suffix DE to obtain the frequent itemset {A,D,E}.
- To determine the frequent itemsets ending in CE, the prefix paths for CE is gathered from the conditional FP-tree for E.
- Again, these prefix paths are rapidly accessed using the corresponding linked list for node C. Since the total frequency count for C is greater than minsup, the itemset {C,E} is declared as frequent.
- Next, the algorithm proceeds to solve the subproblem of finding frequent itemsets ending in CE.
- A conditional FP-tree for CE will be constructed, as shown in Figure .





Prefix paths ending in CE *Conditional FP-tree for CE*

- Since the conditional FP-tree contains only a single item, we only need to check whether the support for A is frequent.
- Since it is frequent, the label A is returned and appended to the suffix CE to obtain the frequent itemset {A,C,E}.
- Finally, to determine the frequent itemsets ending in AE, the prefix paths for AE is gathered from the conditional FP-tree for E. Since the frequency count for A is the same as minsup, {A,E} is declared as a frequent itemset.
- The conditional FP-tree for AE contains only the root node. Thus, no processing is needed.
- The above procedure illustrates the divide-and-conquer approach used in the FPgrowth algorithm.
- At each recursive step, a conditional FP-tree is constructed by updating the frequency counts along the prefix paths and removing all infrequent items.
- By removing the infrequent items, no unnecessary candidates will be generated by the FP-growth algorithm.
- In general, FP-growth is an interesting algorithm because it illustrates how a compact representation of the transaction data set helps to efficiently generate frequent itemsets.
- In addition, for certain transaction data sets, FP-growth outperforms the standard Apriori algorithm by several orders of magnitude.
- The run-time performance of FP-growth depends on the compaction factor of the data set.

- If the resulting conditional FP-trees are very bushy (in the worst case, a full prefix tree), then the performance of the algorithm degrades significantly because it has to generate a large number of subproblems and merges the results returned by each subproblem.