Unit-2

Suppose you are a sales manager at *AllElectronics*, and you are talking to a customer who recently bought a PC and a digital camera from the store. What should you recommend to her next?

Information about which products are frequently purchased by your customers following their purchases of a PC and a digital camera in sequence would be very helpful in making your recommendation.

Frequent patterns are patterns that appear frequently in a data set. For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set is a *frequent itemset*. A subsequence, such as buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a shopping history database, is a (*frequent*) *sequential pattern*.

A *substructure* can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (*frequent*) *structured pattern*.

Finding frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data.

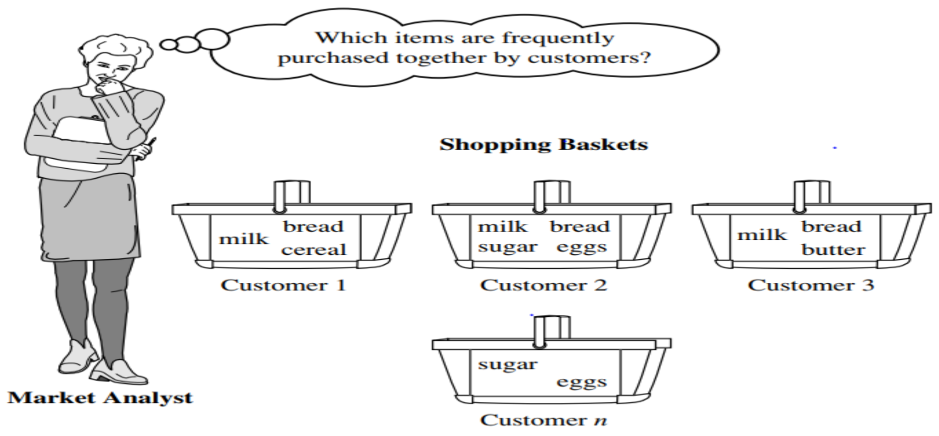
Market Basket Analysis: A Motivating Example

Frequent itemset mining leads to the discovery of associations and correlations among items in large transactional or relational data sets. With massive amounts of data continuously being collected and stored, many industries are becoming interested in mining such patterns from their databases.

A typical example of frequent itemset mining is market basket analysis. This process analyzes customer buying habits by finding associations between the different items that customers place in their "shopping baskets". The discovery of these associations can help retailers develop marketing strategies by gaining insight into which items are frequently purchased together by customers. For instance, if customers are buying milk, how likely are they to also buy bread.

Market basket analysis: Suppose, as manager of an *AllElectronics* branch, you would like to learn more about the buying habits of your customers. market basket analysis may be performed on the retail data of customer transactions at your store. You can then use the results to plan marketing or advertising strategies, or in the design of a new catalog.

For instance, market basket analysis may help you design different store layouts. In one strategy, items that are frequently purchased together can be placed in proximity to further encourage the combined sale of such items. If customers who purchase computers also tend to buy antivirus software at the same time, then placing the hardware display close to the software display may help increase the sales of both items.



Each basket can then be represented by a Boolean vector of values assigned to these variables. The Boolean vectors can be analyzed for buying patterns that reflect items that are frequently associated or purchased together. These patterns can be represented in the form of *association rules*.

computer => antivirus software *[s*upport = *2%,*confidence = *60%].*

support and confidence are two measures of rule interestingness. They respectively reflect the usefulness and certainty of discovered rules.

A support of 2% for Rule means that 2% of all the transactions under analysis show that computer and antivirus software are purchased together.

A confidence of 60% means that 60% of the customers who purchased a computer also bought the software. Typically, association rules are considered interesting if they satisfy both a minimum support threshold and a minimum confidence threshold.

Frequent Itemsets, Closed Itemsets, and Association Rules

Let I =*{I*1, *I*2,……, *Im}* be an itemset. Let *D*, the task-relevant data, be a set of database transactions where each transaction *T* is a nonempty itemset such that $T \subseteq$ I.

Each transaction is associated with an identifier, called a *TID*. Let *A* be a set of items. A transaction *T* is said to contain *A* if $A \subseteq T$.

An association rule is an implication of the form *A* => *B*, where $A \subset$ I, $B \subset$ I, $A \neq B$ and $A \neq \varphi$, $B \neq \varphi$. The rule *A*=> *B* holds in the transaction set *D* with support *s*, where *s* is the percentage of transactions in *D* that contain *AU B* (i.e., the *union* of sets *A* and *B* say, or, both *A* and *B*). This is taken to be the probability, *P(AUB).*

The rule *A* => *B* has confidence *c* in the transaction set *D*, where *c* is the percentage of transactions in *D* containing *A* that also contain *B*. This is taken to be the conditional probability, *P(B/A)*.

$$support(A \Rightarrow B) = P(A \cup B)$$
$$confidence(A \Rightarrow B) = P(B|A).$$

Rules that satisfy both a minimum support threshold (min_sup) and a minimum confidence threshold (min_conf) are called *strong*.

A set of items is referred to as an itemset. An itemset that contains *k* items is a *k*-itemset. The set *{computer, antivirus software}* is a 2-itemset.

The occurrence frequency of an itemset is the number of transactions that contain the itemset. This is also known as frequency, support count, or count of the itemset.

$$confidence(A \Rightarrow B) = P(B|A) = \frac{support(A \cup B)}{support(A)} = \frac{support\_count(A \cup B)}{support\_count(A)}$$

the confidence of rule A => B can be easily derived from the support counts of A and A *U* B. That is, once the support counts of A, B, and A *U* B are found, it is straightforward to derive the corresponding association rules A => B and B => A and check whether they are strong.

Association rule mining can be viewed as a two-step process:

1. *Find all frequent itemsets:* each of these itemsets will occur at least as frequently as a predetermined minimum support count, *min sup*.

2. *Generate strong association rules from the frequent itemsets:* these rules must satisfy minimum support and minimum confidence

An itemset X is *closed* in a data set D if there exists no proper super-itemset Y, such that Y has the same support_count as X in D. An itemset X is a *closed frequent itemset* in set D

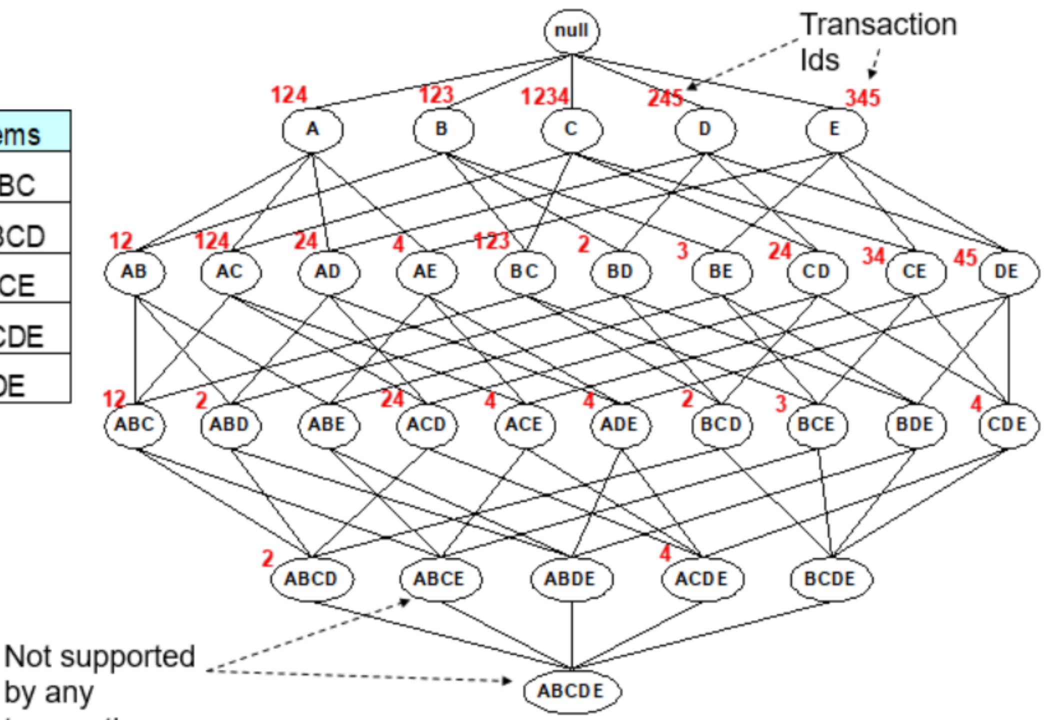
An itemset X is a *maximal frequent itemset* (or *max-itemset*) in a data set D if X is frequent, and there exists no super-itemset Y such that X $\subset$Y and Y is frequent in D

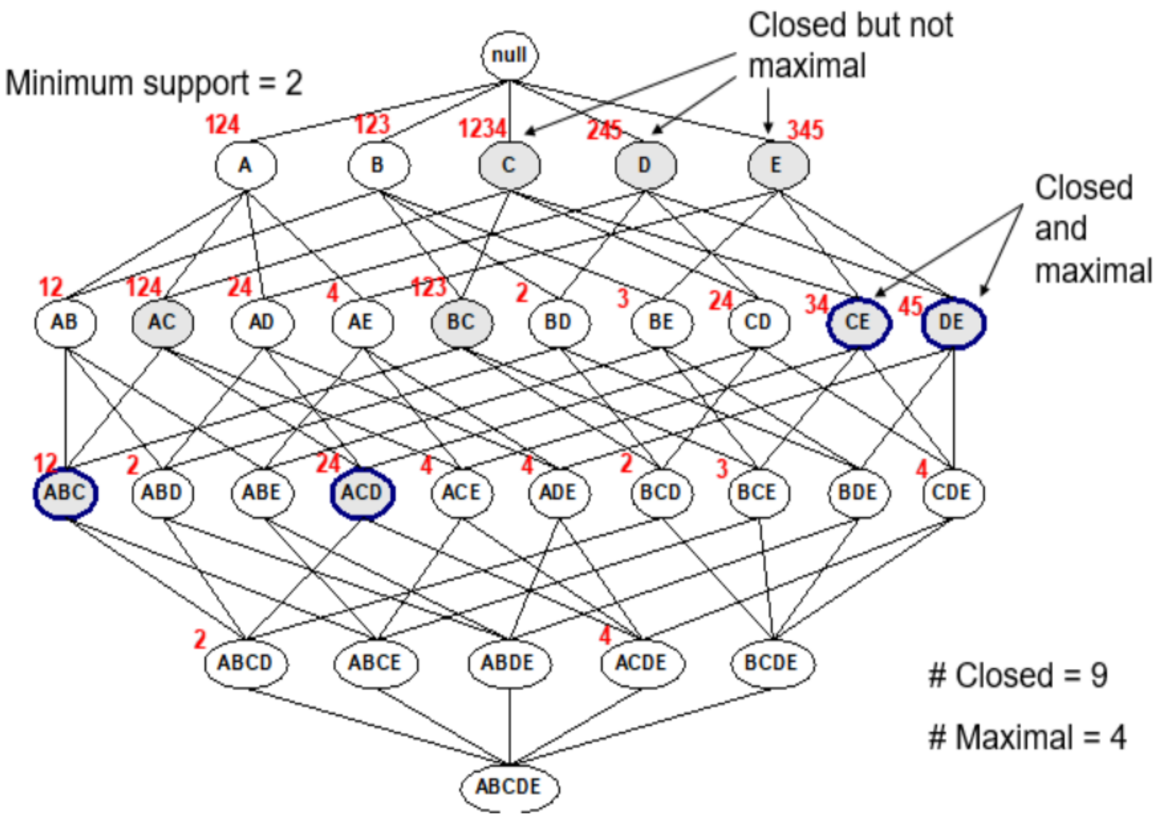
Suppose that a transaction database has only two transactions: {<*a*1, *a*2,……., *a*100> ,<*a*1, *a*2,…….., *a*50>}. Let the minimum support count threshold be *min sup* D 1. We find two closed frequent itemsets and their support counts, that is, *C* = {{*a*1, *a*2,*:::*, *a*100} : 1; {*a*1, *a*2,*:::*, *a*50} : 2}.

There is only one maximal frequent itemset: *M* D {{*a*1, *a*2,*:::*, *a*100} : 1}.

Maximal vs Closed Itemsets

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





Apriori Algorithm: Finding Frequent Itemsets by Confined Candidate Generation:

Apriori is a seminal algorithm proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses prior knowledge of frequent itemset properties.

Apriori employs an iterative approach known as a level-wise search, where k-itemsets are used to explore *(*k + 1*)*-itemsets.

First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. The resulting set is denoted by L1.

Next, L1 is used to find L2, the set of frequent 2-itemsets, which is used to find L3, and so on, until no more frequent k-itemsets can be found. The finding of each Lk requires one full scan of the database.

To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the Apriori property is used to reduce the search space.

Apriori property: All nonempty subsets of a frequent itemset must also be frequent

The Apriori property is based on the following observation. By definition, if an itemset I does not satisfy the minimum support threshold, min sup, then I is not frequent, that is, P*(*I*)* < min sup. If an item A is added to the itemset I, then the resulting itemset (i.e., I *U* A) cannot occur more frequently than I. Therefore, I *U* A is not frequent either, that is, P*(*I *U* A*)* < min sup.

This property belongs to a special category of properties called antimonotonicity in the sense that *if a set cannot pass a test, all of its supersets will fail the same test as well*.

let us look at how Lk-1 is used to find Lk for k ≥ 2. A two-step process is followed, consisting of *join* and *prune* actions

The join step: To find *Lk*, a set of candidate *k*-itemsets is generated by joining *Lk*-1 with itself. This set of candidates is denoted *Ck*. Let *l*1 and *l*2 be itemsets in *Lk*-1. The notation *li*[*j*] refers to the *j*th item in *li* (e.g., *l*1[*k* - 2] refers to the second to the last item in *l*1).

For efficient implementation, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. For the *(k* – 1*)*-itemset, *li,* this means that the items are sorted such that *li*[1] < *li*[2] < ··· < *li*[*k* - 1]. The join, *Lk*-1 ⋈ *Lk*-1, is performed, where members of *Lk*-1 are joinable if their first *(k* – 2*)* items are in common.

That is, members *l*1 and *l*2 of *Lk*-1 are joined if (*l*1[1] = *l*2[1]*)* ^ *(l*1[2] = *l*2[2]*)* ^ ··· ^ *(l*1[*k* - 2] = *l*2[*k* - 2])^*(l*1[*k* - 1] < *l*2[*k* - 1]*)*.

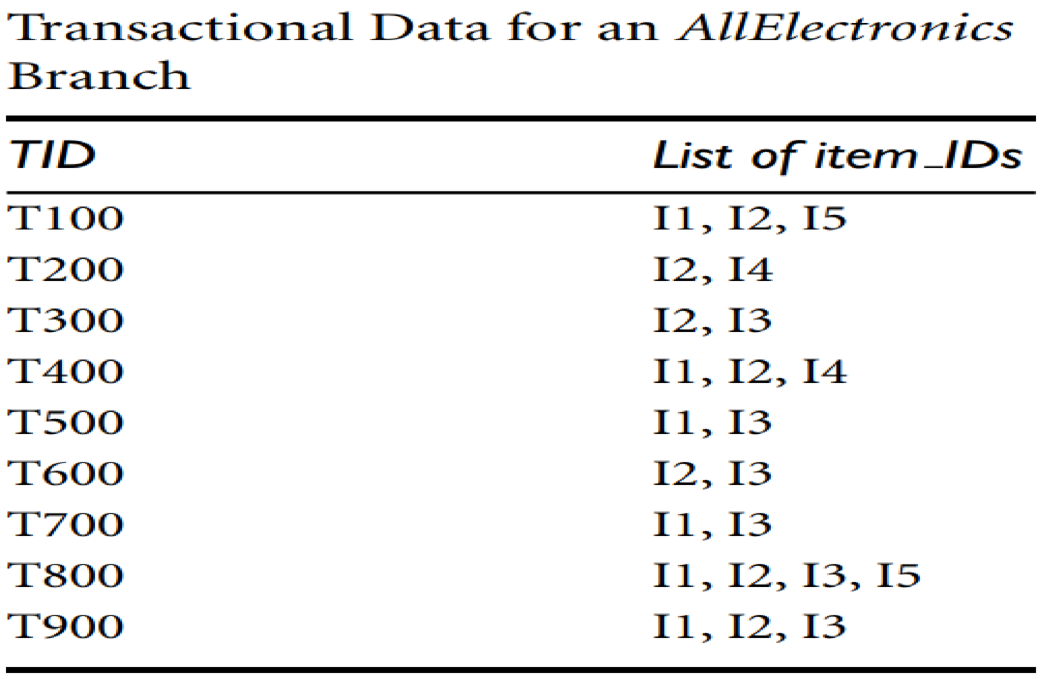
The condition *l*1[*k* - 1] < *l*2[*k* - 1] simply ensures that no duplicates are generated. The resulting itemset formed by joining *l*1 and *l*2 is *{l*1[1], *l*1[2],…….., *l*1[*k* - 2], *l*1[*k* - 1], *l*2[*k* - 1]*}*.

2. **The prune step**: *Ck* is a superset of *Lk*, that is, its members may or may not be frequent, but all of the frequent *k*-itemsets are included in *Ck*.

A database scan to determine the count of each candidate in *Ck* would result in the determination of *Lk* (i.e., all candidates having a count no less than the minimum support count are frequent by definition, and therefore belong to *Lk*).

Ck, however, can be huge, and so this could involve heavy computation. To reduce the size of *Ck*, the Apriori property is used as follows. Any (*k* – 1)-itemset that is not frequent cannot be a subset of a frequent *k*-itemset. Hence, if any (*k* – 1)-subset of a candidate *k*-itemset is not in *Lk*-1, then the candidate cannot be frequent either and so can be removed from *Ck*.

Example:

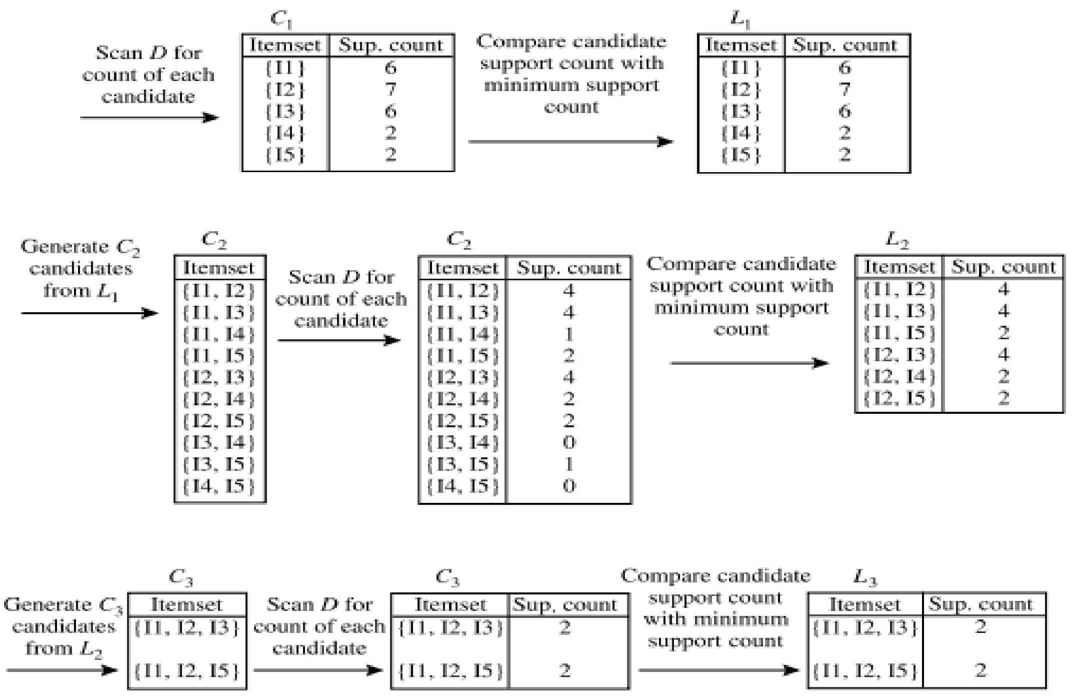


Transactional Data for an *AllElectronics* Branch

TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

1. In the first iteration of the algorithm, each item is a member of the set of candidate 1-itemsets, C1. The algorithm simply scans all of the transactions to count the number of occurrences of each item.
2. Suppose that the minimum support count required is 2, that is, min_sup= 2. The set of frequent 1-itemsets, L1, can then be determined. It consists of the candidate 1-itemsets satisfying minimum support. In our example, all of the candidates in C1 satisfy minimum support.
3. To discover the set of frequent 2-itemsets, L2, the algorithm uses the join L1 $\bowtie$ L1 to generate a candidate set of 2-itemsets, C2 Note that no candidates are removed from C2 during the prune step because each subset of the candidates is also frequent.
4. Next, the transactions in D are scanned and the support count of each candidate itemset in C2 is accumulated.

5. The set of frequent 2-itemsets, L2, is then determined, consisting of those candidate 2-itemsets in C2 having minimum support.
6. The generation of the set of the candidate 3-itemsets, C3, from the join step, we first get C3 = L2 ⋈ L2 = *{*{I1, I2, I3*}*, {I1, I2, I5*}*, {I1, I3, I5*}*, {I2, I3, I4*}*, {I2, I3, I5*}*, {I2, I4, I5*}}*. Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we can determine that the four latter candidates cannot possibly be frequent. We therefore remove them from C3, thereby saving the effort of unnecessarily obtaining their counts during the subsequent scan of D to determine L3.


Scan D for count of each candidate
Compare candidate support count with minimum support count
Generate C_2 candidates from L_1
Scan D for count of each candidate
Compare candidate support count with minimum support count
Generate C_3 candidates from L_2
Scan D for count of each candidate
Compare candidate support count with minimum support count

C_1

Itemset	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

L_1

Itemset	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

C_2

Itemset
{I1, I2}
{I1, I3}
{I1, I4}
{I1, I5}
{I2, I3}
{I2, I4}
{I2, I5}
{I3, I4}
{I3, I5}
{I4, I5}

C_2

Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I4}	1
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2
{I3, I4}	0
{I3, I5}	1
{I4, I5}	0

L_2

Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2

C_3

Itemset
{I1, I2, I3}
{I1, I2, I5}

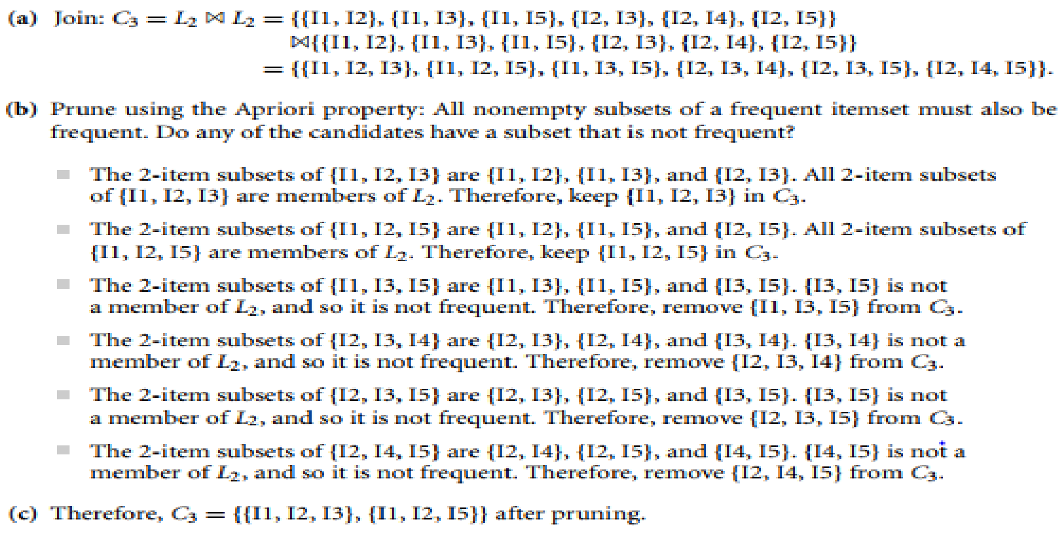
C_3

Itemset	Sup. count
{I1, I2, I3}	2
{I1, I2, I5}	2

L_3

Itemset	Sup. count
{I1, I2, I3}	2
{I1, I2, I5}	2

7. The transactions in D are scanned to determine L3, consisting of those candidate 3-itemsets in C3 having minimum support.
8. The algorithm uses L3 ⋈ L3 to generate a candidate set of 4-itemsets, C4. Although the join results in *{*{I1, I2, I3, I5*}}*, itemset{I1, I2, I3, I5*}* is pruned because its subset {I2, I3, I5*}* is not frequent. Thus, C4 = φ, and the algorithm terminates, having found



(a) Join: $C_3 = L_2 \bowtie L_2$ = {{I1, I2}, {I1, I3}, {I1, I5}, {I2, I3}, {I2, I4}, {I2, I5}}
⋈{{I1, I2}, {I1, I3}, {I1, I5}, {I2, I3}, {I2, I4}, {I2, I5}}
= {{I1, I2, I3}, {I1, I2, I5}, {I1, I3, I5}, {I2, I3, I4}, {I2, I3, I5}, {I2, I4, I5}}.

(b) Prune using the Apriori property: All nonempty subsets of a frequent itemset must also be frequent. Do any of the candidates have a subset that is not frequent?

- The 2-item subsets of {I1, I2, I3} are {I1, I2}, {I1, I3}, and {I2, I3}. All 2-item subsets of {I1, I2, I3} are members of L_2. Therefore, keep {I1, I2, I3} in C_3.
- The 2-item subsets of {I1, I2, I5} are {I1, I2}, {I1, I5}, and {I2, I5}. All 2-item subsets of {I1, I2, I5} are members of L_2. Therefore, keep {I1, I2, I5} in C_3.
- The 2-item subsets of {I1, I3, I5} are {I1, I3}, {I1, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I1, I3, I5} from C_3.
- The 2-item subsets of {I2, I3, I4} are {I2, I3}, {I2, I4}, and {I3, I4}. {I3, I4} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I4} from C_3.
- The 2-item subsets of {I2, I3, I5} are {I2, I3}, {I2, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I5} from C_3.
- The 2-item subsets of {I2, I4, I5} are {I2, I4}, {I2, I5}, and {I4, I5}. {I4, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I4, I5} from C_3.

(c) Therefore, C_3 = {{I1, I2, I3}, {I1, I2, I5}} after pruning.

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

- D, a database of transactions;
- *min_sup*, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

```
(1)   $L_1$ = find_frequent_1-itemsets(D);
(2)   for ($k = 2; L_{k-1} \neq \phi; k{+}{+}$) {
(3)       $C_k$ = apriori_gen($L_{k-1}$);
(4)       for each transaction $t \in D$ { // scan $D$ for counts
(5)           $C_t$ = subset($C_k$, $t$); // get the subsets of $t$ that are candidates
(6)           for each candidate $c \in C_t$
(7)               c.count++;
(8)       }
(9)       $L_k = \{c \in C_k | c.count \geq min\_sup\}$
(10)  }
(11)  return $L = \cup_k L_k$;

procedure apriori_gen($L_{k-1}$:frequent $(k-1)$-itemsets)
(1)   for each itemset $l_1 \in L_{k-1}$
(2)       for each itemset $l_2 \in L_{k-1}$
(3)           if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2])$
                  $\wedge ... \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$ then {
(4)               $c = l_1 \bowtie l_2$; // join step: generate candidates
(5)               if has_infrequent_subset($c$, $L_{k-1}$) then
(6)                   delete $c$; // prune step: remove unfruitful candidate
(7)               else add $c$ to $C_k$;
(8)           }
(9)   return $C_k$;

procedure has_infrequent_subset($c$: candidate $k$-itemset;
          $L_{k-1}$: frequent $(k-1)$-itemsets); // use prior knowledge
(1)   for each $(k-1)$-subset $s$ of $c$
(2)       if $s \notin L_{k-1}$ then
(3)           return TRUE;
(4)   return FALSE;
```

Generating Association Rules from Frequent Itemsets

Once the frequent itemsets from transactions in a database *D* have been found, it is straight forward to generate strong association rules from them.

Confidence(A=>B)=P(B/A) = support_count(A U B)/support_count(A)

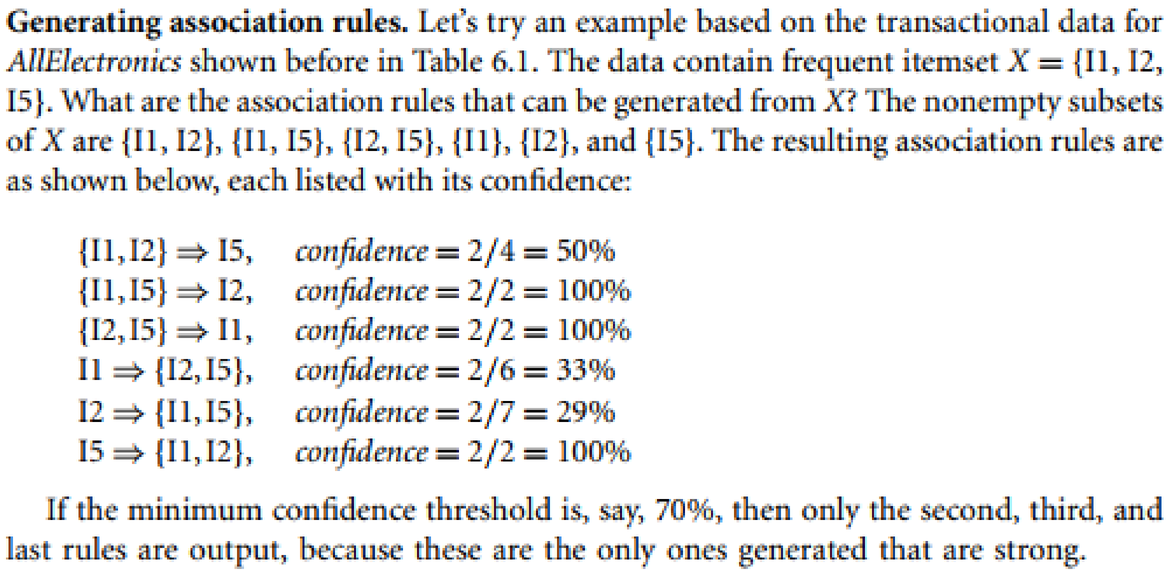
The conditional probability is expressed in terms of itemset support_count, where *support_count(A U B)* is the number of transactions containing the itemsets *A U B*, and *support count(A)* is the number of transactions containing the itemset *A*.

Based on this equation, association rules can be generated as follows: For each frequent itemset *l*, generate all nonempty subsets of *l*. For every nonempty subset *s* of *l*, output the rule

"*s => (l – s)*" if *support_count(l)/support_count(s)* ≥*min_conf*,

where *min_conf* is the minimum confidence threshold.

Because the rules are generated from frequent itemsets, each one automatically satisfies the minimum support.



Generating association rules. Let's try an example based on the transactional data for *AllElectronics* shown before in Table 6.1. The data contain frequent itemset $X = \{I1, I2, I5\}$. What are the association rules that can be generated from X? The nonempty subsets of X are $\{I1, I2\}$, $\{I1, I5\}$, $\{I2, I5\}$, $\{I1\}$, $\{I2\}$, and $\{I5\}$. The resulting association rules are as shown below, each listed with its confidence:

$$\begin{aligned}
&\{I1, I2\} \Rightarrow I5, && \mathit{confidence} = 2/4 = 50\% \\
&\{I1, I5\} \Rightarrow I2, && \mathit{confidence} = 2/2 = 100\% \\
&\{I2, I5\} \Rightarrow I1, && \mathit{confidence} = 2/2 = 100\% \\
&I1 \Rightarrow \{I2, I5\}, && \mathit{confidence} = 2/6 = 33\% \\
&I2 \Rightarrow \{I1, I5\}, && \mathit{confidence} = 2/7 = 29\% \\
&I5 \Rightarrow \{I1, I2\}, && \mathit{confidence} = 2/2 = 100\%
\end{aligned}$$

If the minimum confidence threshold is, say, 70%, then only the second, third, and last rules are output, because these are the only ones generated that are strong.

Improving the Efficiency of Apriori:

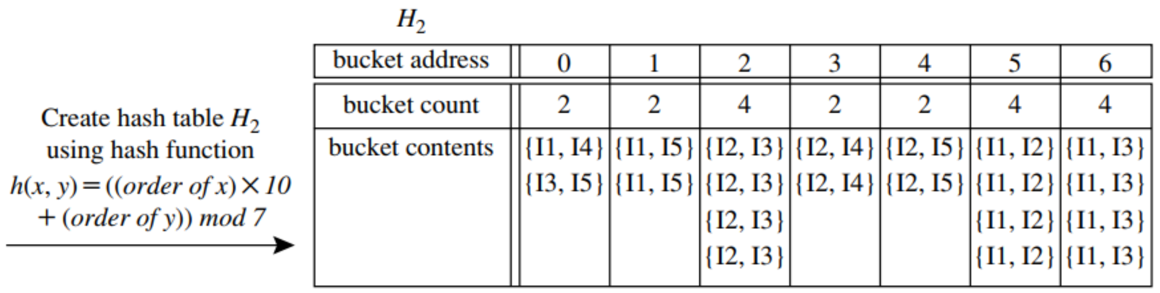
Many variations of the Apriori algorithm have been proposed that focus on improving the efficiency of the original algorithm. Several of these variations are summarized as follows:

Techniques for improving the efficiency of Apriori Algorithm:

1. Hash-based technique
2. Transaction reduction
3. Partitioning
4. Sampling

5. Dynamic itemset counting

1. **Hash based Technique** (hashing itemsets into corresponding buckets): A hash-based technique can be used to reduce the size of the candidate *k*-itemsets, *Ck*, for *k* > 1. For example, when scanning each transaction in the database to generate the frequent 1-itemsets, *L*1, we can generate all the 2-itemsets for each transaction, hash (i.e., map) them into the different *buckets* of a *hash table* structure, and increase the corresponding bucket counts. A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set. Such a hash-based technique may substantially reduce the number of candidate *k*-itemsets examined.



Create hash table H_2 using hash function $h(x, y) = ((order\ of\ x) \times 10 + (order\ of\ y))\ mod\ 7$

H_2

bucket address	0	1	2	3	4	5	6
bucket count	2	2	4	2	2	4	4
bucket contents	{I1, I4} {I3, I5}	{I1, I5} {I1, I5}	{I2, I3} {I2, I3} {I2, I3} {I2, I3}	{I2, I4} {I2, I4}	{I2, I5} {I2, I5}	{I1, I2} {I1, I2} {I1, I2} {I1, I2}	{I1, I3} {I1, I3} {I1, I3} {I1, I3}

Hash table, H_2, for candidate 2-itemsets. This hash table was generated by scanning Table 6.1's transactions while determining L_1. If the minimum support count is, say, 3, then the itemsets in buckets 0, 1, 3, and 4 cannot be frequent and so they should not be included in C_2.

Transaction reduction (reducing the number of transactions scanned in future iterations): A transaction that does not contain any frequent *k*-itemsets cannot contain any frequent (*k*+ 1)-itemsets. Therefore, such a transaction can be marked or removed from further consideration because subsequent database scans for *j* -itemsets, where *j* > *k*, will not need to consider such a transaction.

An attribute Size_Of_Transaction (SOT), containing number of items in individual transaction in database.

Suppose the minimum support count min_sup=3.

The algorithm is as follows:

Step 1: First we have to convert database into the desired database that is with SOT column.

Step 2: In the first iteration of the algorithm, each item is a member of the set of candidate 1-itemsets, C1. The algorithm simply scans all of the transactions in order to count the number of occurrences of each item

Step 3: This algorithm will then generate number of items in each transaction. We called this Size_Of _Transaction (SOT).

Step 4: Because of min_sup=3, the set of frequent 1-itemset, L1 can be determined. It consists of the candidate 1-itemset, C1, satisfying minimum support.

Step 5: Since the support count of I5,I6,I7 are less than 3, they won't appear in L1. Delete these data from D. In addition, when L1 is generated, now, the value of k is 2, delete those records of transaction having SOT=1 in D. And there won't exist any elements of C2 in the records we find there is only one data in the T9. We delete the data and obtain transaction database D1.

Step 6: To discover the set of frequent 2-itemsets, L2, the algorithm uses the join L1 ⋈L1 to generate a candidate set of 2- itemsets, C2.

Step 7: The transactions in D1 are scanned and the support count and SOT of each candidate itemset in C2 is accumulated.

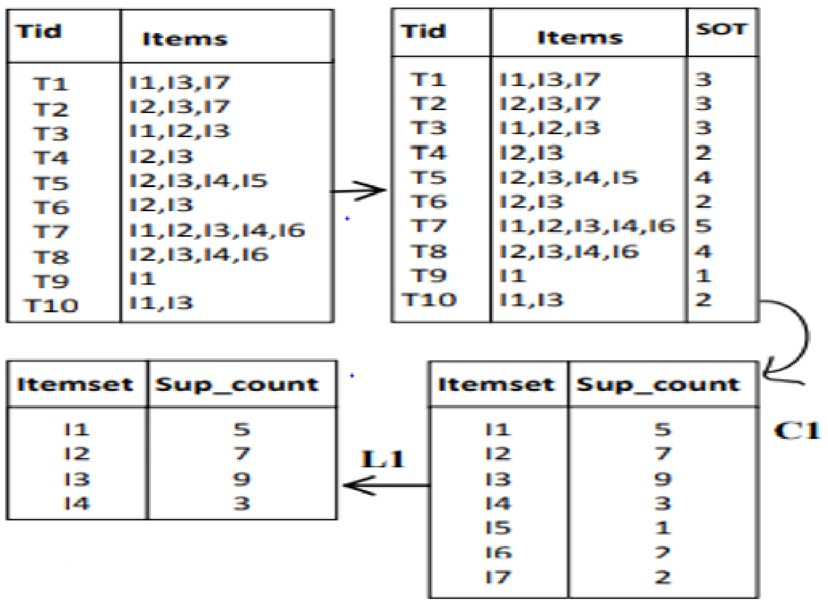
Step 8: The set of frequent 2-itemsets, L2, is then determined, consisting of those candidate 2-itemsets in C2 having minimum support.

Step 9: After L2 is generated, we can find the transaction record of T1, T2,T4, T6, T10 are only two in D1.Now,the value of k is 2,delete those records of transaction having SOT=2.And there won't exist any elements of C3 in the records. Therefore, these records can be deleted and we obtain transaction database D2.

Step 10: To discover the set of frequent 3-itemsets, L3, the algorithm uses the join L2∞ L2 to generate a candidate set of 3- itemsets C3, where C3= L2 ∞L2= I1， I2， I3},{ I2， I3， I4}}. There are a number of elements in C3. According to the property of Apriori algorithm, C3 needs to prune. Because {I1， I2} not belongs to L2, we remove it from C3. Because the 2-subsets {I2， I3}, I2， I4} and {I3， I4} all belong to L2, they should remain in C3.

Step 11: The transactions in D2 are scanned and the support count of each candidate itemset in C3 is accumulated. Use C3 to generate L3.

Step 12: L3 has only one 3-itemsets so that C4 =Φ. The algorithm will stop and give out all the frequent itemsets. Step 13: Algorithm will be generated for Ck until Ck+1 becomes empty.

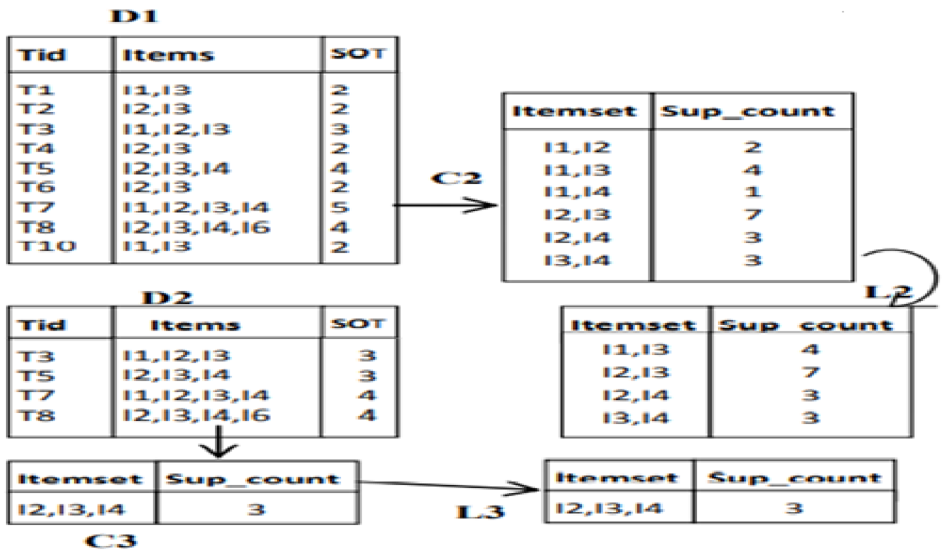


Tid	Items
T1	I1,I3,I7
T2	I2,I3,I7
T3	I1,I2,I3
T4	I2,I3
T5	I2,I3,I4,I5
T6	I2,I3
T7	I1,I2,I3,I4,I6
T8	I2,I3,I4,I6
T9	I1
T10	I1,I3

Tid	Items	SOT
T1	I1,I3,I7	3
T2	I2,I3,I7	3
T3	I1,I2,I3	3
T4	I2,I3	2
T5	I2,I3,I4,I5	4
T6	I2,I3	2
T7	I1,I2,I3,I4,I6	5
T8	I2,I3,I4,I6	4
T9	I1	1
T10	I1,I3	2

Itemset	Sup_count
I1	5
I2	7
I3	9
I4	3
I5	1
I6	2
I7	2

Itemset	Sup_count
I1	5
I2	7
I3	9
I4	3



Tid	Items	SOT
T1	I1,I3	2
T2	I2,I3	2
T3	I1,I2,I3	3
T4	I2,I3	2
T5	I2,I3,I4	4
T6	I2,I3	2
T7	I1,I2,I3,I4	5
T8	I2,I3,I4,I6	4
T10	I1,I3	2

Itemset	Sup_count
I1,I2	2
I1,I3	4
I1,I4	1
I2,I3	7
I2,I4	3
I3,I4	3

Tid	Items	SOT
T3	I1,I2,I3	3
T5	I2,I3,I4	3
T7	I1,I2,I3,I4	4
T8	I2,I3,I4,I6	4

Itemset	Sup_count
I1,I3	4
I2,I3	7
I2,I4	3
I3,I4	3

Itemset	Sup_count
I2,I3,I4	3

Itemset	Sup_count
I2,I3,I4	3

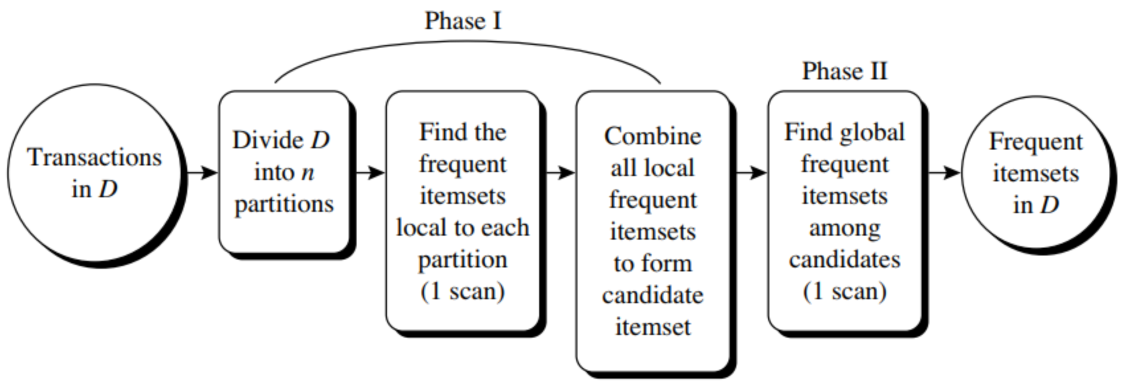
Partitioning (partitioning the data to find candidate itemsets):

A partitioning technique can be used that requires just two database scans to mine the frequent itemsets. It consists of two phases.

In phase I, the algorithm divides the transactions of *D* into *n*-non overlapping partitions. If the minimum relative support threshold for transactions in *D* is *min_sup*, then the minimum support count for a partition is *min_sup* × *the number of transactions in that partition*. For each partition, all the *local frequent itemsets* are found.

A local frequent itemset may or may not be frequent with respect to the entire database, D. However, any itemset that is potentially frequent with respect to D must occur as a frequent itemset in at least one of the partitions. Therefore, all local frequent itemsets are candidate itemsets with respect to D. The collection of frequent itemsets from all partitions forms the global candidate itemsets with respect to D.

In Phase II, A second scan of D is conducted in which the actual support of each candidate is assessed to determine the global frequent itemsets. Partition size and the number of partitions are set so that each partition can fit into main memory and therefore be read only once in each phase.


Phase I
Phase II
Transactions in D
Divide D into n partitions
Find the frequent itemsets local to each partition (1 scan)
Combine all local frequent itemsets to form candidate itemset
Find global frequent itemsets among candidates (1 scan)
Frequent itemsets in D

Sampling (mining on a subset of the given data): The basic idea of the sampling approach is to pick a random sample *S* of the given data *D*, and then search for frequent itemsets in *S* instead of *D*. In this way, we trade off some degree of accuracy against efficiency.

The *S* sample size is such that the search for frequent itemsets in *S* can be done in main memory, and so only one scan of the transactions in *S* is required overall. Because we are searching for frequent itemsets in *S* rather than in *D*, it is possible that we will miss some of the global frequent itemsets.

To reduce this possibility, we use a lower support threshold than minimum support to find the frequent itemsets local to *S* (denoted *LS*). The rest of the database is then used to compute the actual frequencies of each itemset in *LS*.

A mechanism is used to determine whether all the global frequent itemsets are included in *LS*. If *LS* actually contains all the frequent itemsets in *D*, then only one scan of *D* is required. Otherwise, a second pass can be done to find the frequent itemsets that were missed in the first pass. The sampling approach is especially beneficial when efficiency is of utmost importance such as in computationally intensive applications that must be run frequently.

Dynamic itemset counting (adding candidate itemsets at different points during a scan): A dynamic itemset counting technique was proposed in which the database is partitioned into blocks marked by start points.

In this variation, new candidate itemsets can be added at any start point, unlike in Apriori, which determines new candidate itemsets only immediately before each complete database scan. The technique uses the count-so-far as the lower bound of the actual count. If the count-so-far passes the minimum support, the itemset is added into the frequent itemset collection and can be used to generate longer candidates. This leads to fewer database scans than with Apriori for finding all the frequent itemsets.

A Pattern-Growth Approach for Mining Frequent Itemsets

The Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain.

It may still need to generate a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.

It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets.

An interesting method in this attempt is called frequent pattern growth, or simply FP-growth, which adopts a *divide-and-conquer* strategy as follows. First, it compresses the database representing frequent items into a frequent pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of *conditional* *databases* each associated with one frequent item or “pattern fragment,” and mines each database separately. For each “pattern fragment,” only its associated data sets need to be examined. Therefore, this approach may substantially reduce the size of the data sets to be searched, along with the “growth” of patterns being examined.

FP-growth (finding frequent itemsets without candidate generation).

Transactional Data for an *AllElectronics* Branch

TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

We re-examine the mining of transaction database, *D*, The first scan of the database is the same as Apriori, which derives the set of frequent items (1-itemsets) and their support counts (frequencies). Let the minimum support count be 2.

The set of frequent items is sorted in the order of descending support count. This resulting set or *list* is denoted by *L*. Thus, we have *L* =*{{*I2: 7*}*, *{*I1: 6*}*, *{*I3: 6*}*,*{*I4: 2*}*, *{*I5: 2*}}*. An FP-tree is then constructed as follows.

First, create the root of the tree, labelled with “null.” Scan database *D* a second time.

The items in each transaction are processed in *L* order (i.e., sorted according to descending support count), and a branch is created for each transaction. For example, the scan of the first transaction, “T100: I1, I2, I5,” which contains three items (I2, I1, I5 in *L* order), leads to the construction of the first branch of the tree with three nodes, <I2: 1>, <I1: 1>, and <I5: 1>, where I2 is linked as a child to the root, I1 is linked to I2, and I5 is linked to I1.

The second transaction, T200, contains the items I2 and I4 in *L* order, which would result in a branch where I2 is linked to the root and I4 is linked to I2. However, this branch would share a common prefix, I2, with the existing path for T100.

Therefore, we instead increment the count of the I2 node by 1, and create a new node, <I4: 1*i*> which is linked as a child to <I2: 2>. In general when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly.

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. The tree obtained after scanning all the transactions with the associated node-links. In this way, the problem of mining frequent patterns in databases is transformed into that of mining the FP-tree.

The FP-tree is mined as follows. Start from each frequent length-1 pattern (as an initial suffix pattern), construct its conditional pattern base (a "sub-database," which consists of the set of *prefix paths* in the FP-tree co-occurring with the suffix pattern), then construct its (*conditional*) FP-tree, and perform mining recursively on the tree.

The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Mining of the FP-tree is summarized in Table 6.2 and detailed as follows.

We first consider I5, which is the last item in *L*, rather than the first. The reason for starting at the end of the list will become apparent as we explain the FP-tree mining process. I5 occurs in two FP-tree branches. (The occurrences of I5 can easily be found by following its chain of node-links.) The paths formed by these branches are <I2, I1,I5: 1> and <I2, I1, I3, I5: 1>.

Therefore, considering I5 as a suffix, its corresponding two prefix paths are <I2, I1: 1> and <I2, I1, I3: 1>, which form its conditional pattern base. Using this conditional pattern base as a transaction database, we build an I5-conditional FP-tree, which contains only a single path, <I2: 2, I1: 2>; I3 is not included because its support count of 1 is less than the minimum support count. The single path generates all the combinations of frequent patterns: *{*I2, I5: 2*}*, *{*I1, I5: 2*}*, *{*I2, I1, I5: 2}.

For I4, its two prefix paths form the conditional pattern base, *{{*I2 I1: 1*}*, *{*I2: 1}}, which generates a single-node conditional FP-tree, *{*I2: 2*}*, and derives one frequent pattern, *{*I2, I4: 2*}*.

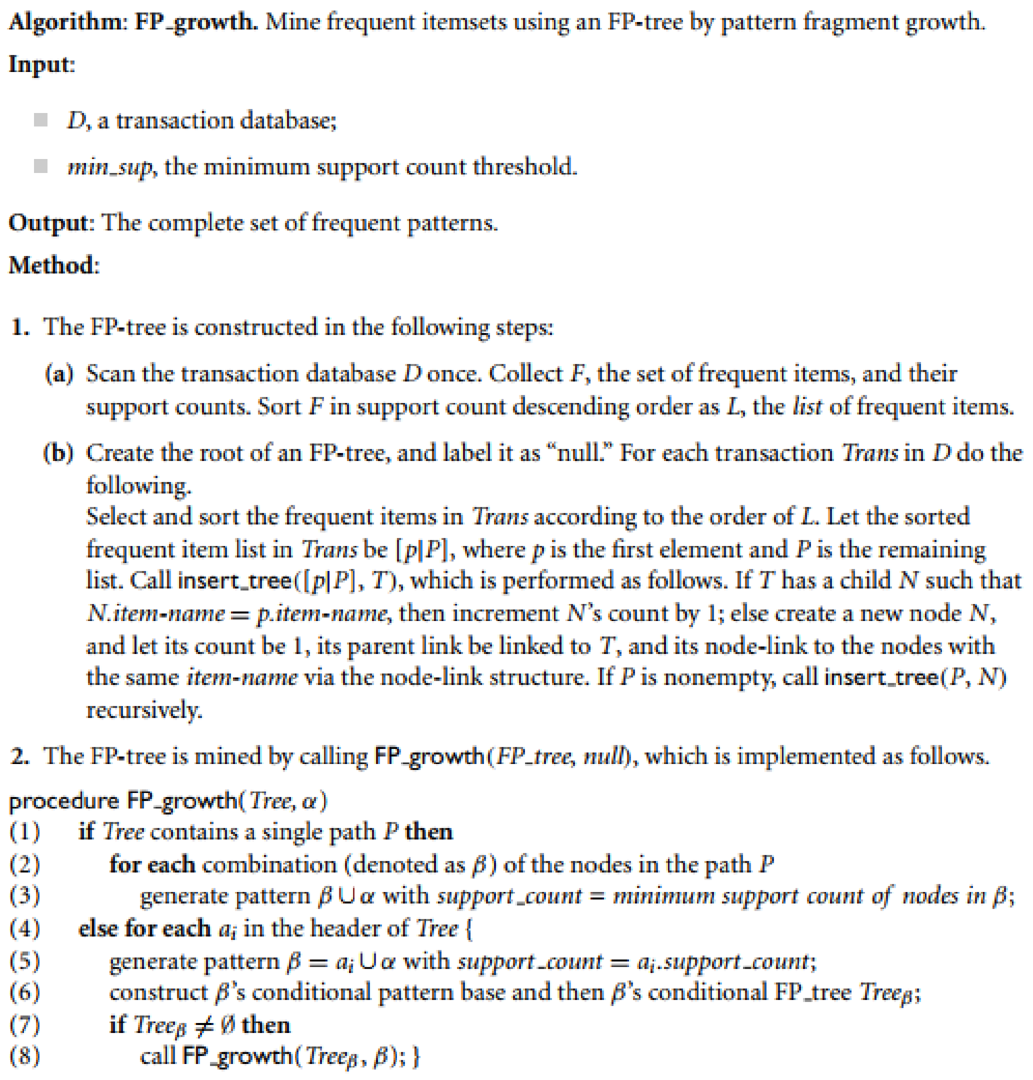
Similar to the preceding analysis, I3's conditional pattern base is {{I2, I1: 2}, {I2: 2},{I1: 2}}. Its conditional FP-tree has two branches, <I2: 4, I1: 2> and <I1: 2>, which generates the set of patterns {{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}}.

Finally, I1's conditional pattern base is {{I2: 4}}, with an FP-tree that contains only one node, <I2: 4>, which generates one frequent pattern, {I2, I1: 4}.

The FP-growth method transforms the problem of finding long frequent patterns into searching for shorter ones in much smaller conditional databases recursively and then concatenating the suffix. It uses the least frequent items as a suffix, offering good selectivity. The method substantially reduces the search costs.

When the database is large, it is sometimes unrealistic to construct a main memory based FP-tree. An interesting alternative is to first partition the database into a set of projected databases, and then construct an FP-tree and mine it in each projected database. This process can be recursively applied to any projected database if its FP-tree still cannot fit in main memory.

A study of the FP-growth method performance shows that it is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm.



Algorithm: **FP_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- D, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.

Method:

1. The FP-tree is constructed in the following steps:
   - **(a)** Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the *list* of frequent items.
   - **(b)** Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in D do the following.
     Select and sort the frequent items in *Trans* according to the order of L. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth($Tree, \alpha$)
(1)   if $Tree$ contains a single path $P$ then
(2)       for each combination (denoted as $\beta$) of the nodes in the path $P$
(3)           generate pattern $\beta \cup \alpha$ with $support\_count$ = $minimum\ support\ count\ of\ nodes\ in\ \beta$;
(4)   else for each $a_i$ in the header of $Tree$ {
(5)       generate pattern $\beta = a_i \cup \alpha$ with $support\_count = a_i.support\_count$;
(6)       construct $\beta$'s conditional pattern base and then $\beta$'s conditional FP_tree $Tree_\beta$;
(7)       if $Tree_\beta \neq \emptyset$ then
(8)           call FP_growth($Tree_\beta, \beta$); }
```

Mining Frequent Itemsets Using the Vertical Data Format

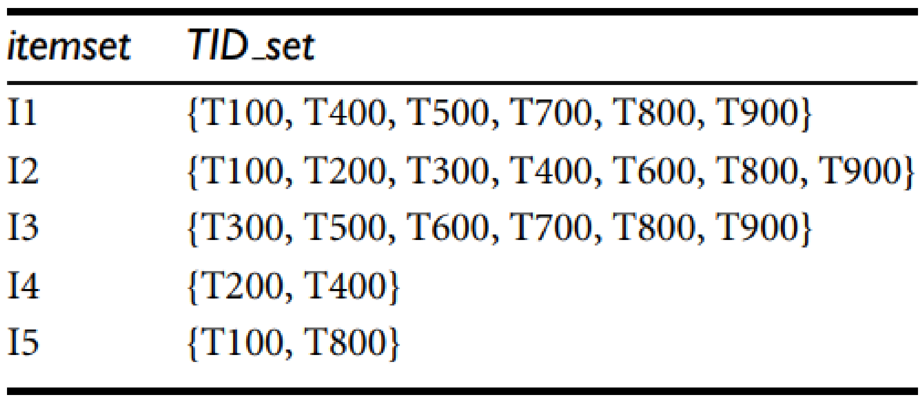
Both the Apriori and FP-growth methods mine frequent patterns from a set of transactions in TID-itemset format where TID is a transaction ID and itemset is the set of items bought in transaction TID. This is known as the *horizontal data format*.

Alternatively, data can be presented in item-TID set format. Mining can be performed on this data set by intersecting the TID sets of every pair of frequent single items. The minimum support count is 2. here are 10 intersections performed in total, which lead to eight nonempty 2-itemsets, as shown in Table 6.4. Notice that because the itemsets {I1, I4} and {I3, I5} each contain only one transaction, they do not belong to the set of frequent 2-itemsets.

Based on the Apriori property, a given 3-itemset is a candidate 3-itemset only if every one of its 2-itemset subsets is frequent. The candidate generation process here will generate only two 3-itemsets: {I1, I2, I3} and {I1, I2, I5}. By intersecting the TID sets of any two corresponding

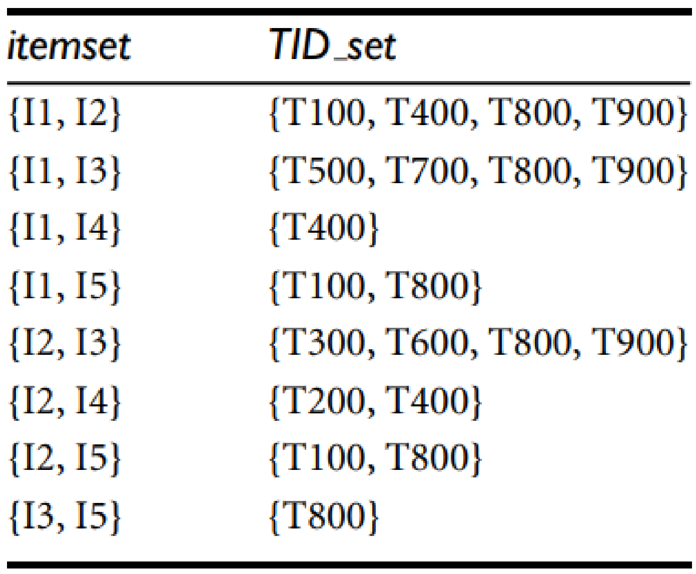
2-itemsets of these candidate 3-itemsets, where there are only two frequent 3-itemsets: {I1, I2, I3: 2} and {I1, I2, I5: 2}.

The Vertical Data Format of the Transaction Data Set *D* of Table 6.1



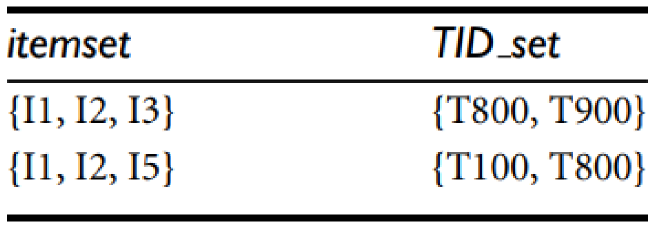
itemset	*TID_set*
I1	{T100, T400, T500, T700, T800, T900}
I2	{T100, T200, T300, T400, T600, T800, T900}
I3	{T300, T500, T600, T700, T800, T900}
I4	{T200, T400}
I5	{T100, T800}

2-Itemsets in Vertical Data Format



itemset	*TID_set*
{I1, I2}	{T100, T400, T800, T900}
{I1, I3}	{T500, T700, T800, T900}
{I1, I4}	{T400}
{I1, I5}	{T100, T800}
{I2, I3}	{T300, T600, T800, T900}
{I2, I4}	{T200, T400}
{I2, I5}	{T100, T800}
{I3, I5}	{T800}

3-Itemsets in Vertical Data Format



itemset	*TID_set*
{I1, I2, I3}	{T800, T900}
{I1, I2, I5}	{T100, T800}

From Association Analysis to Correlation Analysis

The support and confidence measures are insufficient at filtering out uninteresting association rules. To tackle this weakness, a correlation measure can be used to augment the support–confidence framework for association rules. This leads to *correlation rules* of the form *A* => *B* [*support*, *confidence*, *correlation*].

That is, a correlation rule is measured not only by its support and confidence but also by the correlation between itemsets *A* and *B*. There are many different correlation measures from which to choose. In this subsection, we study several correlation measures to determine which would be good for mining large data sets.

Lift is a simple correlation measure that is given as follows. The occurrence of itemset *A* is independent of the occurrence of itemset *B* if *P(A UB)* = *P(A)P(B)*; otherwise, itemsets *A* and *B* are dependent and correlated as events. This definition can easily be extended to more than two itemsets. The lift between the occurrence of *A* and *B* can be measured by computing

$$lift(A, B) = \frac{P(A \cup B)}{P(A)P(B)}.$$

χ2 Correlation Test for Nominal Data

For nominal data, a correlation relationship between two attributes, A and B, can be discovered by a χ2 (chi-square) test. Suppose A has c distinct values, namely a1,a2,…,ac. B has r distinct values, namely b1,b2,….,br. The data tuples described by A and B can be shown as a contingency table, with the c values of A making up the columns and the r values of B making up the rows. Let (ai,bj) denote the joint event that attribute A takes on value ai and attribute B takes on value bj, that is, where A= ai,B=bj. Each and every possible (Ai,Bj) joint event has its own cell (or slot) in the table. The χ2 value (also known as the Pearson χ2 statistic) is computed as

$$\chi^2 = \sum_{i=1}^{c}\sum_{j=1}^{r} \frac{(o_{ij} - e_{ij})^2}{e_{ij}},$$

$$e_{ij} = \frac{count(A = a_i) \times count(B = b_j)}{n},$$

where n is the number of data tuples, count. A = |ai| is the number of tuples having value ai for A, and count.B= bj is the number of tuples having value bj for B. The sum is computed over all of the r × c cells.

The $\chi 2$ statistic tests the hypothesis that A and B are independent, that is, there is no correlation between them. The test is based on a significance level, with $(r-1)\times(c-1)$ degrees of freedom.

Example 2.1's 2 × 2 Contingency Table Data

	male	*female*	*Total*
fiction	250 (90)	200 (360)	450
non_fiction	50 (210)	1000 (840)	1050
Total	300	1200	1500

Note: Are *gender* and *preferred_reading* correlated?

Using Eq. (3.1) for χ^2 computation, we get

$$\chi^2 = \frac{(250-90)^2}{90} + \frac{(50-210)^2}{210} + \frac{(200-360)^2}{360} + \frac{(1000-840)^2}{840}$$
$$= 284.44 + 121.90 + 71.11 + 30.48 = 507.93.$$

A Comparison of Pattern Evaluation Measures

Given two itemsets, A and B, the *pattern evaluation* measures of A and B is defined as

1. all confidence,
2. max confidence,
3. Kulczynski, and
4. cosine.

$$all\_conf(A,B) = \frac{sup(A \cup B)}{max\{sup(A), sup(B)\}} = min\{P(A|B), P(B|A)\},$$

$$max\_conf(A, B) = max\{P(A\,|\,B), P(B\,|\,A)\}.$$

$$Kulc(A, B) = \frac{1}{2}(P(A|B) + P(B|A)).$$

$$cosine(A, B) = \frac{P(A \cup B)}{\sqrt{P(A) \times P(B)}} = \frac{sup(A \cup B)}{\sqrt{sup(A) \times sup(B)}}$$
$$= \sqrt{P(A|B) \times P(B|A)}.$$