Association analysis: problem definition, frequent item set generation: The Apriori principle, frequent item set generation in the Apriori algorithm, candidate generation and pruning, support counting, rule generation, compact representation of frequent item sets, FP-Growth algorithms. (Tan)

Association analysis:

It is useful for discovering interesting relationships hidden in large data sets. The uncovered relationships can be represented in the form of **association rules** or sets of frequent items. The following Table illustrates an example of **market basket transactions**.

TID	Items
1	{Bread, Milk}
2	{Bread, Diapers, Beer,
3	ggs}
4	{Milk, Diapers, Beer, C
5	ola}

Table. An example of market basket transactions

The following rule can be extracted from the data set shown in aboveTable:

{Diapers} → {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.

Problem Definition:

Binary Representation Market basket data can be represented in a binary format as shown in following Table, where 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 in a transaction and zero** otherwise. Because the presence of an item in a transaction is often considered more important than its absence, 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. A binary 0/1 representation of market basket data.

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-itelnset. For instance, {Beer, Diapers, Milk} is an example of a 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 *tj* is said to contain an itemset *X* if *X* is a subset of *tj*. For example, the second transaction shown in above Table contains the itemset {Bread, Diapers} but not {Bread, Milk}. An important property of an itemset is its support count, which refers to the number of transactions that contain a particular itemset. Mathematically, the support COunt, σ*(X),* for an itemset *X* can be stated as follows:

$$\sigma(X) = \left|\{t_i | X \subseteq t_i,\ t_i \in T\}\right|,$$

In the data set shown in above Table the support count for {Beer, Diapers, Milk} is equal to two because there are only two transactions that contain all three items.

Association Rule An association rule is an implication expression of the form *X* => Y, where *X* and Yare disjoint itemsets, i.e., *X* ∩ Y = φ. 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 (or) The rule X => Y holds in the transaction set D with support s, where s is the percentage of transactions in D that contain XU Y.

support(X =>Y) = Prob{XUY} (or)

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

Confidence determines how frequently items in Y appear in transactions that contain *X.(or)* The rule X=>Y has confidence c in the transaction set D if c is the percentage of transactions in D containing X which also contain Y.

confidence(X=>Y) = Prob{Y/X} (or)

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

Consider the rule {Milk, Diapers} => {Beer}. Since the support count for {Milk, Diapers, Beer} is 2 and the total number of transactions is 5, the rule's support is 2/5 = 0.4. The rule's confidence is obtained by dividing the support count for {Milk, Diapers, Beer} by the support count for {Milk, Diapers}. Since there are 3 transactions that contain milk and diapers, the confidence for this rule is 2/3 = 0.67.

Association Rule Discovery

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

A brute-force approach for mining association rules is to compute the support and confidence for every possible rule. This approach is prohibitively expensive because there are exponentially many rules that can be extracted from a data set. More specifically, the total number of possible rules extracted from a data set that contains *d* items is

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

Even for the small data set shown in first Table, this approach requires us to compute the support and confidence for 36 - 27+ 1 = 602 rules. More than 80% of

the rules are discarded after applying *minsup* = 20% and *minconf* = 50%, thus making most of the computations become wasted.

For example, the following rules have **identical support** because they involve items from the same itemset, {Beer, Diapers, Milk}:

$\{\texttt{Beer}, \texttt{Diapers}\} \longrightarrow \{\texttt{Milk}\}$, $\{\texttt{Beer}, \texttt{Milk}\} \longrightarrow \{\texttt{Diapers}\}$,
$\{\texttt{Diapers}, \texttt{Milk}\} \longrightarrow \{\texttt{Beer}\}$, $\{\texttt{Beer}\} \longrightarrow \{\texttt{Diapers}, \texttt{Milk}\}$,
$\{\texttt{Milk}\} \longrightarrow \{\texttt{Beer},\texttt{Diapers}\}$, $\{\texttt{Diapers}\} \longrightarrow \{\texttt{Beer},\texttt{Milk}\}$.

If the itemset is infrequent, then all six candidate rules can be pruned immediately without having to compute their confidence values. Therefore, 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 itemsets that satisfy the *minsup* threshold. These itemsets are called frequent itemsets.
2. Rule Generation, whose objective is to extract all the high-confidence rules from the frequent itemsets found in the previous step. These rules are called strong rules.

Frequent item set generation

A lattice structure can be used to enumerate the list of all possible itemsets.

The following Figure shows an itemset lattice for *1= {a,b,c,d,e}.* In general, a data set that contains *k* items can potentially generate up to 2^k - 1 frequent itemsets, excluding the null set.

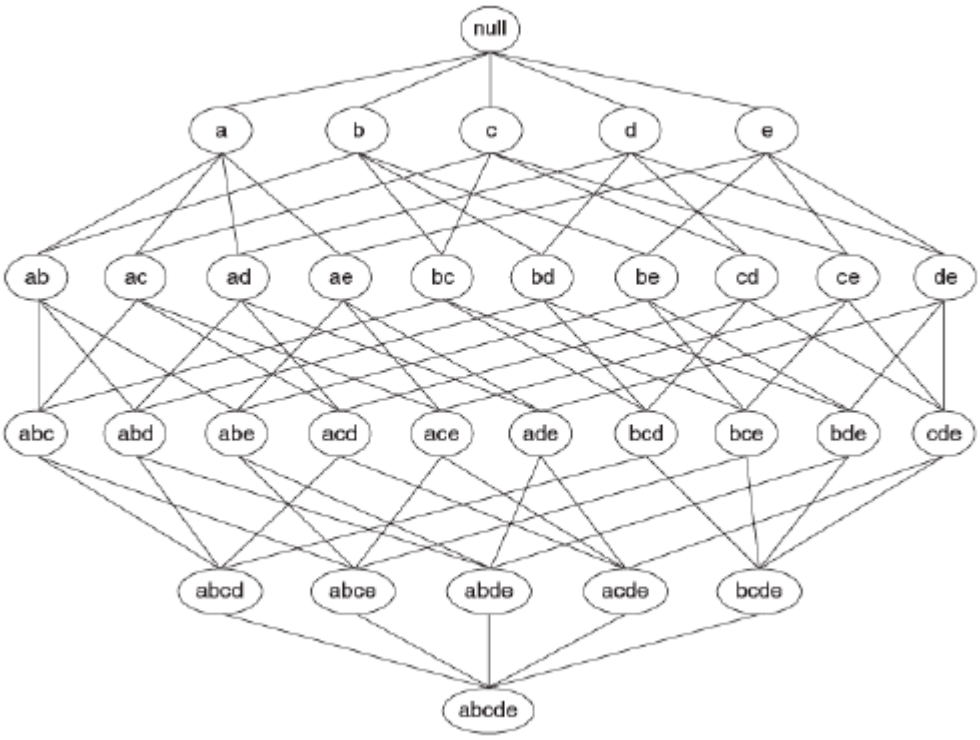


Figure. An itemset lattice.

A brute-force approach for finding frequent itemsets is to determine the support count for every candidate itemset in the lattice structure. To do this, we need to compare each candidate against every transaction, an operation that is shown in

following Figure. If the candidate is contained in a transaction, its support count will be incremented. For example, the support for {Bread, Milk} is incremented three times because the itemset is contained in transactions 1, 4, and 5.Such an approach can he very expensive because it requires *O(NMw)* comparisons, where *N* is the number of transactions, M = 2^k -1 is the number of candidate itemsets, and *w* is the maximum transaction width.

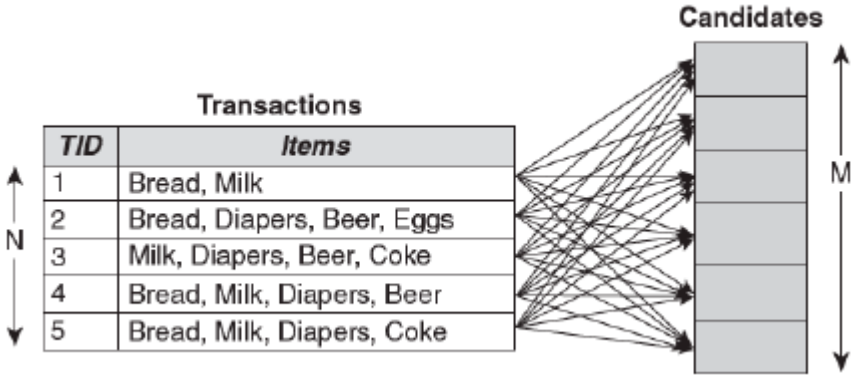


Figure. Counting the support of candidate itemsets

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

1. Reduce the number of candidate itemsets *(M).*
2. Reduce the number of comparisons.

The Apriori principle

If an itemset is *frequent, then all of* its *subsets must also be frequent.*

For ex consider the itemset lattice shown in following Figure. Suppose {c, *d,* e} is a frequent *itemset.* Clearly, any transaction that contains *{c,d,e}* must also contain *its* subsets, *{c,d}, {c,e}, {d,e},* {c}, *{d},* and {e}. As a result, if *{c,d,e}* is frequent, then all subsets of *{c,d,e}* (Le., the shaded itemsets in this figure) must *also* be frequent.

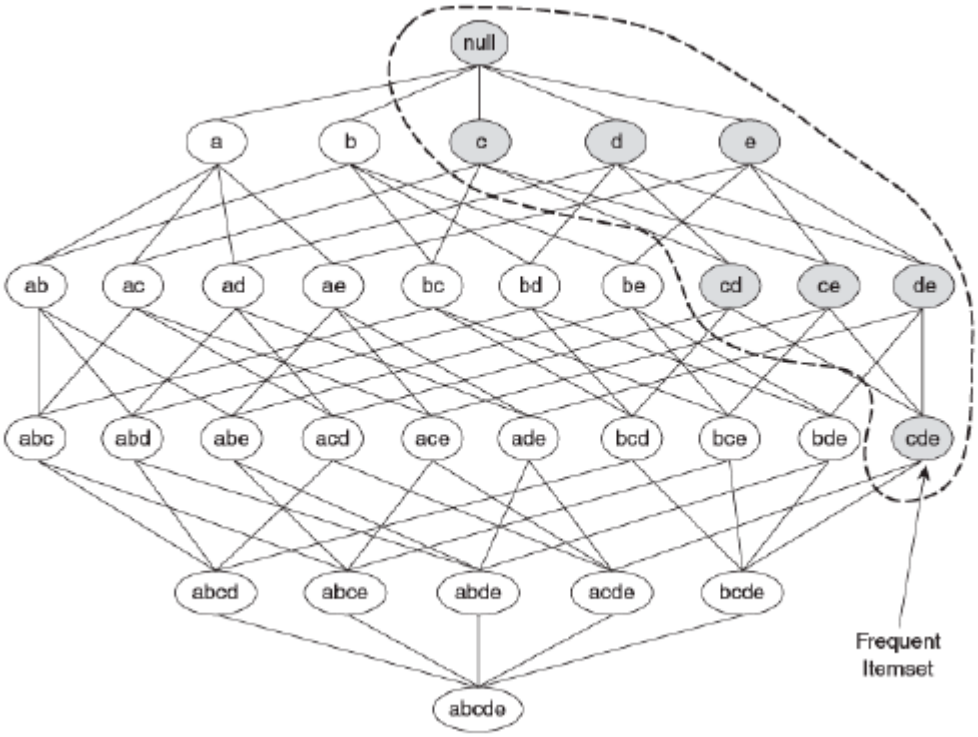


Figure. An illustration of the *Apriori* principle. If *{c, d,* e} is frequent, then all subsets of this itemset are frequent,

Conversely, if an itemset such as *{a, b}* is infrequent, then all of its supersets must be infrequent too. *For ex* in the following Figure, the entire sub graph containing the supersets of *{a, b}* can be pruned immediately once *{a, b}* is found to be infrequent.

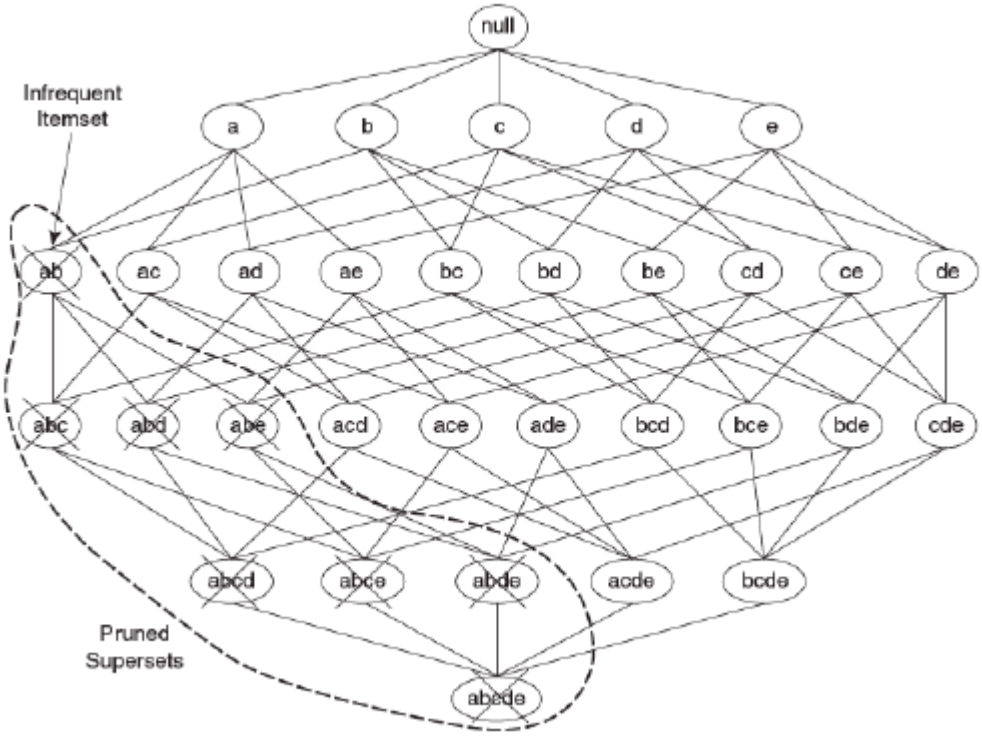


Figure. An illustration of support-based pruning. if *{a, b}* is infrequent, then all supersets of *{a, b}*
are infrequent.

This strategy of trimming the exponential search space based on the support measure is known as support-based pruning. In this the support for an itemset never exceeds the support for its subsets. This property is also known as the anti-monotone property of the support measure.

frequent item set generation in the Apriori algorithm

Apriori is an influential algorithm 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, as we shall see below. 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. This set is denoted L1. L1 is used to find L2, the 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, presented below, is used to reduce the search space.

The Apriori property. All non-empty subsets of a frequent itemset must also be frequent. This property is based on the following observation. By definition, if an itemset I does not satisfy the minimum support threshold, s, then I is not frequent, i.e., Prob{I} < s. 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, i.e.,Prob{ IUA} < s.

A two step process is followed, consisting of join and prune actions.

1. The **join step**: To find L_k, a set of **candidate** k-itemsets is generated by joining L_{k-1} with itself. This set of candidates is denoted C_k. The join, $L_{k-1} \bowtie L_{k-1}$, is performed, where members of L_{k-1} are joinable if they have $(k-2)$ items in common, that is, $L_{k-1} \bowtie L_{k-1} = \{A \bowtie B | A, B \in L_{k-1}, |A \cap B| = k-2\}$.

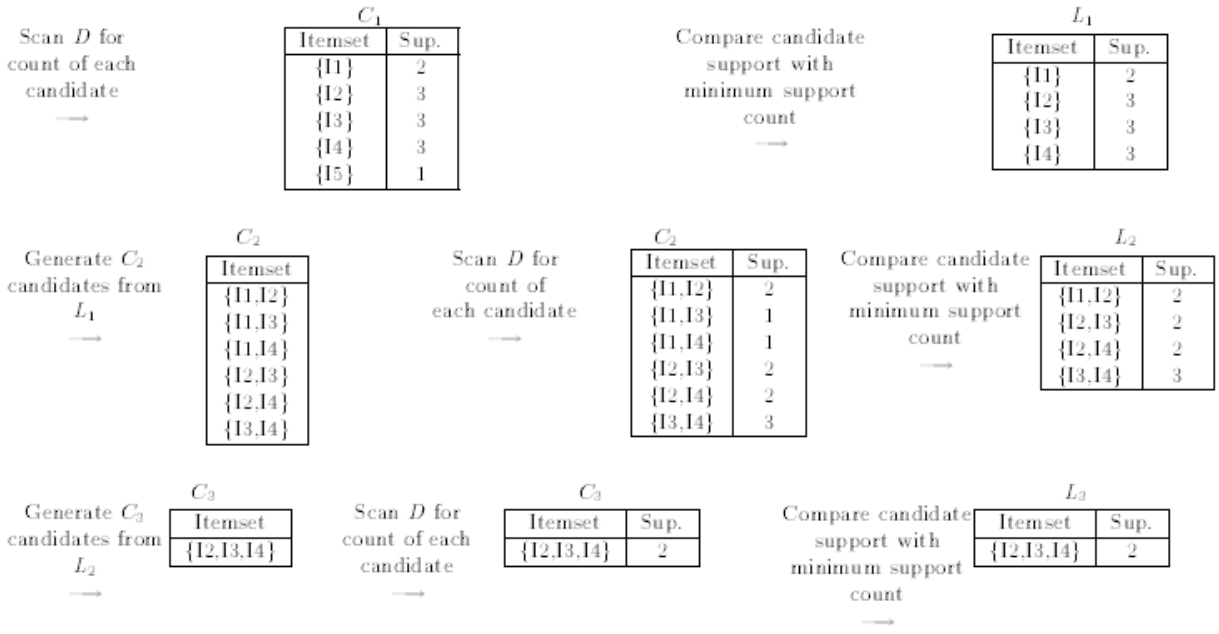
2. The prune step: Ck is a superset of Lk, that is, its members may or may not be frequent, but all of the frequent k-itemsets are included in Ck. A scan of the database to determine the count of each candidate in Ck would result in the determination of Lk. To reduce the size of Ck, the Apriori property 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.

AllElectronics database

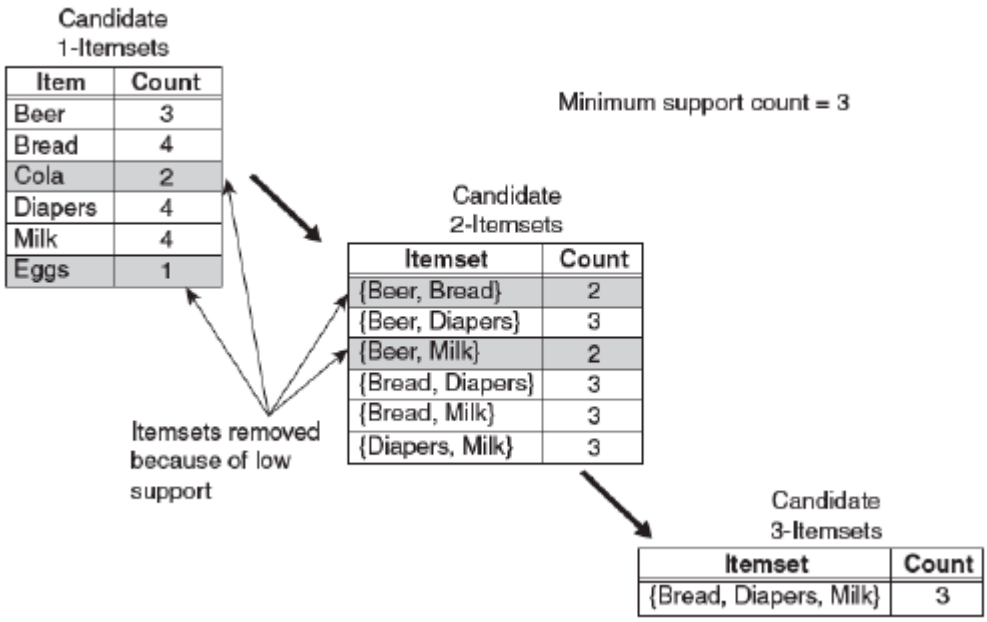
TID	List of item_ID's
T100	I1, I2, I5
T200	I2, I3, I4
T300	I3, I4
T400	I1, I2, I3, I4

Fig. Transactional data

Suppose that the minimum transaction support count required is 2 (i.e., min sup = 50%)



The following Figure provides a high-level illustration of the frequent itemset generation part of the *Apriori* algorithm for the transactions shown in first Table. We assume that the support threshold is 60%, which is equivalent to a minimum support count equal to 3.



Initially, every item is considered as a candidate l-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 l-itemsets because the *Apriori* principle ensures that all supersets of the infrequent l-itemsets must be infrequent. Because there are only four frequent l-itemsets, the number of candidate 2-itemsets generated by the algorithm is 6. Two of these six candidates, {Beer, Bread} and {Beer, Milk}, are subsequently found to be infrequent after computing their support values. The remaining four candidates are frequent, and thus will be used to generate candidate 3-itemsets. Without support-based pruning, there are 20 candidate 3-itemsets 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 strategy of enumerating all itemsets (up to size 3) as candidates will produce

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

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 pseudocode for the frequent itemset generation part of the *Apriori* algorithm is shown in Algorithm

1: $k = 1$.
2: $F_k = \{ i \mid i \in I \wedge \sigma(\{i\}) \geq N \times minsup\}$. {Find all frequent 1-itemsets}
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 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: $F_k = \{ c \mid c \in C_k \wedge \sigma(c) \geq N \times minsup\}$. {Extract the frequent k-itemsets}
13: **until** $F_k = \emptyset$
14: Result $= \bigcup F_k$.

Candidate Generation and Pruning

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.

Next, we will briefly describe several candidate generation procedures, including the one used by the apriori-gen function.

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.

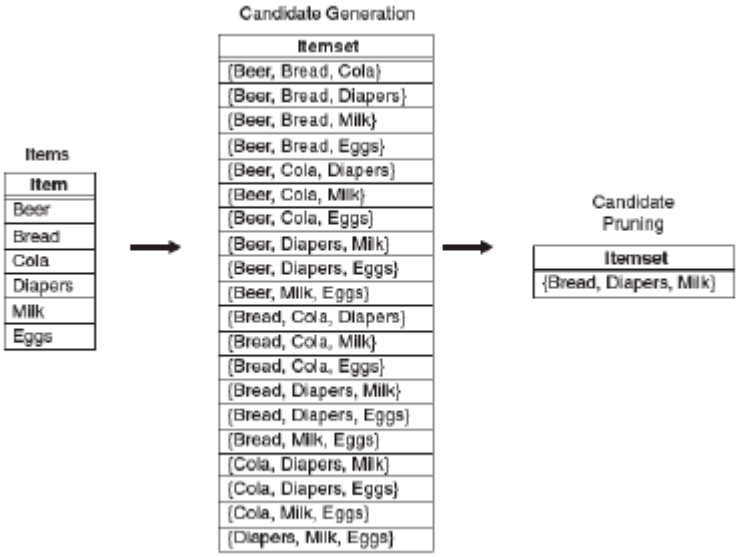


Figure. A brute-force method for generating candidate 3itemsets.

F_{k-l} X F_1 Method An alternative method for candidate generation is to extend each frequent *(k* - l)-itemset with other frequent items. The following Figure illustrates how a frequent 2-itemset such as {Beer, Diapers} can be augmented with a frequent item such as Bread to produce a candidate 3-itemset {Beer, Diapers, Bread}.

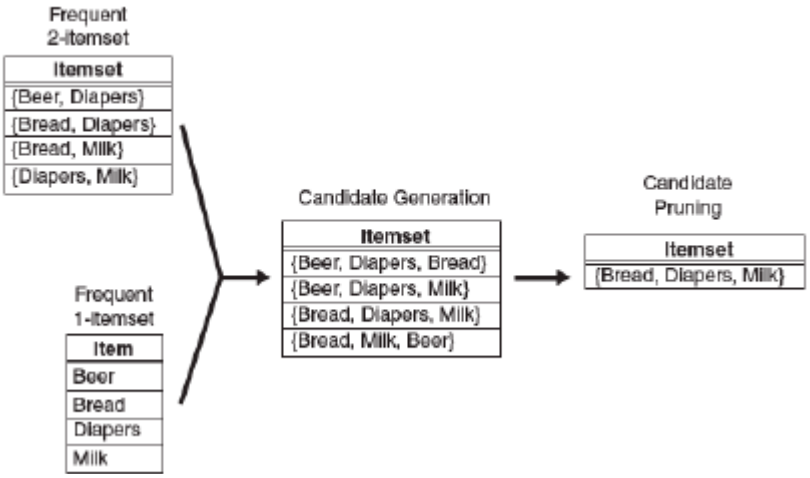


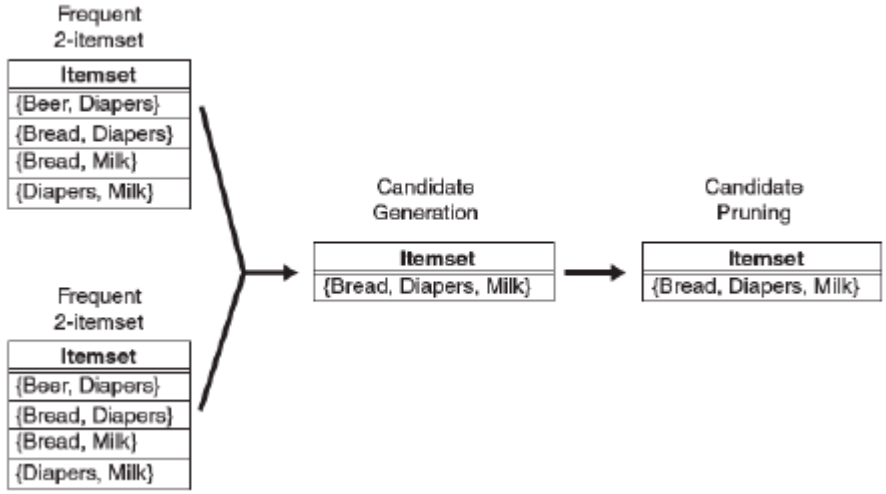
Figure. Generating and pruning candidate k itemsets by merging a frequent *(k - 1)*-itemset with a frequent item. Note that some of the candidates are unnecessary because their subsets are infrequent.

While this procedure is a substantial improvement over the brute-force method, it can still produce a large number of unnecessary candidates. For example, the candidate itemset obtained by merging {Beer, Diapers} with {Milk} is unnecessary because one of its subsets, {Beer, Milk}, is infrequent.

F_{k-1} xF_{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:

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

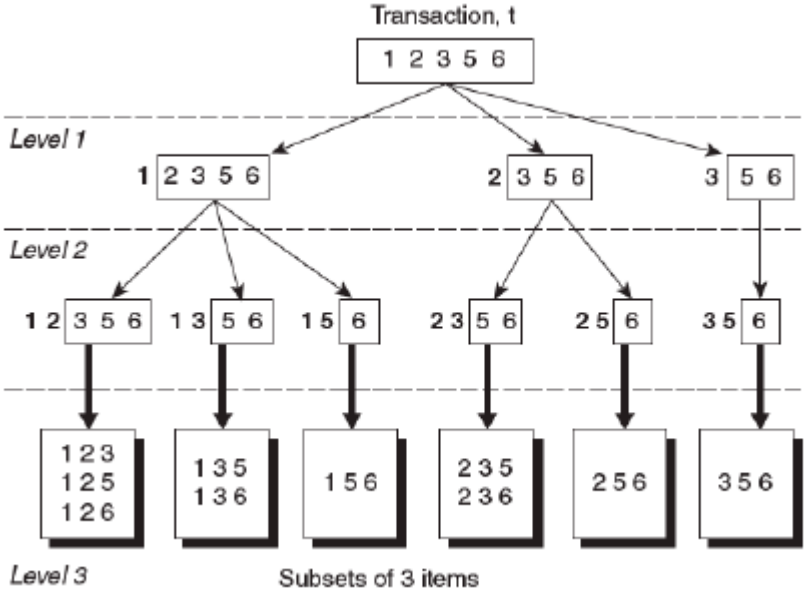


In the above Figure the frequent itemsets {Bread, Diapers} and {Bread, Milk} are merged to form a candidate 3-itemset {Bread, Diapers, Milk}. The algorithm does not have to merge {Beer, Diapers} with {Diapers, Milk} because the first item in both itemsets is different. Indeed, if {Beer, Diapers, Milk} is a viable candidate, it would have been obtained by merging {Beer, Diapers} with {Beer, Milk} instead.

Support Counting

One approach for doing this is to compare each transaction against every candidate itemset and to update the support counts of candidates contained in the transaction. This approach is computationally expensive, especially when the numbers of transactions and candidate itemsets are large.

An alternative approach is to enumerate the itemsets contained in each transaction and use them to update the support counts of their respective candidate itemsets. To illustrate, consider a transaction *t* that contains five items, {1, 2, 3,5, 6}. There are 10 itemsets of size 3 contained in this transaction. The following Figure shows a systematic way for enumerating the 3-itemsets contained in *t.* Assuming that each itemset keeps its items in increasing lexicographic order.



Rule Generation

Here we describe how to extract association rules efficiently from a given frequent itemset. Each frequent k-itemset, Y, can produce up to 2^{k-2} association rules, ignoring rules that have empty antecedents or consequents ($\varphi \rightarrow Y$) or ($Y \rightarrow \varphi$).

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.

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

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.*

For example, if *{acd}* $\rightarrow$ *{b}* and *{abd}* $\rightarrow$ *{c}* are high-confidence rules, then the candidate rule *{ad}* $\rightarrow$*{be}* is generated by merging the consequents of both rules.

The following Figure shows a lattice structure for the association rules generated from the frequent itemset *{a, b,* c, *d}.* If any node in the lattice has low confidence, then according to above Theorem, the entire sub graph spanned by the node can be pruned immediately. Suppose the confidence for *{bcd}→{a}* is low. All the rules containing item *a* in its consequent, including *{cd}→{ab}, {bd} → {ae}, {be}→{ad},* and *{d} → {abe}* can be discarded.

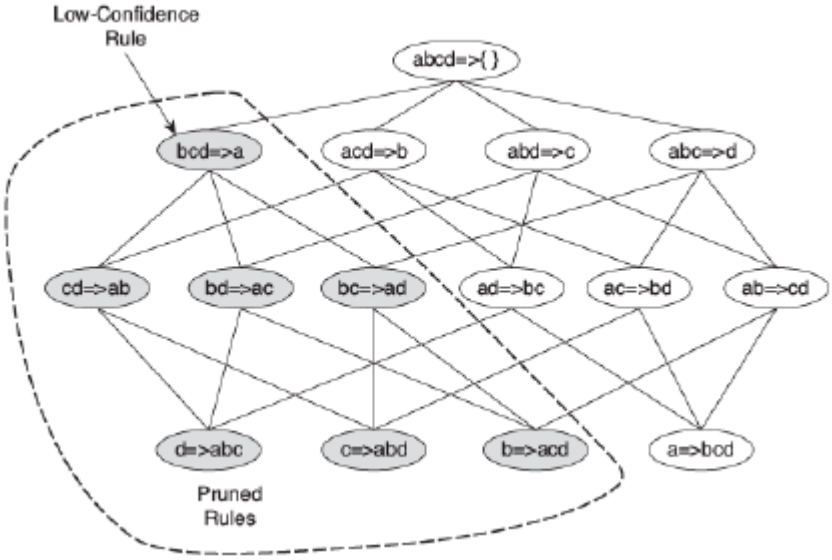


Figure . Pruning of association rules using the confidence measure.

Compact Representation of Frequent Itemsets

It is useful to identify a small representative set of itemsets from which all other frequent iternsets can be derived. Two such representations are as follows.

Maximal Frequent Itemsets

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

Consider the itemset lattice shown in following Figure. The itemsets in the lattice are divided into two groups: those that are frequent and those that are infrequent. A frequent itemset order, 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 (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,e}* is non-maximal because one of its immediate supersets, {*a,* c, e} 1 is frequent.

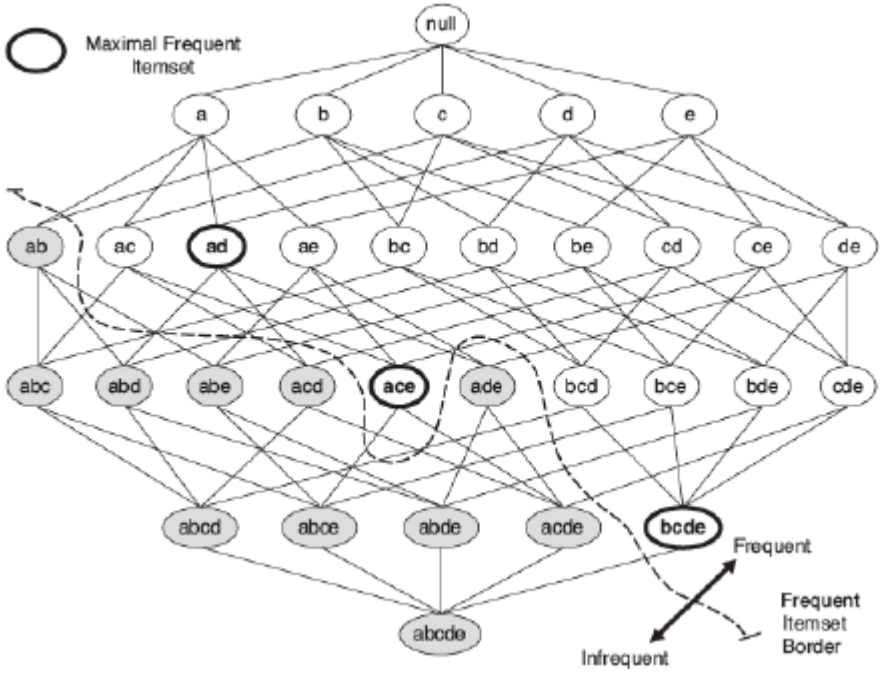


Figure Maximal frequent itemset.

Despite providing a compact representation, maximal frequent itemsets do not contain the support information of their subsets. For example, the support of the maximal frequent itemsets *{a*, c, *e}*, *{a*, *d}*, and *{b,c,d,e}* do not provide any hint about the support of their subsets.

Closed Frequent Itemsets

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

Examples of closed itemsets are shown in following Figure. To better illustrate the support count of each itemset, we have associated each node (itemset) in the lattice with a list of its corresponding transaction IDs.

An itemset is a closed frequent itemset if it is closed and its support is greater than or equal to *minsup.*

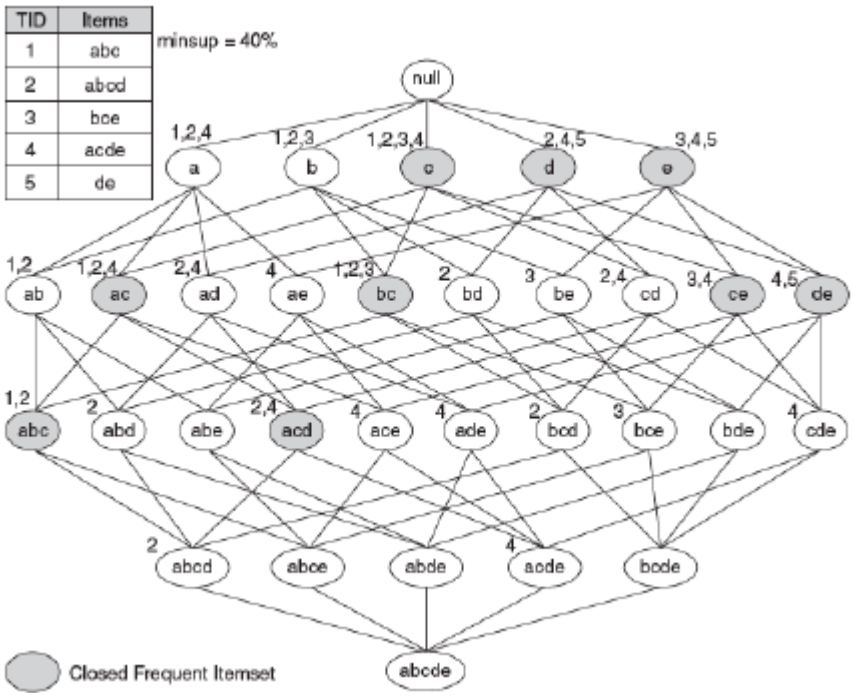


Figure An example of the closed frequent ttemsets (with minimum support count equal to 40%).

FP-Growth Algorithm

This section presents an alternative algorithm called FP-growth that takes a radically different approach to discovering frequent itemsets. The algorithm does not subscribe to the generate-and-test paradigm of *Apriori.* Instead, it encodes the data set 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 and mapping each transaction onto a path in the FP-tree. As different transactions can have several items in common, their paths may overlap. The more the paths overlap with one another, the more compression we can achieve using the FP-tree structure.

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}

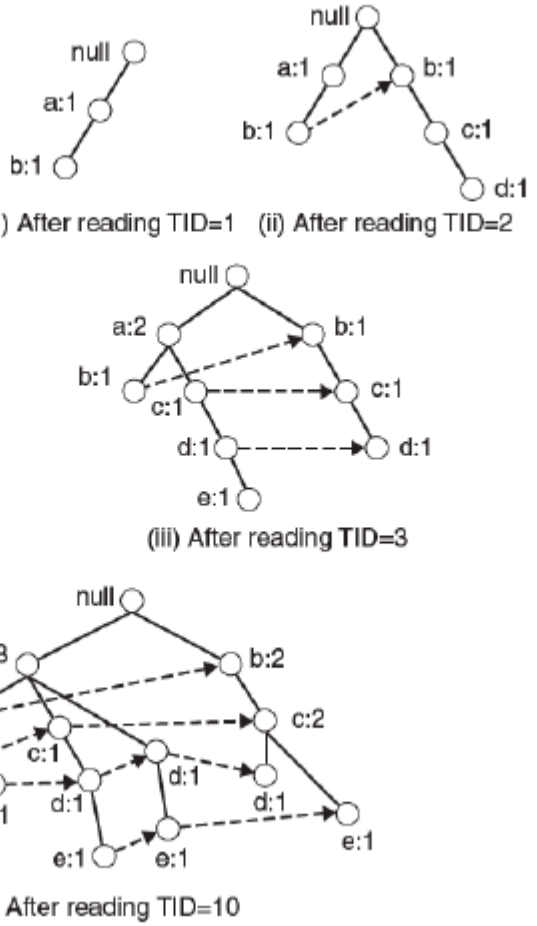


Figure. Construction of an FP-tree.

The above Figure shows a data set that contains ten transactions and five items.

The structures of the FP-tree after reading the first three transactions are also depicted in the diagram. Each node in the tree contains the label of an item along with a counter that shows the number 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 as follows

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 above Figure, *a* is the most frequent item, followed by *b*, c, *d*, and e.

The algorithm makes a second pass over the data to construct the FPtree.

1.After reading the first transaction, *{a, b}*, the nodes labeled 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..

2. The algorithm makes a second pass over the data to construct the FPtree.

After reading the first transaction, *{a, b}*, the nodes labeled 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 represent 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 (which is *a)* 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 Figure.

The size of an FP-tree is typically smaller than the size of the uncompressed data because many transactions in market basket data often share a few items in common.

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. Given the example tree shown in above Figure, the algorithm looks for frequent itemsets ending in e first, followed by *d,* c, *b,* and finally, *a.*

We can derive the frequent itemsets ending with a particular item. The extracted paths are shown in following Figure

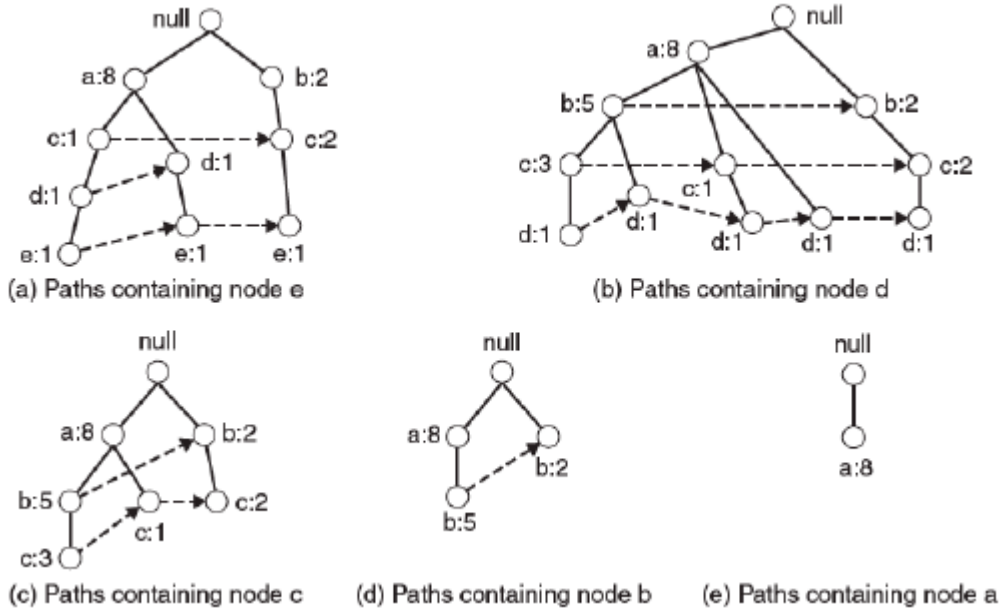


Figure. Decomposing the frequent itemset generation problem into multiple sub problems, where each sub problem involves finding frequent itemsets ending in *e, d,* e, *b,* and *a.*

FP-growth finds all the frequent itemsets ending with a particular suffix by employing a divide-and-conquer strategy to split the problem into smaller sub problems. For example, suppose we are interested in finding all frequent itemsets ending in e. To do this, we must first check whether the itemset {e} itself is frequent. If it is frequent, we consider the sub problem of finding frequent itemsets ending in *de*, followed by *ce*, *be*, and *ae*. In turn, each of these sub problems are further decomposed into smaller sub problems. By merging the solutions obtained from the sub problems, all the frequent itemsets ending in *e* can be found. This divide-and-conquer approach is the key strategy employed by the FP-growth algorithm.

Suffix	Frequent Itemsets
e	{e}, {d,e}, {a,d,e}, {c,e},{a,e}
d	{d}, {c,d}, {b,c,d}, {a,c,d}, {b,d}, {a,b,d}, {a,d}
c	{c}, {b,c}, {a,b,c}, {a,c}
b	{b}, {a,b}
a	{a}

Table. The list of frequent itemsets ordered by their corresponding suffixes

FP-growth is an interesting algorithm because it illustrates how a compact representation of the transaction data set helps to efficiently generate frequent itemsets. 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 sub problems and merge the results returned by each sub problem.