Association Analysis: Basic Concepts and Algorithms

Many business enterprises accumulate large quantities of data from their day-to-day operations. For example, huge amounts of customer purchase data are collected daily at the checkout counters of grocery stores. Table 4.1 gives an example of such data, commonly known as **market basket transactions**. Each row in this table corresponds to a transaction, which contains a unique identifier labeled TID and a set of items bought by a given customer. Retailers are interested in analyzing the data to learn about the purchasing behavior of their customers. Such valuable information can be used to support a variety of business-related applications such as marketing promotions, inventory management, and customer relationship management.

This chapter presents a methodology known as **association analysis**, which is useful for discovering interesting relationships hidden in large data sets. The uncovered relationships can be represented in the form of sets of items present in many transactions, which are known as **frequent itemsets**,

Table 4.1. An example of market basket transactions.

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

or **association rules**, that represent relationships between two itemsets. For example, the following rule can be extracted from the data set shown in Table 4.1:

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

The rule suggests a relationship between the sale of diapers and beer because many customers who buy diapers also buy beer. Retailers can use these types of rules to help them identify new opportunities for cross-selling their products to the customers.

Besides market basket data, association analysis is also applicable to data from other application domains such as bioinformatics, medical diagnosis, web mining, and scientific data analysis. In the analysis of Earth science data, for example, association patterns may reveal interesting connections among the ocean, land, and atmospheric processes. Such information may help Earth scientists develop a better understanding of how the different elements of the Earth system interact with each other. Even though the techniques presented here are generally applicable to a wider variety of data sets, for illustrative purposes, our discussion will focus mainly on market basket data.

There are two key issues that need to be addressed when applying association analysis to market basket data. First, discovering patterns from a large transaction data set can be computationally expensive. Second, some of the discovered patterns may be spurious (happen simply by chance) and even for non-spurious patterns, some are more interesting than others. The remainder of this chapter is organized around these two issues. The first part of the chapter is devoted to explaining the basic concepts of association analysis and the algorithms used to efficiently mine such patterns. The second part of the chapter deals with the issue of evaluating the discovered patterns in order to help prevent the generation of spurious results and to rank the patterns in terms of some interestingness measure.

4.1 Preliminaries

This section reviews the basic terminology used in association analysis and presents a formal description of the task.

Binary Representation Market basket data can be represented in a binary format as shown in Table 4.2, 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

Table 4.2. A binary 0/1 representation of market basket data.

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

more important than its absence, an item is an **asymmetric** binary variable. This representation is a simplistic view of real market basket data because it ignores important aspects of the data such as the quantity of items sold or the price paid to purchase them. Methods for handling such non-binary data will be explained in Chapter 7.

Itemset and Support Count Let $I = \{i_1, i_2, \ldots, i_d\}$ be the set of all items in a market basket data and $T = \{t_1, t_2, \ldots, t_N\}$ be the set of all transactions. Each transaction t_i contains a subset of items chosen from I. In association analysis, a collection of zero or more items is termed an itemset. If an itemset contains k items, it is called a k-itemset. 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.

A transaction t_j is said to contain an itemset X if X is a subset of t_j. For example, the second transaction shown in Table 4.2 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, $\sigma(X)$, for an itemset X can be stated as follows:

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

where the symbol $|\cdot|$ denotes the number of elements in a set. In the data set shown in Table 4.2, the support count for {`Beer`, `Diapers`, `Milk`} is equal to two because there are only two transactions that contain all three items.

Often, the property of interest is the support, which is fraction of transactions in which an itemset occurs:

$$s(X) = \sigma(X)/N.$$

An itemset X is called frequent if $s(X)$ is greater than some user-defined threshold, *minsup*.

Association Rule An association rule is an implication expression of the form $X \longrightarrow Y$, where X and Y are disjoint itemsets, i.e., $X \cap Y = \emptyset$. The strength of an association rule can be measured in terms of its **support** and **confidence**. Support determines how often a rule is applicable to a given data set, while confidence determines how frequently items in Y appear in transactions that contain X. The formal definitions of these metrics are

$$\text{Support, } s(X \longrightarrow Y) = \frac{\sigma(X \cup Y)}{N}; \tag{4.1}$$

$$\text{Confidence, } c(X \longrightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}. \tag{4.2}$$

Example 4.1. Consider the rule {`Milk`, `Diapers`} $\longrightarrow$ {`Beer`}. Because 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 $2/3 = 0.67$. ■

Why Use Support and Confidence? Support is an important measure because a rule that has very low support might occur simply by chance. Also, from a business perspective a low support rule is unlikely to be interesting because it might not be profitable to promote items that customers seldom buy together (with the exception of the situation described in Section 4.8). For these reasons, we are interested in finding rules whose support is greater than some user-defined threshold. As will be shown in Section 4.2.1, support also has a desirable property that can be exploited for the efficient discovery of association rules.

Confidence, on the other hand, measures the reliability of the inference made by a rule. For a given rule $X \longrightarrow Y$, the higher the confidence, the more likely it is for Y to be present in transactions that contain X. Confidence also provides an estimate of the conditional probability of Y given X.

Association analysis results should be interpreted with caution. The inference made by an association rule does not necessarily imply causality. Instead, it can sometimes suggest a strong co-occurrence relationship between items in the antecedent and consequent of the rule. Causality, on the other hand, requires knowledge about which attributes in the data capture cause and effect, and typically involves relationships occurring over time (e.g., greenhouse gas emissions lead to global warming). See Section 4.7.1 for additional discussion.

Formulation of the Association Rule Mining Problem The association rule mining problem can be formally stated as follows:

Definition 4.1 (Association Rule Discovery). Given a set of transactions T, find all the rules having support $\geq minsup$ and confidence $\geq minconf$, 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, assuming that neither the left nor the right-hand side of the rule is an empty set, the total number of possible rules, R, extracted from a data set that contains d items is

$$R = 3^d - 2^{d+1} + 1. \tag{4.3}$$

The proof for this equation is left as an exercise to the readers (see Exercise 5 on page 296). Even for the small data set shown in Table 4.1, this approach requires us to compute the support and confidence for $3^6 - 2^7 + 1 = 602$ rules. More than 80% of the rules are discarded after applying $minsup = 20\%$ and $minconf = 50\%$, thus wasting most of the computations. To avoid performing needless computations, it would be useful to prune the rules early without having to compute their support and confidence values.

An initial step toward improving the performance of association rule mining algorithms is to decouple the support and confidence requirements. From Equation 4.1, notice that the support of a rule $X \longrightarrow Y$ is the same as the support of its corresponding itemset, $X \cup Y$. For example, the following rules have identical support because they involve items from the same itemset, {Beer, Diapers, Milk}:

{Beer, Diapers} ⟶ {Milk}, {Beer, Milk} ⟶ {Diapers},
{Diapers, Milk} ⟶ {Beer}, {Beer} ⟶ {Diapers, Milk},
{Milk} ⟶ {Beer,Diapers}, {Diapers} ⟶ {Beer,Milk}.

If the itemset is infrequent, then all six candidate rules can be pruned immediately without our 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.

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.

The computational requirements for frequent itemset generation are generally more expensive than those of rule generation. Efficient techniques for generating frequent itemsets and association rules are discussed in Sections 4.2 and 4.3, respectively.

4.2 Frequent Itemset Generation

A lattice structure can be used to enumerate the list of all possible itemsets. Figure 4.1 shows an itemset lattice for $I = \{a, b, c, d, e\}$. In general, a data set that contains k items can potentially generate up to $2^k - 1$ frequent itemsets, excluding the null set. Because k can be very large in many practical applications, the search space of itemsets that need to be explored is exponentially large.

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 Figure 4.2. 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 be 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. **Transaction width** is the number of items present in a transaction.

There are three main approaches for reducing the computational complexity of frequent itemset generation.

1. **Reduce the number of candidate itemsets (M).** The *Apriori* principle, described in the next section, is an effective way to eliminate some of the candidate itemsets without counting their support values.

2. **Reduce the number of comparisons.** Instead of matching each candidate itemset against every transaction, we can reduce the number of comparisons by using more advanced data structures, either to store the candidate itemsets or to compress the data set. We will discuss these strategies in Sections 4.2.4 and 4.6, respectively.

3. **Reduce the number of transactions (N).** As the size of candidate itemsets increases, fewer transactions will be supported by the itemsets.

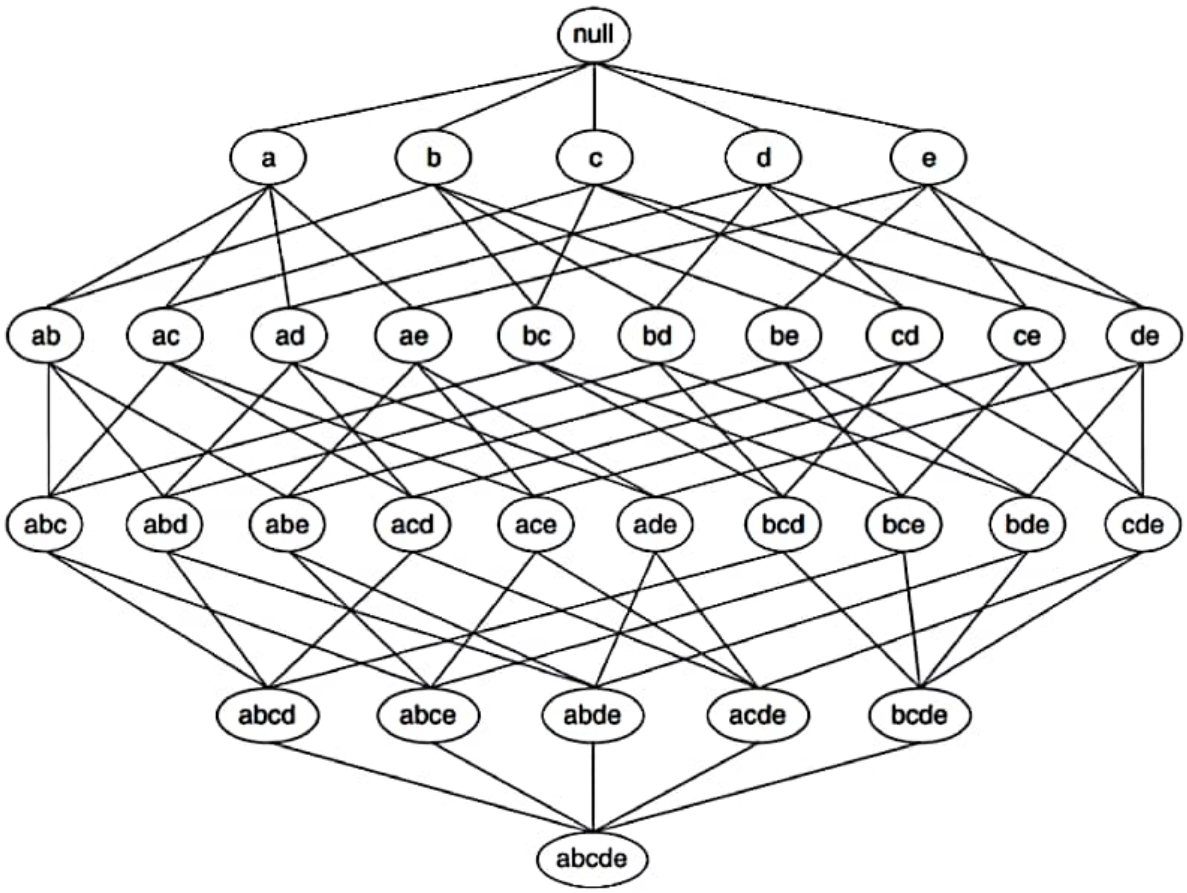


Figure 4.1. An itemset lattice.

For instance, since the width of the first transaction in Table 4.1 is 2, it would be advantageous to remove this transaction before searching for frequent itemsets of size 3 and larger. Algorithms that employ such a strategy are discussed in the Bibliographic Notes.

4.2.1 The *Apriori* Principle

This section describes how the support measure can be used to reduce the number of candidate itemsets explored during frequent itemset generation. The use of support for pruning candidate itemsets is guided by the following principle.

Theorem 4.1 (*Apriori* Principle). *If an itemset is frequent, then all of its subsets must also be frequent.*

To illustrate the idea behind the *Apriori* principle, consider the itemset lattice shown in Figure 4.3. Suppose $\{c, d, e\}$ is a frequent itemset. Clearly, any transaction that contains $\{c, d, e\}$ must also contain its subsets, $\{c, d\}$, $\{c, e\}$,

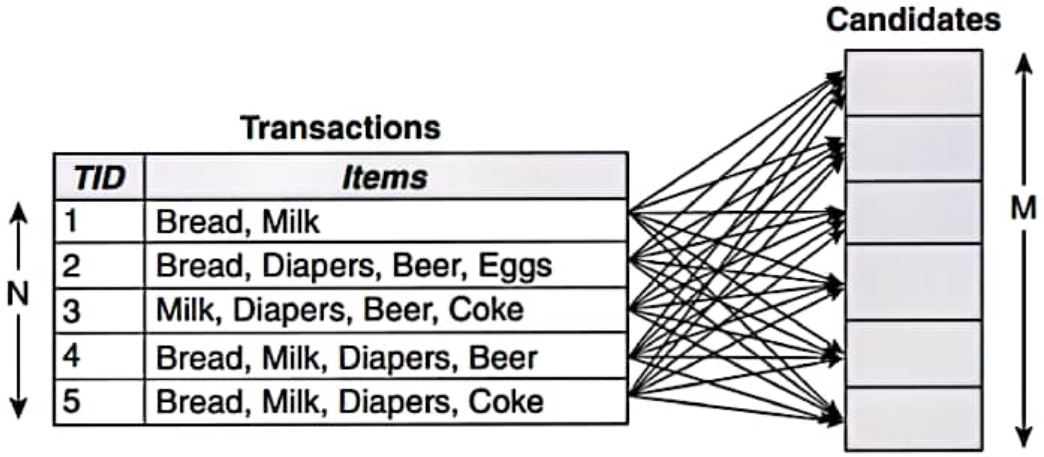


Figure 4.2. Counting the support of candidate itemsets.

$\{d, e\}$, $\{c\}$, $\{d\}$, and $\{e\}$. As a result, if $\{c, d, e\}$ is frequent, then all subsets of $\{c, d, e\}$ (i.e., the shaded itemsets in this figure) must also be frequent.

Conversely, if an itemset such as $\{a, b\}$ is infrequent, then all of its supersets must be infrequent too. As illustrated in Figure 4.4, the entire subgraph containing the supersets of $\{a, b\}$ can be pruned immediately once $\{a, b\}$ is found to be infrequent. This strategy of trimming the exponential search space based on the support measure is known as **support-based pruning**. Such a pruning strategy is made possible by a key property of the support measure, namely, that 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.

Definition 4.2 (Anti-monotone Property). A measure f possesses the anti-monotone property if for every itemset X that is a proper subset of itemset Y, i.e. $X \subset Y$, we have $f(Y) \leq f(X)$.

More generally, a large number of measures–see Section 4.7.1–can be applied to itemsets to evaluate various properties of itemsets. As will be shown in the next section, any measure that has the anti-monotone property can be incorporated directly into an itemset mining algorithm to effectively prune the exponential search space of candidate itemsets.

4.2.2 Frequent Itemset Generation in the *Apriori* Algorithm

Apriori is the first association rule mining algorithm that pioneered the use of support-based pruning to systematically control the exponential growth of candidate itemsets. Figure 4.5 provides a high-level illustration of the frequent itemset generation part of the *Apriori* algorithm for the transactions shown in

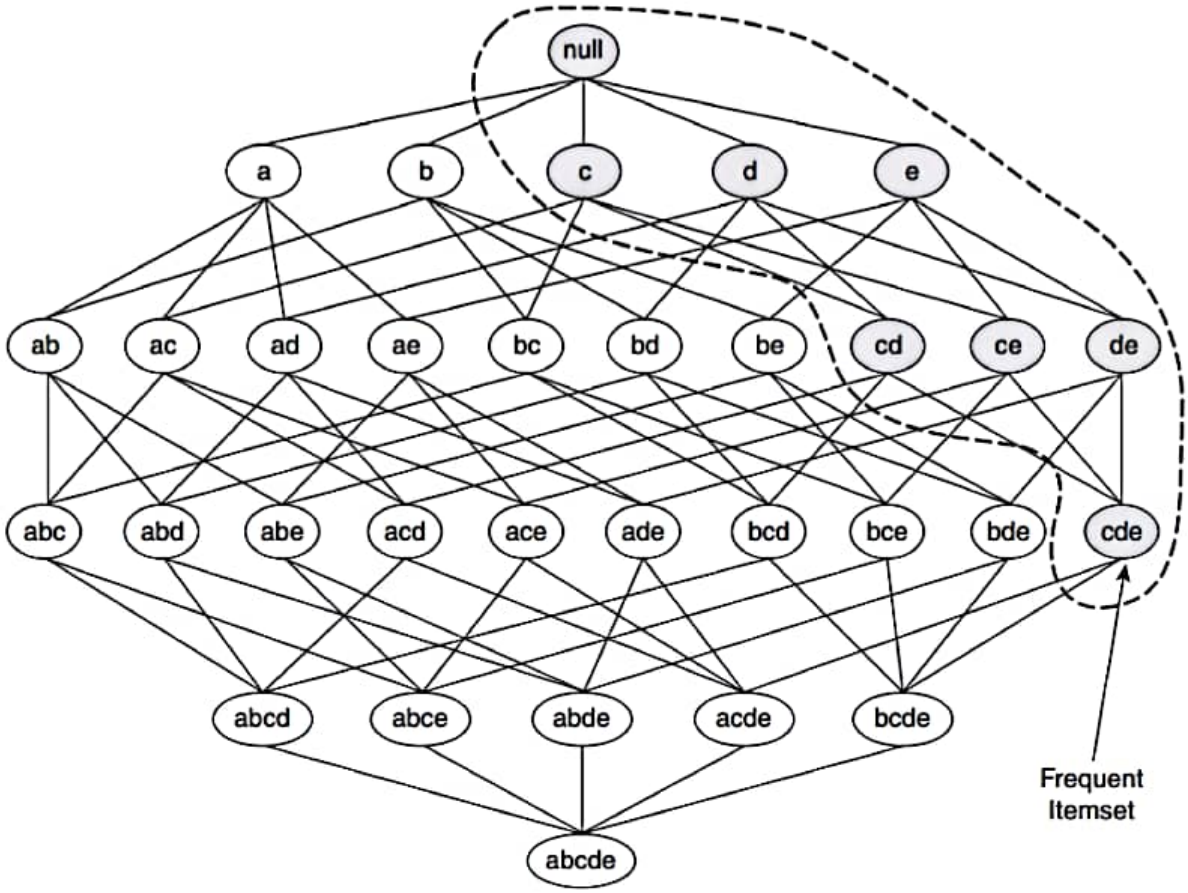


Figure 4.3. An illustration of the *Apriori* principle. If $\{c, d, e\}$ is frequent, then all subsets of this itemset are frequent.

Table 4.1. 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 1-itemset. After counting their supports, the candidate itemsets {`Cola`} and {`Eggs`} are discarded because they appear in fewer than three transactions. In the next iteration, candidate 2-itemsets are generated using only the frequent 1-itemsets because the *Apriori* principle ensures that all supersets of the infrequent 1-itemsets must be infrequent. Because there are only four frequent 1-itemsets, the number of candidate 2-itemsets generated by the algorithm is $\binom{4}{2} = 6$. 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 $\binom{6}{3} = 20$ 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`,

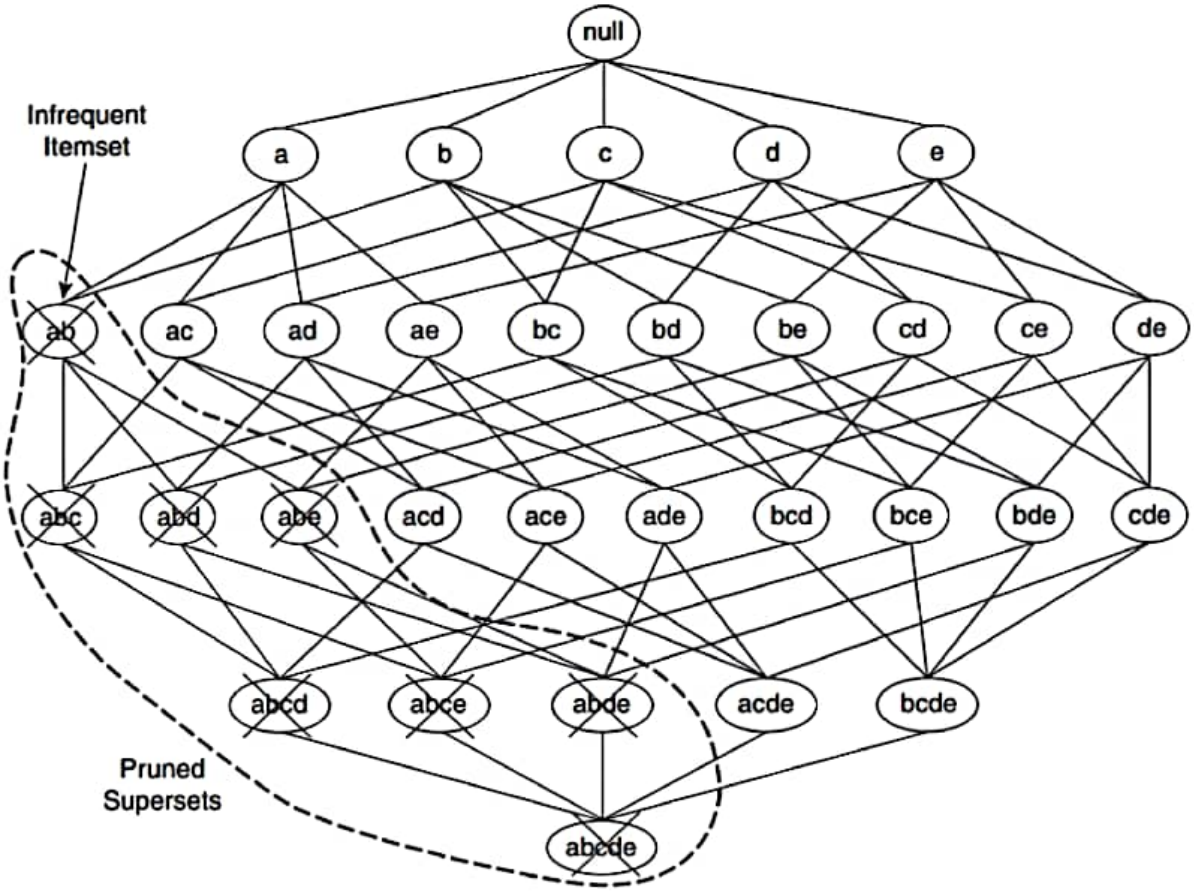


Figure 4.4. An illustration of support-based pruning. If $\{a, b\}$ is infrequent, then all supersets of $\{a, b\}$ are infrequent.

`Diapers`, `Milk`}. However, even though the subsets of {`Bread`, `Diapers`, `Milk`} are frequent, the itemset itself is not.

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.

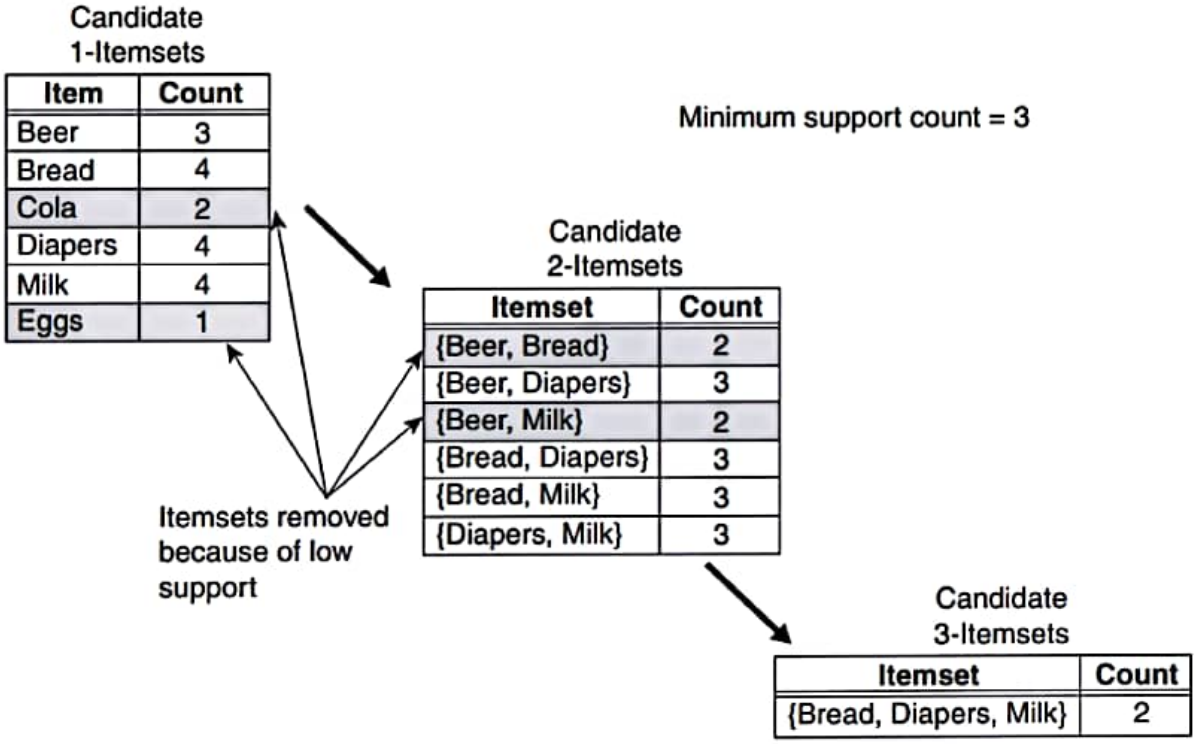


Figure 4.5. Illustration of frequent itemset generation using the *Apriori* algorithm.

The pseudocode for the frequent itemset generation part of the *Apriori* algorithm is shown in Algorithm 4.1. Let C_k denote the set of candidate k-itemsets and F_k denote the set of frequent k-itemsets:

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

- Next, the algorithm will iteratively generate new candidate k-itemsets and prune unnecessary candidates that are guaranteed to be infrequent given the frequent $(k-1)$-itemsets found in the previous iteration (steps 5 and 6). Candidate generation and pruning is implemented using the functions candidate-gen and candidate-prune, which are described in Section 4.2.3.

- To count the support of the candidates, the algorithm needs to make an additional pass over the data set (steps 7–12). The subset function is used to determine all the candidate itemsets in C_k that are contained in each transaction t. The implementation of this function is described in Section 4.2.4.

- After counting their supports, the algorithm eliminates all candidate itemsets whose support counts are less than $N \times minsup$ (step 13).
- The algorithm terminates when there are no new frequent itemsets generated, i.e., $F_k = \emptyset$ (step 14).

The frequent itemset generation part of the *Apriori* algorithm has two important characteristics. First, it is a **level-wise** algorithm; i.e., it traverses the itemset lattice one level at a time, from frequent 1-itemsets to the maximum size of frequent itemsets. Second, it employs a **generate-and-test** strategy for finding frequent itemsets. At each iteration (level), new candidate itemsets are generated from the frequent itemsets found in the previous iteration. The support for each candidate is then counted and tested against the *minsup* threshold. The total number of iterations needed by the algorithm is $k_{\max} + 1$, where $k_{\max}$ is the maximum size of the frequent itemsets.

4.2.3 Candidate Generation and Pruning

The candidate-gen and candidate-prune functions shown in Steps 5 and 6 of Algorithm 4.1 generate candidate itemsets and prunes unnecessary ones by performing the following two operations, respectively:

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

Algorithm 4.1 Frequent itemset generation of the *Apriori* algorithm.

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

2. **Candidate Pruning.** This operation eliminates some of the candidate k-itemsets using support-based pruning, i.e. by removing k-itemsets whose subsets are known to be infrequent in previous iterations. Note that this pruning is done without computing the actual support of these k-itemsets (which could have required comparing them against each transaction).

Candidate Generation

In principle, there are many ways to generate candidate itemsets. An effective candidate generation procedure must be complete and non-redundant. A candidate generation procedure is said to be *complete* if it does not omit any frequent itemsets. To ensure completeness, the set of candidate itemsets must subsume the set of all frequent itemsets, i.e., $\forall k : F_k \subseteq C_k$. A candidate generation procedure is *non-redundant* if it does not generate the same candidate itemset more than once. For example, the candidate itemset $\{a, b, c, d\}$ can be generated in many ways—by merging $\{a, b, c\}$ with $\{d\}$, $\{b, d\}$ with $\{a, c\}$, $\{c\}$ with $\{a, b, d\}$, etc. Generation of duplicate candidates leads to wasted computations and thus should be avoided for efficiency reasons. Also, an effective candidate generation procedure should avoid generating too many unnecessary candidates. A candidate itemset is unnecessary if at least one of its subsets is infrequent, and thus, eliminated in the candidate pruning step.

Next, we will briefly describe several candidate generation procedures, including the one used by the candidate-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 whose subsets are infrequent (see Figure 4.6). The number of candidate itemsets generated at level k is equal to $\binom{d}{k}$, where d is the total number of items. Although candidate generation is rather trivial, candidate pruning becomes extremely expensive because a large number of itemsets must be examined.

$\mathbf{F_{k-1} \times F_1}$ Method An alternative method for candidate generation is to extend each frequent $(k-1)$-itemset with frequent items that are not part of the $(k-1)$-itemset. Figure 4.7 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`}.

The procedure is complete because every frequent k-itemset is composed of a frequent $(k-1)$-itemset and a frequent 1-itemset. Therefore, all frequent

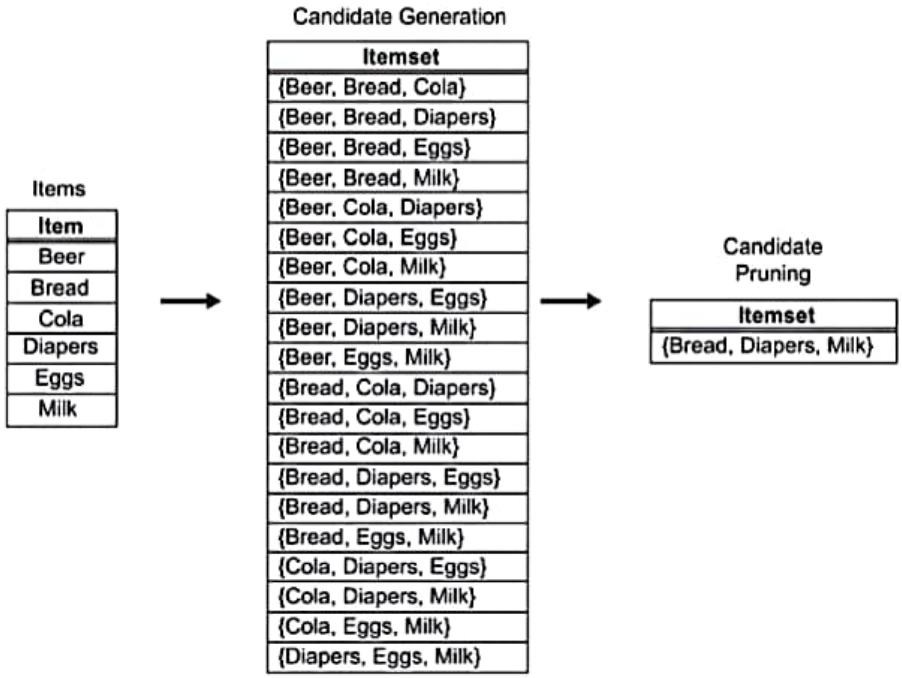


Figure 4.6. A brute-force method for generating candidate 3-itemsets.

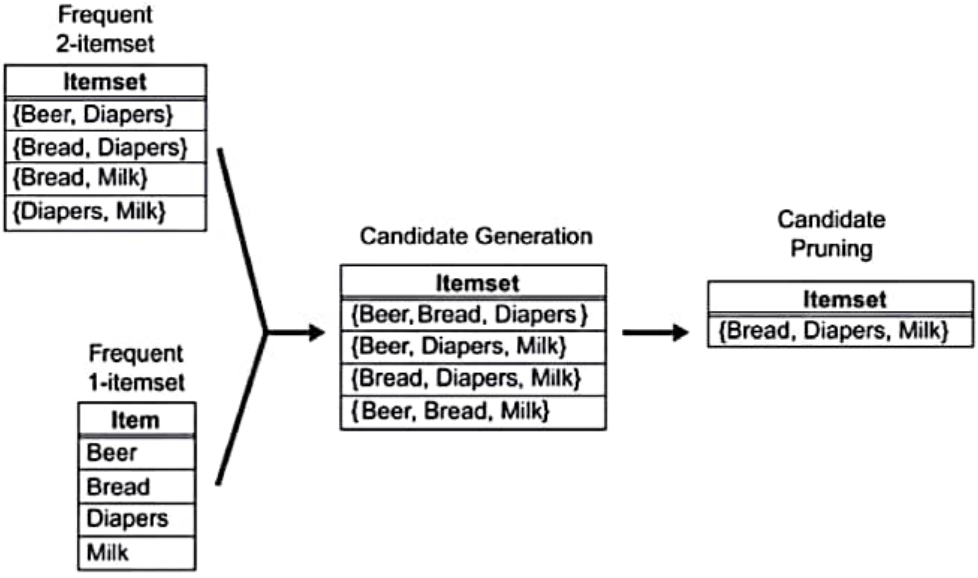


Figure 4.7. Generating and pruning candidate k-itemsets by merging a frequent $(k-1)$-itemset with a frequent item. Note that some of the candidates are unnecessary because their subsets are infrequent.

k-itemsets are part of the candidate k-itemsets generated by this procedure. Figure 4.7 shows that the $\mathbf{F}_{k-1} \times \mathbf{F}_1$ candidate generation method only produces four candidate 3-itemsets, instead of the $\binom{6}{3} = 20$ itemsets produced by the brute-force method. The $\mathbf{F}_{k-1} \times \mathbf{F}_1$ method generates lower number

of candidates because every candidate is guaranteed to contain at least one frequent $(k-1)$-itemset. While this procedure is a substantial improvement over the brute-force method, it can still produce a large number of unnecessary candidates, as the remaining subsets of a candidate itemset can still be infrequent.

Note that the approach discussed above does not prevent the same candidate itemset from being generated more than once. For instance, {Bread, Diapers, Milk} can be generated by merging {Bread, Diapers} with {Milk}, {Bread, Milk} with {Diapers}, or {Diapers, Milk} with {Bread}. One way to avoid generating duplicate candidates is by ensuring that the items in each frequent itemset are kept sorted in their lexicographic order. For example, itemsets such as {Bread, Diapers}, {Bread, Diapers, Milk}, and {Diapers, Milk} follow lexicographic order as the items within every itemset are arranged alphabetically. Each frequent $(k-1)$-itemset X is then extended with frequent items that are lexicographically larger than the items in X. For example, the itemset {Bread, Diapers} can be augmented with {Milk} because Milk is lexicographically larger than Bread and Diapers. However, we should not augment {Diapers, Milk} with {Bread} nor {Bread, Milk} with {Diapers} because they violate the lexicographic ordering condition. Every candidate k-itemset is thus generated exactly once, by merging the lexicographically largest item with the remaining $k-1$ items in the itemset. If the $\mathbf{F}_{k-1} \times \mathbf{F}_1$ method is used in conjunction with lexicographic ordering, then only two candidate 3-itemsets will be produced in the example illustrated in Figure 4.7. {Beer, Bread, Diapers} and {Beer, Bread, Milk} will not be generated because {Beer, Bread} is not a frequent 2-itemset.

$\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ Method This candidate generation procedure, which is used in the candidate-gen function of the *Apriori* algorithm, merges a pair of frequent $(k-1)$-itemsets only if their first $k-2$ items, arranged in lexicographic order, 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, arranged lexicographically. A and B are merged if they satisfy the following conditions:

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

Note that in this case, $a_{k-1} \neq b_{k-1}$ because A and B are two distinct itemsets. The candidate k-itemset generated by merging A and B consists of the first $k-2$ common items followed by a_{k-1} and b_{k-1} in lexicographic order. This candidate generation procedure is complete, because for every lexicographically ordered frequent k-itemset, there exists two lexicographically

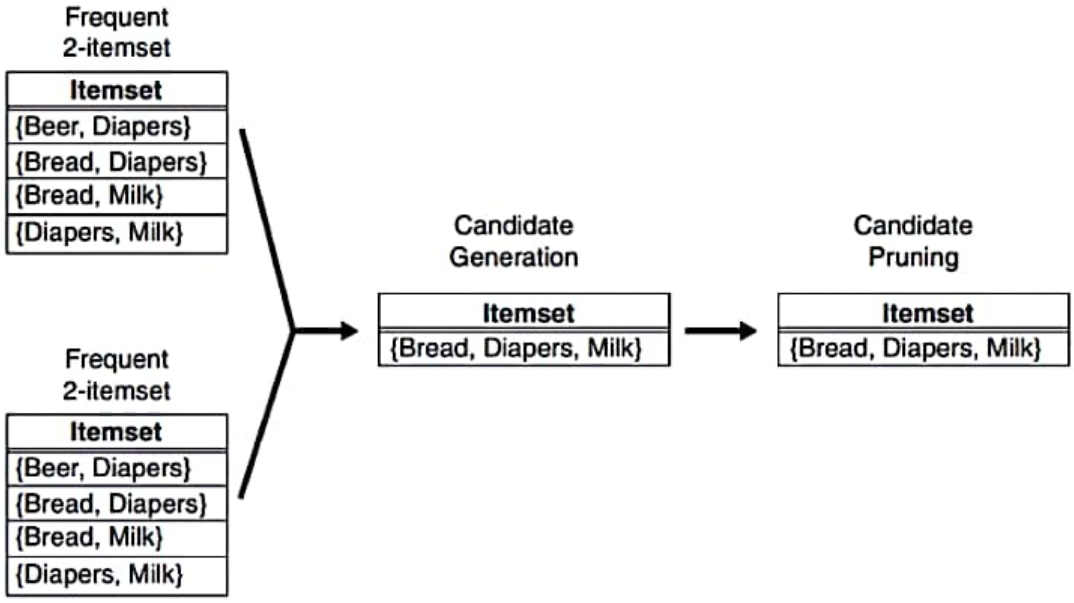


Figure 4.8. Generating and pruning candidate k-itemsets by merging pairs of frequent $(k-1)$-itemsets.

ordered frequent $(k-1)$-itemsets that have identical items in the first $k-2$ positions.

In Figure 4.8, 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. This example illustrates both the completeness of the candidate generation procedure and the advantages of using lexicographic ordering to prevent duplicate candidates. Also, if we order the frequent $(k-1)$-itemsets according to their lexicographic rank, itemsets with identical first $k-2$ items would take consecutive ranks. As a result, the $\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ candidate generation method would consider merging a frequent itemset only with ones that occupy the next few ranks in the sorted list, thus saving some computations.

Figure 4.8 shows that the $\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ candidate generation procedure results in only one candidate 3-itemset. This is a considerable reduction from the four candidate 3-itemsets generated by the $\mathbf{F}_{k-1} \times \mathbf{F}_1$ method. This is because the $\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ method ensures that every candidate k-itemset contains at least two frequent $(k-1)$-itemsets, thus greatly reducing the number of candidates that are generated in this step.

Note that there can be multiple ways of merging two frequent $(k-1)$-itemsets in the $\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ procedure, one of which is merging if their first

$k-2$ items are identical. An alternate approach could be to merge two frequent $(k-1)$-itemsets A and B if the last $k-2$ items of A are identical to the first $k-2$ itemsets of B. For example, {`Bread`, `Diapers`} and {`Diapers`, `Milk`} could be merged using this approach to generate the candidate 3-itemset {`Bread`, `Diapers`, `Milk`}. As we will see later, this alternate $\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ procedure is useful in generating sequential patterns, which will be discussed in Chapter 7.

Candidate Pruning

To illustrate the candidate pruning operation for a candidate k-itemset, $X = \{i_1, i_2, \ldots, i_k\}$, consider its k proper subsets, $X - \{i_j\}$ $(\forall j = 1, 2, \ldots, k)$. If any of them are infrequent, then X is immediately pruned by using the *Apriori* principle. Note that we don't need to explicitly ensure that all subsets of X of size less than $k-1$ are frequent (see Exercise 8). This approach greatly reduces the number of candidate itemsets considered during support counting. For the brute-force candidate generation method, candidate pruning requires checking only k subsets of size $k-1$ for each candidate k-itemset. However, since the $\mathbf{F}_{k-1} \times \mathbf{F}_1$ candidate generation strategy ensures that at least one of the $(k-1)$-size subsets of every candidate k-itemset is frequent, we only need to check for the remaining $k-1$ subsets. Likewise, the $\mathbf{F}_{k-1} \times \mathbf{F}_{k-1}$ strategy requires examining only $k-2$ subsets of every candidate k-itemset, since two of its $(k-1)$-size subsets are already known to be frequent in the candidate generation step.

4.2.4 Support Counting

Support counting is the process of determining the frequency of occurrence for every candidate itemset that survives the candidate pruning step. Support counting is implemented in steps 6 through 11 of Algorithm 4.1. A brute-force approach for doing this is to compare each transaction against every candidate itemset (see Figure 4.2) and to update the support counts of candidates contained in a 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 $\binom{5}{3} = 10$ itemsets of size 3 contained in this transaction. Some of the itemsets may correspond to the candidate 3-itemsets under investigation, in which case, their support counts are incremented. Other subsets of t that do not correspond to any candidates can be ignored.

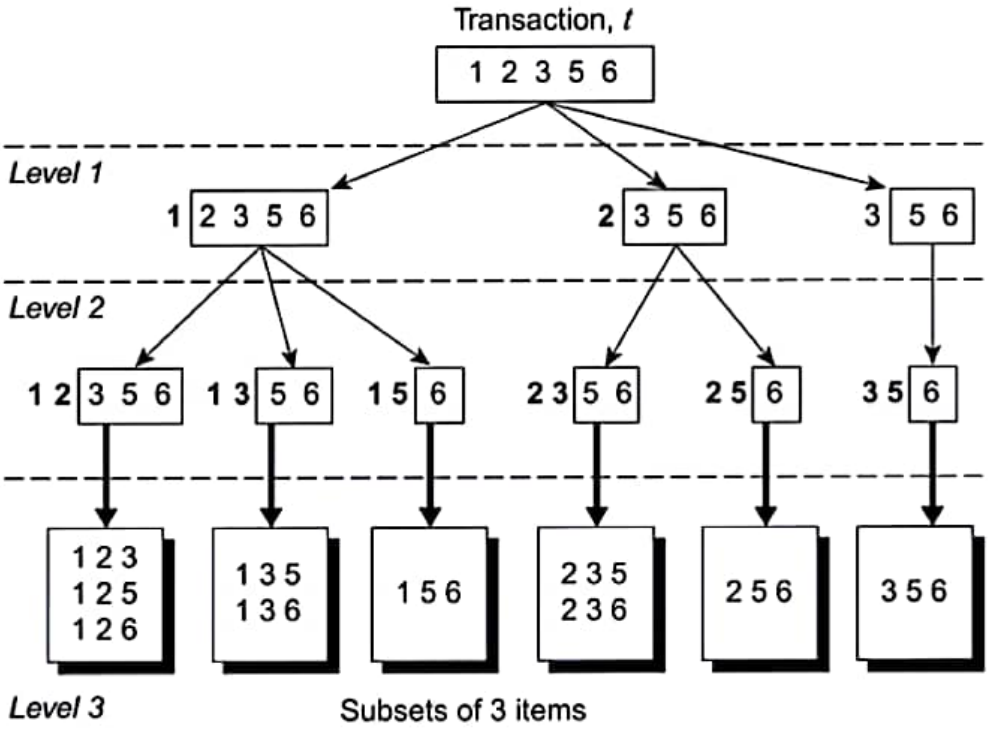


Figure 4.9. Enumerating subsets of three items from a transaction t.

Figure 4.9 shows a systematic way for enumerating the 3-itemsets contained in t. Assuming that each itemset keeps its items in increasing lexicographic order, an itemset can be enumerated by specifying the smallest item first, followed by the larger items. For instance, given $t = \{1, 2, 3, 5, 6\}$, all the 3-itemsets contained in t must begin with item 1, 2, or 3. It is not possible to construct a 3-itemset that begins with items 5 or 6 because there are only two items in t whose labels are greater than or equal to 5. The number of ways to specify the first item of a 3-itemset contained in t is illustrated by the Level 1 prefix tree structure depicted in Figure 4.9. For instance, 1 2 3 5 6 represents a 3-itemset that begins with item 1, followed by two more items chosen from the set $\{2, 3, 5, 6\}$.

After fixing the first item, the prefix tree structure at Level 2 represents the number of ways to select the second item. For example, 1 2 3 5 6 corresponds to itemsets that begin with the prefix (1 2) and are followed by the items 3, 5, or 6. Finally, the prefix tree structure at Level 3 represents the complete set of 3-itemsets contained in t. For example, the 3-itemsets that begin with prefix $\{1\ 2\}$ are $\{1, 2, 3\}$, $\{1, 2, 5\}$, and $\{1, 2, 6\}$, while those that begin with prefix $\{2\ 3\}$ are $\{2, 3, 5\}$ and $\{2, 3, 6\}$.

The prefix tree structure shown in Figure 4.9 demonstrates how itemsets contained in a transaction can be systematically enumerated, i.e., by specifying

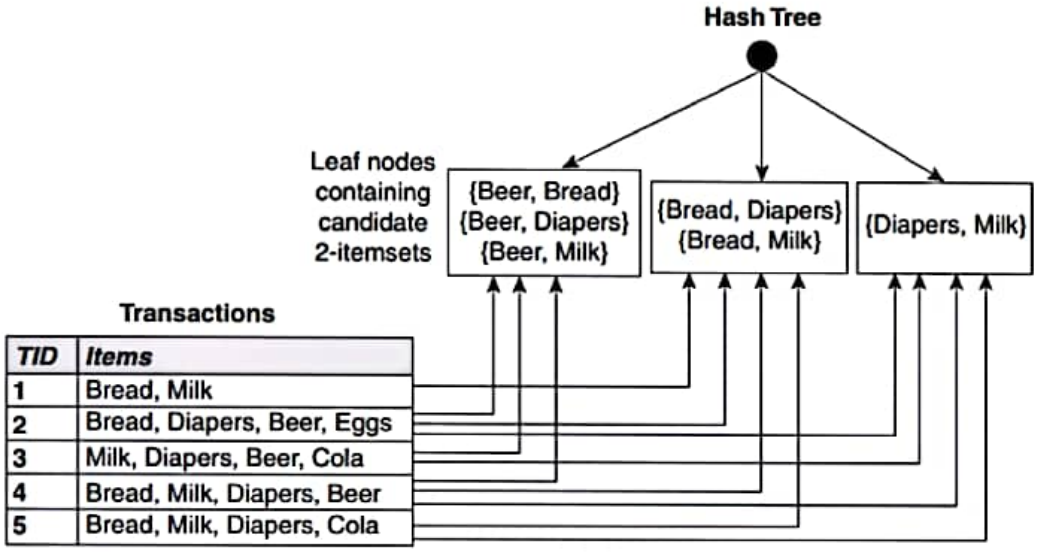


Figure 4.10. Counting the support of itemsets using hash structure.

their items one by one, from the leftmost item to the rightmost item. We still have to determine whether each enumerated 3-itemset corresponds to an existing candidate itemset. If it matches one of the candidates, then the support count of the corresponding candidate is incremented. In the next section, we illustrate how this matching operation can be performed efficiently using a hash tree structure.

Support Counting Using a Hash Tree*

In the *Apriori* algorithm, candidate itemsets are partitioned into different buckets and stored in a hash tree. During support counting, itemsets contained in each transaction are also hashed into their appropriate buckets. That way, instead of comparing each itemset in the transaction with every candidate itemset, it is matched only against candidate itemsets that belong to the same bucket, as shown in Figure 4.10.

Figure 4.11 shows an example of a hash tree structure. Each internal node of the tree uses the following hash function, $h(p) = (p - 1) \bmod 3$, where mode refers to the modulo (remainder) operator, to determine which branch of the current node should be followed next. For example, items 1, 4, and 7 are hashed to the same branch (i.e., the leftmost branch) because they have the same remainder after dividing the number by 3. All candidate itemsets are stored at the leaf nodes of the hash tree. The hash tree shown in Figure 4.11 contains 15 candidate 3-itemsets, distributed across 9 leaf nodes.

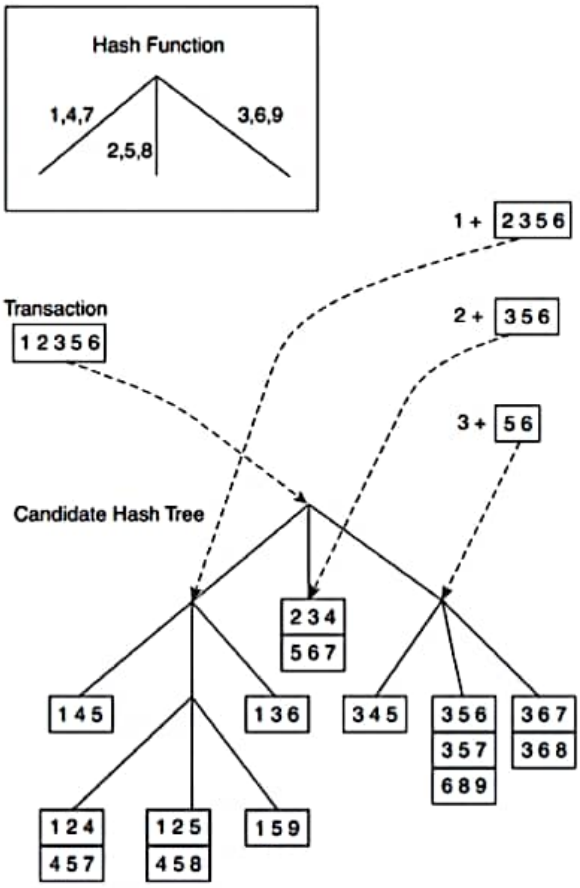


Figure 4.11. Hashing a transaction at the root node of a hash tree.

Consider the transaction, $t = \{1, 2, 3, 5, 6\}$. To update the support counts of the candidate itemsets, the hash tree must be traversed in such a way that all the leaf nodes containing candidate 3-itemsets belonging to t must be visited at least once. Recall that the 3-itemsets contained in t must begin with items 1, 2, or 3, as indicated by the Level 1 prefix tree structure shown in Figure 4.9. Therefore, at the root node of the hash tree, the items 1, 2, and 3 of the transaction are hashed separately. Item 1 is hashed to the left child of the root node, item 2 is hashed to the middle child, and item 3 is hashed to the right child. At the next level of the tree, the transaction is hashed on the second item listed in the Level 2 tree structure shown in Figure 4.9. For example, after hashing on item 1 at the root node, items 2, 3, and 5 of the transaction are hashed. Based on the hash function, items 2 and 5 are hashed to the middle child, while item 3 is hashed to the right child, as shown in Figure 4.12. This process continues until the leaf nodes of the hash tree are reached. The candidate itemsets stored at the visited leaf nodes are compared against the transaction. If a candidate is a subset of the transaction, its support count is incremented. Note that not all the leaf nodes are visited

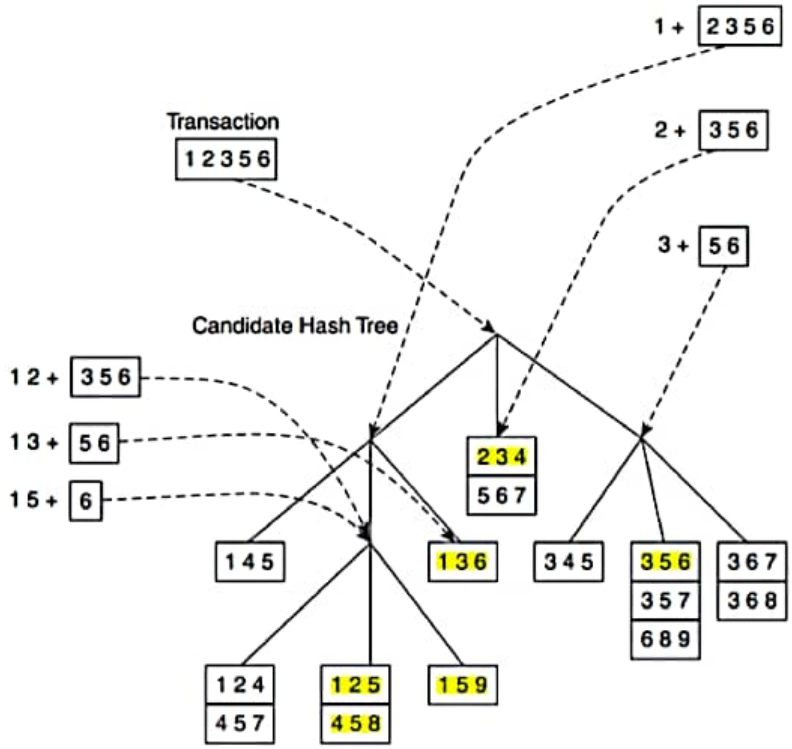


Figure 4.12. Subset operation on the leftmost subtree of the root of a candidate hash tree.

while traversing the hash tree, which helps in reducing the computational cost. In this example, 5 out of the 9 leaf nodes are visited and 9 out of the 15 itemsets are compared against the transaction.

4.2.5 Computational Complexity

The computational complexity of the *Apriori* algorithm, which includes both its runtime and storage, can be affected by the following factors.

Support Threshold Lowering the support threshold often results in more itemsets being declared as frequent. This has an adverse effect on the computational complexity of the algorithm because more candidate itemsets must be generated and counted at every level, as shown in Figure 4.13. The maximum size of frequent itemsets also tends to increase with lower support thresholds. This increases the total number of iterations to be performed by the *Apriori* algorithm, further increasing the computational cost.

Number of Items (Dimensionality) As the number of items increases, more space will be needed to store the support counts of items. If the number of frequent items also grows with the dimensionality of the data, the runtime and

Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. Thomas Seidl
Max Berrendorf, Julian Busch

Knowledge Discovery and Data Mining I

WS 2018/19

Exercise 4: Hash Tree, FP-Growth, Association Rules

Exercise 4-1 Hash-Tree

(a) **Construction.** Using the hash function

$$h(x) = x \mod 3 \tag{1}$$

construct a hash tree with maximum number of itemsets in inner nodes equal to 4 given the following set of candidates:

(1, 9, 11)	(2, 5, 10)	(3, 6, 8)	(4, 7, 9)	(6, 12, 13)	(9, 12, 14)
(1, 10, 12)	(2, 5, 12)	(3, 7, 10)	(4, 7, 13)	(6, 12, 14)	(10, 11, 15)
(2, 4, 7)	(2, 9, 10)	(3, 12, 14)	(5, 7, 9)	(8, 11, 11)	(12, 12, 15)
(2, 5, 8)	(3, 3, 5)	(4, 5, 8)	(5, 7, 13)	(8, 11, 15)	(14, 14, 15)

In the root node, the itemsets are splitted according to the hash value of the first item in the itemset. Hence, after the root node we have 3 child nodes with content:

N_0	N_1	N_2
(3, 3, 5)	(1, 9, 11)	(2, 4, 7)
(3, 6, 8)	(1, 10, 12)	(2, 5, 8)
(3, 7, 10)	(4, 5, 8)	(2, 5, 10)
(3, 12, 14)	(4, 7, 9)	(2, 5, 12)
(6, 12, 13)	(4, 7, 13)	(2, 9, 10)
(6, 12, 14)	(10, 11, 15)	(5, 7, 9)
(9, 12, 14)		(5, 7, 13)
(12, 12, 15)		(8, 11, 11)
		(8, 11, 15)
		(14, 14, 15)

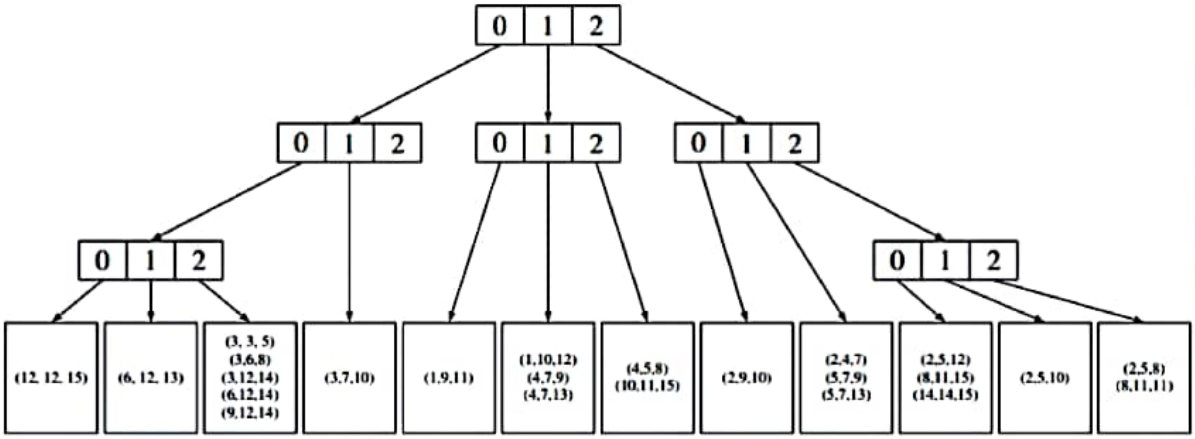
As the fill degree of all nodes is larger 4, all have to be split, now according to the second item.

N_{00}	N^*_{01}	N^*_{10}	N^*_{11}	N^*_{12}	N^*_{20}	N^*_{21}	N_{22}
(3, 3, 5)	(3, 7, 10)	(1, 9, 11)	(1, 10, 12)	(4, 5, 8)	(2, 9, 10)	(2, 4, 7)	(2, 5, 8)
(3, 6, 8)			(4, 7, 9)	(10, 11, 15)		(5, 7, 9)	(2, 5, 10)
(3, 12, 14)			(4, 7, 13)			(5, 7, 13)	(2, 5, 12)
(6, 12, 13)							(8, 11, 11)
(6, 12, 14)							(8, 11, 15)
(9, 12, 14)							(14, 14, 15)
(12, 12, 15)							

Here, only N_{00} and N_{22} have a higher fill degree than allowed (the leaf nodes are marked with *). Hence, they are splitted again, this time using the third item.

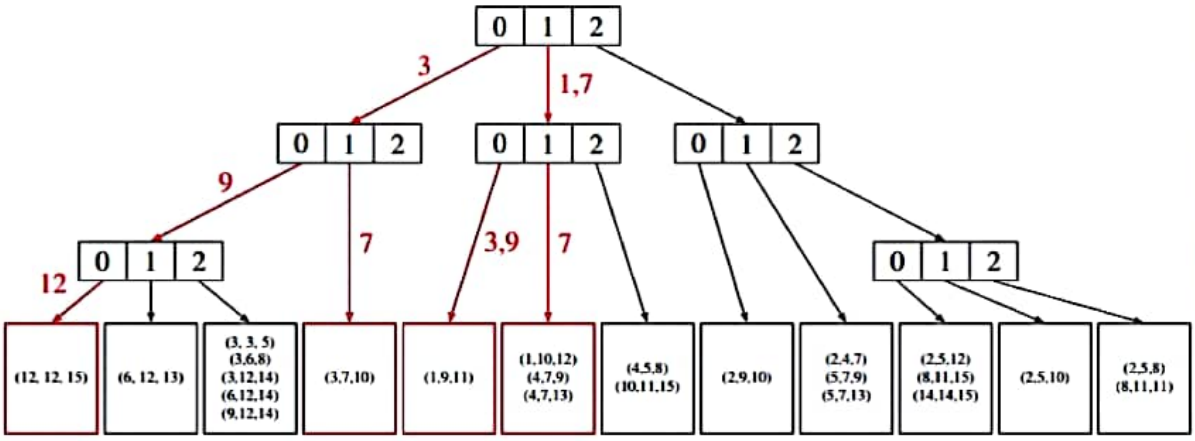
N^*_{000}	N^*_{001}	N^*_{002}	N^*_{220}	N^*_{221}	N^*_{222}
(12, 12, 15)	(6, 12, 13)	(3, 3, 5)	(2, 5, 12)	(2, 5, 10)	(2,5,8)
		(3, 6, 8)	(8, 11, 15)		(8, 11, 11)
		(3, 12, 14)	(14, 14, 15)		
		(6, 12, 14)			
		(9, 12, 14)			

Although N_{002}'s fill degree is larger then 4, there is no remaining item to be used for further splitting. Hence, the hash-tree construction finishes. The final hash-tree is depicted below:



(b) **Counting.** Given the transaction $t = (t_1, \ldots, t_5) = (1, 3, 7, 9, 12)$, find all candidates of length $k = 3$ in the previously constructed tree from exercise (a). In absolute and relative numbers: How many candidates need to be refined? How many nodes are visited?

Applying the hash function to the transaction gives $(1, 0, 1, 0, 0)$. The following diagram shows the accessed nodes. A detailed explanation follows below.



(i) Depth $d = 1$. Compute hash values for $t_1, \ldots, t_{n-k+d} = t_3$:

$$h(1) = 1 \qquad h(3) = 0 \qquad h(7) = 1 \tag{2}$$

. Continue search in N_0, N_1 (i.e. exclude N_2).

(ii) Depth $d = 2$. Additionally compute $h(t_4) = h(9) = 0$.

- In N_0 reached by item t_2, the nodes for hash values 0 (N_{00} reached by t_4) and 1 (N^*_{01} reached by t_3) are of interest.
- In N_1 reached by item t_1 and t_3, the nodes for hash values 0 (N^*_{10} reached by t_2 and t_4) and 1 (N^*_{11} reached by t_3) are of interest.

(iii) Depth $d = 3$. Additionally compute $h(t_5) = h(12) = 0$.

- In N_{00} reached by $t_2, t_4 = 3, 9$ continue with N^*_{000}.
- In N^*_{01} reached by $t_2, t_3 = 3, 7$ search for $t_2, t_3, t_4 = 3, 7, 9$ and $t_2, t_3, t_5 = 3, 7, 12$. Both are not found.
- In N^*_{10} reached by
  - $t_1t_2 = 1, 3$,
  - $t_1t_4 = 1, 9$, or
  - $t_3t_4 = 7, 9$

  search for
  - $t_1t_2t_3 = 1, 3, 7$
  - $t_1t_2t_4 = 1, 3, 9$
  - $t_1t_2t_5 = 1, 3, 12$
  - $t_1t_4t_5 = 1, 9, 12$
  - $t_3t_4t_5 = 7, 9, 12$

  None of them is found.
- In N^*_{11} reached by $t_1, t_3 = 1, 7$search for $t_1, t_3, t_4 = 1, 7, 9$ and $t_1, t_3, t_5 = 1, 7, 12$. Both are not found.

(iv) Depth $d = 4$.

- In N^*_{000} reached by $t_2, t_4, t_5 = 3, 9, 12$ search for this transaction. It is not found.

In total, $4/12 \approx 33\%$ of the leaf nodes are visited, $8/18 \approx 44\%$ of the nodes are visited and $6/24 = 25\%$ of the candidates are compared. As result, none of the candidates is supported by the transaction.