Chapter 4

Association Pattern Mining

"The pattern of the prodigal is: rebellion, ruin, repentance, reconciliation, restoration."—Edwin Louis Cole

4.1 Introduction

The classical problem of association pattern mining is defined in the context of supermarket data containing sets of items bought by customers, which are referred to as *transactions*. The goal is to determine *associations* between groups of items bought by customers, which can intuitively be viewed as k-way correlations between items. The most popular model for association pattern mining uses the frequencies of sets of items as the quantification of the level of association. The discovered sets of items are referred to as *large itemsets*, *frequent itemsets*, or *frequent patterns*. The association pattern mining problem has a wide variety of applications:

1. *Supermarket data:* The supermarket application was the original motivating scenario in which the association pattern mining problem was proposed. This is also the reason that the term *itemset* is used to refer to a frequent pattern in the context of supermarket *items* bought by a customer. The determination of frequent itemsets provides useful insights about target marketing and shelf placement of the items.

2. *Text mining:* Because text data is often represented in the bag-of-words model, frequent pattern mining can help in identifying co-occurring terms and keywords. Such co-occurring terms have numerous text-mining applications.

3. *Generalization to dependency-oriented data types:* The original frequent pattern mining model has been generalized to many dependency-oriented data types, such as time-series data, sequential data, spatial data, and graph data, with a few modifications. Such models are useful in applications such as Web log analysis, software bug detection, and spatiotemporal event detection.

C. C. Aggarwal, *Data Mining: The Textbook*, DOI 10.1007/978-3-319-14142-8_4

© Springer International Publishing Switzerland 2015

4. *Other major data mining problems:* Frequent pattern mining can be used as a subroutine to provide effective solutions to many data mining problems such as clustering, classification, and outlier analysis.

Because the frequent pattern mining problem was originally proposed in the context of market basket data, a significant amount of terminology used to describe both the data (e.g., *transactions*) and the output (e.g., *itemsets*) is borrowed from the supermarket analogy. From an application-neutral perspective, a frequent pattern may be defined as a frequent subset, defined on the universe of all possible sets. Nevertheless, because the market basket terminology has been used popularly, this chapter will be consistent with it.

Frequent itemsets can be used to generate *association rules* of the form $X \Rightarrow Y$, where X and Y are sets of items. A famous example of an association rule, which has now become part[1] of the data mining folklore, is $\{Beer\} \Rightarrow \{Diapers\}$. This rule suggests that buying beer makes it more likely that diapers will also be bought. Thus, there is a certain directionality to the implication that is quantified as a conditional probability. Association rules are particularly useful for a variety of target market applications. For example, if a supermarket owner discovers that $\{Eggs, Milk\} \Rightarrow \{Yogurt\}$ is an association rule, he or she can promote yogurt to customers who often buy eggs and milk. Alternatively, the supermarket owner may place yogurt on shelves that are located in proximity to eggs and milk.

The frequency-based model for association pattern mining is very popular because of its simplicity. However, the raw frequency of a pattern is not quite the same as the statistical significance of the underlying correlations. Therefore, numerous models for frequent pattern mining have been proposed that are based on statistical significance. This chapter will also explore some of these alternative models, which are also referred to as *interesting patterns*.

This chapter is organized as follows. Section 4.2 introduces the basic model for association pattern mining. The generation of association rules from frequent itemsets is discussed in Sect. 4.3. A variety of algorithms for frequent pattern mining are discussed in Sect. 4.4. This includes the *Apriori* algorithm, a number of enumeration tree algorithms, and a suffix-based recursive approach. Methods for finding interesting frequent patterns are discussed in Sect. 4.5. Meta-algorithms for frequent pattern mining are discussed in Sect. 4.6. Section 4.7 discusses the conclusions and summary.

4.2 The Frequent Pattern Mining Model

The problem of association pattern mining is naturally defined on unordered set-wise data. It is assumed that the database $\mathcal{T}$ contains a set of n transactions, denoted by $T_1 \dots T_n$. Each transaction T_i is drawn on the universe of items U and can also be represented as a multidimensional record of dimensionality, $d = |U|$, containing only binary attributes. Each binary attribute in this record represents a particular item. The value of an attribute in this record is 1 if that item is present in the transaction, and 0 otherwise. In practical settings, the universe of items U is very large compared to the typical number of items in each transaction T_i. For example, a supermarket database may have tens of thousands of items, and a single transaction will typically contain less than 50 items. This property is often leveraged in the design of frequent pattern mining algorithms.

An *itemset* is a set of items. A k-itemset is an itemset that contains exactly k items. In other words, a k-itemset is a set of items of cardinality k. The fraction of transactions

[1] This rule was derived in some early publications on supermarket data. No assertion is made here about the likelihood of such a rule appearing in an arbitrary supermarket data set.

Table 4.1: Example of a snapshot of a market basket data set

tid	Set of items	Binary representation
1	$\{Bread, Butter, Milk\}$	110010
2	$\{Eggs, Milk, Yogurt\}$	000111
3	$\{Bread, Cheese, Eggs, Milk\}$	101110
4	$\{Eggs, Milk, Yogurt\}$	000111
5	$\{Cheese, Milk, Yogurt\}$	001011

in $T_1 \ldots T_n$ in which an itemset occurs as a subset provides a crisp quantification of its frequency. This frequency is also known as the *support*.

Definition 4.2.1 (Support) *The support of an itemset I is defined as the fraction of the transactions in the database $\mathcal{T} = \{T_1 \ldots T_n\}$ that contain I as a subset.*

The support of an itemset I is denoted by $sup(I)$. Clearly, items that are correlated will frequently occur together in transactions. Such itemsets will have high support. Therefore, the frequent pattern mining problem is that of determining itemsets that have the requisite level of *minimum support*.

Definition 4.2.2 (Frequent Itemset Mining) *Given a set of transactions $\mathcal{T} = \{T_1 \ldots T_n\}$, where each transaction T_i is a subset of items from U, determine all itemsets I that occur as a subset of at least a predefined fraction minsup of the transactions in $\mathcal{T}$.*

The predefined fraction *minsup* is referred to as the minimum support. While the default convention in this book is to assume that *minsup* refers to a fractional relative value, it is also sometimes specified as an absolute integer value in terms of the raw number of transactions. This chapter will always assume the convention of a relative value, unless specified otherwise. Frequent patterns are also referred to as frequent itemsets, or large itemsets. This book will use these terms interchangeably.

The unique identifier of a transaction is referred to as a *transaction identifier*, or *tid* for short. The frequent itemset mining problem may also be stated more generally in set-wise form.

Definition 4.2.3 (Frequent Itemset Mining: Set-wise Definition) *Given a set of sets $\mathcal{T} = \{T_1 \ldots T_n\}$, where each element of the set T_i is drawn on the universe of elements U, determine all sets I that occur as a subset of at least a predefined fraction minsup of the sets in $\mathcal{T}$.*

As discussed in Chap. 1, binary multidimensional data and set data are equivalent. This equivalence is because each multidimensional attribute can represent a set element (or item). A value of 1 for a multidimensional attribute corresponds to inclusion in the set (or transaction). Therefore, a transaction data set (or set of sets) can also be represented as a multidimensional binary database whose dimensionality is equal to the number of items.

Consider the transactions illustrated in Table 4.1. Each transaction is associated with a unique transaction identifier in the leftmost column, and contains a baskets of items that were bought together at the same time. The right column in Table 4.1 contains the binary multidimensional representation of the corresponding basket. The attributes of this binary representation are arranged in the order $\{Bread, Butter, Cheese, Eggs, Milk, Yogurt\}$. In

this database of 5 transactions, the *support* of $\{Bread, Milk\}$ is $2/5 = 0.4$ because both items in this basket occur in 2 out of a total of 5 transactions. Similarly, the support of $\{Cheese, Yogurt\}$ is 0.2 because it appears in only the last transaction. Therefore, if the minimum support is set to 0.3, then the itemset $\{Bread, Milk\}$ will be reported but not the itemset $\{Cheese, Yogurt\}$.

The number of frequent itemsets is generally very sensitive to the minimum support level. Consider the case where a minimum support level of 0.3 is used. Each of the items $Bread$, $Milk$, $Eggs$, $Cheese$, and $Yogurt$ occur in more than 2 transactions, and can therefore be considered frequent items at a minimum support level of 0.3. These items are frequent 1-itemsets. In fact, the only item that is not frequent at a support level of 0.3 is $Butter$. Furthermore, the frequent 2-itemsets at a minimum support level of 0.3 are $\{Bread, Milk\}$, $\{Eggs, Milk\}$, $\{Cheese, Milk\}$, $\{Eggs, Yogurt\}$, and $\{Milk, Yogurt\}$. The only 3-itemset reported at a support level of 0.3 is $\{Eggs, Milk, Yogurt\}$. On the other hand, if the minimum support level is set to 0.2, it corresponds to an absolute support value of only 1. In such a case, every subset of every transaction will be reported. Therefore, the use of lower minimum support levels yields a larger number of frequent patterns. On the other hand, if the support level is too high, then no frequent patterns will be found. Therefore, an appropriate choice of the support level is crucial for discovering a set of frequent patterns with meaningful size.

When an itemset I is contained in a transaction, all its subsets will also be contained in the transaction. Therefore, the support of any subset J of I will always be at least equal to that of I. This property is referred to as the *support monotonicity property*.

Property 4.2.1 (Support Monotonicity Property) *The support of every subset J of I is at least equal to that of the support of itemset I.*

$$sup(J) \geq sup(I) \quad \forall J \subseteq I \tag{4.1}$$

The monotonicity property of support implies that every subset of a frequent itemset will also be frequent. This is referred to as the *downward closure property*.

Property 4.2.2 (Downward Closure Property) *Every subset of a frequent itemset is also frequent.*

The downward closure property of frequent patterns is algorithmically very convenient because it provides an important constraint on the inherent structure of frequent patterns. This constraint is often leveraged by frequent pattern mining algorithms to prune the search process and achieve greater efficiency. Furthermore, the downward closure property can be used to create concise representations of frequent patterns, wherein only the *maximal* frequent subsets are retained.

Definition 4.2.4 (Maximal Frequent Itemsets) *A frequent itemset is maximal at a given minimum support level minsup, if it is frequent, and no superset of it is frequent.*

In the example of Table 4.1, the itemset $\{Eggs, Milk, Yogurt\}$ is a maximal frequent itemset at a minimum support level of 0.3. However, the itemset $\{Eggs, Milk\}$ is not maximal because it has a superset that is also frequent. Furthermore, the set of *maximal* frequent patterns at a minimum support level of 0.3 is $\{Bread, Milk\}$, $\{Cheese, Milk\}$, and $\{Eggs, Milk, Yogurt\}$. Thus, there are only 3 maximal frequent itemsets, whereas the number of frequent itemsets in the entire transaction database is 11. All frequent itemsets can be derived from the maximal patterns by enumerating the subsets of the maximal frequent

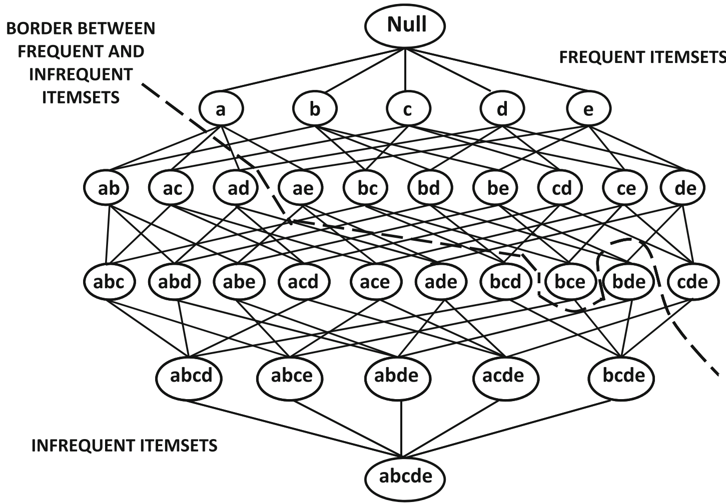

Figure 4.1: The itemset lattice

patterns. Therefore, the maximal patterns can be considered condensed representations of the frequent patterns. However, this condensed representation does not retain information about the support values of the subsets. For example, the support of $\{Eggs, Milk, Yogurt\}$ is 0.4, but it does not provide any information about the support of $\{Eggs, Milk\}$, which is 0.6. A different condensed representation, referred to as *closed frequent itemsets*, is able to retain support information as well. The notion of closed frequent itemsets will be studied in detail in Chap. 5.

An interesting property of itemsets is that they can be conceptually arranged in the form of a *lattice of itemsets*. This lattice contains one node for each of the $2^{|U|}$ sets drawn from the universe of items U. An edge exists between a pair of nodes, if the corresponding sets differ by exactly one item. An example of an itemset lattice of size $2^5 = 32$ on a universe of 5 items is illustrated in Fig. 4.1. The lattice represents the search space of frequent patterns. All frequent pattern mining algorithms, implicitly or explicitly, traverse this search space to determine the frequent patterns.

The lattice is separated into frequent and infrequent itemsets by a *border*, which is illustrated by a dashed line in Fig. 4.1. All itemsets above this border are frequent, whereas those below the border are infrequent. Note that all maximal frequent itemsets are adjacent to this border of itemsets. Furthermore, any valid border representing a true division between frequent and infrequent itemsets will always respect the downward closure property.

4.3 Association Rule Generation Framework

Frequent itemsets can be used to generate *association rules*, with the use of a measure known as the *confidence*. The confidence of a rule $X \Rightarrow Y$ is the conditional probability that a transaction contains the set of items Y, given that it contains the set X. This probability is estimated by dividing the support of itemset $X \cup Y$ with that of itemset X.

Definition 4.3.1 (Confidence) *Let X and Y be two sets of items. The confidence $conf(X \cup Y)$ of the rule $X \cup Y$ is the conditional probability of $X \cup Y$ occurring in a*

transaction, given that the transaction contains X. Therefore, the confidence $conf(X \Rightarrow Y)$ is defined as follows:

$$conf(X \Rightarrow Y) = \frac{sup(X \cup Y)}{sup(X)}. \tag{4.2}$$

The itemsets X and Y are said to be the *antecedent* and the *consequent* of the rule, respectively. In the case of Table 4.1, the support of $\{Eggs, Milk\}$ is 0.6, whereas the support of $\{Eggs, Milk, Yogurt\}$ is 0.4. Therefore, the confidence of the rule $\{Eggs, Milk\} \Rightarrow \{Yogurt\}$ is $(0.4/0.6) = 2/3$.

As in the case of support, a minimum confidence threshold $minconf$ can be used to generate the most relevant association rules. Association rules are defined using both *support* and *confidence* criteria.

Definition 4.3.2 (Association Rules) *Let X and Y be two sets of items. Then, the rule $X \Rightarrow Y$ is said to be an association rule at a minimum support of minsup and minimum confidence of $minconf$, if it satisfies both the following criteria:*

1. *The support of the itemset $X \cup Y$ is at least minsup.*

2. *The confidence of the rule $X \Rightarrow Y$ is at least $minconf$.*

The first criterion ensures that a sufficient number of transactions are relevant to the rule; therefore, it has the required critical mass for it to be considered relevant to the application at hand. The second criterion ensures that the rule has sufficient strength in terms of conditional probabilities. Thus, the two measures quantify different aspects of the association rule.

The overall framework for association rule generation uses two phases. These phases correspond to the two criteria in Definition 4.3.2, representing the support and confidence constraints.

1. In the first phase, all the frequent itemsets are generated at the minimum support of *minsup*.

2. In the second phase, the association rules are generated from the frequent itemsets at the minimum confidence level of *minconf*.

The first phase is more computationally intensive and is, therefore, the more interesting part of the process. The second phase is relatively straightforward. Therefore, the discussion of the first phase will be deferred to the remaining portion of this chapter, and a quick discussion of the (more straightforward) second phase is provided here.

Assume that a set of frequent itemsets $\mathcal{F}$ is provided. For each itemset $I \in \mathcal{F}$, a simple way of generating the rules would be to partition the set I into all possible combinations of sets X and $Y = I - X$, such that $I = X \cup Y$. The confidence of each rule $X \Rightarrow Y$ can then be determined, and it can be retained if it satisfies the minimum confidence requirement. Association rules also satisfy a confidence monotonicity property.

Property 4.3.1 (Confidence Monotonicity) *Let X_1, X_2, and I be itemsets such that $X_1 \subset X_2 \subset I$. Then the confidence of $X_2 \Rightarrow I - X_2$ is at least that of $X_1 \Rightarrow I - X_1$.*

$$conf(X_2 \Rightarrow I - X_2) \geq conf(X_1 \Rightarrow I - X_1) \tag{4.3}$$

This property follows directly from definition of confidence and the property of support monotonicity. Consider the rules $\{Bread\} \Rightarrow \{Butter, Milk\}$ and $\{Bread, Butter\} \Rightarrow \{Milk\}$. The second rule is redundant with respect to the first because it will have the same support, but a confidence that is no less than the first. Because of confidence monotonicity, it is possible to report only the non-redundant rules. This issue is discussed in detail in the next chapter.

4.4 Frequent Itemset Mining Algorithms

In this section, a number of popular algorithms for frequent itemset generation will be discussed. Because there are a large number of frequent itemset mining algorithms, the focus of the chapter will be to discuss specific algorithms in detail to introduce the reader to the key tricks in algorithmic design. These tricks are often reusable across different algorithms because the same *enumeration tree framework* is used by virtually all frequent pattern mining algorithms.

4.4.1 Brute Force Algorithms

For a universe of items U, there are a total of $2^{|U|} - 1$ distinct subsets, excluding the empty set. All 2^5 subsets for a universe of 5 items are illustrated in Fig. 4.1. Therefore, one possibility would be to generate all these *candidate* itemsets, and count their support against the transaction database $\mathcal{T}$. In the frequent itemset mining literature, the term *candidate itemsets* is commonly used to refer to itemsets that might *possibly* be frequent (or *candidates* for being frequent). These candidates need to be verified against the transaction database by *support counting*. To count the support of an itemset, we would need to check whether a given itemset I is a subset of each transaction $T_i \in \mathcal{T}$. Such an exhaustive approach is likely to be impractical, when the universe of items U is large. Consider the case where $d = |U| = 1000$. In that case, there are a total of $2^{1000} > 10^{300}$ candidates. To put this number in perspective, if the fastest computer available today were somehow able to process one candidate in one elementary machine cycle, then the time required to process all candidates would be hundreds of orders of magnitude greater than the age of the universe. Therefore, this is not a practical solution.

Of course, one can make the brute-force approach faster by observing that no $(k+1)$-patterns are frequent if no k-patterns are frequent. This observation follows directly from the downward closure property. Therefore, one can enumerate and count the support of all the patterns with increasing length. In other words, one can enumerate and count the support of all patterns containing one item, two items, and so on, until for a certain length l, none of the candidates of length l turn out to be frequent. For sparse transaction databases, the value of l is typically very small compared to $|U|$. At this point, one can terminate. This is a significant improvement over the previous approach because it requires the enumeration of $\sum_{i=1}^{l} \binom{|U|}{i} \ll 2^{|U|}$ candidates. Because the longest frequent itemset is of *much* smaller length than $|U|$ in sparse transaction databases, this approach is orders of magnitude faster. However, the resulting computational complexity is still not satisfactory for large values of U. For example, when $|U| = 1000$ and $l = 10$, the value of $\sum_{i=1}^{10} \binom{|U|}{i}$ is of the order of 10^{23}. This value is still quite large and outside reasonable computational capabilities available today.

One observation is that even a very minor and rather blunt application of the downward closure property made the algorithm hundreds of orders of magnitude faster. Many of the fast algorithms for itemset generation use the downward closure property in a more refined way, both to generate the candidates and to prune them before counting. Algorithms for

frequent pattern mining search the lattice of possibilities (or candidates) for frequent patterns (see Fig. 4.1) and use the transaction database to count the support of candidates in this lattice. Better efficiencies can be achieved in a frequent pattern mining algorithm by using one or more of the following approaches:

1. Reducing the size of the explored search space (lattice of Fig. 4.1) by pruning candidate *itemsets* (lattice nodes) using tricks, such as the *downward closure* property.
2. Counting the support of each candidate more efficiently by pruning *transactions* that are known to be irrelevant for counting a candidate itemset.
3. Using compact data structures to represent either candidates or transaction databases that support efficient counting.

The first algorithm that used an effective pruning of the search space with the use of the downward closure property was the *Apriori* algorithm.

4.4.2 The Apriori Algorithm

The *Apriori* algorithm uses the downward closure property in order to prune the candidate search space. The downward closure property imposes a clear structure on the set of frequent patterns. In particular, information about the *infrequency* of itemsets can be leveraged to generate the superset candidates more carefully. Thus, if an itemset is infrequent, there is little point in counting the support of its superset candidates. This is useful for avoiding wasteful counting of support levels of itemsets that are known not to be frequent. The *Apriori* algorithm generates candidates with smaller length k first and counts their supports before generating candidates of length $(k+1)$. The resulting frequent k-itemsets are used to restrict the number of $(k+1)$-candidates with the downward closure property. Candidate generation and support counting of patterns with increasing length is interleaved in *Apriori*. Because the counting of candidate supports is the most expensive part of the frequent pattern generation process, it is extremely important to keep the number of candidates low.

For ease in description of the algorithm, it will be assumed that the items in U have a lexicographic ordering, and therefore an itemset $\{a, b, c, d\}$ can be treated as a (lexicographically ordered) string *abcd* of items. This can be used to impose an ordering among itemsets (patterns), which is the same as the order in which the corresponding strings would appear in a dictionary.

The *Apriori* algorithm starts by counting the supports of the individual items to generate the frequent 1-itemsets. The 1-itemsets are combined to create candidate 2-itemsets, whose support is counted. The frequent 2-itemsets are retained. In general, the frequent itemsets of length k are used to generate the candidates of length $(k + 1)$ for increasing values of k. Algorithms that count the support of candidates with increasing length are referred to as *level-wise* algorithms. Let $\mathcal{F}_k$ denote the set of frequent k-itemsets, and $\mathcal{C}_k$ denote the set of candidate k-itemsets. The core of the approach is to iteratively generate the $(k + 1)$-candidates $\mathcal{C}_{k+1}$ from frequent k-itemsets in $\mathcal{F}_k$ already found by the algorithm. The frequencies of these $(k + 1)$-candidates are counted with respect to the transaction database. While generating the $(k + 1)$-candidates, the search space may be pruned by checking whether all k-subsets of $\mathcal{C}_{k+1}$ are included in $\mathcal{F}_k$. So, how does one generate the relevant $(k + 1)$-candidates in $\mathcal{C}_{k+1}$ from frequent k-patterns in $\mathcal{F}_k$?

If a pair of itemsets X and Y in $\mathcal{F}_k$ have $(k - 1)$ items in common, then a join between them using the $(k - 1)$ common items will create a candidate itemset of size $(k + 1)$. For example, the two 3-itemsets $\{a, b, c\}$ (or *abc* for short) and $\{a, b, d\}$ (or *abd* for short), when

Algorithm *Apriori*(Transactions: $\mathcal{T}$, Minimum Support: *minsup*)
begin
  $k = 1$;
  $\mathcal{F}_1 = \{$ All Frequent 1-itemsets $\}$;
  while $\mathcal{F}_k$ is not empty **do begin**
    Generate $\mathcal{C}_{k+1}$ by joining itemset-pairs in $\mathcal{F}_k$;
    Prune itemsets from $\mathcal{C}_{k+1}$ that violate downward closure;
    Determine $\mathcal{F}_{k+1}$ by support counting on $(\mathcal{C}_{k+1}, \mathcal{T})$ and retaining
        itemsets from $\mathcal{C}_{k+1}$ with support at least *minsup*;
    $k = k + 1$;
  end;
  return$(\cup_{i=1}^{k} \mathcal{F}_i)$;
end

Figure 4.2: The *Apriori* algorithm

joined together on the two common items a and b, will yield the candidate 4-itemset *abcd*. Of course, it is possible to join other frequent patterns to create the same candidate. One might also join *abc* and *bcd* to achieve the same result. Suppose that all four of the 3-subsets of *abcd* are present in the set of frequent 3-itemsets. One can create the candidate 4-itemset in $\binom{4}{2} = 6$ different ways. To avoid redundancy in candidate generation, the convention is to impose a lexicographic ordering on the items and use the first $(k-1)$ items of the itemset for the join. Thus, in this case, the only way to generate *abcd* would be to join using the first two items a and b. Therefore, the itemsets *abc* and *abd* would need to be joined to create *abcd*. Note that, if either of *abc* and *abd* are *not* frequent, then *abcd* will *not* be generated as a candidate using this join approach. Furthermore, in such a case, it is assured that *abcd* will not be frequent because of the downward closure property of frequent itemsets. Thus, the downward closure property ensures that the candidate set generated using this approach does not miss any itemset that is truly frequent. As we will see later, this *non-repetitive* and *exhaustive* way of generating candidates can be interpreted in the context of a conceptual hierarchy of the patterns known as the *enumeration tree*. Another point to note is that the joins can usually be performed very efficiently. This efficiency is because, if the set $\mathcal{F}_k$ is sorted in lexicographic (dictionary) order, all itemsets with a common set of items in the first $k-1$ positions will appear contiguously, allowing them to be located easily.

A *level-wise pruning trick* can be used to further reduce the size of the $(k+1)$-candidate set. All the k-subsets (i.e., subsets of cardinality k) of an itemset $I \in \mathcal{C}_{k+1}$ need to be present in $\mathcal{F}_k$ because of the downward closure property. Otherwise, it is guaranteed that the itemset I is not frequent. Therefore, it is checked whether all k-subsets of each itemset $I \in \mathcal{C}_{k+1}$ are present in $\mathcal{F}_k$. If this is not the case, then such itemsets I are removed from $\mathcal{C}_{k+1}$.

After the candidate itemsets $\mathcal{C}_{k+1}$ of size $(k+1)$ have been generated, their support can be determined by counting the number of occurrences of each candidate in the transaction database $\mathcal{T}$. Only the candidate itemsets that have the required minimum support are retained to create the set of $(k+1)$-frequent itemsets $\mathcal{F}_{k+1} \subseteq \mathcal{C}_{k+1}$. In the event that the set $\mathcal{F}_{k+1}$ is empty, the algorithm terminates. At termination, the union $\cup_{i=1}^{k} \mathcal{F}_i$ of the frequent patterns of different sizes is reported as the final output of the algorithm.

The overall algorithm is illustrated in Fig. 4.2. The heart of the algorithm is an iterative loop that generates $(k+1)$-candidates from frequent k-patterns for successively higher values of k and counts them. The three main operations of the algorithm are candidate

generation, pruning, and support counting. Of these, the support counting process is the most expensive one because it depends on the size of the transaction database $\mathcal{T}$. The level-wise approach ensures that the algorithm is relatively efficient at least from a disk-access cost perspective. This is because each set of candidates in $\mathcal{C}_{k+1}$ can be counted in a single pass over the data without the need for random disk accesses. The number of passes over the data is, therefore, equal to the cardinality of the longest frequent itemset in the data. Nevertheless, the counting procedure is still quite expensive especially if one were to use the naive approach of checking whether each itemset is a subset of a transaction. Therefore, efficient support counting procedures are necessary.

4.4.2.1 Efficient Support Counting

To perform support counting, *Apriori* needs to *efficiently* examined whether each candidate itemset is present in a transaction. This is achieved with the use of a data structure known as the *hash tree*. The hash tree is used to carefully organize the candidate patterns in $\mathcal{C}_{k+1}$ for more efficient counting. Assume that the items in the transactions and the candidate itemsets are sorted lexicographically. A hash tree is a tree with a fixed degree of the internal nodes. Each internal node is associated with a random hash function that maps to the index of the different children of that node in the tree. A leaf node of the hash tree contains a list of lexicographically sorted itemsets, whereas an interior node contains a hash table. Every itemset in $\mathcal{C}_{k+1}$ is contained in exactly one leaf node of the hash tree. The hash functions in the interior nodes are used to decide which candidate itemset belongs to which leaf node with the use of a methodology described below.

It may be assumed that all interior nodes use the same hash function $f(\cdot)$ that maps to $[0 \ldots h-1]$. The value of h is also the branching degree of the hash tree. A candidate itemset in $\mathcal{C}_{k+1}$ is mapped to a leaf node of the tree by defining a path from the root to the leaf node with the use of these hash functions at the internal nodes. Assume that the root of the hash tree is level 1, and all successive levels below it increase by 1. As before, assume that the items in the candidates and transactions are arranged in lexicographically sorted order. At an interior node in level i, a hash function is applied to the ith item of a candidate itemset $I \in \mathcal{C}_{k+1}$ to decide which branch of the hash tree to follow for the candidate itemset. The tree is constructed recursively in top-down fashion, and a minimum threshold is imposed on the number of candidates in the leaf node to decide where to terminate the hash tree extension. The candidate itemsets in the leaf node are stored in sorted order.

To perform the counting, all possible candidate k-itemsets in $\mathcal{C}_{k+1}$ that are subsets of a transaction $T_j \in \mathcal{T}$ are discovered in a single exploration of the hash tree. To achieve this goal, all possible paths in the hash tree, whose leaves *might* contain subset itemsets of the transaction T_j, are discovered using a recursive traversal. The selection of the relevant leaf nodes is performed by recursive traversal as follows. At the root node, all branches are followed such that any of the items in the transaction T_j hash to one of the branches. At a given interior node, if the ith item of the transaction T_j was last hashed (at the parent node), then all items following it in the transaction are hashed to determine the possible children to follow. Thus, by following all these paths, the relevant leaf nodes in the tree are determined. The candidates in the leaf node are stored in sorted order and can be compared efficiently to the transaction T_j to determine whether they are relevant. This process is repeated for each transaction to determine the final support count of each itemset in $\mathcal{C}_{k+1}$.

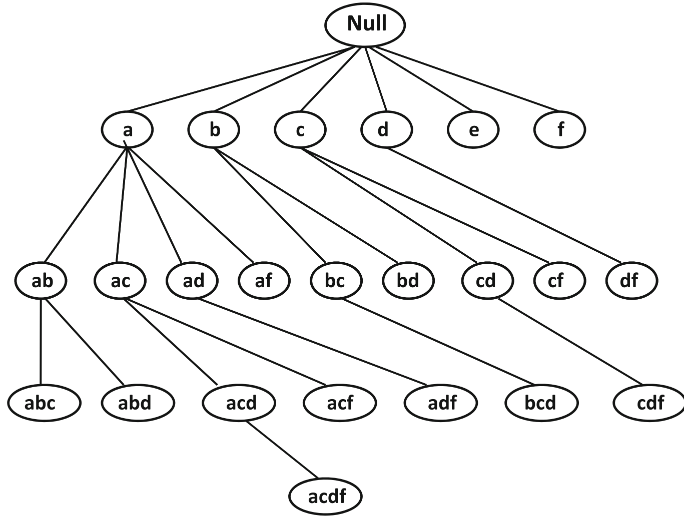

Figure 4.3: The lexicographic or enumeration tree of frequent itemsets

4.4.3 Enumeration-Tree Algorithms

These algorithms are based on set enumeration concepts, in which the different candidate itemsets are generated in a tree-like structure known as the *enumeration tree*, which is a subgraph of the lattice of itemsets introduced in Fig. 4.1. This tree-like structure is also referred to as a lexicographic tree because it is dependent on an upfront lexicographic ordering among the items. The candidate patterns are generated by growing this lexicographic tree. This tree can be grown in a wide variety of different strategies to achieve different trade-offs between storage, disk access costs, and computational efficiency. Because most of the discussion in this section will use this structure as a base for algorithmic development, this concept will be discussed in detail here. The main characteristic of the enumeration tree (or lexicographic tree) is that it provides an abstract hierarchical representation of the itemsets. This representation is leveraged by frequent pattern mining algorithms for systematic exploration of the candidate patterns in a non-repetitive way. The final output of these algorithms can also be viewed as an enumeration tree structure that is defined only on the frequent itemsets. The enumeration tree is defined on the frequent itemsets in the following way:

1. A node exists in the tree corresponding to each frequent itemset. The root of the tree corresponds to the *null* itemset.

2. Let $I = \{i_1, \ldots i_k\}$ be a frequent itemset, where $i_1, i_2 \ldots i_k$ are listed in lexicographic order. The parent of the node I is the itemset $\{i_1, \ldots i_{k-1}\}$. Thus, the child of a node can only be extended with items occurring lexicographically *after* all items occurring in that node. The enumeration tree can also be viewed as a *prefix* tree on the lexicographically ordered string representation of the itemsets.

This definition of an ancestral relationship naturally creates a tree structure on the nodes, which is rooted at the *null* node. An example of the frequent portion of the enumeration tree is illustrated in Fig. 4.3. An item that is used to extend a node to its (frequent) child in the enumeration tree is referred to as a *frequent tree extension*, or simply a tree extension. In the example of Fig. 4.3, the frequent tree extensions of node a are b, c, d, and f, because

these items extend node a to the frequent itemsets ab, ac, ad, and af, respectively. The lattice provides many paths to extend the *null* itemset to a node, whereas an enumeration tree provides only one path. For example, itemset ab can be extended either in the order $a \rightarrow ab$, or in the order $b \rightarrow ab$ in the lattice. However, only the former is possible in the enumeration tree after the lexicographic ordering has been fixed. Thus, the lexicographic ordering imposes a strictly hierarchical structure on the itemsets. This hierarchical structure enables *systematic* and *non-redundant* exploration of the itemset search space by algorithms that generate candidates by extending frequent itemsets with one item at a time. The enumeration tree can be constructed in many ways with different lexicographic orderings of items. The impact of this ordering will be discussed later.

Most of the enumeration tree algorithms work by growing this enumeration tree of frequent itemsets with a predefined strategy. First, the root node of the tree is extended by finding the frequent 1-items. Then, these nodes may be extended to create *candidates*. These are checked against the transaction database to determine the ones that are frequent. The enumeration tree framework provides an order and structure to the frequent itemset discovery, which can be leveraged to improve the counting and pruning process of candidates. In the following discussion, the terms "node" and "itemset" will be used interchangeably. Therefore, the notation P will be used to denote both an itemset, and its corresponding node in the enumeration tree.

So, how can candidates nodes be generated in a systematic way from the frequent nodes in the enumeration tree that have already been discovered? For an item i to be considered a candidate for extending a frequent node P to $P \cup \{i\}$, it must also be a frequent extension of the parent Q of P. This is because of the downward closure property, and it can be used to systematically define the candidate extensions of a node P after the frequent extensions of its parent Q have been determined. Let $F(Q)$ represent the frequent lexicographic tree extensions of node Q. Let $i \in F(Q)$ be the frequent extension item that extends frequent node Q to frequent node $P = Q \cup \{i\}$. Let $C(P)$ denote the subset of items from $F(Q)$ occurring lexicographically *after* the item i used to extend node Q to node P. The set $C(P)$ defines the *candidate extension items* of node P, which are defined as items that can be appended at the end of P to create candidate itemsets. This provides a systematic methodology to generate candidate children of node P. As we will see in Sect. 4.4.3.1, the resulting candidates are identical to those generated by *Apriori* joins. Note that the relationship $F(P) \subseteq C(P) \subset F(Q)$ is always true. The value of $F(P)$ in Fig. 4.3, when $P = ab$, is $\{c, d\}$. The value of $C(P)$ for $P = ab$ is $\{c, d, f\}$ because these are frequent extensions of parent itemset $Q = \{a\}$ of P occurring lexicographically after the item b. Note that the set of *candidate* extensions $C(ab)$ also contains the (infrequent) item f that the set of *frequent* extensions $F(ab)$ does not. Such infrequent item extensions correspond to *failed* candidate tests in all enumeration tree algorithms. Note that the infrequent itemset abf is not included in the *frequent* itemset tree of Fig. 4.3. It is also possible to create an enumeration tree structure on the *candidate* itemsets, which contains an additional layer of infrequent candidate extensions of the nodes in Fig. 4.3. Such a tree would contain abf.

Enumeration tree algorithms iteratively grow the enumeration tree $\mathcal{ET}$ of frequent patterns. A very generic description of this iterative step, which is executed repeatedly to extend the enumeration tree $\mathcal{ET}$, is as follows:

Select one or more nodes $\mathcal{P}$ in $\mathcal{ET}$;
Determine candidate extensions $C(P)$ for each such node $P \in \mathcal{P}$;
Count support of generated candidates;
Add frequent candidates to $\mathcal{ET}$ (tree growth);

```
Algorithm GenericEnumerationTree(Transactions: T,
            Minimum Support: minsup)
begin
  Initialize enumeration tree ET to single Null node;
  while any node in ET has not been examined do begin
    Select one of more unexamined nodes P from ET for examination;
    Generate candidates extensions C(P) of each node P ∈ P;
    Determine frequent extensions F(P) ⊆ C(P) for each P ∈ P with support counting;
    Extend each node P ∈ P in ET with its frequent extensions in F(P);
  end
  return enumeration tree ET;
end
```

Figure 4.4: Generic enumeration-tree growth with unspecified growth strategy and counting method

This approach is continued until none of the nodes can be extended any further. At this point, the algorithm terminates. A more detailed description is provided in Fig. 4.4. Interestingly, almost all frequent pattern mining algorithms can be viewed as variations and extensions of this simple enumeration-tree framework. Within this broader framework, a wide variability exists both in terms of the growth strategy of the tree and the specific data structures used for support counting. Therefore, the description of Fig. 4.4 is very generic because none of these aspects are specified. The different choices of growth strategy and counting methodology provide different trade-offs between efficiency, space-requirements, and disk access costs. For example, in breadth-first strategies, the node set $\mathcal{P}$ selected in an iteration of Fig. 4.4 corresponds to all nodes at one level of the tree. This approach may be more relevant for disk-resident databases because all nodes at a single level of the tree can be extended during one counting pass on the transaction database. Depth-first strategies select a single node at the deepest level to create $\mathcal{P}$. These strategies may have better ability to explore the tree deeply and discover long frequent patterns early. The early discovery of longer patterns is especially useful for computational efficiency in maximal pattern mining and for better memory management in certain classes of *projection-based* algorithms.

Because the counting approach is the most expensive part, the different techniques attempt to use growth strategies that optimize the work done during counting. Furthermore, it is crucial for the counting data structures to be efficient. This section will explore some of the common algorithms, data structures, and pruning strategies that leverage the enumeration-tree structure in the counting process. Interestingly, the enumeration-tree framework is so general that even the *Apriori* algorithm can be interpreted within this framework, although the concept of an enumeration tree was not used when *Apriori* was proposed.

4.4.3.1 Enumeration-Tree-Based Interpretation of Apriori

The *Apriori* algorithm can be viewed as the level-wise construction of the enumeration tree in breadth-first manner. The *Apriori* join for generating candidate $(k+1)$-itemsets is performed in a non-redundant way by using only the *first* $(k-1)$ items from two frequent k-itemsets. This is equivalent to joining all pairs of *immediate siblings* at the kth level of the enumeration tree. For example, the children of ab in Fig. 4.3 may be obtained by joining

ab with all its frequent siblings (other children of node a) that occur lexicographically later than it. In other words, the join operation of node P with its lexicographically later frequent siblings produces the candidates corresponding to the extension of P with each of its candidate tree-extensions $C(P)$. In fact, the candidate extensions $C(P)$ for all nodes P at a given level of the tree can be *exhaustively* and *non-repetitively* generated by using joins between all pairs of frequent *siblings* at that level. The *Apriori* pruning trick then discards some of the enumeration tree nodes because they are guaranteed not to be frequent. A single pass over the transaction database is used to count the support of these candidate extensions, and generate the *frequent* extensions $F(P) \subseteq C(P)$ for each node P in the level being extended. The approach terminates when the tree cannot be grown further in a particular pass over the database. Thus, the join operation of *Apriori* has a direct interpretation in terms of the enumeration tree, and the *Apriori* algorithm implicitly extends the enumeration tree in a level-wise fashion with the use of joins.

4.4.3.2 TreeProjection and DepthProject

TreeProjection is a family of methods that uses recursive *projections* of the transactions down the enumeration tree structure. The goal of these recursive projections is to reuse the counting work that has already been done at a given node of the enumeration tree at its descendent nodes. This reduces the overall counting effort by orders of magnitude. *TreeProjection* is a general framework that shows how to use database projection in the context of a variety of different strategies for construction of the enumeration tree, such as breadth-first, depth-first, or a combination of the two. The *DepthProject* approach is a specific instantiation of this framework with the depth-first strategy. Different strategies have different trade-offs between the memory requirements and disk-access costs.

The main observation in projection-based methods is that if a transaction does not contain the itemset corresponding to an enumeration-tree node, then this transaction will not be relevant for counting at any descendent (superset itemset) of that node. Therefore, when counting is done at an enumeration-tree node, the information about irrelevant transactions should somehow be preserved for counting at its descendent nodes. This is achieved with the notion of *projected databases*. Each projected transaction database is specific to an enumeration-tree node. Transactions that do not contain the itemset P are not included in the projected databases at node P and its descendants. This results in a significant reduction in the number of projected transactions. Furthermore, only the candidate extension items of P, denoted by $C(P)$, are relevant for counting at any of the subtrees rooted at node P. Therefore, the projected database at node P can be expressed only in terms of the items in $C(P)$. The size of $C(P)$ is much smaller than the universe of items, and therefore the projected database contains a smaller number of items per transaction with increasing size of P. We denote the projected database at node P by $\mathcal{T}(P)$. For example, consider the node $P = ab$ in Fig. 4.3, in which the candidate items for extending ab are $C(P) = \{c, d, f\}$. Then, the transaction $abcfg$ maps to the projected transaction cf in $\mathcal{T}(P)$. On the other hand, the transaction $acfg$ is not even present in $\mathcal{T}(P)$ because $P = ab$ is not a subset of $acfg$. The special case $\mathcal{T}(Null) = \mathcal{T}$ corresponds to the top level of the enumeration tree and is equal to the full transaction database. In fact, the subproblem at node P with transaction database $\mathcal{T}(P)$ is structurally identical to the top-level problem, except that it is a much smaller problem focused on determining frequent patterns with a prefix of P. Therefore, the frequent node P in the enumeration tree can be extended further by counting the support of individual items in $C(P)$ using the relatively small database $\mathcal{T}(P)$. This

```
Algorithm ProjectedEnumerationTree(Transactions: 𝒯,
              Minimum Support: minsup)
begin
  Initialize enumeration tree ℰ𝒯 to a single (Null, 𝒯) root node;
  while any node in ℰ𝒯 has not been examined do begin
    Select an unexamined node (P, 𝒯(P)) from ℰ𝒯 for examination;
    Generate candidates item extensions C(P) of node (P, 𝒯(P));
    Determine frequent item extensions F(P) ⊆ C(P) by support counting
         of individual items in smaller projected database 𝒯(P);
    Remove infrequent items in 𝒯(P);
    for each frequent item extension i ∈ F(P) do begin
      Generate 𝒯(P ∪ {i}) from 𝒯(P);
      Add (P ∪ {i}, 𝒯(P ∪ {i})) as child of P in ℰ𝒯;
    end
  end
  return enumeration tree ℰ𝒯;
end
```

Figure 4.5: Generic enumeration-tree growth with unspecified growth strategy and database projections

results in a simplified and efficient counting process of candidate 1-item *extensions* rather than *itemsets*.

The enumeration tree can be grown with a variety of strategies such as the breadth-first or depth-first strategies. At each node, the counting is performed with the use of the projected database rather than the entire transaction database, and a further reduced and projected transaction database is propagated to the children of P. At each level of the hierarchical projection down the enumeration tree, the number of items and the number of transactions in the projected database are reduced. The basic idea is that $\mathcal{T}(P)$ contains the minimal portion of the transaction database that is *relevant* for counting the subtree rooted at P, based on the removal of irrelevant transactions and items by the counting process that has already been performed at higher levels of the tree. By *recursively* projecting the transaction database down the enumeration tree, this counting work is reused. We refer to this approach as *projection-based reuse* of counting effort.

The generic enumeration-tree algorithm with hierarchical projections is illustrated in Fig. 4.5. This generic algorithm does not assume any specific exploration strategy, and is quite similar to the generic enumeration-tree pseudocode shown in Fig. 4.4. There are two differences between the pseudocodes.

1. For simplicity of notation, we have shown the exploration of a single node P at one time in Fig. 4.5, rather than a group of nodes $\mathcal{P}$ (as in Fig. 4.4). However, the pseudocode shown in Fig. 4.5 can easily be rewritten for a group of nodes $\mathcal{P}$. Therefore, this is not a significant difference.

2. The key difference is that the projected database $\mathcal{T}(P)$ is used to count support at node P. Each node in the enumeration tree is now represented by the itemset and projected database pair $(P, \mathcal{T}(P))$. This is a very important difference because $\mathcal{T}(P)$ is much smaller than the original database. Therefore, a significant amount of information gained by counting the supports of ancestors of node P, is preserved in $\mathcal{T}(P)$. Furthermore, one only needs to count the support of single item *extensions* of node P in $\mathcal{T}(P)$ (rather than entire itemsets) in order to grow the subtree at P further.

The enumeration tree can be constructed in many different ways depending on the lexicographic ordering of items. How should the items be ordered? The structure of the enumeration tree has a built-in bias towards creating unbalanced trees in which the lexicographically smaller items have more descendants. For example, in Fig. 4.3, node a has many more descendants than node f. Therefore, ordering the items from least support to greatest support ensures that the computationally heavier branches of the enumeration tree have fewer relevant transactions. This is helpful in maximizing the selectivity of projections and ensuring better efficiency.

The strategy used for selection of the node P defines the order in which the nodes of the enumeration tree are materialized. This strategy has a direct impact on memory management because projected databases, which are no longer required for future computation, can be deleted. In depth-first strategies, the lexicographically smallest unexamined node P is selected for extension. In this case, one only needs to maintain projected databases along the current path of the enumeration tree being explored. In breadth-first strategies, an entire group of nodes $\mathcal{P}$ corresponding to all patterns of a particular size are grown first. In such cases, the projected databases need to be simultaneously maintained along the full breadth of the enumeration tree $\mathcal{ET}$ at the two current levels involved in the growth process. Although it may be possible to perform the projection on such a large number of nodes for smaller transaction databases, some modifications to the basic framework of Fig. 4.5 are needed for the general case of larger databases.

In particular, breadth-first variations of the *TreeProjection* framework perform hierarchical projections on the fly during counting from their ancestor nodes. The depth-first variations of *TreeProjection*, such as *DepthProject*, achieve full projection-based reuse because the projected transactions can be consistently maintained at each materialized node along the relatively small path of the enumeration tree from the root to the current node. The breadth-first variations do have the merit that they can optimize disk-access costs for arbitrarily large databases at the expense of losing some of the power of projection-based reuse. As will be discussed later, all (full) projection-based reuse methods face memory-management challenges with increasing database size. These additional memory requirements can be viewed as the price for persistently storing the relevant work done in earlier iterations in the indirect form of projected databases. There is usually a different trade-off between disk-access costs and memory/computational requirements in various strategies, which is exploited by the *TreeProjection* framework. The bibliographic notes contain pointers to specific details of these optimized variations of *TreeProjection*.

Optimized counting at deeper level nodes: The projection-based approach enables specialized counting techniques at deeper level nodes near the leaves of the enumeration tree. These specialized counting methods can provide the counts of *all* the itemsets in a lower-level subtree in the time required to *scan* the projected database. Because such nodes are more numerous, this can lead to large computational improvements.

What is the point at which such counting methods can be used? When the number of frequent extensions $F(P)$ of a node P falls below a threshold t such that 2^t fits in memory, an approach known as *bucketing* can be used. To obtain the best computational results, the value of t used should be such that 2^t is much smaller than the number of transactions in the projected database. This can occur only when there are many repeated transactions in the projected database.

A two-phase approach is used. In the first phase, the count of each distinct transaction in the projected database is determined. This can be accomplished easily by maintaining $2^{|F(P)|}$ buckets or counters, scanning the transactions one by one, and adding counts to the buckets. This phase can be completed in a simple scan of the small (projected) database

of transactions. Of course, this process only provides transaction counts and not itemset counts.

In the second phase, the transaction frequency counts can be further aggregated in a systematic way to create itemset frequency counts. Conceptually, the process of aggregating projected transaction counts is similar to arranging all the $2^{|F(P)|}$ possibilities in the form of a lattice, as illustrated in Fig. 4.1. The counts of the lattice nodes, which are computed in the first phase, are aggregated up the lattice structure by adding the count of immediate supersets to their subsets. For small values of $|F(P)|$, such as 10, this phase is not the limiting computational factor, and the overall time is dominated by that required to scan the projected database in the first phase. An efficient implementation of the second phase is discussed in detail below.

Consider a string composed of 0, 1, and $*$ that refers to an itemset in which the positions with 0 and 1 are fixed to those values (corresponding to presence or absence of items), whereas a position with a $*$ is a "don't care." Thus, all transactions can be expressed in terms of 0 and 1 in their binary representation. On the other hand, all itemsets can be expressed in terms of 1 and $*$ because itemsets are traditionally defined with respect to presence of items and ambiguity with respect to absence. Consider, for example, the case when $|F(P)| = 4$, and there are four items, numbered $\{1, 2, 3, 4\}$. An itemset containing items 2 and 4 is denoted by $*1*1$. We start with the information on $2^4 = 16$ bitstrings that are composed 0 and 1. These represent all possible distinct transactions. The algorithm aggregates the counts in $|F(P)|$ iterations. The count for a string with a "*" in a particular position may be obtained by adding the counts for the strings with a 0 and 1 in those positions. For example, the count for the string *1*1 may be expressed as the sum of the counts of the strings 01*1 and 11*1. The positions may be processed in any order, although the simplest approach is to aggregate them from the least significant to the most significant.

A simple pseudocode to perform the aggregation is described below. In this pseudocode, the initial value of $bucket[i]$ is equal to the count of the transaction corresponding to the bitstring representation of integer i. The final value of $bucket[i]$ is one in which the transaction count has been converted to an itemset count by successive aggregation. In other words, the 0s in the bitstring are replaced by "don't cares."

for $i := 1$ **to** k **do begin**
  for $j := 1$ **to** 2^k **do begin**
    if the ith bit of bitstring representation
      of j is 0 **then** $bucket[j] = bucket[j] + bucket[j + 2^{i-1}]$;
  endfor
endfor

An example of bucketing for $|F(P)| = 4$ is illustrated in Fig. 4.6. The bucketing trick is performed commonly at lower nodes of the tree because the value of $|F(P)|$ falls drastically at the lower levels. Because the nodes at the lower levels dominate the total number of nodes in the enumeration-tree structure, the impact of bucketing can be very significant.

Optimizations for maximal pattern mining: The *DepthProject* method, which is a depth-first variant of the approach, is particularly adaptable for maximal pattern discovery. In this case, the enumeration tree is explored in depth-first order to maximize the advantages of pruning the search space of regions containing only non-maximal patterns. The order of construction of the enumeration tree is important in the particular case of maximal frequent

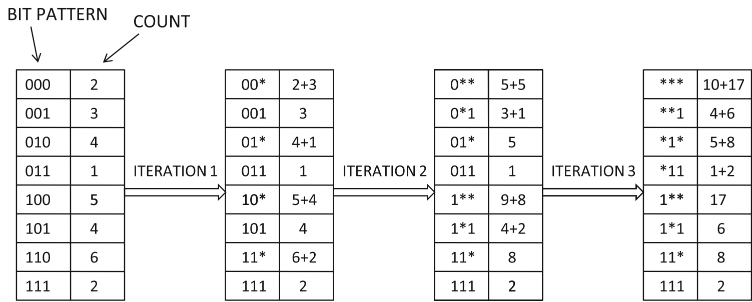

Figure 4.6: Performing the second phase of bucketing

pattern mining because certain kinds of non-maximal search-space pruning are optimized with the depth-first order. The notion of *lookaheads* is one such optimization.

Let $C(P)$ be the set of candidate item extensions of node P. Before support counting, it is tested whether $P \cup C(P)$ is a subset of a frequent pattern that has already been found. If such is indeed the case, then the pattern $P \cup C(P)$ is a non-maximal frequent pattern, and the entire subtree (of the enumeration tree) rooted at P can be pruned. This kind of pruning is referred to as *superset-based pruning*. When P cannot be pruned, the supports of its candidate extensions need to be determined. During this support counting, the support of $P \cup C(P)$ is counted along with the individual item extensions of P. If $P \cup C(P)$ is found to be frequent, then it eliminates any further work of counting the support of (non-maximal) nodes in the subtree rooted at node P.

While lookaheads can also be used with breadth-first algorithms, they are more effective with a depth-first strategy. In depth-first methods, longer patterns tend to be found first, and are, therefore, already available in the frequent set for superset-based pruning. For example, consider a frequent pattern of length 20 with 2^{20} subsets. In a depth-first strategy, it can be shown that the pattern of length 20 will be discovered after exploring only 19 of its immediate prefixes. On the other hand, a breadth-first method may remain trapped by discovery of shorter patterns. Therefore, the longer patterns become available very early in depth-first methods such as *DepthProject* to prune large portions of the enumeration tree with superset-based pruning.

4.4.3.3 Vertical Counting Methods

The *Partition* [446] and *Monet* [273] methods pioneered the concept of *vertical database representations* of the transaction database $\mathcal{T}$. In the *vertical representation*, each item is associated with a list of its transaction identifiers (*tids*). It can also be thought of as using the transpose of the binary transaction data matrix representing the transactions so that columns are transformed to rows. These rows are used as the new "records." Each item, thus, has a *tid* list of identifiers of transactions containing it. For example, the vertical representation of the database of Table 4.1 is illustrated in Table 4.2. Note that the binary matrix in Table 4.2 is the transpose of that in Table 4.1.

The intersection of two item *tid* lists yields a new *tid* list whose length is equal to the support of that 2-itemset. Further intersection of the resulting *tid* list with that of another item yields the support of 3-itemsets. For example, the intersection of the *tid* lists of *Milk* and *Yogurt* yields $\{2, 4, 5\}$ with length 3. Further intersection of the *tid* list of $\{Milk, Yogurt\}$ with that of *Eggs* yields the *tid* list $\{2, 4\}$ of length 2. This means that the support of