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.

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 $T_1 \dots T_n$ 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 $\mathcal{T} = \{T_1 \dots T_n\}$ 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 $\mathcal{T} = \{T_1 \dots T_n\}$, 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 $\mathcal{T} = \{T_1 \dots T_n\}$, 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

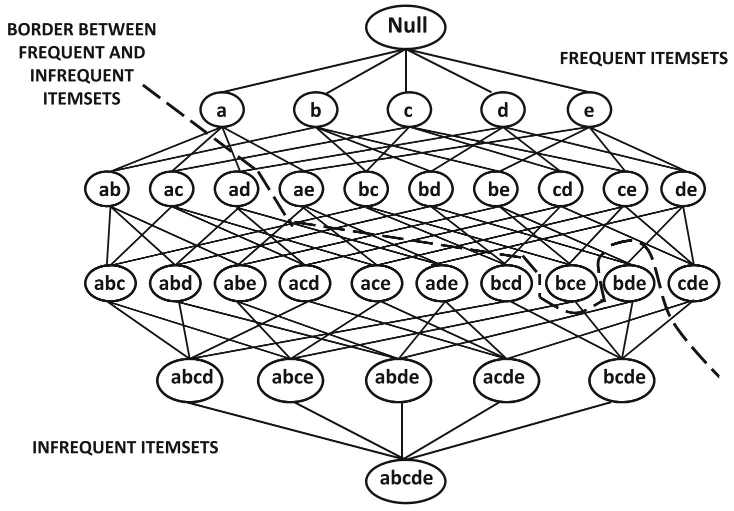


Figure 4.1: The itemset lattice

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 $\mathcal{C}_{k+1}$.

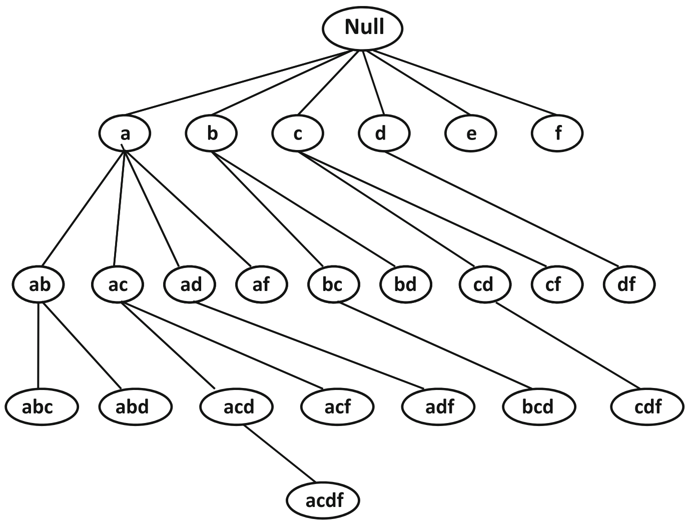


Figure 4.3: The lexicographic or enumeration tree of frequent itemsets

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: $\mathcal{T}$,
            Minimum Support: $minsup$)
begin
  Initialize enumeration tree $\mathcal{ET}$ to single $Null$ node;
  while any node in $\mathcal{ET}$ has not been examined do begin
    Select one of more unexamined nodes $\mathcal{P}$ from $\mathcal{ET}$ for examination;
    Generate candidates extensions $C(P)$ of each node $P \in \mathcal{P}$;
    Determine frequent extensions $F(P) \subseteq C(P)$ for each $P \in \mathcal{P}$ with support counting;
    Extend each node $P \in \mathcal{P}$ in $\mathcal{ET}$ with its frequent extensions in $F(P)$;
  end
  return enumeration tree $\mathcal{ET}$;
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: $\mathcal{T}$,
                Minimum Support: minsup)
begin
  Initialize enumeration tree $\mathcal{ET}$ to a single $(Null, \mathcal{T})$ root node;
  while any node in $\mathcal{ET}$ has not been examined do begin
    Select an unexamined node $(P, \mathcal{T}(P))$ from $\mathcal{ET}$ for examination;
    Generate candidates item extensions $C(P)$ of node $(P, \mathcal{T}(P))$;
    Determine frequent item extensions $F(P) \subseteq C(P)$ by support counting
          of individual items in smaller projected database $\mathcal{T}(P)$;
    Remove infrequent items in $\mathcal{T}(P)$;
    for each frequent item extension $i \in F(P)$ do begin
      Generate $\mathcal{T}(P \cup \{i\})$ from $\mathcal{T}(P)$;
      Add $(P \cup \{i\}, \mathcal{T}(P \cup \{i\}))$ as child of $P$ in $\mathcal{ET}$;
    end
  end
  return enumeration tree $\mathcal{ET}$;
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 $i$th 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

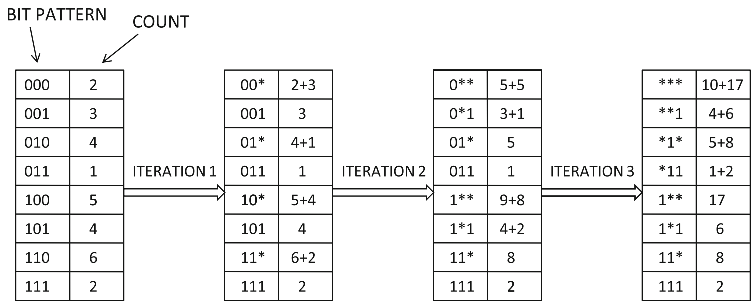


Figure 4.6: Performing the second phase of bucketing

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

Table 4.2: Vertical representation of market basket data set

Item	Set of tids	Binary representation
Bread	$\{1, 3\}$	10100
Butter	$\{1\}$	10000
Cheese	$\{3, 5\}$	00101
Eggs	$\{2, 3, 4\}$	01110
Milk	$\{1, 2, 3, 4, 5\}$	11111
Yogurt	$\{2, 4, 5\}$	01011

$\{Milk, Yogurt\}$ is $3/5 = 0.6$ and that of $\{Milk, Eggs, Yogurt\}$ is $2/5 = 0.4$. Note that one can also intersect the smaller *tid* lists of $\{Milk, Yogurt\}$ and $\{Milk, Eggs\}$ to achieve the same result. For a pair of k-itemsets that join to create a $(k + 1)$-itemset, it is possible to intersect the *tid* lists of the k-itemset pair to obtain the *tid*-list of the resulting $(k + 1)$-itemset. Intersecting *tid* lists of k-itemsets is preferable to intersecting *tid* lists of 1-itemsets because the *tid* lists of k-itemsets are typically smaller than those of 1-itemsets, which makes intersection faster. Such an approach is referred to as *recursive tid* list intersection. This insightful notion of recursive *tid* list intersection was introduced[2] by the *Monet* [273] and *Partition* [446] algorithms. The *Partition* framework [446] proposed a vertical version of the *Apriori* algorithm with *tid* list intersection. The pseudocode of this vertical version of the *Apriori* algorithm is illustrated in Fig. 4.7. The only difference from the horizontal *Apriori* algorithm is the use of recursive *tid* list intersections for counting. While the vertical *Apriori* algorithm is computationally more efficient than horizontal *Apriori*, it is memory-intensive because of the need to store *tid* lists with each itemset. Memory requirements can be reduced with the use of a *partitioned ensemble* in which the database is divided into smaller chunks which are independently processed. This approach reduces the memory requirements at the expense of running-time overheads in terms of postprocessing, and it is discussed in Sect. 4.6.2. For smaller databases, no partitioning needs to be applied. In such cases, the vertical *Apriori* algorithm of Fig. 4.7 is also referred to as *Partition-1*, and it is the progenitor of all modern vertical pattern mining algorithms.

The vertical database representation can, in fact, be used in almost any enumeration-tree algorithm with a growth strategy that is different from the breadth-first method. As in the case of the vertical *Apriori* algorithm, the *tid* lists can be stored with the itemsets (nodes) during the growth of the tree. If the *tid* list of any node P is known, it can be intersected with the *tid* list of a sibling node to determine the support count (and *tid* list) of the corresponding extension of P. This provides an efficient way of performing the counting. By varying the strategy of growing the tree, the memory overhead of storing the *tid* lists can be reduced but not the number of operations. For example, while both breadth-first and depth-first strategies will require exactly the same *tid* list intersections for a particular pair of nodes, the depth-first strategy will have a smaller memory footprint because the *tid* lists need to be stored only at the nodes on the tree-path being explored and their immediate siblings. Reducing the memory footprint is, nevertheless, important because it increases the size of the database that can be processed entirely in core.

Subsequently, many algorithms, such as *Eclat* and *VIPER*, adopted *Partition*'s recursive *tid* list intersection approach. *Eclat* is a lattice-partitioned memory-optimization of the algo-

[2]Strictly speaking, *Monet* is the name of the vertical database, on top of which this (unnamed) algorithm was built.

```
Algorithm VerticalApriori(Transactions: $\mathcal{T}$, Minimum Support: minsup)
begin
  $k = 1$;
  $\mathcal{F}_1 = \{$ All Frequent 1-itemsets $\}$;
  Construct vertical tid lists of each frequent item;
  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;
    Generate tid list of each candidate itemset in $\mathcal{C}_{k+1}$ by intersecting
       tid lists of the itemset-pair in $\mathcal{F}_k$ that was used to create it;
    Determine supports of itemsets in $\mathcal{C}_{k+1}$ using lengths of their tid lists;
    $\mathcal{F}_{k+1}$= Frequent itemsets of $\mathcal{C}_{k+1}$ together with their tid lists;
    $k = k + 1$;
  end;
  return($\cup_{i=1}^{k} \mathcal{F}_i$);
end
```

Figure 4.7: The vertical *Apriori* algorithm of Savasere et al. [446]

rithm in Fig. 4.7. In *Eclat* [537], an independent *Apriori*-like breadth-first strategy is used on each of the sublattices of itemsets with a common prefix. These groups of itemsets are referred to as equivalence classes. Such an approach can reduce the memory requirements by partitioning the candidate space into groups that are processed independently in conjunction with the relevant vertical lists of their prefixes. This kind of candidate partitioning is similar to parallel versions of *Apriori*, such as the *Candidate Distribution* algorithm [54]. Instead of using the candidate partitioning to distribute various sublattices to different processors, the *Eclat* approach sequentially processes the sublattices one after another to reduce peak memory requirements. Therefore, *Eclat* can avoid the postprocessing overheads associated with Savasere et al.'s *data* partitioning approach, if the database is too large to be processed in core by *Partition-1*, but small enough to be processed in core by *Eclat*. In such cases, *Eclat* is faster than *Partition*. Note that the number of computational operations for support counting in *Partition-1* is fundamentally no different from that of *Eclat* because the *tid* list intersections between any pair of itemsets remain the same. Furthermore, *Eclat* implicitly assumes an upper bound on the database size. This is because it assumes that multiple *tid* lists, each of size at least a fraction *minsup* of the number of database records, fit in main memory. The cumulative memory overhead of the multiple *tid* lists always scales proportionally with database size, whereas the memory overhead of the ensemble-based *Partition* algorithm is independent of database size.

4.4.4 Recursive Suffix-Based Pattern Growth Methods

Enumeration trees are constructed by extending *prefixes* of itemsets that are expressed in a lexicographic order. It is also possible to express some classes of itemset exploration methods recursively with *suffix*-based exploration. Although recursive pattern-growth is often understood as a completely different class of methods, it can be viewed as a special case of the generic enumeration-tree algorithm presented in the previous section. This relationship between recursive pattern-growth methods and enumeration-tree methods will be explored in greater detail in Sect. 4.4.4.5.

Recursive suffix-based pattern growth methods are generally understood in the context of the well-known FP-Tree data structure. While the FP-Tree provides a space- and time-efficient way to implement the recursive pattern exploration, these methods can also be implemented with the use of arrays and pointers. This section will present the recursive pattern growth approach in a simple way without introducing any specific data structure. We also present a number of simplified implementations[3] with various data structures to facilitate better understanding. The idea is to move from the simple to the complex by providing a top-down data structure-agnostic presentation, rather than a tightly integrated presentation with the commonly used FP-Tree data structure. This approach provides a clear understanding of how the search space of patterns is explored and the relational with conventional enumeration tree algorithms.

Consider the transaction database $\mathcal{T}$ which is expressed in terms of only frequent 1-items. It is assumed that a counting pass has already been performed on $\mathcal{T}$ to remove the infrequent items and count the supports of the items. Therefore, the input to the recursive procedure described here is slightly different from the other algorithms discussed in this chapter in which this database pass has not been performed. The items in the database are ordered with decreasing support. This lexicographic ordering is used to define the ordering of items within itemsets and transactions. This ordering is also used to define the notion of prefixes and suffixes of itemsets and transactions. The input to the algorithm is the transaction database $\mathcal{T}$ (expressed in terms of frequent 1-items), a current frequent itemset suffix P, and the minimum support *minsup*. The goal of a recursive call to the algorithm is to determine all the frequent patterns that have the suffix P. Therefore, at the top-level recursive call of the algorithm, the suffix P is empty. At deeper-level recursive calls, the suffix P is not empty. The assumption for deeper-level calls is that $\mathcal{T}$ contains only those transactions from the original database that include the itemset P. Furthermore, each transaction in $\mathcal{T}$ is represented using only those frequent extension items of P that are lexicographically smaller than all items of P. Therefore $\mathcal{T}$ is a *conditional transaction set*, or *projected database* with respect to suffix P. This suffix-based projection is similar to the prefix-based projection in *TreeProjection* and *DepthProject*.

In any given recursive call, the first step is to construct the itemset $P_i = \{i\} \cup P$ by concatenating each item i in the transaction database $\mathcal{T}$ to the beginning of suffix P, and reporting it as frequent. The itemset P_i is frequent because $\mathcal{T}$ is defined in terms of frequent items of the projected database of suffix P. For each item i, it is desired to further extend P_i by using a recursive call with the projected database of the (newly extended) frequent suffix P_i. The projected database for extended suffix P_i is denoted by $\mathcal{T}_i$, and it is created as follows. The first step is to extract all transactions from $\mathcal{T}$ that contain the item i. Because it is desired to extend the suffix P_i backwards, all items that are lexicographically greater than or equal to i are removed from the extracted transactions in $\mathcal{T}_i$. In other words, the part of the transaction occurring lexicographically after (and including) i is not relevant for counting frequent patterns ending in P_i. The frequency of each item in $\mathcal{T}_i$ is counted, and the infrequent items are removed.

It is easy to see that the transaction set $\mathcal{T}_i$ is sufficient to generate all the frequent patterns with P_i as a suffix. The problem of finding all frequent patterns ending in P_i using the transaction set $\mathcal{T}_i$ is an identical but smaller problem than the original one on $\mathcal{T}$. Therefore, the original procedure is called recursively with the smaller projected database $\mathcal{T}_i$ and extended suffix P_i. This procedure is repeated for each item i in $\mathcal{T}$.

[3]Variations of these strategies are actually used in some implementations of these methods. We stress that the simplified versions are not optimized for efficiency but are provided for clarity.

```
Algorithm RecursiveSuffixGrowth(Transactions in terms of frequent 1-items: $\mathcal{T}$,
                Minimum Support: minsup, Current Suffix: $P$)
begin
  for each item $i$ in $\mathcal{T}$ do begin
    report itemset $P_i = \{i\} \cup P$ as frequent;
    Extract all transactions $\mathcal{T}_i$ from $\mathcal{T}$ containing item $i$;
    Remove all items from $\mathcal{T}_i$ that are lexicographically $\geq i$;
    Remove all infrequent items from $\mathcal{T}_i$;
    if ($\mathcal{T}_i \neq \phi$) then RecursiveSuffixGrowth($\mathcal{T}_i, minsup, P_i$);
  end
end
```

Figure 4.8: Generic recursive suffix growth on transaction database expressed in terms of frequent 1-items

The projected transaction set $\mathcal{T}_i$ will become successively smaller at deeper levels of the recursion in terms of the number of items and the number of transactions. As the number of transactions reduces, all items in it will eventually fall below the minimum support, and the resulting projected database (constructed on only the frequent items) will be empty. In such cases, a recursive call with $\mathcal{T}_i$ is not initiated; therefore, this branch of the recursion is not explored. For some data structures, such as the FP-Tree, it is possible to impose stronger boundary conditions to terminate the recursion even earlier. This boundary condition will be discussed in a later section.

The overall recursive approach is presented in Fig. 4.8. While the parameter *minsup* has always been assumed to be a (relative) fractional value in this chapter, it is assumed to be an absolute integer support value in this section and in Fig. 4.8. This deviation from the usual convention ensures consistency of the minimum support value across different recursive calls in which the size of the conditional transaction database reduces.

4.4.4.1 Implementation with Arrays but No Pointers

So, how can the projected database $\mathcal{T}$ be decomposed into the *conditional transaction sets* $\mathcal{T}_1 \ldots \mathcal{T}_d$, corresponding to d different 1-item suffixes? The simplest solution is to use arrays. In this solution, the original transaction database $\mathcal{T}$ and the conditional transaction sets $\mathcal{T}_1 \ldots \mathcal{T}_d$ can be represented in arrays. The transaction database $\mathcal{T}$ may be scanned within the "**for**" loop of Fig. 4.8, and the set $\mathcal{T}_i$ is created from $\mathcal{T}$. The infrequent items from $\mathcal{T}_i$ are removed within the loop. However, it is expensive and wasteful to repeatedly scan the database $\mathcal{T}$ inside a "**for**" loop. One alternative is to extract all projections $\mathcal{T}_i$ of $\mathcal{T}$ corresponding to the different suffix items simultaneously in a single scan of the database just before the "**for**" loop is initiated. On the other hand, the simultaneous creation of many such item-specific projected data sets can be memory-intensive. One way of obtaining an excellent trade-off between computational and storage requirements is by using pointers. This approach is discussed in the next section.

4.4.4.2 Implementation with Pointers but No FP-Tree

The array-based solution either needs to repeatedly scan the database $\mathcal{T}$ or simultaneously create many smaller item-specific databases in a single pass. Typically, the latter achieves

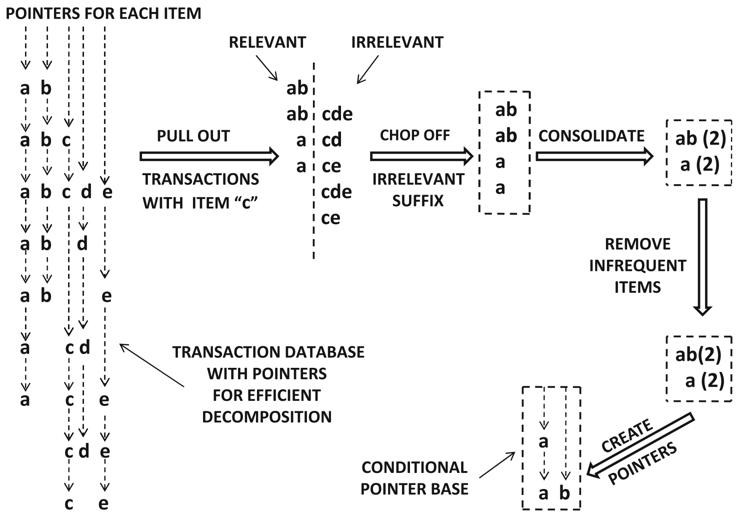


Figure 4.9: Illustration of recursive pattern growth with pointers and no FP-Tree

better efficiency but is more memory-intensive. One simple solution to this dilemma is to set up a data structure in the form of pointers in the first pass, which *implicitly* stores the decomposition of $\mathcal{T}$ into different item-specific data sets at a lower memory cost. This data structure is set up at the time that infrequent items are removed from the transaction database $\mathcal{T}$, and then utilized for extracting different conditional transaction sets $\mathcal{T}_i$ from $\mathcal{T}$. For each item i in $\mathcal{T}$, a pointer threads through the transactions containing that item in lexicographically sorted (dictionary) order. In other words, after arranging the database $\mathcal{T}$ in lexicographically sorted order, each item i in each transaction has a pointer to the same item i in the next transaction that contains it. Because a pointer is required at each item in each transaction, the storage overhead in this case is proportional to that of the original transaction database $\mathcal{T}$. An additional optimization is to consolidate repeated transactions and store them with their counts. An example of a sample database with nine transactions on the five items $\{a, b, c, d, e\}$ is illustrated in Fig. 4.9. It is clear from the figure that there are five sets of pointers, one for each item in the database.

After the pointers have been set up, $\mathcal{T}_i$ is extracted by just "chasing" the pointer thread for item i. The time for doing this is proportional to the number of transactions in $\mathcal{T}_i$. The infrequent items in $\mathcal{T}_i$ are removed, and the pointers for the conditional transaction data need to be reconstructed to create a *conditional pointer base* which is basically the conditional transaction set augmented with pointers. The modified pseudocode with the use of pointers is illustrated in Fig. 4.10. Note that the only difference between the pseudocode of Figs. 4.8 and 4.10 is the setting up of pointers after extraction of conditional transaction sets and the use of these pointers to efficiently extract the conditional transaction data sets $\mathcal{T}_i$. A recursive call is initiated at the next level with the extended suffix $P_i = \{i\} \cup P$, and conditional database $\mathcal{T}_i$.

To illustrate how $\mathcal{T}_i$ can be extracted, an example of a transaction database with 5 items and 9 transactions is illustrated in Fig. 4.9. For simplicity, we use a (raw) minimum support value of 1. The transactions corresponding to the item c are extracted, and the irrelevant suffix including and after item c are removed for further recursive calls. Note that this leads to shorter transactions, some of which are repeated. As a result, the conditional database

```
Algorithm RecursiveGrowthPointers(Transactions in terms of frequent 1-items: $\mathcal{T}$,
          Minimum Support: $minsup$, Current Suffix: $P$)
begin
  for each item $i$ in $\mathcal{T}$ do begin
    report itemset $P_i = \{i\} \cup P$ as frequent;
    Use pointers to extract all transactions $\mathcal{T}_i$
        from $\mathcal{T}$ containing item $i$;
    Remove all items from $\mathcal{T}_i$ that are lexicographically $\geq i$;
    Remove all infrequent items from $\mathcal{T}_i$;
    Set up pointers for $\mathcal{T}_i$;
    if ($\mathcal{T}_i \neq \phi$) then RecursiveGrowthPointers($\mathcal{T}_i, minsup, P_i$);
  end
end
```

Figure 4.10: Generic recursive suffix growth with pointers

for $\mathcal{T}_i$ contains only two distinct transactions after consolidation. The infrequent items from this conditional database need to be removed. No items are removed at a minimum support of 1. Note that if the minimum support had been 3, then the item b would have been removed. The pointers for the new conditional transaction set do need to be set up again because they will be different for the conditional transaction database than in the original transactions. Unlike the pseudocode of Fig. 4.8, an additional step of setting up pointers is included in the pseudocode of Fig. 4.10.

The pointers provide an efficient way to extract the conditional transaction database. Of course, the price for this is that the pointers are a space overhead, with size exactly proportional to the original transaction database $\mathcal{T}$. Consolidating repeated transactions does save some space. The FP-Tree, which will be discussed in the next section, takes this approach one step further by consolidating not only repeated transactions, but also repeated *prefixes* of transactions with the use of a trie data structure. This representation reduces the space-overhead by consolidating prefixes of the transaction database.

4.4.4.3 Implementation with Pointers and FP-Tree

The FP-Tree is designed with the primary goal of space efficiency of the projected database. The FP-Tree is a trie data structure representation of the conditional transaction database by consolidating the prefixes. This trie replaces the array-based implementation of the previous sections, but it retains the pointers. The path from the root to the leaf in the trie represents a (possibly repeated) transaction in the database. The path from the root to an internal node may represent either a transaction or the prefix of a transaction in the database. Each internal node is associated with a count representing the number of transactions in the original database that contain the prefix corresponding to the path from the root to that node. The count on a leaf represents the number of repeated instances of the transaction defined by the path from the root to that leaf. Thus, the FP-Tree maintains all counts of all the repeated transactions as well as their prefixes in the database. As in a standard trie data-structure, the prefixes are sorted in dictionary order. The lexicographic ordering of items is from the most frequent to the least frequent to maximize the advantages of prefix-based compression. This ordering also provides excellent selectivity in reducing the size of various conditional transaction sets in a balanced way. An example of the FP-Tree

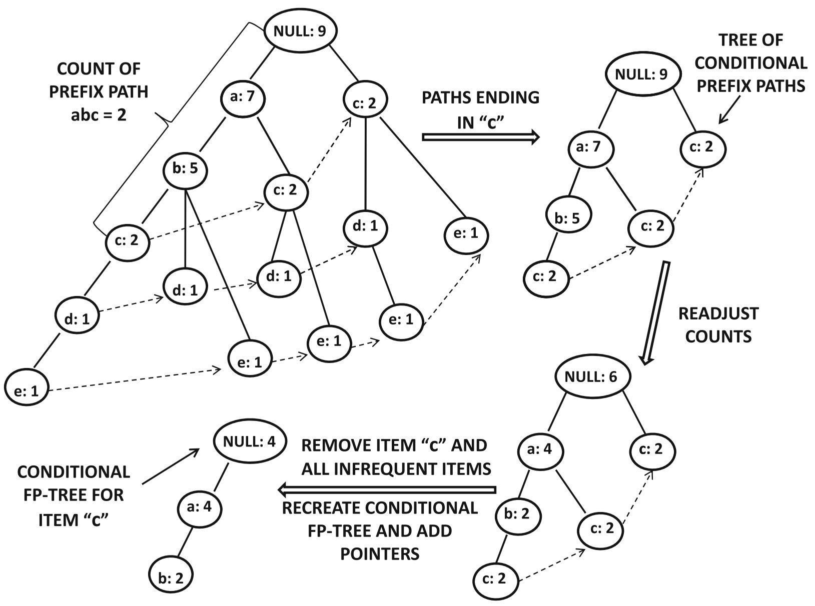

COUNT OF PREFIX PATH abc = 2
NULL: 9
a: 7
c: 2
b: 5
c: 2
c: 2
d: 1
e: 1
d: 1
d: 1
d: 1
e: 1
e: 1
e: 1
e: 1
PATHS ENDING IN "c"
TREE OF CONDITIONAL PREFIX PATHS
NULL: 9
a: 7
c: 2
b: 5
c: 2
c: 2
READJUST COUNTS
NULL: 6
a: 4
c: 2
b: 2
c: 2
c: 2
REMOVE ITEM "c" AND ALL INFREQUENT ITEMS
RECREATE CONDITIONAL FP-TREE AND ADD POINTERS
CONDITIONAL FP-TREE FOR ITEM "c"
NULL: 4
a: 4
b: 2

Figure 4.11: Illustration of recursive pattern growth with pointers and FP-Tree

data structure for the same database (as the previous example of Fig. 4.9) is shown in Fig. 4.11. In the example, the number "2" associated with the leftmost item c in the FP-Tree, represents the count of prefix path *abc*, as illustrated in Fig. 4.11.

The initial FP-Tree $\mathcal{FPT}$ can be constructed as follows. First the infrequent items in the database are removed. The resulting transactions are then successively inserted into the trie. The counts on the overlapping nodes are incremented by 1 when the prefix of the inserted transaction overlaps with an existing path in the trie. For the non-overlapping portion of the transaction, a new path needs to be created containing this portion. The newly created nodes are assigned a count of 1. This process of insertion is identical to that of trie creation, except that counts are also associated with nodes. The resulting tree is a compressed representation because common items in the prefixes of multiple transactions are represented by a single node.

The pointers can be constructed in an analogous way to the simpler array data structure of the previous section. The pointer for each item points to the next occurrence of the same item in the trie. Because a trie stores the transactions in dictionary order, it is easy to create pointers threading each of the items. However, the number of pointers is smaller, because many nodes have been consolidated. As an illustrative example, one can examine the relationship between the array-based data structure of Fig. 4.9, and the FP-Tree in Fig. 4.11. The difference is that the prefixes of the arrays in Fig. 4.9 are consolidated and compressed into a trie in Fig. 4.11.

The conditional FP-Tree $\mathcal{FPT}_i$ (representing the conditional database $\mathcal{T}_i$) needs to be extracted and reorganized for each item $i \in \mathcal{FPT}$. This extraction is required to initiate recursive calls with conditional FP-Trees. As in the case of the simple pointer-based structure of the previous section, it is possible to use the pointers of an item to extract the subset of the projected database containing that item. The following steps need to be performed for extraction of the conditional FP-Tree of item i:

1. The pointers for item i are chased to extract the *tree of conditional prefix paths* for the item. These are the paths from the item to the root. The remaining branches are pruned.

2. The counts of the nodes in the tree of prefix-paths are adjusted to account for the pruned branches. The counts can be adjusted by aggregating the counts on the leaves upwards.

3. The frequency of each item is counted by aggregating the counts over all occurrences of that item in the tree of prefix paths. The items that do not meet the minimum support requirement are removed from the prefix paths. Furthermore, the last item i is also removed from each prefix path. The resulting conditional FP-Tree might have a completely different organization than the extracted tree of prefix-paths because of the removal of infrequent items. Therefore, the conditional FP-Tree may need to be recreated by reinserting the conditional prefix paths obtained after removing infrequent items. The pointers for the conditional FP-Tree need to be reconstructed as well.

Consider the example in Fig. 4.11 which is the same data set as in Fig. 4.9. As in Fig. 4.9, it is possible to follow the pointers for item c in Fig. 4.11 to extract a tree of conditional prefix paths (shown in Fig. 4.11). The counts on many nodes in the tree of conditional prefix paths need to be reduced because many branches from the original FP-Tree (that do not contain the item c) are not included. These reduced counts can be determined by aggregating the counts on the leaves upwards. After removing the item c and infrequent items, two frequency-annotated conditional prefix paths $ab(2)$ and $a(2)$ are obtained, which are identical to the two projected and consolidated transactions of Fig. 4.9. The conditional FP-tree is then constructed for item c by reinserting these two conditional prefix paths into a new conditional FP-Tree. Again, this conditional FP-Tree is a trie representation of the conditional pointer base of Fig. 4.9. In this case, there are no infrequent items because a minimum support of 1 is used. If a minimum support of 3 had been used, then the item b would have to be removed. The resulting conditional FP-Tree is used in the next level recursive call. After extracting the conditional FP-Tree $\mathcal{FPT}_i$, it is checked whether it is empty. An empty conditional FP-Tree could occur when there are no frequent items in the extracted tree of conditional prefix paths. If the tree is not empty, then the next level recursive call is initiated with suffix $P_i = \{i\} \cup P$, and the conditional FP-Tree $\mathcal{FPT}_i$.

The use of the FP-Tree allows an additional optimization in the form of a boundary condition for quickly extracting frequent patterns at deeper levels of the recursion. In particular, it is checked whether all the nodes of the FP-Tree lie on a single path. In such a case, the frequent patterns can be directly extracted from this path by extracting all combinations of nodes on this path together with the aggregated support counts. For example, in the case of Fig. 4.11, all nodes on the conditional FP-Tree lie on a single path. Therefore, in the next recursive call, the bottom of the recursion will be reached. The pseudocode for *FP-growth* is illustrated in Fig. 4.12. This pseudocode is similar to the pointer-based pseudocode of Fig. 4.10, except that a compressed FP-Tree is used.

4.4.4.4 Trade-offs with Different Data Structures

The main advantage of an FP-Tree over pointer-based implementation is one of space compression. The FP-Tree requires less space than pointer-based implementation because of trie-based compression, although it might require more space than an array-based implementation because of the pointer overhead. The precise space requirements depend on the

```
Algorithm FP-growth(FP-Tree of frequent items: $\mathcal{FPT}$, Minimum Support: $minsup$,
                    Current Suffix: $P$)
begin
  if $\mathcal{FPT}$ is a single path
      then determine all combinations $C$ of nodes on the
          path, and report $C \cup P$ as frequent;
  else (Case when $\mathcal{FPT}$ is not a single path)
  for each item $i$ in $\mathcal{FPT}$ do begin
    report itemset $P_i = \{i\} \cup P$ as frequent;
    Use pointers to extract conditional prefix paths
        from $\mathcal{FPT}$ containing item $i$;
    Readjust counts of prefix paths and remove $i$;
    Remove infrequent items from prefix paths and reconstruct
          conditional FP-Tree $\mathcal{FPT}_i$;
    if ($\mathcal{FPT}_i \neq \phi$) then FP-growth($\mathcal{FPT}_i, minsup, P_i$);
  end
end
```

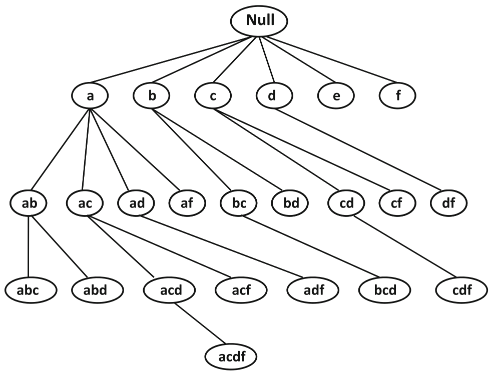
Figure 4.12: The *FP-growth* algorithm with an FP-Tree representation of the transaction database expressed in terms of frequent 1-items

level of consolidation at higher level nodes in the trie-like FP-Tree structure for a particular data set. Different data structures may be more suitable for different data sets.

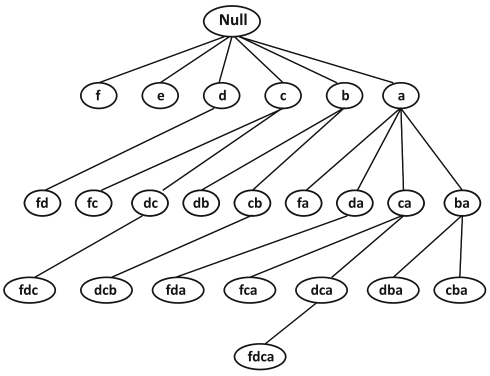
Because projected databases are repeatedly constructed and scanned during recursive calls, it is crucial to maintain them in main memory. Otherwise, drastic disk-access costs will be incurred by the potentially exponential number of recursive calls. The sizes of the projected databases increase with the original database size. For certain kinds of databases with limited consolidation of repeated transactions, the number of *distinct* transactions in the projected database will always be approximately proportional to the number of transactions in the original database, where the proportionality factor f is equal to the (fractional) minimum support. For databases that are larger than a factor $1/f$ of the main memory availability, projected databases may not fit in main memory either. Therefore, the limiting factor on the use of the approach is the size of the original transaction database. This issue is specific to almost all projection-based methods and vertical counting methods. Memory is always at a premium in such methods and therefore it is crucial for projected transaction data structures to be designed as compactly as possible. As we will discuss later, the *Partition* framework of Savasere et al. [446] provides a partial solution to this issue at the expense of running time.

4.4.4.5 Relationship Between FP-Growth and Enumeration-Tree Methods

FP-growth is popularly believed to be radically different from enumeration-tree methods. This is, in part, because *FP-growth* was originally presented as a method that extracts frequent patterns without candidate generation. However, such an exposition provides an incomplete understanding of how the search space of patterns is explored. *FP-growth* is an instantiation of enumeration-tree methods. All enumeration-tree methods generate candidate extensions to grow the tree. In the following, we will show the equivalence between enumeration-tree methods and *FP-growth*.



(a) Prefix extensions with ordering of a, b, c, d, e, f (Enumeration Tree Prefixes shown)



(b) *FP-growth* with ordering of f, e, d, c, b, a (Recursion Tree Suffixes shown)

Figure 4.13: Enumeration trees are identical to *FP-growth* recursion trees with reverse lexicographic ordering

FP-growth is a recursive algorithm that extends suffixes of frequent patterns. Any recursive approach has a tree-structure associated with it that is referred to as its *recursion tree*, and a dynamic *recursion stack* that stores the recursion variables on the current path of the recursion tree during execution. Therefore, it is instructive to examine the suffix-based recursion tree created by the *FP-growth* algorithm, and compare it with the classical prefix-based enumeration tree used by enumeration-tree algorithms.

In Fig. 4.13a, the enumeration tree from the earlier example of Fig. 4.3 has been replicated. This tree of frequent patterns is counted by all enumeration-tree algorithms along with a single layer of infrequent candidate extensions of this tree corresponding to failed candidate tests. Each call of *FP-growth* discovers the set of frequent patterns extending a particular suffix of items, just as each branch of an enumeration tree explores the itemsets for a particular prefix. So, what is the hierarchical recursive relationship among the suffixes whose conditional pattern bases are explored? First, we need to decide on an ordering of items. Because the recursion is performed on suffixes and enumeration trees are constructed on prefixes, the *opposite* ordering $\{f, e, d, c, b, a\}$ is assumed to adjust for the different convention in the two methods. Indeed, most enumeration-tree methods order items from the least frequent to the most frequent, whereas *FP-growth* does the reverse. The corresponding recursion tree of *FP-growth*, when the 1-itemsets are ordered from left to right in dictionary order, is illustrated in Fig. 4.13b. The trees in Figs 4.13a and 4.13b are identical, with the only difference being that they are drawn differently, to account for the opposite lexicographic ordering. The *FP-growth* recursion tree on the reverse lexicographic ordering has an identical structure to the traditional enumeration tree on the prefixes. During any given recursive call of *FP-growth*, the current (recursion) stack of suffix items is the path in the enumeration tree that is currently being explored. This enumeration tree is explored in depth-first order by *FP-growth* because of its recursive nature.

Traditional enumeration-tree methods typically count the support of a single layer of infrequent extensions of the frequent patterns in the enumeration-tree, as (failed) candidates, to rule them out. Therefore, it is instructive to explore whether *FP-growth* avoids counting these infrequent candidates. Note that when conditional transaction databases $\mathcal{FPT}_i$ are created (see Fig. 4.12), infrequent items must be removed from them. This requires the counting of the support of these (implicitly failed) *candidate extensions*. In a traditional candidate generate-and-test algorithm, the frequent candidate extensions would be reported immediately after the counting step as a successful candidate test. However, in *FP-growth*, these frequent extensions are encoded back into the conditional transaction database $\mathcal{FPT}_i$, and the reporting is delayed to the next level recursive call. In the next level recursive call, these frequent extensions are then extracted from $\mathcal{FPT}_i$ and reported. The counting and removal of infrequent items from conditional transaction sets is an implicit candidate evaluation and testing step. The number of such failed candidate tests[4] in *FP-growth* is exactly equal to that of enumeration-tree algorithms, such as *Apriori* (without the level-wise pruning step). This equality follows directly from the relationship of all these algorithms to how they explore the enumeration tree and rule out infrequent portions. All pattern-growth methods, including *FP-growth*, should be considered enumeration-tree methods, as should *Apriori*. Whereas traditional enumeration trees are constructed on prefixes, the (implicit) *FP-growth* enumeration trees are constructed using suffixes. This is a difference only in the item-ordering convention.

The depth-first strategy is the approach of choice in database projection methods because it is more memory-efficient to maintain the conditional transaction sets along a (relatively small) depth of the enumeration (recursion) tree rather than along the (much larger) breadth of the enumeration tree. As discussed in the previous section, memory management becomes a problem even with the depth-first strategy beyond a certain database size. However, the specific strategy used for tree exploration does not have any impact on the size of the enumeration tree (or candidates) explored over the course of the entire algorithm execution. The only difference is that breadth-first methods process candidates in large batches based on pattern size, whereas depth-first methods process candidates in smaller batches of immediate siblings in the enumeration tree. From this perspective, *FP-growth* cannot avoid the exponential candidate search space exploration required by enumeration-tree methods, such as *Apriori*.

Whereas methods such as *Apriori* can also be interpreted as counting methods on an enumeration-tree of exactly the same size as the recursion tree of *FP-growth*, the counting work done at the higher levels of the enumeration tree is lost. This loss is because the counting is done from scratch at each level in *Apriori* with the entire transaction database rather than a projected database that remembers and *reuses* the work done at the higher levels of the tree. Projection-based reuse is also utilized by Savasere et al.'s vertical counting methods [446] and *DepthProject*. The use of a pointer-trie combination data structure for projected transaction representation is the primary difference of *FP-growth* from other projection-based methods. In the context of depth-first exploration, these methods can be understood either as divide-and-conquer strategies or as projection-based reuse strategies. The notion of projection-based reuse is more general because it applies to both the breadth-first and depth-first versions of the algorithm, and it provides a clearer picture of how computational savings are achieved by avoiding wasteful and repetitive counting. Projection-based reuse enables the efficient testing of candidate *item extensions* in a restricted portion of the database rather than the testing of candidate *itemsets* in the full database. Therefore, the efficiencies in *FP-growth* are a result of more efficient counting *per candidate* and not because of fewer candidates. The only differences in search space size between various methods are

[4] An *ad hoc* pruning optimization in *FP-growth* terminates the recursion when all nodes in the FP-Tree lie on a single path. This pruning optimization reduces the number of successful candidate tests but not the number of failed candidate tests. Failed candidate tests often dominate successful candidate tests in real data sets.

the result of *ad hoc* pruning optimizations, such as level-wise pruning in *Apriori*, bucketing in the *DepthProject* algorithm, and the single-path boundary condition of *FP-growth*.

The bookkeeping of the projected transaction sets can be done differently with the use of different data structures, such as arrays, pointers, or a pointer-trie combination. Many different data structure variations are explored in different projection algorithms, such as *TreeProjection*, *DepthProject*, *FP-growth*, and *H-Mine* [419]. Each data structure is associated with a different set of efficiencies and overheads.

In conclusion, the enumeration tree[5] is the most general framework to describe all previous frequent pattern mining algorithms. This is because the enumeration tree is a subgraph of the lattice (candidate space) and it provides a way to explore the candidate patterns in a systematic and non-redundant way. The support testing of the frequent portion of the enumeration tree along with a single layer of infrequent candidate extensions of these nodes is fundamental to all frequent itemset mining algorithms for ruling in and ruling out *possible* (or *candidate*) frequent patterns. Any algorithm, such as *FP-growth*, which uses the enumeration tree to rule in and rule out possible extensions of frequent patterns with support counting, is a candidate generate-and-test algorithm.

4.5 Alternative Models: Interesting Patterns

The traditional model for frequent itemset generation has found widespread popularity and acceptance because of its simplicity. The simplicity of using raw frequency counts for the support, and that of using the conditional probabilities for the confidence is very appealing. Furthermore, the downward closure property of frequent itemsets enables the design of efficient algorithms for frequent itemset mining. This algorithmic convenience does not, however, mean that the patterns found are always significant from an *application-specific* perspective. Raw frequencies of itemsets do not always correspond to the most *interesting* patterns.

For example, consider the transaction database illustrated in Fig. 4.1. In this database, *all* the transactions contain the item $Milk$. Therefore, the item $Milk$ can be appended to *any* set of items, without changing its frequency. However, this does not mean that $Milk$ is truly associated with any set of items. Furthermore, for any set of items X, the association rule $X \Rightarrow \{Milk\}$ has 100% confidence. However, it would not make sense for the supermarket merchant to assume that the basket of items X is *discriminatively* indicative of $Milk$. Herein lies the limitation of the traditional support-confidence model.

Sometimes, it is also desirable to design measures that can adjust to the skew in the individual item support values. This adjustment is especially important for negative pattern mining. For example, the support of the pair of items $\{Milk, Butter\}$ is very different from that of $\{\neg Milk, \neg Butter\}$. Here, $\neg$ indicates negation. On the other hand, it can be argued that the statistical coefficient of correlation is exactly the same in both cases. Therefore, the measure should quantify the association between both pairs in exactly the same way. Clearly, such measures are important for negative pattern mining. Measures that satisfy this property are said to satisfy the *bit symmetric* property because values of 0 in the binary matrix are treated in a similar way to values of 1.

[5] *FP-growth* has been presented in a separate section from enumeration tree methods only because it uses a different convention of constructing *suffix*-based enumeration trees. It is not necessary to distinguish "pattern growth" methods from "candidate-based" methods to meaningfully categorize various frequent pattern mining methods. Enumeration tree methods are best categorized on the basis of their (i) tree exploration strategy, (ii) projection-based reuse properties, and (iii) relevant data structures.

Although it is possible to quantify the affinity of sets of items in ways that are statistically more robust than the support-confidence framework, the major computational problem faced by most such *interestingness-based models* is that the downward closure property is generally not satisfied. This makes algorithmic development rather difficult on the exponentially large search space of patterns. In some cases, the measure is defined only for the special case of 2-itemsets. In other cases, it is possible to design more efficient algorithms. The following contains a discussion of some of these models.

4.5.1 Statistical Coefficient of Correlation

A natural statistical measure is the Pearson coefficient of correlation between a pair of items. The Pearson coefficient of correlation between a pair of random variables X and Y is defined as follows:

$$\rho = \frac{E[X \cdot Y] - E[X] \cdot E[Y]}{\sigma(X) \cdot \sigma(Y)}. \tag{4.4}$$

In the case of market basket data, X and Y are binary variables whose values reflect presence or absence of items. The notation $E[X]$ denotes the expectation of X, and $\sigma(X)$ denotes the standard deviation of X. Then, if $sup(i)$ and $sup(j)$ are the relative supports of individual items, and $sup(\{i, j\}$ is the relative support of itemset $\{i, j\}$, then the overall correlation can be estimated from the data as follows:

$$\rho_{ij} = \frac{sup(\{i,j\}) - sup(i) \cdot sup(j)}{\sqrt{sup(i) \cdot sup(j) \cdot (1 - sup(i)) \cdot (1 - sup(j))}}. \tag{4.5}$$

The coefficient of correlation always lies in the range $[-1, 1]$, where the value of $+1$ indicates perfect positive correlation, and the value of -1 indicates perfect negative correlation. A value near 0 indicates weakly correlated data. This measure satisfies the bit symmetric property. While the coefficient of correlation is statistically considered the most robust way of measuring correlations, it is often intuitively hard to interpret when dealing with items of varying but low support values.

4.5.2 χ^2 Measure

The χ^2 measure is another bit-symmetric measure that treats the presence and absence of items in a similar way. Note that for a set of k binary random variables (items), denoted by X, there are 2^k-possible states representing presence or absence of different items of X in the transaction. For example, for $k = 2$ items $\{Bread, Butter\}$, the 2^2 states are $\{Bread, Butter\}$, $\{Bread, \neg Butter\}$, $\{\neg Bread, Butter\}$, and $\{\neg Bread, \neg Butter\}$. The expected fractional presence of each of these combinations can be quantified as the product of the supports of the states (presence or absence) of the individual items. For a given data set, the *observed* value of the support of a state may vary significantly from the expected value of the support. Let O_i and E_i be the observed and expected values of the absolute support of state i. For example, the expected support E_i of $\{Bread, \neg Butter\}$ is given by the total number of transactions multiplied by each of the fractional supports of *Bread* and $\neg$*Butter*, respectively. Then, the χ^2-measure for set of items X is defined as follows:

$$\chi^2(X) = \sum_{i=1}^{2^{|X|}} \frac{(O_i - E_i)^2}{E_i}. \tag{4.6}$$

For example, when $X = \{Bread, Butter\}$, one would need to perform the summation in Eq. 4.6 over the $2^2 = 4$ states corresponding to $\{Bread, Butter\}$, $\{Bread, \neg Butter\}$, $\{\neg Bread, Butter\}$, and $\{\neg Bread, \neg Butter\}$. A value that is close to 0 indicates statistical independence among the items. Larger values of this quantity indicate greater dependence between the variables. However, large χ^2 values do not reveal whether the dependence between items is positive or negative. This is because the χ^2 test measures dependence between variables, rather than the nature of the correlation between the specific states of these variables.

The χ^2 measure is bit-symmetric because it treats the presence and absence of items in a similar way. The χ^2-test satisfies the *upward closure property* because of which an efficient algorithm can be devised for discovering interesting k-patterns. On the other hand, the computational complexity of the measure in Eq. 4.6 increases exponentially with $|X|$.

4.5.3 Interest Ratio

The interest ratio is a simple and intuitively interpretable measure. The interest ratio of a set of items $\{i_1 \ldots i_k\}$ is denoted as $I(\{i_1, \ldots i_k\})$, and is defined as follows:

$$I(\{i_1 \ldots i_k\}) = \frac{sup(\{i_1 \ldots i_k\})}{\prod_{j=1}^{k} sup(i_j)}. \tag{4.7}$$

When the items are statistically independent, the joint support in the numerator will be equal to the product of the supports in the denominator. Therefore, an interest ratio of 1 is the break-even point. A value greater than 1 indicates that the variables are positively correlated, whereas a ratio of less than 1 is indicative of negative correlation.

When some items are extremely rare, the interest ratio can be misleading. For example, if an item occurs in only a single transaction in a large transaction database, each item that co-occurs with it in that transaction can be paired with it to create a 2-itemset with a very high interest ratio. This is statistically misleading. Furthermore, because the interest ratio does not satisfy the downward closure property, it is difficult to design efficient algorithms for computing it.

4.5.4 Symmetric Confidence Measures

The traditional confidence measure is *asymmetric* between the antecedent and consequent. However, the support measure is symmetric. Symmetric confidence measures can be used to replace the support-confidence framework with a single measure. Let X and Y be two 1-itemsets. Symmetric confidence measures can be derived as a function of the confidence of $X \Rightarrow Y$ and the confidence of $Y \Rightarrow X$. The various symmetric confidence measures can be any one of the minimum, average, or maximum of these two confidence values. The minimum is not desirable when either X or Y is very infrequent, causing the combined measure to be too low. The maximum is not desirable when either X or Y is very frequent, causing the combined measure to be too high. The average provides the most robust trade-off in many scenarios. The measures can be generalized to k-itemsets by using all k possible individual items in the consequent for computation. Interestingly, the *geometric* mean of the two confidences evaluates to the cosine measure, which is discussed below. The computational problem with symmetric confidence measures is that the relevant itemsets satisfying a specific threshold on the measure do not satisfy the downward closure property.

4.5.5 Cosine Coefficient on Columns

The cosine coefficient is usually applied to the rows to determine the similarity among transactions. However, it can also be applied to the columns, to determine the similarity between items. The cosine coefficient is best computed using the vertical *tid* list representation on the corresponding binary vectors. The cosine value on the binary vectors computes to the following:

$$cosine(i,j) = \frac{sup(\{i,j\})}{\sqrt{sup(i)} \cdot \sqrt{sup(j)}}. \tag{4.8}$$

The numerator can be evaluated as the length of the intersection of the *tid* lists of items i and j. The cosine measure can be viewed as the geometric mean of the confidences of the rules $\{i\} \Rightarrow \{j\}$ and $\{j\} \Rightarrow \{i\}$. Therefore, the cosine is a kind of symmetric confidence measure.

4.5.6 Jaccard Coefficient and the Min-hash Trick

The Jaccard coefficient was introduced in Chap. 3 to measure similarity between sets. The *tid* lists on a column can be viewed as a set, and the Jaccard coefficient between two *tid* lists can be used to compute the similarity. Let S_1 and S_2 be two sets. As discussed in Chap. 3, the Jaccard coefficient $J(S_1, S_2)$ between the two sets can be computed as follows:

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}. \tag{4.9}$$

The Jaccard coefficient can easily be generalized to multiway sets, as follows:

$$J(S_1 \dots S_k) = \frac{|\cap S_i|}{|\cup S_i|}. \tag{4.10}$$

When the sets $S_1 \dots S_k$ correspond to the *tid* lists of k items, the intersection and union of the *tid* lists can be used to determine the numerator and denominator of the aforementioned expression. This provides the Jaccard-based significance for that k-itemset. It is possible to use a minimum threshold on the Jaccard coefficient to determine all the relevant itemsets.

A nice property of Jaccard-based significance is that it satisfies the set-wise monotonicity property. The k-way Jaccard coefficient $J(S_1 \dots S_k)$ is always no smaller than the $(k+1)$-way Jaccard coefficient $J(S_1 \dots S_{k+1})$. This is because the numerator of the Jaccard coefficient is monotonically non-increasing with increasing values of k (similar to support), whereas the denominator is monotonically non-decreasing. Therefore, the Jaccard coefficient cannot increase with increasing values of k. Therefore, when a minimum threshold is used on the Jaccard-based significance of an itemset, the resulting itemsets satisfy the downward closure property, as well. This means that most of the traditional algorithms, such as *Apriori* and enumeration tree methods, can be generalized to the Jaccard coefficient quite easily.

It is possible to use sampling to speed up the computation of the Jaccard coefficient further, and transform it to a standard frequent pattern mining problem. This kind of sampling uses hash functions to simulate sorted samples of the data. So, how can the Jaccard coefficient be computed using sorted sampling? Let D be the $n \times d$ binary data matrix representing the n rows and d columns. Without loss of generality, consider the case when the Jaccard coefficient needs to be computed on the first k columns. Suppose one were to sort the rows in D, and pick the first row in which *at least* one of the first k columns in this row has a value of 1 in this column. Then, it is easy to see that the probability of the event that *all* the k columns have a value of 1 is equal to the k-way Jaccard coefficient. If one were to sort the rows multiple times, it is possible to estimate this probability as the fraction of sorts over which the event of all k columns taking on unit values occurs. Of course, it is rather inefficient to do it in this way because every sort requires a pass over the database. Furthermore, this approach can only estimate the Jaccard coefficient for a *particular* set of k columns, and it does not *discover* all the k-itemsets that satisfy the minimum criterion on the Jaccard coefficient.

The *min-hash* trick can be used to efficiently perform the sorts in an implicit way and transform to a concise sampled representation on which traditional frequent pattern mining algorithms can be applied to discover combinations satisfying the Jaccard threshold. The basic idea is as follows. A random hash function $h(\cdot)$ is applied to each *tid*. For each column of binary values, the *tid*, with the smallest hash function value, is selected among all entries *that have a unit value in that column*. This results in a vector of d different *tids*. What is the probability that the *tids* in the first k columns are the same? It is easy to see that this is equal to the Jaccard coefficient because the hashing process simulates the sort, and reports the *index* of the first non-zero element in the binary matrix. Therefore, by using independent hash functions to create multiple *samples*, it is possible to *estimate* the Jaccard coefficient. It is possible to repeat this process with r different hash functions, to create r different samples. Note that the r hash-functions can be applied simultaneously in a single pass over the transaction database. This creates a $r \times d$ categorical data matrix of *tids*. By determining the subsets of columns where the *tid* value is the same with support equal to a minimum support value, it is possible to estimate all sets of k-items whose Jaccard coefficient is at least equal to the minimum support value. This is a standard frequent pattern mining problem, except that it is defined on categorical values instead of a binary data matrix.

One way of transforming this $r \times d$ categorical data matrix to a binary matrix is to pull out the column identifiers where the *tids* are the same from each row and create a new transaction of column-identifier "items." Thus, a single row from the $r \times d$ matrix will map to multiple transactions. The resulting transaction data set can be represented by a new binary matrix D'. Any off-the-shelf frequent pattern mining algorithm can be applied to this binary matrix to discover relevant column-identifier combinations. The advantage of an off-the-shelf approach is that many efficient algorithms for the conventional frequent pattern mining model are available. It can be shown that the accuracy of the approach increases exponentially fast with the number of data samples.

4.5.7 Collective Strength

The collective strength of an itemset is defined in terms of its *violation rate*. An itemset I is said to be in *violation* of a transaction, if some of the items are present in the transaction, and others are not. The *violation rate* $v(I)$ of an itemset I is the fraction of violations of the itemset I over all transactions. The *collective strength* $C(I)$ of an itemset I is defined in terms of the violation rate as follows:

$$C(I) = \frac{1 - v(I)}{1 - E[v(I)]} \cdot \frac{E[v(I)]}{v(I)}. \tag{4.11}$$

The collective strength is a number between 0 to ∞. A value of 0 indicates a perfect negative correlation, whereas a value of ∞ indicates a perfectly positive correlation. The value of 1 is the break-even point. The expected value of $v(I)$ is calculated assuming statistical independence of the individual items. No violation occurs when all items in I are included in transaction, *or* when no items in I are included in a transaction. Therefore, if p_i is the fraction of transactions in which the item i occurs, we have:

$$E[v(I)] = 1 - \prod_{i \in I} p_i - \prod_{i \in I}(1 - p_i). \tag{4.12}$$

Intuitively, if the violation of an itemset in a transaction is a "bad event" from the perspective of trying to establish a high correlation among items, then $v(I)$ is the fraction of bad events, and $(1 - v(I))$ is the fraction of "good events." Therefore, collective strength may be understood as follows:

$$C(I) = \frac{\text{Good Events}}{\text{E[Good Events]}} \cdot \frac{\text{E[Bad Events]}}{\text{Bad Events}}. \tag{4.13}$$

The concept of collective-strength may be strengthened to *strongly collective* itemsets.

Definition 4.5.1 *An itemset I is denoted to be strongly collective at level s, if it satisfies the following properties:*

1. *The collective strength $C(I)$ of the itemset I is at least s.*
2. **Closure property:** *The collective strength $C(J)$ of every subset J of I is at least s.*

It is necessary to force the closure property to ensure that unrelated items may not be present in an itemset. Consider, for example, the case when itemset I_1 is $\{Milk, Bread\}$ and itemset I_2 is $\{Diaper, Beer\}$. If I_1 and I_2 each have a high collective strength, then it may often be the case that the itemset $I_1 \cup I_2$ may also have a high collective strength, even though items such as milk and beer may be independent. Because of the closure property of this definition, it is possible to design an *Apriori*-like algorithm for the problem.

4.5.8 Relationship to Negative Pattern Mining

In many applications, it is desirable to determine patterns between items *or their absence.* Negative pattern mining requires the use of *bit-symmetric* measures that treat the presence or absence of an item evenly. The traditional support-confidence measure is not designed for finding such patterns. Measures such as the statistical coefficient of correlation, χ^2 measure, and collective strength are better suited for finding such positive or negative correlations between items. However, many of these measures are hard to use in practice because they do not satisfy the downward closure property. The multiway Jaccard coefficient and collective strength are among the few measures that do satisfy the downward closure property.

4.6 Useful Meta-algorithms

A number of meta-algorithms can be used to obtain different insights from pattern mining. A *meta-algorithm* is defined as an algorithm that uses a particular algorithm as a subroutine, either to make the original algorithm more efficient (e.g., by sampling), or to gain new insights. Two types of meta-algorithms are most common in pattern mining. The first type uses sampling to improve the efficiency of association pattern mining algorithms. The second uses preprocessing and postprocessing subroutines to apply the algorithm to other scenarios. For example, after using these wrappers, standard frequent pattern mining algorithms can be applied to quantitative or categorical data.

4.6.1 Sampling Methods

When the transaction database is very large, it cannot be stored in main memory. This makes the application of frequent pattern mining algorithms more challenging. This is because such databases are typically stored on disk, and only level-wise algorithms may be used. Many depth-first algorithms on the enumeration tree may be challenged by these scenarios because they require random access to the transactions. This is inefficient for disk-resident data. As discussed earlier, such depth-first algorithms are usually the most efficient for *memory-resident* data. By sampling, it is possible to apply many of these algorithms in an efficient way, with only limited loss in accuracy. When a standard itemset mining algorithm is applied to sampled data, it will encounter two main challenges:

1. *False positives:* These are patterns that meet the support threshold on the sample but not on the base data.

2. *False negatives:* These are patterns that do not meet the support threshold on the sample, but meet the threshold on the data.

False positives are easier to address than false negatives because the former can be removed by scanning the disk-resident database only once. However, to address false negatives, one needs to reduce the support thresholds. By reducing support thresholds, it is possible to probabilistically guarantee the level of loss for specific thresholds. Pointers to these probabilistic guarantees may be found in the bibliographic notes. Reducing the support thresholds too much will lead to many spurious itemsets and increase the work in the postprocessing phase. Typically, the number of false positives increases rapidly with small changes in support levels.

4.6.2 Data Partitioned Ensembles

One approach that can guarantee no false positives and no false negatives, is the use of partitioned ensembles by the *Partition* algorithm [446]. This approach may be used either for reduction of disk-access costs or for reduction of memory requirements of projection-based algorithms. In partitioned ensembles, the transaction database is partitioned into k disjoint segments, each of which is main-memory resident. The frequent itemset mining algorithm is independently applied to each of these k different segments with the required minimum support level. An important property is that every frequent pattern *must* appear in at least one of the segments. Otherwise, its cumulative support across different segments will not meet the minimum support requirement. Therefore, the union of the frequent itemset generated from different segments provides a superset of the frequent patterns. In other words, the union contains false positives but no false negatives. A postprocessing phase of support counting can be applied to this superset to remove the false positives. This approach is particularly useful for memory-intensive projection-based algorithms when the projected databases do not fit in main memory. In the original *Partition* algorithm, the data structure used to perform projection-based reuse was the vertical *tid* list. While partitioning is almost always necessary for memory-based implementations of projection-based algorithms in databases of arbitrarily large size, the cost of postprocessing overhead can sometimes be significant. Therefore, one should use the minimum number of partitions based on the available memory. Although *Partition* is well known mostly for its ensemble approach, an even more significant but unrecognized contribution of the method was to propose the notion of vertical lists. The approach is credited with recognizing the projection-based reuse properties of recursive *tid* list intersections.

4.6.3 Generalization to Other Data Types

The generalization to other data types is quite straightforward with the use of type-transformation methods discussed in Chap. 2.

4.6.3.1 Quantitative Data

In many applications, it is desirable to discover quantitative association rules when some of the attributes take on quantitative values. Many online merchants collect profile information, such as age, which have numeric values. For example, in supermarket applications, it may be desirable to relate demographic information to item attributes in the data. An example of such a rule is as follows:

$$(Age = 90) \Rightarrow Checkers.$$

This rule may not have sufficient support if the transactions do not contain enough individuals of that age. However, the rule may be relevant to the broader age group. Therefore, one possibility is to create a rule that groups the different ages into one range:

$$Age[85, 95] \Rightarrow Checkers.$$

This rule will have the required level of minimum support. In general, for quantitative association rule mining, the quantitative attributes are discretized and converted to binary form. Thus, the entire data set (including the item attributes) can be represented as a binary matrix. A challenge with the use of such an approach is that the appropriate level of discretization is often hard to know *a priori*. A standard association rule mining algorithm may be applied to this representation. Furthermore, rules on adjacent ranges can be merged to create summarized rules on larger ranges.

4.6.3.2 Categorical Data

Categorical data is common in many application domains. For example, attributes such as the gender and ZIP code are typical categorical. In other cases, the quantitative and categorical data may be mixed. An example of a rule with mixed attributes is as follows:

$$(Gender = Male), \quad Age[20, 30] \Rightarrow Basketball.$$

Categorical data can be transformed to binary values with the use of the binarization approach discussed in Chap. 2. For each categorical attribute value, a single binary value is used to indicate the presence or absence of the item. This can be used to determine the association rules. In some cases, when domain knowledge is available, clusters on categorical values on may used as binary attributes. For example, the ZIP codes may be clustered by geography into k clusters, and then these k clusters may be treated as binary attributes.

4.7 Summary

The problem of association rule mining is used to identify relationships between different attributes. Association rules are typically generated using a two-phase framework. In the first phase, all the patterns that satisfy the minimum support requirement are determined. In the second phase, rules that satisfy the minimum confidence requirement are generated from the patterns.

The *Apriori* algorithm is one of the earliest and most well known methods for frequent pattern mining. In this algorithm, candidate patterns are generated with the use of joins between frequent patterns. Subsequently, a number of enumeration-tree algorithms were proposed for frequent pattern mining techniques. Many of these methods use projections to count the support of transactions in the database more efficiently. The traditional support-confidence framework has the shortcoming that it is not based on robust statistical measures. Many of the patterns generated are not interesting. Therefore, a number of interest measures have been proposed for determining more relevant patterns.

A number of sampling methods have been designed for improving the efficiency of frequent pattern mining. Sampling methods result in both false positives and false negatives, though the former can be addressed by postprocessing. A partitioned sample ensemble is also able to avoid false negatives. Association rules can be determined in quantitative and categorical data with the use of type transformations.

4.8 Bibliographic Notes

The problem of frequent pattern mining was first proposed in [55]. The *Apriori* algorithm discussed in this chapter was first proposed in [56], and an enhanced variant of the approach was proposed in [57]. Maximal and non-maximal frequent pattern mining algorithms are usually different from one another primarily in terms of additional pruning steps in the former. The *MaxMiner* algorithm used superset-based non-maximality pruning [82] for more efficient counting. However, the exploration is in breadth-first order, to reduce the number of passes over the data. The *DepthProject* algorithm recognized that superset-based non-maximality pruning is more effective with a depth-first approach.

The *FP-growth* [252] and *DepthProject* [3, 4] methods independently proposed the notion of projection-based reuse in the horizontal database layout. A variety of different data structures are used by different projection-based reuse algorithms such as *TreeProjection* [3], *DepthProject* [4], *FP-growth* [252], and *H-Mine* [419]. A method, known as *Opportune-Project* [361], chooses opportunistically between array-based and tree-based structures to represent the projected transactions. The *TreeProjection* framework also recognized that breadth-first and depth-first strategies have different trade-offs. Breadth-first variations of *TreeProjection* sacrifice some of the power of projection-based reuse to enable fewer disk-based passes on arbitrarily large data sets. Depth-first variations of *TreeProjection*, such as *DepthProject*, achieve full projection-based reuse but the projected databases need to be consistently maintained in main memory. A book and a survey on frequent pattern mining methods may be found in [34] and [253], respectively.

The use of the vertical representation for frequent pattern mining was independently pioneered by Holsheimer et al. [273] and Savasere et al. [446]. These works introduced the clever insight that *recursive tid* list intersections provide significant computational savings in support counting because k-itemsets have shorter *tid* lists than those of $(k-1)$-itemsets or individual items. The vertical *Apriori* algorithm is based on an ensemble component of the *Partition* framework [446]. Although the use of vertical lists by this algorithm was mentioned [537, 534, 465] in the earliest vertical pattern mining papers, some of the contributions of the *Partition* algorithm and their relationship to the subsequent work seem to have remained unrecognized by the research community over the years. Savasere et al.'s *Apriori*-like algorithm, in fact, formed the basis for all vertical algorithms such as *Eclat* [534] and *VIPER* [465]. *Eclat* is described as a breadth-first algorithm in the book by Han et al. [250], and as a depth-first algorithm in the book by Zaki et al. [536]. A careful examination of the

Eclat paper [537] reveals that it is a memory optimization of the breadth-first approach by Savasere et al. [446]. The main contribution of *Eclat* is a memory optimization of the individual ensemble component of Savasere et al.'s algorithm with lattice partitioning (instead of data partitioning), thereby increasing the maximum size of the databases that can be processed in memory without the computational overhead of data-partitioned postprocessing. The number of computational operations for support counting in a *single* component version of *Partition* is fundamentally no different from that of *Eclat*. The *Eclat* algorithm partitions the lattice based on common prefixes, calling them equivalence classes, and then uses a breadth-first approach [537] over each of these smaller sublattices in main memory. This type of lattice partitioning was adopted from parallel versions of *Apriori*, such as the *Candidate Distribution* algorithm [54], where a similar choice exists between lattice partitioning and data partitioning. Because a breadth-first approach is used for search on each sublattice, such an approach has significantly higher memory requirements than a pure depth-first approach. As stated in [534], *Eclat* explicitly *decouples* the lattice decomposition phase from the pattern search phase. This is different from a pure depth-first strategy in which both are tightly integrated. Depth-first algorithms do not require an explicitly decoupled approach for reduction of memory requirements. Therefore, the lattice-partitioning in *Eclat*, which was motivated by the *Candidate Distribution* algorithm [54], seems to have been specifically designed with a breadth-first approach in mind for the second (pattern search) phase. Both the conference [537] and journal versions [534] of the *Eclat* algorithm state that a breadth-first (bottom-up) procedure is used in the second phase for all experiments. *FP-growth* [252] and *DepthProject* [4] were independently proposed as the first depth-first algorithms for frequent pattern mining. *MAFIA* was the first vertical method to use a pure depth-first approach [123]. Other later variations of vertical algorithms, such as *GenMax* and *dEclat* [233, 538], also incorporated the depth-first approach. The notion of *diffsets* [538, 233], which uses incremental vertical lists along the enumeration tree hierarchy, was also proposed in these algorithms. The approach provides memory and efficiency advantages for certain types of data sets.

Numerous measures for finding interesting frequent patterns have been proposed. The χ^2 measure was one of the first such tests, and was discussed in [113]. This measure satisfies the *upward closure property*. Therefore, efficient pattern mining algorithms can be devised. The use of the min-hashing technique for determining interesting patterns without support counting was discussed in [180]. The impact of skews in the support of individual items has been addressed in [517]. An affinity-based algorithm for mining interesting patterns in data with skews has been proposed in the same work. A common scenario in which there is significant skew in support distributions is that of mining negative association rules [447]. The collective strength model was proposed in [16], and a level-wise algorithm for finding all strongly collective itemsets was discussed in the same work. The collective strength model can also discover negative associations from the data. The work in [486] addresses the problem of selecting the right measure for finding interesting association rules.

Sampling is a popular approach for finding frequent patterns in an efficient way with memory-resident algorithms. The first sampling approach was discussed in [493], and theoretical bounds were presented. The work in [446] enables the application of memory-based frequent pattern mining algorithms on large data sets by using ensembles on data partitions. The problem of finding quantitative association rules, and different kinds of patterns from quantitative data is discussed in [476]. The *CLIQUE* algorithm can also be considered an association pattern mining algorithm on quantitative data [58].

4.9 Exercises

1. Consider the transaction database in the table below:

tid	Items
1	a, b, c, d
2	b, c, e, f
3	a, d, e, f
4	a, e, f
5	b, d, f

Determine the absolute support of itemsets $\{a, e, f\}$, and $\{d, f\}$. Convert the absolute support to the relative support.

2. For the database in Exercise 1, compute all frequent patterns at absolute minimum support values of 2, 3, and 4.

3. For the database in Exercise 1, determine all the maximal frequent patterns at absolute minimum support values of 2, 3, and 4.

4. Represent the database of Exercise 1 in vertical format.

5. Consider the transaction database in the table below:

tid	items
1	a, c, d, e
2	a, d, e, f
3	b, c, d, e, f
4	b, d, e, f
5	b, e, f
6	c, d, e
7	c, e, f
8	d, e, f

Determine all frequent patterns and maximal patterns at support levels of 3, 4, and 5.

6. Represent the transaction database of Exercise 5 in vertical format.

7. Determine the confidence of the rules $\{a\} \Rightarrow \{f\}$, and $\{a, e\} \Rightarrow \{f\}$ for the transaction database in Exercise 1.

8. Determine the confidence of the rules $\{a\} \Rightarrow \{f\}$, and $\{a, e\} \Rightarrow \{f\}$ for the transaction database in Exercise 5.

9. Show the candidate itemsets and the frequent itemsets in each level-wise pass of the *Apriori* algorithm in Exercise 1. Assume an absolute minimum support level of 2.

10. Show the candidate itemsets and the frequent itemsets in each level-wise pass of the *Apriori* algorithm in Exercise 5. Assume an absolute minimum support level of 3.

11. Show the prefix-based enumeration tree of frequent itemsets, for the data set of Exercise 1 at an absolute minimum support level of 2. Assume a lexicographic ordering of a, b, c, d, e, f. Construct the tree for the reverse lexicographic ordering.

12. Show the prefix-based enumeration tree of frequent itemsets, for the data set in Exercise (5), at an absolute minimum support of 3. Assume a lexicographic ordering of a, b, c, d, e, f. Construct the tree for the reverse lexicographic ordering.

13. Show the frequent suffixes generated in the recursion tree of the generic pattern growth method for the data set and support level in Exercise 9. Assume the lexicographic ordering of a, b, c, d, e, f, and f, e, d, c, b, a. How do these trees compare with those generated in Exercise 11?

14. Show the frequent suffixes generated in the recursion tree of the generic pattern growth method for the data set and support level in Exercise 10. Assume the lexicographic ordering of a, b, c, d, e, f, and f, e, d, c, b, a. How do these trees compare with those generated in Exercise 12?

15. Construct a prefix-based FP-Tree for the lexicographic ordering a, b, c, d, e, f for the data set in Exercise 1. Create the same tree for the reverse lexicographic ordering.

16. Construct a prefix-based FP-Tree for the lexicographic ordering a, b, c, d, e, f for the data set in Exercise 5. Create the same tree for the reverse lexicographic ordering.

17. The pruning approach in *Apriori* was inherently designed for a breadth-first strategy because all frequent k-itemsets are generated before $(k+1)$-itemsets. Discuss how one might implement such a pruning strategy with a depth-first algorithm.

18. Implement the pattern growth algorithm with the use of (a) an array-based data structure, (b) a pointer-based data structure with no FP-Tree, and (c) a pointer-based data structure with FP-Tree.

19. Implement Exercise 18(c) by growing patterns from prefixes and the FP-Tree on suffixes.

20. For the itemset $\{d, f\}$ and the data set of Exercise 1, compute the (a) statistical correlation coefficient, (b) interest ratio, (c) cosine coefficient, and (d) Jaccard coefficient.

21. For the itemset $\{d, f\}$ and the data set of Exercise 1, compute the (a) statistical correlation coefficient, (b) interest ratio, (c) cosine coefficient, and (d) Jaccard coefficient.

22. Discuss the similarities and differences between *TreeProjection*, *DepthProject*, *VerticalApriori*, and *FP-growth*.