CHAPTER 8 Itemset Mining

In many applications one is interested in how often two or more objects of interest co-occur. For example, consider a popular website, which logs all incoming traffic to its site in the form of weblogs. Weblogs typically record the source and destination pages requested by some user, as well as the time, return code whether the request was successful or not, and so on. Given such weblogs, one might be interested in finding if there are sets of web pages that many users tend to browse whenever they visit the website. Such "frequent" sets of web pages give clues to user browsing behavior and can be used for improving the browsing experience.

The quest to mine frequent patterns appears in many other domains. The prototypical application is *market basket analysis*, that is, to mine the sets of items that are frequently bought together at a supermarket by analyzing the customer shopping carts (the so-called "market baskets"). Once we mine the frequent sets, they allow us to extract *association rules* among the item sets, where we make some statement about how likely are two sets of items to co-occur or to conditionally occur. For example, in the weblog scenario frequent sets allow us to extract rules like, "Users who visit the sets of pages `main`, `laptops` and `rebates` also visit the pages `shopping-cart` and `checkout`", indicating, perhaps, that the special rebate offer is resulting in more laptop sales. In the case of market baskets, we can find rules such as "Customers who buy milk and cereal also tend to buy bananas," which may prompt a grocery store to co-locate bananas in the cereal aisle. We begin this chapter with algorithms to mine frequent itemsets, and then show how they can be used to extract association rules.

8.1 FREQUENT ITEMSETS AND ASSOCIATION RULES

Itemsets and Tidsets

Let $\mathcal{I} = \{x_1, x_2, \ldots, x_m\}$ be a set of elements called *items*. A set $X \subseteq \mathcal{I}$ is called an *itemset*. The set of items $\mathcal{I}$ may denote, for example, the collection of all products sold at a supermarket, the set of all web pages at a website, and so on. An itemset of cardinality (or size) k is called a k-itemset. Further, we denote by $\mathcal{I}^{(k)}$ the set of all k-itemsets, that is, subsets of $\mathcal{I}$ with size k. Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be another set of elements called

transaction identifiers or *tids*. A set $T \subseteq \mathcal{T}$ is called a *tidset*. We assume that itemsets and tidsets are kept sorted in lexicographic order.

A *transaction* is a tuple of the form $\langle t, X \rangle$, where $t \in \mathcal{T}$ is a unique transaction identifier, and X is an itemset. The set of transactions $\mathcal{T}$ may denote the set of all customers at a supermarket, the set of all the visitors to a website, and so on. For convenience, we refer to a transaction $\langle t, X \rangle$ by its identifier t.

Database Representation

A binary database $\mathbf{D}$ is a binary relation on the set of tids and items, that is, $\mathbf{D} \subseteq \mathcal{T} \times \mathcal{I}$. We say that tid $t \in \mathcal{T}$ *contains* item $x \in \mathcal{I}$ iff $(t, x) \in \mathbf{D}$. In other words, $(t, x) \in \mathbf{D}$ iff $x \in X$ in the tuple $\langle t, X \rangle$. We say that tid t *contains* itemset $X = \{x_1, x_2, \ldots, x_k\}$ iff $(t, x_i) \in \mathbf{D}$ for all $i = 1, 2, \ldots, k$.

Example 8.1. Figure 8.1a shows an example binary database. Here $\mathcal{I} = \{A, B, C, D, E\}$, and $\mathcal{T} = \{1, 2, 3, 4, 5, 6\}$. In the binary database, the cell in row t and column x is 1 iff $(t, x) \in \mathbf{D}$, and 0 otherwise. We can see that transaction 1 contains item B, and it also contains the itemset BE, and so on.

For a set X, we denote by 2^X the powerset of X, that is, the set of all subsets of X. Let $\mathbf{i} : 2^{\mathcal{T}} \to 2^{\mathcal{I}}$ be a function, defined as follows:

$$\mathbf{i}(T) = \{x \mid \forall t \in T,\ t \text{ contains } x\} \tag{8.1}$$

where $T \subseteq \mathcal{T}$, and $\mathbf{i}(T)$ is the set of items that are common to *all* the transactions in the tidset T. In particular, $\mathbf{i}(t)$ is the set of items contained in tid $t \in \mathcal{T}$. Note that in this chapter we drop the set notation for convenience (e.g., we write $\mathbf{i}(t)$ instead of $\mathbf{i}(\{t\})$). It is sometimes convenient to consider the binary database $\mathbf{D}$, as a *transaction database* consisting of tuples of the form $\langle t, \mathbf{i}(t) \rangle$, with $t \in \mathcal{T}$. The transaction or itemset database can be considered as a horizontal representation of the binary database, where we omit items that are not contained in a given tid.

Let $\mathbf{t} : 2^{\mathcal{I}} \to 2^{\mathcal{T}}$ be a function, defined as follows:

$$\mathbf{t}(X) = \{t \mid t \in \mathcal{T} \text{ and } t \text{ contains } X\} \tag{8.2}$$

where $X \subseteq \mathcal{I}$, and $\mathbf{t}(X)$ is the set of tids that contain *all* the items in the itemset X. In particular, $\mathbf{t}(x)$ is the set of tids that contain the single item $x \in \mathcal{I}$. It is also sometimes convenient to think of the binary database $\mathbf{D}$, as a tidset database containing a collection of tuples of the form $\langle x, \mathbf{t}(x) \rangle$, with $x \in \mathcal{I}$. The tidset database is a vertical representation of the binary database, where we omit tids that do not contain a given item.

Example 8.2. Figure 8.1b shows the corresponding transaction database for the binary database in Figure 8.1a. For instance, the first transaction is $\langle 1, \{A, B, D, E\} \rangle$, where we omit item C since $(1, C) \notin \mathbf{D}$. Henceforth, for convenience, we drop the set notation for itemsets and tidsets if there is no confusion. Thus, we write $\langle 1, \{A, B, D, E\} \rangle$ as $\langle 1, ABDE \rangle$.

D	A	B	C	D	E
1	1	1	0	1	1
2	0	1	1	0	1
3	1	1	0	1	1
4	1	1	1	0	1
5	1	1	1	1	1
6	0	1	1	1	0

(a) Binary database

t	$\mathbf{i}(t)$
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

(b) Transaction database

x	A	B	C	D	E
$\mathbf{t}(x)$	1	1	2	1	1
	3	2	4	3	2
	4	3	5	5	3
	5	4	6	6	4
		5			5
		6			

(c) Vertical database

Figure 8.1. An example database.

Figure 8.1c shows the corresponding vertical database for the binary database in Figure 8.1a. For instance, the tuple corresponding to item A, shown in the first column, is $\langle A, \{1,3,4,5\}\rangle$, which we write as $\langle A, 1345\rangle$ for convenience; we omit tids 2 and 6 because $(2, A) \notin \mathbf{D}$ and $(6, A) \notin \mathbf{D}$.

Support and Frequent Itemsets

The *support* of an itemset X in a dataset $\mathbf{D}$, denoted $sup(X, \mathbf{D})$, is the number of transactions in $\mathbf{D}$ that contain X:

$$sup(X, \mathbf{D}) = \big|\{t \mid \langle t, \mathbf{i}(t)\rangle \in \mathbf{D} \text{ and } X \subseteq \mathbf{i}(t)\}\big| = |\mathbf{t}(X)|$$

The *relative support* of X is the fraction of transactions that contain X:

$$rsup(X, \mathbf{D}) = \frac{sup(X, \mathbf{D})}{|\mathbf{D}|}$$

It is an estimate of the *joint probability* of the items comprising X.

An itemset X is said to be *frequent* in $\mathbf{D}$ if $sup(X, \mathbf{D}) \geq minsup$, where $minsup$ is a user defined *minimum support threshold*. When there is no confusion about the database $\mathbf{D}$, we write support as $sup(X)$, and relative support as $rsup(X)$. If $minsup$ is specified as a fraction, then we assume that relative support is implied. We use the set $\mathcal{F}$ to denote the set of all frequent itemsets, and $\mathcal{F}^{(k)}$ to denote the set of frequent k-itemsets.

Example 8.3. Given the example dataset in Figure 8.1, let $minsup = 3$ (in relative support terms we mean $minsup = 0.5$). Table 8.1 shows all the 19 frequent itemsets in the database, grouped by their support value. For example, the itemset BCE is contained in tids 2, 4, and 5, so $\mathbf{t}(BCE) = 245$ and $sup(BCE) = |\mathbf{t}(BCE)| = 3$. Thus, BCE is a frequent itemset. The 19 frequent itemsets shown in the table comprise the set $\mathcal{F}$. The sets of all frequent k-itemsets are

$$\mathcal{F}^{(1)} = \{A, B, C, D, E\}$$
$$\mathcal{F}^{(2)} = \{AB, AD, AE, BC, BD, BE, CE, DE\}$$
$$\mathcal{F}^{(3)} = \{ABD, ABE, ADE, BCE, BDE\}$$
$$\mathcal{F}^{(4)} = \{ABDE\}$$

Table 8.1. Frequent itemsets with $minsup = 3$

sup	itemsets
6	B
5	E, BE
4	$A, C, D, AB, AE, BC, BD, ABE$
3	$AD, CE, DE, ABD, ADE, BCE, BDE, ABDE$

Association Rules

An *association rule* is an expression $X \xrightarrow{s,c} Y$, where X and Y are itemsets and they are disjoint, that is, $X, Y \subseteq \mathcal{I}$, and $X \cap Y = \emptyset$. Let the itemset $X \cup Y$ be denoted as XY. The *support* of the rule is the number of transactions in which both X and Y co-occur as subsets:

$$s = sup(X \longrightarrow Y) = |\mathbf{t}(XY)| = sup(XY)$$

The *relative support* of the rule is defined as the fraction of transactions where X and Y co-occur, and it provides an estimate of the joint probability of X and Y:

$$rsup(X \longrightarrow Y) = \frac{sup(XY)}{|\mathbf{D}|} = P(X \wedge Y)$$

The *confidence* of a rule is the conditional probability that a transaction contains Y given that it contains X:

$$c = conf(X \longrightarrow Y) = P(Y|X) = \frac{P(X \wedge Y)}{P(X)} = \frac{sup(XY)}{sup(X)}$$

A rule is *frequent* if the itemset XY is frequent, that is, $sup(XY) \geq minsup$ and a rule is *strong* if $conf \geq minconf$, where *minconf* is a user-specified minimum confidence threshold.

Example 8.4. Consider the association rule $BC \longrightarrow E$. Using the itemset support values shown in Table 8.1, the support and confidence of the rule are as follows:

$$s = sup(BC \longrightarrow E) = sup(BCE) = 3$$

$$c = conf(BC \longrightarrow E) = \frac{sup(BCE)}{sup(BC)} = 3/4 = 0.75$$

Itemset and Rule Mining

From the definition of rule support and confidence, we can observe that to generate frequent and high confidence association rules, we need to first enumerate all the frequent itemsets along with their support values. Formally, given a binary database $\mathbf{D}$ and a user defined minimum support threshold *minsup*, the task of frequent itemset mining is to enumerate all itemsets that are frequent, i.e., those that have support at least *minsup*. Next, given the set of frequent itemsets $\mathcal{F}$ and a minimum confidence value *minconf*, the association rule mining task is to find all frequent and strong rules.

8.2 ITEMSET MINING ALGORITHMS

We begin by describing a naive or brute-force algorithm that enumerates all the possible itemsets $X \subseteq \mathcal{I}$, and for each such subset determines its support in the input dataset $\mathbf{D}$. The method comprises two main steps: (1) candidate generation and (2) support computation.

Candidate Generation

This step generates all the subsets of $\mathcal{I}$, which are called *candidates*, as each itemset is potentially a candidate frequent pattern. The candidate itemset search space is clearly exponential because there are $2^{|\mathcal{I}|}$ potentially frequent itemsets. It is also instructive to note the structure of the itemset search space; the set of all itemsets forms a lattice structure where any two itemsets X and Y are connected by a link iff X is an *immediate subset* of Y, that is, $X \subseteq Y$ and $|X| = |Y| - 1$. In terms of a practical search strategy, the itemsets in the lattice can be enumerated using either a breadth-first (BFS) or depth-first (DFS) search on the *prefix tree*, where two itemsets X, Y are connected by a link iff X is an immediate subset and prefix of Y. This allows one to enumerate itemsets starting with an empty set, and adding one more item at a time.

Support Computation

This step computes the support of each candidate pattern X and determines if it is frequent. For each transaction $\langle t, \mathbf{i}(t) \rangle$ in the database, we determine if X is a subset of $\mathbf{i}(t)$. If so, we increment the support of X.

The pseudo-code for the brute-force method is shown in Algorithm 8.1. It enumerates each itemset $X \subseteq \mathcal{I}$, and then computes its support by checking if $X \subseteq \mathbf{i}(t)$ for each $t \in \mathcal{T}$.

ALGORITHM 8.1. Algorithm BRUTEFORCE

```
BRUTEFORCE (D, I, minsup):
1  F ← ∅ // set of frequent itemsets
2  foreach X ⊆ I do
3      sup(X) ← COMPUTESUPPORT (X, D)
4      if sup(X) ≥ minsup then
5          F ← F ∪ {(X, sup(X))}
6  return F

COMPUTESUPPORT (X, D):
7  sup(X) ← 0
8  foreach ⟨t, i(t)⟩ ∈ D do
9      if X ⊆ i(t) then
10         sup(X) ← sup(X) + 1
11 return sup(X)
```

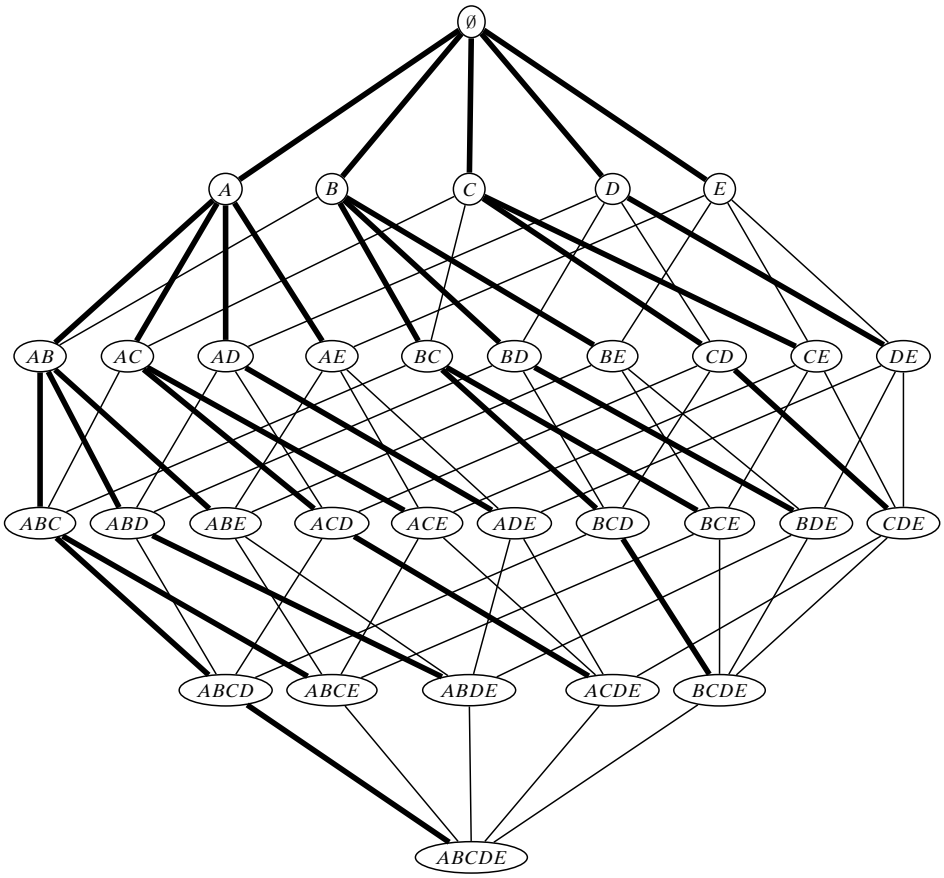


Figure 8.2. Itemset lattice and prefix-based search tree (in bold).

Example 8.5. Figure 8.2 shows the itemset lattice for the set of items $\mathcal{I} = \{A, B, C, D, E\}$. There are $2^{|\mathcal{I}|} = 2^5 = 32$ possible itemsets including the empty set. The corresponding prefix search tree is also shown (in bold). The brute-force method explores the entire itemset search space, regardless of the *minsup* threshold employed. If $minsup = 3$, then the brute-force method would output the set of frequent itemsets shown in Table 8.1.

Computational Complexity

Support computation takes time $O(|\mathcal{I}| \cdot |\mathbf{D}|)$ in the worst case, and because there are $O(2^{|\mathcal{I}|})$ possible candidates, the computational complexity of the brute-force method is $O(|\mathcal{I}| \cdot |\mathbf{D}| \cdot 2^{|\mathcal{I}|})$. Because the database $\mathbf{D}$ can be very large, it is also important to measure the input/output (I/O) complexity. Because we make one complete database scan to compute the support of each candidate, the I/O complexity of BRUTEFORCE is $O(2^{|\mathcal{I}|})$ database scans. Thus, the brute force approach is computationally infeasible for even small itemset spaces, whereas in practice $\mathcal{I}$ can be very large (e.g., a supermarket carries thousands of items). The approach is impractical from an I/O perspective as well.

We shall see next how to systematically improve on the brute force approach, by improving both the candidate generation and support counting steps.

8.2.1 Level-wise Approach: Apriori Algorithm

The brute force approach enumerates all possible itemsets in its quest to determine the frequent ones. This results in a lot of wasteful computation because many of the candidates may not be frequent. Let $X, Y \subseteq \mathcal{I}$ be any two itemsets. Note that if $X \subseteq Y$, then $sup(X) \geq sup(Y)$, which leads to the following two observations: (1) if X is frequent, then any subset $Y \subseteq X$ is also frequent, and (2) if X is not frequent, then any superset $Y \supseteq X$ cannot be frequent. The *Apriori algorithm* utilizes these two properties to significantly improve the brute-force approach. It employs a level-wise or breadth-first exploration of the itemset search space, and prunes all supersets of any infrequent candidate, as no superset of an infrequent itemset can be frequent. It also avoids generating any candidate that has an infrequent subset. In addition to improving the candidate generation step via itemset pruning, the Apriori method also significantly improves the I/O complexity. Instead of counting the support for a single itemset, it explores the prefix tree in a breadth-first manner, and computes the support of all the valid candidates of size k that comprise level k in the prefix tree.

Example 8.6. Consider the example dataset in Figure 8.1; let $minsup = 3$. Figure 8.3 shows the itemset search space for the Apriori method, organized as a prefix tree where two itemsets are connected if one is a prefix and immediate subset of the other. Each node shows an itemset along with its support, thus $AC(2)$ indicates that $sup(AC) = 2$. Apriori enumerates the candidate patterns in a level-wise manner, as shown in the figure, which also demonstrates the power of pruning the search space via the two Apriori properties. For example, once we determine that AC is infrequent, we can prune any itemset that has AC as a prefix, that is, the entire subtree under AC can be pruned. Likewise for CD. Also, the extension BCD from BC can be pruned, since it has an infrequent subset, namely CD.

Algorithm 8.2 shows the pseudo-code for the Apriori method. Let $\mathcal{C}^{(k)}$ denote the prefix tree comprising all the candidate k-itemsets. The method begins by inserting the single items into an initially empty prefix tree to populate $\mathcal{C}^{(1)}$. The while loop (lines 5–11) first computes the support for the current set of candidates at level k via the COMPUTESUPPORT procedure that generates k-subsets of each transaction in the database $\mathbf{D}$, and for each such subset it increments the support of the corresponding candidate in $\mathcal{C}^{(k)}$ if it exists. This way, the database is scanned only once per level, and the supports for all candidate k-itemsets are incremented during that scan. Next, we remove any infrequent candidate (line 9). The leaves of the prefix tree that survive comprise the set of frequent k-itemsets $\mathcal{F}^{(k)}$, which are used to generate the candidate $(k+1)$-itemsets for the next level (line 10). The EXTENDPREFIXTREE procedure employs prefix-based extension for candidate generation. Given two frequent k-itemsets X_a and X_b with a common $k-1$ length prefix, that is, given two sibling leaf nodes with a common parent, we generate the $(k+1)$-length candidate $X_{ab} = X_a \cup X_b$. This candidate is retained only if it has no infrequent subset. Finally, if a k-itemset X_a has no extension, it is pruned from the prefix tree, and we recursively

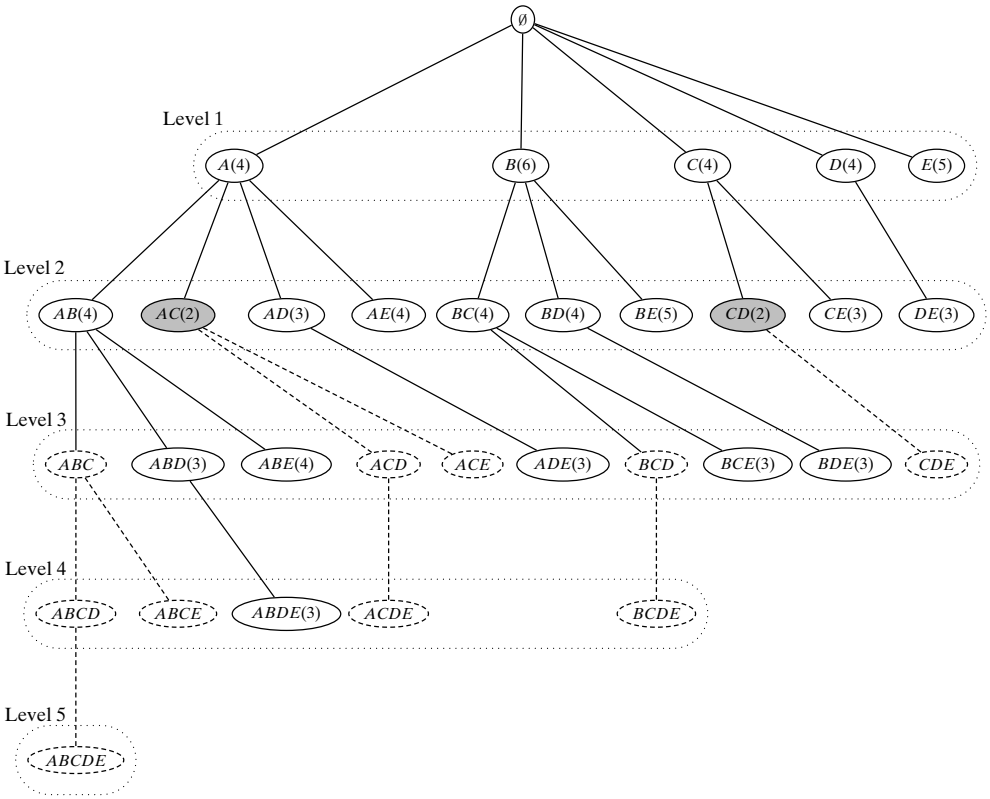


Figure 8.3. Apriori: prefix search tree and effect of pruning. Shaded nodes indicate infrequent itemsets, whereas dashed nodes and lines indicate all of the pruned nodes and branches. Solid lines indicate frequent itemsets.

prune any of its ancestors with no k-itemset extension, so that in $\mathcal{C}^{(k)}$ all leaves are at level k. If new candidates were added, the whole process is repeated for the next level. This process continues until no new candidates are added.

Example 8.7. Figure 8.4 illustrates the Apriori algorithm on the example dataset from Figure 8.1 using $minsup = 3$. All the candidates $\mathcal{C}^{(1)}$ are frequent (see Figure 8.4a). During extension all the pairwise combinations will be considered, since they all share the empty prefix $\emptyset$ as their parent. These comprise the new prefix tree $\mathcal{C}^{(2)}$ in Figure 8.4b; because E has no prefix-based extensions, it is removed from the tree. After support computation $AC(2)$ and $CD(2)$ are eliminated (shown in gray) since they are infrequent. The next level prefix tree is shown in Figure 8.4c. The candidate BCD is pruned due to the presence of the infrequent subset CD. All of the candidates at level 3 are frequent. Finally, $\mathcal{C}^{(4)}$ (shown in Figure 8.4d) has only one candidate $X_{ab} = ABDE$, which is generated from $X_a = ABD$ and $X_b = ABE$ because this is the only pair of siblings. The mining process stops after this step, since no more extensions are possible.

The worst-case computational complexity of the Apriori algorithm is still $O(|\mathcal{I}| \cdot |\mathbf{D}| \cdot 2^{|\mathcal{I}|})$, as all itemsets may be frequent. In practice, due to the pruning of the search

ALGORITHM 8.2. Algorithm APRIORI

```
APRIORI (D, I, minsup):
1  F ← ∅
2  C^(1) ← {∅} // Initial prefix tree with single items
3  foreach i ∈ I do Add i as child of ∅ in C^(1) with sup(i) ← 0
4  k ← 1 // k denotes the level
5  while C^(k) ≠ ∅ do
6      COMPUTESUPPORT (C^(k), D)
7      foreach leaf X ∈ C^(k) do
8          if sup(X) ≥ minsup then F ← F ∪ {(X, sup(X))}
9          else remove X from C^(k)
10     C^(k+1) ← EXTENDPREFIXTREE (C^(k))
11     k ← k + 1
12 return F^(k)

COMPUTESUPPORT (C^(k), D):
13 foreach ⟨t, i(t)⟩ ∈ D do
14     foreach k-subset X ⊆ i(t) do
15         if X ∈ C^(k) then sup(X) ← sup(X) + 1

EXTENDPREFIXTREE (C^(k)):
16 foreach leaf X_a ∈ C^(k) do
17     foreach leaf X_b ∈ SIBLING(X_a), such that b > a do
18         X_ab ← X_a ∪ X_b
           // prune candidate if there are any infrequent subsets
19         if X_j ∈ C^(k), for all X_j ⊂ X_ab, such that |X_j| = |X_ab| − 1 then
20             Add X_ab as child of X_a with sup(X_ab) ← 0
21     if no extensions from X_a then
22         remove X_a, and all ancestors of X_a with no extensions, from C^(k)
23 return C^(k)
```

space the cost is much lower. However, in terms of I/O cost Apriori requires $O(|\mathcal{I}|)$ database scans, as opposed to the $O(2^{|\mathcal{I}|})$ scans in the brute-force method. In practice, it requires only l database scans, where l is the length of the longest frequent itemset.

8.2.2 Tidset Intersection Approach: Eclat Algorithm

The support counting step can be improved significantly if we can index the database in such a way that it allows fast frequency computations. Notice that in the level-wise approach, to count the support, we have to generate subsets of each transaction and check whether they exist in the prefix tree. This can be expensive because we may end up generating many subsets that do not exist in the prefix tree.

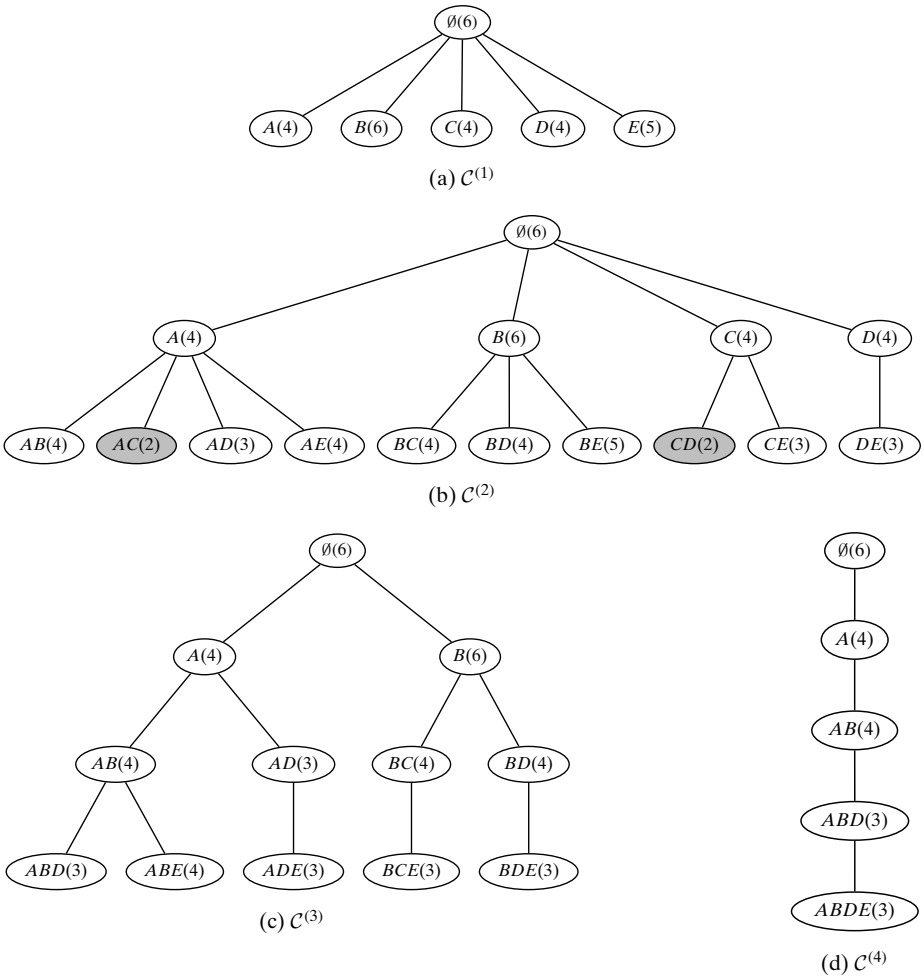


Figure 8.4. Itemset mining: Apriori algorithm. The prefix search trees $\mathcal{C}^{(k)}$ at each level are shown. Leaves (unshaded) comprise the set of frequent k-itemsets $\mathcal{F}^{(k)}$.

The Eclat algorithm leverages the tidsets directly for support computation. The basic idea is that the support of a candidate itemset can be computed by intersecting the tidsets of suitably chosen subsets. In general, given $\mathbf{t}(X)$ and $\mathbf{t}(Y)$ for any two frequent itemsets X and Y, we have

$$\mathbf{t}(XY) = \mathbf{t}(X) \cap \mathbf{t}(Y)$$

The support of candidate XY is simply the cardinality of $\mathbf{t}(XY)$, that is, $sup(XY) = |\mathbf{t}(XY)|$. Eclat intersects the tidsets only if the frequent itemsets share a common prefix, and it traverses the prefix search tree in a DFS-like manner, processing a group of itemsets that have the same prefix, also called a *prefix equivalence class*.

Example 8.8. For example, if we know that the tidsets for item A and C are $\mathbf{t}(A) = 1345$ and $\mathbf{t}(C) = 2456$, respectively, then we can determine the support of AC by intersecting the two tidsets, to obtain $\mathbf{t}(AC) = \mathbf{t}(A) \cap \mathbf{t}(C) = 1345 \cap 2456 = 45$.

ALGORITHM 8.3. Algorithm ECLAT

```
// Initial Call: F ← ∅, P ← {⟨i, t(i)⟩ | i ∈ I, |t(i)| ≥ minsup}
ECLAT (P, minsup, F):
1 foreach ⟨X_a, t(X_a)⟩ ∈ P do
2     F ← F ∪ {(X_a, sup(X_a))}
3     P_a ← ∅
4     foreach ⟨X_b, t(X_b)⟩ ∈ P, with X_b > X_a do
5         X_ab = X_a ∪ X_b
6         t(X_ab) = t(X_a) ∩ t(X_b)
7         if sup(X_ab) ≥ minsup then
8             P_a ← P_a ∪ {⟨X_ab, t(X_ab)⟩}
9     if P_a ≠ ∅ then ECLAT (P_a, minsup, F)
```

In this case, we have $sup(AC) = |45| = 2$. An example of a prefix equivalence class is the set $P_A = \{AB, AC, AD, AE\}$, as all the elements of P_A share A as the prefix.

The pseudo-code for Eclat is given in Algorithm 8.3. It employs a vertical representation of the binary database **D**. Thus, the input is the set of tuples $\langle i, \mathbf{t}(i)\rangle$ for all frequent items $i \in \mathcal{I}$, which comprise an equivalence class P (they all share the empty prefix); it is assumed that P contains only frequent itemsets. In general, given a prefix equivalence class P, for each frequent itemset $X_a \in P$, we try to intersect its tidset with the tidsets of all other itemsets $X_b \in P$. The candidate pattern is $X_{ab} = X_a \cup X_b$, and we check the cardinality of the intersection $\mathbf{t}(X_a) \cap \mathbf{t}(X_b)$ to determine whether it is frequent. If so, X_{ab} is added to the new equivalence class P_a that contains all itemsets that share X_a as a prefix. A recursive call to Eclat then finds all extensions of the X_a branch in the search tree. This process continues until no extensions are possible over all branches.

Example 8.9. Figure 8.5 illustrates the Eclat algorithm. Here $minsup = 3$, and the initial prefix equivalence class is

$$P_\emptyset = \{\langle A, 1345\rangle, \langle B, 123456\rangle, \langle C, 2456\rangle, \langle D, 1356\rangle, \langle E, 12345\rangle\}$$

Eclat intersects $\mathbf{t}(A)$ with each of $\mathbf{t}(B)$, $\mathbf{t}(C)$, $\mathbf{t}(D)$, and $\mathbf{t}(E)$ to obtain the tidsets for AB, AC, AD and AE, respectively. Out of these AC is infrequent and is pruned (marked gray). The frequent itemsets and their tidsets comprise the new prefix equivalence class

$$P_A = \{\langle AB, 1345\rangle, \langle AD, 135\rangle, \langle AE, 1345\rangle\}$$

which is recursively processed. On return, Eclat intersects $\mathbf{t}(B)$ with $\mathbf{t}(C)$, $\mathbf{t}(D)$, and $\mathbf{t}(E)$ to obtain the equivalence class

$$P_B = \{\langle BC, 2456\rangle, \langle BD, 1356\rangle, \langle BE, 12345\rangle\}$$

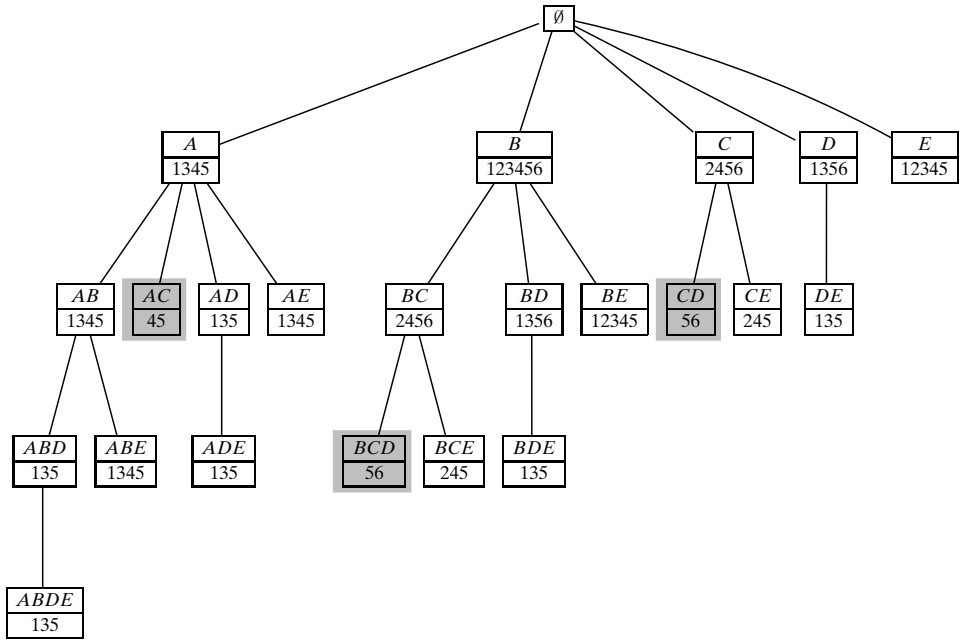


Figure 8.5. Eclat algorithm: tidlist intersections (gray boxes indicate infrequent itemsets).

Other branches are processed in a similar manner; the entire search space that Eclat explores is shown in Figure 8.5. The gray nodes indicate infrequent itemsets, whereas the rest constitute the set of frequent itemsets.

The computational complexity of Eclat is $O(|\mathbf{D}| \cdot 2^{|\mathcal{I}|})$ in the worst case, since there can be $2^{|\mathcal{I}|}$ frequent itemsets, and an intersection of two tidsets takes at most $O(|\mathbf{D}|)$ time. The I/O complexity of Eclat is harder to characterize, as it depends on the size of the intermediate tidsets. With t as the average tidset size, the initial database size is $O(t \cdot |\mathcal{I}|)$, and the total size of all the intermediate tidsets is $O(t \cdot 2^{|\mathcal{I}|})$. Thus, Eclat requires $\frac{t \cdot 2^{|\mathcal{I}|}}{t \cdot |\mathcal{I}|} = O(2^{|\mathcal{I}|}/|\mathcal{I}|)$ database scans in the worst case.

Diffsets: Difference of Tidsets

The Eclat algorithm can be significantly improved if we can shrink the size of the intermediate tidsets. This can be achieved by keeping track of the differences in the tidsets as opposed to the full tidsets. Formally, let $X_k = \{x_1, x_2, \ldots, x_{k-1}, x_k\}$ be a k-itemset. Define the *diffset* of X_k as the set of tids that contain the prefix $X_{k-1} = \{x_1, \ldots, x_{k-1}\}$ but do not contain the item x_k, given as

$$\mathbf{d}(X_k) = \mathbf{t}(X_{k-1}) \setminus \mathbf{t}(X_k)$$

Consider two k-itemsets $X_a = \{x_1, \ldots, x_{k-1}, x_a\}$ and $X_b = \{x_1, \ldots, x_{k-1}, x_b\}$ that share the common $(k-1)$-itemset $X = \{x_1, x_2, \ldots, x_{k-1}\}$ as a prefix. The diffset of $X_{ab} = X_a \cup X_b = \{x_1, \ldots, x_{k-1}, x_a, x_b\}$ is given as

$$\mathbf{d}(X_{ab}) = \mathbf{t}(X_a) \setminus \mathbf{t}(X_{ab}) = \mathbf{t}(X_a) \setminus \mathbf{t}(X_b) \tag{8.3}$$

However, note that

$$\mathbf{t}(X_a) \setminus \mathbf{t}(X_b) = \mathbf{t}(X_a) \cap \overline{\mathbf{t}(X_b)}$$

and taking the union of the above with the emptyset $\mathbf{t}(X) \cap \overline{\mathbf{t}(X)}$, we can obtain an expression for $\mathbf{d}(X_{ab})$ in terms of $\mathbf{d}(X_a)$ and $\mathbf{d}(X_b)$ as follows:

$$\begin{aligned}
\mathbf{d}(X_{ab}) &= \mathbf{t}(X_a) \setminus \mathbf{t}(X_b) \\
&= \mathbf{t}(X_a) \cap \overline{\mathbf{t}(X_b)} \\
&= \big(\mathbf{t}(X_a) \cap \overline{\mathbf{t}(X_b)}\big) \cup \big(\mathbf{t}(X) \cap \overline{\mathbf{t}(X)}\big) \\
&= \Big(\big(\mathbf{t}(X_a) \cup \mathbf{t}(X)\big) \cap \big(\overline{\mathbf{t}(X_b)} \cup \overline{\mathbf{t}(X)}\big)\Big) \cap \big(\mathbf{t}(X_a) \cup \overline{\mathbf{t}(X)}\big) \cap \big(\overline{\mathbf{t}(X_b)} \cup \mathbf{t}(X)\big) \\
&= \big(\mathbf{t}(X) \cap \overline{\mathbf{t}(X_b)}\big) \cap \overline{\big(\mathbf{t}(X) \cap \overline{\mathbf{t}(X_a)}\big)} \cap \mathcal{T} \\
&= \mathbf{d}(X_b) \setminus \mathbf{d}(X_a)
\end{aligned}$$

Thus, the diffset of X_{ab} can be obtained from the diffsets of its subsets X_a and X_b, which means that we can replace all intersection operations in Eclat with diffset operations. Using diffsets the support of a candidate itemset can be obtained by subtracting the diffset size from the support of the prefix itemset:

$$sup(X_{ab}) = sup(X_a) - |\mathbf{d}(X_{ab})|$$

which follows directly from Eq. (8.3).

The variant of Eclat that uses the diffset optimization is called dEclat, whose pseudo-code is shown in Algorithm 8.4. The input comprises all the frequent single items $i \in \mathcal{I}$ along with their diffsets, which are computed as

$$\mathbf{d}(i) = \mathbf{t}(\emptyset) \setminus \mathbf{t}(i) = \mathcal{T} \setminus \mathbf{t}(i)$$

Given an equivalence class P, for each pair of distinct itemsets X_a and X_b we generate the candidate pattern $X_{ab} = X_a \cup X_b$ and check whether it is frequent via the use of diffsets (lines 6–7). Recursive calls are made to find further extensions. It is important

ALGORITHM 8.4. Algorithm dECLAT

```
// Initial Call: F ← ∅,
    P ← {⟨i, d(i), sup(i)⟩ | i ∈ I, d(i) = T \ t(i), sup(i) ≥ minsup}
dECLAT (P, minsup, F):
1  foreach ⟨X_a, d(X_a), sup(X_a)⟩ ∈ P do
2      F ← F ∪ {(X_a, sup(X_a))}
3      P_a ← ∅
4      foreach ⟨X_b, d(X_b), sup(X_b)⟩ ∈ P, with X_b > X_a do
5          X_ab = X_a ∪ X_b
6          d(X_ab) = d(X_b) \ d(X_a)
7          sup(X_ab) = sup(X_a) − |d(X_ab)|
8          if sup(X_ab) ≥ minsup then
9              P_a ← P_a ∪ {⟨X_ab, d(X_ab), sup(X_ab)⟩}
10     if P_a ≠ ∅ then dECLAT (P_a, minsup, F)
```

to note that the switch from tidsets to diffsets can be made during any recursive call to the method. In particular, if the initial tidsets have small cardinality, then the initial call should use tidset intersections, with a switch to diffsets starting with 2-itemsets. Such optimizations are not described in the pseudo-code for clarity.

Example 8.10. Figure 8.6 illustrates the dEclat algorithm. Here $minsup = 3$, and the initial prefix equivalence class comprises all frequent items and their diffsets, computed as follows:

$$\mathbf{d}(A) = \mathcal{T} \setminus 1345 = 26$$
$$\mathbf{d}(B) = \mathcal{T} \setminus 123456 = \emptyset$$
$$\mathbf{d}(C) = \mathcal{T} \setminus 2456 = 13$$
$$\mathbf{d}(D) = \mathcal{T} \setminus 1356 = 24$$
$$\mathbf{d}(E) = \mathcal{T} \setminus 12345 = 6$$

where $\mathcal{T} = 123456$. To process candidates with A as a prefix, dEclat computes the diffsets for AB, AC, AD and AE. For instance, the diffsets of AB and AC are given as

$$\mathbf{d}(AB) = \mathbf{d}(B) \setminus \mathbf{d}(A) = \emptyset \setminus \{2, 6\} = \emptyset$$
$$\mathbf{d}(AC) = \mathbf{d}(C) \setminus \mathbf{d}(A) = \{1, 3\} \setminus \{2, 6\} = 13$$

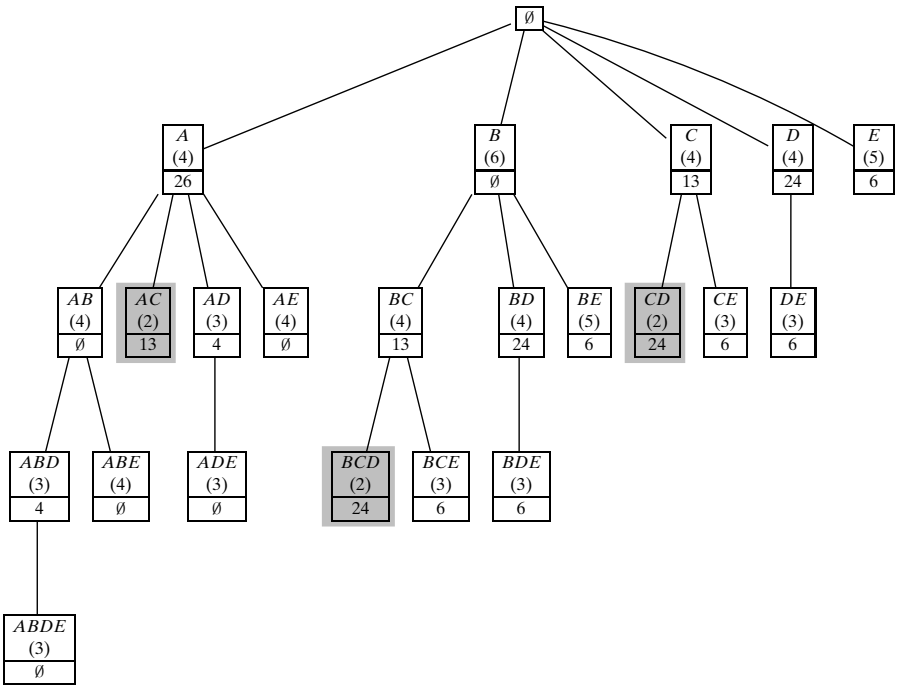


Figure 8.6. dEclat algorithm: diffsets (gray boxes indicate infrequent itemsets).

and their support values are

$$sup(AB) = sup(A) - |\mathbf{d}(AB)| = 4 - 0 = 4$$
$$sup(AC) = sup(A) - |\mathbf{d}(AC)| = 4 - 2 = 2$$

Whereas AB is frequent, we can prune AC because it is not frequent. The frequent itemsets and their diffsets and support values comprise the new prefix equivalence class:

$$P_A = \{\langle AB, \emptyset, 4\rangle, \langle AD, 4, 3\rangle, \langle AE, \emptyset, 4\rangle\}$$

which is recursively processed. Other branches are processed in a similar manner. The entire search space for dEclat is shown in Figure 8.6. The support of an itemset is shown within brackets. For example, A has support 4 and diffset $\mathbf{d}(A) = 26$.

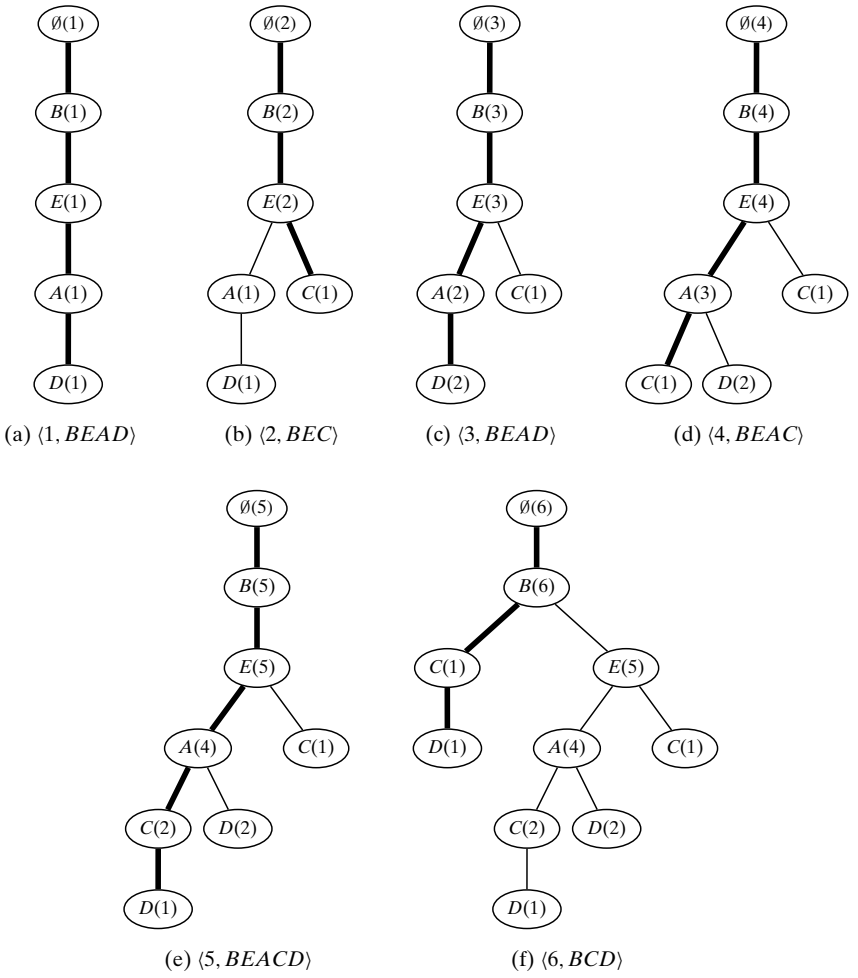
8.2.3 Frequent Pattern Tree Approach: FPGrowth Algorithm

The FPGrowth method indexes the database for fast support computation via the use of an augmented prefix tree called the *frequent pattern tree* (FP-tree). Each node in the tree is labeled with a single item, and each child node represents a different item. Each node also stores the support information for the itemset comprising the items on the path from the root to that node. The FP-tree is constructed as follows. Initially the tree contains as root the null item $\emptyset$. Next, for each tuple $\langle t, X\rangle \in \mathbf{D}$, where $X = \mathbf{i}(t)$, we insert the itemset X into the FP-tree, incrementing the count of all nodes along the path that represents X. If X shares a prefix with some previously inserted transaction, then X will follow the same path until the common prefix. For the remaining items in X, new nodes are created under the common prefix, with counts initialized to 1. The FP-tree is complete when all transactions have been inserted.

The FP-tree can be considered as a prefix compressed representation of $\mathbf{D}$. Because we want the tree to be as compact as possible, we want the most frequent items to be at the top of the tree. FPGrowth therefore reorders the items in decreasing order of support, that is, from the initial database, it first computes the support of all single items $i \in \mathcal{I}$. Next, it discards the infrequent items, and sorts the frequent items by decreasing support. Finally, each tuple $\langle t, X\rangle \in \mathbf{D}$ is inserted into the FP-tree after reordering X by decreasing item support.

Example 8.11. Consider the example database in Figure 8.1. We add each transaction one by one into the FP-tree, and keep track of the count at each node. For our example database the sorted item order is $\{B(6), E(5), A(4), C(4), D(4)\}$. Next, each transaction is reordered in this same order; for example, $\langle 1, ABDE\rangle$ becomes $\langle 1, BEAD\rangle$. Figure 8.7 illustrates step-by-step FP-tree construction as each sorted transaction is added to it. The final FP-tree for the database is shown in Figure 8.7f.

Once the FP-tree has been constructed, it serves as an index in lieu of the original database. All frequent itemsets can be mined from the tree directly via the FPGROWTH method, whose pseudo-code is shown in Algorithm 8.5. The method accepts as input a FP-tree R constructed from the input database $\mathbf{D}$, and the current itemset prefix P, which is initially empty.



(a) $\langle 1, BEAD \rangle$ (b) $\langle 2, BEC \rangle$ (c) $\langle 3, BEAD \rangle$ (d) $\langle 4, BEAC \rangle$

(e) $\langle 5, BEACD \rangle$ (f) $\langle 6, BCD \rangle$

Figure 8.7. Frequent pattern tree: bold edges indicate current transaction.

Given a FP-tree R, projected FP-trees are built for each frequent item i in R in increasing order of support. To project R on item i, we find all the occurrences of i in the tree, and for each occurrence, we determine the corresponding path from the root to i (line 13). The count of item i on a given path is recorded in $cnt(i)$ (line 14), and the path is inserted into the new projected tree R_X, where X is the itemset obtained by extending the prefix P with the item i. While inserting the path, the count of each node in R_X along the given path is incremented by the path count $cnt(i)$. We omit the item i from the path, as it is now part of the prefix. The resulting FP-tree is a projection of the itemset X that comprises the current prefix extended with item i (line 9). We then call FPGROWTH recursively with projected FP-tree R_X and the new prefix itemset X as the parameters (line 16). The base case for the recursion happens when the input FP-tree R is a single path. FP-trees that are paths are handled by enumerating all itemsets that are subsets of the path, with the support of each such itemset being given by the least frequent item in it (lines 2–6).

ALGORITHM 8.5. Algorithm FPGROWTH

```
// Initial Call: R ← FP-tree(D), P ← ∅, F ← ∅
FPGROWTH (R, P, F, minsup):
1  Remove infrequent items from R
2  if ISPATH(R) then // insert subsets of R into F
3      foreach Y ⊆ R do
4          X ← P ∪ Y
5          sup(X) ← min_{x∈Y}{cnt(x)}
6          F ← F ∪ {(X, sup(X))}
7  else  // process projected FP-trees for each frequent item i
8      foreach i ∈ R in increasing order of sup(i) do
9          X ← P ∪ {i}
10         sup(X) ← sup(i) // sum of cnt(i) for all nodes labeled i
11         F ← F ∪ {(X, sup(X))}
12         R_X ← ∅ // projected FP-tree for X
13         foreach path ∈ PATHFROMROOT(i) do
14             cnt(i) ← count of i in path
15             Insert path, excluding i, into FP-tree R_X with count cnt(i)
16         if R_X ≠ ∅ then FPGROWTH (R_X, X, F, minsup)
```

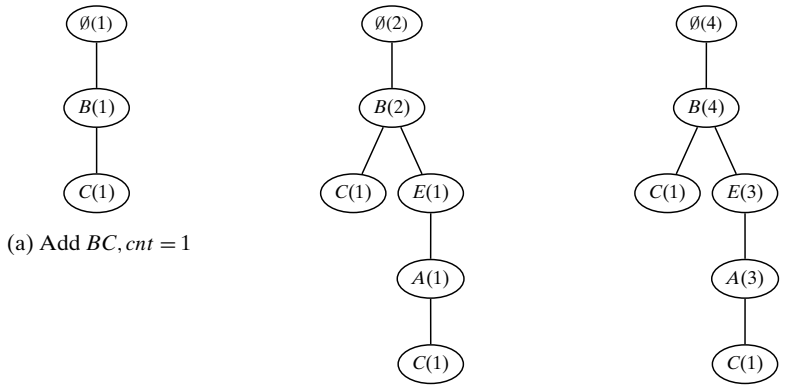
Example 8.12. We illustrate the FPGrowth method on the FP-tree R built in Example 8.11, as shown in Figure 8.7f. Let $minsup = 3$. The initial prefix is $P = \emptyset$, and the set of frequent items i in R are $B(6)$, $E(5)$, $A(4)$, $C(4)$, and $D(4)$. FPGrowth creates a projected FP-tree for each item, but in increasing order of support.

The projected FP-tree for item D is shown in Figure 8.8c. Given the initial FP-tree R shown in Figure 8.7f, there are three paths from the root to a node labeled D, namely

$$\begin{aligned} BCD, &\quad cnt(D) = 1 \\ BEACD, &\quad cnt(D) = 1 \\ BEAD, &\quad cnt(D) = 2 \end{aligned}$$

These three paths, excluding the last item $i = D$, are inserted into the new FP-tree R_D with the counts incremented by the corresponding $cnt(D)$ values, that is, we insert into R_D, the paths BC with count of 1, $BEAC$ with count of 1, and finally BEA with count of 2, as shown in Figures 8.8a–c. The projected FP-tree for D is shown in Figure 8.8c, which is processed recursively.

When we process R_D, we have the prefix itemset $P = D$, and after removing the infrequent item C (which has support 2), we find that the resulting FP-tree is a single path $B(4)$–$E(3)$–$A(3)$. Thus, we enumerate all subsets of this path and prefix them



(a) Add $BC, cnt = 1$

(b) Add $BEAC, cnt = 1$

(c) Add $BEA, cnt = 2$

Figure 8.8. Projected frequent pattern tree for D.

with D, to obtain the frequent itemsets $DB(4)$, $DE(3)$, $DA(3)$, $DBE(3)$, $DBA(3)$, $DEA(3)$, and $DBEA(3)$. At this point the call from D returns.

In a similar manner, we process the remaining items at the top level. The projected trees for C, A, and E are all single-path trees, allowing us to generate the frequent itemsets $\{CB(4), CE(3), CBE(3)\}$, $\{AE(4), AB(4), AEB(4)\}$, and $\{EB(5)\}$, respectively. This process is illustrated in Figure 8.9.

8.3 GENERATING ASSOCIATION RULES

Given a collection of frequent itemsets $\mathcal{F}$, to generate association rules we iterate over all itemsets $Z \in \mathcal{F}$, and calculate the confidence of various rules that can be derived from the itemset. Formally, given a frequent itemset $Z \in \mathcal{F}$, we look at all proper subsets $X \subset Z$ to compute rules of the form

$$X \xrightarrow{s,c} Y, \text{ where } Y = Z \setminus X$$

where $Z \setminus X = Z - X$. The rule must be frequent because

$$s = sup(XY) = sup(Z) \geq minsup$$

Thus, we have to only check whether the rule confidence satisfies the *minconf* threshold. We compute the confidence as follows:

$$c = \frac{sup(X \cup Y)}{sup(X)} = \frac{sup(Z)}{sup(X)}$$

If $c \geq minconf$, then the rule is a strong rule. On the other hand, if $conf(X \longrightarrow Y) < c$, then $conf(W \longrightarrow Z \setminus W) < c$ for all subsets $W \subset X$, as $sup(W) \geq sup(X)$. We can thus avoid checking subsets of X.

Algorithm 8.6 shows the pseudo-code for the association rule mining algorithm. For each frequent itemset $Z \in \mathcal{F}$, with size at least 2, we initialize the set of antecedents

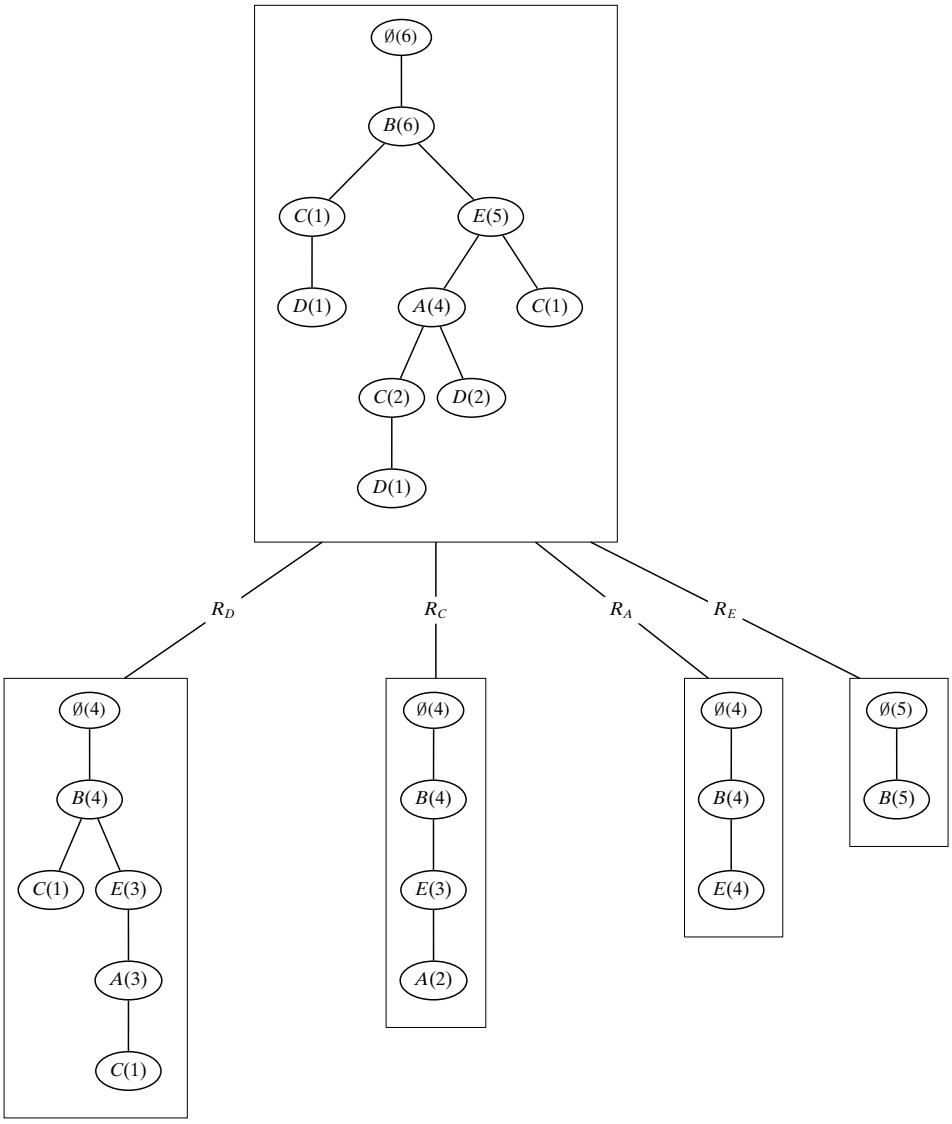


Figure 8.9. FPGrowth algorithm: frequent pattern tree projection.

$\mathcal{A}$ with all the nonempty subsets of Z (line 2). For each $X \in \mathcal{A}$ we check whether the confidence of the rule $X \longrightarrow Z \setminus X$ is at least *minconf* (line 7). If so, we output the rule. Otherwise, we remove all subsets $W \subset X$ from the set of possible antecedents (line 10).

Example 8.13. Consider the frequent itemset $ABDE(3)$ from Table 8.1, whose support is shown within the brackets. Assume that $minconf = 0.9$. To generate strong association rules we initialize the set of antecedents to

$$\mathcal{A} = \{ABD(3), ABE(4), ADE(3), BDE(3), AB(3), AD(4), AE(4),$$
$$BD(4), BE(5), DE(3), A(4), B(6), D(4), E(5)\}$$

ALGORITHM 8.6. Algorithm AssociationRules

```
AssociationRules (F, minconf):
1  foreach Z ∈ F, such that |Z| ≥ 2 do
2      A ← {X | X ⊂ Z, X ≠ ∅}
3      while A ≠ ∅ do
4          X ← maximal element in A
5          A ← A \ X// remove X from A
6          c ← sup(Z)/sup(X)
7          if c ≥ minconf then
8              print X ⟶ Y, sup(Z), c
9          else
10             A ← A \ {W | W ⊂ X} // remove all subsets of X from A
```

The first subset is $X = ABD$, and the confidence of $ABD \longrightarrow E$ is $3/3 = 1.0$, so we output it. The next subset is $X = ABE$, but the corresponding rule $ABE \longrightarrow D$ is not strong since $conf(ABE \longrightarrow D) = 3/4 = 0.75$. We can thus remove from $\mathcal{A}$ all subsets of ABE; the updated set of antecedents is therefore

$$\mathcal{A} = \{ADE(3), BDE(3), AD(4), BD(4), DE(3), D(4)\}$$

Next, we select $X = ADE$, which yields a strong rule, and so do $X = BDE$ and $X = AD$. However, when we process $X = BD$, we find that $conf(BD \longrightarrow AE) = 3/4 = 0.75$, and thus we can prune all subsets of BD from $\mathcal{A}$, to yield

$$\mathcal{A} = \{DE(3)\}$$

The last rule to be tried is $DE \longrightarrow AB$ which is also strong. The final set of strong rules that are output are as follows:

$$ABD \longrightarrow E, conf = 1.0$$
$$ADE \longrightarrow B, conf = 1.0$$
$$BDE \longrightarrow A, conf = 1.0$$
$$AD \longrightarrow BE, conf = 1.0$$
$$DE \longrightarrow AB, conf = 1.0$$

8.4 FURTHER READING

The association rule mining problem was introduced in Agrawal, Imieliński, and Swami (1993). The Apriori method was proposed in Agrawal and Srikant (1994), and a similar approach was outlined independently in Mannila, Toivonen, and Verkamo

(1994). The tidlist intersection based Eclat method is described in Zaki et al. (1997), and the dEclat approach that uses diffset appears in Zaki and Gouda (2003). Finally, the FPGrowth algorithm is described in Han, Pei, and Yin (2000). For an experimental comparison between several of the frequent itemset mining algorithms see Goethals and Zaki (2004). There is a very close connection between itemset mining and association rules, and formal concept analysis (Ganter, Wille, and Franzke, 1997). For example, association rules can be considered to be *partial implications* (Luxenburger, 1991) with frequency constraints.

Agrawal, R., Imieliński, T., and Swami, A. (May 1993). "Mining association rules between sets of items in large databases." *In Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM.

Agrawal, R. and Srikant, R. (Sept. 1994). "Fast algorithms for mining association rules." *In Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 487–499.

Ganter, B., Wille, R., and Franzke, C. (1997). *Formal Concept Analysis: Mathematical Foundations.* New York: Springer-Verlag.

Goethals, B. and Zaki, M. J. (2004). "Advances in frequent itemset mining implementations: report on FIMI'03." *ACM SIGKDD Explorations*, 6 (1): 109–117.

Han, J., Pei, J., and Yin, Y. (May 2000). "Mining frequent patterns without candidate generation." *In Proceedings of the ACM SIGMOD International Conference on Management of Data,* ACM.

Luxenburger, M. (1991). "Implications partielles dans un contexte." *Mathématiques et Sciences Humaines*, 113: 35–55.

Mannila, H., Toivonen, H., and Verkamo, I. A. (1994). Efficient algorithms for discovering association rules. *In Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, AAAI Press.

Zaki, M. J. and Gouda, K. (2003). "Fast vertical mining using diffsets." *In Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, pp. 326–335.

Zaki, M. J., Parthasarathy, S., Ogihara, M., and Li, W. (1997). "New algorithms for fast discovery of association rules." *In Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pp. 283–286.

8.5 EXERCISES

Q1. Given the database in Table 8.2.

- **(a)** Using $minsup = 3/8$, show how the Apriori algorithm enumerates all frequent patterns from this dataset.
- **(b)** With $minsup = 2/8$, show how FPGrowth enumerates the frequent itemsets.

Q2. Consider the vertical database shown in Table 8.3. Assuming that $minsup = 3$, enumerate all the frequent itemsets using the Eclat method.

Table 8.2. Transaction database for Q1

tid	itemset
t_1	$ABCD$
t_2	$ACDF$
t_3	$ACDEG$
t_4	$ABDF$
t_5	BCG
t_6	DFG
t_7	ABG
t_8	$CDFG$

Table 8.3. Dataset for Q2

A	B	C	D	E
1	2	1	1	2
3	3	2	6	3
5	4	3		4
6	5	5		5
	6	6		

Q3. Given two k-itemsets $X_a = \{x_1, \ldots, x_{k-1}, x_a\}$ and $X_b = \{x_1, \ldots, x_{k-1}, x_b\}$ that share the common $(k-1)$-itemset $X = \{x_1, x_2, \ldots, x_{k-1}\}$ as a prefix, prove that

$$sup(X_{ab}) = sup(X_a) - |\mathbf{d}(X_{ab})|$$

where $X_{ab} = X_a \cup X_b$, and $\mathbf{d}(X_{ab})$ is the diffset of X_{ab}.

Q4. Given the database in Table 8.4. Show all rules that one can generate from the set ABE.

Table 8.4. Dataset for Q4

tid	itemset
t_1	ACD
t_2	BCE
t_3	$ABCE$
t_4	BDE
t_5	$ABCE$
t_6	$ABCD$

Q5. Consider the *partition* algorithm for itemset mining. It divides the database into k partitions, not necessarily equal, such that $\mathbf{D} = \cup_{i=1}^{k} \mathbf{D}_i$, where $\mathbf{D}_i$ is partition i, and for any $i \neq j$, we have $\mathbf{D}_i \cap \mathbf{D}_j = \emptyset$. Also let $n_i = |\mathbf{D}_i|$ denote the number of transactions in partition $\mathbf{D}_i$. The algorithm first mines only locally frequent itemsets, that is, itemsets whose relative support is above the *minsup* threshold specified as a fraction. In the second step, it takes the union of all locally frequent itemsets, and computes their support in the entire database $\mathbf{D}$ to determine which of them are globally frequent. Prove that if a pattern is globally frequent in the database, then it must be locally frequent in at least one partition.

Q6. Consider Figure 8.10. It shows a simple taxonomy on some food items. Each leaf is a simple item and an internal node represents a higher-level category or item. Each item (single or high-level) has a unique integer label noted under it. Consider the database composed of the simple items shown in Table 8.5 Answer the following questions:

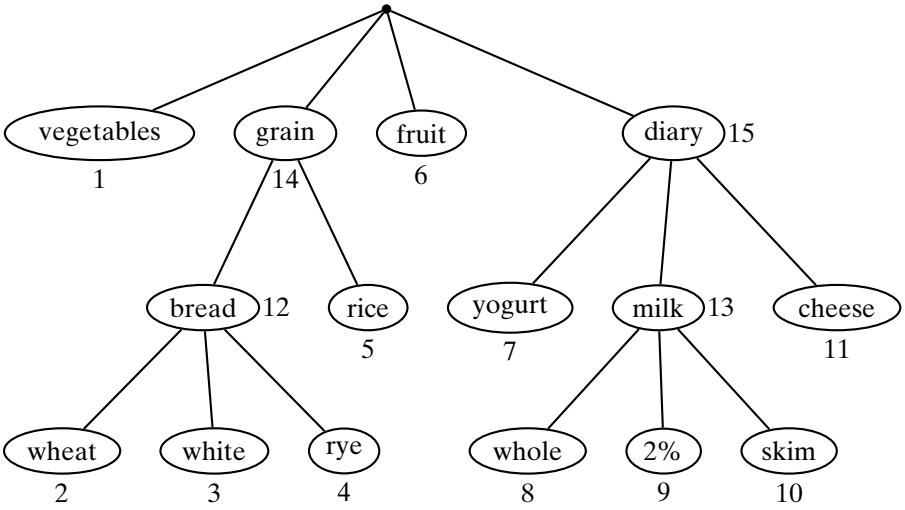


Figure 8.10. Item taxonomy for Q6.

Table 8.5. Dataset for Q6

tid	itemset
1	2 3 6 7
2	1 3 4 8 11
3	3 9 11
4	1 5 6 7
5	1 3 8 10 11
6	3 5 7 9 11
7	4 6 8 10 11
8	1 3 5 8 11

(a) What is the size of the itemset search space if one restricts oneself to only itemsets composed of simple items?

(b) Let $X = \{x_1, x_2, \ldots, x_k\}$ be a frequent itemset. Let us replace some $x_i \in X$ with its parent in the taxonomy (provided it exists) to obtain X', then the support of the new itemset X' is:

- **i.** more than support of X
- **ii.** less than support of X
- **iii.** not equal to support of X
- **iv.** more than or equal to support of X
- **v.** less than or equal to support of X

(c) Use $minsup = 7/8$. Find all frequent itemsets composed only of high-level items in the taxonomy. Keep in mind that if a simple item appears in a transaction, then its high-level ancestors are all assumed to occur in the transaction as well.

Q7. Let $\mathbf{D}$ be a database with n transactions. Consider a sampling approach for mining frequent itemsets, where we extract a random sample $\mathbf{S} \subset \mathbf{D}$, with say m transactions, and we mine all the frequent itemsets in the sample, denoted as $\mathcal{F}_S$. Next, we make one complete scan of $\mathbf{D}$, and for each $X \in \mathcal{F}_S$, we find its actual support in the whole database. Some of the itemsets in the sample may not be truly frequent in the database; these are the false positives. Also, some of the true frequent itemsets in the original database may never be present in the sample at all; these are the false negatives.

Prove that if X is a false negative, then this case can be detected by counting the support in $\mathbf{D}$ for every itemset belonging to the *negative border* of $\mathcal{F}_S$, denoted $Bd^-(\mathcal{F}_S)$, which is defined as the set of minimal infrequent itemsets in sample $\mathbf{S}$. Formally,

$$Bd^-(\mathcal{F}_S) = \inf\{Y \mid sup(Y) < minsup \text{ and } \forall Z \subset Y, sup(Z) \geq minsup\}$$

where inf returns the minimal elements of the set.

Q8. Assume that we want to mine frequent patterns from relational tables. For example consider Table 8.6, with three attributes A, B, and C, and six records. Each attribute has a domain from which it draws its values, for example, the domain of A is $dom(A) = \{a_1, a_2, a_3\}$. Note that no record can have more than one value of a given attribute.

Table 8.6. Data for Q8

tid	A	B	C
1	a_1	b_1	c_1
2	a_2	b_3	c_2
3	a_2	b_3	c_3
4	a_2	b_1	c_1
5	a_2	b_3	c_3
6	a_3	b_3	c_3

We define a *relational pattern* P over some k attributes $X_1, X_2, \ldots, X_k$ to be a subset of the Cartesian product of the domains of the attributes, i.e., $P \subseteq dom(X_1) \times dom(X_2) \times \cdots \times dom(X_k)$. That is, $P = P_1 \times P_2 \times \cdots \times P_k$, where each $P_i \subseteq dom(X_i)$. For example, $\{a_1, a_2\} \times \{c_1\}$ is a possible pattern over attributes A and C, whereas $\{a_1\} \times \{b_1\} \times \{c_1\}$ is another pattern over attributes A, B and C.

The support of relational pattern $P = P_1 \times P_2 \times \cdots \times P_k$ in dataset $\mathbf{D}$ is defined as the number of records in the dataset that belong to it; it is given as

$$sup(P) = \big|\{r = (r_1, r_2, \ldots, r_n) \in \mathbf{D} : r_i \in P_i \text{ for all } P_i \text{ in } P\}\big|$$

For example, $sup(\{a_1, a_2\} \times \{c_1\}) = 2$, as both records 1 and 4 contribute to its support. Note, however that the pattern $\{a_1\} \times \{c_1\}$ has a support of 1, since only record 1 belongs to it. Thus, relational patterns **do not** satisfy the Apriori property that we

used for frequent itemsets, that is, subsets of a frequent relational pattern can be infrequent.

We call a relational pattern $P = P_1 \times P_2 \times \cdots \times P_k$ over attributes $X_1, \ldots, X_k$ as *valid* iff for all $u \in P_i$ and all $v \in P_j$, the pair of values $(X_i = u, X_j = v)$ occurs together in some record. For example, $\{a_1, a_2\} \times \{c_1\}$ is a valid pattern since both $(A = a_1, C = c_1)$ and $(A = a_2, C = c_1)$ occur in some records (namely, records 1 and 4, respectively), whereas $\{a_1, a_2\} \times \{c_2\}$ is not a valid pattern, since there is no record that has the values $(A = a_1, C = c_2)$. Thus, for a pattern to be valid every pair of values in P from distinct attributes must belong to some record.

Given that $minsup = 2$, find all frequent, valid, relational patterns in the dataset in Table 8.6.

Q9. Given the following multiset dataset:

tid	multiset
1	$ABCA$
2	$ABABA$
3	$CABBA$

Using $minsup = 2$, answer the following:

(a) Find all frequent multisets. Recall that a multiset is still a set (i.e., order is not important), but it allows multiple occurrences of an item.

(b) Find all minimal infrequent multisets, that is, those multisets that have no infrequent sub-multisets.