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 web site, 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 web site. Such "frequent" sets of web pages give clues to the user browsing behavior and can be used for improving the user's browsing experience.

The quest to mine frequent patterns appears in many other domains. The prototypical application is *market basket analysis*, i.e., 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 like, "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.

8.1 Foundations

8.1.1 Database Representation

Let $\mathcal{I} = \{x_1, x_2, \cdots, x_m\}$ be a set of binary-valued attributes, which we call *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

web site, 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, i.e., subsets of $\mathcal{I}$ with size k. Let $\mathcal{T} = \{t_1, t_2, \cdots, t_n\}$ be a set of transaction identifiers, also called *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 web site, and so on. For convenience, we refer to a transaction $\langle t, X \rangle$ by its identifier t.

A binary database $\mathcal{D}$ is a binary relation on the set of items and tids, i.e., $\mathcal{D} \subseteq \mathcal{I} \times \mathcal{T}$. We say that tid $t \in \mathcal{T}$ *contains* item $x \in \mathcal{I}$ if and only if $(x, t) \in \mathcal{D}$. In other words, $(x, t) \in \mathcal{D}$ if and only if $x \in X$ in the tuple $\langle t, X \rangle$.

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 has a 1 if and only if $(x, t) \in \mathcal{D}$, and 0 otherwise.

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

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

That is, $\mathbf{i}(T)$ is the set of items that are contained in *all* the tids in the tidset T. In particular, $\mathbf{i}(t)$ is the set of items contained in tid $t \in \mathcal{T}$. It is sometimes convenient to consider the binary database $\mathcal{D}$, as a transaction database consisting of tuples of the form $\langle t, \mathbf{i}(t) \rangle$, with $t \in \mathcal{T}$. The transaction database can be considered as a "horizontal" projection of the binary database, where we omit any item that has a '0' for a given tid.

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

$$\mathbf{t}(X) = \{t \mid X \subseteq \mathbf{i}(t)\} \tag{8.2}$$

That is, $\mathbf{t}(X)$ is the set of transactions 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 $\mathcal{D}$, 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" projection of the binary database, where we omit any tid that has a '0' for 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

	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

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

(b) Transaction Database

	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

$\langle 1, \{A, B, D, E\}\rangle$. Henceforth, for convenience, we will drop set notations when writing itemsets and tidsets when there is no confusion. Thus we write $\langle 1, \{A, B, D, E\}\rangle$ as $\langle 1, ABDE\rangle$.

Figure 8.1c shows the corresponding vertical database for the binary database in Figure 8.1a. For instance, the tidset corresponding to item A is given in the first column as $\langle A, \{1, 3, 4, 5\}\rangle$, which we write as $\langle A, 1345\rangle$ for convenience.

8.1.2 Interestingness Measures

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

$$sup(X, \mathcal{D}) = |\{t_i \mid \langle t_i, \mathbf{i}(t_i)\rangle \in \mathcal{D} \text{ and } X \subseteq \mathbf{i}(t_i)\}| \tag{8.3}$$

It should be easy to see that the support of itemset X can also be defined in terms of the cardinality of its corresponding tidset $\mathbf{t}(X)$, as follows

$$sup(X, \mathcal{D}) = |\mathbf{t}(X)| \tag{8.4}$$

Note that the support of X gives an estimate of the *joint probability* of the items of X in the database, given as follows

$$P(X) = \frac{sup(X)}{|\mathcal{D}|} \tag{8.5}$$

X is said to be *frequent* in $\mathcal{D}$ if $sup(X, \mathcal{D}) \geq minsup$, where $minsup$ is a user defined minimum support threshold. When there is no confusion about $\mathcal{D}$ we write support simply as $sup(X)$. 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.

An *association rule* is an expression $A \xrightarrow{s,c} B$, where A and B are itemsets, and $A \cap B = \emptyset$. The *support* of the rule is derived from the joint probability of a transaction containing both A and B, given as

$$s = sup(A \implies B) = P(A \wedge B) \times |\mathcal{D}| = \frac{sup(A \cup B)}{|\mathcal{D}|} \times |\mathcal{D}| = sup(A \cup B) \quad (8.6)$$

The *confidence* of a rule is the conditional probability that a transaction contains B, given that it contains A, given as

$$c = conf(A \implies B) = P(B|A) = \frac{P(A \wedge B)}{P(A)} = \frac{\frac{sup(A \cup B)}{|\mathcal{D}|}}{\frac{sup(A)}{|\mathcal{D}|}} = \frac{sup(A \cup B)}{sup(A)} \quad (8.7)$$

A rule is frequent if the itemset $A \cup B$ is frequent. A rule is *confident* if $conf \geq minconf$, where $minconf$ is a user-specified minimum threshold. If the two itemsets A and B are independent, then $P(B|A) = P(B) \times P(A)$. Thus, once a rule's confidence is known it is useful to compare how much it deviates from the independence assumption, by computing the rule *lift*: $\frac{P(A \cup B)}{P(A)P(B)}$.

Given a transaction database $\mathcal{D}$ and the user defined minimum support threshold $minsup$, the task of frequent itemset mining is to enumerate all itemsets that are frequent.

Example 8.3: Given the example dataset in Figure 8.1, let $minsup = 3$. Figure 8.2 shows all the 19 frequent itemsets in the database, grouped by their support value. Other itemsets have support lower than 3 and are thus infrequent. For example, the itemset BCE is a subset of transactions 2, 4, and 5, i.e., $\mathbf{t}(BCE) = \{2, 4, 5\}$. Thus $sup(BCE) = |\mathbf{t}(BCE)| = 3$, and it is frequent.

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

Figure 8.2: Frequent Itemsets with $minsup = 3$

8.1.3 Generating Association Rules

Given a collection of frequent itemsets $\mathcal{F}$, it is simple to iterate over all itemsets $X \in \mathcal{F}$ calculating the confidence of various rules that can be derived from the

itemset. That is, for a given $X \in \mathcal{F}$, we look at all proper subsets $A \subset X$ to compute rules of the form

$$A \overset{s,c}{\Longrightarrow} X - A \tag{8.8}$$

We already know that the rule must be frequent, since

$$s = sup(A \cup (X - A)) = sup(X) \geq minsup \tag{8.9}$$

Thus, we only have to check whether the confidence of the rule is at least $minconf$. We compute the confidence as follows

$$c = \frac{sup(A \cup (X - A))}{sup(A)} = \frac{sup(X)}{sup(A)} \tag{8.10}$$

If $c \geq minconf$, then the minimum confidence threshold is met, we print the rule as a strong association rule.

8.2 Algorithms for Itemset Mining

8.2.1 Brute Force Approach

Algorithm 8.1: Algorithm BRUTEFORCE

```
BRUTEFORCE (D, I, minsup):
1 foreach X ⊆ I do
2     sup(X) ← COMPUTESUPPORT (X, D);
3     if sup(X) ≥ minsup then
4         print X, sup(X)

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

Consider Algorithm 8.1, that uses a brute force approach to mine all frequent itemsets. It enumerates all the possible subsets of $\mathcal{I}$, and for each such subset $X \subseteq \mathcal{I}$, the algorithm computes its support. A check against the $minsup$ threshold determines the frequent itemsets.

The algorithm illustrates the two main steps in mining frequent itemsets

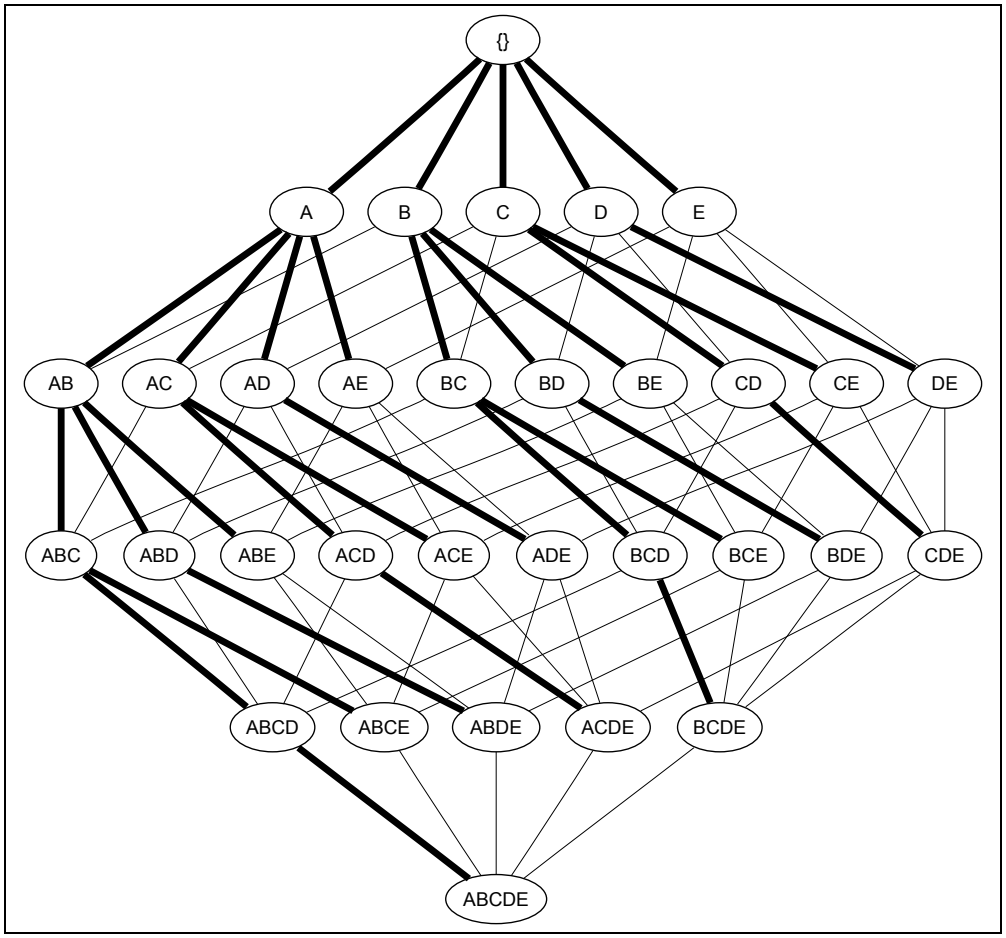

Figure 8.3: Itemset Lattice & Prefix-based Search Tree (in Bold)

Itemset Enumeration: This step generates all the subsets of $\mathcal{I}$, which are called *candidates*, since each itemset is potentially a candidate to be frequent. The candidate itemset search space is clearly exponential, since there are $2^{|\mathcal{I}|}$ potentially frequent itemsets. The computational complexity of enumerating all the itemsets is thus $O(2^{|\mathcal{I}|})$. It is also instructive to note the structure of the itemset search space; the set of all itemsets forms a lattice structure as shown in Figure 8.3, where any two itemsets X and Y are connected by a link if and only if X is a direct subset of Y (i.e., $X \subseteq Y$ and $|X| = |Y| - 1$). In terms of the 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 (shown in bold lines), where two itemsets X, Y are connected by a bold link if and only if X is a direct subset and prefix of Y.

Support Computation: This step computes the support of each candidate and

determines if it is frequent. This step requires access to the database $\mathcal{D}$ to determine the support. Notice that the complexity of each database scan is $O(\mathcal{I} \times |\mathcal{D}|)$.

The procedure COMPUTESUPPORT in Algorithm 8.1 shows the steps in computing the support of an itemset X. 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. After checking all the transactions COMPUTESUPPORT returns the support of X. Checking if X is a subset of $\mathbf{i}(t)$ can take time at most $|\mathcal{I}|$, since both X and $\mathbf{i}(t)$ are bounded in length by the cardinality of $\mathcal{I}$. Since we have to check against each transaction in $\mathcal{D}$, the total computational complexity of support counting for an itemset X, is $O(|\mathcal{I}| \cdot |\mathcal{D}|)$.

The overall computational complexity of BRUTEFORCE is $O(|\mathcal{I}| \cdot |\mathcal{D}| \cdot 2^{|\mathcal{I}|})$ combining the complexity of itemset enumeration and support computation steps. Since the database size in data mining can be very large, it is also important to measure the I/O (input/output) complexity of the algorithm in terms of the number of database scans required. Since 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. It is thus clear that the brute force approach is computationally infeasible for even small itemset spaces, whereas in practice $\mathcal{I}$ can be very large (for example, 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 upon the brute force approach, by improving both the candidate generation and support counting steps.

8.2.2 The Apriori Algorithm – Level-Wise Approach

The brute force approach enumerates all possible itemsets in its quest to determine the frequent ones. This typically results in a lot of wasteful computation since many of the candidates may not be frequent.

Let $X, Y \subseteq \mathcal{I}$ be any two itemsets. Observe that if $X \subseteq Y$, then $sup(X) \geq sup(Y)$, which leads to the following two corollaries

- If X is frequent, then any subset $Y \subseteq X$ is also frequent.
- If X is not frequent, then any superset $Y \supseteq X$ cannot be frequent.

Based on the above observations, we can significantly improve the itemset mining algorithm by reducing the number of candidates we generate, by limiting the candidates to be only those that will potentially be frequent. First we can stop generating supersets of a candidate once we determine that it is infrequent, since no superset of an infrequent itemset can be frequent. Second, we can avoid any candidate that has an infrequent subset. These two observations can result in significant *pruning* of the search space.

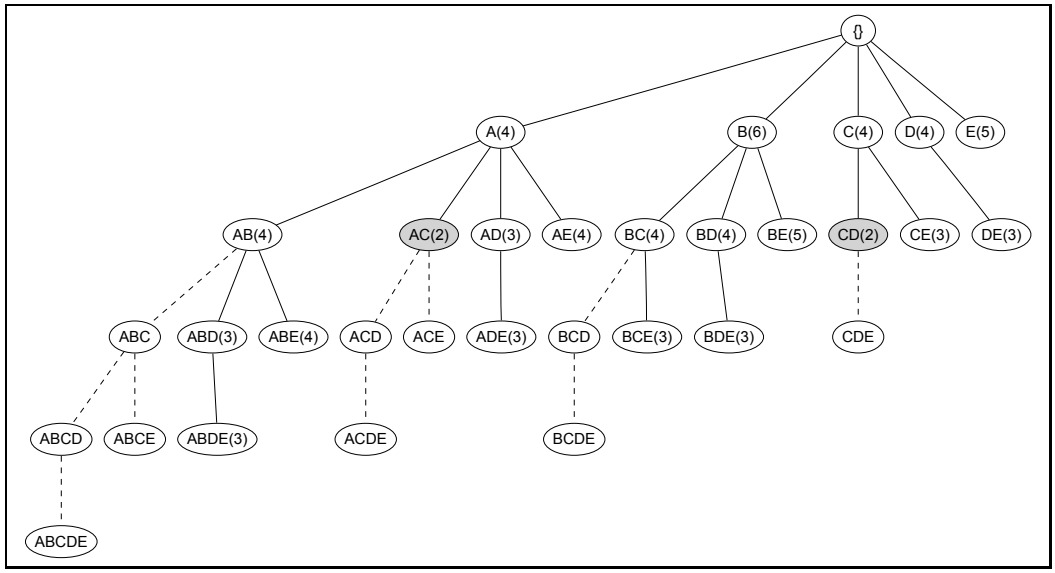

Figure 8.4: Prefix Search Tree and Effect of Pruning

Example 8.4: Given the example dataset in Figure 8.1, let $minsup = 3$. Algorithm BRUTEFORCE enumerates all $2^{|\mathcal{I}|} - 1$ non-empty itemsets, so the total number of candidates are $2^5 - 1 = 31$.

Figure 8.4 shows the prefix search tree; each node shows an itemset along with its support, for example, $AC(2)$ indicates that $sup(AC) = 2$. The figure demonstrates the power of pruning. For example, once we determine that AC is infrequent, we can prune any itemset that has AC as a prefix, i.e., the entire subtree under AC can be pruned. Note also that since AB is frequent, we first generate a potential candidate ABC, however, since AC is infrequent, we can immediately prune ABC without counting its support. In a similar manner other branches of the tree are pruned (shown in dashed lines). The bold lines indicate all the candidates whose support has to be counted against the database to check if they are frequent or not. Thus the total number of candidates generated in the new approach is 21, out of which 19 are frequent.

Improving Support Computation The I/O complexity can be significantly improved if we can count the support of groups of candidates instead of a single itemset at a time. Algorithm APRIORI enumerates the candidates using a level-wise (or BFS) approach, as illustrated in Figure 8.5. It generates the entire set of potential candidate k-itemsets at level k (denoted as $\mathcal{C}^{(k)}$), and counts their support in the database. The pseudo-code for the APRIORI approach is shown in Algorithm 8.2. It begins by

Algorithm 8.2: Algorithm APRIORI

```
   APRIORI (D, I, minsup)      :
1  C^(1) = {∅} // Create initial prefix tree of single items
2  foreach i ∈ I do
3      Add i as child of ∅ 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
9              print X, sup(X)
10         else
11             remove X from C^(k)
12     C^(k+1) ← EXTENDPREFIXTREE (C^(k))
13     k ← k + 1

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

   EXTENDPREFIXTREE (C^(k)) :
18 foreach leaf X_a ∈ C^(k) do
19     foreach
       leaf X_b ∈ C^(k), such that b > a and PARENT(X_a) = PARENT(X_b) do
20         X_ab ← X_a ∪ X_b
21         valid ← 1
22         foreach X_j ⊂ X_ab ∧ |X_j| = |X_ab| − 1 do
               // prune candidate if there are any infrequent subsets
23             if X_j ∉ C^(k) then valid ← 0
24         if valid then Add X_ab as child of X_a with sup(X_ab) ← 0
25 return C^(k)
```

inserting the single items into the prefix tree $\mathcal{C}^{(1)}$. Within the while loop (lines 5-13), the method first counts the support for the current candidates at level k, and removes any candidate below the minimum support (line 11). In general, during support counting (COMPUTESUPPORT), the method generates the k-subsets of each transaction in the database $\mathcal{D}$, and if that subset is found in the prefix tree, the cor-

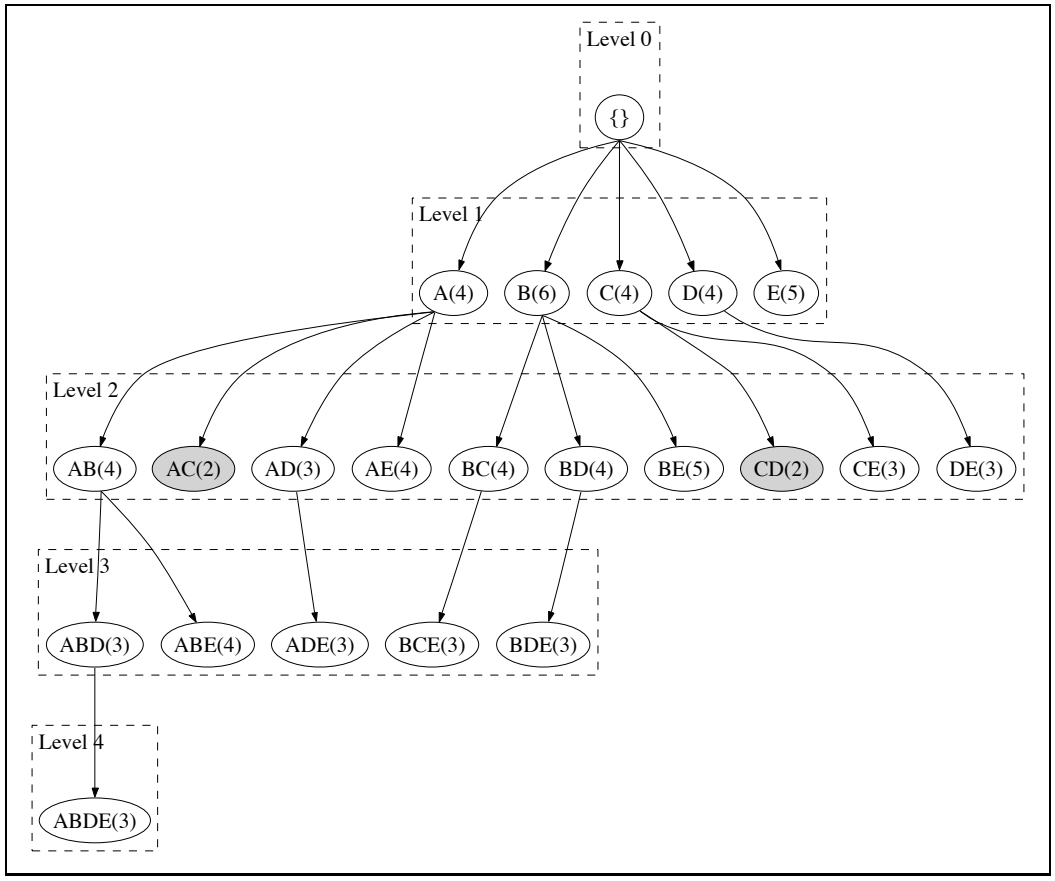

Figure 8.5: Itemset Mining: Apriori Algorithm

responding count is incremented. From those candidates that are frequent, the next level is added to the prefix tree (procedure EXTENDPREFIXTREE). If new candidates were added, the whole process is repeated for the next level. This process continues until no new candidates are added.

The worst-case computational complexity of APRIORI is still $O(|\mathcal{I}| \cdot |\mathcal{D}| \cdot 2^{|\mathcal{I}|})$, since all itemsets may be frequent. In practice, however, due to the pruning effects, the cost is much smaller, though it is hard to characterize. In terms of I/O, APRIORI requires l database scans (where l is the length of the longest frequent itemset), one per level.

8.2.3 The Eclat Algorithm – Tidset Intersection Approach

One way to improve the support counting step is to "index" the database in such a way that it allows fast support computations. Notice that in the level-wise approach, to count the support, we have to generate subsets of each transaction and check

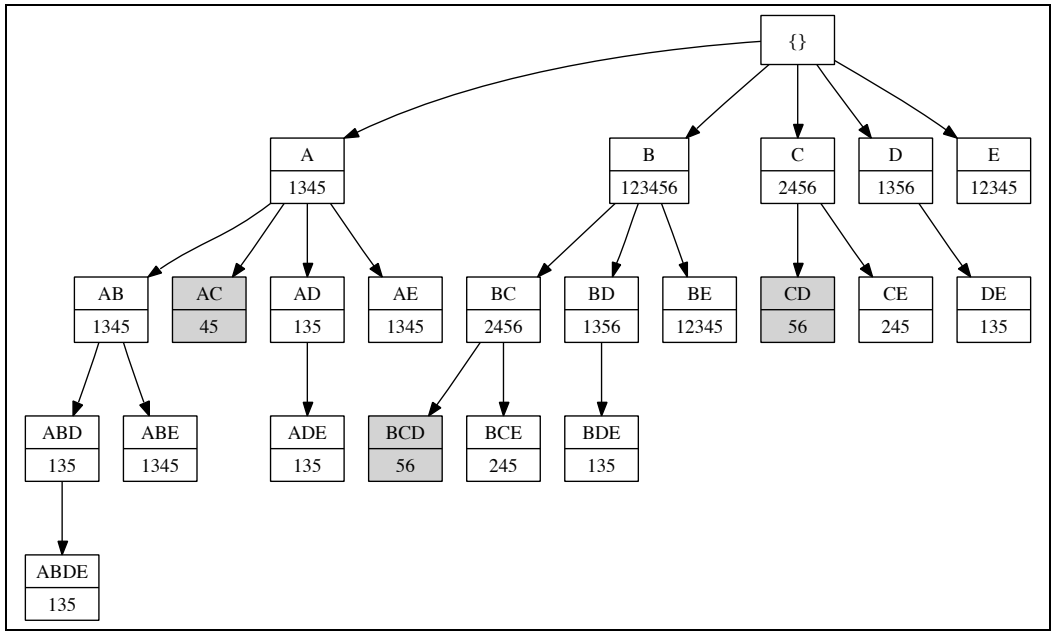

Figure 8.6: Itemset Mining: Tidlist Intersections

Algorithm 8.3: Algorithm ECLAT

```
// Initial Call: ECLAT ({⟨i, t(i)⟩ : i ∈ I}, minsup)
ECLAT (P, minsup):
1 foreach ⟨X, t(X)⟩ ∈ P do
2     P_X ← ∅
3     foreach ⟨Y, t(Y)⟩ ∈ P, with Y > X do
4         N_XY = X ∪ Y
5         t(N_XY) = t(X) ∩ t(Y)
6         if sup(N_XY) ≥ minsup then
7             P_X ← P_X ∪ {⟨N_XY, t(N_XY)⟩}
8             print N_XY, sup(N_XY)
9     ECLAT (P_X, minsup)
```

whether they exist in the prefix tree. This can be expensive since we may end up generating subsets that do not exist in the prefix tree.

One solution is to leverage the tidsets directly and use them for computation. The basic idea is that the support of a candidate itemset can be computed by intersecting the tidsets of suitably chosen subsets. For example, if we know that tidsets $\mathbf{t}(A) = 1345$ and $\mathbf{t}(C) = 2456$, then we can determine the support of AC by intersecting the two tidsets, to obtain $\mathbf{t}(AC) = \mathbf{t}(A) \cap \mathbf{t}(C) = 1345 \cap 2456 = 45$. To enumerate

all the frequent itemsets, we can start with the tidsets for each item in $\mathcal{I}$, and aided by the prefix search tree, we can determine the support of each candidate via tidset intersections, as illustrated in Figure 8.6.

The complete algorithm is given in Algorithm 8.3. It enumerates the candidates in a DFS manner. The initial call is with all the single items along with their tidsets. Let us call a tuple of the form $\langle X, \mathbf{t}(X)\rangle$ an itemset-tidset pair (IT-pair). In each recursive call ECLAT intersects each IT-pair $\langle X, \mathbf{t}(X)\rangle$ with all the other IT-pairs $\langle Y, \mathbf{t}(Y)\rangle$ to generate new candidates N_{XY}. If the new candidate is frequent, it is added to the set P_X. A recursive call to ECLAT then finds all extensions of the X branch in the search tree (see Figure 8.6). The process continues in a DFS manner until all frequent sets have been found.

The computational complexity of ECLAT is $O(|\mathcal{D}| \cdot 2^{|\mathcal{I}|})$ in the worst case. In practice it is very efficient, since support is directly obtained by fast intersection operations. If t is the average transaction size, and if l is the longest frequent itemset, the computational complexity is close to $O(t \cdot 2^l)$. The I/O complexity of ECLAT is hard to characterize, since it depends on the size of the intermediate tidsets. Also due to the DFS search, as soon as the IT-pairs fit in memory no further database scans are required. Since the database size is $O(t|\mathcal{I}|)$, the I/O cost is $\frac{t \cdot 2^l}{t|\mathcal{I}|} = O(\frac{2^l}{|\mathcal{I}|})$ scans, in the worst case.

8.2.4 The FPGrowth Algorithm – Frequent Pattern Tree Approach

Another approach to indexing the database for fast support computation is via the use of an augmented prefix tree called the *frequent pattern tree* (FP-tree) that stores the support information for the itemset prefixes over all transactions. Each node in the tree records the support of the itemset that node represents, which is composed of the items on the path from the root to that node. The FP-tree is constructed as follows. Initially the tree, say R, contains as root the null itemset $\emptyset$. We next compute the support of each single item $i \in \mathcal{I}$, and after discarding the infrequent items, we sort them in decreasing order of support. Next, for each transaction $\langle t, X\rangle \in \mathcal{D}$, we sort X using the item ordering above, and insert X into R, incrementing the support of all nodes along the path spelling X, with each node labeled with a single item in 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, a new nodes are created under the common prefix, with counts initialized to 1. The FP-tree is complete when all transactions have been inserted.

Example 8.5: Consider the example database in Figure 8.1. We add each transaction one by one into the prefix tree, and keep track of the path frequencies at each node. Since we want the tree to be as compact as possible, we want the most frequent items to be at the top of the tree. We reorder the items and rank them in decreasing order of support. For our example database we obtain the sorted

order as: $\{B(6), E(5), A(4), C(4), D(4)\}$. Next each transaction can be reordered by the item rank, for example, $\langle 1, ABDE\rangle$ becomes $\langle 1, BEAD\rangle$ in the sorted order. Figure 8.7 illustrates the prefix tree as each sorted transaction is added.

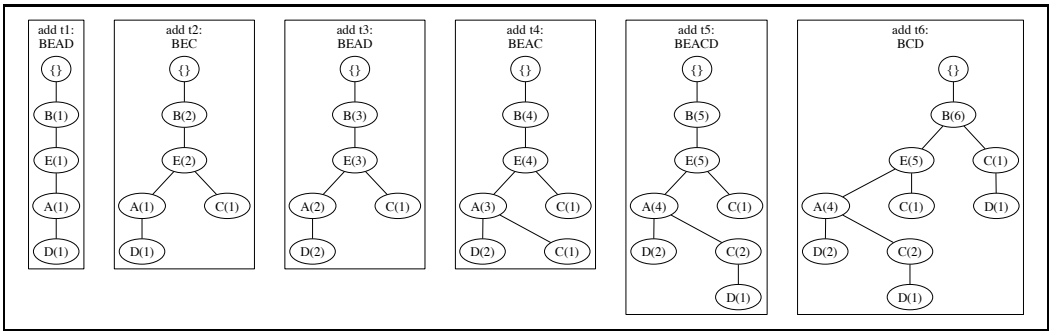

Figure 8.7: Frequent Pattern Tree

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 8.4. The method accepts as input a FP-tree R, and the current itemset prefix P. Initially both R and P are empty.

Given a FP-tree R, projected FP-trees are built for each frequent item i in R in increasing order of support. We find all the occurrences of i in the tree, and for each occurrence, we determine the corresponding path to the root (line 13). The support of item i on each such path is recorded (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 weight of each node in R_X is incremented by the path support $pathsup(i)$. The resulting FP-tree is a projection of the itemset X that comprises the current prefix extended with the item i (line 3). We then call FPGROWTH recursively with projected FP-tree R_X and 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. Single-path FP-trees 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 1-5).

Example 8.6: Figure 8.8 illustrates how to extract the frequent itemsets from the FP tree built in Example 8.5. We start by determining the projected augmented prefix trees for items D, C, A, E, and B, in the order listed, in a depth-first manner. That is, we first generate the projected tree for D, which in turn, is recursively processed.

Algorithm 8.4: Algorithm FPGROWTH

```
// Initial Call: FPGROWTH (∅, ∅, minsup)
FPGROWTH (R, P, minsup):
1  if IsPath (R) then
2      foreach X_i ⊆ R do
3          X ← P ∪ X_i
4          supp(X) ← min_{i∈X}{sup(i)}
5          print X, sup(X)
6  else
7      I ← {i | i ∈ R, sup(i) ≥ minsup}
8      foreach i ∈ I in increasing order of sup(i) do
9          X ← P ∪ {i}
10         sup(X) ← sup(i)
11         print X, sup(X)
12         R_X ← ∅ // projected FP-tree for X
13         foreach path ∈ PathToRoot(i) do
14             pathsup(i) ← support of i in path path
15             Insert p into FP-tree R_X with support pathsup(i)
16         if R_X ≠ ∅ then FPGROWTH (R_X,X,minsup)
```

From the FP-tree for D we project onto DA, DE, and DB, in turn. From the projected tree of DA, which is a single path tree, we generate the frequent itemsets {DAE, DAEB, DAB} all with support 3. From the projected tree of DE, we generate the frequent itemset {DEB}. The tree for DB is empty.

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, CEB, CB}, {AE, AEB, AB}, and {EB}, respectively.

8.3 Annotated References

8.4 Exercises and Projects

1. Let D be divided into k equal sized partitions, denoted D_i, for $i = 1...k$. Show that a frequent itemset in D is also frequent in some partition D_i for a given minimum support σ (note: σ is the ratio of the itemset count divided by the number of records in a database).

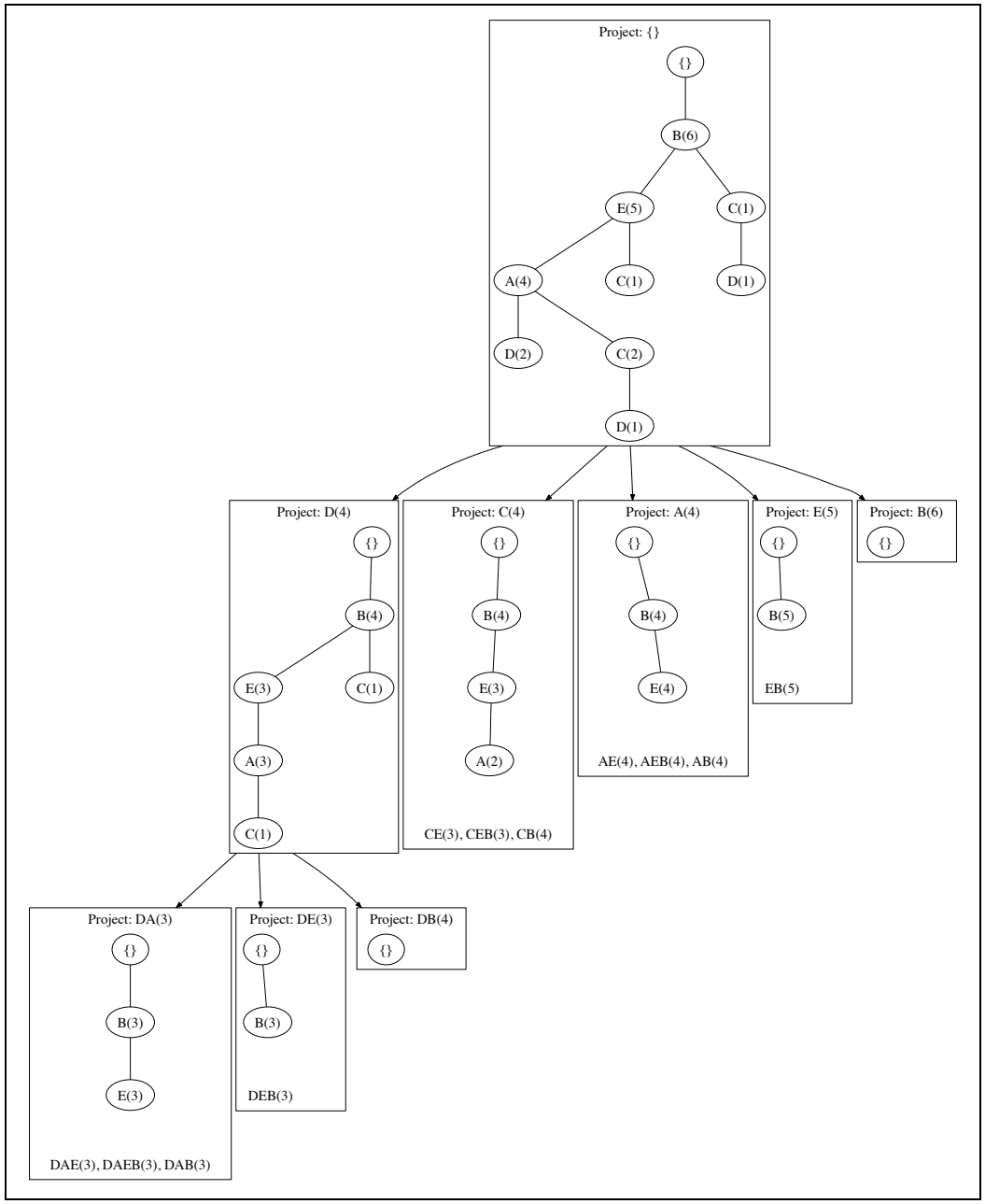

Figure 8.8: Itemset Mining: Augmented Prefix Tree

2. Let X be a frequent itemset, and let $Y = \{Y_1, Y_2, ..., Y_k\}$ be the collection of all its subsets. Prove that the confidence of the rule $Y_i \Rightarrow X - Y_i$ cannot be more than the confidence of the rule $Y_k \Rightarrow X - Y_k$, if $Y_i \subset Y_k$.