

## Chapter 5

# 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 frequent bought together, at a supermarket by analyzing the customer shopping carts (the so-called “market baskets”).

Once we mine the frequent sets, they allows 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.

### 5.1 Foundations

Let  $I = \{x_1, x_2, \dots, x_m\}$  be a set of binary-valued attributes, which we call *items*. A set  $X \subseteq I$  is called an *itemset*. The set of items  $I$  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  $I^{(k)}$  the set of all  $k$ -itemsets, i.e., subsets of  $I$  with size  $k$ . Let  $\mathcal{T} = \{t_1, t_2, \dots, t_n\}$  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.,  $\mathcal{D} \subseteq I \times \mathcal{T}$ . We say that tid  $t \in \mathcal{T}$  *contains* item  $x \in I$  if and only if  $(x, t) \in \mathcal{D}$ . In other words,  $(x, t) \in \mathcal{D}$  iff  $x \in X$  in the tuple  $\langle t, X \rangle$ .

	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 5.1: An Example Database

**Example:** Figure 5.1(a) shows an example binary database. Here  $I = \{A, B, C, D, E\}$ , 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  $(t, x) \in \mathcal{D}$ , 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  $\mathbf{i} : 2^{\mathcal{T}} \rightarrow 2^I$  be a function, defined as follows:

$$\mathbf{i}(T) = \{x \mid \forall t \in T, t \text{ contains } x\} \quad (5.1)$$

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

$$\mathbf{t}(X) = \{t \mid X \subseteq \mathbf{i}(t)\} \quad (5.2)$$

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  $x \in I$ . 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  $x \in I$ . 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:** Figure 5.1(b) shows the corresponding transaction database for the binary database in (a). For instance, the first transaction is  $\langle 1, \{A, B, D, E\} \rangle$ . 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 5.1(c) shows the corresponding vertical database for the binary database in (a). 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. ■

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

$$\text{sup}(X, \mathcal{D}) = |\{t_i \mid \langle t_i, \mathbf{i}(t_i) \rangle \in \mathcal{D} \text{ and } X \subseteq \mathbf{i}(t_i)\}| \quad (5.3)$$

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:

$$\text{sup}(X, \mathcal{D}) = |\mathbf{t}(X)| \quad (5.4)$$

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

$$P(X) = \frac{\text{sup}(X)}{|\mathcal{D}|} \quad (5.5)$$

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

$$s = \text{sup}(A \implies B) = P(A \wedge B) \times |\mathcal{D}| = \frac{\text{sup}(A \cup B)}{|\mathcal{D}|} \times |\mathcal{D}| = \text{sup}(A \cup B) \quad (5.6)$$

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

$$c = \text{conf}(A \implies B) = P(B|A) = \frac{P(A \wedge B)}{P(A)} = \frac{\frac{\text{sup}(A \cup B)}{|\mathcal{D}|}}{\frac{\text{sup}(A)}{|\mathcal{D}|}} = \frac{\text{sup}(A \cup B)}{\text{sup}(A)} \quad (5.7)$$

A rule is frequent if the itemset  $A \cup B$  is frequent. A rule is *confident* if  $\text{conf} \geq \text{minconf}$ , 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:** Given the example dataset in Figure 5.1, let *minsup* = 3. Figure 5.2 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  $\text{sup}(BCE) = |\mathbf{t}(BCE)| = 3$ , and it is frequent. ■

<i>sup</i>	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 5.2: Frequent Itemsets with *minsup* = 3

## 5.2 Generating Strong 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

$$A \xrightarrow{s,c} X - A \quad (5.8)$$

We already know that the rule must be frequent, since

$$s = \text{sup}(A \cup (X - A)) = \text{sup}(X) \geq \text{minsup} \quad (5.9)$$

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

$$c = \frac{\text{sup}(A \cup (X - A))}{\text{sup}(A)} = \frac{\text{sup}(X)}{\text{sup}(A)} \quad (5.10)$$

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

### 5.3 Search for Frequent Itemsets

```

BRUTEFORCE ( $\mathcal{D}$ ,  $I$ , minsup):
1 foreach  $X \subseteq I$  do
2    $\text{sup}(X) \leftarrow \text{COMPUTESUPPORT}(X, \mathcal{D})$ ;
3   if  $\text{sup}(X) \geq \text{minsup}$  then
4      $\text{print } X, \text{sup}(X)$ 

```

**Algorithm 5.1:** Algorithm BRUTEFORCE

Consider the algorithm BRUTEFORCE, that uses a brute force approach to mine all frequent itemsets. Algorithm BRUTEFORCE enumerates all the possible subsets of  $I$ , and for each such subset  $X \subseteq I$ , 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:

**Itemset Enumeration:** This step generates all the subsets of  $I$ , 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  $2^{|I|}$  potentially frequent itemsets. The computational complexity of enumerating all the itemsets is thus  $O(2^{|I|})$ . 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 5.3, where any two itemsets  $X$  and  $Y$  are connected by a link *iff*  $X$  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 *iff*  $X$  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 the database  $\mathcal{D}$  to determine the support.

```

COMPUTESUPPORT ( $X, \mathcal{D}$ ):
1  $\text{sup}(X) \leftarrow 0$ 
2 foreach  $\langle t, \mathbf{i}(t) \rangle \in \mathcal{D}$  do
3   if  $X \subseteq \mathbf{i}(t)$  then
4      $\text{sup}(X) \leftarrow \text{sup}(X) + 1$ 
5 return  $\text{sup}(X)$ 

```

**Algorithm 5.2:** Algorithm COMPUTESUPPORT

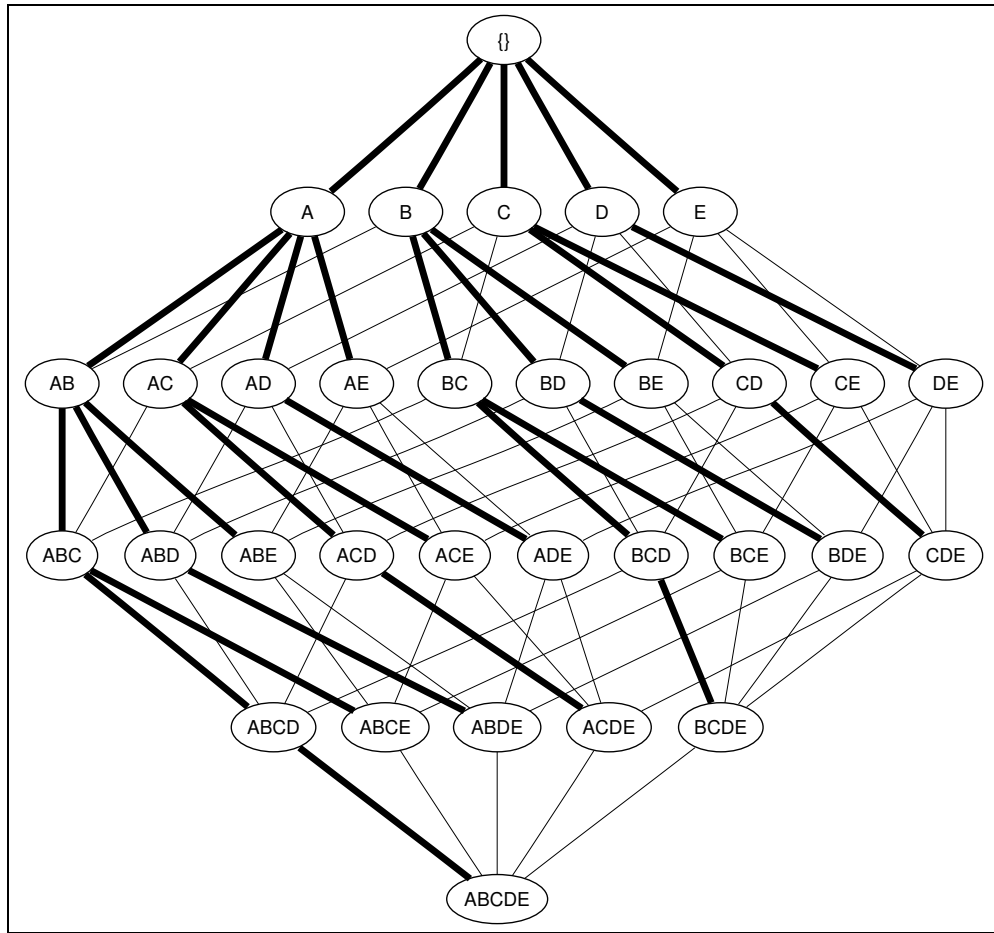


Figure 5.3: Itemset Lattice & Prefix-based Search Tree (in Bold)

Algorithm 5.2 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  $t$ . 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  $|I|$ , since both  $X$  and  $\mathbf{i}(t)$  are bounded in length by the cardinality of  $I$ . Since we have to check against each transaction in  $\mathcal{D}$ , the total computational complexity of support counting for an itemset  $X$ , is  $O(|I| \cdot |\mathcal{D}|)$ .

The overall computational complexity of `BRUTEFORCE` is  $O(|I| \cdot |\mathcal{D}| \cdot 2^{|I|})$  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  $O(2^{|I|})$  database scans. It is thus clear that this brute force approach is clearly computationally infeasible for even small itemset spaces, whereas in practice  $I$  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.

### 5.3.1 Improving Candidate Generation

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  $X, Y \subseteq I$  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.

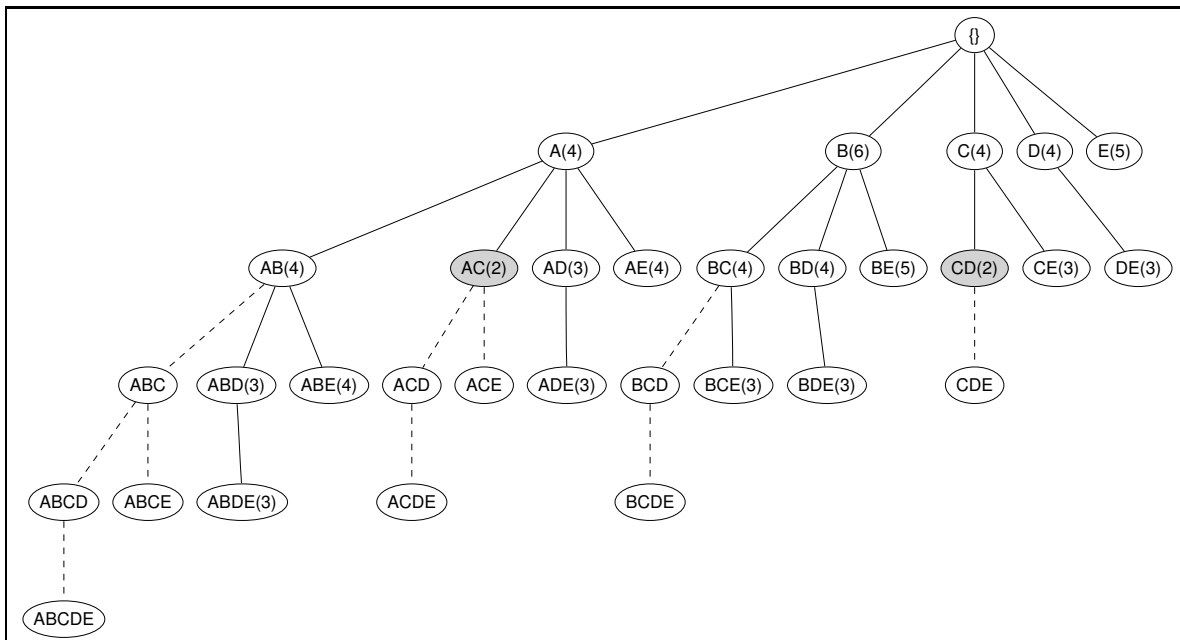


Figure 5.4: Prefix Search Tree and Effect of Pruning

**Example:** Given the example dataset in Figure 5.1, let  $minsup = 3$ . Algorithm BRUTEFORCE enumerates all  $2^{|I|} - 1$  non-empty itemsets, so the total number of candidates are  $2^5 - 1 = 31$ .

Figure 5.4 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. ■

### 5.3.2 Improving Support Computation

```

LEVELWISE ( $\mathcal{D}, I, \text{minsup}$ ) :
1  $k \leftarrow 1$  //  $k$  denotes the level
2  $\mathcal{C}^{(k)} \leftarrow \text{INITIALIZEPREFIXTREE}(I)$ 
3 while  $\mathcal{C}^{(k)} \neq \emptyset$  do
4   COMPUTESUPPORT( $\mathcal{C}^{(k)}, \mathcal{D}$ );
5   foreach  $X \in \mathcal{C}^{(k)}$  do
6     if  $\text{sup}(X) < \text{minsup}$  then
7       remove  $X$  from  $\mathcal{C}^{(k)}$ 
8    $\mathcal{C}^{(k+1)} \leftarrow \text{EXTENDPREFIXTREE}(\mathcal{C}^{(k)})$ 
9    $k \leftarrow k + 1$ 

COMPUTESUPPORT ( $\mathcal{C}^{(k)}, \mathcal{D}$ ):
10 foreach  $X \in \mathcal{C}^{(k)}$  do
11    $\text{sup}(X) \leftarrow 0$ 
12 foreach  $\langle t, \mathbf{i}(t) \rangle \in \mathcal{D}$  do
13   foreach  $k$ -subset  $X \subseteq \mathbf{i}(t)$  do
14     if  $X \in \mathcal{C}^{(k)}$  then
15        $\text{sup}(X) \leftarrow \text{sup}(X) + 1$ 

```

**Algorithm 5.3:** Algorithm LEVELWISE

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 LEVELWISE enumerates the candidates using a levelwise (or BFS) approach, as illustrated in Figure 5.5. It generates the entire set of potential candidate  $k$ -itemsets at level  $k$  and counts their support in the database. The pseudocode for the levelwise approach is shown in Algorithm 5.3. The levelwise approach begins by inserting the single items into the prefix tree (function INITIALIZEPREFIXTREE). It then counts the support for the current level  $k = 1$ , and removes any candidate below the minimum support (line 7). In general, during support counting (COMPUTESUPPORT), we generate the  $k$ -subsets of each transaction in the database  $\mathcal{D}$ , and if that subset is found in the prefix tree, we increment its count. From those candidates that are frequent, the next level is added to the prefix tree (function EXTENDPREFIXTREE). If new candidates were added, the whole process is repeated at the next level. This process continues until no new candidates are added.

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

## 5.4 Itemset Mining via Tidset Intersections

One way to improve the support counting step is to “index” the database in such a way that allows fast support computations. Notice that in the levelwise approach, to count the support, we have to generate

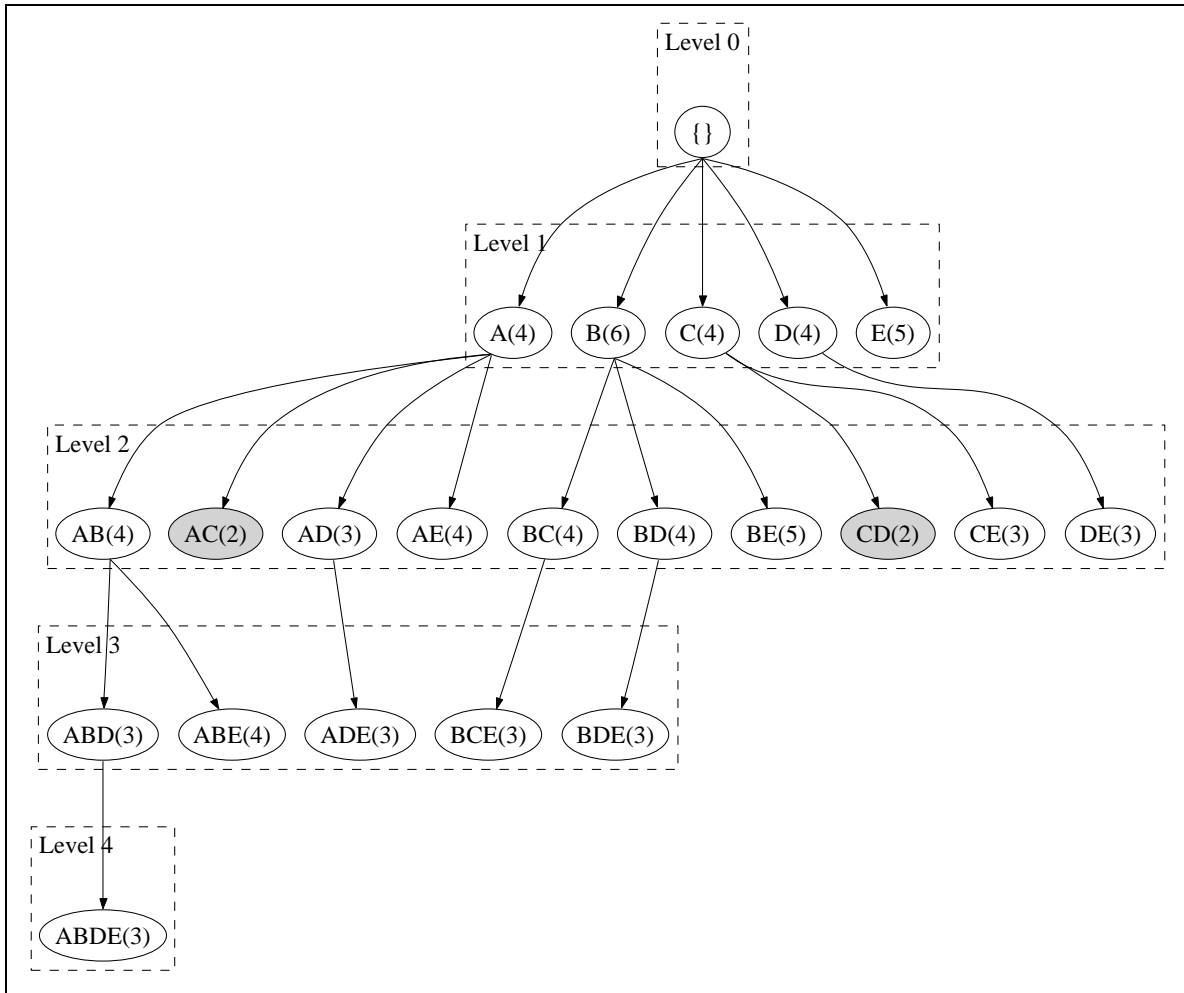


Figure 5.5: Itemset Mining: Levelwise Approach

subsets of each transaction and check if 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  $I$ , and aided by the prefix search tree, we can determine the support of each candidate via tidset intersections, as illustrated in Figure 5.6.

The complete algorithm is given in Algorithm 5.4. 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  $\langle X, \mathbf{t}(X) \rangle$  as an itemset-tidset pair (IT-pair). In each recursive call `INTERSECTTIDSETS` 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 `INTERSECTTIDSETS` then finds all extensions of the  $X$  branch in the search tree (see Figure 5.6). The process continues in a DFS manner until all frequent sets have been found.

The computational complexity of `INTERSECTTIDSETS` is  $O(|\mathcal{D}| \cdot 2^{|\mathcal{I}|})$  in the worst case. In practice is it

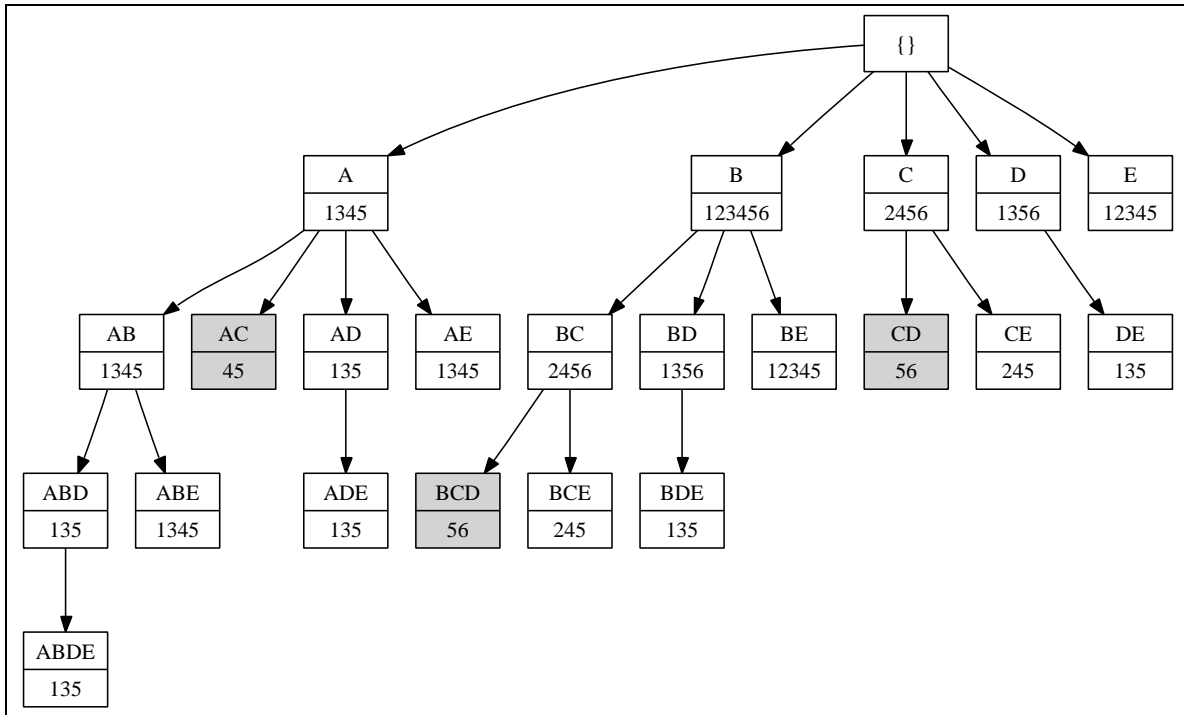


Figure 5.6: Itemset Mining: Tidlist Intersections

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 INTERSECTTIDSETS 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  $O(t|I|)$ , the I/O cost is  $\frac{t \cdot 2^l}{t|I|} = O(\frac{2^l}{|I|})$  scans, in the worst case.

## 5.5 Itemset Mining via Augmented Prefix Trees

Another approach to indexing the database, for fast support computation, is via the use of augmented prefix trees. An augmented prefix tree stores the support information for the prefixes over the set of transactions. Each node in the tree records a support value, which is the support of the itemset that node represents, which, in turn, is composed of the items on the path from the root to that node.

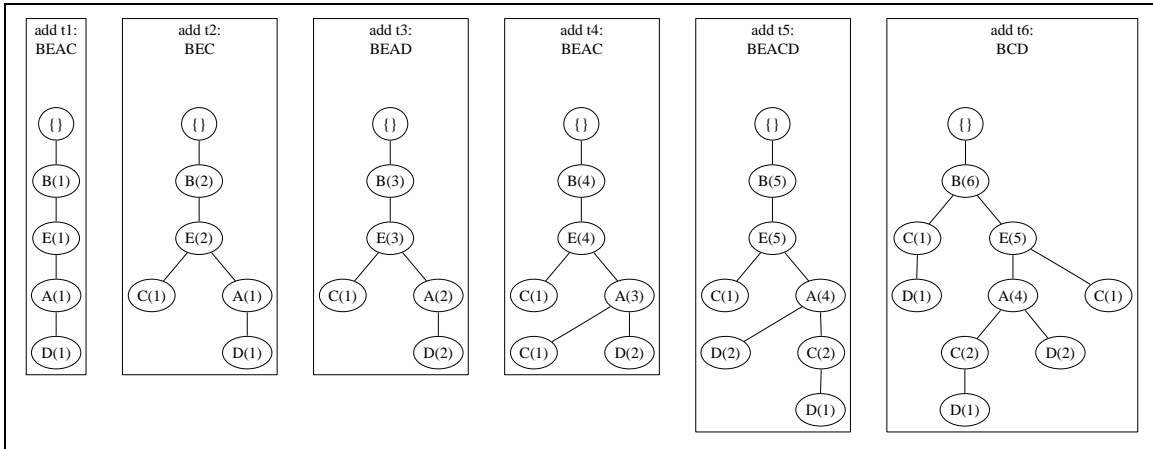
Consider the example database in Figure 5.1. We add each transaction one by one into the prefix tree, and keep track of the path frequencies at each node. Since we want the tree to be as compact as possible, we want the most frequent items to be at the top of the tree. We can reorder the items and rank 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 5.7 illustrates the prefix tree as each sorted transaction is added.

```

// Initial Call: INTERSECTTIDSETS( $\{\langle i, \mathbf{t}(i) \rangle : i \in I\}, \text{minsup}$ )
INTERSECTTIDSETS( $P, \text{minsup}$ ):
1 foreach  $\langle X, \mathbf{t}(X) \rangle \in P$  do
2    $P_X \leftarrow \emptyset$ 
3   foreach  $\langle Y, \mathbf{t}(Y) \rangle \in P$ , with  $Y > X$  do
4      $N_{XY} = X \cup Y$ 
5      $\mathbf{t}(N_{XY}) = \mathbf{t}(X) \cap \mathbf{t}(Y)$ 
6     if  $\text{sup}(N) \geq \text{minsup}$  then
7        $P_X \leftarrow P_X \cup \{\langle N_{XY}, \mathbf{t}(N_{XY}) \rangle\}$ 
8       print  $N_{XY}, \text{sup}(N_{XY})$ 
9   INTERSECTTIDSETS( $P_X, \text{minsup}$ )

```

**Algorithm 5.4:** Algorithm INTERSECTTIDSETS



**Figure 5.7:** Augmented Prefix Tree using Sorted Items

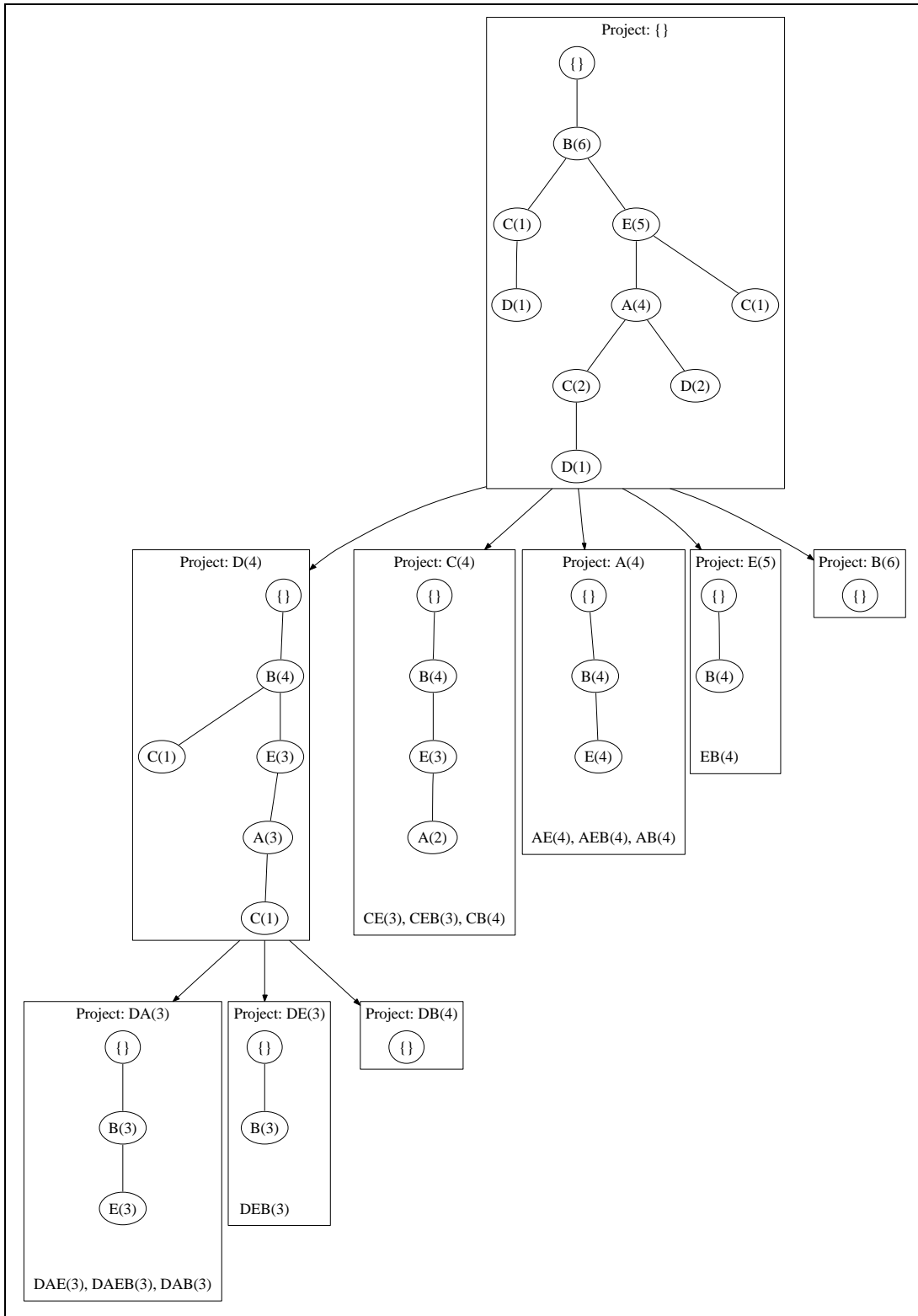


Figure 5.8: Itemset Mining: Augmented Prefix Tree