As we have seen, in many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However, it can suffer from two nontrivial costs:

- It may still need to generate a huge number of candidate sets. For example, if there are 104 frequent 1-itemsets, the Apriori algorithm will need to generate more than 107 candidate 2-itemsets.
- It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets.

"Can we design a method that mines the complete set of frequent itemsets without such a costly candidate generation process?" An interesting method in this attempt is called frequent pattern growth, or simply FP-growth, which adopts a divide-and-conquer strategy as follows. First, it compresses the database representing frequent items into a frequent pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases (a special kind of projected database), each associated with one frequent item or "pattern fragment," and mines each database separately. For each "pattern fragment," only its associated data sets need to be examined. Therefore, this approach may substantially reduce the size of the data sets to be searched, along with the "growth" of patterns being examined. You will see how it works in Example 6.5.

Example 6.5 FP-growth (finding frequent itemsets without candidate generation). We reexamine the mining of transaction database, D, of Table 6.1 in Example 6.3 using the frequent pattern growth approach.

The first scan of the database is the same as Apriori, which derives the set of frequent items (1-itemsets) and their support counts (frequencies). Let the minimum support count be 2. The set of frequent items is sorted in the order of descending support count. This resulting set or list is denoted by L. Thus, we have L = {{I2: 7}, {I1: 6}, {I3: 6}, {I4: 2}, {I5: 2}}.

An FP-tree is then constructed as follows. First, create the root of the tree, labeled with "null." Scan database D a second time. The items in each transaction are processed in L order (i.e., sorted according to descending support count), and a branch is created for each transaction. For example, the scan of the first transaction, "T100: I1, I2, I5," which contains three items (I2, I1, I5 in L order), leads to the construction of the first branch of the tree with three nodes, (I2: 1), (I1: 1), and (I5: 1), where I2 is linked as a child to the root, I1 is linked to I2, and I5 is linked to I1. The second transaction, T200, contains the items I2 and I4 in L order, which would result in a branch where I2 is linked to the root and I4 is linked to I2. However, this branch would share a common prefix, I2, with the existing path for T100. Therefore, we instead increment the count of the I2 node by 1, and create a new node, (I4: 1), which is linked as a child to (I2: 2).

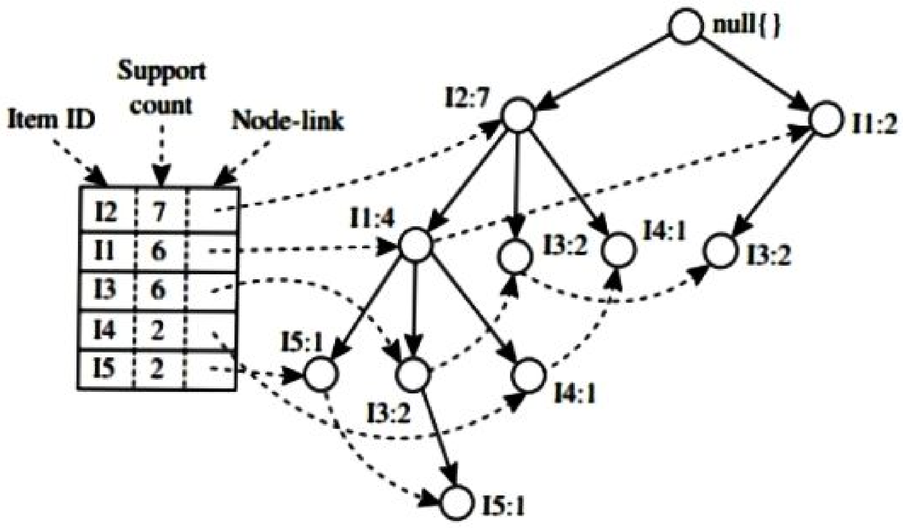

**Figure 6.7** An FP-tree registers compressed, frequent pattern information.

In general, when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly.

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. The tree obtained after scanning all the transactions is shown in Figure 6.7 with the associated node-links. In this way, the problem of mining frequent patterns in databases is transformed into that of mining the FP-tree.

The FP-tree is mined as follows. Start from each frequent length-1 pattern (as an initial suffix pattern), construct its conditional pattern base (a "sub-database," which consists of the set of prefix paths in the FP-tree co-occurring with the suffix pattern), then construct its (conditional) FP-tree, and perform mining recursively on the tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Mining of the FP-tree is summarized in Table 6.2 and detailed as follows. We first consider I5, which is the last item in L, rather than the first. The reason for starting at the end of the list will become apparent as we explain the FP-tree mining process. I5 occurs in two FP-tree branches of Figure 6.7. (The occurrences of I5 can easily be found by following its chain of node-links.) The paths formed by these branches are ⟨I2, I1, I5: 1⟩ and ⟨I2, I1, I3, I5: 1⟩. Therefore, considering I5 as a suffix, its corresponding two prefix paths are ⟨I2, I1: 1⟩ and ⟨I2, I1, I3: 1⟩, which form its conditional pattern base. Using this conditional pattern base as a transaction database, we build an I5-conditional FP-tree, which contains only a single path, ⟨I2: 2, I1: 2⟩; I3 is not included because its support count of 1 is less than the minimum support count. The single path generates all the combinations of frequent patterns: {I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}.

For I4, its two prefix paths form the conditional pattern base, {{I2 I1: 1}, {I2: 1}}, which generates a single-node conditional FP-tree, hI2: 2i, and derives one frequent pattern, {I2, I4: 2}.

**Table 6.2** Mining the FP-Tree by Creating Conditional (Sub-)Pattern Bases

| *Item* | *Conditional Pattern Base* | *Conditional FP-tree* | *Frequent Patterns Generated* |
|---|---|---|---|
| I5 | {{I2, I1: 1}, {I2, I1, I3: 1}} | ⟨I2: 2, I1: 2⟩ | {I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2} |
| I4 | {{I2, I1: 1}, {I2: 1}} | ⟨I2: 2⟩ | {I2, I4: 2} |
| I3 | {{I2, I1: 2}, {I2: 2}, {I1: 2}} | ⟨I2: 4, I1: 2⟩, ⟨I1: 2⟩ | {I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2} |
| I1 | {{I2: 4}} | ⟨I2: 4⟩ | {I2, I1: 4} |

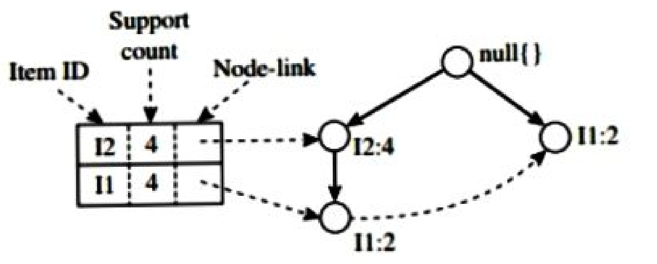

**Figure 6.8** The conditional FP-tree associated with the conditional node I3.

Similar to the preceding analysis, I3's conditional pattern base is {{I2, I1: 2}, {I2: 2}, {I1: 2}}. Its conditional FP-tree has two branches, ⟨I2: 4, I1: 2⟩ and ⟨I1: 2⟩, as shown in Figure 6.8, which generates the set of patterns {{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}}. Finally, I1's conditional pattern base is {{I2: 4}}, with an FP-tree that contains only one node, ⟨I2: 4⟩, which generates one frequent pattern, {I2, I1: 4}. This mining process is summarized in Figure 6.9.

Algorithm: FP growth. Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

D, a transaction database;

min sup, the minimum support count threshold.

Output: The complete set of frequent patterns.

**Method:**

1. The FP-tree is constructed in the following steps:

   **(a)** Scan the transaction database $D$ once. Collect $F$, the set of frequent items, and their support counts. Sort $F$ in support count descending order as $L$, the *list* of frequent items.

   **(b)** Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in $D$ do the following.
   Select and sort the frequent items in *Trans* according to the order of $L$. Let the sorted frequent item list in *Trans* be $[p|P]$, where $p$ is the first element and $P$ is the remaining list. Call **insert_tree**($[p|P]$, $T$), which is performed as follows. If $T$ has a child $N$ such that *N.item-name* = *p.item-name*, then increment $N$'s count by 1; else create a new node $N$, and let its count be 1, its parent link be linked to $T$, and its node-link to the nodes with the same *item-name* via the node-link structure. If $P$ is nonempty, call **insert_tree**($P$, $N$) recursively.

2. The FP-tree is mined by calling **FP_growth**(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, $\alpha$)
(1)  if Tree contains a single path P then
(2)      for each combination (denoted as $\beta$) of the nodes in the path P
(3)          generate pattern $\beta \cup \alpha$ with support_count = minimum support count of nodes in $\beta$;
(4)  else for each $a_i$ in the header of Tree {
(5)      generate pattern $\beta = a_i \cup \alpha$ with support_count = $a_i$.support_count;
(6)      construct $\beta$'s conditional pattern base and then $\beta$'s conditional FP_tree $Tree_\beta$;
(7)      if $Tree_\beta \neq \emptyset$ then
(8)          call FP_growth($Tree_\beta$, $\beta$); }
```

**Figure 6.9** FP-growth algorithm for discovering frequent itemsets without candidate generation.

The FP-growth method transforms the problem of finding long frequent patterns into searching for shorter ones in much smaller conditional databases recursively and then concatenating the suffix. It uses the least frequent items as a suffix, offering good selectivity. The method substantially reduces the search costs.

When the database is large, it is sometimes unrealistic to construct a main memorybased FP-tree. An interesting alternative is to first partition the database into a set of projected databases, and then construct an FP-tree and mine it in each projected database. This process can be recursively applied to any projected database if its FP-tree still cannot fit in main memory.

A study of the FP-growth method performance shows that it is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm.

## Cluster Analysis

Imagine that you are given a set of data objects for analysis where, unlike in classi_cation, the class label of each object is not known. Clustering is the process of grouping the data into classes or clusters so that objects within a cluster have high