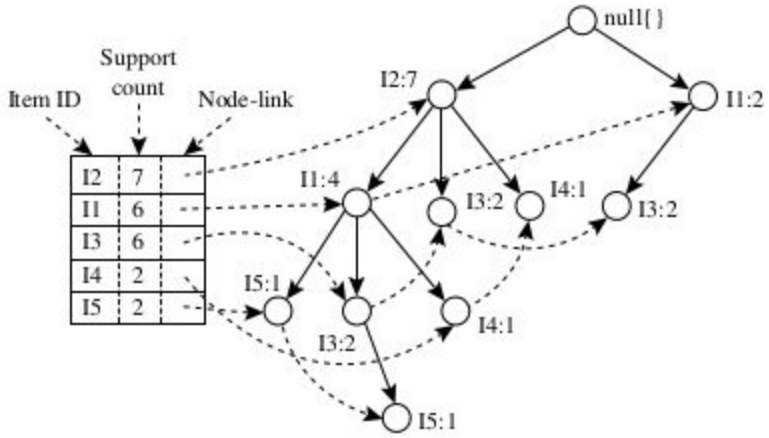
Experiments/Optimisations associated with Apriori generation and FP-Growth

Note : This doc only points out at the differences between standard approaches and the experiments/optimisations , not the entire algorithm, for exact understanding you need to refer to Han, Kimber, Pei . This doc is just for Help understanding the experiments in detail .

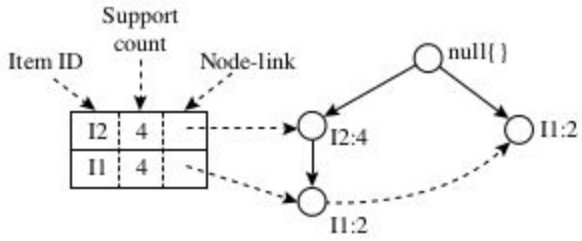
FP-Growth :



An FP-tree registers compressed, frequent pattern information.

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}



Bottom-Up Projection Technique :

The standard bottom-up projection technique works this way :

- Start with the item with the least support count (holding the bottom place in the table) . In our example, I5 .
- For I5 we go to the node link as stored in the table (pick item ID, Support Count, Node-Link) .
- Traverse from the node to the root to get 1 prefix path associated with the item I5, (ignore the item itself), here the 1st prefix path : {I2,I1:1}
- Notice that we keep a next pointer from the node link of I5 to its next instance, go to the next instance and similarly generate another prefix path, here the 2nd prefix path : {I2,I1,I3 : 1} . Keep going on to the next instance until there is none .
- So for I5, we gathered the set of prefix paths, which we call, Conditional Pattern Base , for I5, {{I2,I1:1},{I2,I1,I3 : 1}} .
- Based on the conditional pattern base, the conditional FP-tree is created and on that a recursive FP-growth is called for . Notice below how the FP-growth expands recursively .

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 $N.item\text{-}name = p.item\text{-}name$, 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, α)
(1)  if Tree contains a single path P then
(2)     for each combination (denoted as β) of the nodes in the path P
(3)        generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)  else for each a_i in the header of Tree {
(5)     generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)     construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)     if Tree_β ≠ ∅ then
(8)        call FP_growth(Tree_β, β); }
```

- Do the same with the rest of the items in non decreasing order of their support counts, move up the table.

Top-Down Projection Technique :

Note, this isn't necessarily an optimisation , this may be slower in runtime than bottom-up .

- Here instead of moving in non decreasing order of support counts, move in non increasing order, i.e., move down the table . So for our example, the first item to be considered is I2 .
- For I2 we go to the node link as stored in the table (pick item ID, Support Count, Node-Link) .

- As we were extracting prefix paths in bottom-up strategy, we gather the suffix paths in top-down projection, so get all the suffix paths from node , here , {I1,I5:1}, {I1,I3,I5:1},{I1,I4:1},{I3:2},{I4:1} , Notice that we missed some information here , what's it ? See the support count of I1 and I3 comes out to be 3 while it should have been 4, conditioned on item I2 . So what did we miss ?
  Suffix paths necessarily do not mean traversing from the root to the leaves only, there might have been a transaction ending at a node in the FP-tree which lost its leaf property because of a later transaction . In this case, I2-I1-I3 could have been an earlier transaction and I2-I1-I3-I5 could have been a later transaction due to which node with item I3 (created because of transaction I2-I1-I3, let's call this node PL) lost its leaf property .
  So how to deal with this ? Check for each node in the subtree if its count > 0 or not, if > 0 traverse to that node and as soon as a path is traversed, subtract the count of leaf node from the counts of all the nodes in the path . So the count of node PL was initially 2 here, as the path I2-I1-I3-I5 was traversed, the count of node associated with I5, i.e, 1 was subtracted from all the nodes in the path, so the count of node PL became 1 . So a traversal had to be made to node PL to generate a suffix path .
  Hence conditioned on item I2, these are the suffix paths : {I1,I5:1}, {I1,I3,I5:1},{I1,I4:1},{I3:2},{I4:1},{I1,I3:1} . Notice the generation of {I1,I3:1} because of the extra handling .
  This way the conditional pattern base is produced .
- The rest of the process, generating FP-tree and then the extraction for frequent itemsets is the same as observed in bottom-up generation technique .

Merging Strategy :

- In bottom-up generation observe for item I5, as we traversed the path I2-I1-I3-I5, we added to the condition pattern base of I5, {I2,I1,I3:1}, but during this traversal, we are also sure that for item 3, this path will again be traversed (except the node for I5) .
- Maintaining a full path of nodes like this, where each node resembles a structure something like this :

```
class node:
    def __init__(self, word, word_count=0, parent=None, link=None):
        self.word=word
        self.word_count=word_count
        self.parent=parent
        self.link=link
        self.children={}
```

  consumes a lot of space .
- One can hence delete the part of the path that becomes redundant after the traversal and rather push into item I3's conditional pattern base {I2,I1 : 2} . This is expected to

somewhat (minor) speed up the algorithm and the deletion of the redundant path will save up some space .

Apriori Generation :

Partitioning :

- Suppose you want to generate frequent itemsets of size k (let's call them k-itemsets) and desired minimum support count is 10 .
- Now suppose you partition the entire dataset into 5 parts and run individual aprio-gen on them with minimum support count = 10/5 = 2. Local frequent itemsets are obtained, let's say L1, L2, L3, L4, L5 .
- We make a claim that any itemset that was supposed to be a global frequent itemset (that is when looked at the entire dataset) should have occurred in at least one of the sets for local frequent itemsets, i.e., in L1,L2,L3,L4 or L5. This can be proved by contradiction . If a supposedly global frequent itemset didn't occur in any of those sets Li, then its total support is necessarily < 10, (as their counts in each of the parts < 2 and hence overall less than 10) .
- Now we are sure the global frequent itemsets are a subset of union(Li) . So for each of the itemset in union(Li), check if their count across the entire dataset is greater than equal to 10 or not . If greater than equal to 10, it is a global frequent itemset .

Transaction Reduction :

- Apriori generation follows the way L1 -> C2 -> L2 -> C3 -> L3 -> C4 -> L4 … Now suppose a transaction Di in the dataset doesn't contain any of the itemsets in L3 , can it contain any itemset in L4 ? Certainly not, so you can remove this transaction for sure after the Database scan for L3 .

Hash based Technique :

Best way to understand this is the example in Han, Kimber , Pei .

Create hash table H_2 using hash function $h(x, y) = ((order\ of\ x) \times 10 + (order\ of\ y)) \bmod 7$

H_2

bucket address	0	1	2	3	4	5	6
bucket count	2	2	4	2	2	4	4
bucket contents	{I1, I4} {I3, I5}	{I1, I5} {I1, I5}	{I2, I3} {I2, I3} {I2, I3} {I2, I3}	{I2, I4} {I2, I4}	{I2, I5} {I2, I5}	{I1, I2} {I1, I2} {I1, I2} {I1, I2}	{I1, I3} {I1, I3} {I1, I3} {I1, I3}

Hash table, H_2, for candidate 2-itemsets. This hash table was generated by scanning Table 6.1's transactions while determining L_1. If the minimum support count is, say, 3, then the itemsets in buckets 0, 1, 3, and 4 cannot be frequent and so they should not be included in C_2.