A pattern growth Approach for Mining Frequent Itemsets

Pattern Growth Approach

- Suffer two nontrivial costs:
  - Generation of huge number of candidate sets.
  - Need to repeatedly scan the database and check large set of candidates by pattern matching
- Needs a method that mines the complete set of frequent itemsets without costly candidate generation process
- Frequent pattern growth adopts divide-and-conquer strategy

Frequent Pattern -growth

- Encompasses the database representing frequent itemsets into FP-tree
- FP-tree retains the item association information.
- Divides the compressed db into set of conditional databases
  - Each consists of one frequent item or pattern fragment
  - For each pattern fragment only its associated data sets need to be examined
  - Approach reduce the size of data set to be searched along with growth of patterns being examined.

FP-Growth without candidate generation

- Scan and derive the set of frequent itemsets(1-frequent) and their support counts.
- The frequent itemsets are sorted in descending order of support count
- Resulting set is denoted by L
- L= {{I2:7},{I1:6},{I3:6},{I4:2},
- {I5:2}}

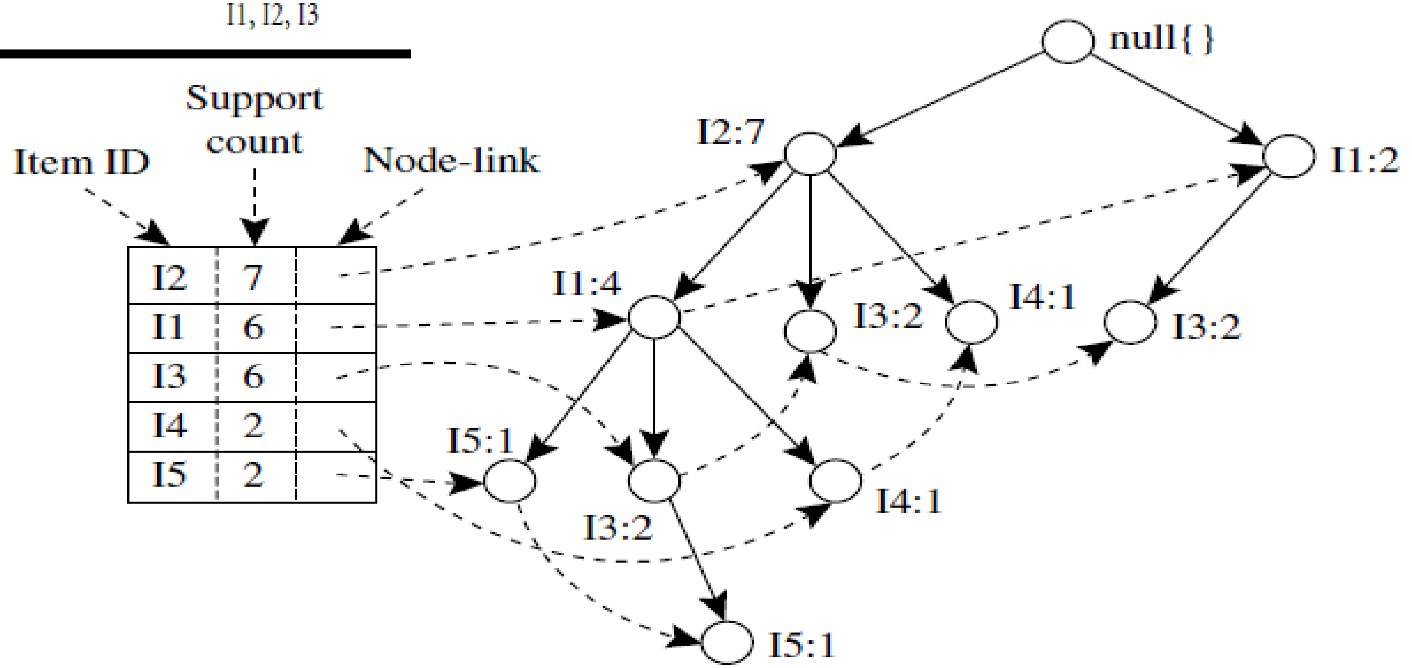
TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

FP-Growth without candidate generation

- FP tree is constructed as follows:
  - Create the root of the tree, labeled with NULL
  - Process the transactions in L order
  - Create a branch for each transactions
  - The items in the transactions acts a node in the branches
- Eg: T100:I1,I2,I5 & I2,I1,I5 in L order
  - Construct the first branch with three nodes <I2:1>, <I1:1>, <I5:1>,
  - Connect I2 to the root and I1 to I2 and I5 to I1
- The next transactions T200 contains L{I2,I4}, connect I2 to the root and attach l4 to I2
- The transaction T200 shares a common prefix I2 with T100 so increment the count of the node I2 by 1.

FP-Growth without candidate generation

TID	List of item_IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3



FP-Growth without candidate generation

- When branch to be added for a transaction
  - The node of the common prefix is incremented by 1
  - Nodes of 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**

FP-Tree mining

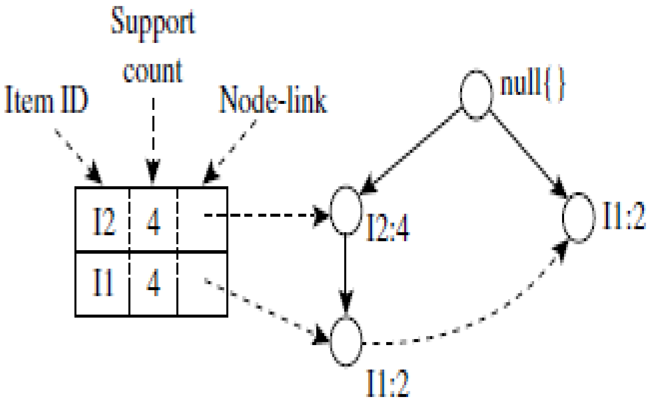
- **Algorithm**
  - Start from each frequent length-1 pattern(**suffix pattern**)
  - Construct **conditional pattern base** (sub database consists set of prefix paths in the FP tree co-occurring with the suffix pattern)
  - Construct **conditional FP-tree** using minimum support and perform mining recursively.
  - **Pattern growth** achieved by concatenation of the suffix pattern with frequent patterns generated from conditional FP-tree

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}

Analysis

- Analysis for the Suffix I3:
  - Two prefix paths <I2,I1:2> , <I2:2>and <I1:2> forms conditional data base
  - Conditional FP-tree has two branches <I2:4,I1:2> and <I1:2>
  - Combination of frequent set of patterns:{I2 ,I3:4}, {I1,I3:4},{I2,I1,I3:2}



FP-Growth Algorithm

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*, 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 $\beta \cup \alpha$ with *support_count* = *minimum support count of nodes in* β;
(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 β's conditional pattern base and then β's conditional FP_tree $Tree_\beta$;
(7)   **if** $Tree_\beta \neq \emptyset$ **then**
(8)   call FP_growth($Tree_\beta$, β); }

FP-Tree mining

- The FP-growth method transforms the problem of finding long frequent patterns into searching for shorter ones in much smaller conditional databases recursively.
- Method reduces the search costs.
- Efficient and scalable for mining both long and short frequent patterns.
- Faster than Apriori algorithm.