Mining Frequent Patterns Without Candidate Generation

Authors:

Jiawei Han
Jian Pei
Yiwen Yin

Contents

Introduction to the problem

Key Contribution

Relevant Prior Work

Methodology

Results

Opinion of the paper

Problem

Bottleneck of Aprriori: candidate generation

- **Huge candidate sets:**

  -10^4 frequent 1-itemsets will generate more than 10^7 candidate 2-itemsets

  -To discover a frequent pattern of size 100, e.g., $\{a_1, a_2, \ldots\ldots, a_{100}\}$, one needs to generate $2^{100} \sim 10^{30}$ candidates.

- **Multiple scans of the database**

Key contributions

- A novel compact data structure, called frequent pattern tree.
- FP- tree- based pattern fragment growth mining method.
- Search technique is a portioning-based divide and conquers method.

Relevant Prior Work

•Transaction Reduction: A transaction that does not contain any frequent k-itemsets cannot contain any frequent (k+1) itemsets. Therefore such a transaction can be removed.

•Partitioning the database: The partitioning technique requires just two scans to mine the frequent itemsets.

Divide database into non-overlapping partitions. Get itemsets with support greater than minimum support for each partition.

Frequent Pattern Tree

- Frequent Pattern Tree consists of one root labeled as null, a set of item prefix sub trees as the children of the root, and a frequent –item header table.

- Each node in the item prefix sub tree consists of three fields: item-name, count and node link where--- item-name registers which item the node represents, count registers the number of transactions represented by the portion of path reaching this node, node link links to the next node in the FP- tree.

- Each entry in the frequent-item header table consists of two fields item name and head of node link, which points to the first node in the FP-tree carrying the item-name.

Algorithm 1(FP-tree construction)

1) Scan the transaction database DB once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, the list of frequent items.

2) Create the root of an FP-tree, T, and label it as null. For each transaction in the database do the following.

- Select and sort the frequent item in each transaction according to the order of L Let the sorted frequent item list be [p|P], where p is the first element and P is the remaining list. Call insert for each item insert ([p|P], T).

- insert function

insert ([p|P], T)

{

// Check if T has a child N where N.item-name=p.item-

name then increment N count by 1.

//else create a new node with count 1,its parent linked to T,

and its node-link be linked to nodes with the same item-

name via node-link structure

//call insert till P is non-empty.

}

Cost analysis of FP-tree construction O (|no of frequent

items in Transaction|).

Now our FP tree has the complete information for frequent

pattern mining.

Example

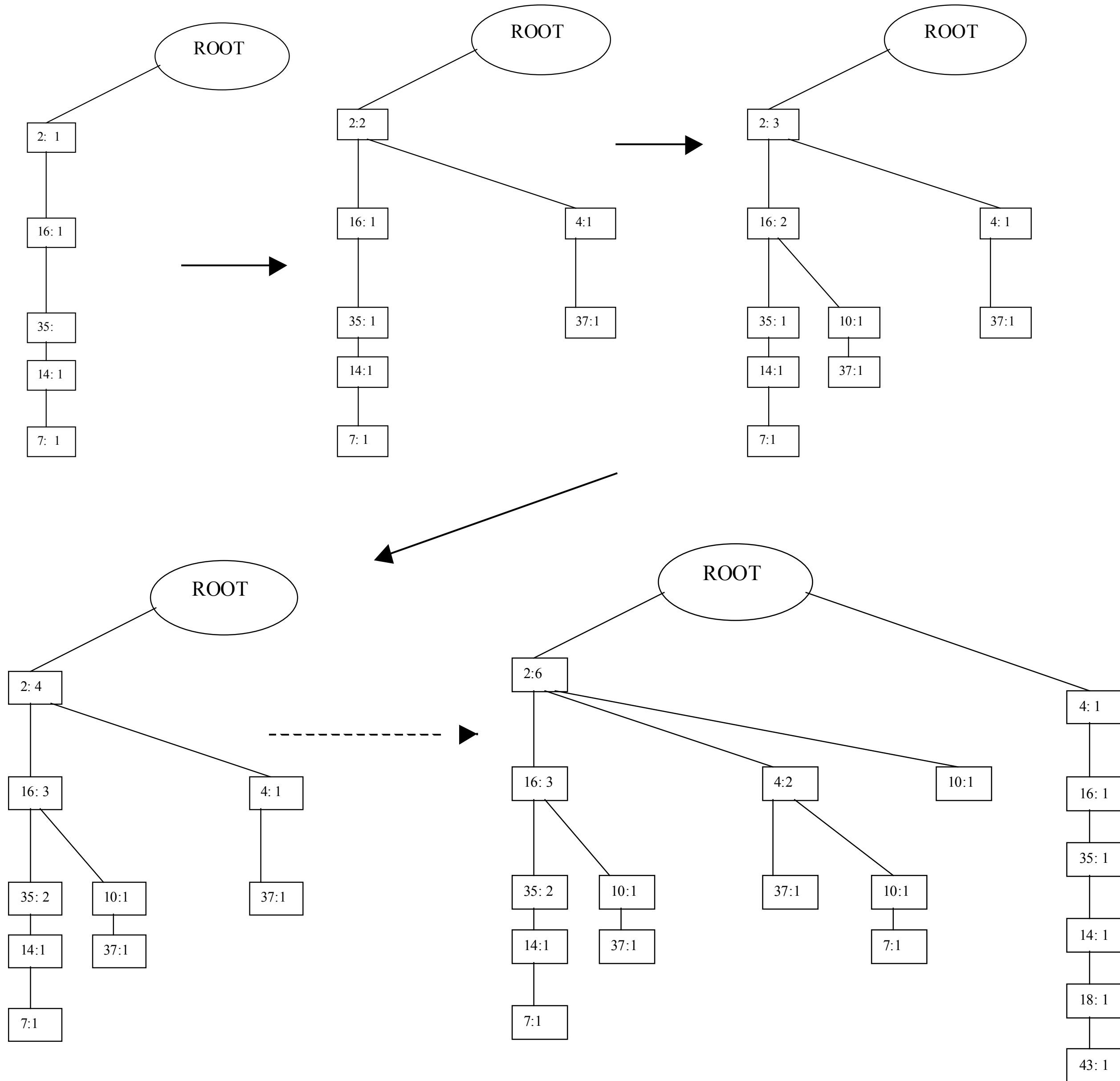
Transaction Example (from assignment #2, we picked 10 transactions with minimum support of 3)

TID	Items in Basket	(Ordered) Frequent Items
100	**2** **7** 11 13 **14** **16** 31 **35** 36	2 16 35 14 7
200	**2** **4** 20 23 28 **37** 44 56 60	2 4 37
300	**2** **10** **16** 20 23 26 33 **37** 72	2 16 10 37
400	**2** **16** 24 26 27 30 33 **35** 51	2 16 35
500	**2** **4** **7** 9 **10** 17 51 56 87	2 4 10 7
600	**2** **10** 11 21 24 27 29 40 45	2 10
700	3 **4** 11 **14** **16** **18** 29 **35** **43**	4 16 35 14 18 43
800	3 **7** **18** 19 21 25 26 32 36	18 7
900	**4** 13 **14** **18** **37** 40 **43** 50 67	4 14 18 37 43
1000	**4** **10** 31 32 **35** **43** 45 51 65	4 10 35 43

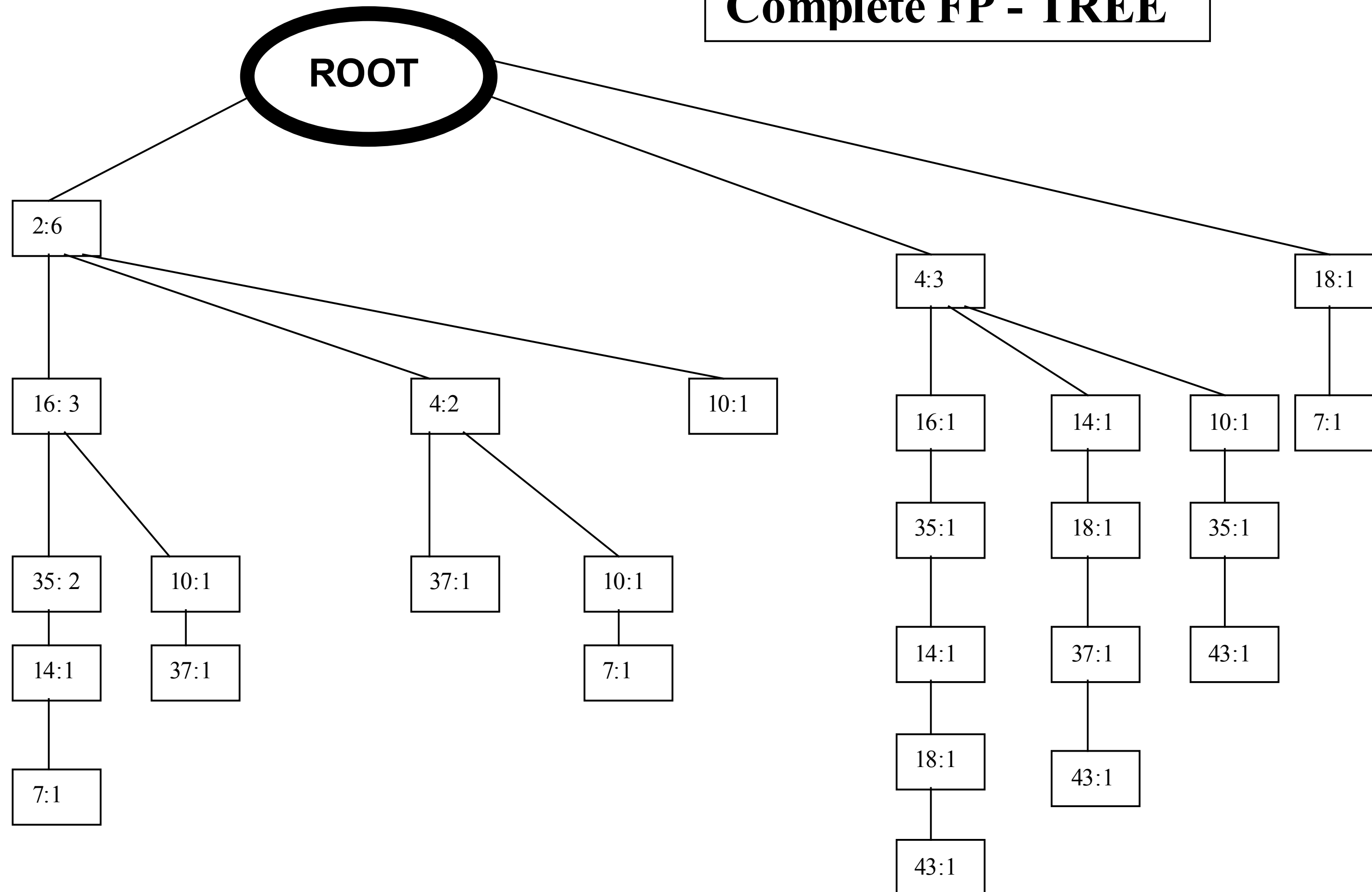
Frequency Count of items (by id):

Item ID	frequency count
2	*6*
4	*5*
16	*4*
10	*4*
35	*4*
14	*3*
18	*3*
37	*3*
43	*3*
7	*3*

FP-Tree Generation



Complete FP - TREE



Mininig Frequent Patterns using FP-tree

Explore Compact information stored in FP-tree and develop complete

set of frequent pattern.

Algorithm 2

Input: constructed FP-tree

Output: complete set of frequent patterns

Method: Call FP-growth(FP-tree, null).

procedure FP-growth(Tree,α)

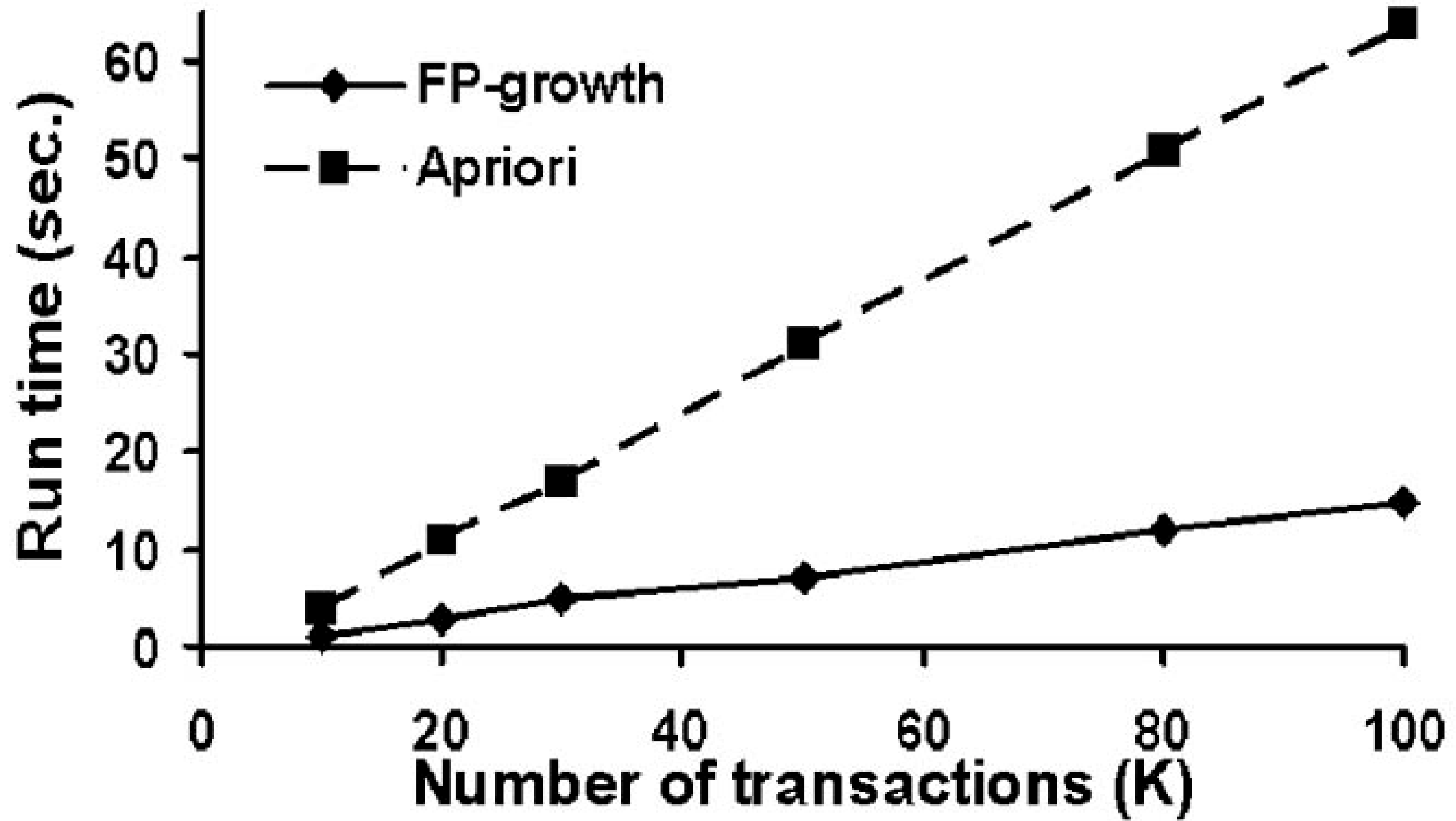
{

if Tree contains a single path P
then for each combination do generate pattern $\beta \cup \alpha$ with support = minimum support of nodes in β.
Else
For each header a_i in the header of Tree do{
Generate pattern $\beta = a_i \cup \alpha$ with support = $a_{i.}$.support;
Construct β's conditional pattern base and then β's conditional FP TreeTree $_\beta$
If Tree $_\beta$ <> null
Then call FP-growth(Tree $_\beta$, $\beta_,$)}

}

ITEM	Conditional patterns base	Conditional FP-Tree	frequent patterns generated
7	{ (2:*1*, 16:*1*, 35:*1*, 14:*1)*, (2:*1*, 4:*1*, 10:*1)*, (18:*1)* }	{ (2: *2*) }| 7	2 7 : *2*
43	{ (4:*1*,16:*1*, 35:*1*, 14:*1*, 18:*1)*, (4:*1*, 14:*1,* 18:*1*, 37:*1)*, (4:*1*, 10:*1*, 35:*1)* }	{ (4:*1*, 35:*1*, 14:*1*, 18:*1*), (4:*1*, 14:*1*, 18:*1*), (4:*1*, 35:*1*) } | 43	4 43:2, 35 43:2, 14 43 :2, 18 43:2, 4 35 43:2, 4 14 43:2, 4 18 43:2, 14 18 43:2, 4 14 18 43 :2
37	{2:*1*, 16:*1*, 10:*1*}, {2:*1*, 4:*1*}, {4:*1*, 14:*1*, 18:*1*}	{ (2: *2,* 4:*1*), (4:*1*) } | 37	2 37:*2*, 4 37:*2*
18	{4:*1*, 16:*1*, 35:*1*, 14:*1*}, {4:*1*, 14:*1*}	{ (4:*2,* 14:2) } | 18	4 18:2, 14 18:*2*, 4 14 18:*2*
14	{2:*1*, 16:*1*, 15:*1*}, {4:*1*, 16:*1*, 35:*1*}, {4:*1*}	{ (4: *2*) } | 14	4 14:*2*
35	{2:*2*, 16:*2*}, {4:*1*, 16:*1*} , {4:*1*, 10:*1*}	{ (2:*2*, 16:*2*), (4:*2*) } | 35	2 35:*2*, 16 35:*2*, 4 35:2, 2 16 35:*2*
10	{2:*1*, 16:*1*}, {2:*1*, 4:*1*}, {4:*1*}	{ (2:*2,* 4:*1)*, (4:*1*) } | 10	2 10:*2*, 4 10:*2*
16	{2:*3*}, {4:*1*}	{ (2:3) } | 16	2 16:*3*
4	{2:*2*}	{2:*2*} | 4	4, 2 : *2*
2	0	0	------

Comparison of FP-growth **and** Apiori



Opinion

- Compare to Apriori-like algorithm, it is difficult to implement FP-Tree on actual coding because of the tree structure.

- In case of large database, it is a good candidate to use for short and long patterns because FP-growth scales much better than Apriori. It becomes very obvious when the "support threshold" goes down.

- FP-tree is constructed the way that the higher frequent items are closer to the root (upper portion). Therefore, from a "searching" or "scanning" point of view, it is very easy to select (mining) the items with high threshold by dropping the lower portion of the tree.