




Mining Frequent Patterns without Candidate Generation

a paper by Jiawei Han, Jian Pei and Yiyen Yin
School of Computing Science
Simon Fraser University
Presented by Maria Cutumisu
Department of Computing Science
University of Alberta



This paper proposes:

- A novel frequent pattern tree structure: FP-tree
- An efficient FP-tree-based mining method: FP-growth



This approach is very efficient due to:

- Compression of a large database into a smaller data structure
- Pattern fragment growth mining method
- Partitioning-based divide-and-conquer search method



FP-tree: Design and Construction

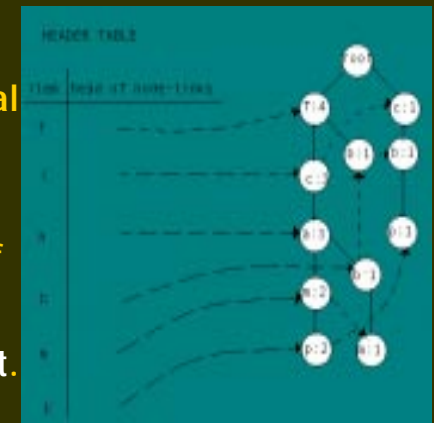
- To ensure that the tree structure is compact, only frequent length-1 items will have nodes in the tree
- More frequently occurring nodes will have better chances of sharing nodes than the others

Example: a transaction database

Transaction ID	Items Bought	(Ordered) Frequent Items
100	f, a, c, d, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

The corresponding FP-tree

Transactions sharing an identical itemset can be merged into one with the number of occurrences registered as count.



An FP-tree is a tree structure which consists of:

- One root labeled as "null"
- A set of item prefix sub-trees with each node formed by three fields: item-name, count, node-link
- A frequent-item header table with two fields for each entry: item-name, head of node-link


FP-tree construction algorithm

- Input: a transaction database DB and a minimum support threshold ϵ
- Output: Its frequent pattern tree, FP-tree
- Method: The FP-tree is constructed in the following steps:



1. Scan 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 a root of an FP-tree, T , and label it as "null"

- For each transaction $Trans$ in DB 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)`



Note: `insert_tree([p | P], T)` 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 by 1, its parent link be linked to T , and its node-link be linked to the nodes with the same $item_name$ via the node-link structure
- IF P is nonempty, call `insert_tree(P, N)` recursively



Analysis

- Two scans of the DB are necessary: the first collects the set of frequent items and the second constructs the FP-tree.
- The cost of inserting a transaction $Trans$ into the FP-tree is $O(|Trans|)$, where $|Trans|$ is the number of frequent items in $Trans$.



- FP-tree contains the complete information for frequent pattern mining.
- The size of the FP-tree is bounded by the size of the database, but due to frequent items sharing, the size of the tree is usually much smaller than its original database.
- High compaction is achieved by placing more frequently items closer to the root (being thus more likely to be shared).



FP-growth: the FP-tree-based mining method

- Starts from a frequent length-1 pattern
- Examines only its conditional pattern base
- Constructs its FP-tree
- Performs mining recursively on the tree




FP-growth algorithm

- **Input:** FP-tree constructed using DB and a minimum support threshold ϵ
- **Output:** The 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 β of the nodes in the path P DO generate pattern $\beta \cup \alpha$ with support = minimum support of nodes in β
 - ELSE for each a_i in the header of Tree DO
 - generate pattern $\beta = a_i \cup \alpha$ with a_i .support;
 - construct β 's conditional pattern base and FP-tree $Tree\beta$
 - IF $Tree\beta \neq \text{void}$ THEN Call FP-growth($Tree\beta$, β)




Analysis of the FP-growth algorithm

- Finds the complete set of frequent itemsets
- Efficient because:
 - it works on a reduced set of pattern bases
 - it performs mining operations less costly than generation and test:
 - prefix count adjustment
 - counting
 - pattern fragment concatenation



Search technique: partitioning-based divide-and-conquer


- Used instead of the Apriori-like bottom-up generation of frequent itemsets combinations
- Reduces the size of the conditional pattern base generated at the subsequent level of search and of its corresponding FP-tree

- 
-
- Transforms the problem of finding long frequent patterns to looking for shorter ones and then concatenating the suffix.
 - Employs the least frequent items as suffix, which offers a good selectivity.



Performance comparison with other algorithms

- TreeProjection is the supporting algorithm of another novel tree structure: lexicographic tree
- Comparative analysis of the FP-growth with Apriori and TreeProjection algorithms show that FP-growth outperforms both of them




Improvements: how to design a disk-resident FP-tree

- Cluster FP-tree nodes by path and by item prefix sub-tree
- B+-tree for FP-tree not fitting into main memory
- Group access mode mining to reduce the I/O cost
- Release space of the conditional pattern base or conditional FP-tree after usage
- Remove the node-links of the FP-tree



Performance improvements

- Materialization of an FP-tree
- Incremental updates of an FP-tree
- FP-tree mining with item constraints
- FP-tree mining of other frequent patterns



Advantages of the FP-growth mining method:

- Efficient and scalable for both long and short frequent patterns; the running memory requirements of FP-growth increase linearly when the support threshold goes down
- An order of magnitude faster than the Apriori algorithm
- Faster than recently reported new frequent pattern mining methods



Drawbacks:

- The tree does not achieve maximal compactness all the time.
- For the databases with mostly short transactions, the reduction ratio of the tree in respect to the database is not very high.
- The FP-tree does not always fit into the main memory.



Conclusions

- FP-growth method has satisfactory performance when tested in large industrial databases
- It is open to a lot of research issues
- Due to compression, sometimes large databases (order of gigabytes) containing many long patterns may generate FP-trees which fit in main memory