Data Mining for Knowledge Management

Association Rules

Themis Palpanas
University of Trento
http://disi.unitn.eu/~themis

Thanks for slides to:

- Jiawei Han
- George Kollios
- Zhenyu Lu
- Osmar R. Zaïane
- Mohammad El-Hajj
- Yu-ting Kung

Frequent Pattern Mining

Given a transaction database DB and a minimum support threshold ξ, find all frequent patterns (item sets) with support no less than ξ.

Input: DB:

TID	*Items bought*
100	{*f, a, c, d, g, i, m, p*}
200	{*a, b, c, f, l, m, o*}
300	{*b, f, h, j, o*}
400	{*b, c, k, s, p*}
500	{*a, f, c, e, l, p, m, n*}

Minimum support: ξ =3

Output: all frequent patterns, i.e., *f, a, …, fa, fac, fam, …*

Problem: How to **efficiently** find all frequent patterns?

Apriori

- The core of the Apriori algorithm:
  - Use frequent (k – 1)-itemsets (L_{k-1}) to generate candidates of frequent k-itemsets C_k
  - Scan database and count each pattern in C_k , get frequent k-itemsets (L_k) .
- E.g.,

TID	*Items bought*
100	{*f, a, c, d, g, i, m, p*}
200	{*a, b, c, f, l, m, o*}
300	{*b, f, h, j, o*}
400	{*b, c, k, s, p*}
500	{*a, f, c, e, l, p, m, n*}

Apriori iteration

C1	*f,a,c,d,g,i,m,p,l,o,h,j,k,s,b,e,n*
L1	*f, a, c, m, b, p*
C2	*fa, fc, fm, fp, ac, am, …bp*
L2	*fa, fc, fm, …*
…	

Performance Bottlenecks of Apriori

- The bottleneck of *Apriori*: candidate generation
  - Huge candidate sets:
    - 10^4 frequent 1-itemset will generate 10^7 candidate 2-itemsets
    - To discover a frequent pattern of size 100, e.g., $\{a_1, a_2, \ldots, a_{100}\}$, one needs to generate $2^{100} \approx 10^{30}$ candidates.
  - Multiple scans of database: each candidate

Ideas

- Compress a large database into a compact, *Frequent-Pattern tree* (FP-tree) structure
  - highly condensed, but complete for frequent pattern mining
  - avoid costly database scans
- Develop an efficient, FP-tree-based frequent pattern mining method (FP-growth)
  - A divide-and-conquer methodology: decompose mining tasks into smaller ones
  - Avoid candidate generation: sub-database test only.

Mining Frequent Patterns Without Candidate Generation

- Grow long patterns from short ones using local frequent items
  - "abc" is a frequent pattern
  - Get all transactions having "abc": DB|abc
  - "d" is a local frequent item in DB|abc → abcd is a frequent pattern

Mining Frequent Patterns Without Candidate Generation

FP-tree Construction from a Transactional DB

min_support = 3

TID	Items bought
100	{f, a, c, d, g, i, m, p}
200	{a, b, c, f, l, m, o}
300	{b, f, h, j, o}
400	{b, c, k, s, p}
500	{a, f, c, e, l, p, m, n}

Steps:

FP-tree Construction from a Transactional DB

min_support = 3

TID	Items bought
100	{f, a, c, d, g, i, m, p}
200	{a, b, c, f, l, m, o}
300	{b, f, h, j, o}
400	{b, c, k, s, p}
500	{a, f, c, e, l, p, m, n}

Steps:

1. Scan DB once, find frequent 1-itemsets (single item patterns)

FP-tree Construction from a Transactional DB

min_support = 3

TID	Items bought
100	{f, a, c, d, g, i, m, p}
200	{a, b, c, f, l, m, o}
300	{b, f, h, j, o}
400	{b, c, k, s, p}
500	{a, f, c, e, l, p, m, n}

Item	***frequency***
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*

Steps:

1. Scan DB once, find frequent 1-itemsets (single item patterns)

FP-tree Construction from a Transactional DB

min_support = 3

TID	Items bought
100	{f, a, c, d, g, i, m, p}
200	{a, b, c, f, l, m, o}
300	{b, f, h, j, o}
400	{b, c, k, s, p}
500	{a, f, c, e, l, p, m, n}

Item	***frequency***
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*

Steps:

1. Scan DB once, find frequent 1-itemsets (single item patterns)
2. Order frequent items in descending order of their frequency

FP-tree Construction from a Transactional DB

min_support = 3

TID	Items bought	(ordered) frequent items
100	{f, a, c, d, g, 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}

Item	***frequency***
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*

Steps:

1. Scan DB once, find frequent 1-itemsets (single item patterns)
2. Order frequent items in descending order of their frequency

FP-tree Construction from a Transactional DB

min_support = 3

TID	Items bought	(ordered) frequent items
100	{f, a, c, d, g, 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}

Item	***frequency***
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*

Steps:

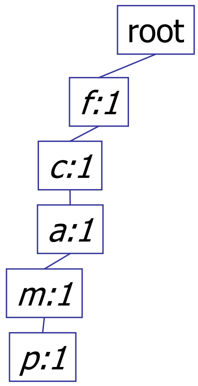
1. Scan DB once, find frequent 1-itemsets (single item patterns)
2. Order frequent items in descending order of their frequency
3. Scan DB again, construct FP-tree

FP-tree Construction

TID	freq. Items bought
~~100~~	~~{f, c, a, m, p}~~
200	{f, c, a, b, m}
300	{f, b}
400	{c, p, b}
500	{f, c, a, m, p}

min_support = 3

Item	***frequency***
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*

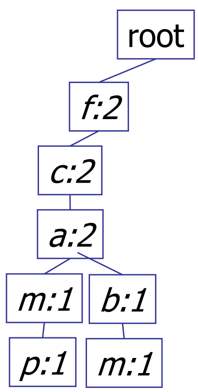


FP-tree Construction

TID	freq. Items bought
~~100~~	~~{f, c, a, m, p}~~
~~200~~	~~{f, c, a, b, m}~~
300	{f, b}
400	{c, p, b}
500	{f, c, a, m, p}

min_support = 3

Item	***frequency***
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*

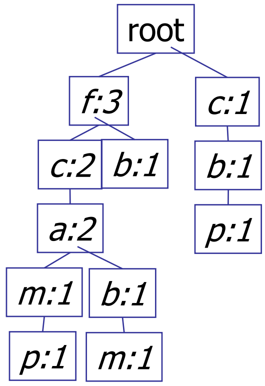


FP-tree Construction

TID	freq. Items bought
~~100~~	~~{f, c, a, m, p}~~
~~200~~	~~{f, c, a, b, m}~~
~~300~~	~~{f, b}~~
~~400~~	~~{c, p, b}~~
500	{f, c, a, m, p}

min_support = 3

Item	***frequency***
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*

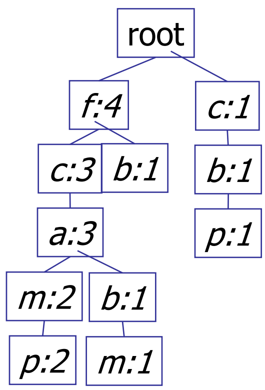


FP-tree Construction

TID	freq. Items bought
~~100~~	~~{f, c, a, m, p}~~
~~200~~	~~{f, c, a, b, m}~~
~~300~~	~~{f, b}~~
~~400~~	~~{c, p, b}~~
~~500~~	~~{f, c, a, m, p}~~

min_support = 3

Item	***frequency***
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*



FP-tree Construction

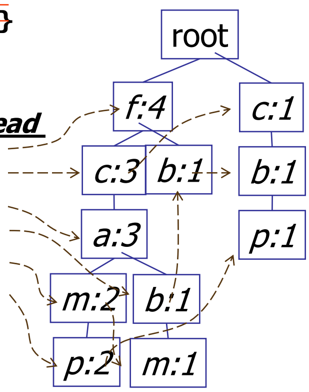
TID	freq. Items bought
~~100~~	~~{f, c, a, m, p}~~
~~200~~	~~{f, c, a, b, m}~~
~~300~~	~~{f, b}~~
~~400~~	~~{c, p, b}~~
~~500~~	~~{f, c, a, m, p}~~

min_support = 3

Item	*frequency*
f	*4*
c	*4*
a	*3*
b	*3*
m	*3*
p	*3*

Header Table

Item	*freq*	*head*
f	*4*	
c	*4*	
a	*3*	
b	*3*	
m	*3*	
p	*3*	



FP-Tree Definition

- FP-tree is **a frequent pattern tree**, defined below:
  - It consists of one root labeled as "null"
  - a set of *item prefix subtrees* as the children of the root, and a *frequent-item header table*.

FP-Tree Definition

- FP-tree is **a frequent pattern tree**, defined below:
  - It consists of one root labeled as "null"
  - a set of *item prefix subtrees* as the children of the root, and a *frequent-item header table*.
- Each node in the *item prefix subtrees* has three fields:
  - item-name to register which item this node represents,
  - count, the number of transactions represented by the portion of the path reaching this node, and
  - node-link that links to the next node in the FP-tree carrying the same item-name, or null if there is none.

FP-Tree Definition

- FP-tree is **a frequent pattern tree**, defined below:
  - It consists of one root labeled as "null"
  - a set of *item prefix subtrees* as the children of the root, and a *frequent-item header table*.
- Each node in the *item prefix subtrees* has three fields:
  - item-name to register which item this node represents,
  - count, the number of transactions represented by the portion of the path reaching this node, and
  - node-link that links to the next node in the FP-tree carrying the same item-name, or null if there is none.
- Each entry in the *frequent-item header table* has two fields,
  - item-name, and
  - head of node-link that points to the first node in the FP-tree carrying the item-name.

Benefits of the FP-tree Structure

- Completeness
  - Preserve complete information for frequent pattern mining
  - Never break a long pattern of any transaction
- Compactness
  - Reduce irrelevant info—infrequent items are gone
  - Items in frequency descending order: the more frequently occurring, the more likely to be shared
  - Never be larger than the original database (not count node-links and the *count* field)
  - For Connect-4 DB, compression ratio could be over 100

Partition Patterns and Databases

- Frequent patterns can be partitioned into subsets according to f-list
  - F-list=f-c-a-b-m-p
  - Patterns containing p
  - Patterns having m but no p
  - …
  - Patterns having c but no a nor b, m, p
  - Pattern f
- Completeness and non-redundency

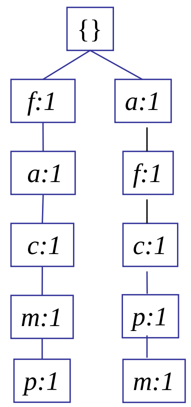
FP-Tree Design Choice

- Why items in FP-Tree in ordered descending order?

FP-Tree Design Choice

- Why items in FP-Tree in ordered descending order?
- Example 1:

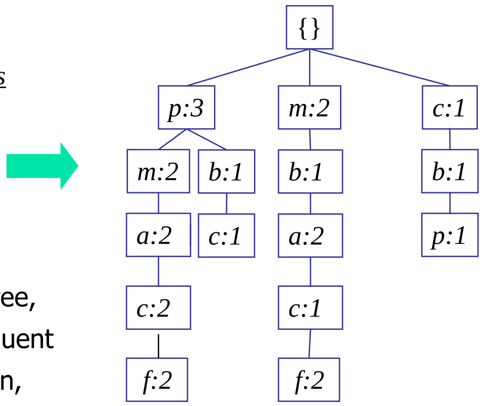
TID	*(unordered) frequent items*
100	{*f*, *a*, *c*, *m*, *p*}
500	{*a*, *f*, *c*, *p*, *m*}



FP-Tree Design Choice

- Example 2:

TID	*(ascended) frequent items*
100	{*p, m, a, c, f*}
200	{*m, b, a, c, f*}
300	{*b, f*}
400	{*p, b, c*}
500	{*p, m, a, c, f*}



- This tree is larger than FP-tree, because in FP-tree, more frequent items have a higher position, which makes branches less

Mining Frequent Patterns Using FP-tree: FP-Growth

- General idea (divide-and-conquer)
  Recursively grow frequent patterns using the FP-tree: looking for shorter ones recursively and then concatenating the suffix:
- Method
  - For each frequent item, construct its
    - conditional pattern base
    - then its conditional FP-tree
  - Repeat the process on each newly created conditional FP-tree until
    - the resulting FP-tree is empty
    - or it contains only one path (single path will generate all the combinations of its sub-paths, each of which is a frequent pattern)

Principles of FP-Growth

- Pattern growth property
  - Let α be a frequent itemset in DB, *CPB* be α's conditional pattern base, and β be an itemset in *CPB*. Then $\alpha \cup \beta$ is a frequent itemset in DB iff β is frequent in *CPB*.

- Is "*fcabm*" a frequent pattern?
  - "fcab" is a branch of m's conditional pattern base
  - "b" is **NOT** frequent in transactions containing "fcab"
  - "bm" is **NOT** a frequent itemset.

3 Major Steps

Starting the processing from the end of list L:

Step 1:

Construct conditional pattern base for each item in the header table

Step 2

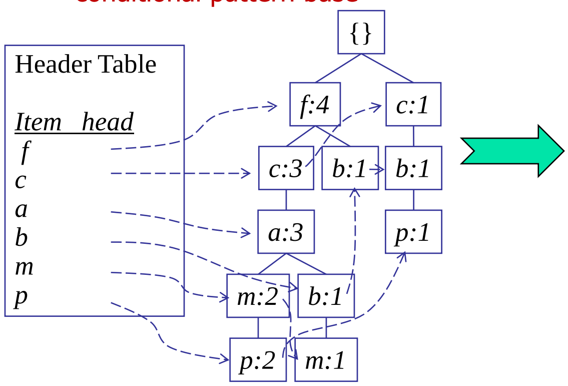
Construct conditional FP-tree from each conditional pattern base

Step 3

Recursively mine conditional FP-trees and grow frequent patterns obtained so far. If the conditional FP-tree contains a single path, simply enumerate all the patterns

Step 1: Construct Conditional Pattern Base

- Starting at the bottom of frequent-item header table in the FP-tree
- Traverse the FP-tree by following the link of each frequent item
- Accumulate all of transformed prefix paths of that item to form a conditional pattern base



***Conditional* pattern bases**

item	*cond. pattern base*
p	***fcam:2, cb:1***
m	***fca:2, fcab:1***
b	***fca:1, f:1, c:1***
a	***fc:3***
c	***f:3***
f	**{ }**

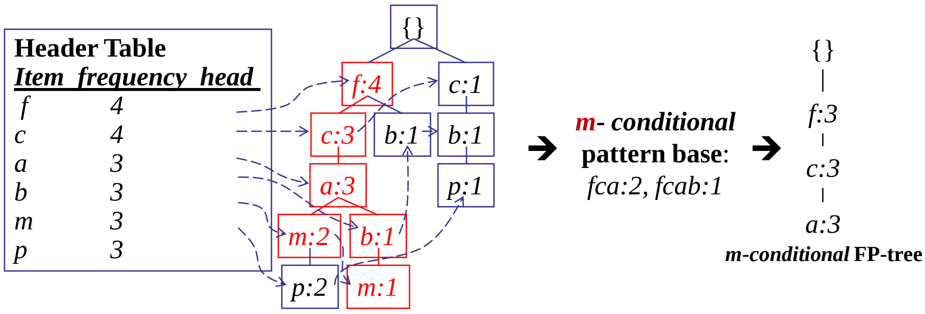
Properties of Step 1

- Node-link property
  - For any frequent item a_i, all the possible frequent patterns that contain a_i can be obtained by following a_i's node-links, starting from a_i's head in the FP-tree header.
- Prefix path property
  - To calculate the frequent patterns for a node a_i in a path P, only the prefix sub-path of a_i in P need to be accumulated, and its frequency count should carry the same count as node a_i.

Step 2: Construct Conditional FP-tree

- For each pattern base
  - Accumulate the count for each item in the base
  - Construct the conditional FP-tree for the frequent items of the pattern base

min support = 3

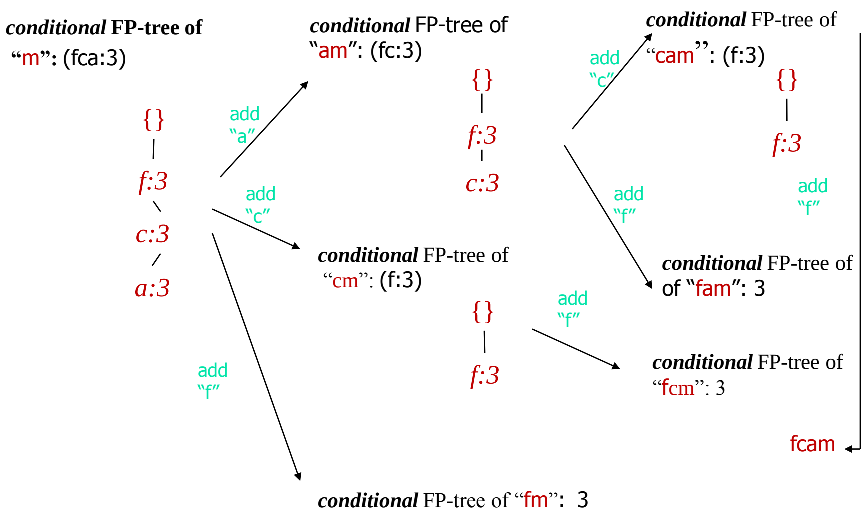


Conditional Pattern Bases and Conditional FP-Tree

Item	Conditional pattern base	Conditional FP-tree
p	{(fcam:2), (cb:1)}	{(c:3)}|p
m	{(fca:2), (fcab:1)}	{(f:3, c:3, a:3)}|m
b	{(fca:1), (f:1), (c:1)}	Empty
a	{(fc:3)}	{(f:3, c:3)}|a
c	{(f:3)}	{(f:3)}|c
f	Empty	Empty

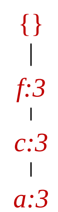
order of L

Step 3: Recursively mine the conditional FP-tree



Single FP-tree Path Generation

- Suppose an FP-tree T has a single path P. The complete set of frequent pattern of T can be generated by enumeration of all the combinations of the sub-paths of P



m-conditional FP-tree

All frequent patterns concerning *m*: combination of {f, c, a} and *m*

m,

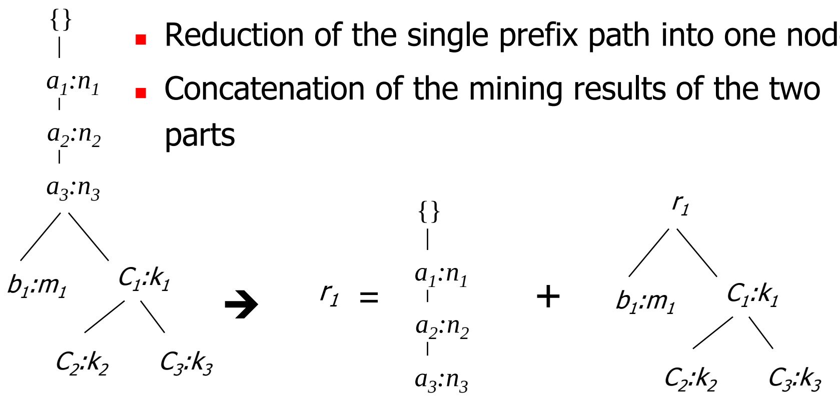
fm, *cm*, *am*,

fcm, *fam*, *cam*,

fcam

A Special Case: Single Prefix Path in FP-tree

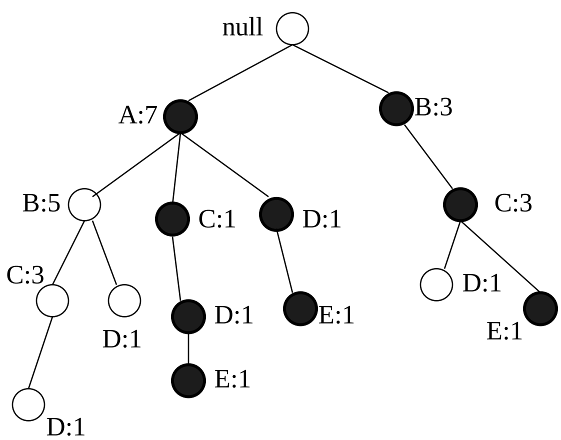
- Suppose a (conditional) FP-tree T has a shared single prefix-path P
- Mining can be decomposed into two parts
  - Reduction of the single prefix path into one node
  - Concatenation of the mining results of the two parts



FP-growth -- E

min support = 2

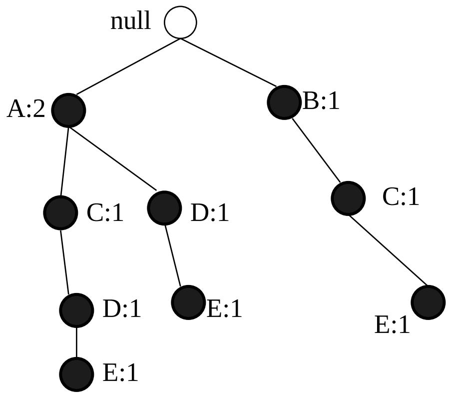
Build conditional pattern base for E:
P = {(A:1,C:1,D:1),
(A:1,D:1),
(B:1,C:1)}



FP-growth -- E

Build conditional pattern base for E:

Conditional tree for E:



min support = 2

Conditional Pattern base for E:

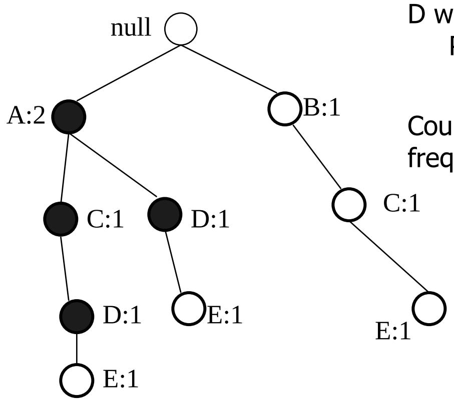
P = {(A:1,C:1,D:1,E:1),
(A:1,D:1,E:1),
(B:1,C:1,E:1)}

Count for E is 3: {E} is frequent itemset

Recursively apply FP-growth on P

FP-growth -- DE

Conditional tree for D within conditional tree for E:



min support = 2

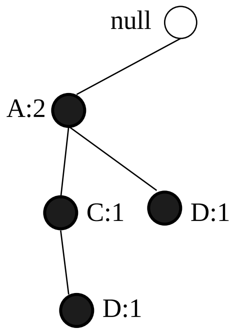
Build Conditional pattern base for D within conditional base for E:

P = {(A:1,C:1,D:1),
(A:1,D:1)}

Count for D is 2: {D,E} is frequent itemset

FP-growth -- DE

Conditional tree for D within conditional tree for E:



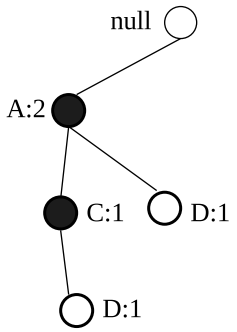
min support = 2

Build Conditional pattern base for D within conditional base for E:
P = {(A:1,C:1,D:1),
(A:1,D:1)}

Count for D is 2: {D,E} is frequent itemset

FP-growth -- CDE

Conditional tree for C within D within E:



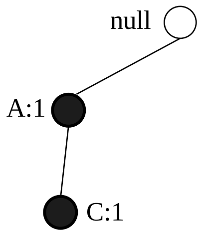
min support = 2

Build Conditional pattern base for C within D within E:
P = {(A:1,C:1)}

Count for C is 1: {C,D,E} is NOT frequent itemset

FP-growth -- CDE

Conditional tree for C within D within E:



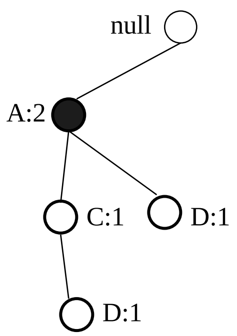
min support = 2

Build Conditional pattern base for C within D within E:
P = {(A:1,C:1)}

Count for C is 1: {C,D,E} is NOT frequent itemset

FP-growth -- ADE

Conditional tree for A within D within E:



min support = 2

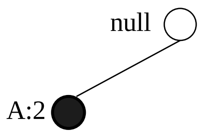
Count for A is 2: {A,D,E} is frequent itemset

FP-growth -- ADE

Conditional tree for A within D within E:

min support = 2

Count for A is 2: {A,D,E} is frequent itemset

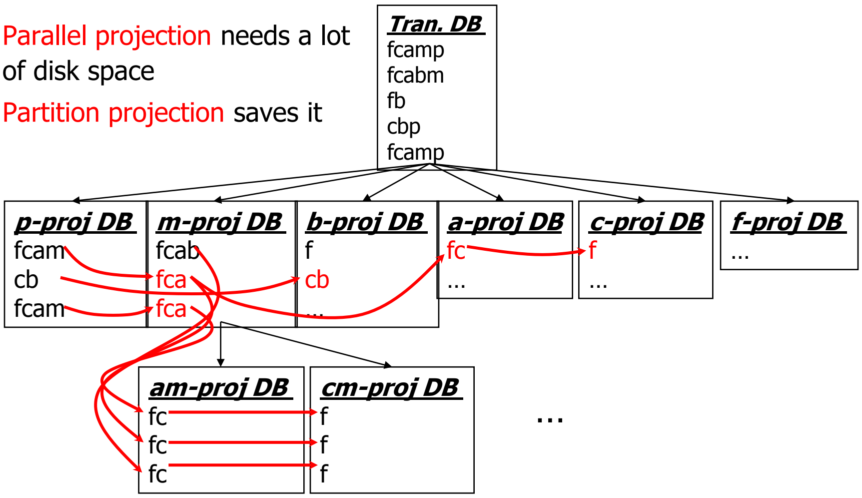


Scaling FP-growth by DB Projection

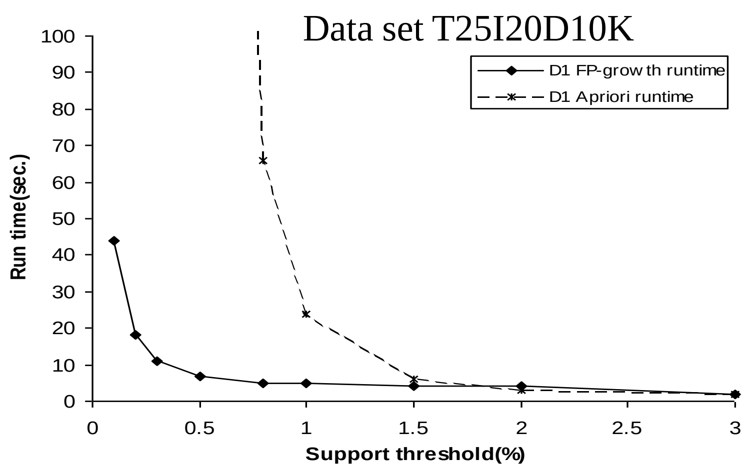
- FP-tree cannot fit in memory?—DB projection
- First partition a database into a set of projected DBs
- Then construct and mine FP-tree for each projected DB
- Parallel projection vs. Partition projection techniques
  - Parallel projection is space costly

Partition-based Projection

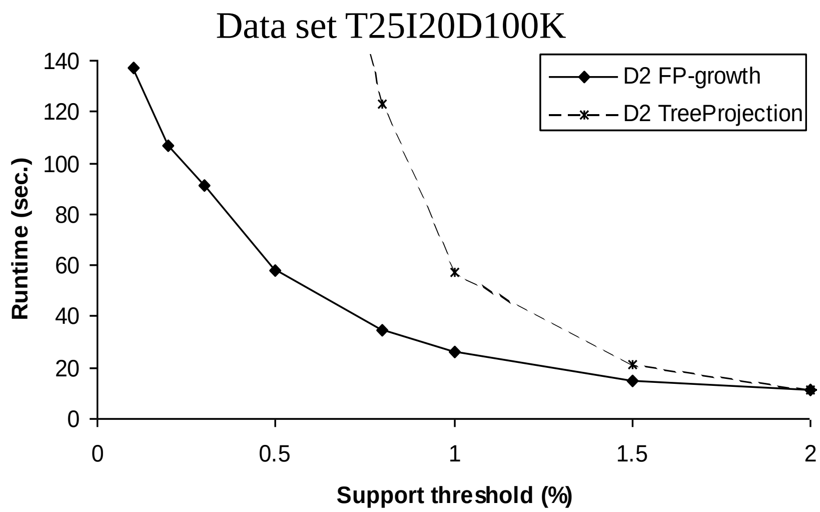
- Parallel projection needs a lot of disk space
- Partition projection saves it



FP-Growth vs. Apriori: Scalability With the Support Threshold



FP-Growth vs. Tree-Projection: Scalability with the Support Threshold



Why Is FP-Growth the Winner?

- Divide-and-conquer:
  - decompose both the mining task and DB according to the frequent patterns obtained so far
  - leads to focused search of smaller databases
- Other factors
  - no candidate generation, no candidate test
  - compressed database: FP-tree structure
  - no repeated scan of entire database
  - basic ops—counting local freq items and building sub FP-tree, no pattern search and matching

Implications of the Methodology

- Mining closed frequent itemsets and max-patterns
  - CLOSET (DMKD'00), CLOSET+ (KDD'03)
- Mining sequential patterns
  - FreeSpan (KDD'00), PrefixSpan (ICDE'01)
- Constraint-based mining of frequent patterns
  - Convertible constraints (KDD'00, ICDE'01)
- Computing iceberg data cubes with complex measures
  - H-tree and H-cubing algorithm (SIGMOD'01)

Implications of the Methodology

- Mining closed frequent itemsets
  - CLOSET+ (KDD'03)

Frequent Closed Itemsets

- **Definition**
  - An itemset Y is a **frequent closed itemset** if it is *frequent* and there exists no proper superset $Y' \supset Y$ such that *sup(Y') = sup(Y)*
  - Ex: min_sup = 2, f_list = <f:4, c:4, a:3, b:3, m:3, p:3>

Tid	Set of items	ordered frequent item list
100	a, c, f, m, p	f, c, a, m, p
200	a, c, d, f, m, p	f, c, a, m, p
300	a, b, c, f, g, m	f, c, a, b, m
400	b, f, i	f, b
500	b, c, n, p	c, b, p

fc -> frequent closed pattern? **correct answer: superset *fcam***

Pruning techniques (for closed itemsets)

- ***Item merging***
  - **Definition:**

    Let *X* be a *frequent itemset*. If *every* transaction containing itemset *X* also ***contains*** itemset *Y*, but ***not*** any *proper superset* of *Y*, then "$X \cup Y$" forms a **frequent closed itemset** and there is no need to search any itemset containing *X* but no *Y*

Pruning techniques (Cont.)

- ***Item merging***
  - ***Ex:***

Tid	Set of items	ordered frequent item list
100	a, c, f, m, p	f, c, a, m, p
200	a, c, d, f, m, p	f, c, a, m, p
300	a, b, c, f, g, m	f, c, a, b, m
400	b, f, i	f, b
500	b, c, n, p	c, b, p

Projected conditional database for prefix itemset *fc*:3 :

{(a,m,p), (a,m,p), (a,b,m)}→ ***am:3***

am is merged with *fc* ➔ a frequent closed itemset ***fcam:3***

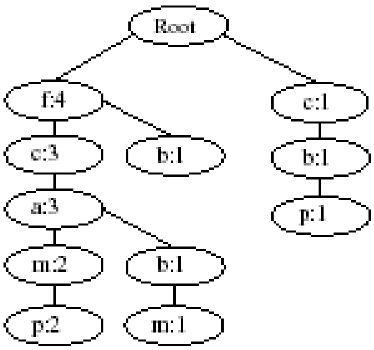
Pruning techniques (Cont.)

- ***Sub-itemset pruning***
  - **Definition**

    Let *X* be the frequent itemset currently under consideration. If *X* is a ***proper subset*** of an already found frequent closed itemset *Y* and ***sup(X) = sup(Y)***, then X and all of X's descendants in the tree cannot be frequent closed itemsets

Pruning techniques (Cont.)

- ***Sub-itemset pruning***
  - ***Ex:***



Already found closed itemset *fcam:3*

Now, mine the patterns with prefix itemset ca:3 :

→Identify *ca:3* is a ***proper subset*** of *fcam:3*

→ ***Stop mining*** the closed patterns with prefix *ca:3*

CLOSET+

- In CLOSET+,
  - A ***hybrid tree-projection*** method is developed, which builds conditional projected databases in two ways:
    - ***Dense*** datasets → ***bottom-up physical*** tree-projection
    - ***Sparse*** datasets → ***top-down pseudo*** tree-projection