Improving Apriori: General Ideas

- Shrink the number of candidates
  - Hash-based technique (DHP algorithm)
- Reduce the number of database scans on disk
  - Partitioning data (Partition algorithm)
- Avoid candidate generation
  - FP-growth

31

Shrink the Number of Candidates (*DHP*)

- *Hash-based technique* can be used to reduce the size of C_k, especially C_2
- Build a hash table when scanning DB to generate L_1.
  - For all 2-itemsets in each transaction, hash into the buckets and increase counts:

Hash function: $h(x, y) = ((id \text{ of } x) \times 10 + (id \text{ of } y)) \bmod 7$

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}

Only this array is stored

32

Shrink the Number of Candidates (*DHP*) (Cont'd)

- If the count of a bucket is less than minimum support count, the itemsets in the bucket cannot be frequent and thus are not put in C_2

The hashtable in the last slide was generated from this data set →

TID	items
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

33

Reduce the number of disk scans (*Partition*)

- Partition DB
  - Each partition is held in main memory
- Any itemset that is potentially frequent in DB must be frequent in at least one of the partitions of DB (can be proved)
  - Scan 1: partition database and for each partition find local frequent patterns
  - Scan 2: consolidate global frequent patterns
- A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association in large databases. In *VLDB'95*

34

Improving Apriori: General Ideas

- *Shrink the number of candidates*
  - *Hash-based technique*
- *Reduce the number of database scans*
  - *Partitioning data*
- Avoid candidate generation
  - FP-growth (next)

35

FP-Growth

- J. Han, J. Pei, and Y. Yin. *Mining Frequent Patterns without Candidate Generation.*, Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00), Dallas, TX, May 2000.
- J. Han, J. Pei, Y. Yin and R. Mao, *Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach*, Data Mining and Knowledge Discovery, 8(1):53-87, 2004. (http://www-faculty.cs.uiuc.edu/~hanj/pdf/dami04_fptree.pdf)
- Chapter 6.2.4 (3rd edition) or Chapter 5.2.4 (2nd Edition)

36

Mining Frequent Patterns Without Candidate Generation (FP-growth)

Two major steps:

- 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
- Mine frequent patterns (itemsets) from an FP-tree
  - A divide-and-conquer methodology: decompose mining tasks into smaller ones
  - Efficient: avoid candidate generation -- generate frequent patterns from the tree directly.

37

Construct FP-tree from a Transaction DB

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}***

min_support = 0.5

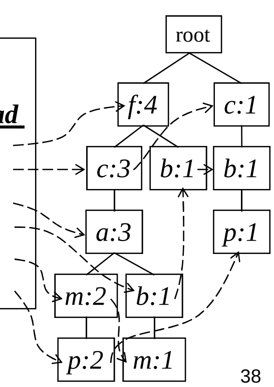
minimum suppot count =3

Steps:

1. Scan DB once, find frequent 1-itemset (single item pattern)
2. Scan DB again, construct FP-tree
   1. Remove infrequent items
   2. Order frequent items in frequency descending order
   3. Put the remaining items in the tree

Header Table

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



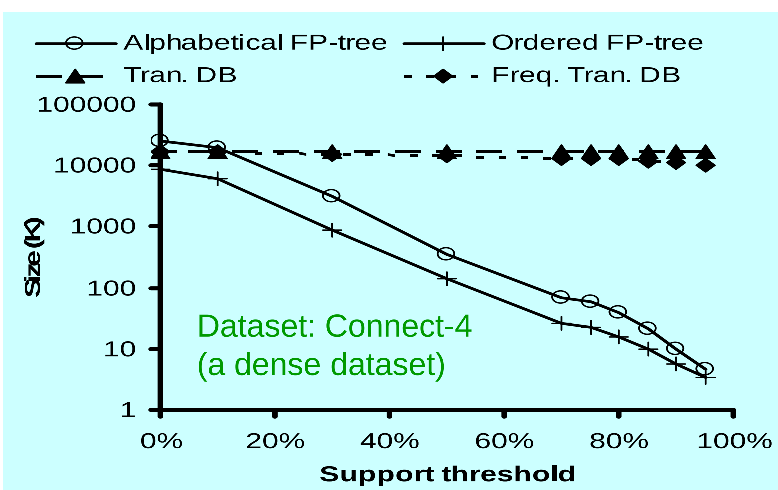
38

Benefits of the FP-tree Structure

- Completeness:
  - map each transaction into a path in the tree
  - preserves complete information for frequent pattern mining
    - no need to scan the database any more
- Compactness
  - reduce irrelevant information—infrequent items are gone
  - A path can store one or more transactions
  - Items in frequency descending order (*f-list*):
    - more frequent items are more likely to be shared
  - never be larger than the original database (not counting node-links and the count fields)

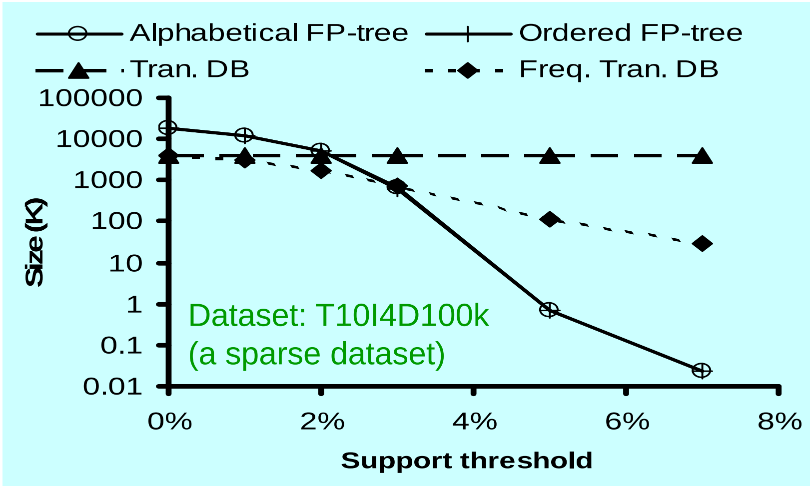
39

How Effective Is FP-tree?



40

Compressing Sparse Dataset



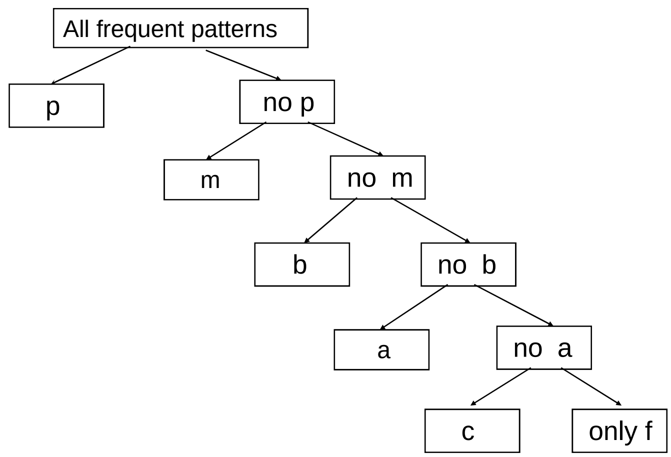
Mining Frequent Patterns from FP-tree

(Frequent pattern = frequent itemset)

- General idea (divide-and-conquer):
  Recursively
  - partition the set of frequent patterns
  - build *conditional pattern base* and *conditional FP-tree* for each partition
- Partition the set of frequent patterns
  - Frequent patterns can be partitioned into subsets according to f-list: f-c-a-b-m-p (the list of freq. items in frequency-descending order)
    - Patterns containing p
    - Patterns having m but no p
    - …
    - Patterns having c but no a nor b, m, or p
    - Pattern f
  - The partitioning is complete and without any overlap

Partitioning Frequent Patterns

f-list: f-c-a-b-m-p



Find Frequent Patterns Having Item "p"

- Only transactions containing p are needed
- Form p-*conditional pattern base* (p-*projected database*) TDB|p Contains transactions containing p
  - Starting at entry p of header table
  - Follow the side-link of frequent item p
  - Accumulate all transformed prefix paths of p

p-conditional pattern base $TDB|_p$

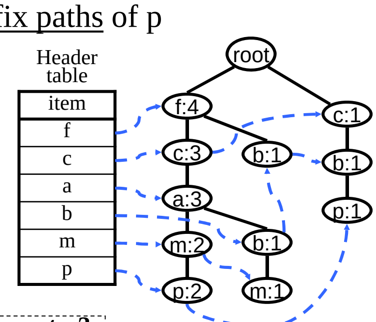
fcam: 2

cb: 1

Local frequent item: c:3

Frequent patterns containing p

cp: 3

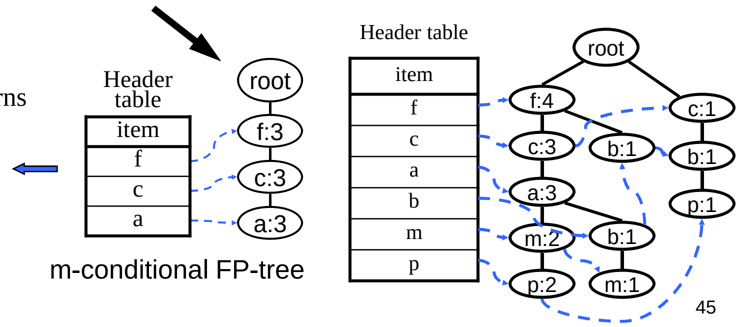


minimum suppot count =3

Find Frequent Patterns Having Item m But No p

- Form m-conditional pattern base (m-projected database) TDB|m
  - Item p is excluded (by looking at only the prefix paths of m)
  - TDB|m contains fca:2, fcab:1
- Recursively apply FP-growth to find freq. patterns from TDB|m
  - Local frequent items: f, c, a
  - After removing local infrequent item: fca:2, fca:1
  - Build m-conditional FP-tree from TDB|m

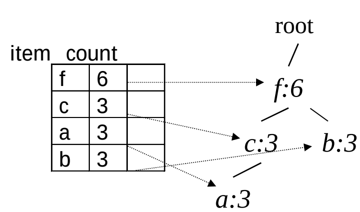
Frequent patterns having m:
fm, cm, am
fcm, fam, cam
fcam



45

Find Frequent Patterns Having Item m But No p (*more complex situation*)

- Suppose m-conditional pattern base is: fca:3, fb:3
- Local frequent items: f:6, c:3, a:3, b: 3
- Build m-conditional FP-tree



- First generate:
  - fm: 6, cm: 3, am:3, bm:3
- To learn longer patterns containing m
  - Further partition frequent patterns containing m (but no p) into
    - Patterns containing b
    - Patterns containing a but no b
    - Patterns containing c but no b or a
    - Patterns containing only f (i.e. fm)
  - Compute *ym*-conditional pattern bases:

ym	conditional pattern base
bm	*f:3*
am	*fc:3*
cm	*f:3*

46

Find Frequent Patterns Having Item m But No p (*more complex situation*)

- Having *ym*-conditional pattern bases:

ym	conditional pattern base
bm	*f:3*
am	*fc:3*
cm	*f:3*

- Built *ym*-conditional FP-trees

bm: root | *f:3*

am: root | *f:3* | *c:3*

cm: root | *f:3*

- General frequent patterns with suffix ym:

  fbm, *fam*, *cam*, *fcam*, *fcm*

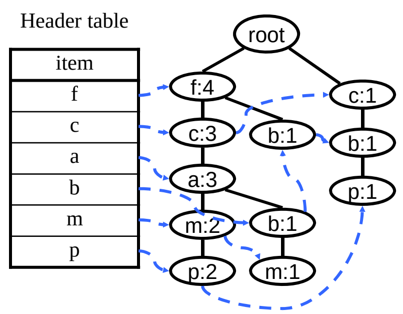
Major Steps to Mine FP-tree

For each item in the FP-tree

1. Construct *conditional pattern base*
2. Construct *conditional FP-tree* from the conditional pattern-base
3. Generate frequent patterns from the conditional FP-tree
   - If the conditional FP-tree contains a single path, simply enumerate all the patterns
   - Otherwise, ***recursively*** mine the conditional FP-tree and grow frequent patterns obtained so far

Step 1: From FP-tree to Conditional Pattern Base

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



***Conditional* pattern bases**

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

49

Step 2: Construct Conditional FP-tree

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

***m-conditional* pattern base:**

fca:2, fcab:1

↓ **Taken out infrequent items**

fca:2, fca:1

↓

root
|
f:3
|
c:3
|
a:3

***m-conditional* FP-tree**

***p-conditional* pattern base:**

fcam:2, cb:1

↓ **Taken out infrequent items**

c:2, c:1

↓

root
|
c:3

***p-conditional* FP-tree**

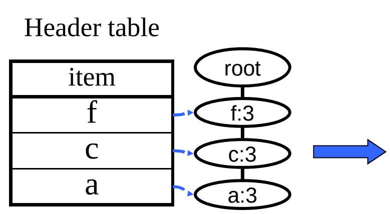
50

Conditional Pattern-Bases and Conditional FP-trees

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

Step 3: Generate Frequent Patterns from Conditional FP-tree

- If an *x*-conditional FP-tree has a single path P
  - The complete set of frequent patterns with suffix *x* can be generated by enumeration of all the combinations of items in P



All frequent patterns with suffix *m*

fm:3, cm:3, am:3, fcm:3, fam:3, cam:3, fcam:3

m-conditional FP-tree

***m-conditional* FP-tree**

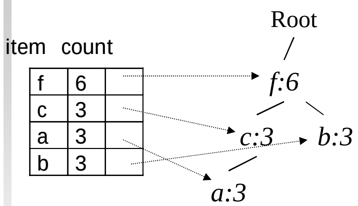
Step 3: Generate Frequent Patterns from Conditional FP-tree (*Contd*.)

- **If an *x*-conditional FP-tree has more than one path**
  - For each item *y* that appears in *x*-conditional FP-tree
    - Generate pattern *yx* with support = the support of y in *x*-conditional FP-tree.
    - Construct *yx*-conditional pattern base and then *yx*-conditional FP-tree to generate frequent patterns with suffix *yx* (a recursive procedure).

53

Step 3: Generate Frequent Patterns from Conditional FP-tree (*Contd*.)

- Suppose m-conditional FP-tree is



***m-conditional* FP-tree**

- Generate frequent 2-itemsets having m:
  fm:6, *cm*:3, *am*:3, *bm*:3
- Compute *ym*-conditional pattern bases:

ym	conditional pattern base
bm	*f:3*
am	*fc:3*
cm	*f:3*

- Built *ym*-conditional FP-trees

bm: R | *f:3*   am: R | f:3 | c:3   cm: R | f:3

- General frequent patterns with suffix ym:
  fbm:3, fam:3, cam:3, fcam:3, fcm:3

54

FP-Growth Algorithm

- Input: *FP-tree* (a FP-tree built by scanning DB)
- Output: the complete set of frequent patterns
- Method: call FP-growth(*FP-tree, null*)
- Procedure FP-growth(*A_conditional_FP_Tree, A*)
  - if Tree contains a single path *P*
    - for each combination (denoted as *B*) of the nodes in the path *P* do
      - generate pattern *BA* with support = minimum support of nodes in *B*
  - else for each item a_i in the header table of *Tree* do
    - generate pattern $B=a_iA$ with support = the support of a_i
    - construct *B*'s conditional pattern base and then *B*'s conditional FP-tree $Tree_B$;
    - if $Tree_B$ in not empty,
      - call **FP-growth**($Tree_B$, *B*)

55

Exercise

- A transaction DB:

TID	items
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

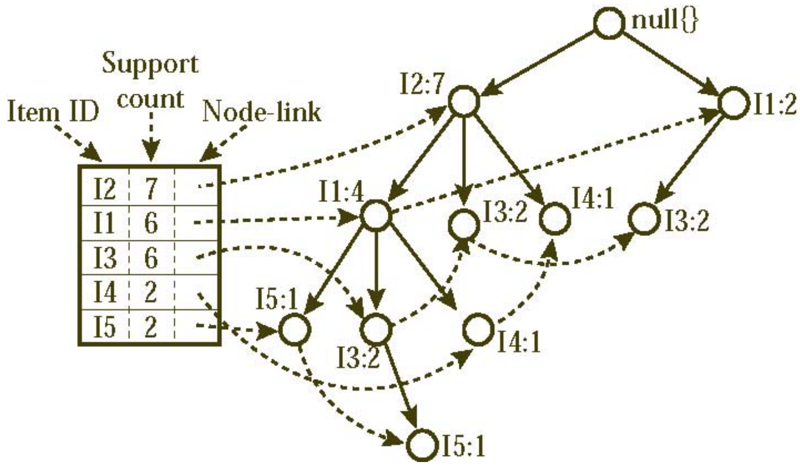
- Support counts for single items:

Item	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

- Find all frequent patterns with minimum support count =2.

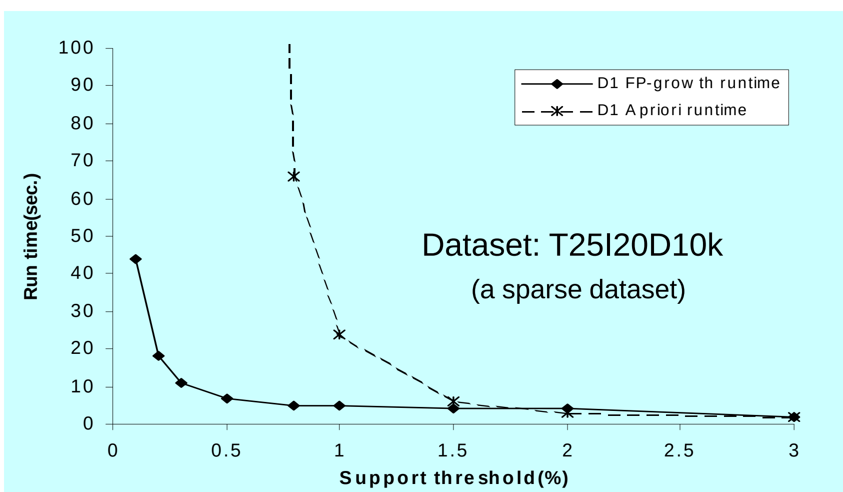
56

FP-tree



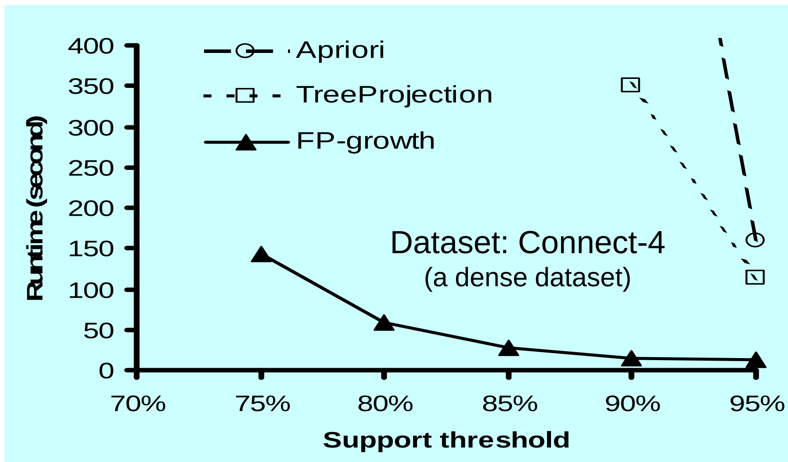
57

FP-growth vs. Apriori



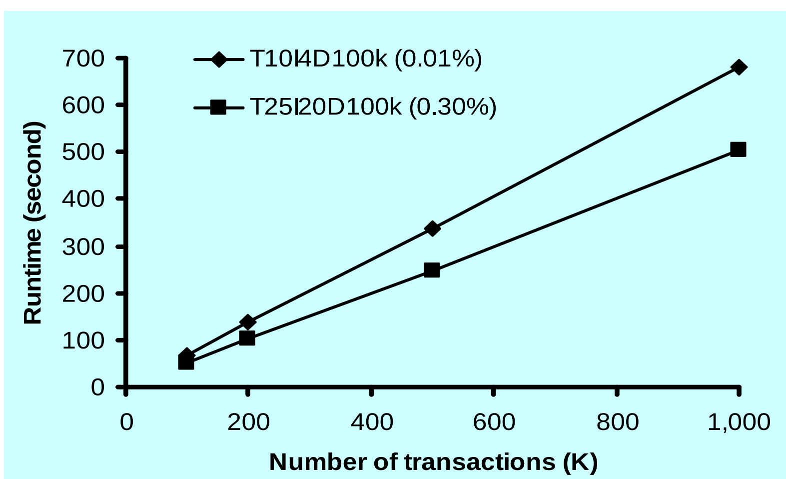
58

Mining Very Dense Dataset



59

Scalability of FP-growth



60

Why Is FP-growth Efficient?

- Divide-and-conquer strategy
  - Decompose both the mining task and DB
  - Lead to focused search of smaller databases
- No candidate generation nor candidate test
- Database compression using FP-tree
  - No repeated scan of entire database
- Basic operations:
  - counting local freq items and building FP-tree, no pattern search nor pattern matching

61

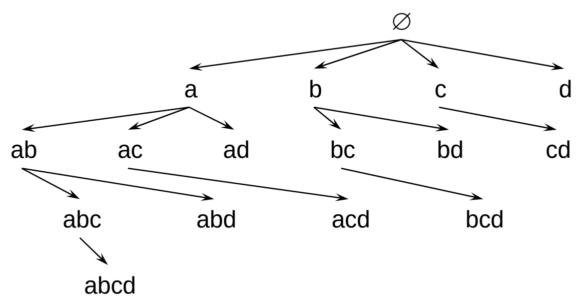
Major Costs in FP-growth

- Building FP-trees
  - A stack of FP-trees
- Redundant information is stored in a stack of FP-trees.
- Can we avoid the redundancy?
  - H-mine (another algorithm by Pei and Han).
  - There is also other improvement to FP-growth in this regard.

62

Compare FP-growth to Apriori

- Search space for DB with 4 items:



- Apriori: breadth-first
- FP-growth: Depth-first

63