Partition: Scan Database Only Twice

- Approach
  - Divide a database into k pieces (local databases called *partition*)

P1	**P2**	**P3**	**P4**

    - Each partition should reside in main memory
  - Find *local frequent patterns* in each partition (scan 1)
    - localMinSup is set as (minSup / k)
    - Local frequent patterns have their localSup larger than localMinSup in any local database
  - Consolidate global frequent patterns (scan 2)

Partition: Scan Database Only Twice

- Guarantee that frequent patterns are never missed
  - Any itemset potentially frequent in DB must be frequent in *at least one partition* of DB

P1	**P2**	**P3**	**P4**

    - localMinSup is set as (minSup / k)
    - Local frequent patterns have their localSup larger than localMinSup in any local database
- A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association in large databases. In *VLDB'95*

DHP: Reduce the Number of Candidates

- Use a hash table for *(k+1)*-itemsets during determining k-itemsets by database scan
  - Candidates of 1-itemset: a, b, c, d, e, f, ….
    - What if 10,000 items? => 100,000,000 candidate 2-itemsets!
  - Hash table for 2-itemsets: {ab, ad, ae} {bd, be, de} …
  - A *(k+1)*-itemset whose corresponding hash bucket count is below the threshold cannot be frequent
  - ab is not a candidate 2-itemset if the sum of count of {ab, ad, ae} is below threshold of minimum support (say, 50)
    - Effective in reducing # of candidate frequent 2-itemsets
- J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *SIGMOD'95*

count	itemsets
35	{ab, ad, ae}
88	{bd, be, de}
. . .	. . .
102	{yz, qs, wt}

Hash Table

Sampling for Frequent Patterns

- Select a sample of an original database, mine frequent patterns within sample using Apriori (in the same way as before)

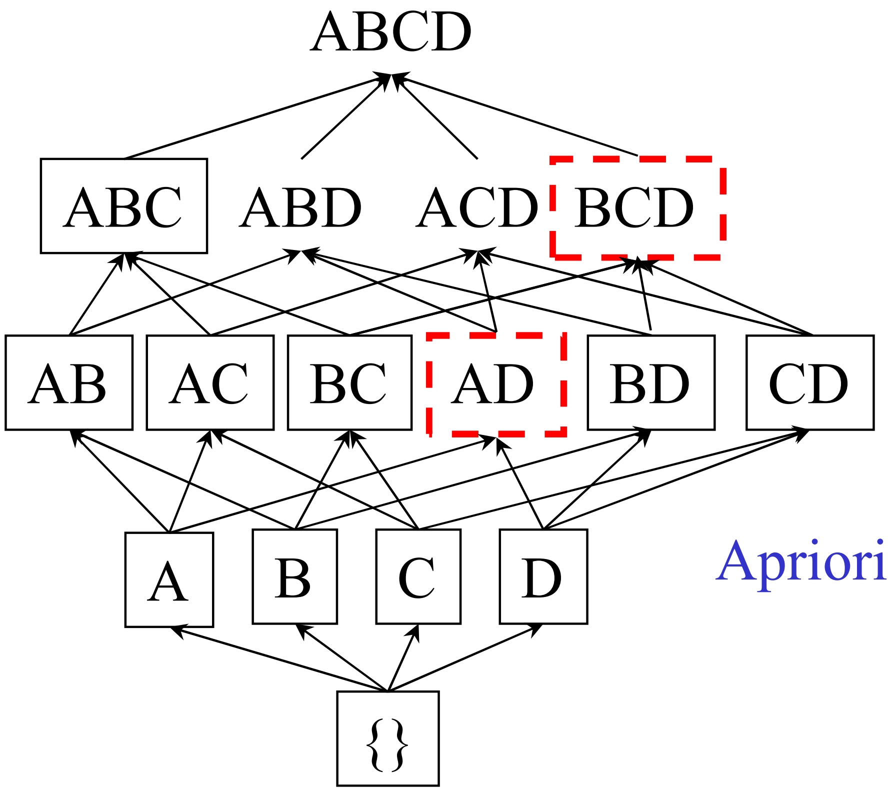
  - Use a smaller value of the minimum support for a sample (say, minSup/4)
- Problems with the simple sampling
  - Some of frequent patterns found in SDB (i.e., **S**) are not really frequent in the original database
  - Some of true frequent patterns could be missed if they are not included in **S**

Sampling for Frequent Patterns

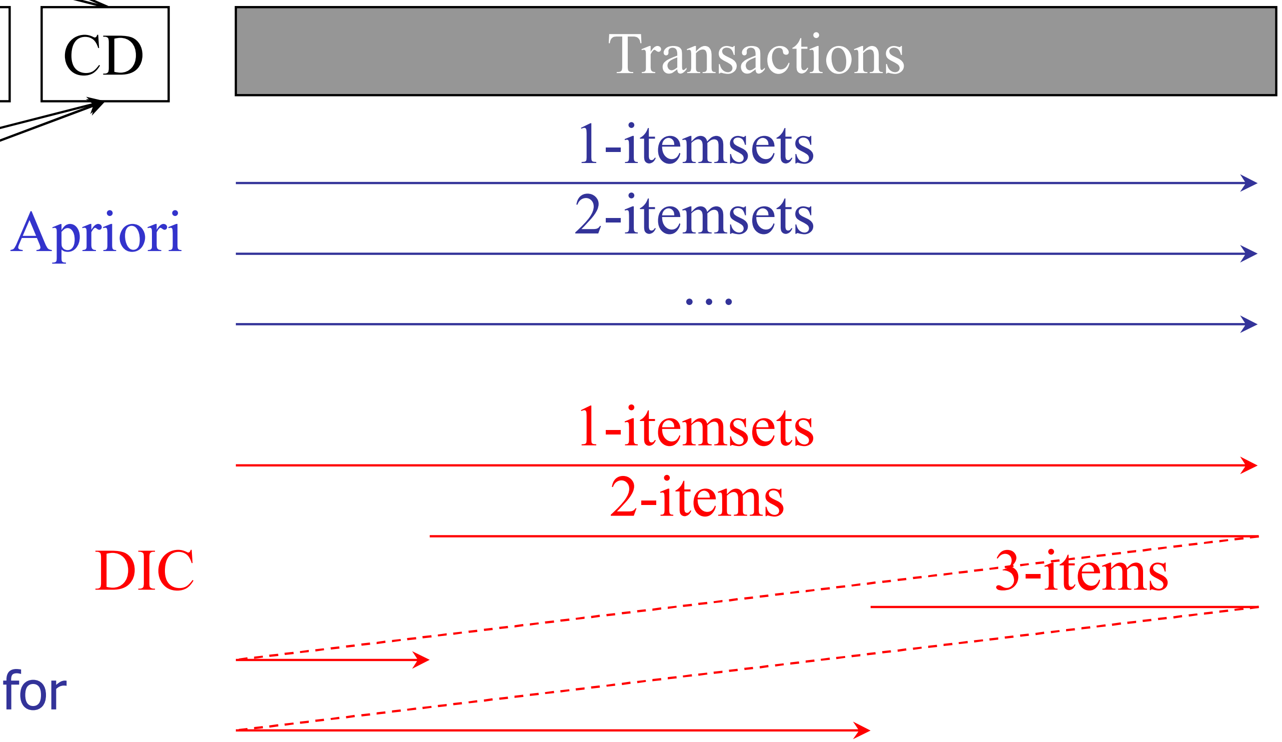
- Solutions: two more scanning for verification
- Scan the whole database once
  - Verify a collection of frequent itemsets, S, found in sample, and its negative borders (NB: not in S, but all its subsets in S)
    - S = {a}, {b}, {c}, {f}, {a,b}, {a,c}, {a,f}, {c,f}, {a,c,f}
    - NB = {b,c}, {b,f}, {d}, {e}
- Scan the whole database again
  - Find missed frequent patterns (due to the success of NBs)
- H. Toivonen. Sampling large databases for association rules. In *VLDB'96*

DIC: Reduce Number of Scans

- Once both A and D are determined frequent, the counting of AD begins
- Once all length-2 subsets of BCD are determined frequent, the counting of BCD begins



Itemset lattice



S. Brin R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD'97*

Bottleneck of Frequent-pattern Mining

- Multiple database scans are costly
- Mining long patterns needs many passes of scanning and generates lots of candidates
  - To find frequent itemset $i_1 i_2 \ldots i_{100}$
    - \# of scans: 100
    - \# of Candidates: $\binom{100}{1} + \binom{100}{2} + \ldots + \binom{100}{100} = 2^{100} - 1 = 1.27*10^{30}$!
- Bottleneck: candidate-generation-and-test
- Can we avoid candidate generation?

FP-Growth: 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"
    - Denoted as DB|abc
  - "d" is a local frequent item in DB|abc → abcd is a frequent pattern

FP-Growth: Construct FP-tree from a Transaction Database

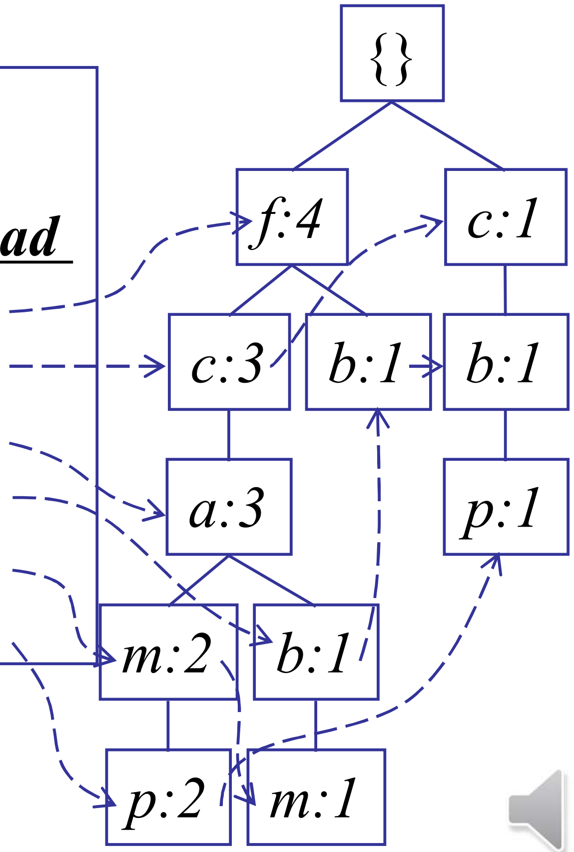
TID	*Items bought*	*(ordered) frequent items*
100	{*f, a, c, d, g, i, m, p*}	{*f, c, a, m, p*}
200	{*a, b, c, f, l, m, o*}	{*f, c, a, b, m*}
300	{*b, f, h, j, o, w*}	{*f, b*}
400	{*b, c, k, s, p*}	{*c, b, p*}
500	{*a, f, c, e, l, p, m, n*}	{*f, c, a, m, p*}

min_support = 3

1. Scan DB once, find frequent 1-itemset (single item pattern)
2. Sort frequent items in frequency descending order, f-list
3. Scan DB again, construct FP-tree

Header Table

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



F-list=f-c-a-b-m-p

Benefits of the FP-tree Structure

- Completeness
  - Preserve *complete (i.e., lossless) information* for frequent pattern mining
  - Never break a long pattern of any transaction
- Compactness
  - Remove 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 counting node-links and the *count* field)
    - For Connect-4 DB, compression ratio could be over 100

Partition Patterns and Databases

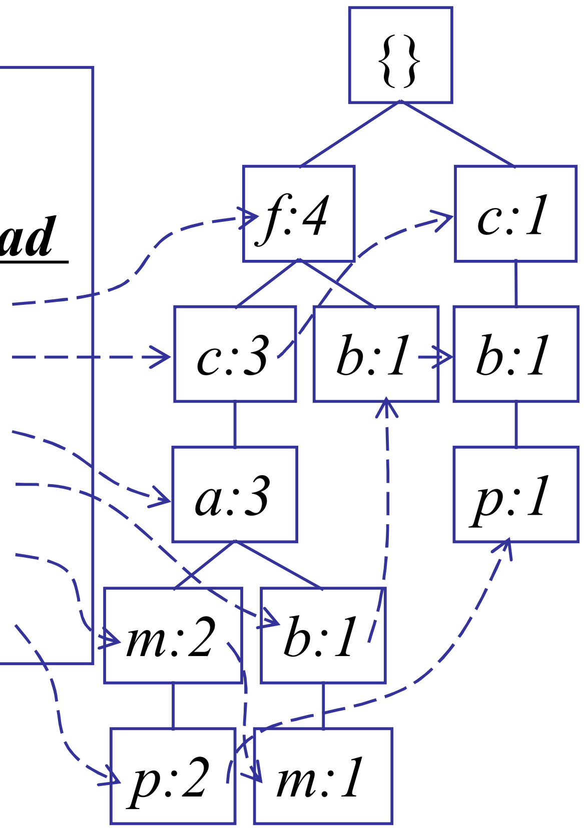
- Frequent patterns can be *partitioned* into (*disjoint*) subsets according to f-list
  - F-list=f-c-a-b-m-p
  - Patterns containing p
  - Patterns having m but no p
  - Patterns having m but no m nor p (i.e., not containing m and p)
  - …
  - Patterns having c but no a nor b, m, p
  - Pattern f
- Completeness and non-redundancy

Find Patterns Having P From P-conditional Database

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

Header Table

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



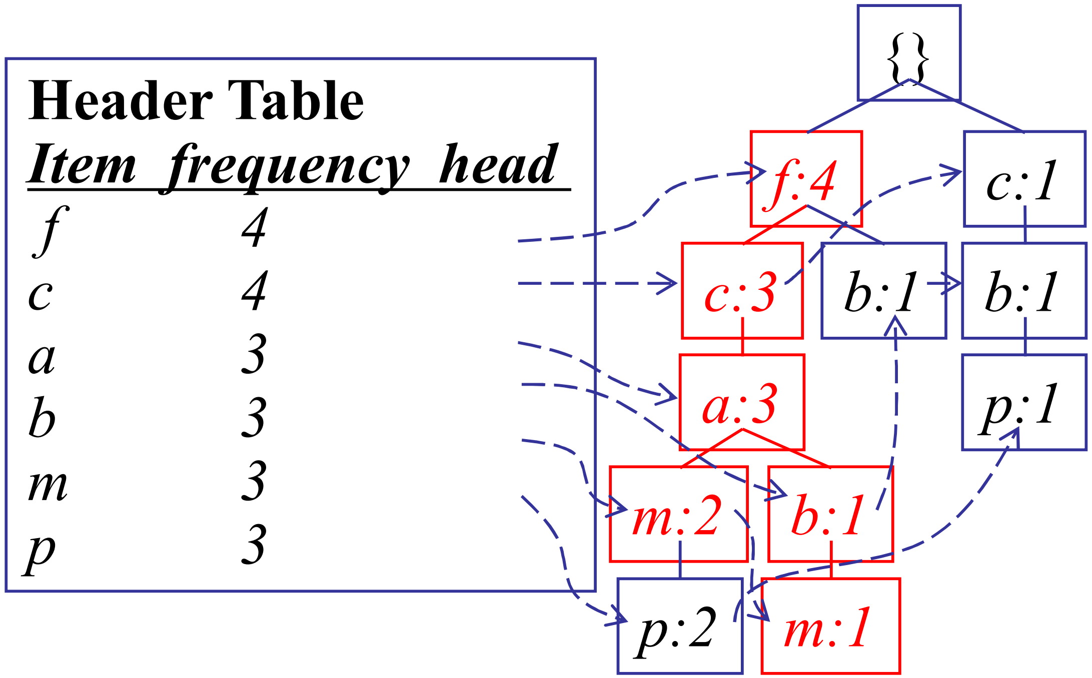
***Conditional* pattern bases**

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**

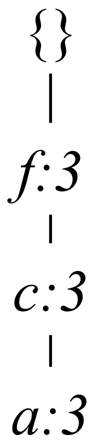
From Conditional Pattern-bases to Conditional FP-trees

- For each conditional database (i.e., pattern-base)
  - Accumulate the count for each item in the base
  - Construct the FP-tree for the frequent items of the pattern base

min_support = 3



***m-conditional* pattern base:**
fca:2, fcab:1



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

All frequent patterns related to *m*

m,
fm, cm, am,
fcm, fam, cam,
fcam

Recursion: Mining Each Conditional FP-tree

{}
|
f:3
|
c:3
|
a:3

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

Cond. pattern base of "am": (fc:3)

{}
|
f:3
|
c:3

***am-conditional* FP-tree**

Cond. pattern base of "cm": (f:3)

{}
|
f:3

***cm-conditional* FP-tree**

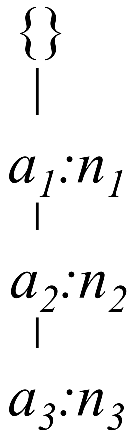
Cond. pattern base of "cam": (f:3)

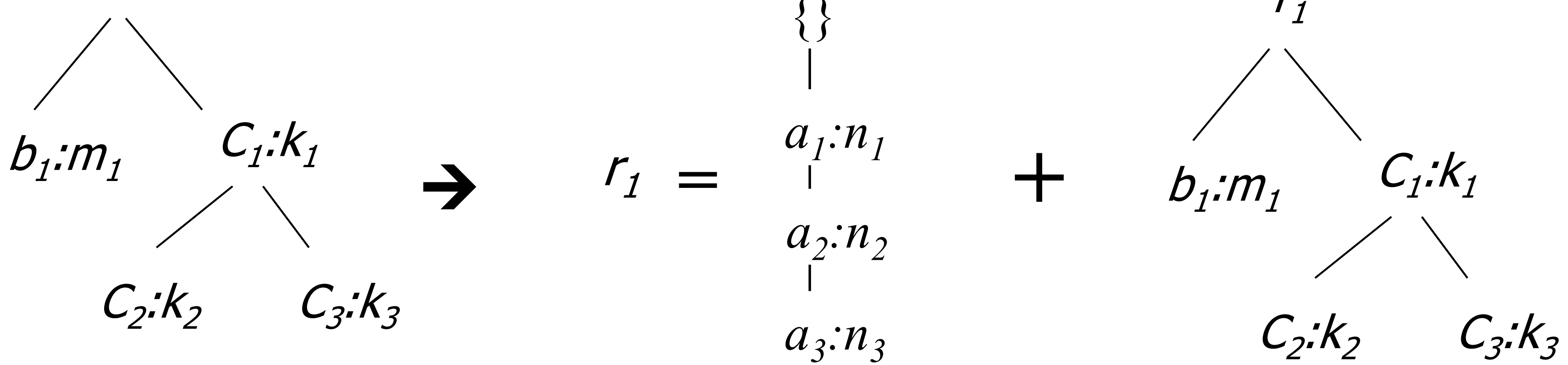
{}
|
f:3

***cam-conditional* FP-tree**

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
    - Reduce the overhead of recursions
  - Concatenation of the mining results of the two parts

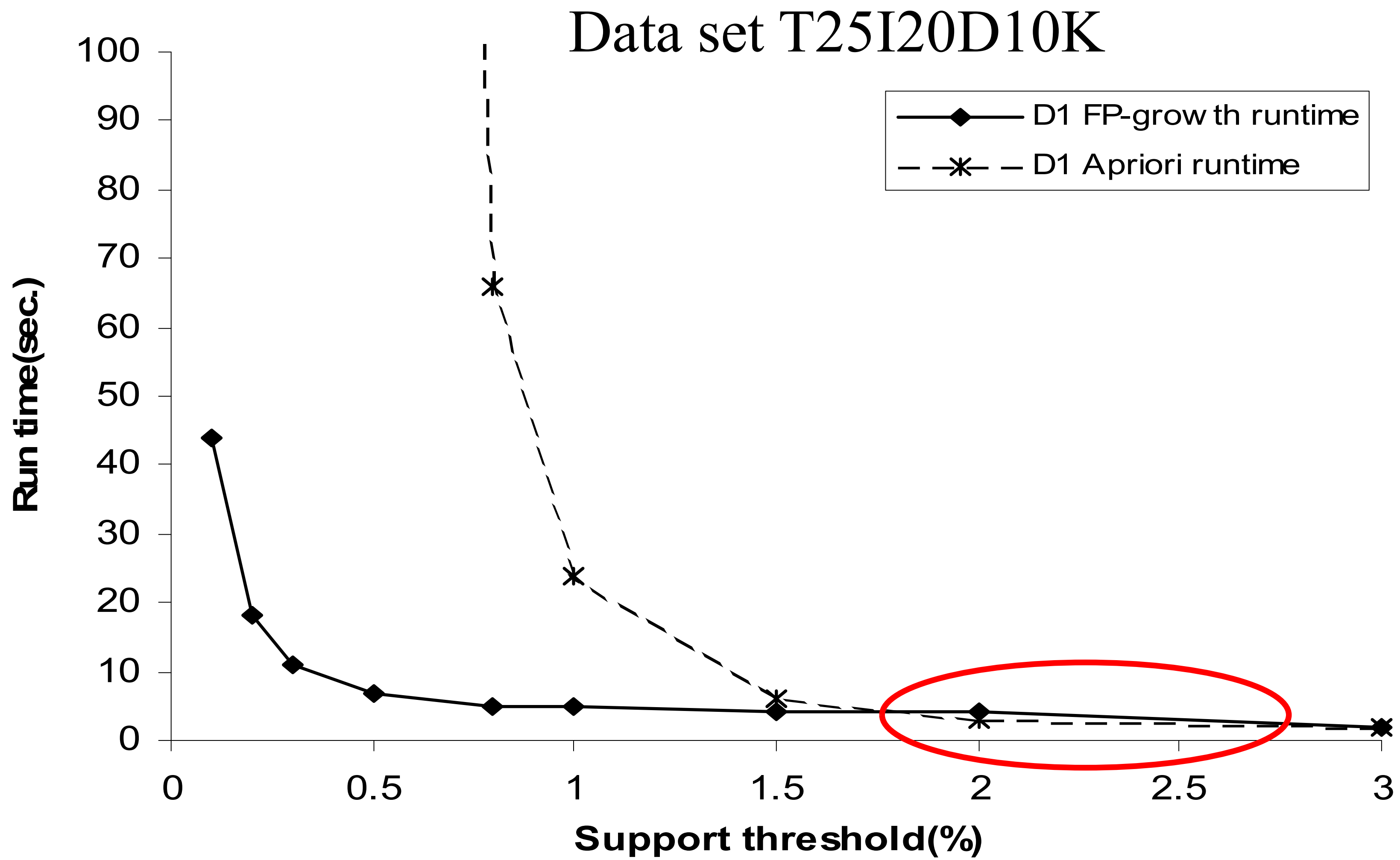




Summary of Ideas with FP-Growth

- Idea: Frequent pattern growth
  - Grow frequent patterns by adding a new frequent item recursively
- Method
  - For each frequent item, construct its conditional pattern-base, and 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 a single path
    - A single path will generate all the combinations of its sub-paths
    - Each of the combinations is a frequent pattern

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



Why Is FP-Growth the Winner?

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