Chapter 5: Mining Frequent Patterns, Association and Correlations

Dong-Kyu Chae

PI of the Data Intelligence Lab @HYU
Department of Computer Science & Data Science
Hanyang University

Data
Intelligence
LAB

FP-Growth: Mining Frequent Patterns Without Candidate Generation

- **Motivation**
  - The candidate-generation-and-test process is the main bottleneck
  - Can we completely **avoid** candidate generation?
- **FP(frequent pattern) growth:**
  - **Main idea:** growing long patterns from short ones using local frequent items
  - Assume that "a" is a frequent pattern
  - Then, get all transactions having "a"
    - Denoted as DB|a
  - If "b" is a local frequent item in DB|a → "ab" is a frequent pattern!
    - Then, get DB|ab, … (recursive)
- **Process**
  - 1. Construct "**global FP-tree**"
  - 2. Take the divide-and-conquer strategy: divide target frequent patterns, where each division recursively "**grows**" the frequent patterns

Construct Global FP-tree from Database

TID	Items bought	frequent items (ordered by frequency descending order)
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_sup = 3

Remove non-frequent items & Sort based on the order of F-list

1. Scan DB once, find frequent 1-itemsets (single item pattern)
2. Sort the 1-size frequent items in frequency descending order, F-list

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

3. Scan DB again, sort each transaction based on F-list using only frequent single items, and construct the FP-tree (next page)

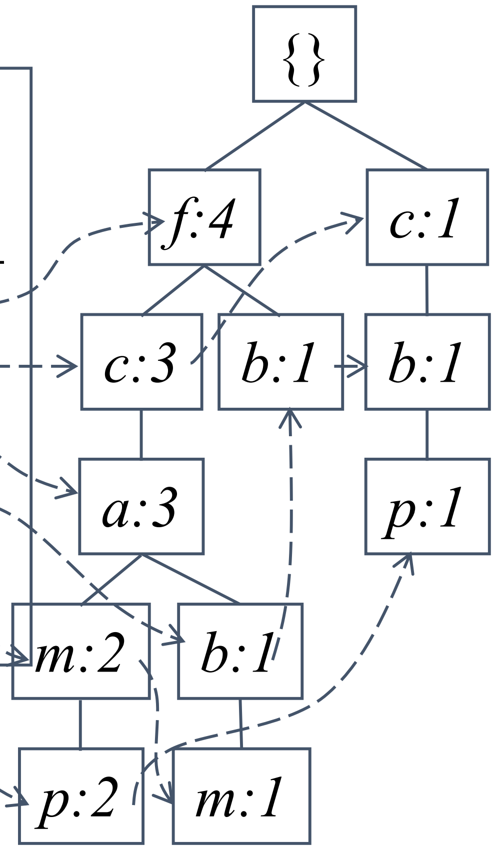
Header Table

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

reverse of alphabet order

같은 frequency 가지는 것들 order 규칙 정해주지 않거나

alphabet order



Construct Global FP-tree from Database

TID	Items bought	frequent items (ordered by frequency descending order)
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_sup = 3

① F-list=f-c-a-b-m-p

② **Header Table**

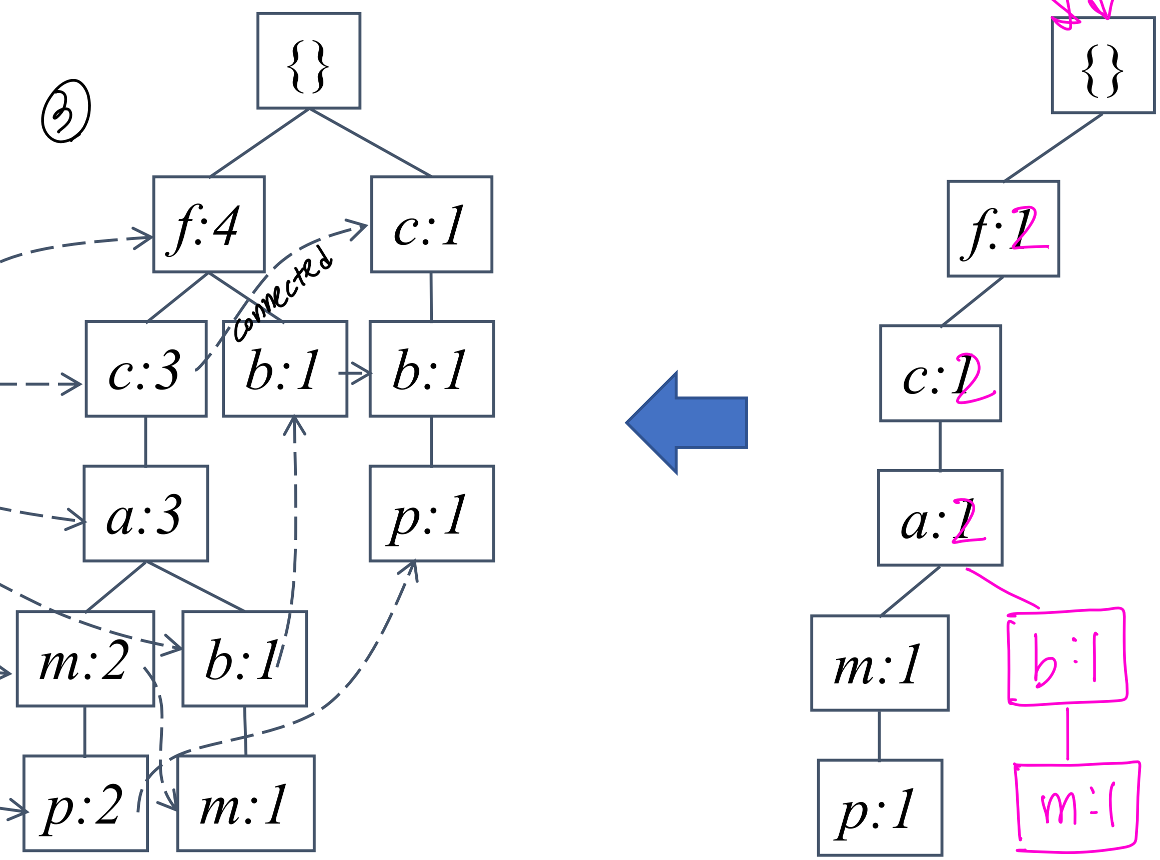
each member of F-list

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

firstly appeared

pointer (linked list)

③



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
  - Much reduced size, can be uploaded on memory

Ideas with FP-Growth

- **Grow frequent patterns by adding a new frequent item <u>recursively</u>**
- **Given the global FP-tree and F-list,**
  - It follows **divide-and-conquer**: it partitions the frequent patterns into **disjoint** subsets (targets)

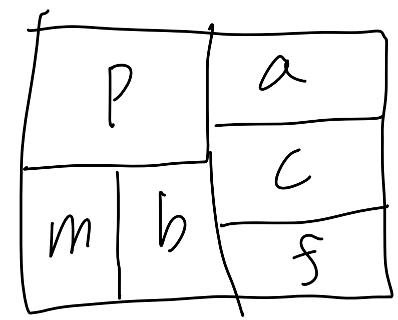
    ex) {a, b, c, d}
  - For the partitioning, it uses each item of F-list. For each item, it constructs conditional pattern-base, and then construct its

    DB|a, DB|b, DB|c, DB|d

    conditional FP-tree to find the targeted frequent items.
    - Find a frequent item on the conditional FP-tree, construct a new FP-tree conditioned by it, and grow the pattern.
    - Recursevily repeat the process on each newly created conditional FP-tree, until the **resulting conditional 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

Partitioning Target Frequent Patterns

- **General idea: Frequent patterns can be *partitioned* into *disjoint* subsets according to F-list**
  - F-list=**f-c-a-b-m-p**

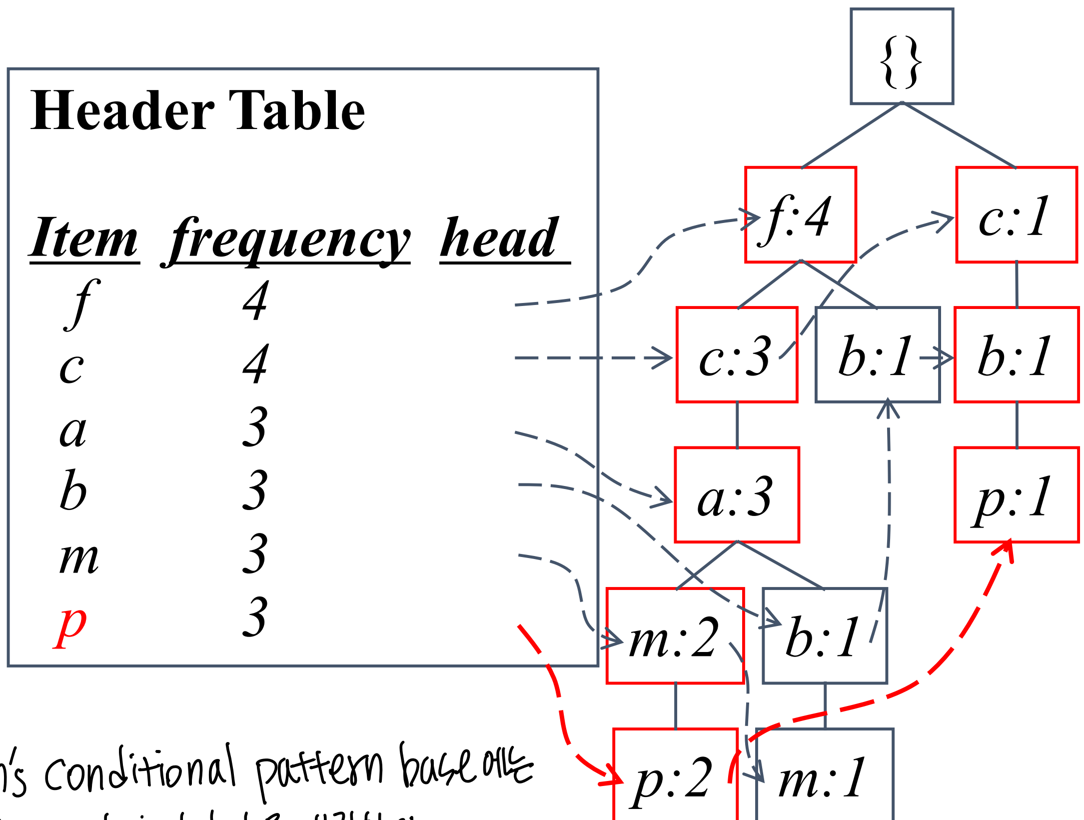


    - **T1**: Freq. patterns containing **p** DB|P
    - **T2**: Freq. patterns having **m** but no p DB|m
    - **T3**: Freq. patterns having **b** but no p, m
    - **T4**: Freq. patterns having **a** but no p, m, b
    - **T5**: Freq. patterns having **c** but no p, m, b, a
    - **T6**: Freq. pattern **f** (no others)
  - Divide-and-conquer
    - From the end of the list, pick one by one to make its **conditional pattern-base**
    - From each conditional pattern-base, we can find all the frequent patterns corresponding to the given taget!

Constructing P-conditional Pattern Base

indicates any given single frequent items

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



m's conditional pattern base에는
P is not included 를 보장한다

Conditional pattern bases

item	cond. pattern base	Target
f	{}	T6
c	*f*:3	T5
a	*fc*:3	T4
b	*fca*:1, *f*:1, *c*:1	T3
m	*fca*:2, *fcab*:1	T2
p	*fcam*:2, *cb*:1	T1

⇒ The frequent patterns found in each local DB are disjoint!

From Conditional Pattern-base to Conditional FP-tree

- **For each conditional database (i.e., pattern-base)**
  - Do the exactly same thing with constructing the global FP-tree
    - Accumulate the count for each item in the base
    - Construct the FP-tree for the frequent items of the pattern base

trans.	
1	f c a
2	f c a
3	f c a ~~b~~

→ f:3, c:3, a:3, ~~b:1~~ not frequent
→ frequent items
→ sort by any order (Cuz same order) : Reverse of alphabet order
→ F-list: f-c-a

***Conditional* pattern bases**

item	***cond. pattern base***	***Target***
f	{}	T6
c	*f:3*	T5
a	*fc:3*	T4
b	*fca:1, f:1, c:1*	T3
m	*fca:2, fcab:1*	T2
p	*fcam:2, cb:1*	T1

{ } → f

{ } f:3 → c, fc

C-conditional FP-Tree

{ } f:3

a-conditional FP-Tree: c:3 → a, fa, ca, fca

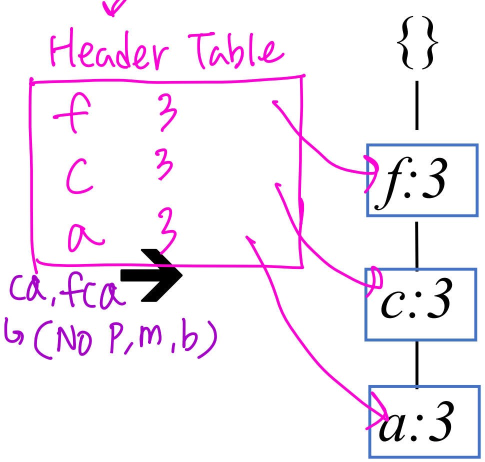
↳ (No p, m, b)

→ No frequent patterns → b

{ } p-conditional FP-Tree : c:3 → frequent patterns p, pc

m-conditional* pattern base:** ***min_sup = 3

fca:2, fcab:1



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

→

All frequent patterns related to *m*

m,

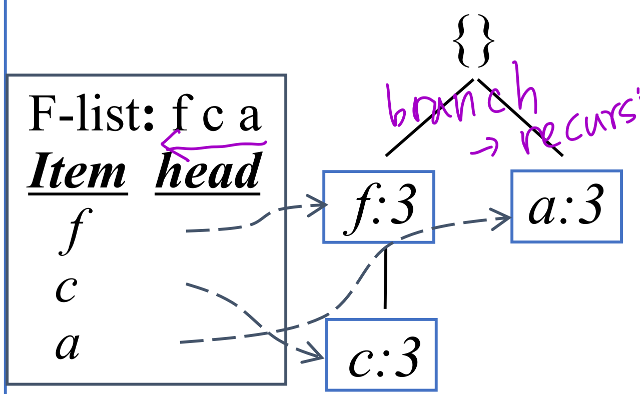
fm, cm, am,

fcm, fam, cam,

fcam

Recursion Case

*Assume **m-conditional pattern base:** fc:3, a:3*



All frequent patterns related to *m*

m,

am, cm, fm,

fcm

fc
fc
fc
a
a

m-conditional pattern base

a | c f

DB|am

1. Cond. pattern base of "am": ( )

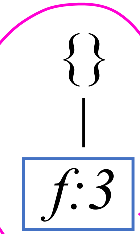
{}

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

All frequent patterns related to *am*

am

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



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

empty이고
single path이므로
recursion stops here

All frequent patterns related to *cm*

cm,

fcm

3. Cond. pattern base of "fm": ( )

{}

***fm-conditional* FP-tree**

All frequent patterns related to *fm*

fm

Single Prefix Case

- **Suppose a conditional FP-tree T is a single prefix-path P**
- **In this case, we don't have to dive into additional resursion**
- **The sub-problem on that tree can be terminated easily**
  - Making combinations of the nodes in the single path
  - The union of each combination and the base items is a frequent pattern

: m

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

→

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

→

All frequent patterns related to *m*

m,

fm, cm, am,

fcm, fam, cam,

fcam

Pseudo Code

Base item

procedure FP-Growth (*Tree*, α)

{

({} empty case 포함)

if *Tree* contains a single path *P* then

for each β = nodes combination in *P* do

pattern = $\beta \cup \alpha$;

support = min(support of the nodes in β);

else

for each *ai* in the header of *Tree* do

pattern $\beta = ai \cup \alpha$;

with support = *ai*. support ;

construct conditional pattern base of β

TreeB = construct conditional FP-tree of β

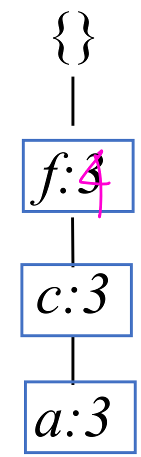
if $\boxed{TreeB != \emptyset}$ then

call FP-Growth(*TreeB* , β)

}

m-conditional **pattern base:**

fca:2, fcab:1



All frequent patterns related to *m*

m,

fm, cm, am,

4 3 3

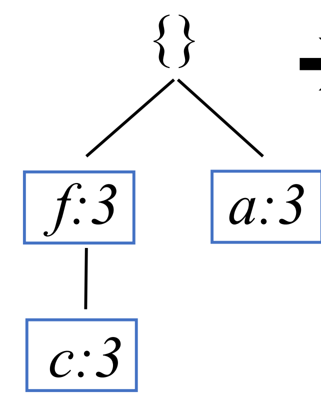
fcm, fam, cam,

3 3 3

fcam

m-conditional **pattern base:**

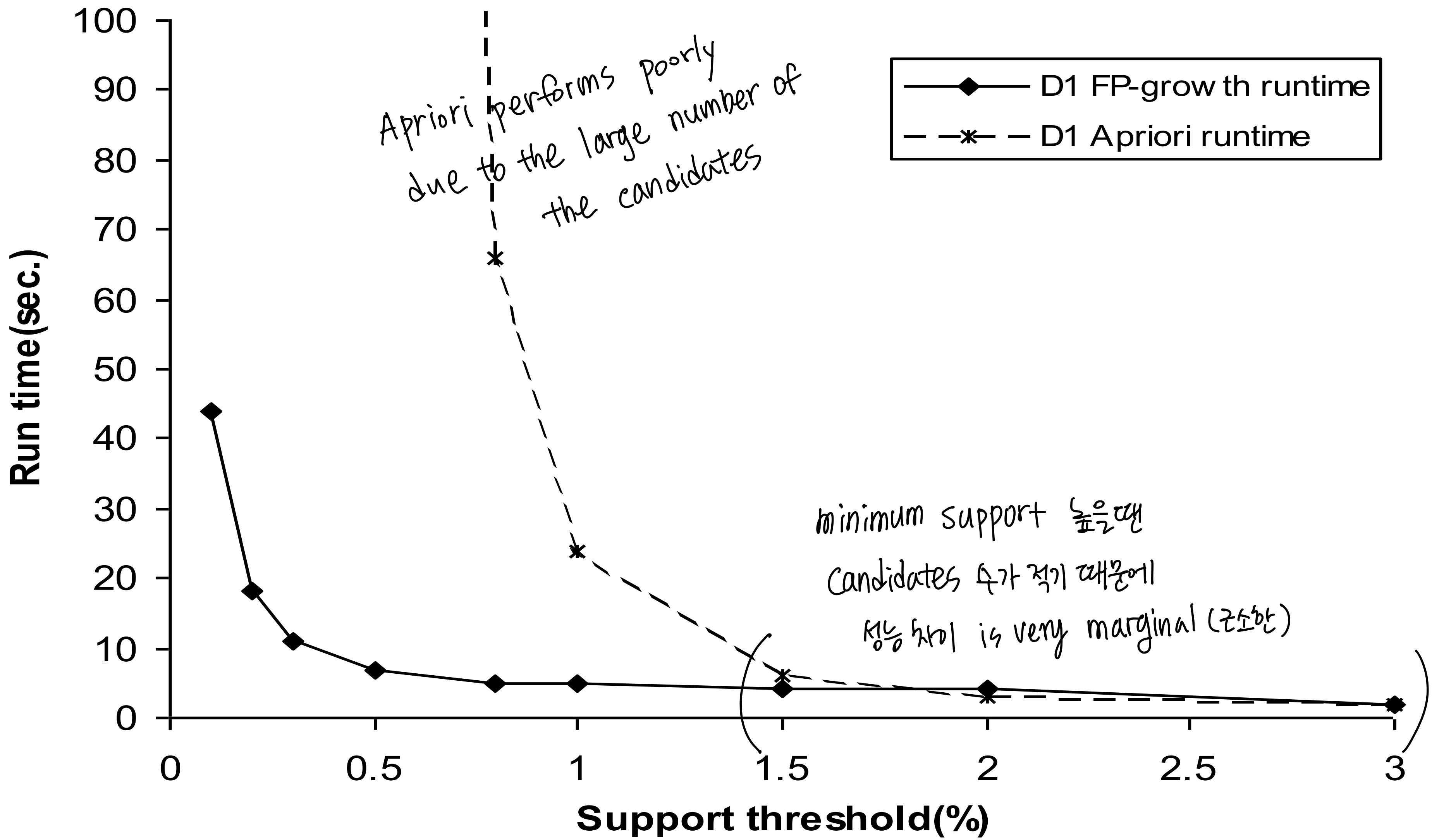
fc:3, a:3



Recursion!

FP-Growth vs. Apriori:

❑ Running time depending on the minimum support



Why Is FP-Growth Effective?

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 → Apriori의 main bottleneck 해결
- Compressed database: FP-tree structure (managed on memory)
- No repeated scans of the entire database: just twice
  - 1: Counting the frequency of each item (size 1)
  - 2: Building the global FP-tree

Thank You