

---

# COMP9318: Data Warehousing and Data Mining

— L6: Association Rule Mining —

---

- Problem definition and preliminaries

# What Is Association Mining?

---

- Association rule mining:
  - Finding frequent patterns, associations, correlations, or causal structures among sets of items or objects in transaction databases, relational databases, and other information repositories.
  - **Frequent pattern**: pattern (set of items, sequence, etc.) that occurs frequently in a database [AIS93]
- Motivation: finding regularities in data
  - What products were often purchased together? — Beer and diapers?!
  - What are the subsequent purchases after buying a PC?
  - What kinds of DNA are sensitive to this new drug?
  - Can we automatically classify web documents?

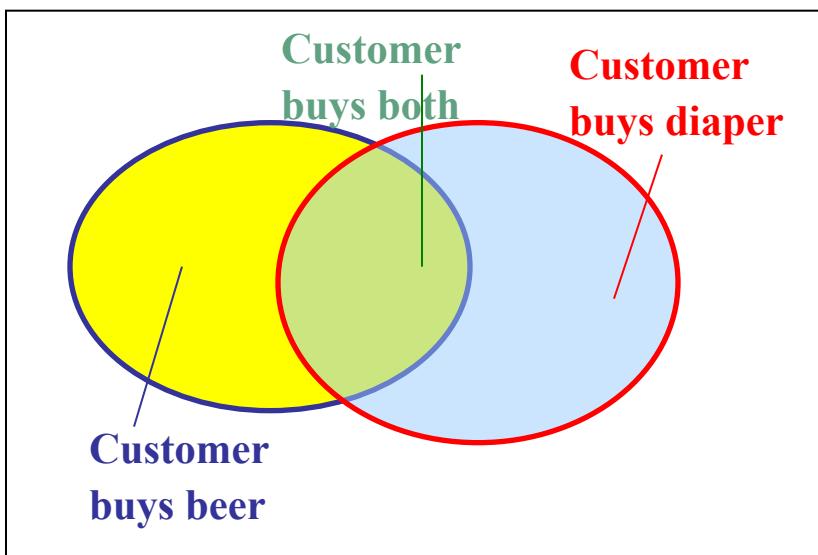
# Why Is Frequent Pattern or Association Mining an Essential Task in Data Mining?

---

- Foundation for many essential data mining tasks
  - Association, correlation, causality
  - Sequential patterns, temporal or cyclic association, partial periodicity, spatial and multimedia association
  - Associative classification, cluster analysis, iceberg cube, fascicles (semantic data compression)
- Broad applications
  - Basket data analysis, cross-marketing, catalog design, sale campaign analysis
  - **Web log** (click stream) **analysis**, DNA sequence analysis, etc. c.f., google's spelling suggestion

# Basic Concepts: Frequent Patterns and Association Rules

Transaction-id	Items bought
10	{ A, B, C }
20	{ A, C }
30	{ A, D }
40	{ B, E, F }



- Itemset  $X = \{x_1, \dots, x_k\}$ 
  - **Shorthand:**  $x_1 x_2 \dots x_k$
- Find all the rules  $X \rightarrow Y$  with min confidence and support
  - **support, s, probability** that a transaction contains  $X \cup Y$
  - **confidence, c, conditional probability** that a transaction having  $X$  also contains  $Y$ .

Let  $\text{min\_support} = 50\%$ ,

$\text{min\_conf} = 70\%:$

$$\text{sup}(AC) = 2$$

$$A \rightarrow C (50\%, 66.7\%)$$

$$C \rightarrow A (50\%, 100\%)$$

frequent itemset

association rule

# Mining Association Rules—an Example

Transaction-id	Items bought
10	A, B, C
20	A, C
30	A, D
40	B, E, F

Min. support 50%  
Min. confidence 50%

Frequent pattern	Support
{A}	75%
{B}	50%
{C}	50%
{A, C}	50%

For rule  $A \rightarrow C$ :

$$\text{support} = \text{support}(\{A\} \cup \{C\}) = 50\%$$

$$\text{confidence} = \text{support}(\{A\} \cup \{C\}) / \text{support}(\{A\}) = 66.6\%$$

major computation challenge: calculate the support of itemsets

← The **frequent itemset mining** problem

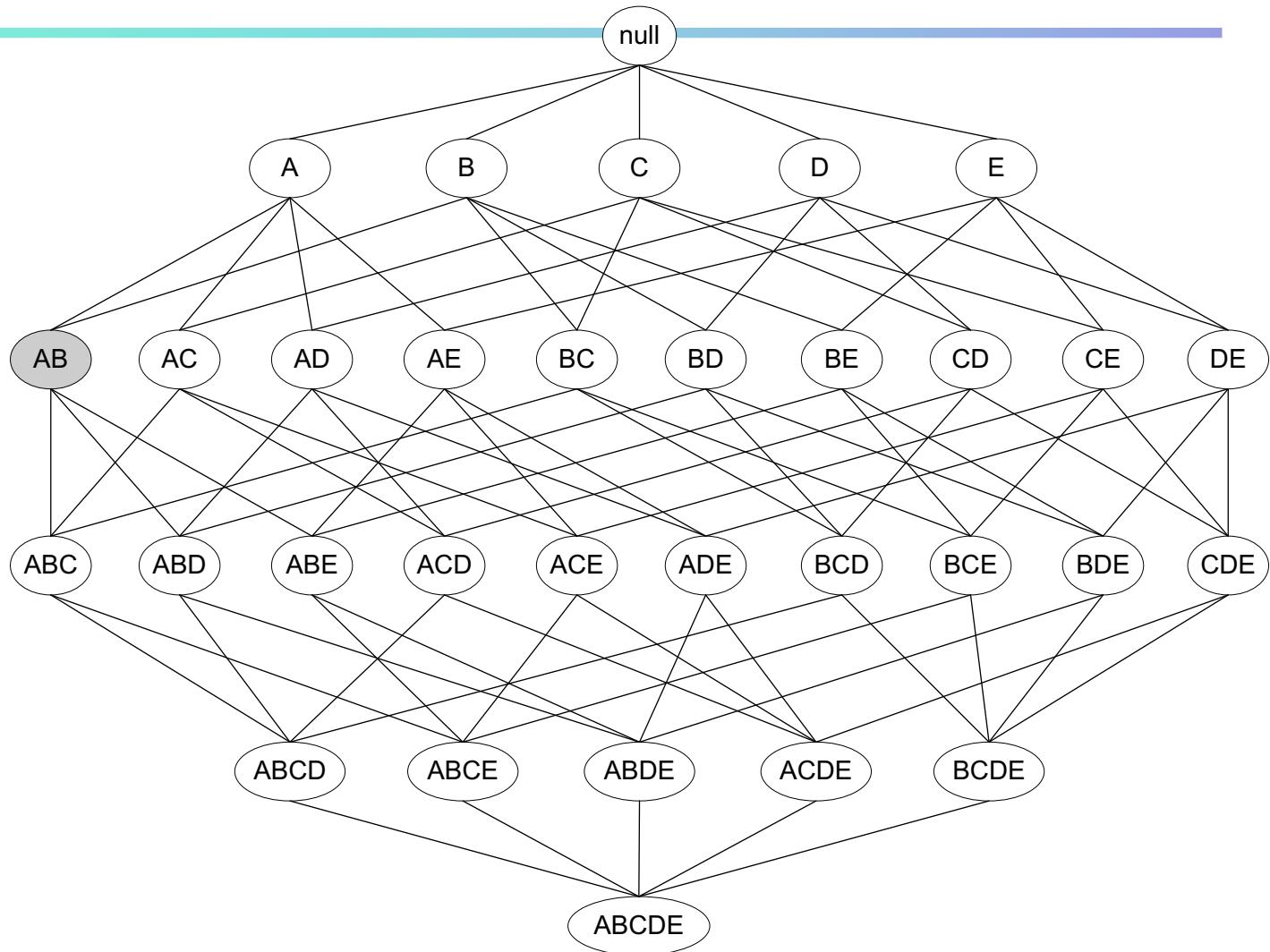
- 
- Algorithms for scalable mining of (single-dimensional Boolean) association rules in transactional databases

# Association Rule Mining Algorithms

Candidate Generation  
& Verification

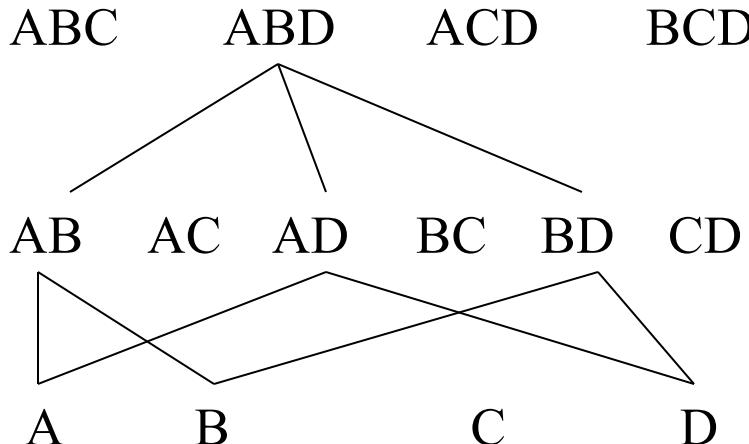
- Naïve algorithm
  - Enumerate all possible itemsets and check their support against *min\_sup*
  - Generate all association rules and check their confidence against *min\_conf*
- The Apriori property
  - Apriori Algorithm
  - FP-growth Algorithm

# All Candidate Itemsets for $\{A, B, C, D, E\}$

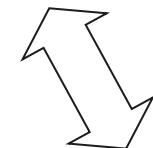


# Apriori Property

- A *frequent* (used to be called *large*) *itemset* is an itemset whose support is  $\geq \text{min\_sup}$ .
- Apriori property (downward closure): any **subsets** of a frequent itemset are also frequent itemsets
- Aka the **anti-monotone** property of support



“any **supersets** of an **infrequent** itemset are also **infrequent** itemsets”

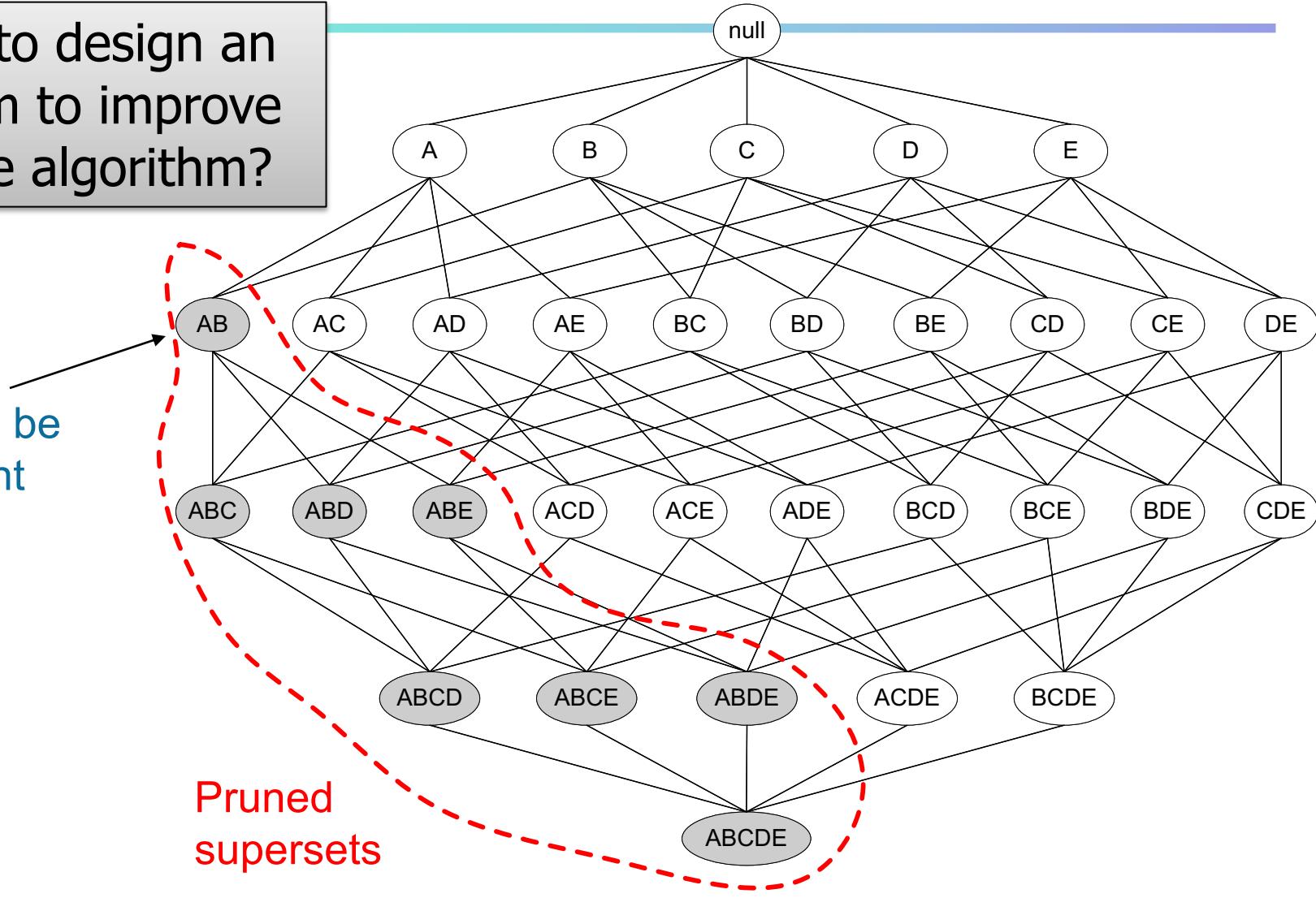


# Illustrating Apriori Principle

Q: How to design an algorithm to improve the naïve algorithm?

Found to be Infrequent

Pruned supersets



# Apriori: A Candidate Generation-and-test Approach

---

- **Apriori pruning principle:** If there is **any** itemset which is infrequent, its superset should not be generated/tested!
- Algorithm [Agrawal & Srikant 1994]
  1.  $C_k \leftarrow$  Perform level-wise candidate generation (from singleton itemsets)
  2.  $L_k \leftarrow$  Verify  $C_k$  against  $L_k$
  3.  $C_{k+1} \leftarrow$  generated from  $L_k$
  4. Goto 2 if  $C_{k+1}$  is not empty

# The Apriori Algorithm

---

- Pseudo-code:

$C_k$ : Candidate itemset of size  $k$

$L_k$ : frequent itemset of size  $k$

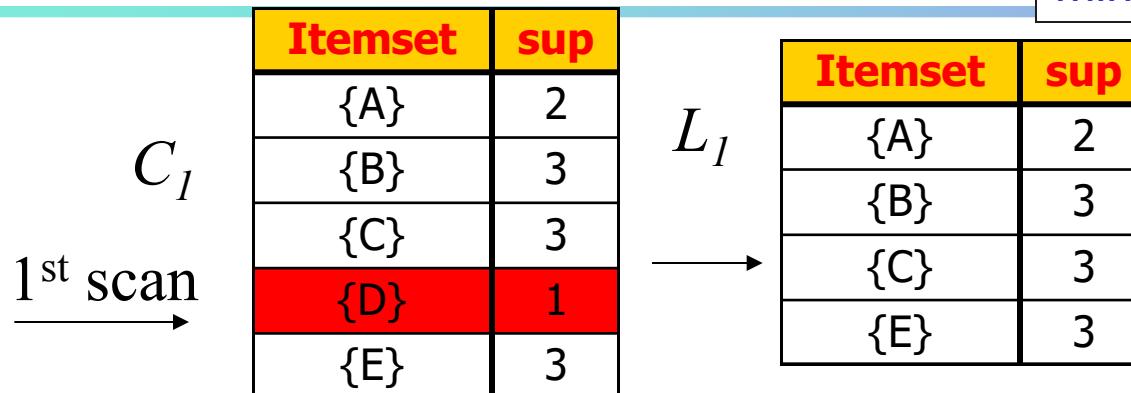
```
 $L_1 = \{\text{frequent items}\};$ 
for ( $k = 1; L_k \neq \emptyset; k++$ ) do begin
     $C_{k+1} = \text{candidates generated from } L_k;$ 
    for each transaction  $t$  in database do begin
        increment the count of all candidates in  $C_{k+1}$ 
        that are contained in  $t$ 
    end
     $L_{k+1} = \text{candidates in } C_{k+1} \text{ with min\_support}$ 
end
return  $\bigcup_k L_k;$ 
```

# The Apriori Algorithm—An Example

$\text{minsup} = 50\%$

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E



$L_2$

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

$C_2$

2<sup>nd</sup> scan ←

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

$C_2$

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

$C_3$

3<sup>rd</sup> scan →  $L_3$

Itemset	sup
{B, C, E}	2

# Important Details of Apriori

---

1. How to generate candidates?
  - Step 1: self-joining  $L_k$  (**what's the join condition? why?**)
  - Step 2: pruning
2. How to count supports of candidates?

## Example of Candidate-generation

- $L_3 = \{abc, abd, acd, ace, bcd\}$
- Self-joining:  $L_3 * L_3$ 
  - $abcd$  from  $abc$  and  $abd$
  - $acde$  from  $acd$  and  $ace$
- Pruning:
  - $acde$  is removed because  $ade$  is not in  $L_3$
- $C_4 = \{abcd\}$

# Generating Candidates in SQL

- Suppose the items in  $L_{k-1}$  are listed in an order
- Step 1: self-joining  $L_{k-1}$

```
insert into  $C_k$ 
select  $p.item_1, p.item_2, \dots, p.item_{k-1}, q.item_{k-1}$ 
from  $L_{k-1} p, L_{k-1} q$ 
where  $p.item_1=q.item_1, \dots, p.item_{k-2}=q.item_{k-2}, p.item_{k-1} <$ 
       $q.item_{k-1}$ 
```

- Step 2: pruning

```
forall itemsets  $c$  in  $C_k$  do
    forall ( $k-1$ )-subsets  $s$  of  $c$  do
        if ( $s$  is not in  $L_{k-1}$ ) then delete  $c$  from  $C_k$ 
```

# Derive rules from frequent itemsets

---

- Frequent itemsets  $\neq$  association rules
- One more step is required to find association rules
- For each frequent itemset  $X$ ,  
For each **proper nonempty** subset  $A$  of  $X$ ,
  - Let  $B = X - A$
  - $A \rightarrow B$  is an association rule if
    - Confidence ( $A \rightarrow B$ )  $\geq min\_conf$ ,  
where support ( $A \rightarrow B$ ) = support (AB), and  
confidence ( $A \rightarrow B$ ) = support (AB) / support (A)

# Example – deriving rules from frequent itemsets

---

- Suppose 234 is frequent, with supp=50%
  - Proper nonempty subsets: 23, 24, 34, 2, 3, 4, with supp=50%, 50%, 75%, 75%, 75%, 75% respectively
  - These generate these association rules:
    - 23 => 4, confidence=100%
    - 24 => 3, confidence=100%
    - 34 => 2, confidence=67% =  $(N * 50\%) / (N * 75\%)$
    - 2 => 34, confidence=67%
    - 3 => 24, confidence=67%
    - 4 => 23, confidence=67%
    - All rules have support = 50%

Q: is there any optimization (e.g., pruning) for this step?

# Deriving rules

---

- To recap, in order to obtain  $A \rightarrow B$ , we need to have  $\text{Support}(AB)$  and  $\text{Support}(A)$
- This step is not as time-consuming as frequent itemsets generation
  - Why?
- It's also easy to speedup using techniques such as parallel processing.
  - How?
- Do we really need candidate generation for deriving association rules?
  - Frequent-Pattern Growth (FP-Tree)

# 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  $i_1 i_2 \dots i_{100}$ 
    - # of scans: **100**
    - # of Candidates:  $\binom{100}{1} + \binom{100}{2} + \dots + \binom{100}{100} = 2^{100} - 1$
- Bottleneck: candidate-generation-and-test

*Can we avoid candidate generation **altogether**?*

---

- FP-growth

# No Pain, No Gain

	<u>Java</u>	<u>Lisp</u>	<u>Scheme</u>	<u>Python</u>	<u>Ruby</u>
Alice	X				X
Bob				X	X
Charlie	X			X	X
Dora		X	X		

minsup = 1

## ■ Apriori:

- $L_1 = \{J, L, S, P, R\}$
- $C_2 = \text{all the } ({}^5_2) \text{ combinations}$ 
  - Most of  $C_2$  do not contribute to the result
  - There is no way to tell because

# No Pain, No Gain

	<u>Java</u>	<u>Lisp</u>	<u>Scheme</u>	<u>Python</u>	<u>Ruby</u>
Alice	X				X
Bob				X	X
Charlie	X			X	X
Dora		X	X		

## Ideas:

- Keep the **support set** for each frequent itemset
- DFS

minsup = 1

$J \rightarrow JL?$

$J \rightarrow ???$

Only need to look at support set for J

{A, C}

J

$\emptyset$

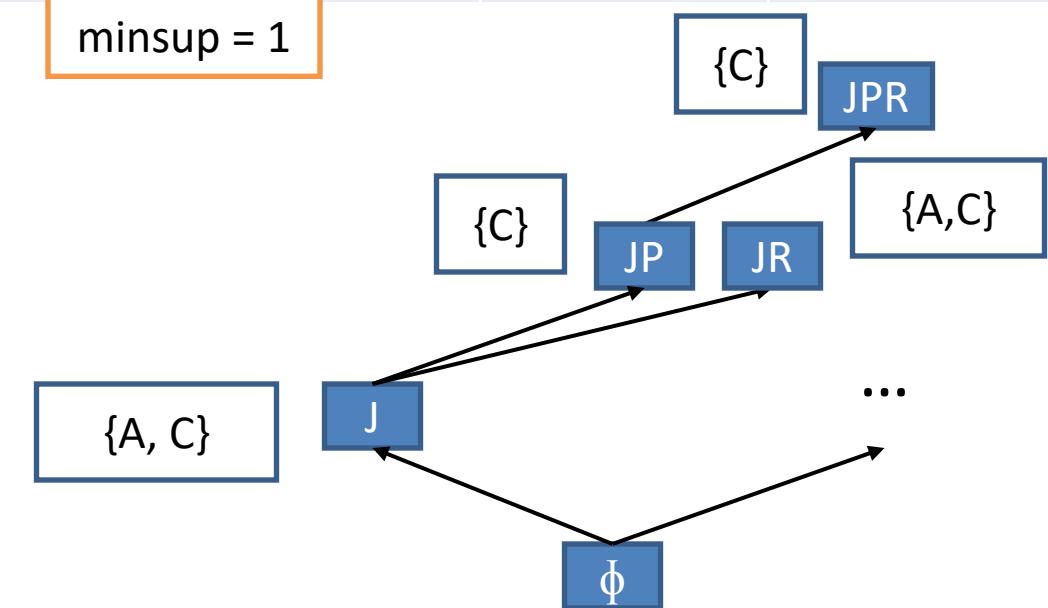
# No Pain, No Gain

	<u>Java</u>	<u>Lisp</u>	<u>Scheme</u>	<u>Python</u>	<u>Ruby</u>
Alice	X				X
Bob				X	X
Charlie	X			X	X
Dora		X	X		

## Ideas:

- Keep the support set for each frequent itemset
- DFS

minsup = 1



# Notations and Invariants

---

- ConditionalDB:
  - $DB|p = \{t \in DB \mid t \text{ contains itemset } p\}$
  - $DB = DB|\emptyset$  (i.e., conditioned on nothing)
  - Shorthand:  $DB|px = DB|(p \cup x)$
- $\text{SupportSet}(p \cup x, DB) = \text{SupportSet}(x, DB|p)$ 
  - $\{x \mid x \bmod 6 = 0 \wedge x \in [100]\} =$   
 $\{x \mid x \bmod 3 = 0 \wedge x \in \text{even}([100])\}$
- A FP-tree|p is equivalent to a DB|p
  - One can be converted to another
  - Next, we illustrate the alg using conditionalDB

# FP-tree Essential Idea /1

- Recursive algorithm again!

- Freq**Itemsets**(DB|p):

- $X = \text{FindFrequentItems}(DB|p)$

easy task, as  
only items (not  
itemsets) are  
needed

all frequent itemsets in  
DB|p belong to one of  
the following  
categories:

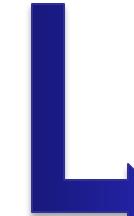
output  $\{ (x \ p) \mid x \in X \}$

- Foreach  $x$  in  $X$

- $DB^*|px = \text{GetConditionalDB}^+(DB^*|p, x)$

- 

- Freq**Itemsets**(DB $^*$ |px)



*obtained  
via  
recursion*

patterns  $\sim x_i p$   
patterns  $\sim \star px_1$   
patterns  $\sim \star px_2$   
patterns  $\sim \star px_i$   
patterns  $\sim \star px_n$

# No Pain, No Gain

DB J	Java	Lisp	Scheme	Python	Ruby
Alice	X				X
Charlie	X			X	X

minsup = 1

- Freq**Itemsets**(DB|J):
  - $\{P, R\} \leftarrow \text{FindFrequentItems}(DB|J)$
  - Output  $\{JP, JR\}$
  - Get  $DB^*|JP$ ; Freq**Itemsets**( $DB^*|JP$ )
  - Get  $DB^*|JR$ ; Freq**Itemsets**( $DB^*|JR$ )
  - // Guaranteed no other frequent itemset in DB|J

# FP-tree Essential Idea /2

- Freq**Itemsets**(DB|p):

- If boundary condition, then ...
- X = FindFrequent**Items**(DB|p)
- [optional] DB\*|p = PruneDB(DB|p, X)  
output { (x p) | x ∈ X }
- Foreach **x** in X
  - DB\*|px = GetConditionalDB+(DB\*|p, x)
  - [optional] if DB\*|px is degenerated, then powerset(DB\*|px)
  - Freq**Itemsets**(DB\*|px)

Also output each item in X (appended with the conditional pattern)

Remove items not in X; potentially reduce # of transactions ( $\emptyset$  or dup). Improves the efficiency.

Also gets rid of items already processed before x → *avoid duplicates*

# Lv 1 Recursion

- $\text{minsup} = 3$

F	C	A	D	G	I	M	P
A	B	C	F	L	M	O	
B	F	H	J	J	O	W	
B	C	K	S	P			
A	F	C	E	L	P	M	N

DB

 $X = \{F, C, A, B, M, P\}$ 

Output: F, C, A, B, M, P

F	C	A	M	P
F	C	A	B	M
F	B			
C	B	P		
F	C	A	M	P

DB\*

F	C	A	M	P
C	B	P		
F	C	A	M	P

DB\*|P

DB\*|M (sans P)

DB\*|B (sans MP)

DB\*|A (sans BMP)

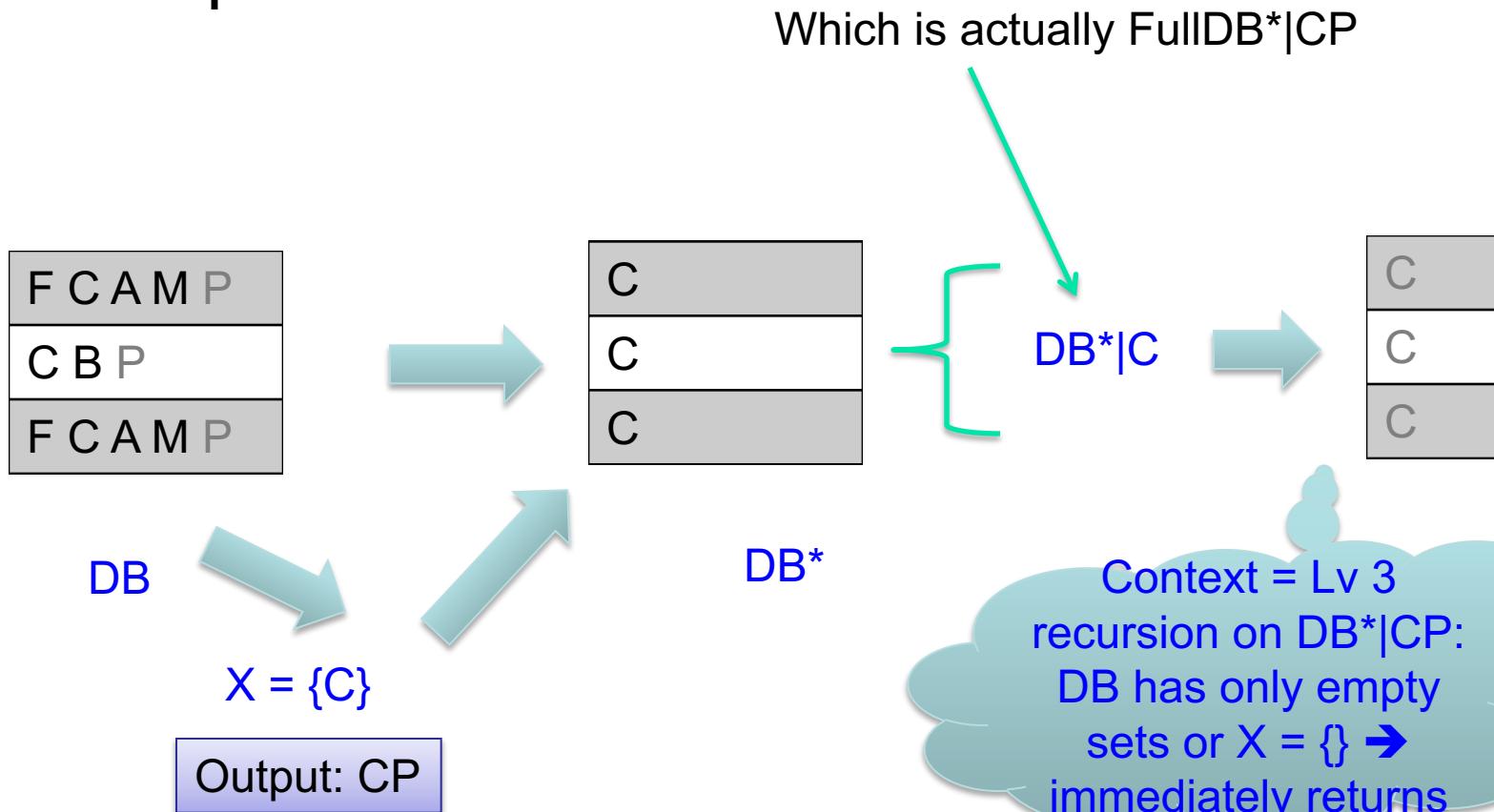
DB\*|C (sans ABMP)

DB\*|F (sans CABMP)

F	C	A
F	C	A
F	C	A

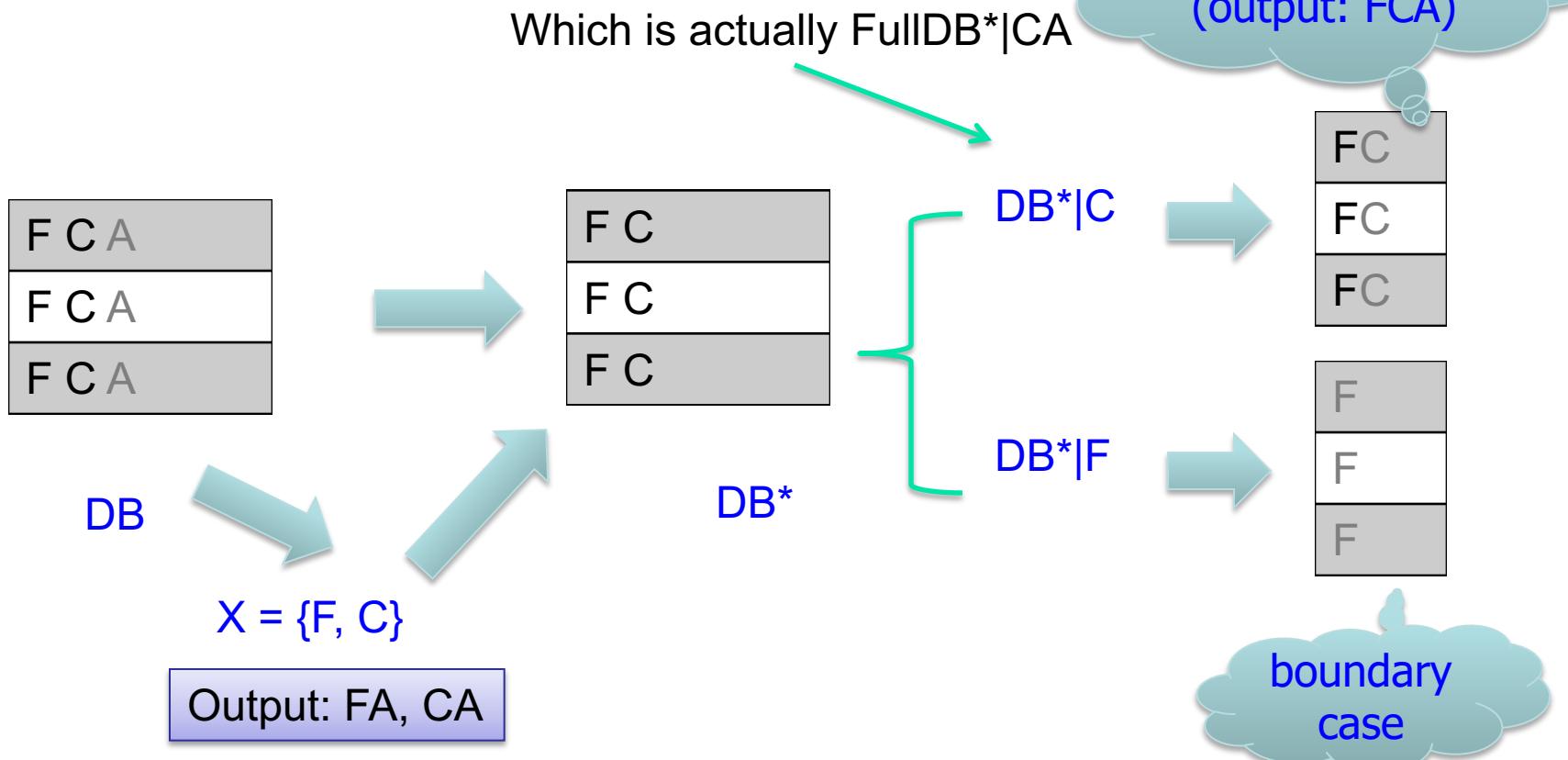
# Lv 2 Recursion on DB\*|P

- $\text{minsup} = 3$



# Lv 2 Recursion on DB\*|A (sans ...)

- $\text{minsup} = 3$



# Different Example: Lv 2 Recursion on DB\*|P

- $\text{minsup} = 2$

F	C	A	M	P
F	C	B	P	
F	A	P		



F	C	A
F	C	
F	A	

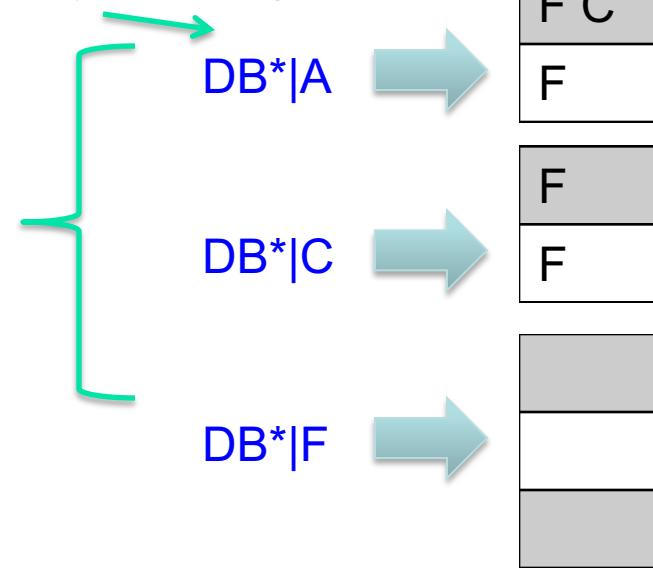
DB\*

DB

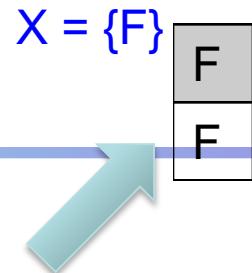
$X = \{F, C, A\}$

Output: FP, CP, AP

Which is actually FullDB\*|AP



Output: FAP



# I will give you back the FP-tree

---

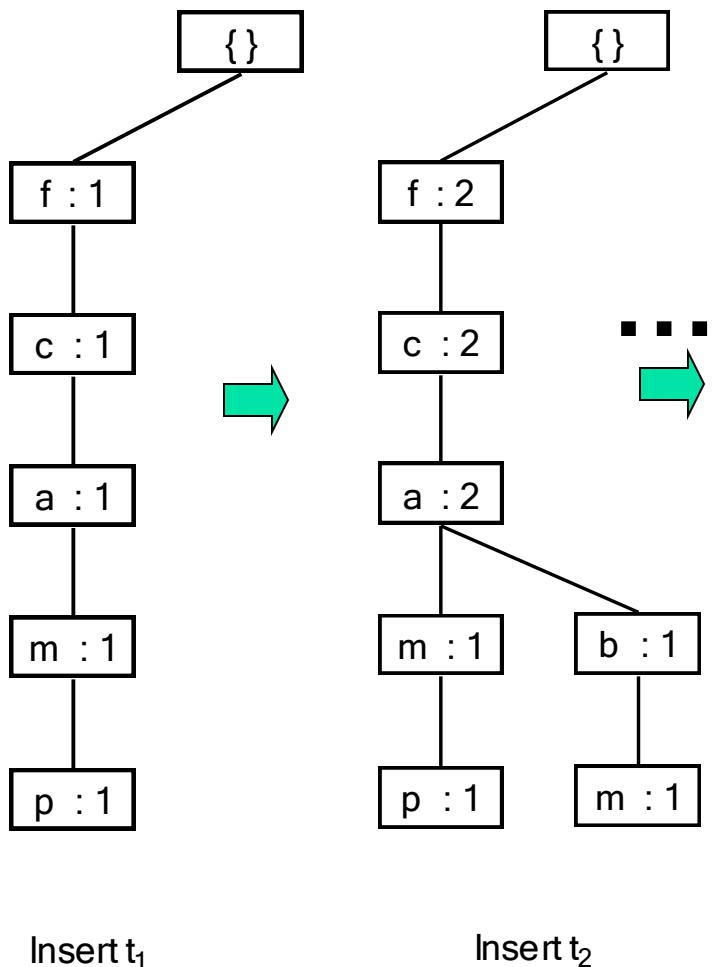
- An FP-tree tree of DB consists of:
  - A fixed **order** among items in DB
  - A prefix, **threaded** tree of **sorted** transactions in DB
  - Header table: (item, freq, ptr)
- When used in the algorithm, the input DB is always pruned (c.f., PruneDB())
  - Remove infrequent items
  - Remove infrequent items in every transaction

# FP-tree Example

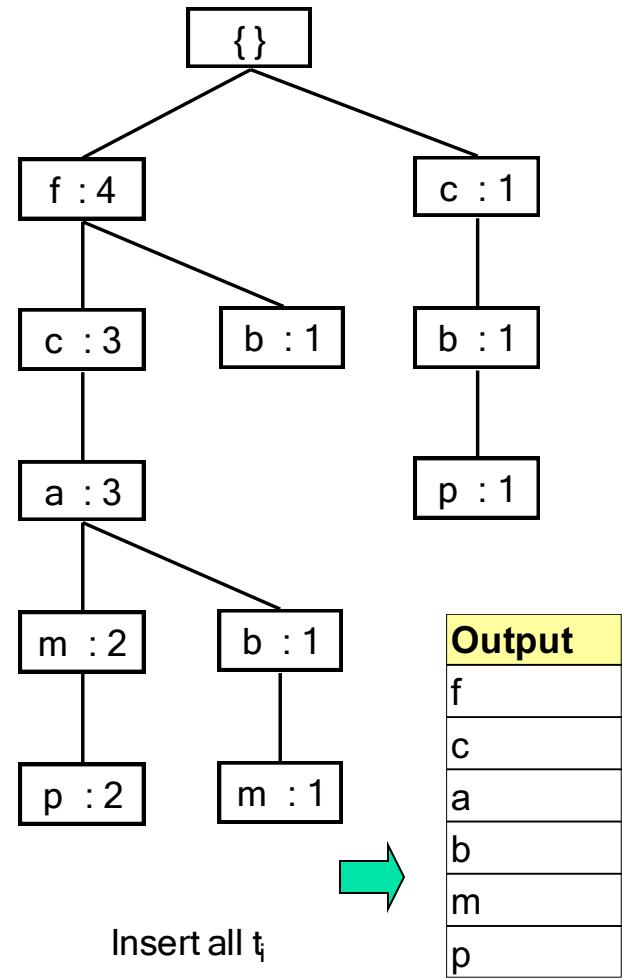
minsup = 3

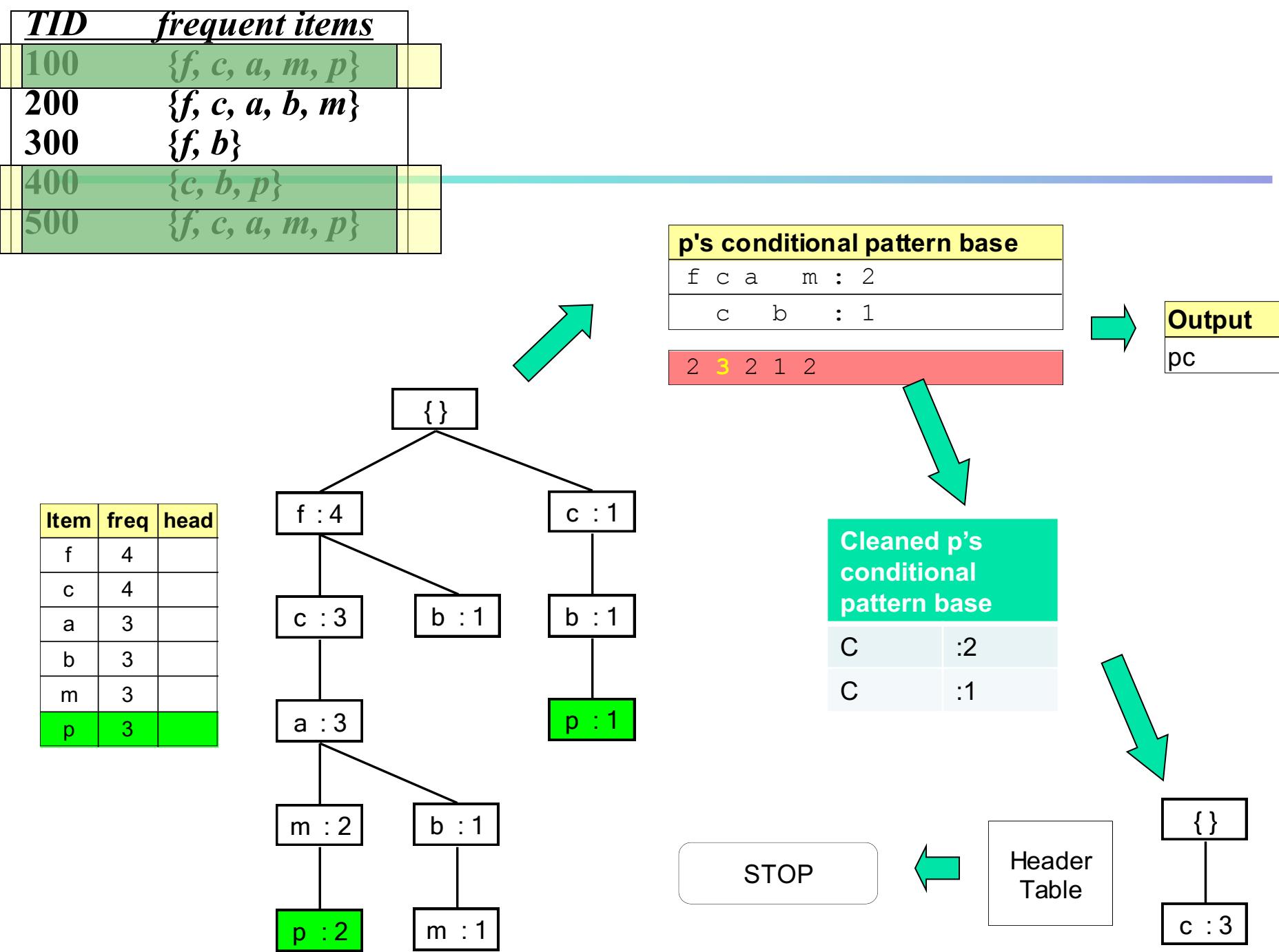
<i>TID</i>	<i>Items bought</i>	<i>(ordered) frequent items</i>
100	{f, a, c, d, g, i, m, p}	{f, c, a, m, p}
200	{a, b, c, f, l, m, o}	{f, c, a, b, m}
300	{b, f, h, j, o, w}	{f, b}
400	{b, c, k, s, p}	{c, b, p}
500	{a, f, c, e, l, p, m, n}	{f, c, a, m, p}

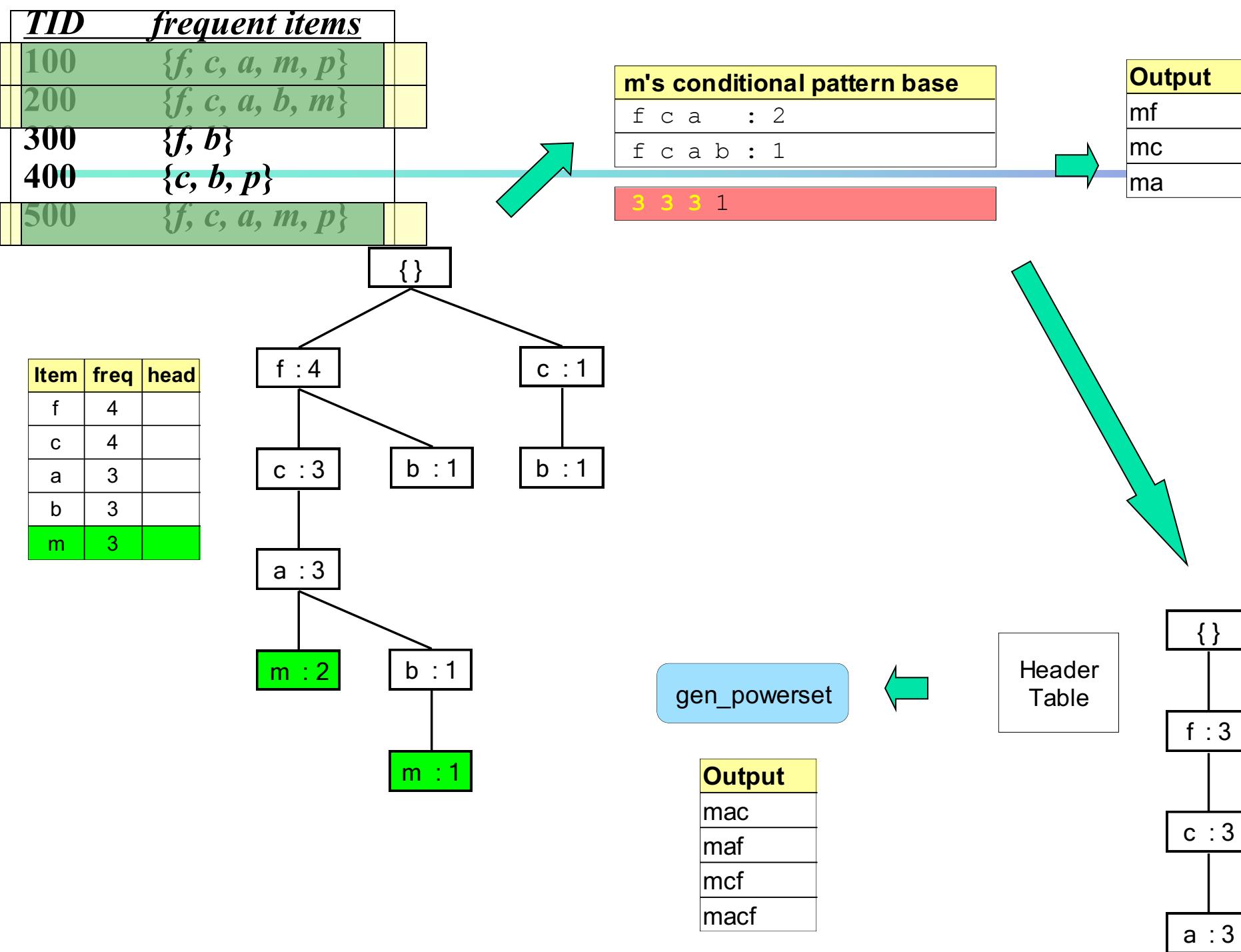
<i>TID</i>	<i>Items bought</i>	<i>(ordered) frequent items</i>
100	{f, a, c, d, g, i, m, p}	{f, c, a, m, p}
200	{a, b, c, f, l, m, o}	{f, c, a, b, m}
300	{b, f, h, j, o, w}	{f, b}
400	{b, c, k, s, p}	{c, b, p}
500	{a, f, c, e, l, p, m, n}	{f, c, a, m, p}



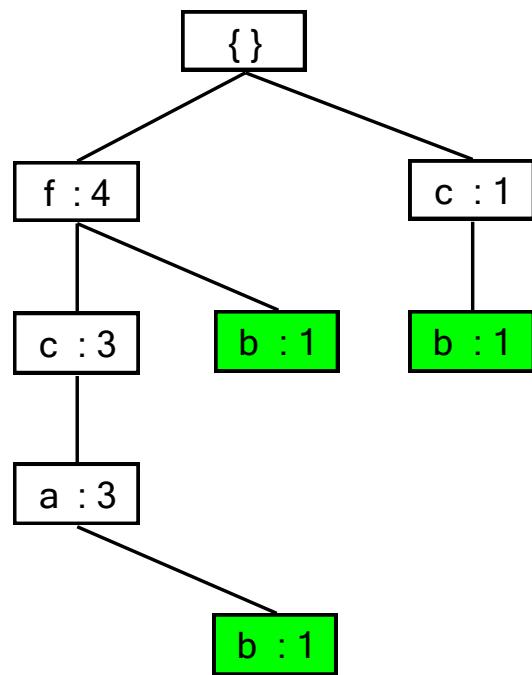
Item	freq	head
f	4	
c	4	
a	3	
b	3	
m	3	
p	3	







Item	freq	head
f	4	
c	4	
a	3	
b	3	

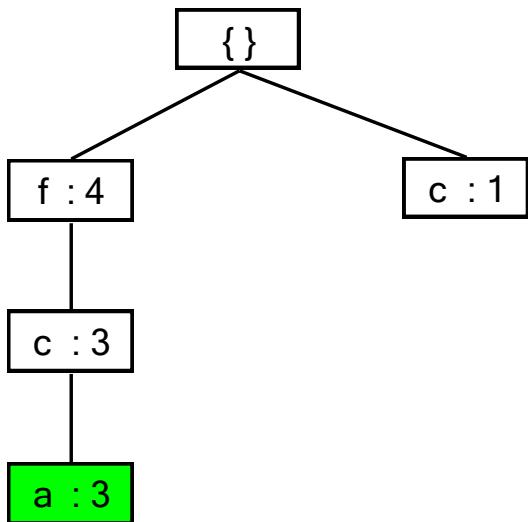


b's conditional pattern base

f	c	a	:	1
f			:	1
c			:	1
2 2 1				

STOP

Item	freq	head
f	4	
c	4	
a	3	



**a's conditional pattern base**

f	c	:	3
3 3			

**Output**

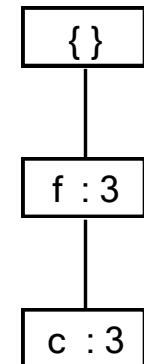
af
ac

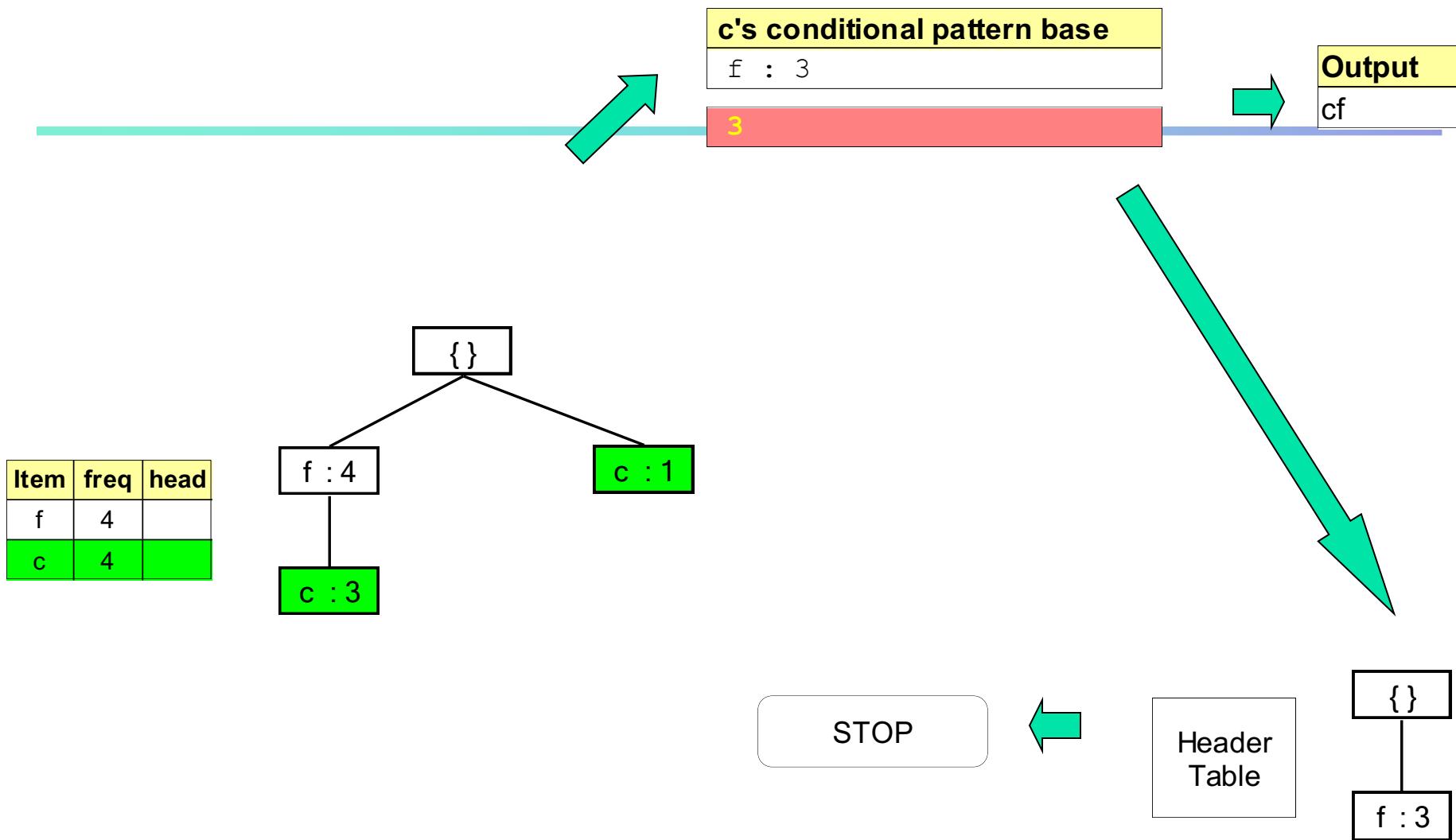
gen\_powerset

Header Table

**Output**

acf
-----







STOP



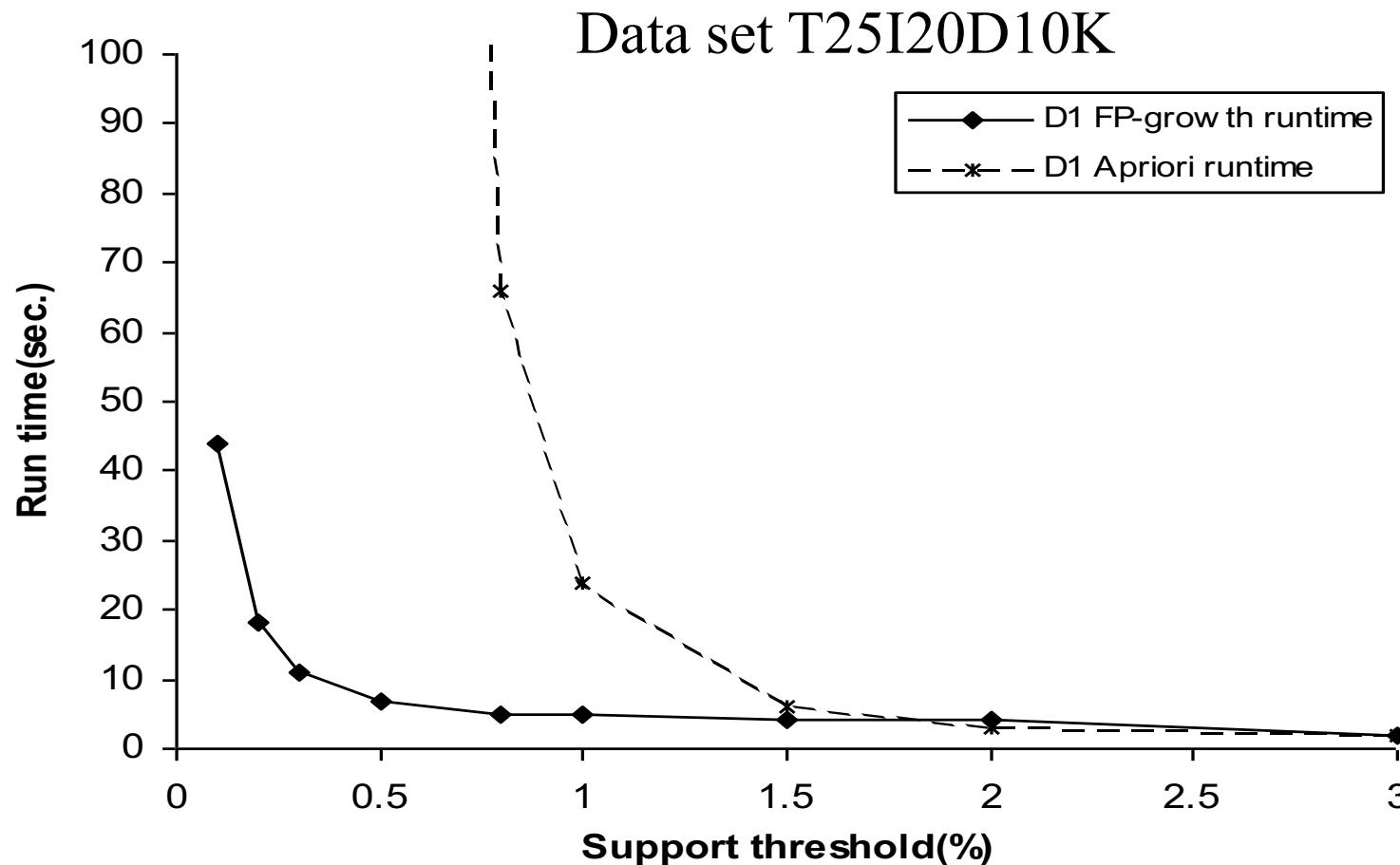
{ }

f : 4

Item	freq	head
f	4	

# FP-Growth vs. Apriori: 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