COMP9318: Data Warehousing and Data Mining

— L ing —

- 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 r
  - Frequent patt ns, sequence, etc.) that occurs frequently in a AIS93]
- Motivation: finding regularitie
  - 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?

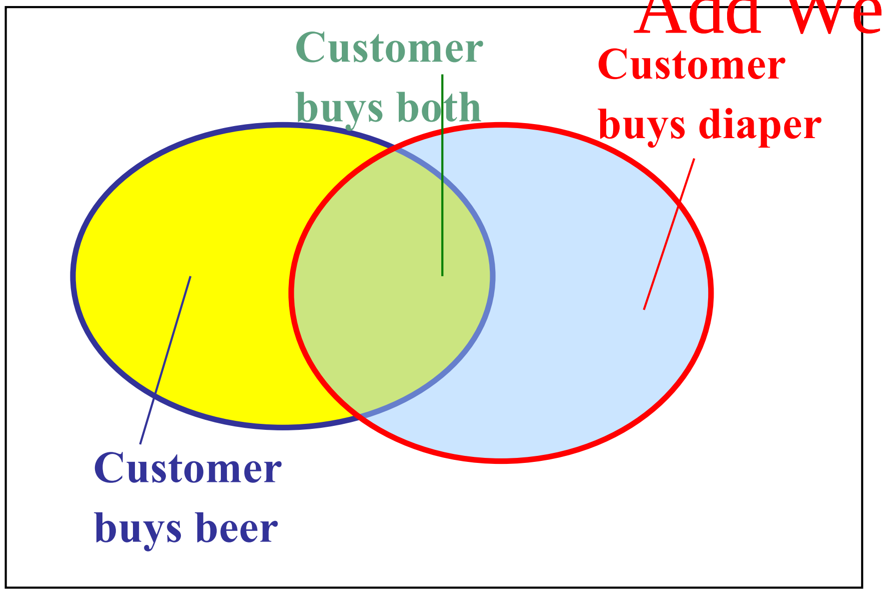
Why Is Frequent Pattern or Assoiciation Mining an Essential Task in Data Mining?

- Foundation for many essential data mining tasks
  - Association, correlation, causality
  - Sequential patterns, temporal or cyclic association, partial period media association
  - Associative c alysis, iceberg cube, fascicles (semantic data co)
- 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
40	{ B,



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

Let min_support = 50%, min_conf = 70%:

frequent itemset

sup(AC) = 2

association rule

~~*A* ➔ *C* (50%, 66.7%)~~

C ➔ *A* (50%, 100%)

Mining Association Rules—an Example

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

Min. support 50%
Min. confidence 50%

nt pattern	Support
{A}	75%
}	50%
}	50%
{A, C}	50%

For rule $A \rightarrow C$:

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

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

major computation challenge: calculate the support of itemsets

← The ***frequent itemset mining*** problem

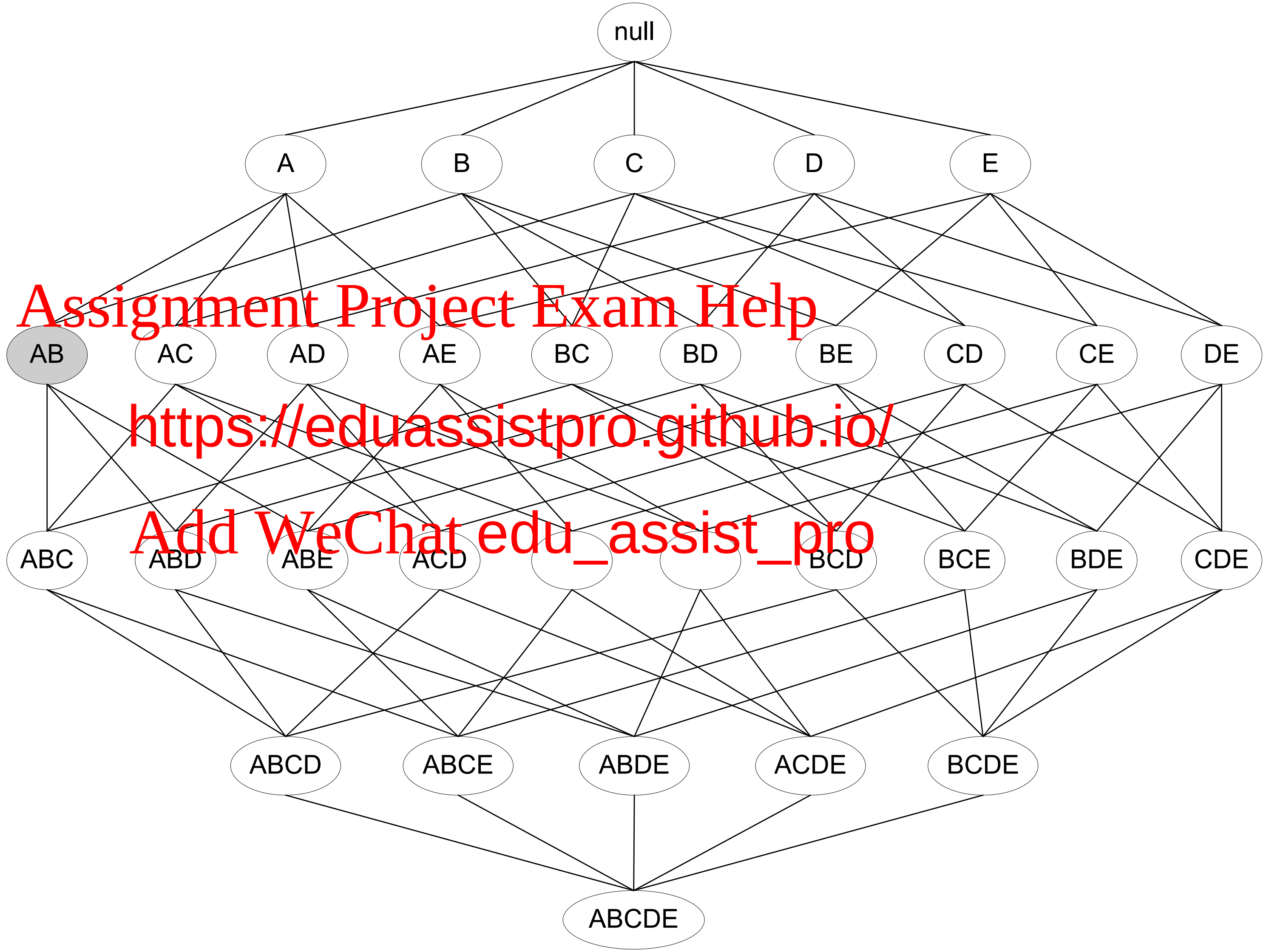
- Algorithms for scalable mining of (single-dimensional Boolean) association rules in transactional databases

Association Rule Mining Algorithms

Candidate Generation & Verification

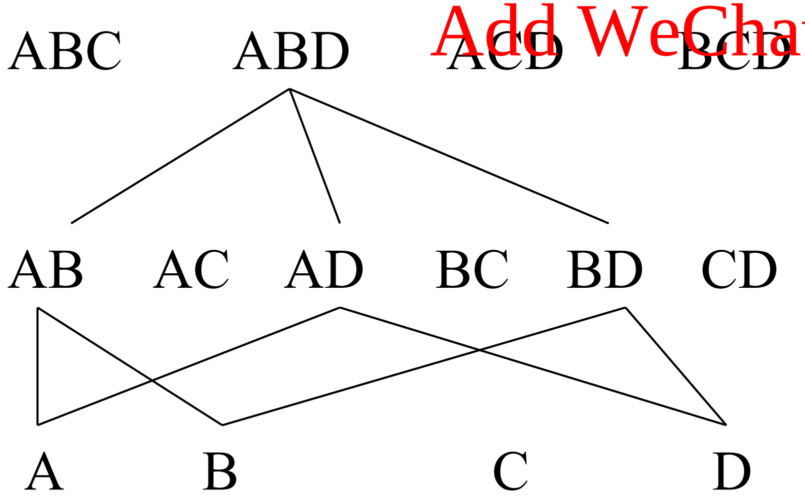
- Naïve algorithm
  - Enumerate all possible itemsets and check their support against *min_sup*
  - Generate a ... and check their confide... against *min_conf*
- The Apriori property
  - Apriori Algorithm
  - FP-growth Algorithm

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



Apriori Property

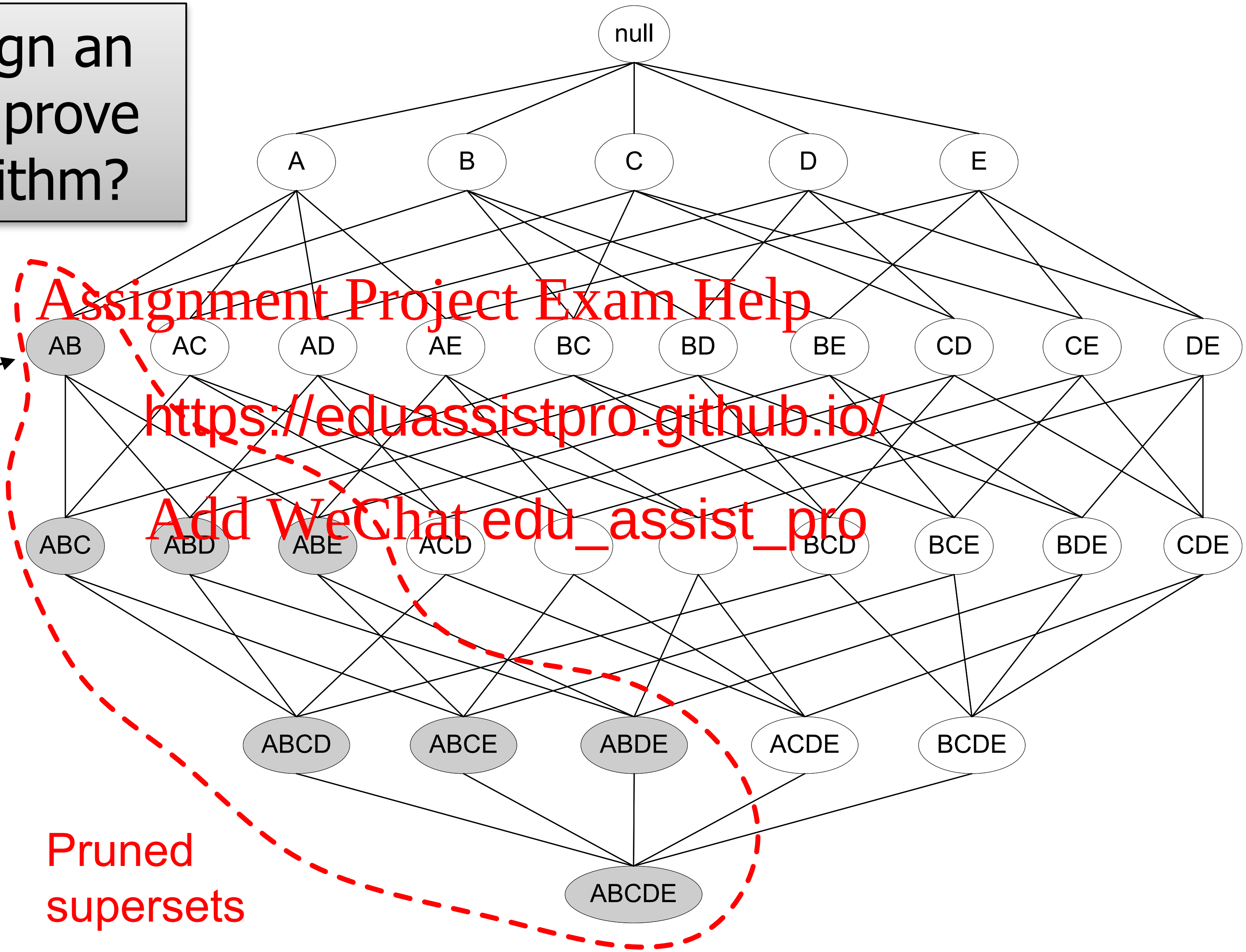
- A *frequent* (used to be called *large*) *itemset* is an itemset whose support is ≥ min_sup.
- Apriori property (downward closure): any subsets of a frequent ... quent itemsets
- Aka the anti- ... of support



...ny 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 [Ag 4]

1. $C_k \leftarrow$ Perf idate gener (from singleton items

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 = {frequent it
for (k = 1; L_k !=∅; k++) do b
     C_k+1 = candidates generat                  ;
     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 = candidates in C_k+1 with min_support
end
return ∪_k L_k;
```

The Apriori Algorithm—An Example

minsup = 50%

Database TDB

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

1st scan

C_1

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

L_1

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

2

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

2

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

L_2

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

C_3

Itemset
{B, C, E}

3rd 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-

- 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$_1$,*** … ***tem$_{k-1}$***

from ***L_{k-1} p, L_{k-}*** …
where ***p.item$_1$=q.item$_1$, …, p.*** … ***em$_{k-2}$, 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 != association rules
- One more step is required to find association rules

- For each freq

  For each prop t A of X,

  - Let B = X - A
  - A → B is an association rule if
    - Confidence (A → B) ≥ *min_conf*,
      where support (A → B) = support (AB), and
      confidence (A → 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,
    - 24 => 3,   confidence=1
    - 34 => 2,   confidence=6
    - 2 => 34,   confidence=67%
    - 3 => 24,   confidence=67%
    - 4 => 23,   confidence=67%
    - All rules have support = 50%

0%)/(N*75%)

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

Deriving rules

- To recap, in order to obtain A → B, we need to have Support(AB) and Support(A)
- This step is not as time-consuming as frequent itemsets generation
  - Why?
- It's also eas techniques such as parallel process
  - How?
- Do we really need candidate generation for deriving association rules?
  - <u>Frequent-Pattern Growth (FP-Tree)</u>

Bottleneck of Frequent-pattern Mining

- Multiple database scans are costly
- Mining long patterns needs many passes of scanning and andidates

  - To find fre 0

    - # of scans: 100

    - # of Candidates: $\binom{100}{1} + \quad_{2} + \ldots + \binom{100}{100} = 2^{100} - 1$
- Bottleneck: candidate-generation-and-test

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

- FP-growth

No Pain, No Gain

	Java	Lisp	Scheme	Python	Ruby
Alice	X				X
Bob				X	X
Charlie	X			X	X
Dora					

minsup =

- Apriori:
  - L1 = {J, L, S, P, R}
  - C2 = all the $\binom{5}{2}$ combinations
    - Most of C2 do not contribute to the result
    - There is no way to tell because

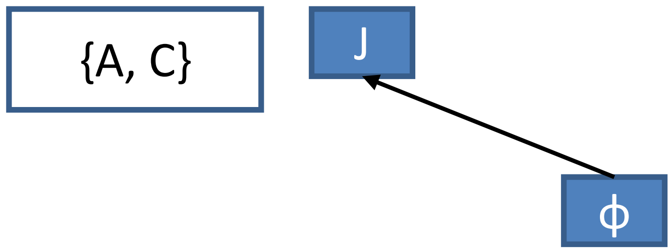
No Pain, No Gain

	Java	Lisp	Scheme	Python	Ruby
Alice	X				X
Bob				X	X
Charlie	X			X	X
Dora					

minsup =

Ideas:
- Keep the support set for each frequent itemset
- DFS

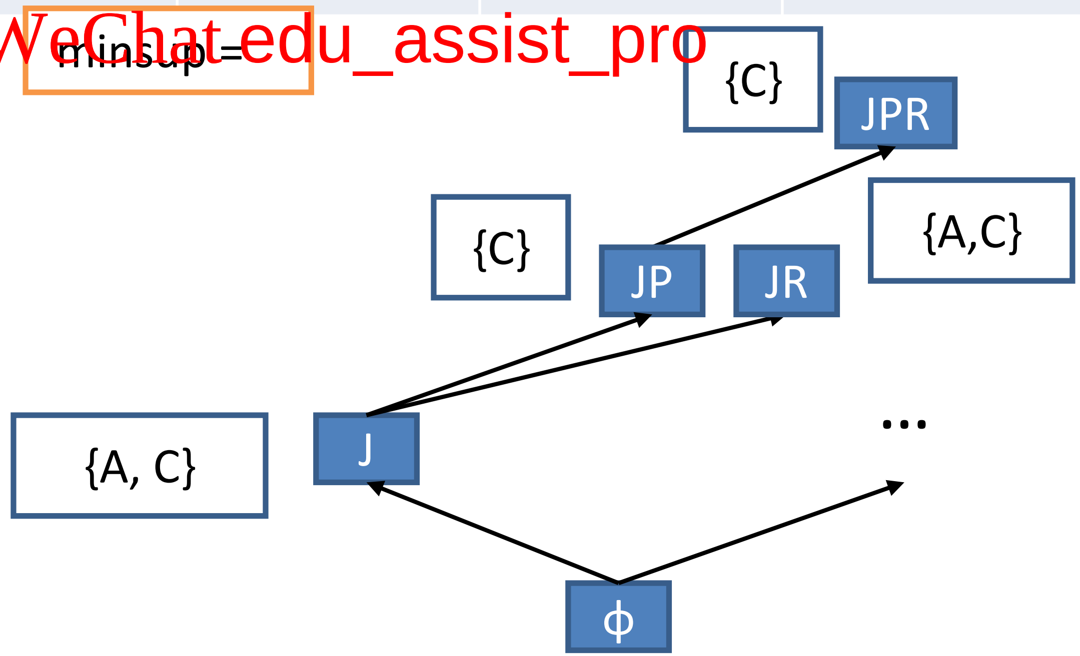
J ➔ JL?
J ➔ ???
Only need to look at support set for J



No Pain, No Gain

	Java	Lisp	Scheme	Python	Ruby
Alice	X				X
Bob				X	X
Charlie	X			X	X
Dora					

Ideas:
- Keep the support set for each frequent itemset
- DFS



Notations and Invariants

- ConddiditonalDB:
  - DB|p = {t ∈ DB | t contains itemset p}
  - DB = DB|∅ (i.e., conditioned on nothing)
  - Shorthand:)
- SupportSet(p Set(x, DB|p)
  - {x | x mod 6 = 0 ∧ x =

    {x | x mod 3 = 0 ∧ x ∈ even([100]) }
- A FP-tree 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 = Find**Locall**)

    `output { (x p) | x ∈ X }`
  - Foreach x in X
    - DB*|px = GetConditionalDB$^+$(DB*|p, x)
    -
    - Freq**Itemsets**(DB*|px)

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

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

- patterns ~ $x_i p$
- patterns ~ ★px_1
- patterns ~ ★px_2
- patterns ~ ★px_i
- patterns ~ ★px_n

obtained via recursion

No Pain, No Gain

DB|J

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

- Freq**Itemsets**(
  - {**P**, **R**} ← Find**Locally**Freque B|J)
  - Output {JP, JR}
  - Get DB*|J**P**; Freq**Itemsets**(DB*|J**P**)
  - Get DB*|J**R**; Freq**Itemsets**(DB*|J**R**)
  - // Guaranteed no other frequent itemset in DB|J

FP-tree Essential Idea /2

- Freq**Itemsets**(DB|p):
  - If boundary con
  - X = Find**LocalI**)
  - [optional] DB*|p = PruneDB(

    $\boxed{\text{output } \{ (x\ p) \mid x \in X \}}$
  - 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 (∅ or dup). Improves the efficiency.

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

Grayed items are for illustration purpose only.

Lv 1 Recursion

- minsup = 3

F C A D G I M P
A B C F L M O
B F H 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
C B P
F C A M P

DB*

DB*|P

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

DB*|M (sans P)

DB*|B (sans MP)

DB*|A (sans BMP)

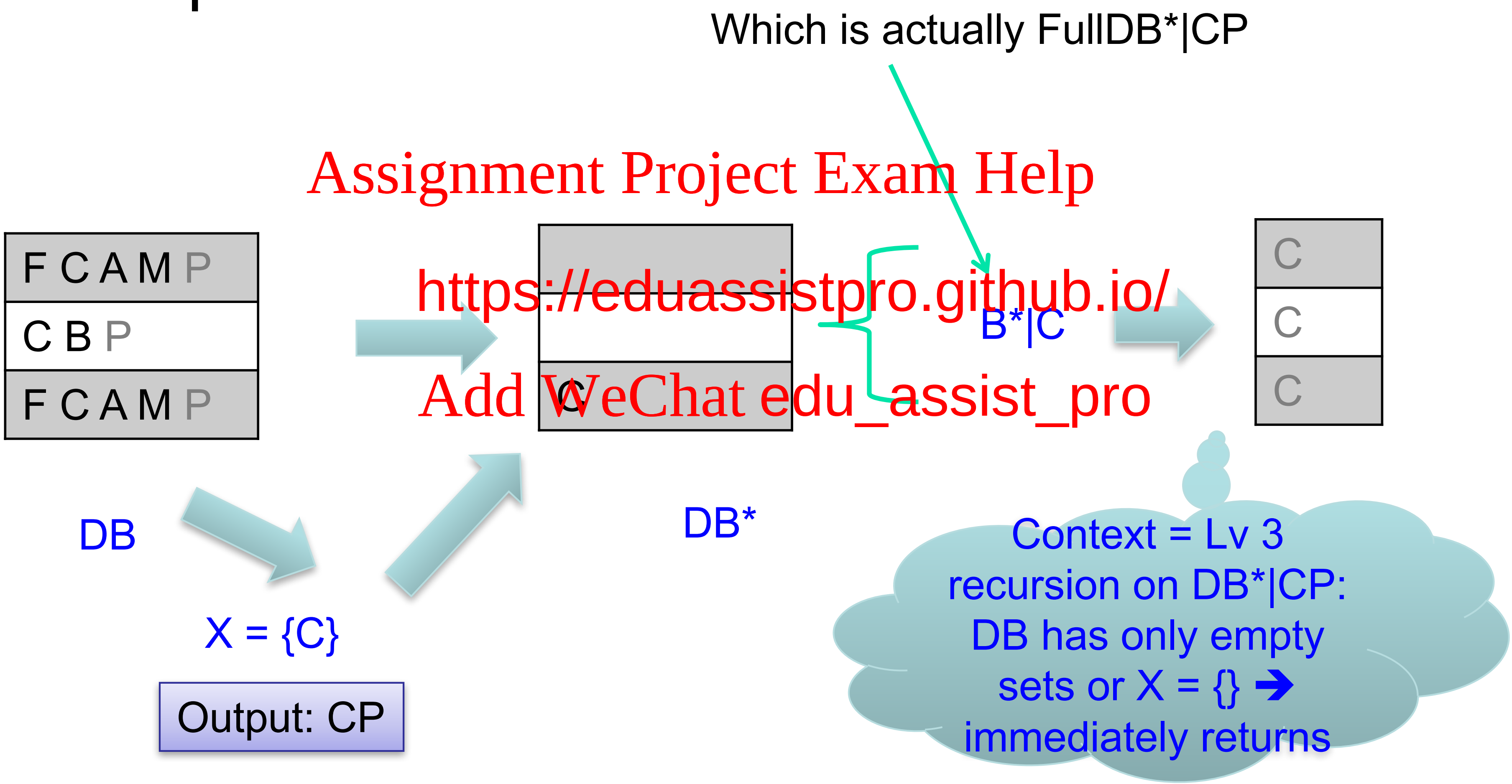
DB*|C (sans ABMP)

DB*|F (sans CABMP)

F C A
F C A
F C A

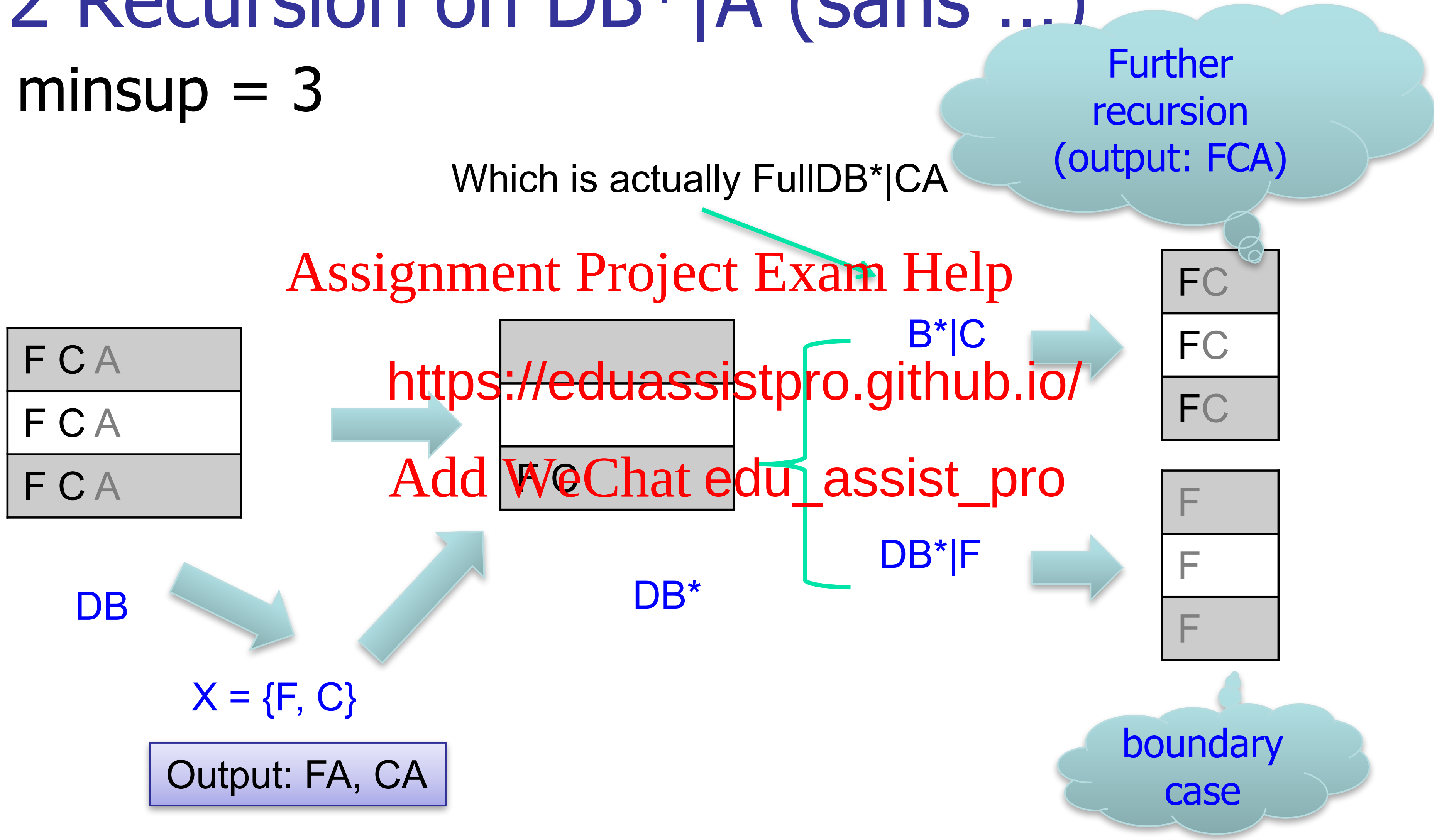
Lv 2 Recursion on DB*|P

- minsup = 3



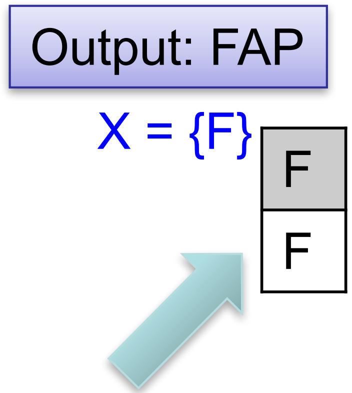
Lv 2 Recursion on DB*|A (sans ...)

- minsup = 3

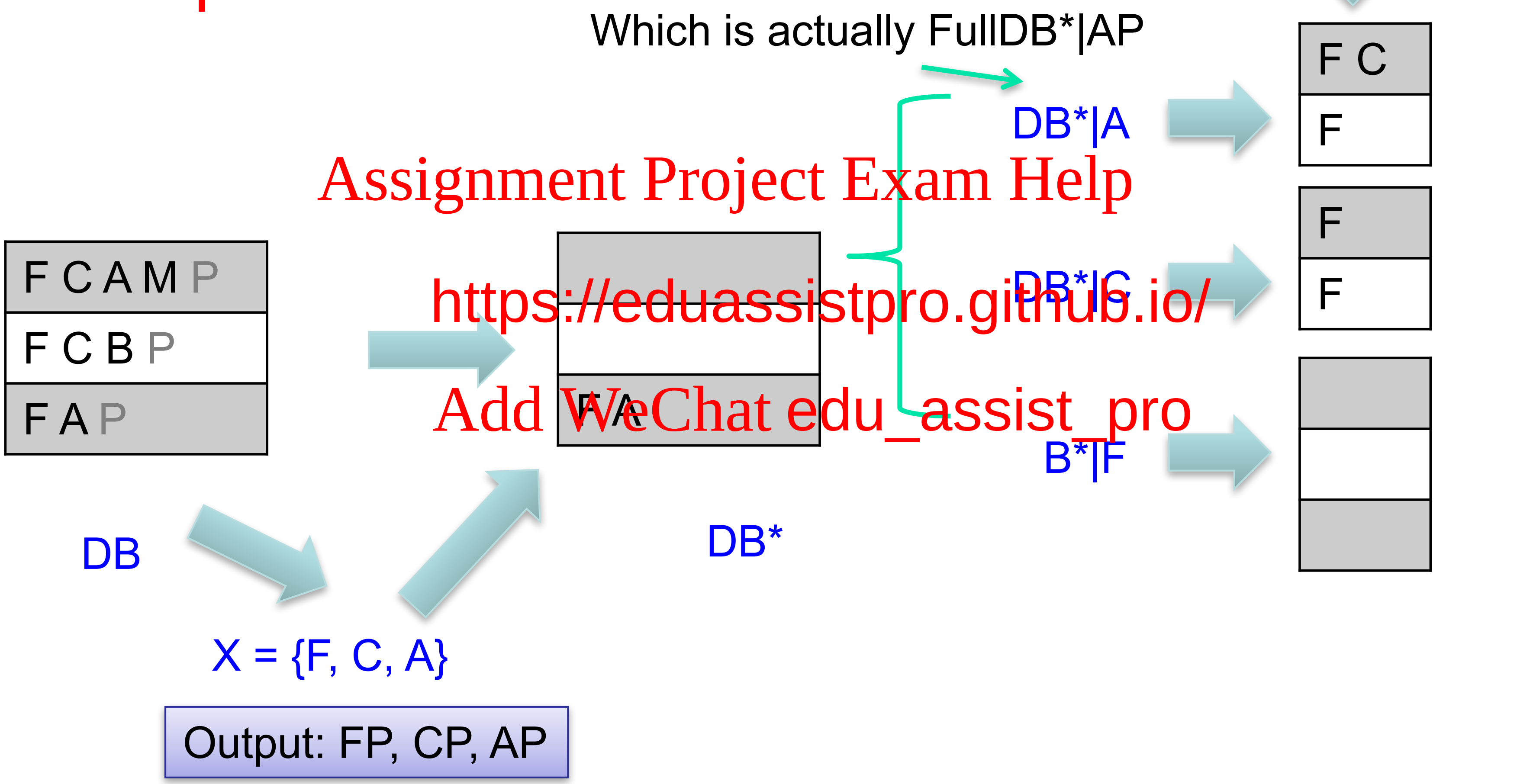


Different Example:

Lv 2 Recursion on DB*|P



- minsup = 2



I will give you back the FP-tree

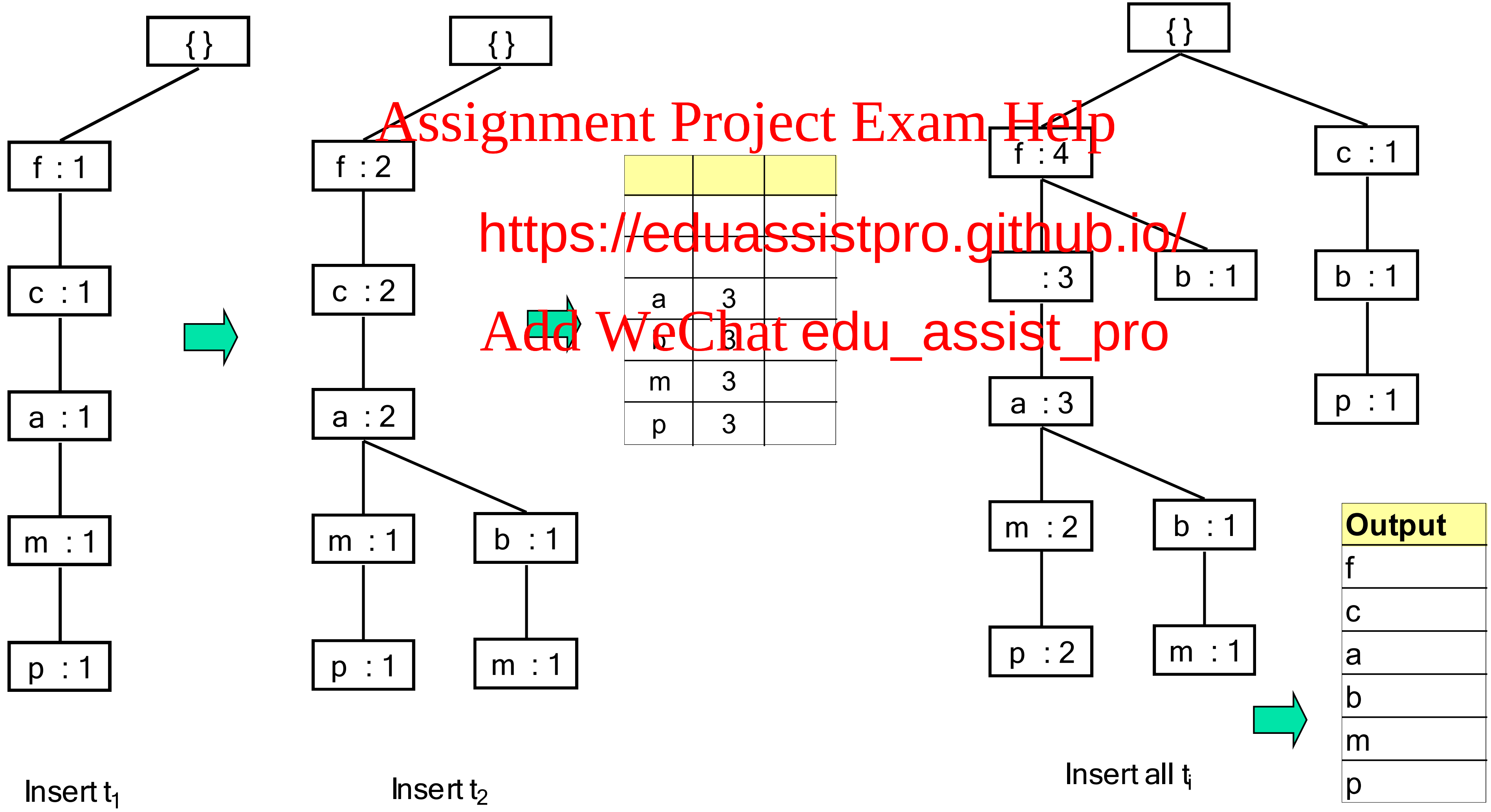
- An FP-tree tree of DB consists of:
  - A fixed **order** among items in DB
  - A prefix, **threaded** tree of **sorted** transactions in DB
  - Header tab
- When used in the algorit put DB is always pruned (c.f., PruneDB())
  - Remove infequent items
  - Remove infrequent items in every transaction

FP-tree Example

minsup = 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, 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}***

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}*



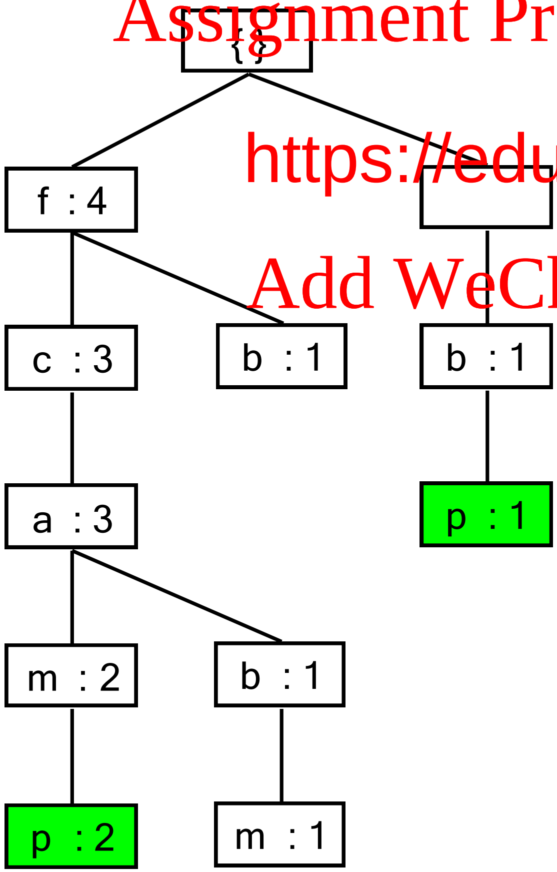
TID	*frequent items*
100	{*f, c, a, m, p*}
200	{*f, c, a, b, m*}
300	{*f, b*}
400	{*c, b, p*}
500	{*f, c, a, m, p*}

p's conditional pattern base
f c a m : 2
c b : 1

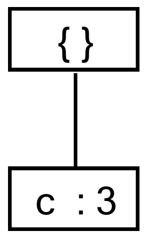
2 3 2 1 2

Output
pc

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

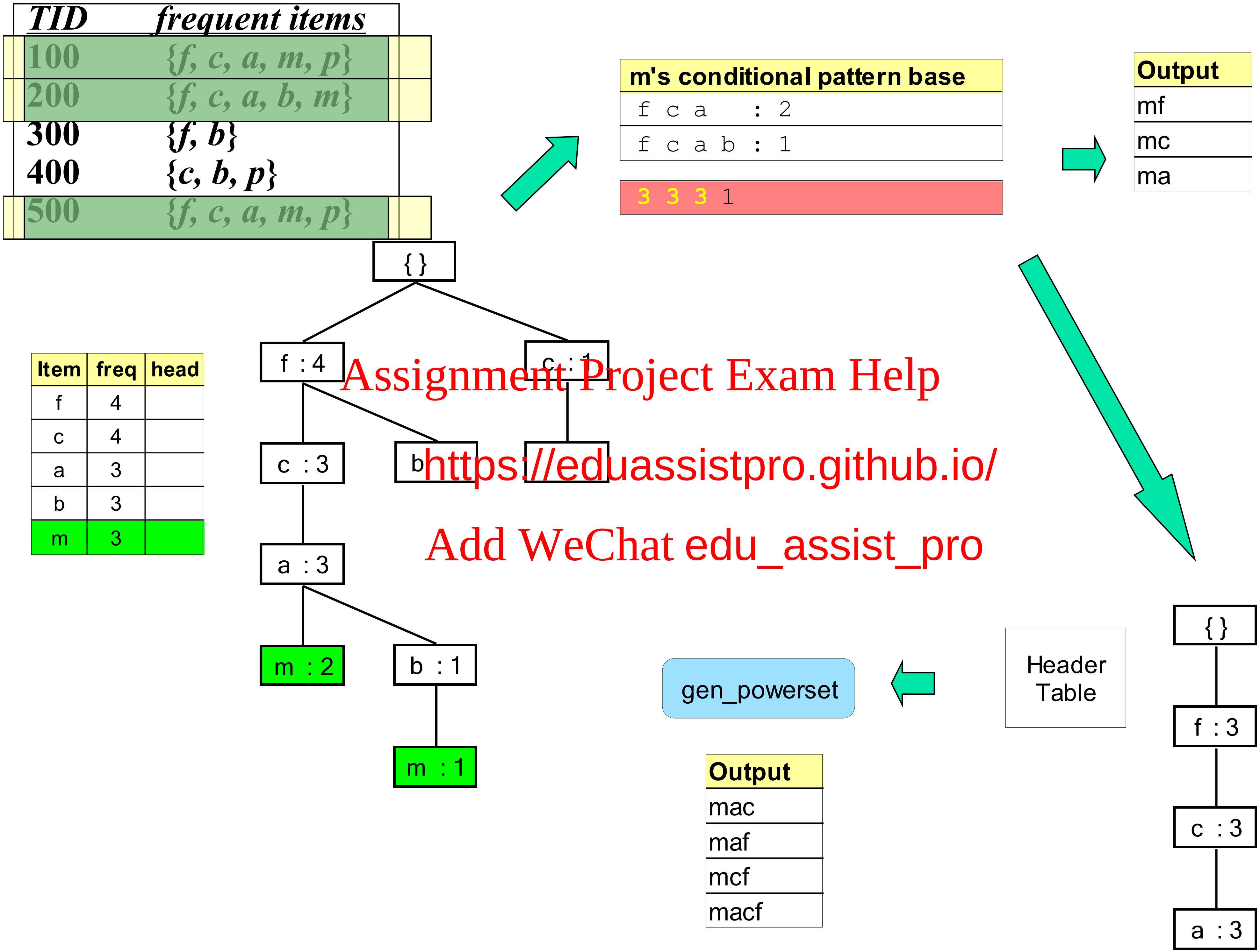


ed p's itional rn base	
C	:2
C	:1



Header Table

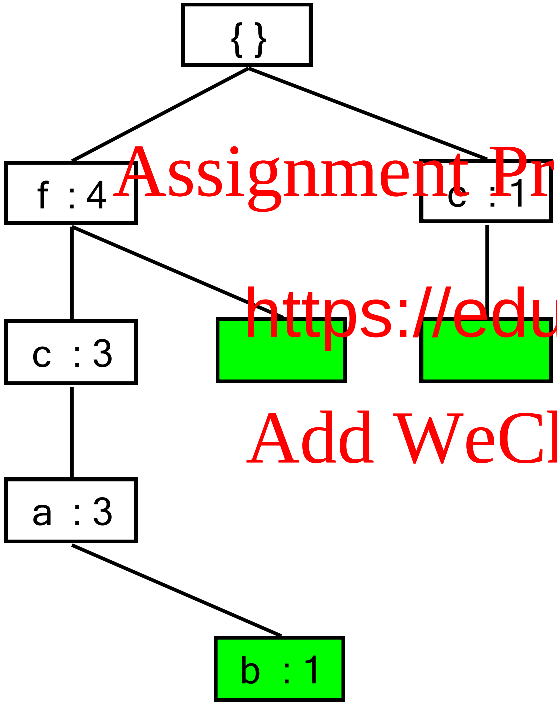
STOP


TID frequent items
100 {f, c, a, m, p}
200 {f, c, a, b, m}
300 {f, b}
400 {c, b, p}
500 {f, c, a, m, p}
m's conditional pattern base
f c a : 2
f c a b : 1
3 3 3 1
Output
mf
mc
ma
Item freq head
f 4
c 4
a 3
b 3
m 3
{}
f : 4
c : 1
c : 3
b
b
a : 3
m : 2
b : 1
m : 1
gen_powerset
Header Table
{}
f : 3
c : 3
a : 3
Output
mac
maf
mcf
macf

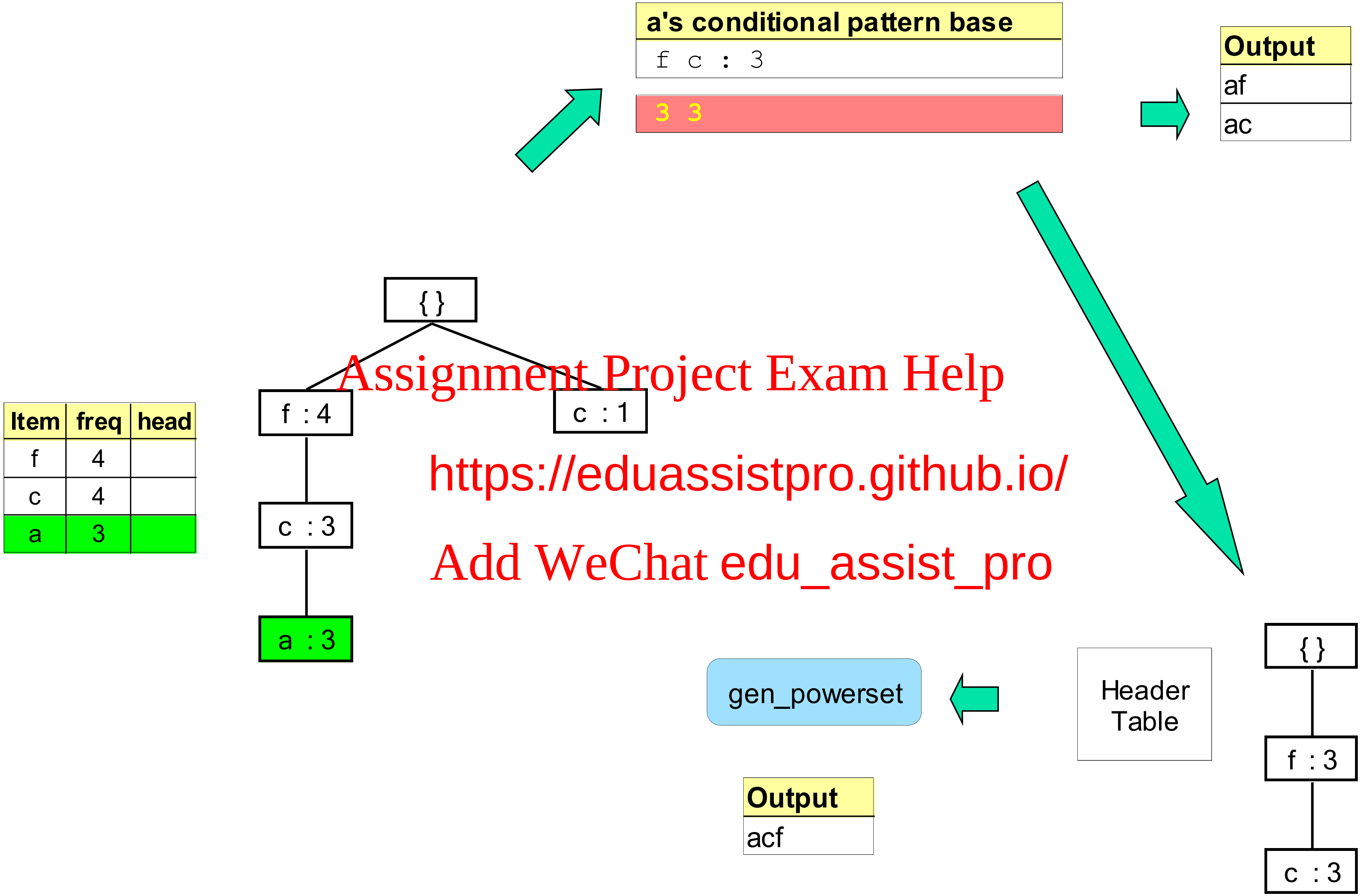
b's conditional pattern base
f c a : 1
f : 1
c : 1

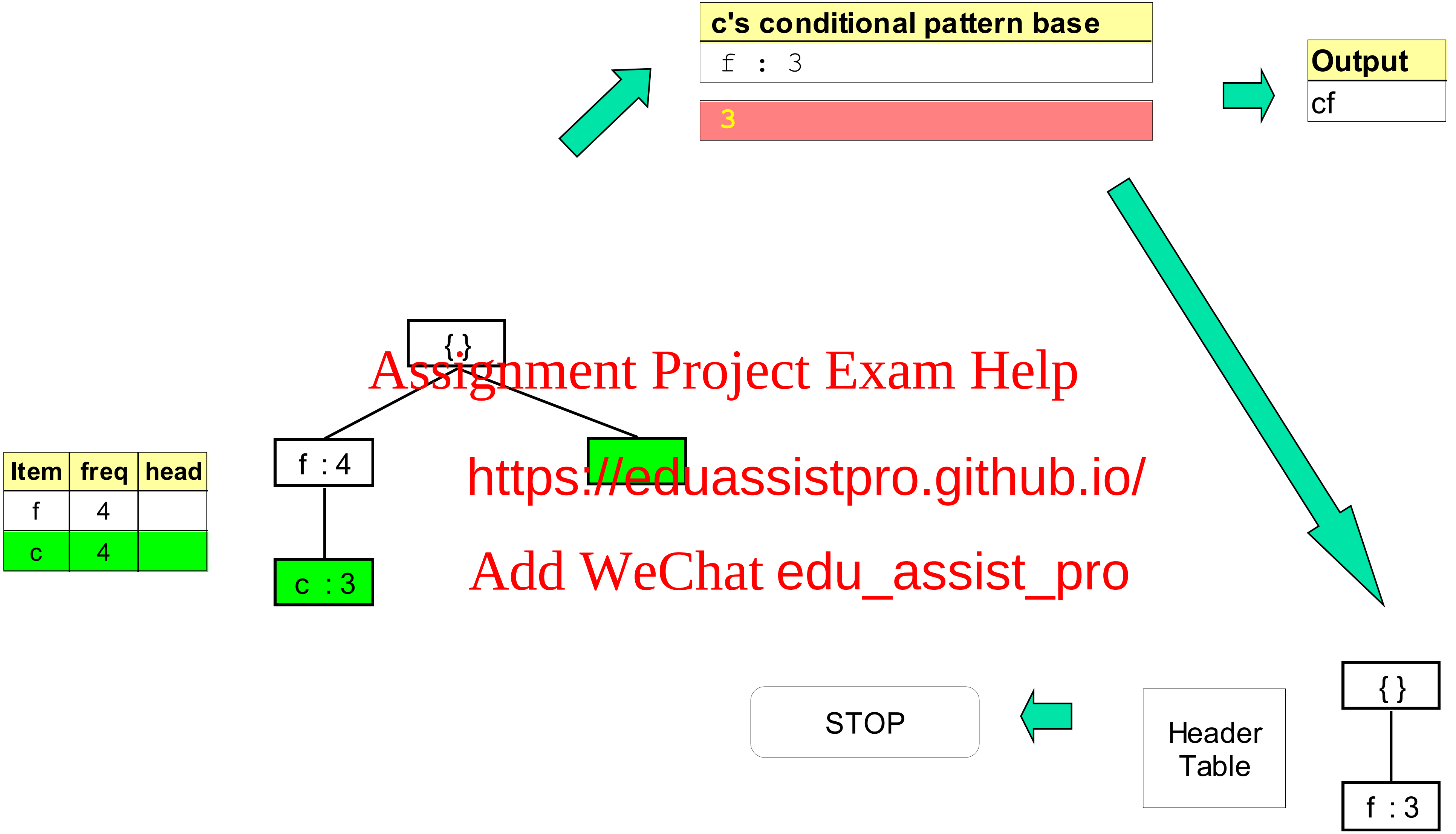
2 2 1

Item	freq	head
f	4	
c	4	
a	3	
b	3	



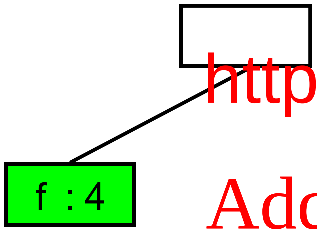
STOP


a's conditional pattern base
f c : 3
3 3
Output
af
ac
{ }
f : 4
c : 1
Item
freq
head
f
4
c
4
a
3
c : 3
a : 3
gen_powerset
Header Table
{ }
f : 3
c : 3
Output
acf

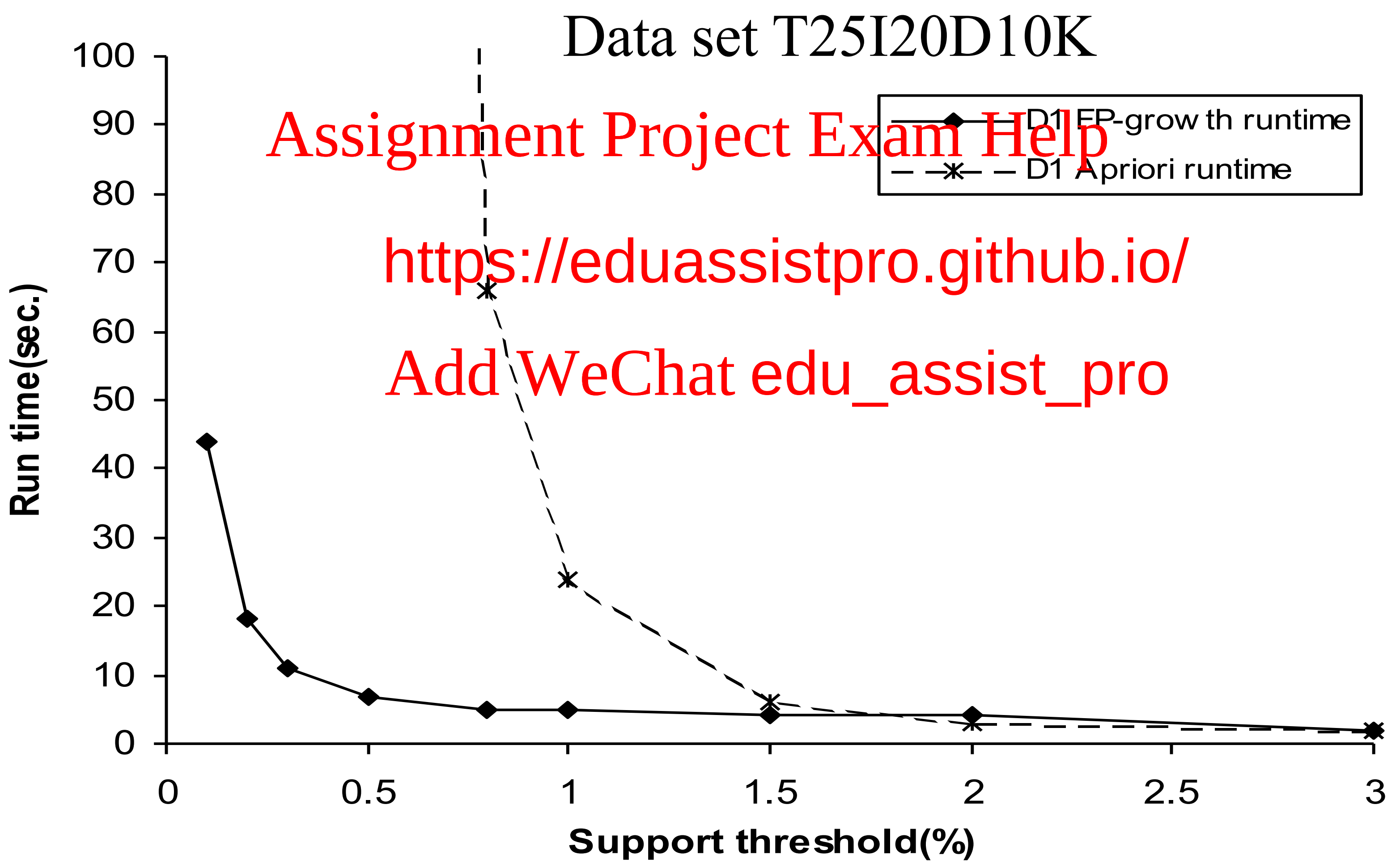

c's conditional pattern base
f : 3
3
Output
cf
{}
f : 4
c : 3
Item
freq
head
f
4
c
4
STOP
Header
Table
{}
f : 3

STOP

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 foc databases

- Other factors

  - no candidate generation, te 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