Mining Association Rules

What are Association Rules?

- Study of "what goes with what"
  - "Customers who bought X also bought Y"
  - What symptoms go with what diagnosis
- Transaction-based or event-based
- Also called "market basket analysis" and "affinity analysis"
- Initially used for Market Basket Analysis to find how items purchased by customers are related

Basket Data

- Retail organizations, e.g., supermarkets, collect and store massive amounts sales data, called basket data.
- A record consist of
  - transaction date
  - items bought
- Or, basket data may consist of items bought by a customer over a period.

Example Association Rule

- 90% of transactions that purchase bread and butter also purchase milk
  - Antecedent: bread and butter
  - Consequent: milk
  - Confidence factor: 90%

Example Queries

- Find all the rules that have “bread” as consequent.
- Find all rules that have “Diet Coke” in the antecedent.
- Find all rules that have “sausage” in the antecedent and “mustard” in the consequent.
- Find all the rules relating items located on shelves A and B in the store.
- Find the “best” (most ***confident***) *k* rules that have “bread” in the consequent.

Applications

- **Market Basket Analysis:** given a database of customer transactions, where each transaction is a set of items the goal is to find groups of items which are frequently purchased together.
- **Telecommunication** (each customer is a transaction containing the set of phone calls)
- **Credit Cards/ Banking Services** (each card/account is a transaction containing the set of customer's payments)
- **Medical Treatments** (each patient is represented as a transaction containing the ordered set of diseases)
- **Recommender system**

The model: data

- $I = \{i_1, i_2, \ldots, i_n\}$: a set of all the items sold in the store
- Transaction t: a set of items such that $t \subseteq I$
  - *TID: unique identifier, associated with each t*
- Transaction Database T: a set of transactions
  - a set of transactions *D* = {t_1, t_2, …, t_n}.

Transaction data: supermarket data

- **Market basket transactions:**

  t1: {bread, cheese, milk}

  t2: {apple, eggs, salt, yogurt}

  … …

  tn: {biscuit, eggs, milk}

Transaction data: a set of documents

- A text document data set. Each document is treated as a “bag” of keywords
  - doc1: Student, Teach, School
  - doc2: Student, School
  - doc3: Teach, School, City, Game
  - doc4: Baseball, Basketball
  - doc5: Basketball, Player, Spectator
  - doc6: Baseball, Coach, Game, Team
  - doc7: Basketball, Team, City, Game

The model: rules

- A transaction *t* contains *X*, a set of items (itemset) in *I*, if $X \subseteq t$.
- An association rule is an implication of the form:
  $X \rightarrow Y$, where $X, Y \subset I$, *and* $X \cap Y = \varnothing$

- An itemset is a set of items.
  - E.g., X = {milk, bread, cereal} is an itemset.
- A *k*-itemset is an itemset with *k* items.
  - E.g., {milk, bread, cereal} is a 3-itemset

Association Rule Definitions

- A set of items is referred as an itemset. A itemset that contains k items is a k-itemset.
- The support s of an itemset X is the percentage of transactions in the transaction database T that contain X.
- The support of the rule $X \Rightarrow Y$ in the transaction database T is the support of the items set $X \Rightarrow Y$ in T.
- The confidence of the rule $X \Rightarrow Y$ in the transaction database T is the ratio of the number of transactions in T that contain $X \Rightarrow Y$ to the number of transactions that contain X in T.

Example

- Given a database of transactions:

Transaction	Items
t_1	Bread,Jelly,PeanutButter
t_2	Bread,PeanutButter
t_3	Bread,Milk,PeanutButter
t_4	Beer,Bread
t_5	Beer,Milk

- Find all the association rules:

$X \Rightarrow Y$	s	α
Bread $\Rightarrow$ PeanutButter	60%	75%
PeanutButter $\Rightarrow$ Bread	60%	100%
Beer $\Rightarrow$ Bread	20%	50%
PeanutButter $\Rightarrow$ Jelly	20%	33.3%
Jelly $\Rightarrow$ PeanutButter	20%	100%
Jelly $\Rightarrow$ Milk	0%	0%

Association Rule Problem

- **Given:**
  - a set I of all the items;
  - a database D of transactions;
  - minimum support s;
  - minimum confidence c;
- **Find:**
  - all association rules $X \Rightarrow Y$ with a minimum support s and confidence c.

Problem Decomposition

- Frequent itemsets: Find all sets of items that have minimum support (user defined threshold)
- Use the frequent itemsets to generate the desired rules

Problem Decomposition

- If the minimum support is 50%, then {Shoes,Jacket} is the only 2-itemset that satisfies the minimum support.
- If the *minimum confidence* is 50%, then the only two rules generated from this 2-itemset, that have confidence greater than 50%, are:

- Shoes ⇒ Jacket Support=50%, Confidence=66%
- Jacket ⇒ Shoes Support=50%, Confidence=100%

Transaction ID	Items Bought
1	Shoes, Shirt, Jacket
2	Shoes,Jacket
3	Shoes, Jeans
4	Shirt, Sweatshirt

Frequent Itemset	Support
{Shoes}	75%
{Shirt}	50%
{Jacket}	50%
{Shoes, Jacket}	50%

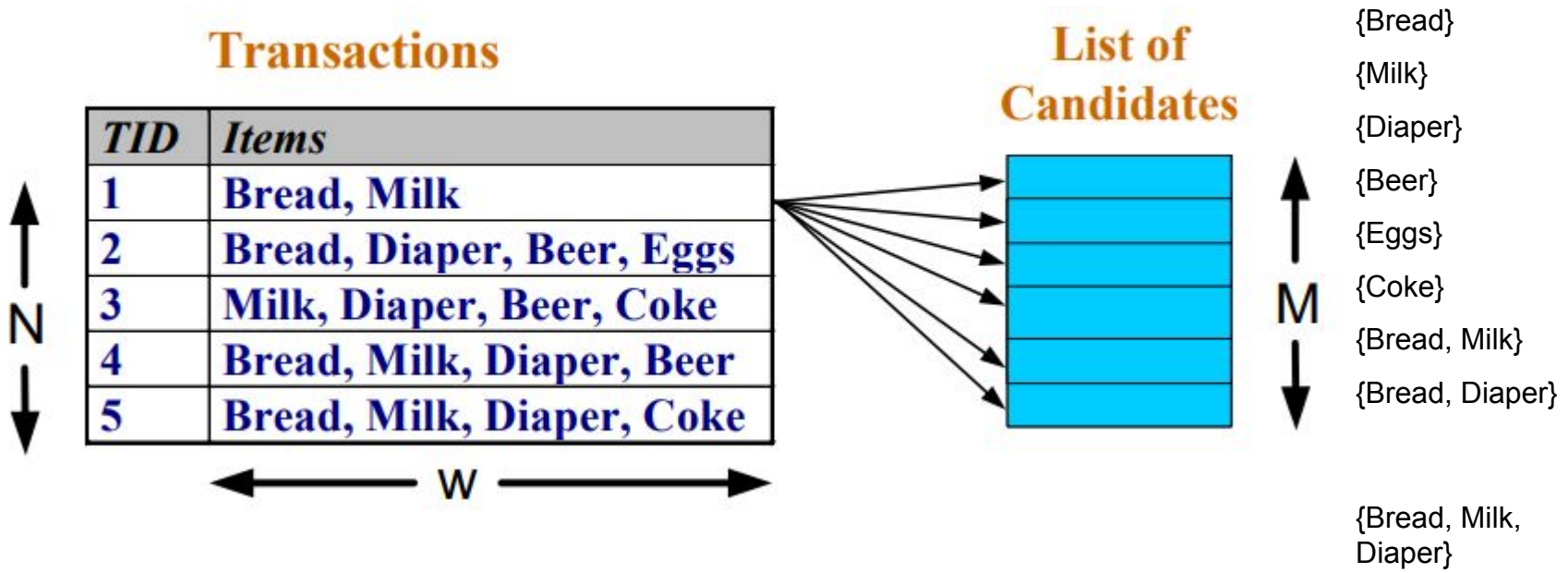
Brute-force algorithm for finding all frequent itemsets?

- Generate all possible itemsets (lattice of itemsets)
  - Start with 1-itemsets, 2-itemsets,...,d-itemsets
- Compute the frequency of each itemset from the data
  - Count in how many transactions each itemset occurs
- If the support of an itemset is above minimum support report it as a frequent itemset

Brute-force algorithm for finding all frequent itemsets?

- Complexity?

- Match every candidate against each transaction
- For M candidates and N transactions, the complexity is~ O(NMw) => Expensive since $M = 2^d$!!!



Speeding-up the brute-force algorithm

- Reduce the number of candidates (M)
  - Complete search: $M = 2^d$
  - Use pruning techniques to reduce M
- Reduce the number of transactions (N)
  - Reduce size of N as the size of itemset increases
  - Use vertical-partitioning of the data to apply the mining algorithms
- Reduce the number of comparisons (NM)
  - Use efficient data structures to store the candidates or transactions
  - No need to match every candidate against every transaction

The Apriori Algorithm

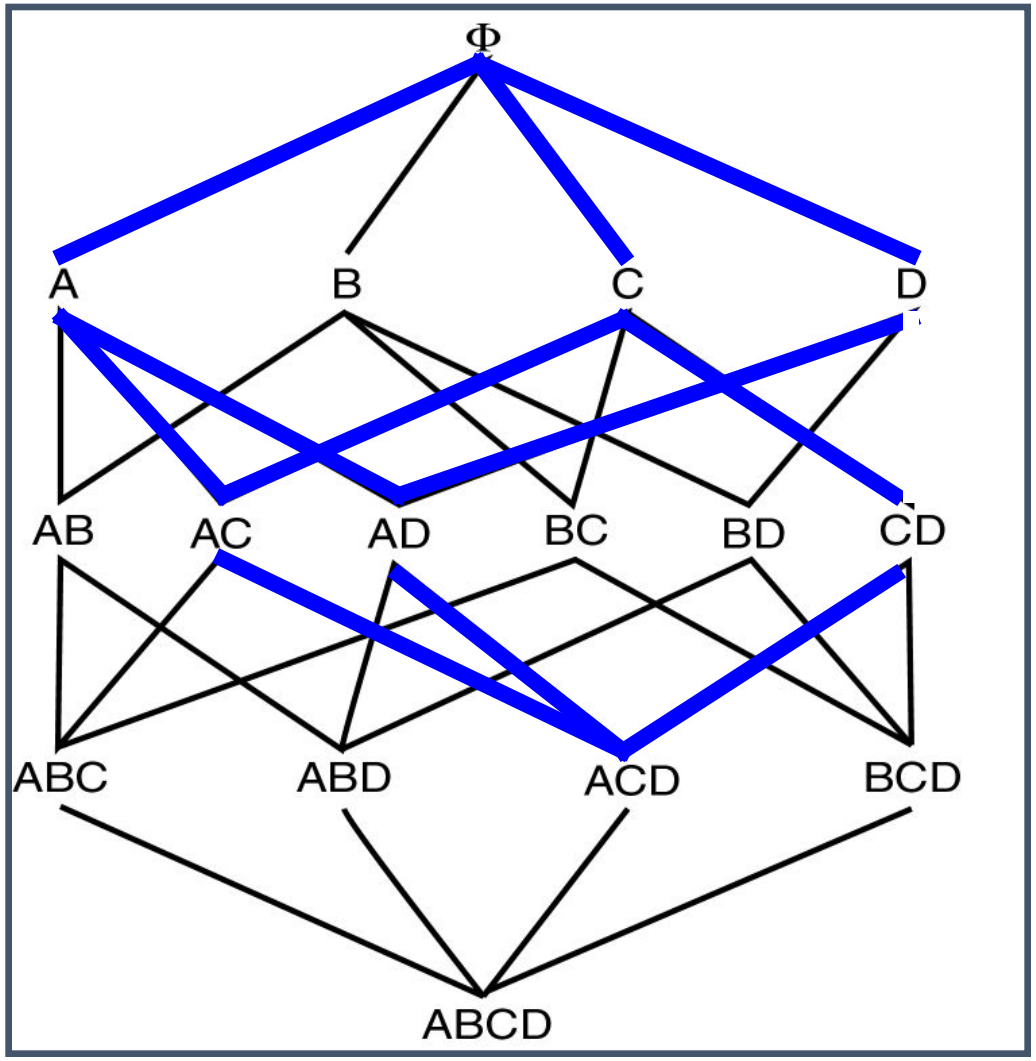
- **Frequent Itemset Property:**
  - **Any subset of a frequent itemset is frequent.**
- **Contrapositive:**
  - **If an itemset is not frequent, none of its supersets are frequent.**

TID	*Items*
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

~~{Bread}~~
{Milk}
{Diaper}
{Beer}
{Eggs}
{Coke}
~~{Bread, Milk}~~
~~{Bread, Diaper}~~

{Bread, Milk, Diaper}

Frequent Itemset Property



The Apriori Algorithm

L_k: Set of frequent itemsets of size k (with minimum support)

C_k: Set of candidate itemset of size k (potentially frequent itemsets)

L_1 = {frequent items};

for (k = 1; L_k !=∅; k++) do

C_{k+1} = candidates generated from L_k;

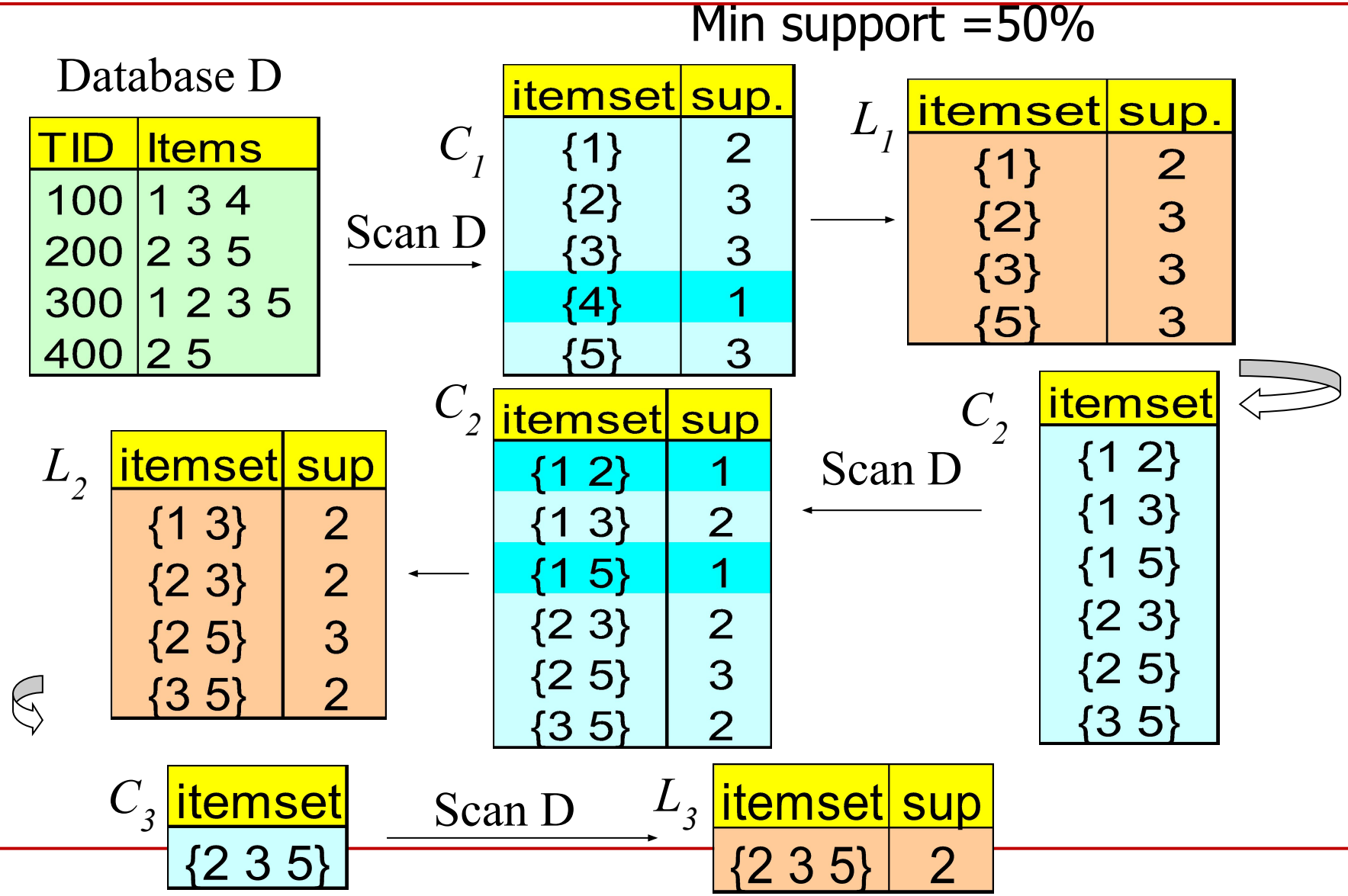
for each transaction t in database do

increment the count of all candidates in C_{k+1} that are contained in t

L_{k+1} = candidates in C_{k+1} with minimum support

return $\cup_k L_k$;

The Apriori Algorithm — Example



How to Generate Candidates

- L3={abc, abd, acd, ace, bcd}
- Generating C4 from L3
  - abcd from abc and abd
  - acde from acd and ace
- Pruning:
  - acde is removed because ade is not in L3
- C4={abcd}

gen_candidate_itemsets *with the given* L_{k-1} as follows:

$C_k = \emptyset$
for all itemsets $l_1 \in L_{k-1}$ ***do***
for all itemsets $l_2 \in L_{k-1}$ ***do***
if $l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \ldots \wedge l_1[k-1] < l_2[k-1]$
then $c = l_1[1], l_1[2] \ldots l_1[k-1], l_2[k-1]$
$C_k = C_k \cup \{c\}$

The Apriori Algorithm

Initialize: $k := 1$, C_1 = all the 1-itemsets;
read the database to count the support of C_1 to determine L_1.
L_1 := {frequent 1-itemsets};
$k := 2$; *// k represents the pass number//*
while $(L_{k-1} \neq \emptyset)$ ***do***
begin
    C_k := ***gen_candidate_itemsets*** *with the given* L_{k-1}

    prune(C_k)
    for all transactions $t \in T$ ***do***
    increment the count of all candidates in C_k that are contained in t;
    L_k := All candidates in C_k with minimum support ;
    $k := k + 1$;
end
Answer := $\cup_k L_k$;

prune(C_k)

for all $c \in C_k$
for all $(k-1)$-subsets d of c ***do***
    if $d \notin L_{k-1}$
    then $C_k = C_k \setminus \{c\}$

Example of Discovering Rules

- Let us consider the 3-itemset {I1, I2, I5}:
- Possible Rules
  - I1 $\wedge$ I2 $\Rightarrow$ I5
  - I1 $\wedge$ I5 $\Rightarrow$ I2
  - I2 $\wedge$ I5 $\Rightarrow$ I1
  - I1 $\Rightarrow$ I2 $\wedge$ I5
  - I2 $\Rightarrow$ I1 $\wedge$ I5
  - I5 $\Rightarrow$ I1 $\wedge$ I2

Discovering Rules

for each frequent itemset I do
  for each subset C of I do [$(I - C) \Rightarrow C$]
    if (support(I) / support($I - C$) >= minimum confidence) then
      output the rule $(I - C) \Rightarrow C$,
      with confidence = support(I) / support ($I - C$)
        and support = support(I)

Example of Discovering Rules

- Let use consider the 3-itemset {I1, I2, I5} with support of 0.22(2)%. Let generate all the association rules from this itemset:

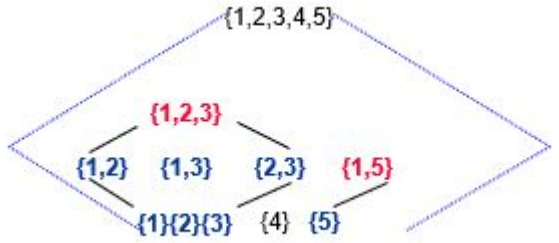
- I1 $\wedge$ I2 $\Rightarrow$ I5 *confidence*= 2/4 = 50%
- I1 $\wedge$ I5 $\Rightarrow$ I2 *confidence*= 2/2 = 100%
- I2 $\wedge$ I5 $\Rightarrow$ I1 *confidence*= 2/2 = 100%
- I1 $\Rightarrow$ I2 $\wedge$ I5 *confidence*= 2/6 = 33%
- I2 $\Rightarrow$ I1 $\wedge$ I5 *confidence*= 2/7 = 29%
- I5 $\Rightarrow$ I1 $\wedge$ I2 *confidence*= 2/2 = 100%

TID	List of Item_IDs
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

Maximal Frequent set, Boarder set

- **Maximal Frequent set**
  - A frequent set is maximal frequent set if it is a frequent set and no superset of this is a frequent set
  - The set of all maximal frequent set can act as a compact representation of the all frequent sets
- **Boarder set**
  - An itemset is a boarder set if it is not a frequent set, but all its proper subsets are frequent sets
- **Question: Maximal frequent set?**

tid	itemset
1	{1,2,3,5}
2	{1,5}
3	{1,2}
4	{1,2,3}



Challenges of Frequent Pattern Mining

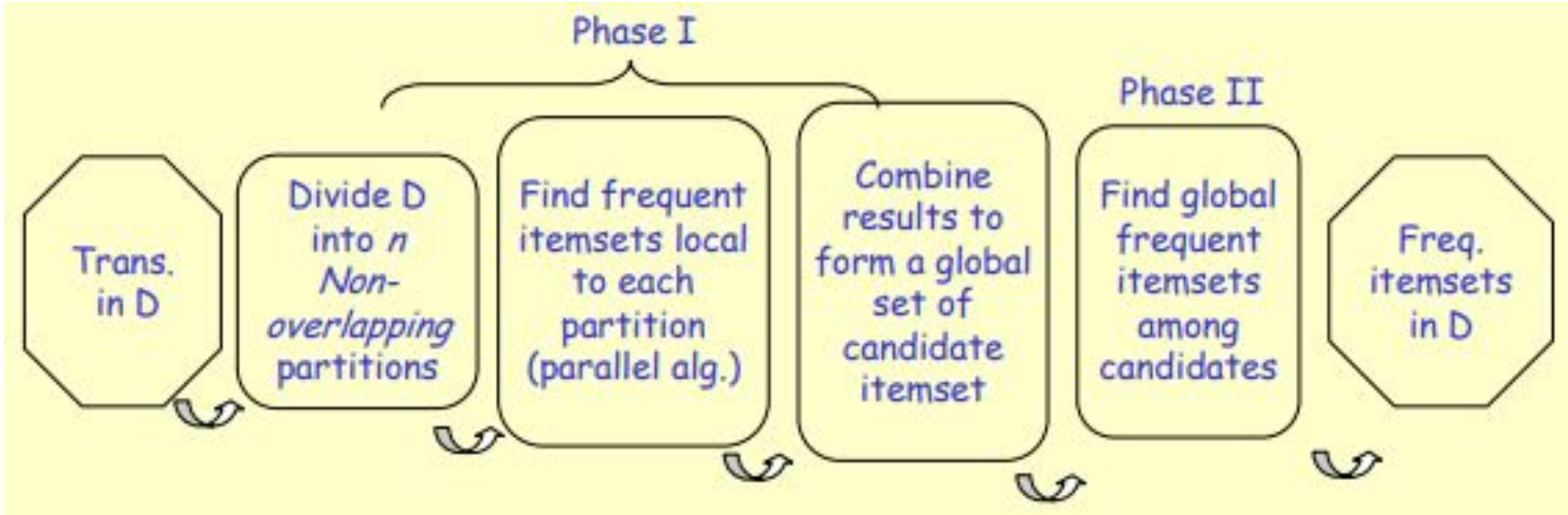
- Challenges
  - Multiple scans of transaction database
  - Huge number of candidates
  - Tedious workload of support counting for candidates
- Improving Apriori: general ideas
  - Reduce passes of transaction database scans
  - Shrink number of candidates
  - Facilitate support counting of candidates

Partition Algorithm

- The partition algorithm is based on the observation that the frequent sets are normally very few in number compared to the set of all itemsets
- As a result, if we partition the set of transactions to smaller segments such that each segment can be accommodated in the main memory, then we can compute the set of frequent sets of each of these partition
- Since each partition can fit in the main memory, there will be no additional disk I/O for each partition after loading the partition into the main memory

Partition: Scan Database Only Twice

- *Any itemset that is potentially frequent in DB must be frequent in at least one of the partitions of DB*
  - **Scan 1: partition database and find local frequent patterns**
  - **Scan 2: consolidate global frequent patterns**



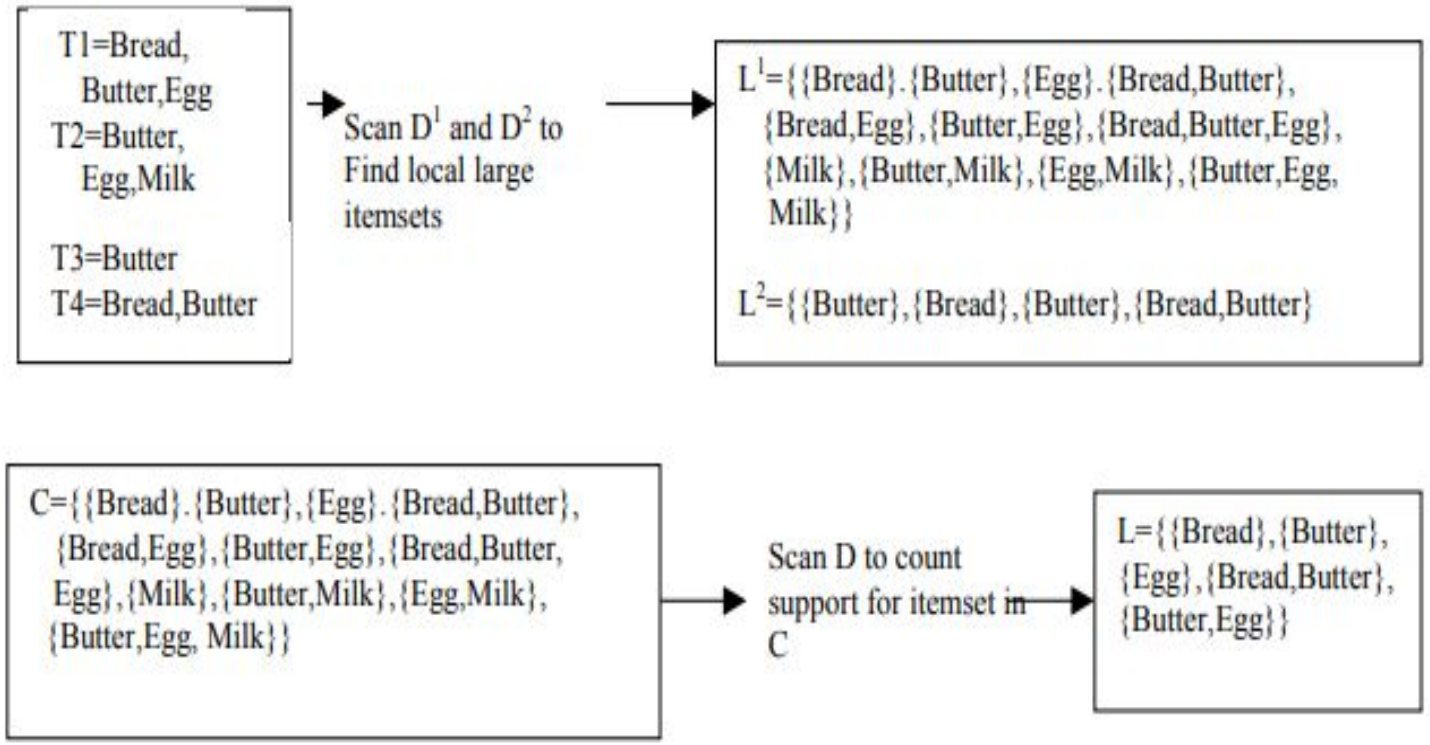
Partition Algorithm

Algorithm Partition:

1) P = partition_database(D)
2) n = Number of partitions
3) **for** i=1 to n **begin** // Phase I
4) read_in_partition($p_i \in P$)
5) $L^i = gen\_large\_itemsets(p_i)$
6) **end**
7) **for** (i=2; $L_i^j \neq \emptyset$, j=1, 2, ..., n; i++) **do**
8) $C_i^G = \cup_{j=1,2,...,n} L_i^j$ // Merge Phase
9) **for** i=1 to n **begin** // Phase II
10) read_in_partition($p_i \in P$)
11) **for all** candidates $c \in C^G$ gen_count(c, p_i)
12) **end**
13) $L^G = \{c \in C^G \mid c.\text{count} \geq \text{min\_sup}\}$

The local large itemsets, L^i , can be found by using a level-wise algorithm such as Apriori.

Partition Algorithm - Example



TID	Items
t_1	Bread, Butter, Eggs
t_2	Butter, Eggs, Milk
t_3	Butter
t_4	Bread, Butter

Sampling for Frequent Patterns

- A sample which can fit in the main memory is first drawn from the database.
- The set of large itemsets in the sample is then found from this sample by using a level-wise algorithm such as Apriori.
- Let the set of large itemsets in the sample be PL, which is used as a set of probable large itemsets and used to generate candidates which are to be verified against the whole database .

Sampling for Frequent Patterns

- The candidates are generated by applying the negative border function, BD^-, to PL.
- Thus the candidates are $BD^-(PL) \cup PL$.
- The negative border of a set of itemsets PL is the minimal set of itemsets which are not in PL, but all their subsets are
- After the candidates are generated, the whole database is scanned once to determine the counts of the candidates.
- If all large itemsets are in PL, i.e., no itemsets in $BD^-(PL)$ turn out to be large, then all large itemsets are found and the algorithm terminates
- Otherwise, repeat the steps until $BD^-(PL) \cup PL$ does not grow

Sampling for Frequent Patterns

- This can guarantee that all large itemsets are found, because $BD^{-}(PL) \cup PL$ actually contains all candidate itemsets of Apriori if PL contains all large itemsets L, i.e., L⊆PL.
- Example
  - Let, PL={{A}, {B}, {C}, {A,B}}
  - The candidate itemsets for the first scan are $BD^{-}(PL) \cup PL$

    = {{A, C}, {B,C}} ∪ {{A}, {B}, {C}, {A,B}}

    = {{A}, {B}, {C}, {A,B}, {A,C}, {B,C}}

Pruning

- Pruning can reduce the size of the candidate set C_k. We want to transform C_k into a set of frequent items L_k. To reduce the work of checking, we may use the rule that all subsets of C_k must also be frequent.

- Example
  - Suppose the items are A, B, C, D, E, F, .., X, Y, Z
  - Suppose L_1 is A, C, E, P, Q, S, T, V, W, X
  - Suppose L_2 is {A, C}, {A, F}, {A, P}, {C, P}, {E, P}, {E, G}, {E, V}, {H, J}, {K, M}, {Q, S}, {Q, X}
  - Are you able to identify errors in the L_2 list?
  - What is C_3?
  - How to prune C_3?
  - C_3 is {A, C, P}, {E, P, V}, {Q, S, X}

Direct Hashing and Pruning

- The direct hashing and pruning (DHP) algorithm attempts to generate large itemsets efficiently and reduces the transaction database size.
- When generating L_1, the algorithm also generates all the 2-itemsets for each transaction, hashes them to a hash table and keeps a count.

Example

- **Consider the transaction database in the first table below. The second table below shows all possible 2-itemsets for each transaction.**

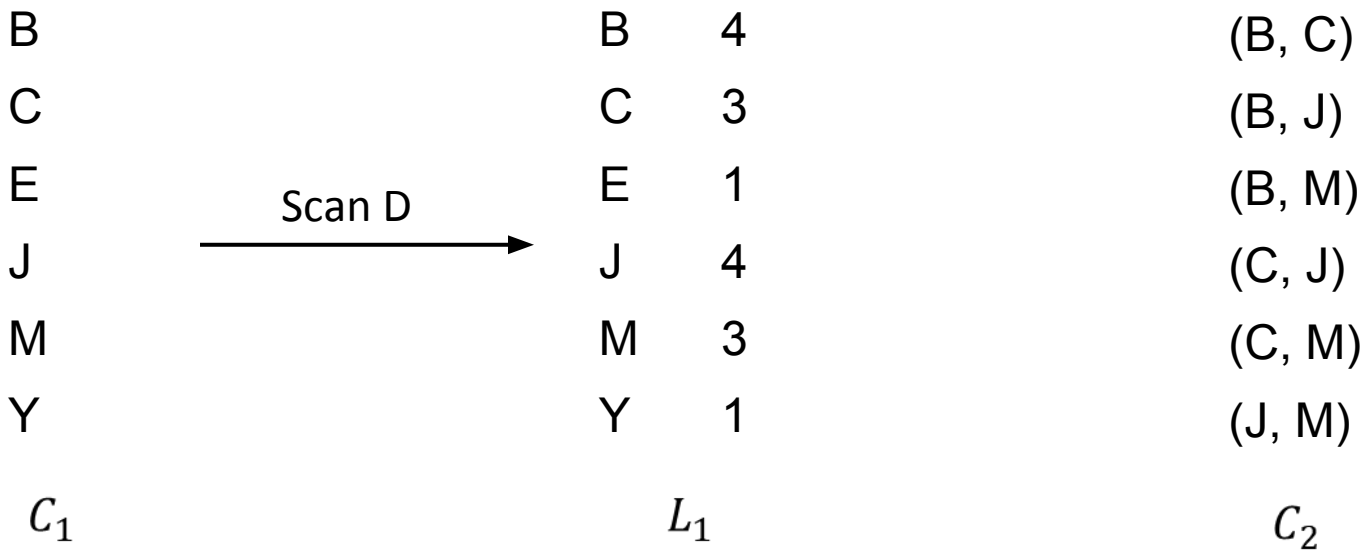
Transaction ID	Items
100	Bread, Cheese, Eggs, Juice
200	Bread, Cheese, Juice
300	Bread, Milk, Yogurt
400	Bread, Juice, Milk
500	Cheese, Juice, Milk

Support: 3

100	(B, C) (B, E) (B, J) (C, E) (C, J) (E, J)
200	(B, C) (B, J) (C, J)
300	(B, M) (B, Y) (M, Y)
400	(B, J) (B, M) (J, M)
500	(C, J) (C, M) (J, M)

Example

Transaction ID	Items
100	Bread, Cheese, Eggs, Juice
200	Bread, Cheese, Juice
300	Bread, Milk, Yogurt
400	Bread, Juice, Milk
500	Cheese, Juice, Milk



Example

Transaction ID	Items
100	Bread, Cheese, Eggs, Juice
200	Bread, Cheese, Juice
300	Bread, Milk, Yogurt
400	Bread, Juice, Milk
500	Cheese, Juice, Milk

100	(B, C) (B, E) (B, J) (C, E) (C, J) (E, J)
200	(B, C) (B, J) (C, J)
300	(B, M) (B, Y) (M, Y)
400	(B, J) (B, M) (J, M)
500	(C, J) (C, M) (J, M)

Bit vector	Bucket number	Count	Pairs	C_2
1	0	5	(C, J) (B, Y) (M, Y)	(C, J)
0	1	1	(C, M)	
0	2	1	(E, J)	
0	3	0		
0	4	2	(B, C)	
1	5	3	(B, E) (J, M)	(J, M)
1	6	3	(B, J)	(B, J)
1	7	3	(C, E) (B, M)	(B, M)

B		B 4	(B, C)
C		C 3	(B, J)
E	Scan D →	~~E 1~~	(B, M)
J		J 4	(C, J)
M		M 3	(C, M)
Y		~~Y 1~~	(J, M)
C_1		L_1	C_2

B	1
C	2
E	3
J	4
M	5
Y	6

Hash Function Used

B	1
C	2
E	3
J	4
M	5
Y	6

- For each pair, a numeric value is obtained by first representing B by 1, C by 2, E 3, J 4, M 5 and Y 6. Now each pair can be represented by a two digit number, for example (B, E) by 13 and (C, M) by 25.
- The two digits are then coded as modulo 8 (let's say) number (dividing by 8 and using the remainder). This is the bucket address.
- A count of the number of pairs hashed is kept. Those addresses that have a count above the support value have the bit vector set to 1 otherwise 0.
- All pairs in rows that have zero bit are removed.

Find C_2

- The major aim of the algorithm is to reduce the size of C_2.
- It is therefore essential that the hash table is large enough so that collisions are low. Collisions result in loss of effectiveness of the hash table.
- This is what happened in the example in which we had collisions in three of the eight rows of the hash table which required us finding which pair was frequent.

Transaction Reduction

Bit vector	Bucket number	Count	Pairs	C_2
1	0	5	(C, J) (B, Y) (M, Y)	(C, J)
0	1	1	(C, M)	
0	2	1	(E, J)	
0	3	0		
0	4	2	(B, C)	
1	5	3	(B, E) (J, M)	(J, M)
1	6	3	(B, J)	(B, J)
1	7	3	(C, E) (B, M)	(B, M)

Bit vector	Bucket number	Count	Pairs	C_2
1	0	5	(C, J) (B, Y) (M, Y)	(C, J)
0	1	1	(C, M)	
0	2	1	(E, J)	
0	3	0		
0	4	2	(B, C)	
1	5	3	(B, E) (J, M)	(J, M)
1	6	3	(B, J)	(B, J)
1	7	3	(C, E) (B, M)	(B, M)

Transaction ID	Items
100	Bread, Cheese, Eggs, Juice
200	Bread, Cheese, Juice
300	Bread, Milk, Yogurt
400	Bread, Juice, Milk
500	Cheese, Juice, Milk

(B,J), (C,J)

(C,J), (J,M)

Transaction Reduction

- Database itself is pruned by removing transactions based on the logic that a transaction can contain a frequent (*k+1*)-itemset only if contains at least *k+1* different frequent *k*-itemsets. So, a transaction that doesn't contain *k+1* frequent *k*-itemsets can be pruned.
  - E.g., say a transaction is {*a*, *b*, *c*, *d*, *e*, *f* }. Now, if it contains a frequent 3-itemset, say *aef*, then it contains the 3 frequent 2-itemsets *ae*, *af*, *ef*.
  - So, at the time that L_k, the frequent *k*-itemsets are determined, one can check transactions according to the condition above for possible pruning before the next stage.
  - Say, we have determined L_2 = {*ac*, *bd*, *eg*, *fg* }. Then, we can drop the transaction {*a*, *b*, *c*, *d*, *e*, *f* } from the database for the next step. Why?

Dynamic Itemset Counting

- Alternative to Apriori Itemset Generation
- Itemsets are dynamically added and deleted as transactions are read
- Relies on the fact that for an itemset to be frequent, all of its subsets must also be frequent, so we only examine those itemsets whose subsets are all frequent

Dynamic Itemset Counting

- Algorithm stops after every *M* transactions to add more itemsets.
  - **Train analogy:**
    - DIC works like a train running over the data with stops at interval M between transactions
    - When the train reaches the end of transaction file, it has made one pass over the data , and it starts again from the beginning for the next pass
    - The passengers are itemsets. Itemsets can get on at any stop as long as they get off at the same stop in the next pass around the database.
    - Only itemsets on the train are counted when they occur in transactions.

Dynamic Itemset Counting

- At the very beginning we can start counting 1-itemsets, at the first station we can start counting some of the 2-itemsets. At the second station we can start counting 3-itemsets as well as any more 2-itemsets that can be counted and so on.

Dynamic Itemset Counting

- **Itemsets are marked in four different ways as they are counted:**
  - **Solid box:**
    - confirmed frequent itemset - an itemset we have finished counting and exceeds the support threshold *minsupp*
  - **Solid circle:**
    - confirmed infrequent itemset - we have finished counting and it is below *minsupp*
  - **Dashed box:**
    - suspected frequent itemset - an itemset we are still counting that exceeds *minsupp*
  - **Dashed circle:**
    - suspected infrequent itemset - an itemset we are still counting that is below *minsupp*

Dynamic Itemset Counting

- Algorithm: Mark the empty itemset with a solid square. Mark all the 1-itemsets with dashed circles. Leave all other itemsets unmarked.
- While any dashed itemsets remain:
  - Read *M* transactions (if we reach the end of the transaction file, continue from the beginning). For each transaction, increment the respective counters for the itemsets that appear in the transaction and are marked with dashes.
  - If a dashed circle's count exceeds *minsupp*, turn it into a dashed square. If any immediate superset of it has all of its subsets as solid or dashed squares, add a new counter for it and make it a dashed circle.
  - Once a dashed itemset has been counted through all the transactions, make it solid and stop counting it.

Dynamic Itemset Counting - Example

min_sup= 2 (=20%) , M = 5

TID	Items
1	a b d e
2	b c d
3	a b d e
4	a c d e
5	b c d e
6	b d e
7	c d
8	a b c
9	a d e
10	b d

TID	a	b	c	d	e
1	1	1	0	1	1
2	0	1	1	1	0
3	1	1	0	1	1
4	1	0	1	1	1
5	0	1	1	1	1
6	0	1	0	1	1
7	0	0	1	1	0
8	1	1	1	0	0
9	1	0	0	1	1
10	0	1	0	1	0

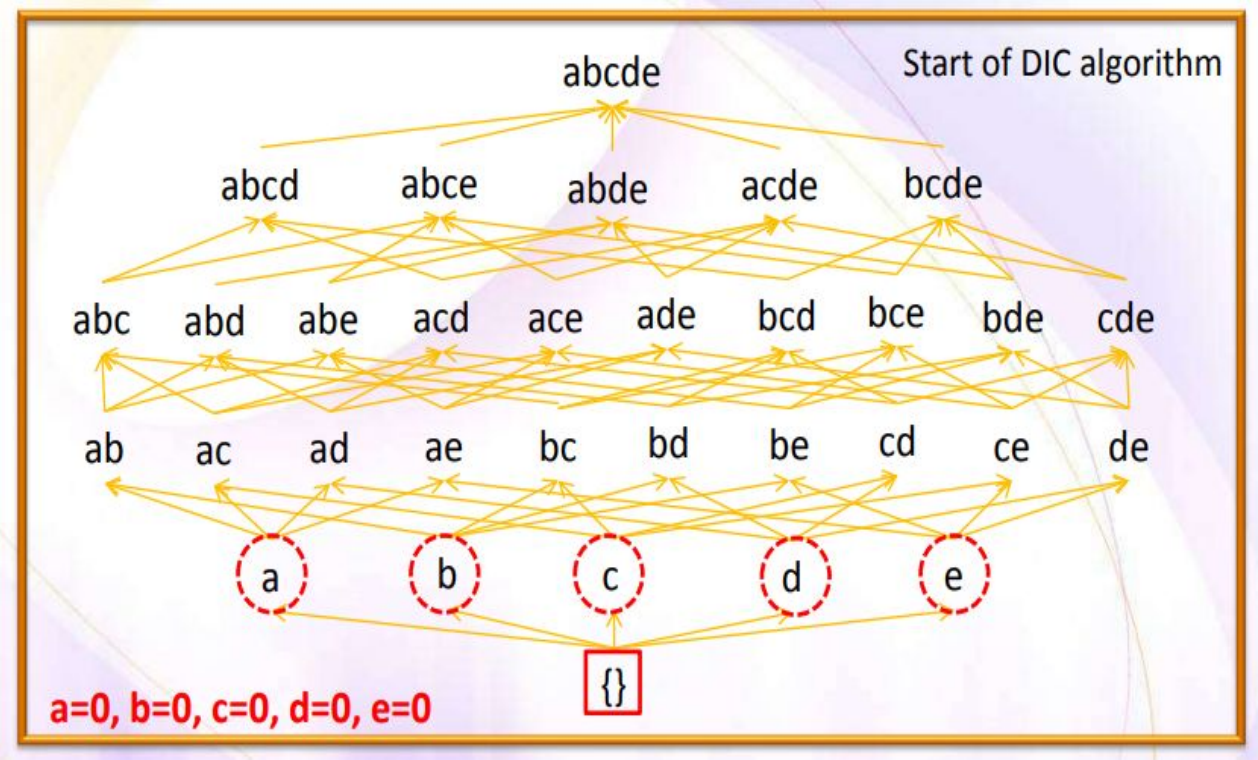
Dynamic Itemset Counting - Example

- Mark the empty itemset with a solid square.
- Mark all the 1-itemsets with dashed circles.
- Leave all other itemsets unmarked.

min_sup= 2 (=20%) , M = 5

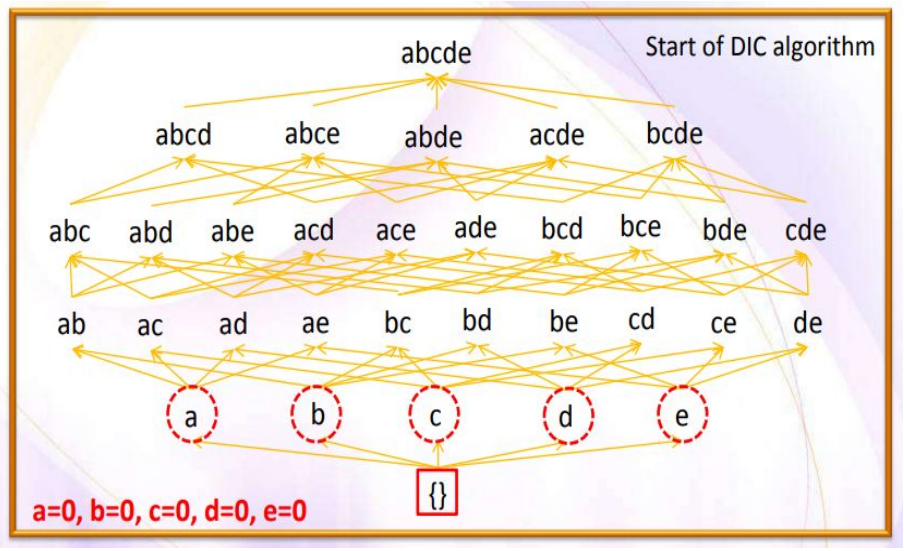
TID	Items
1	a b d e
2	b c d
3	a b d e
4	a c d e
5	b c d e
6	b d e
7	c d
8	a b c
9	a d e
10	b d

TID	a	b	c	d	e
1	1	1	0	1	1
2	0	1	1	1	0
3	1	1	0	1	1
4	1	0	1	1	1
5	0	1	1	1	1
6	0	1	0	1	1
7	0	0	1	1	0
8	1	1	1	0	0
9	1	0	0	1	1
10	0	1	0	1	0



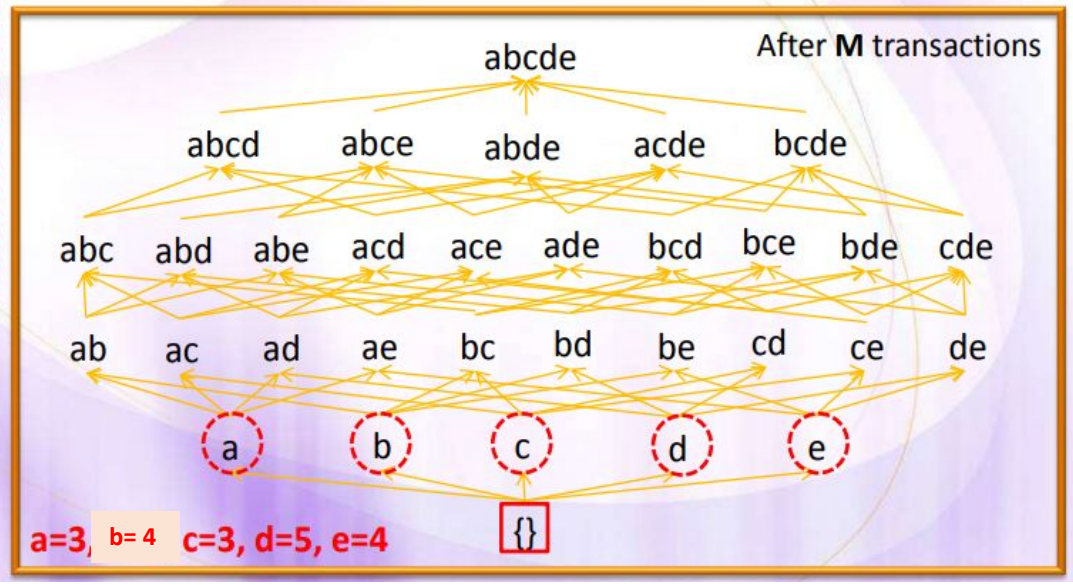
Dynamic Itemset Counting - Example

- **While any dashed itemsets remain:**
  - **Read M transactions. For each transaction, increment the respective counters for the itemsets that appear in the transaction and are marked with dashes.**



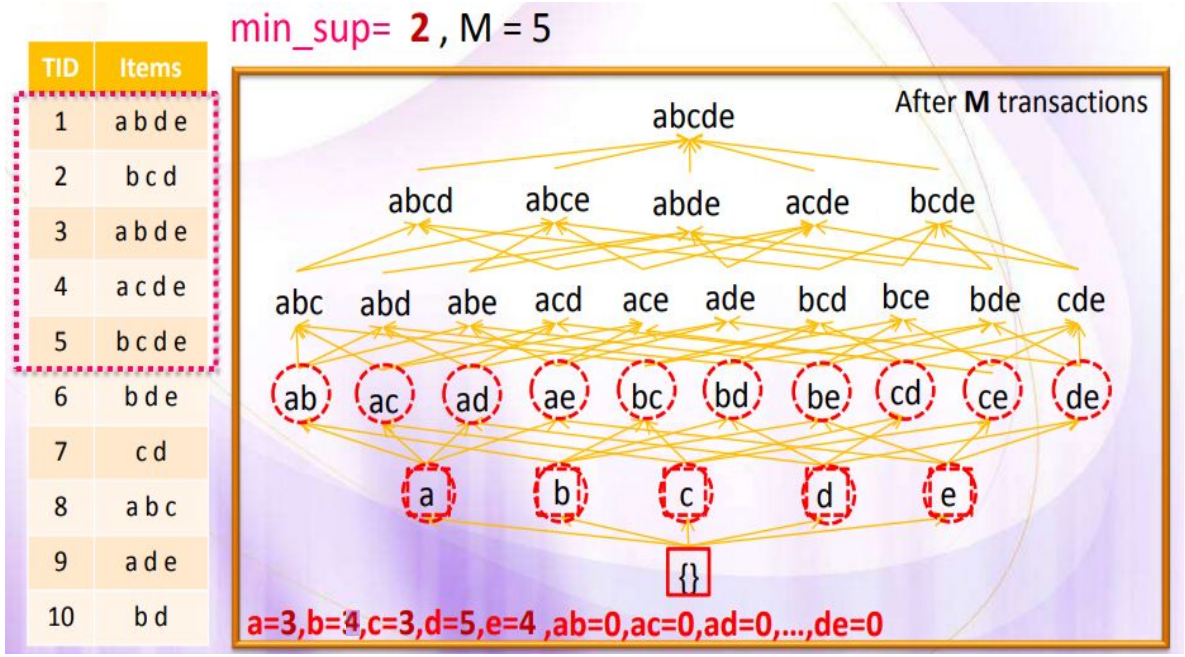
TID	Items
1	a b d e
2	b c d
3	a b d e
4	a c d e
5	b c d e
6	b d e
7	c d
8	a b c
9	a d e
10	b d

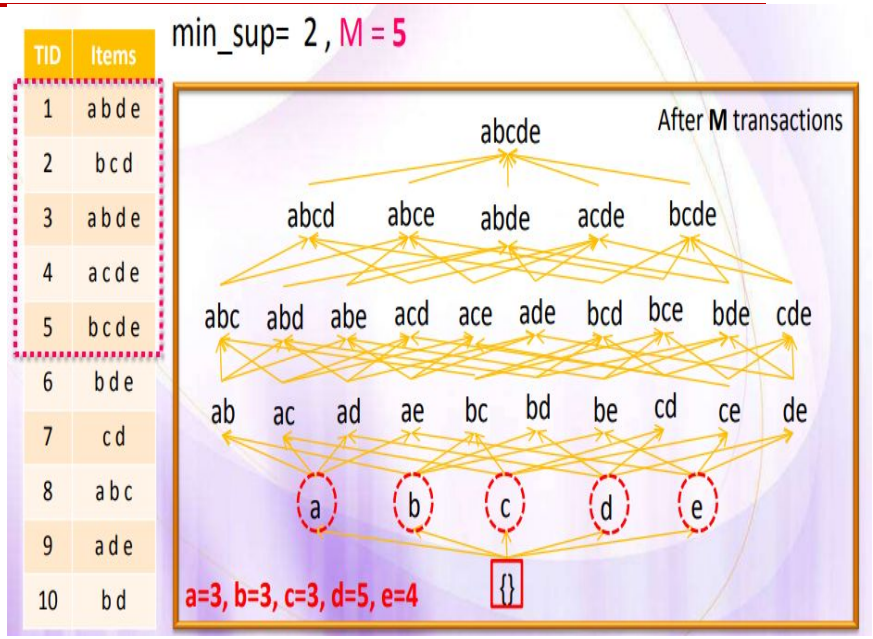
min_sup= 2 , M = 5



Dynamic Itemset Counting - Example

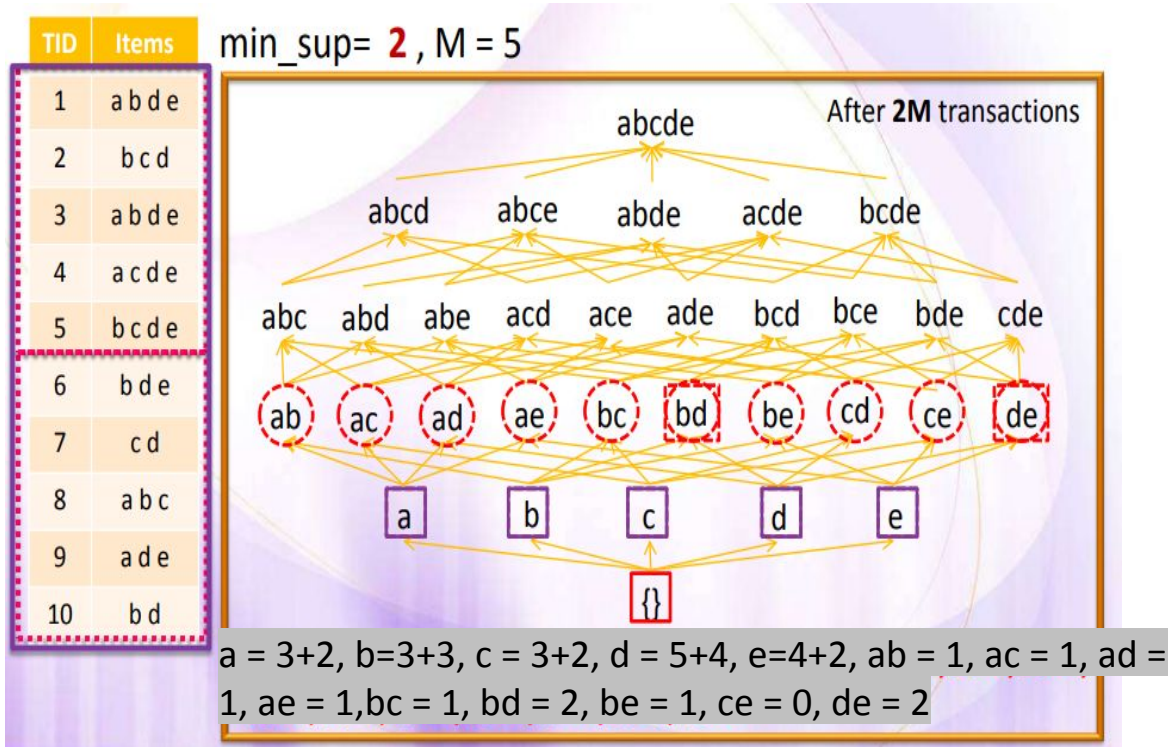
- If a dashed circle's count exceeds minsupp, turn it into a dashed square.
- If any immediate superset of it has all of its subsets as solid or dashed squares, add a new counter for it and make it a dashed circle.

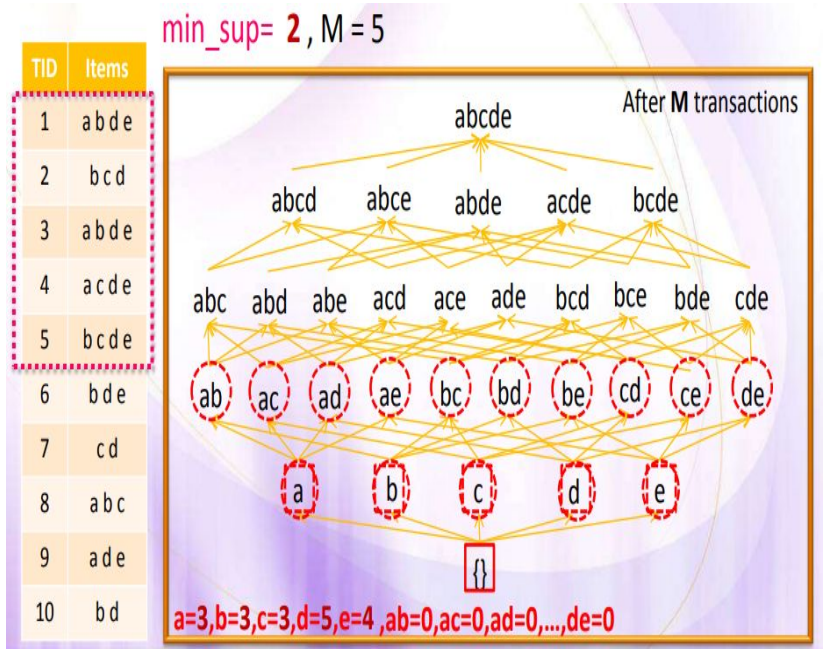




Dynamic Itemset Counting - Example

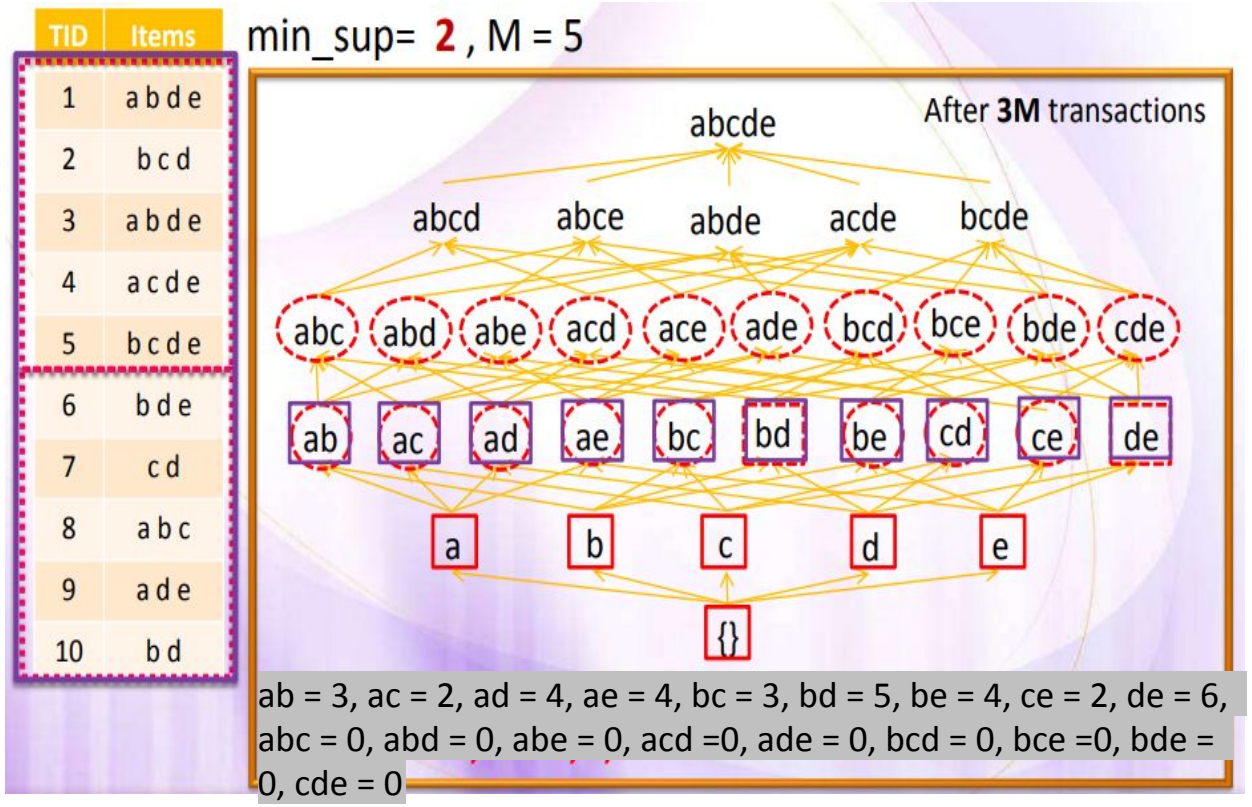
- **If a dashed itemset has been counted through all the transactions, make it solid and stop counting it.**

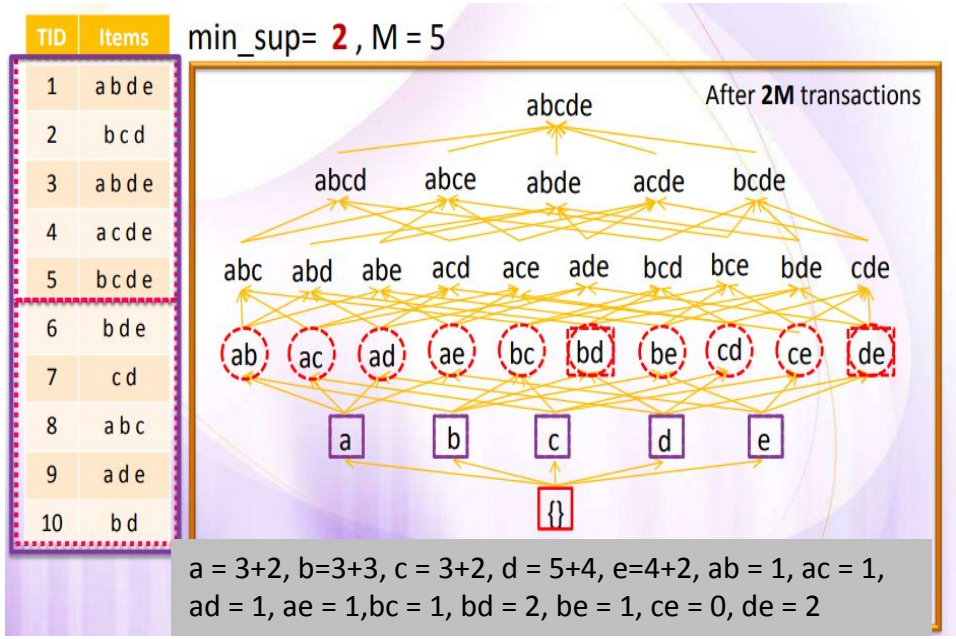




Dynamic Itemset Counting - Example

- If we are at the end of the transaction file, rewind to the beginning
- If any dashed itemsets remain, go to step 1

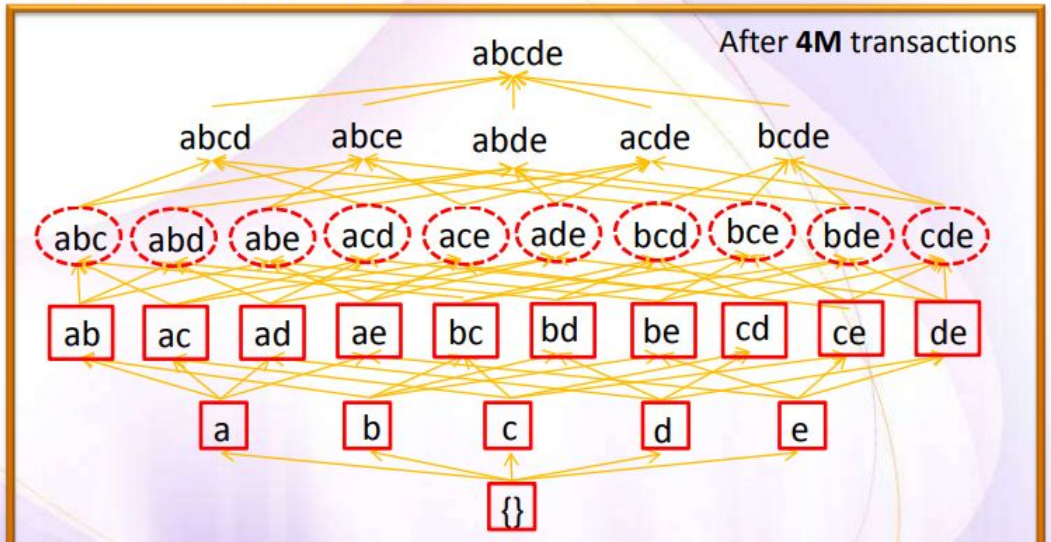




Dynamic Itemset Counting - Example

TID	Items
1	a b d e
2	b c d
3	a b d e
4	a c d e
5	b c d e
6	b d e
7	c d
8	a b c
9	a d e
10	b d

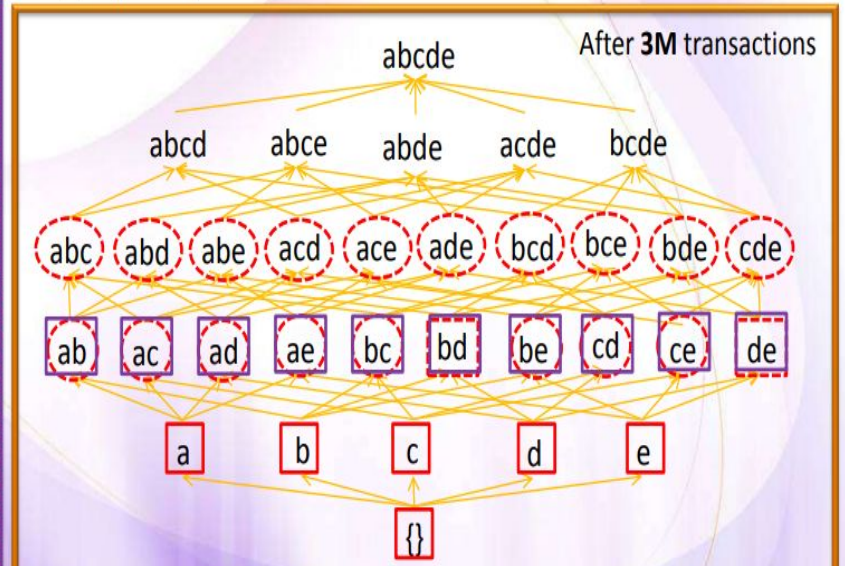
min_sup= **2** , M = 5



abc = 1, abd = 0, abe =0, acd =0, ade = 1, bcd = 0, bce =0, bde = 1, cde = 0

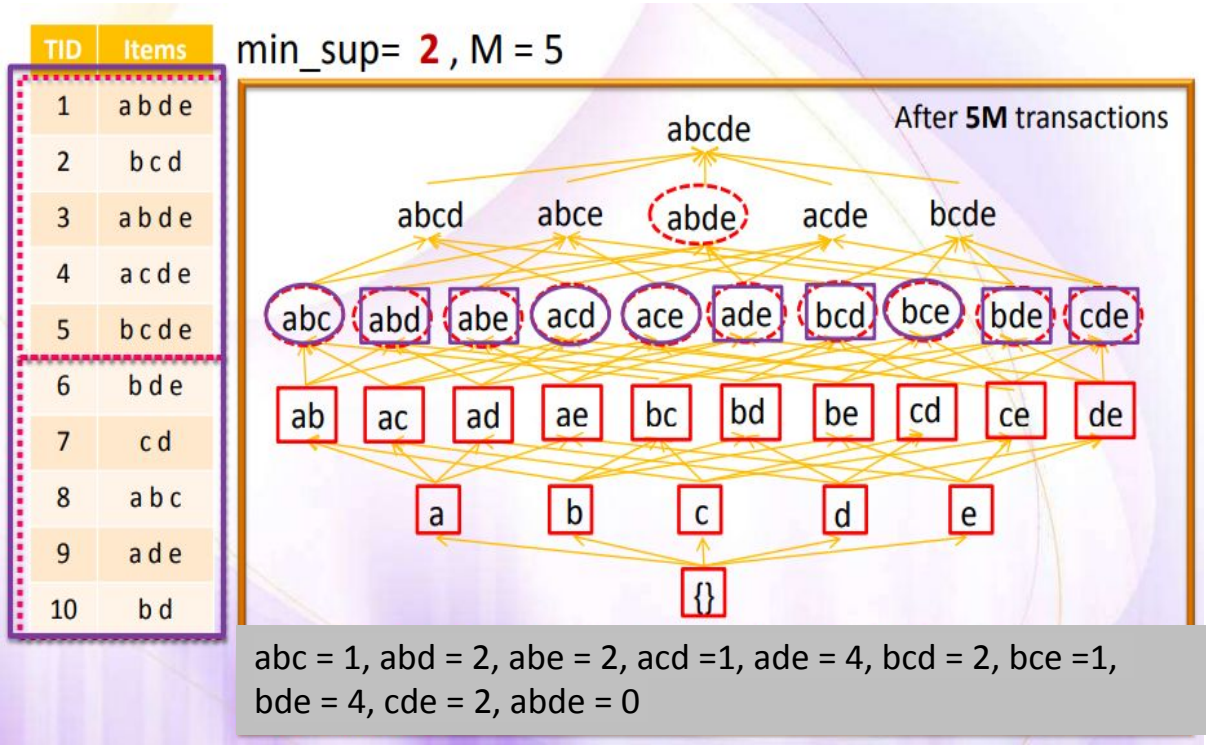
TID	Items
1	a b d e
2	b c d
3	a b d e
4	a c d e
5	b c d e
6	b d e
7	c d
8	a b c
9	a d e
10	b d

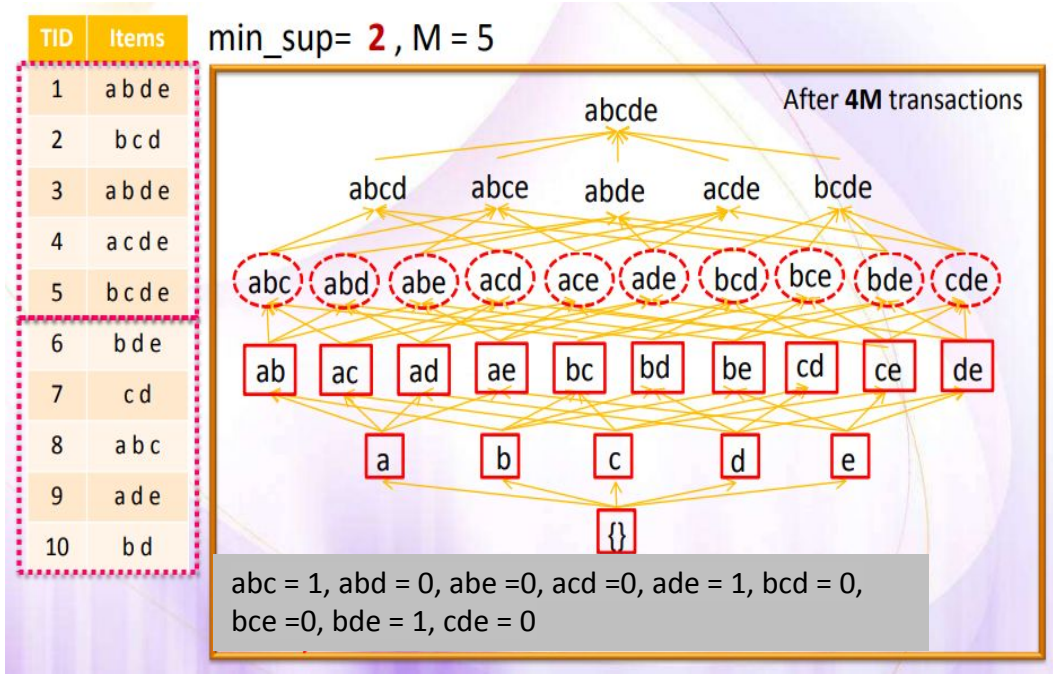
min_sup= **2** , M = 5



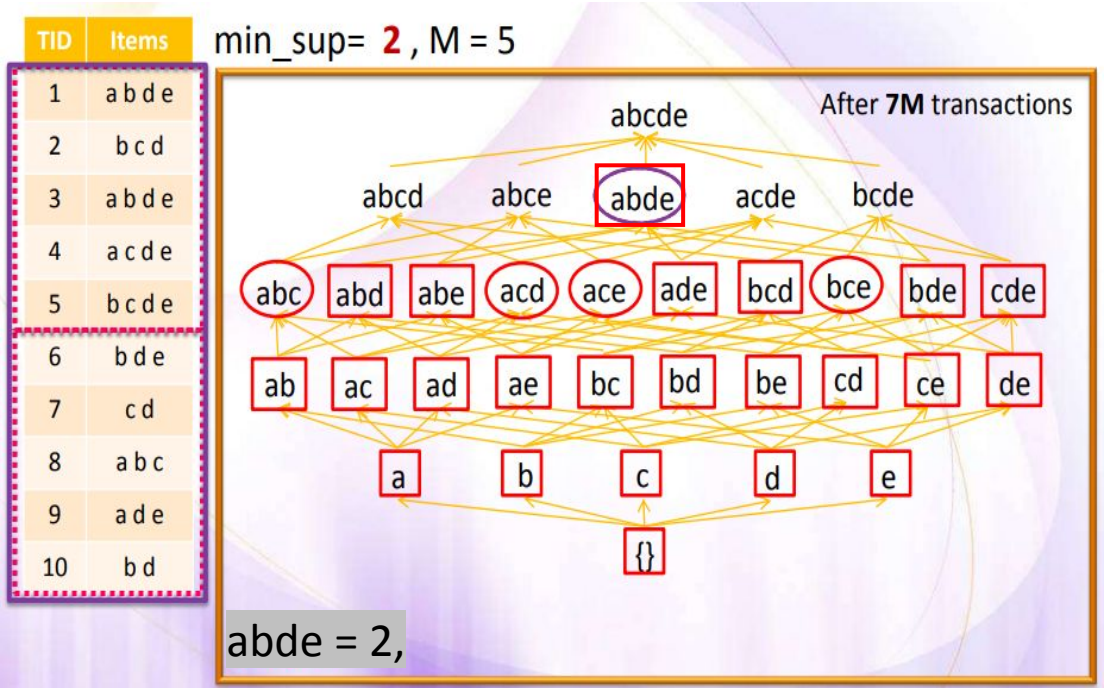
ab = 3, ac = 2, ad = 4, ae = 4, bc = 3, bd = 5, be = 4, ce = 2, de = 6, abc = 0, abd = 0, abe = 0, acd =0, ade = 0, bcd = 0, bce =0, bde = 0, cde = 0

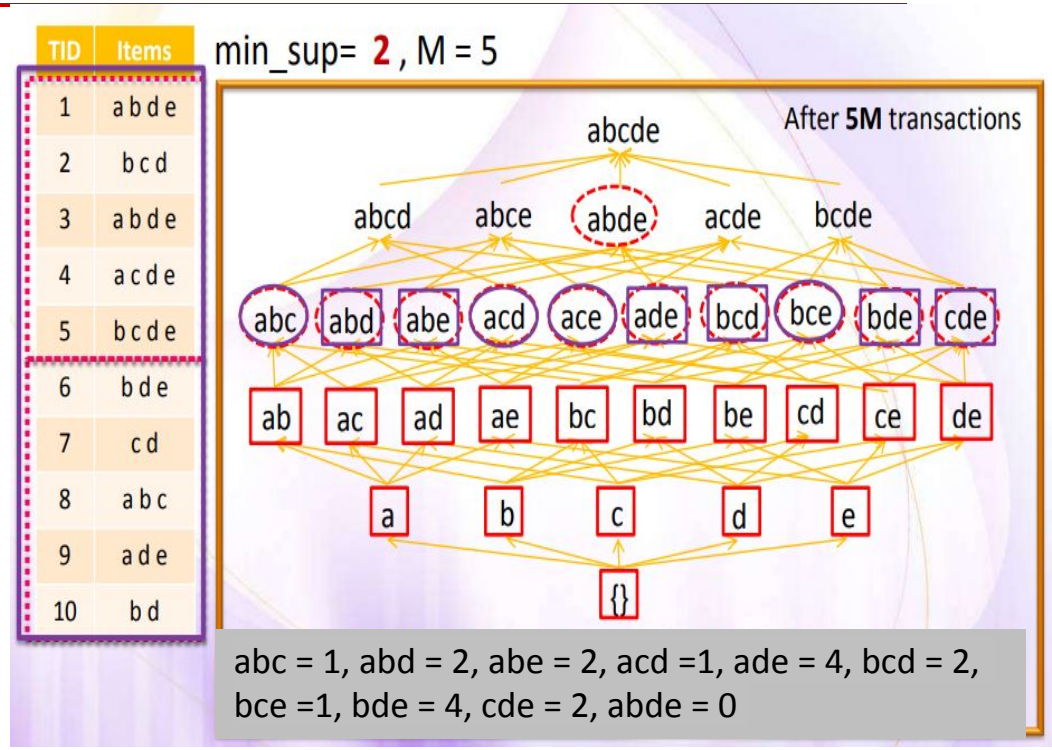
Dynamic Itemset Counting - Example





Dynamic Itemset Counting - Example





Pincer Search Algorithm

- Apriori Algorithm
  - Bottom-up, breadth-first search
  - number of database passes = largest size of frequent itemset
- Pincer Search Algorithm
  - Bi-directional search, which takes advantages of both the bottom-up as well as the top-down process
  - It attempts to find the frequent itemsets in a bottom-up manner but, at the same time it maintains a list of maximal frequent itemsets
  - In each pass, in addition to counting the supports of the candidate in the bottom-up direction, it also counts the supports of the some itemsets using a top-down approach, these are called the *Maximal Frequent Candidate set*

Pincer Search Algorithm

- Consider a pass k, during which itemsets of size k are to be classified.
- If some itemset that is an element of MFCS, say X, of cardinality greater than k is found to be frequent in this pass, then all its subset must be frequent.
- Therefore, all of its of cardinality k can be pruned from the candidate sets considered in the bottom-up direction in the pass.

Apriori Algorithm

Initialize: $k := 1$, C_1 = all the 1-itemsets;
read the database to count the support of C_1 to determine L_1.
L_1 := {frequent 1-itemsets};
$k := 2$; *// k represents the pass number//*
while $(L_{k-1} \neq \varnothing)$ ***do***
begin
  $C_k :=$ ***gen_candidate_itemsets*** *with the given* L_{k-1}
  prune(C_k)
  for all transactions $t \in T$ ***do***
  increment the count of all candidates in C_k that are contained in t;
  $L_k :=$ All candidates in C_k with minimum support ;
  $k := k + 1$;
end
Answer := $\cup_k L_k$;

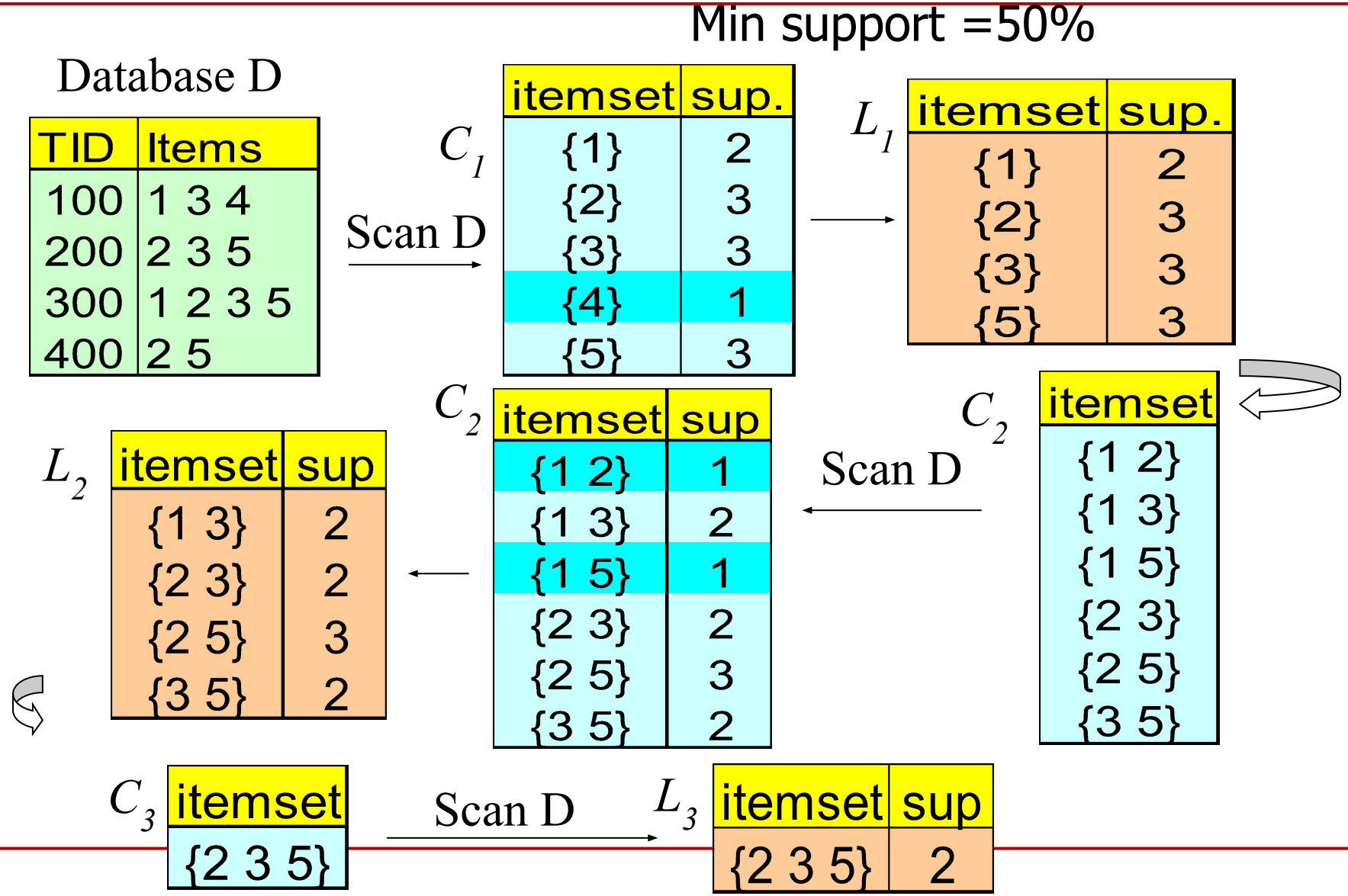
gen_candidate_itemsets *with the given* L_{k-1} as follows:

$C_k = \varnothing$
for all itemsets $l_1 \in L_{k-1}$ ***do***
for all itemsets $l_2 \in L_{k-1}$ ***do***
  if $l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \ldots \wedge l_1[k-1] < l_2[k-1]$
  then $c = l_1[1], l_1[2] \ldots l_1[k-1], l_2[k-1]$
  $C_k = C_k \cup \{c\}$

prune(C_k)

for all $c \in C_k$
for all $(k-1)$-subsets d of c ***do***
  if $d \notin L_{k-1}$
  then $C_k = C_k \setminus \{c\}$

The Apriori Algorithm — Example



Pincer Search Algorithm

$L_0 := \emptyset$; $k := 1$; $C_1 := \{\{i\} \mid i \in I\}$; $S_0 = \emptyset$;
MFCS := $\{\{1,2, \ldots, n\}\}$; MFS := $\emptyset$;
do until $C_k = \emptyset$ and $S_{k-1} = \emptyset$
    read database and count supports for C_k and MFCS;
    MFS := MFS $\cup$ {frequent itemsets in MFCS};
    S_k := {infrequent itemsets in C_k};
    call *MFCS-gen* algorithm if $S_k \neq \emptyset$;
    call *MFS-pruning* procedure;
    generate candidates C_{k+1} from C_k; (similar to a priori's generate & prune)
    if any frequent itemset in C_k is removed in MFS-pruning procedure
        call the recovery procedure to recover candidates to C_{k+1};
    call *MFCS prune procedure* to prune candidates in C_{k+1};
    $k := k+1$;
return MFS

MFCS-gen

for all itemsets $s \in S_k$
    for all itemsets $m \in$ MFCS
        if s is a subset of m
        MFCS := MFCS\{m};
        for all items $e \in$ itemset s
            if $m\backslash\{e\}$ is not a subset of any itemset in MFCS
                MFCS := MFCS $\cup \{m\backslash\{e\}\}$;
return MFCS

$$MFCS = \{1,2,3,4\}$$
$$S = \{1,2\}$$

$$MFCS = \{\{3,4\}\}, S = \{1,2\}$$
$$= \{\{3,4\},\{1,3,4\},\{2,3,4\}\}$$
$$= \{\{1,3,4\},\{2,3,4\}\}$$

Pincer Search Algorithm

Recovery

for all itemsets $l \in C_k$
  for all itemsets $m \in$ MFS
  if the first $k-1$ items in l are also in m
    /* suppose $m.item_j = l.item_{k-1}$ */
    for i from $j+1$ to $|m|$
      $C_{k+1} := C_{k+1} \cup \{\{l.item_1, l.item_2, \ldots, l.item_k, m.item_i\}\}$

MFS-Prune

  for all itemsets c in C_k
  if c is a subset of any itemset is the current MFS
  delete c from C_k;

MFCS-Prune

  for all itemsets c in C_{k+1}
  if c is not a subset of any itemset in the current MFCS
  delete c from C_{k+1};

- Suppose that the original frequent itemset L_3 is {{1,2,3}, {1,2,4}, {1,2,5}, {1,3,4}, {1,3,5}, {1,4,5}, {2,3,4}, {2,3,5}, {2,4,5}, {2,4,6}, {2,5,6}, {3,4,5}, {4,5,6}}
- Assume itemset {1,2,3,4,5} in the MFCS is determined to be frequent.
- Then all 3-itemsets of the original frequent set L_3 will be removed, except for {2,4,6}, {2,5,6}, and {4,5,6}.
- Since the Apriori-gen algorithm uses a (k − 1)-prefix test on the frequent set to generate new candidates, and no two itemsets in the current frequent set {{2,4,6}, {2,5,6}, {4,5,6}} share a 2-prefix, no candidate will be generated by applying the join procedure on this frequent set.
- However, the correct preliminary candidate set should be {{2,4,5,6}}.

Pincer Search Algorithm - Example

- $\sigma = 20\%$
- since T contains 15 transactions, it means that an itemsets that is supported by at least three transactions is a frequent set
- Pincer Search Algorithm
- Step 1:

$L_0 = \phi$; k =1;
$C_1 = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}\}$
$MFCS = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$MFS = \phi$

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- $L_0 = \phi$; k =1;
  $C_1 = \{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\},\{9\}\}$
  $MFCS = \{1,2,3,4,5,6,7,8,9\}$
  $MFS = \phi$
- **PASS ONE:**

  $\{1\} \rightarrow 2, \{2\} \rightarrow 6, \{3\} \rightarrow 6\ \{4\} \rightarrow 4, \{5\} \rightarrow 8,$
  $\{6\} \rightarrow 5, \{7\} \rightarrow 7, \{8\} \rightarrow 4\ \{9\} \rightarrow 2$
  $\{1,2,3,4,5,6,7,8,9\} \rightarrow 0$

  So MFCS: $= \{1,2,3,4,5,6,7,8,9\}$ and $MFS = \phi$;
  $L_1 = \{\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\}\}$
  $S_1 = \{\{1\},\{9\}\}$

  S_1 **is not empty, need to update MFCS**

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- MFCS: $= \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  S_1 is not empty, need to update MFCS
  - For 1 in S_1 and for $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ in MFCS, we get the new elements in MFCS as $\{2, 3, 4, 5, 6, 7, 8, 9\}$
  - similarly, For 9 in S_1 and for $\{2, 3, 4, 5, 6, 7, 8, 9\}$ in MFCS, we get the new elements in MFCS as $\{2, 3, 4, 5, 6, 7, 8\}$
  - candidate Itemsets C_2

    $\{\{2, 3\}, \{2, 4\}, \{2, 5\}, \{2, 6\}, \{2, 7\}, \{2, 8\}, \{3, 4\}, \{3, 5\},$
    $\{3, 6\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{4, 6\}, \{4, 7\}, \{4, 8\},$
    $\{5, 6\}, \{5, 7\}, \{5, 8\}, \{6, 7\}, \{6, 8\}, \{7, 8\}\}$

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- Pass TWO
  - $C_2 = \{\{2,3\},\{2,4\},\{2,5\},\{2,6\},\{2,7\},\{2,8\},\{3,4\},\{3,5\},\{3,6\},\{3,7\},\{3,8\},\{4,5\},\{4,6\},\{4,7\},\{4,8\},\{5,6\},\{5,7\},\{5,8\},\{6,7\},\{6,8\},\{7,8\}\}$
  - MFCS: $\{2,3,4,5,6,7,8\}$
  - Read the database to count the support of elements in C_2

$\{2,3\} \rightarrow 3, \{2,4\} \rightarrow 3, \{2,5\} \rightarrow 0, \{2,6\} \rightarrow 2, \{2,7\} \rightarrow 2, \{2,8\} \rightarrow 1, \{3,4\} \rightarrow 1, \{3,5\} \rightarrow 3, \{3,6\} \rightarrow 0, \{3,7\} \rightarrow 3, \{3,8\} \rightarrow 0, \{4,5\} \rightarrow 1, \{4,6\} \rightarrow 1, \{4,7\} \rightarrow 2, \{4,8\} \rightarrow 1, \{5,6\} \rightarrow 3, \{5,7\} \rightarrow 5, \{5,8\} \rightarrow 2, \{6,7\} \rightarrow 3, \{6,8\} \rightarrow 2, \{7,8\} \rightarrow 0\}$

$\{2,3,4,5,6,7,8\} \rightarrow 0; MFS = \phi$

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- $\{2,3\} \rightarrow 3, \{2,4\} \rightarrow 3, \{2,5\} \rightarrow 0, \{2,6\} \rightarrow 2, \{2,7\} \rightarrow 2, \{2,8\} \rightarrow 1, \{3,4\} \rightarrow 1, \{3,5\} \rightarrow 3, \{3,6\} \rightarrow 0, \{3,7\} \rightarrow 3, \{3,8\} \rightarrow 0, \{4,5\} \rightarrow 1, \{4,6\} \rightarrow 1, \{4,7\} \rightarrow 2, \{4,8\} \rightarrow 1, \{5,6\} \rightarrow 3, \{5,7\} \rightarrow 5, \{5,8\} \rightarrow 2, \{6,7\} \rightarrow 3, \{6,8\} \rightarrow 2, \{7,8\} \rightarrow 0\}$

$L_2 = \{\{2,3\}, \{2,4\}, \{3,5\}, \{3,7\}, \{5,6\}, \{5,7\}, \{6,7\}\}$

$\{2,3,4,5,6,7,8\} \rightarrow 0;$

$S_2 = \{\{2,5\}, \{2,6\}, \{2,7\}, \{2,8\}, \{3,4\}, \{3,6\}, \{3,8\}, \{4,5\}, \{4,6\}, \{4,7\}, \{4,8\}, \{5,8\}, \{6,8\}, \{7,8\}\}$

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- $L_2 = \{\{2,3\},\{2,4\},\{3,5\},\{3,7\},\{5,6\},\{5,7\},\{6,7\}\}$

  $S_2 = \{\{2,5\},\{2,6\},\{2,7\},\{2,8\},\{3,4\},\{3,6\},$
  $\{3,8\},\{4,5\},\{4,6\},\{4,7\},\{4,8\},\{5,8\},\{6,8\},\{7,8\}\}$

  $MFCS: \{2,3,4,5,6,7,8\}$

  - For {2,5} $in\ S_2\ and\ for$ {2,3,4,5,6,7,8} in MFCS, we get the new elements in MFCS as {3,4,5,6,7,8} and {2,3,4, 6,7,8}
  - For {2,6} $in\ S_2\ and\ for$ {3,4,5,6,7,8} in MFCS, since {2,6} is not contained in this elements of MFCS and hence no action
  - For {2,6} $in\ S_2\ and\ for$ {2,3,4,6,7,8} in MFCS, we get the new elements in MFCS as {3,4,6,7,8} and {2,3,4,7,8}
  - Since {3,4,6,7,8} is already contained MFCS ({3,4,5,6,7,8}), it is excluded from MFCS

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- $S_2 = \{\{2,5\},\{2,6\},\{2,7\},\{2,8\},\{3,4\},\{3,6\},\{3,8\},\{4,5\},\{4,6\},\{4,7\},\{4,8\},\{5,8\},\{6,8\},\{7,8\}\}$

- So at this stage MFCS= {{3,4,5,6,7,8}, {2,3,4,7,8} }
- For {2,7} in S_2
  MFCS= {{3,4,5,6,7,8}, {3,4,7,8}, {2,3,4,8}}
  MFCS= {{3,4,5,6,7,8}, {2,3,4,8}}
- For {2,8} in S_2
  MFCS= {{3,4,5,6,7,8}, {3,4,8}, {2,3,4}}
  MFCS= {{3,4,5,6,7,8}, {2,3,4}}
- For {3,4} in S_2
  MFCS= {{4,5,6,7,8}, {3,5,6,7,8} ,{2,4}, {2,3}}

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- $S_2 = \{\{2,5\},\{2,6\},\{2,7\},\{2,8\},\{3,4\},\{3,6\},\{3,8\},\{4,5\},\{4,6\},\{4,7\},\{4,8\},\{5,8\},\{6,8\},\{7,8\}\}$
- So at this stage MFCS= MFCS= {{4,5,6,7,8}, {3,5,6,7,8} ,{2,4}, {2,3}}
- For {3,6} in S_2
  MFCS= {{4,5,6,7,8}, {5,6,7,8}, {3,5,7,8} {2,4}, {2,3}} = {{4,5,6,7,8}, {3,5,7,8} {2,4}, {2,3}}
- For {3,8} in S_2
  MFCS= {4,5,6,7,8}, {5,7,8}, {3,5,7}, {2,4}, {2,3}}
  ={4,5,6,7,8}, {3,5,7}, {2,4}, {2,3}}
- For {4,5} in S_2
  MFCS= {{5,6,7,8}, {4,6,7,8}, {3,5,7}, {2,4}, {2,3}}

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- $S_2 = \{\{2,5\},\{2,6\},\{2,7\},\{2,8\},\{3,4\},\{3,6\},\{3,8\},\{4,5\},\{4,6\},\{4,7\},\{4,8\},\{5,8\},\{6,8\},\{7,8\}\}$
- So at this stage MFCS= {{5,6,7,8},{4,6,7,8},{3,5,7},{2,4},{2,3}}
- For {4,6} in S_2
  - MFCS= {{5,6,7,8},{6,7,8},{4,7,8},{3,5,7},{2,4},{2,3}}
  - MFCS= {{5,6,7,8},{4,7,8},{3,5,7},{2,4},{2,3}}
- For {4,7} in S_2
  - MFCS= {{5,6,7,8},{7,8},{4,8},{3,5,7},{2,4},{2,3}}
  - MFCS= {{5,6,7,8},{4,8},{3,5,7},{2,4},{2,3}}
- For {4,8} in S_2
  - MFCS= {{5,6,7,8},{3,5,7},{2,4},{2,3}}

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- $S_2 = \{\{2,5\},\{2,6\},\{2,7\},\{2,8\},\{3,4\},\{3,6\},\{3,8\},\{4,5\},\{4,6\},\{4,7\},\{4,8\},\{5,8\},\{6,8\},\{7,8\}\}$
- So at this stage MFCS={{5,6,7,8}, {3,5,7}, {2,4}, {2,3}}
- For {5,8} in S_2
  MFCS= {{6,7,8}, {5,6,7}, {3,5,7}, {2,4}, {2,3}}
- For {6,8} in S_2
  MFCS= {{7,8}, {6,7}, {5,6,7}, {3,5,7}, {2,4}, {2,3}}
  MFCS= {{7,8}, {5,6,7}, {3,5,7}, {2,4}, {2,3}}
- For {7,8} in S_2
  MFCS= {{8}, {7}, {5,6,7}, {3,5,7}, {2,4}, {2,3}}
  MFCS= {{8}, {5,6,7}, {3,5,7}, {2,4}, {2,3}}

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

Pincer Search Algorithm - Example

- $L_2 = \{\{2,3\},\{2,4\},\{3,5\},\{3,7\},\{5,6\},\{5,7\},\{6,7\}\}$
- MFCS= {{8}, {5,6,7}, {3,5,7}, {2,4}, {2,3}}
- We generate the candidate sets as
  - $C_3 = \{\{2,3,4\},\{3,5,7\},\{5,6,7\}\}$
  - MFCS Pruning Stage,
    - In the pruning stage the itemsets {2,3,4} are pruned and hence,
    - $C_3 = \{\{3,5,7\},\{5,6,7\}\}$
- **At this stage we make one more pass of the data set to count the supports of $\{\{3,5,7\},\{5,6,7\}\}$**
  - $\{3,5,7\} \rightarrow 3, \{5,6,7\} \rightarrow 1$
  - $L_3 = \{3,5,7\}$
  - $S_3 = \{5,6,7\}$
  - For $\{5,6,7\}$ in S_3
    - MFCS= {{8}, {3,5,7}, {2,4}, {2,3}}

A1	A2	A3	A4	A5	A6	A7	A8	A9
1	0	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	0
0	0	0	1	1	0	1	0	0
0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	1	1	1	0	0	0	0	0
0	1	0	0	0	1	1	0	1
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	0
1	0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0	1

FP-tree Growth Algorithm

- Why FP-Tree and not Apriori?
  - Lots of frequent patterns
    - Big set of items
    - Low minimum support threshold
  - Long patterns
- Why: Candidate sets become huge
  - 10^4 frequent patterns of length 1 → 10^7 length 2 candidates
  - Discovering pattern of length 100 requires at least 2^{100} candidates (nr of subsets)
  - Repeated database scans costly (long patterns)

FP-tree Growth Algorithm

- FP-Growth: allows frequent itemset discovery without candidate itemset generation. Two step approach
  - Step 1: Build a compact data structure called the FP-tree
    - Built using 2 passes over the data-set.
    - A frequent pattern tree (FP-tree) is a tree structure consisting of an item-prefix-tree and a frequent-item-header table
  - Step 2: Extracts frequent itemsets directly from the FP-tree

Step 1: FP-Tree Construction

- FP-Tree is constructed using 2 passes over the data-set
- Pass 1:
  - Scan data and find support for each item.
  - Discard infrequent items.
  - Sort frequent items in decreasing order based on their support.
- Use this order when building the FP-Tree, so common prefixes can be shared.
- In FP-Tree, nodes correspond to items and have a counter

Step 1: FP-Tree Construction

Pass 2:

1. FP-Growth reads 1 transaction at a time and maps it to a path
2. Fixed order is used, so paths can overlap when transactions share items (when they have the same prefix).
   - In this case, counters are incremented
3. Pointers are maintained between nodes containing the same item, creating singly linked lists (dotted lines)
   - The more paths that overlap, the higher the compression. FP-tree may fit in memory.
4. Frequent itemsets extracted from the FP-Tree.

Step 1: FP-Tree Construction (Example)

Transactions

A B C E F O
A C G
E I
A C D E G
A C E G L
E J
A B C E F P
A C D
A C E G M
A C E G N

Freq. 1-Itemsets.
Supp. Count ≥2

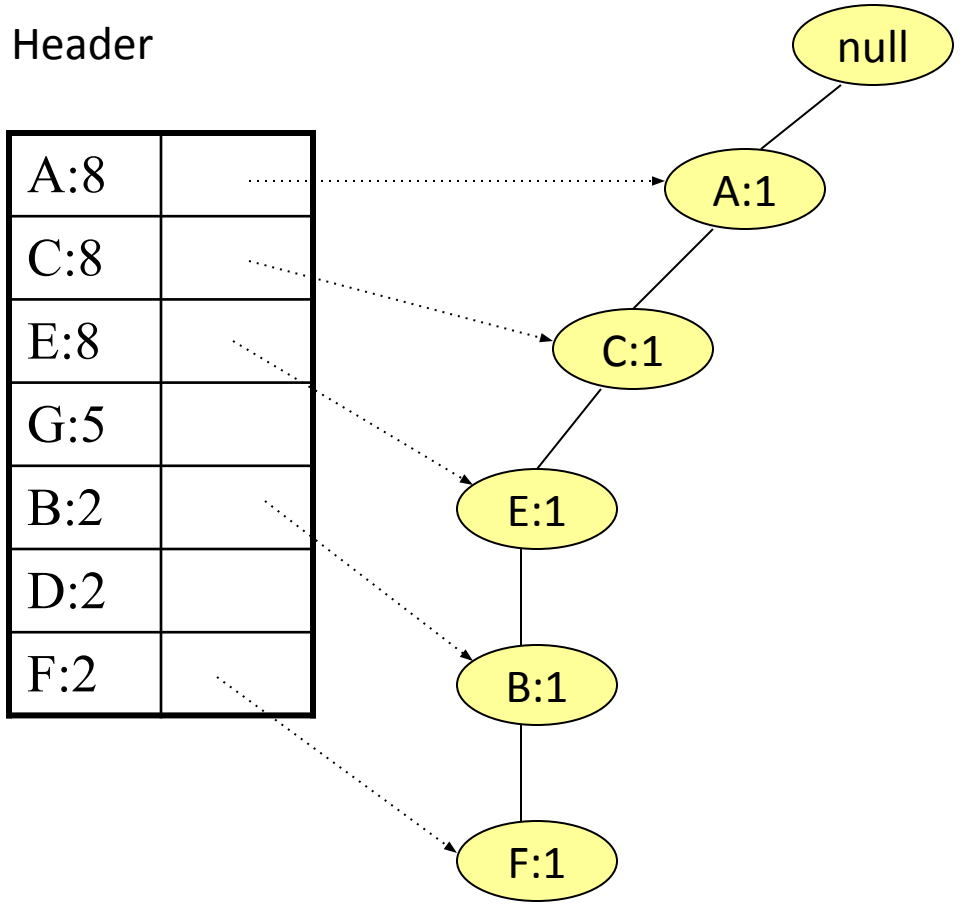
A:8	
C:8	
E:8	
G:5	
B:2	
D:2	
F:2	

Transactions with items sorted based on frequencies, and ignoring the infrequent items.

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G

FP-Tree after reading 1st transaction

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree after reading 2nd transaction

A C E B F

A C G

E

A C E G D

A C E G

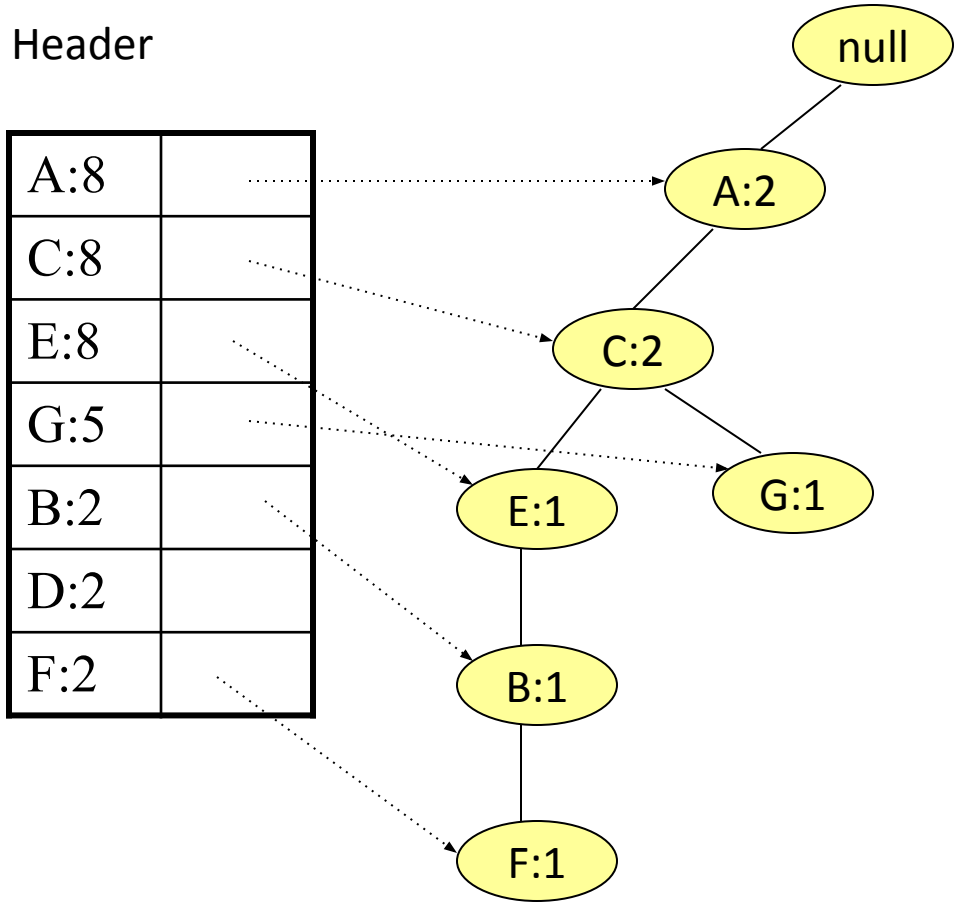
E

A C E B F

A C D

A C E G

A C E G



FP-Tree after reading 3rd transaction

A C E B F

A C G

E

A C E G D

A C E G

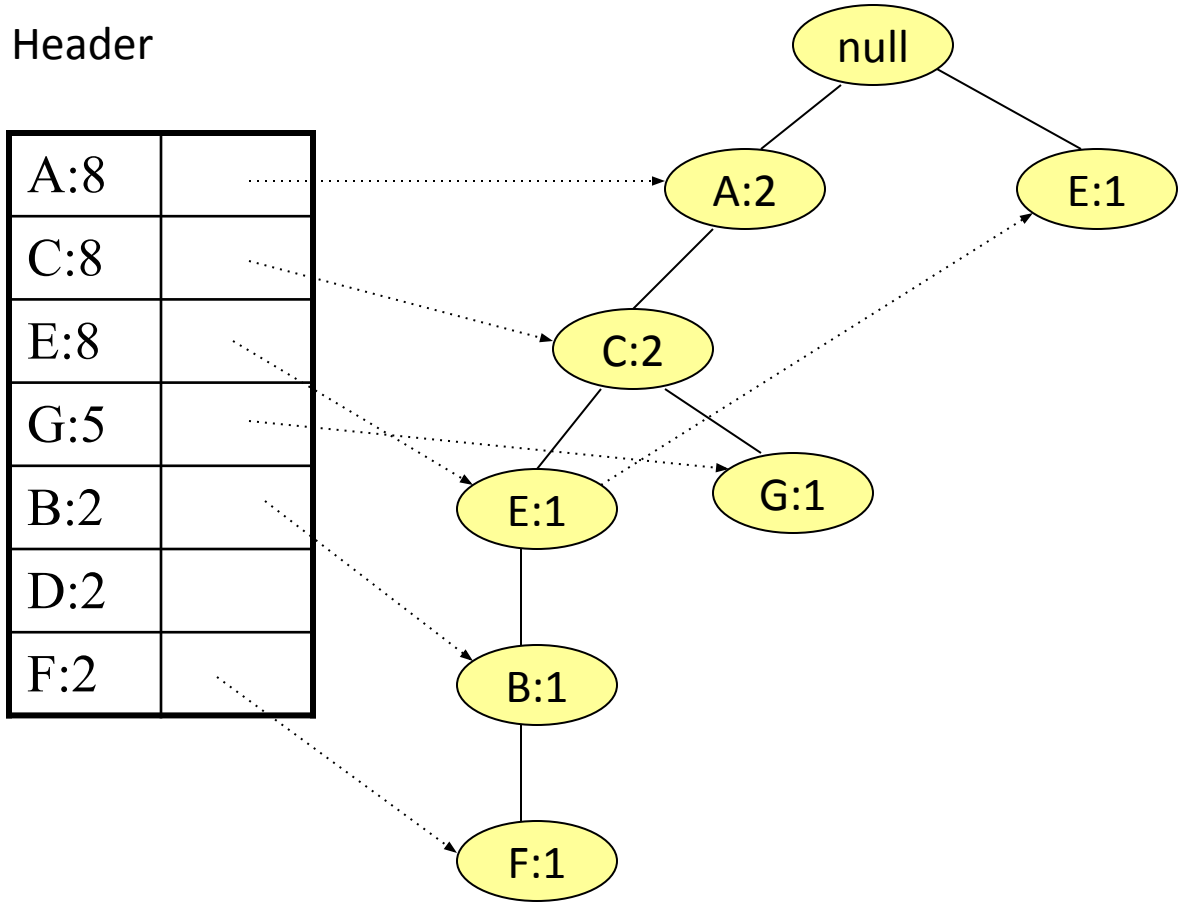
E

A C E B F

A C D

A C E G

A C E G



FP-Tree after reading 4th transaction

A C E B F

A C G

E

A C E G D

A C E G

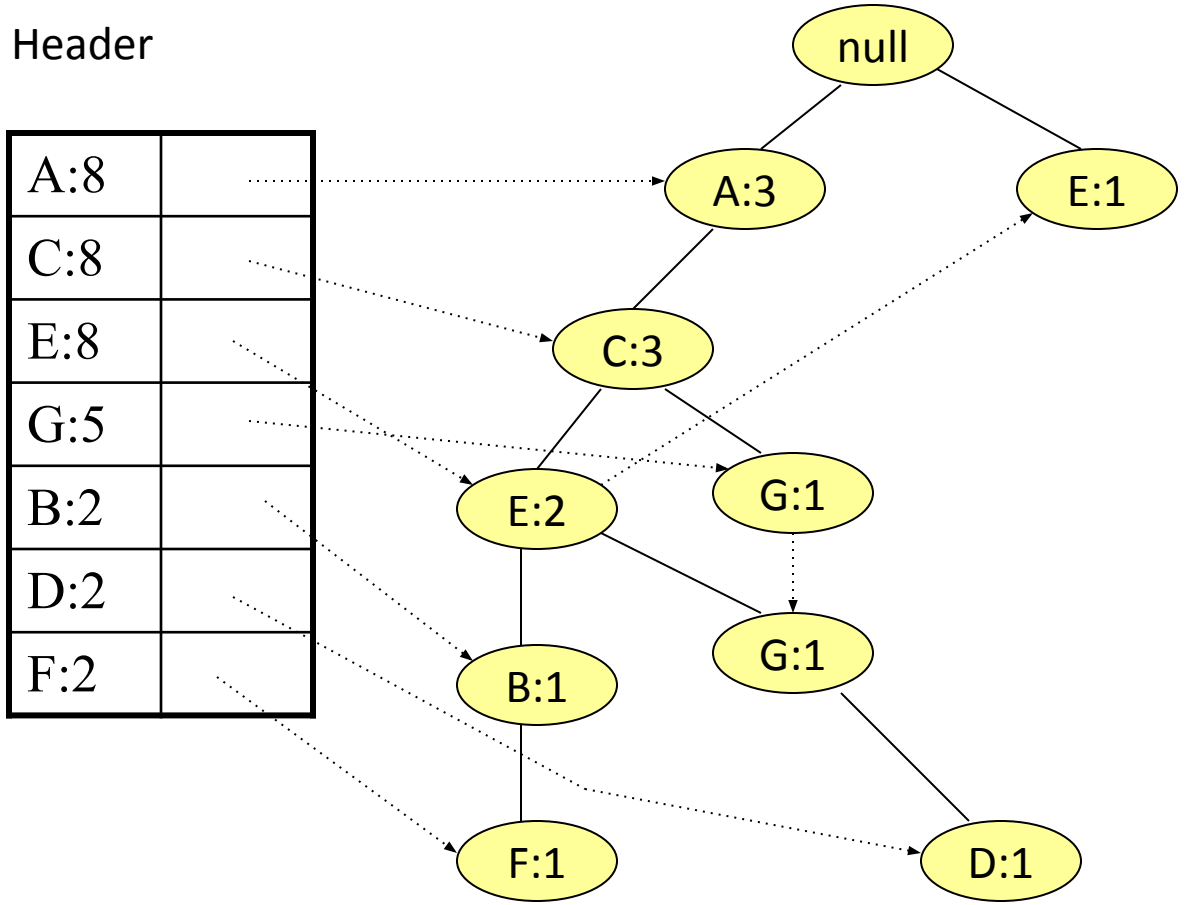
E

A C E B F

A C D

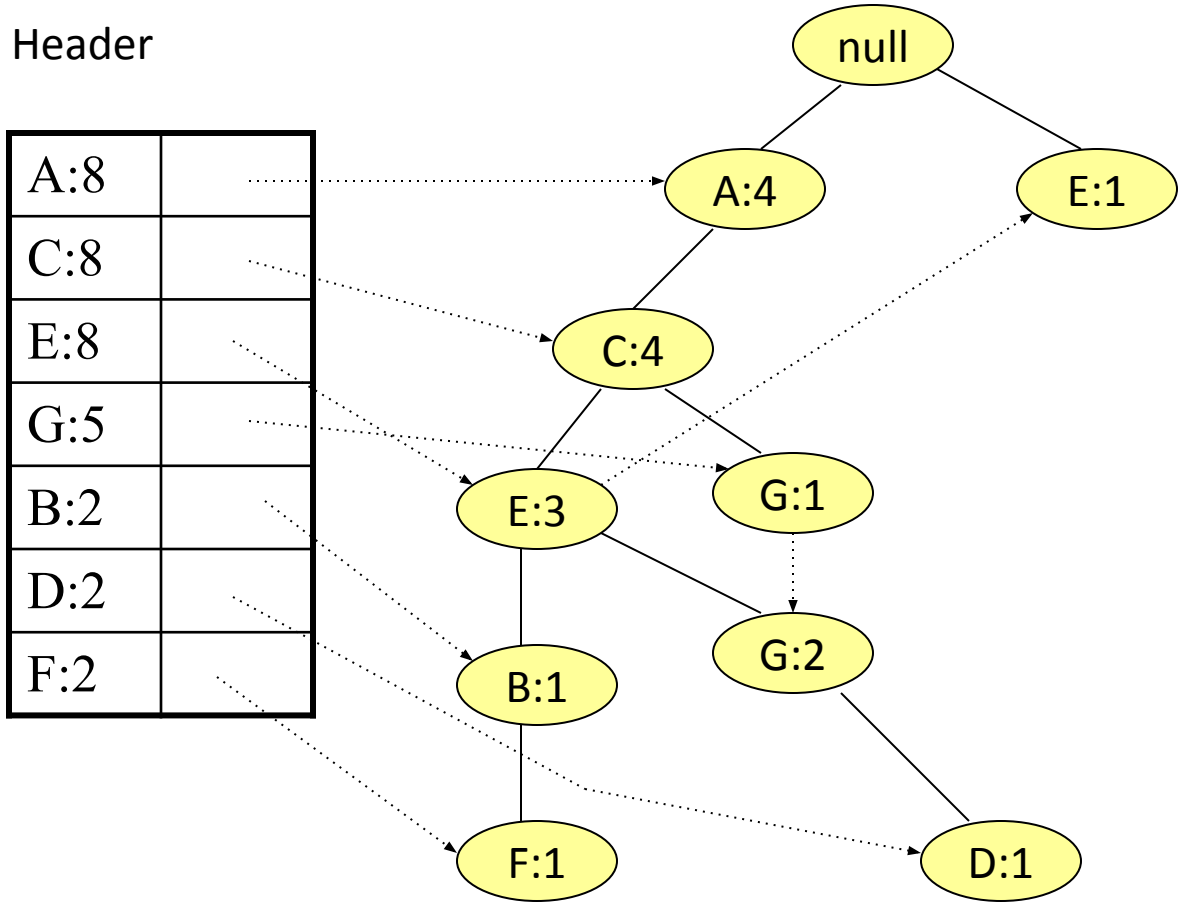
A C E G

A C E G



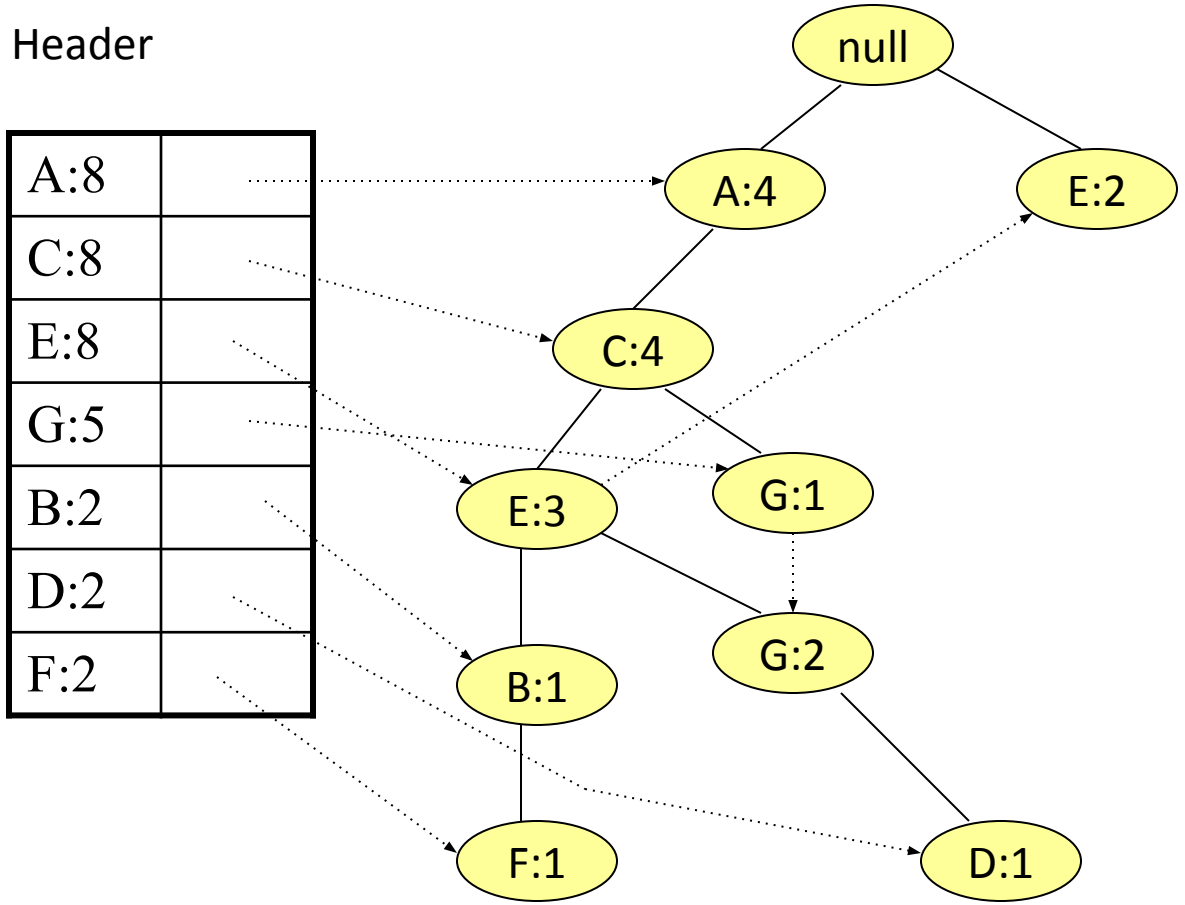
FP-Tree after reading 5th transaction

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



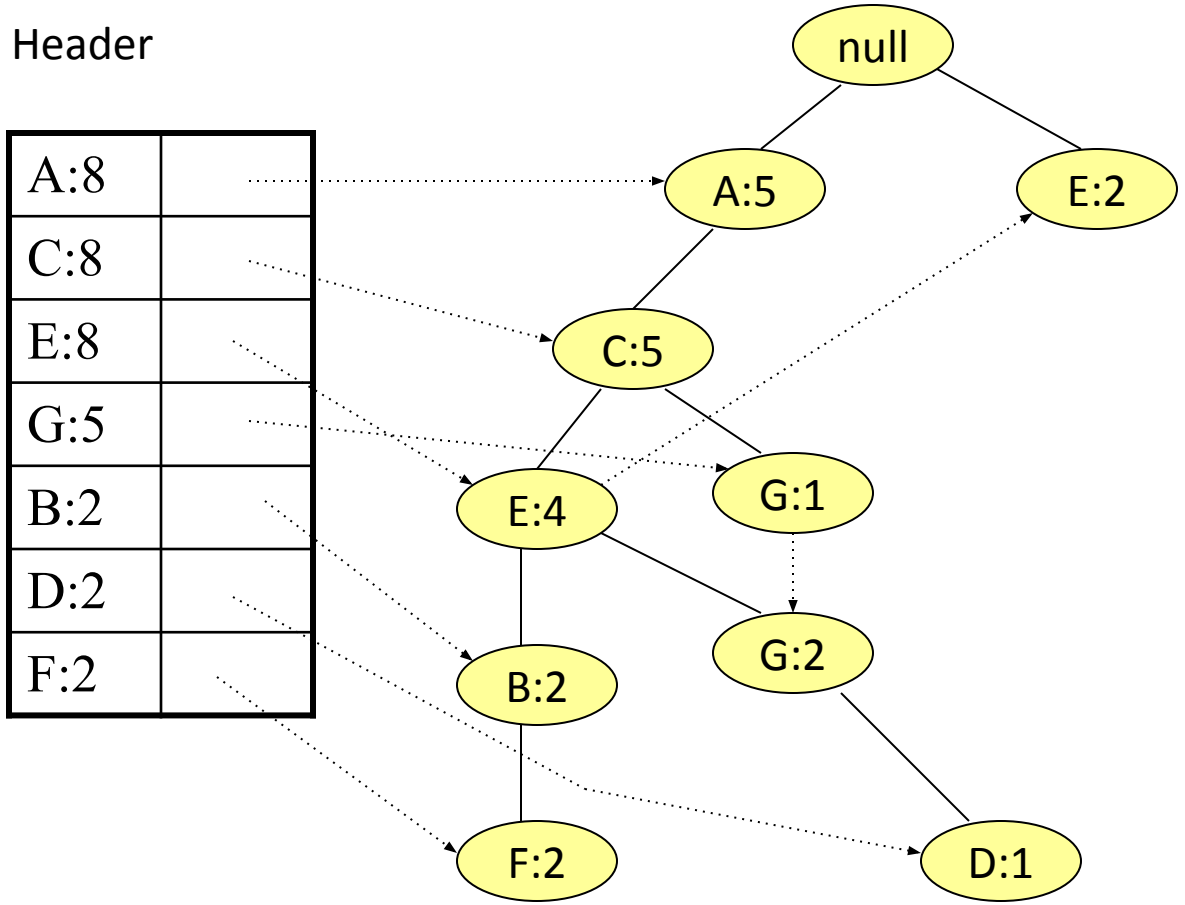
FP-Tree after reading 6th transaction

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



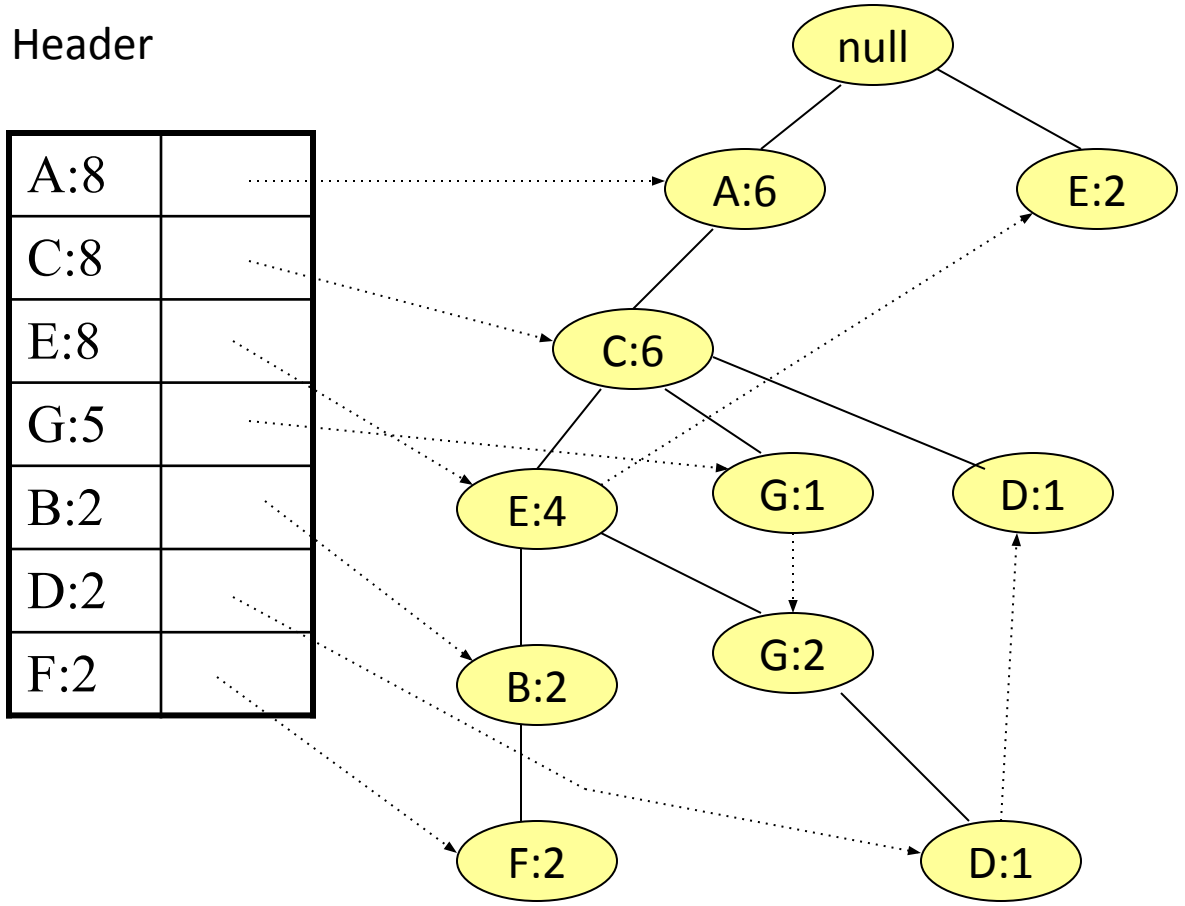
FP-Tree after reading 7th transaction

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



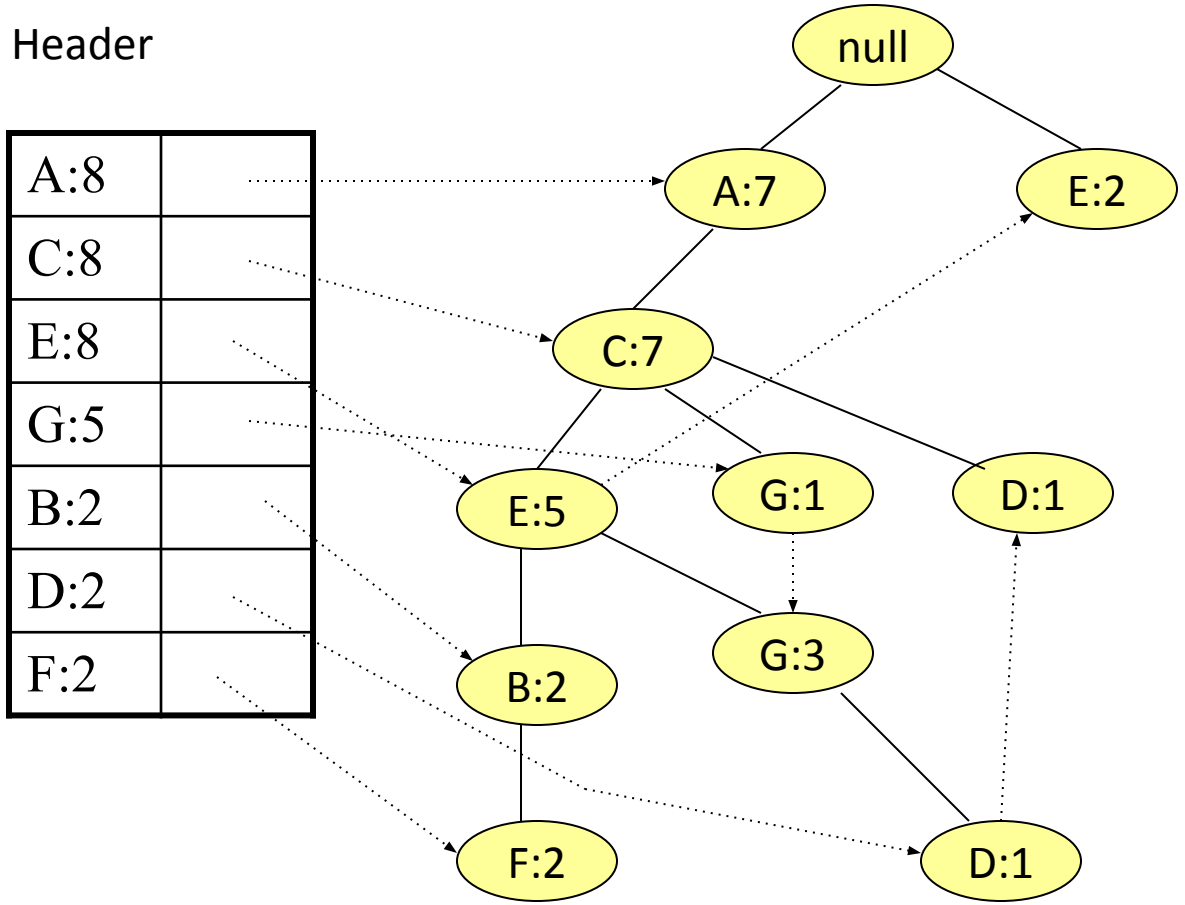
FP-Tree after reading 8th transaction

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



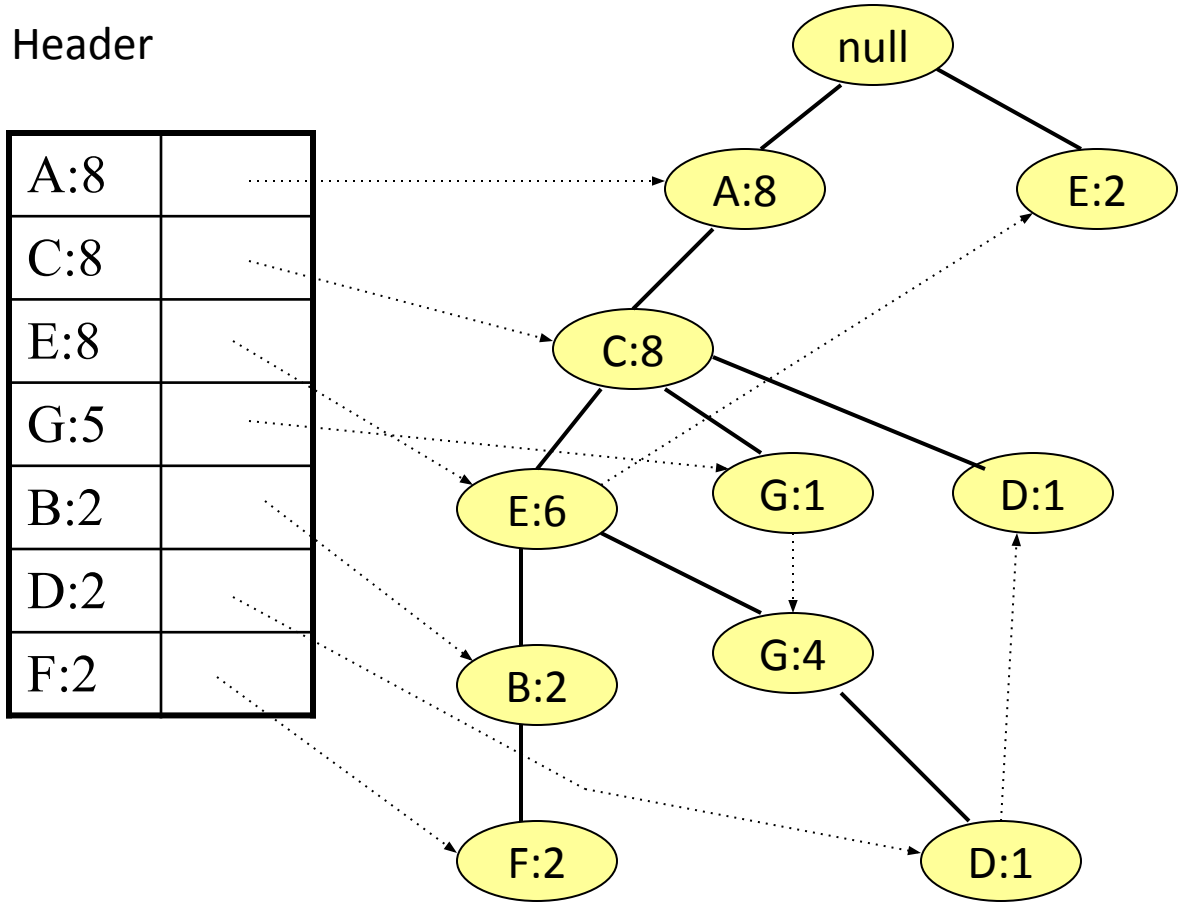
FP-Tree after reading 9th transaction

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G



FP-Tree after reading 10th transaction

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G

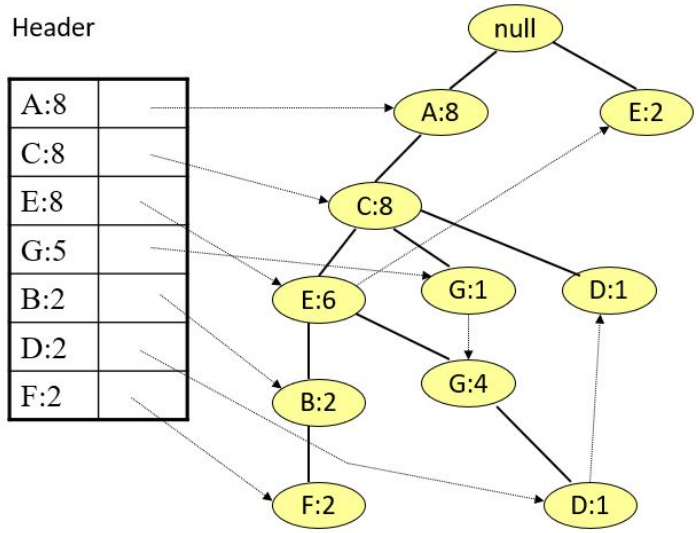


FP-Tree size

- The FP-Tree usually has a smaller size than the uncompressed data - typically many transactions share items (and hence prefixes).
  - Best case scenario: all transactions contain the same set of items.
    - 1 path in the FP-tree
  - Worst case scenario: every transaction has a unique set of items (no items in common)
    - Size of the FP-tree is at least as large as the original data.
    - Storage requirements for the FP-tree are higher - need to store the pointers between the nodes and the counters.
- The size of the FP-tree depends on how the items are ordered
- Ordering by decreasing support is typically used but it does not always lead to the smallest tree (it's a heuristic).

Step 2: Frequent Itemset Generation

- FP growth generates frequent itemsets from an FP tree by exploring the tree in a bottom up fashion - from the leaves towards the root
- Divide and conquer: first look for frequent itemsets ending in F, then BF, etc. . . then D, then GD, etc. . .
- First, extract prefix path sub-trees ending in an item(set).

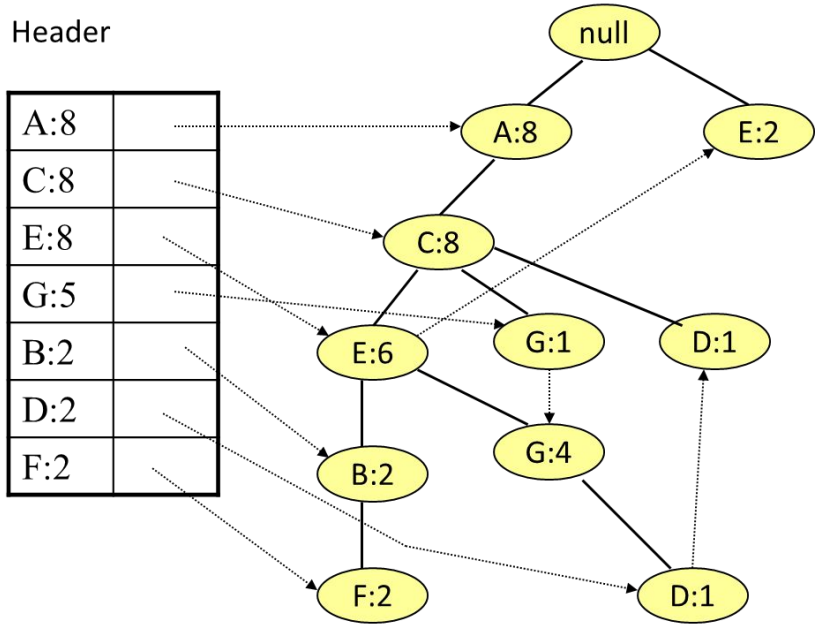


3 Major Steps

- Starting the processing from the end of list L:
  - Step1:
    - Construct conditional pattern base for each item in the header table
  - Step 2:
    - Construct conditional FP-tree from each conditional for each pattern base
  - Step 3:
    - Recursively mine conditional FP-trees and grow frequent patterns obtained so far. If the conditional FP-tree contains a single path, simply enumerate all the patterns

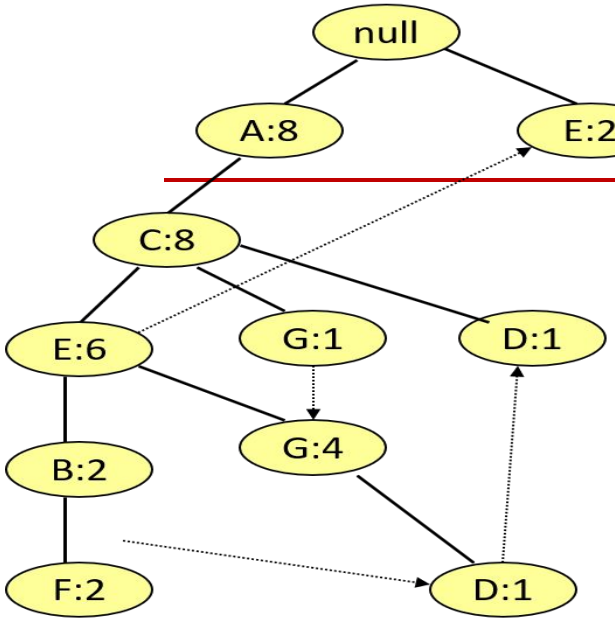
Step 1: Construct Conditional Pattern Base

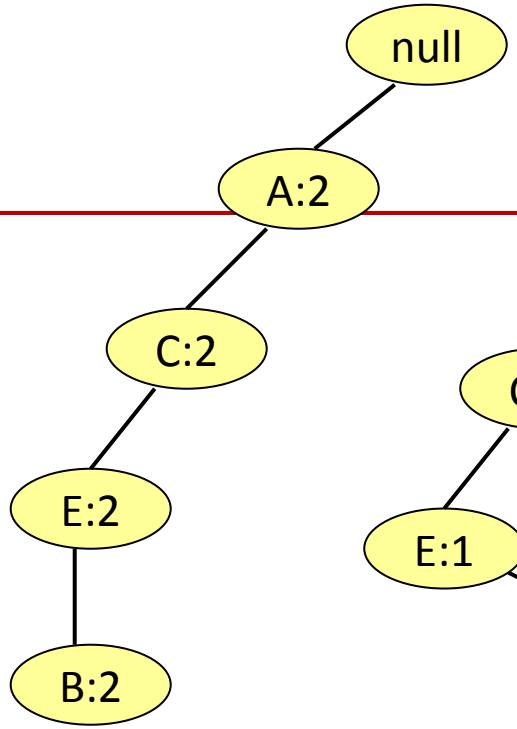
- Starting at the bottom of frequent-item header table in the FP-tree
- Traverse the FP-tree by following the link of each frequent item
- Accumulate all of transformed prefix paths of that item to form a conditional pattern base



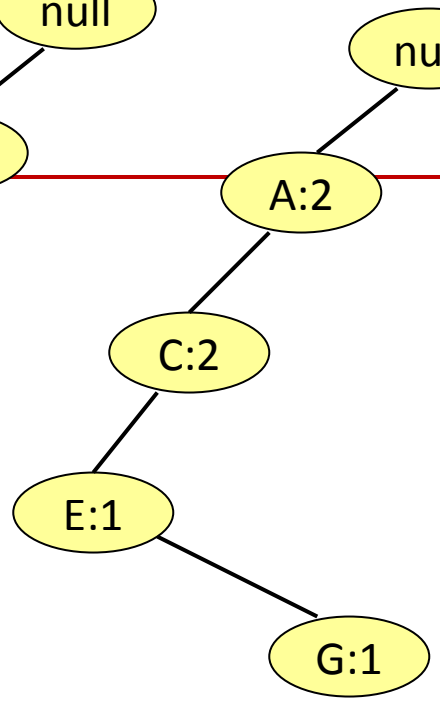
***Conditional* pattern bases**

itemcond.	***pattern base***
F	*ACEB:2*
D	*ACEG:1, AC:1*
B	*ACE:2*
G	*ACE: 4, AC:1,*
E	*AC:6*
C	*A:8*
A	*{}*

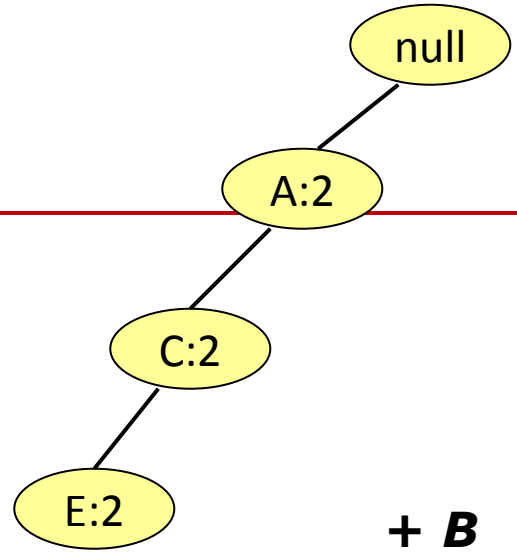




+ *F*



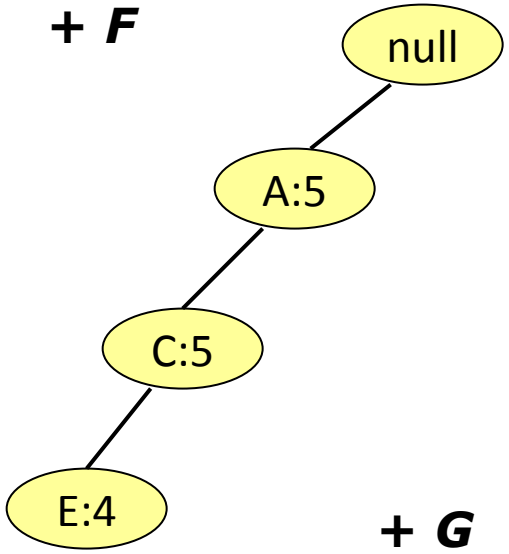
+ *D*



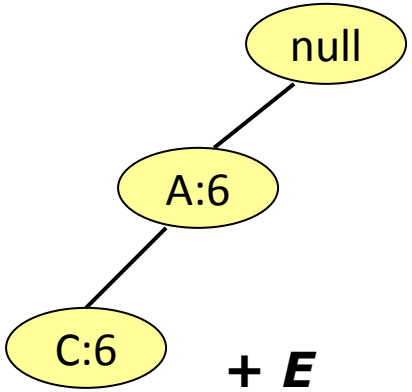
+ *B*

***Conditional* pattern bases**

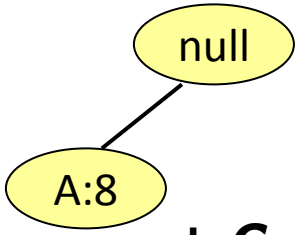
item	***cond. pattern base***
F	*ACEB:2*
D	*ACEG:1, AC:1*
B	*ACE:2*
G	*ACE: 4, AC:1,*
E	*AC:6*
C	*A:8*
A	*{}*



+ *G*



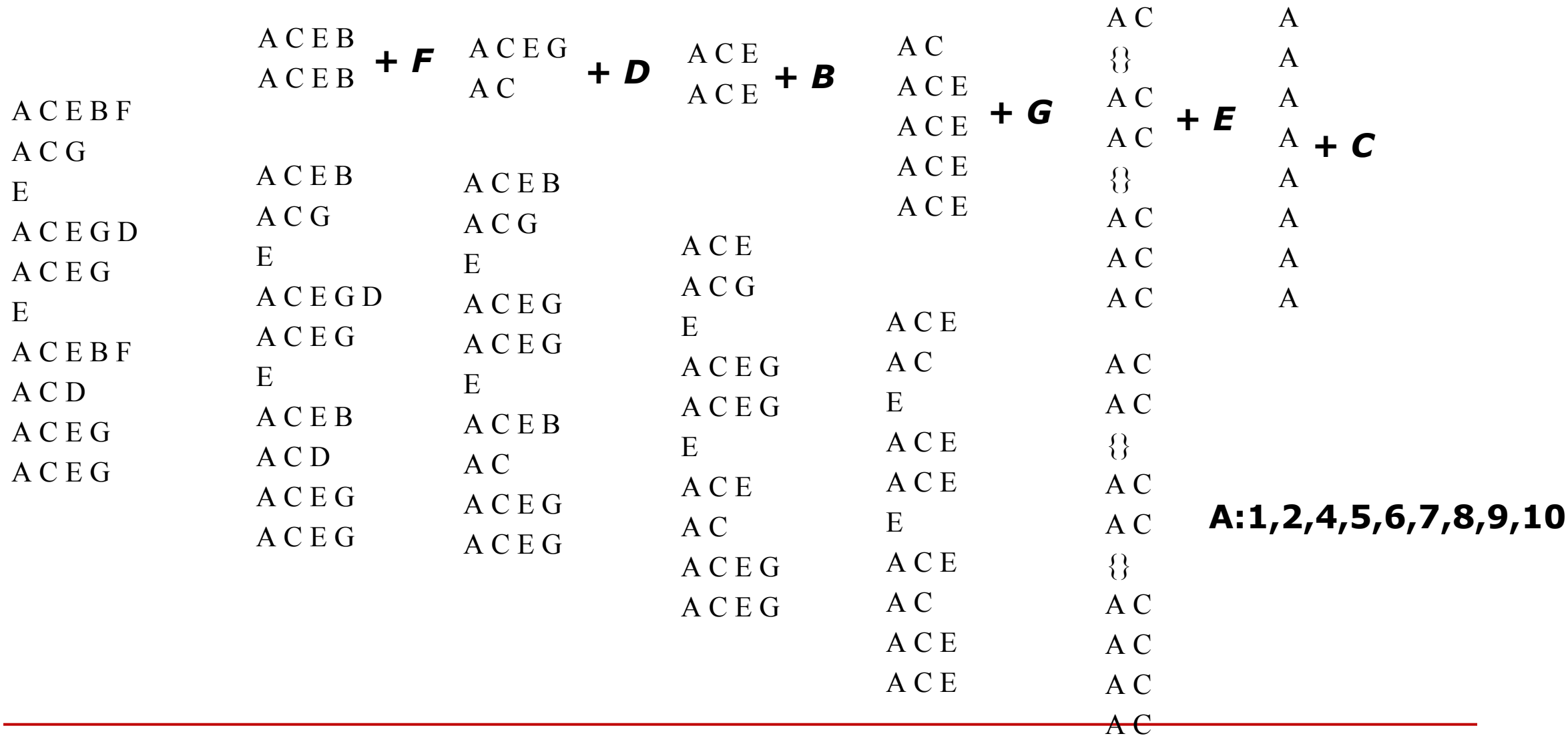
+ *E*



+ *C*

A:8

Step 2: Construct Conditional FP Tree



min_sup = 3

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G

A C E B
A C E B
+ *F*

A C E G
A C
+ *D*

A C E
A C E
+ *B*

A C
A C E
A C E
A C E
A C E
+ *G*

A C
A C E
A C E
A C E
A C E
+ *G*

G: 5
AG: 5
CG: 5
EG: 4
ACG: 5
AEG: 4
CEG: 4
ACEG: 4

min_sup = 3

A C E B F
A C G
E
A C E G D
A C E G
E
A C E B F
A C D
A C E G
A C E G

A C
{}
A C
A C
{}
A C
A C
A C

+ *E*

A C
{}
A C
A C
{}
A C
A C
A C

+ *E*

E: 8
AE: 6
CE: 6
ACE: 6

A
A
A
A
A
A
A
A

+ *C*

A
A
A
A
A
A
A
A

+ *C*

C: 8
AC: 8

A:8

Incremental Algorithm

- One common assumption in all the methods discussed so far is that the database itself does not change
- In practice, no transaction database is static. It is possible that more transactions are added intermittently to the database
- How to update the associations rules when the database is updated ?
  - One simple method will be to compute afresh the set of frequent itemsets
    - Infeasible
- Can we make use of earlier computation so that that I/O operations requires for computation of updated set of frequent itemset reduce

Incremental Algorithm

- T_{old} old database
- L_{old} frequent itemsets in T_{old}
- Assume that the support value of frequent sets are also known
- T_{new} new database (new set of transactions)
- $T_{whole} = T_{old} \cup T_{new}$
- L_{new} frequent itemsets in T_{new} (It is assumed that L_{new} is unknown)
- Goal: determine the set of all frequent sets L_{whole} of T_{whole}

Incremental Algorithm

- Observations
  1. An itemsets will be in L_{whole} if it is an element of both L_{new} and L_{old}
  2. Any itemset which is neither is L_{new} nor in L_{old} cannot be in L_{whole}
- Thus, if we simply start finding L_{new} we can use observation 1 to determine L_{whole} by making some pass over T_{new}
- Now, consider the following cases
  - The itemsets of L_{old} that are frequent in T_{new} can be determined
  - The itemset that belongs to L_{old} but not is L_{new} are automatically eliminated
  - The itemsets that are neither in L_{new} nor in L_{old} are not considered

Promoted Boarder

- An itemset that was boarder set before update and becomes a frequent set after update
- The existence of a promoted boarder is an indication that we may have to read the old transaction database once again.
- The incremental algorithm to discover frequent sets critically depends on this observation

Border set generation

Initialize: $k := 1$, $C_1 =$ all the 1-itemsets;
read the database to count the support of C_1 to determine L_1.
$L_1 :=$ {frequent 1-itemsets};
$k := 2$; *// k represents the pass number//*
while $(L_{k-1} \neq \emptyset)$ ***do***
begin
    $C_k :=$ ***gen_candidate_item sets*** *with the given* L_{k-1}
    prune C_k
    for all transactions $t \in T$ ***do***
    increment the count of all candidates in C_k that are contained in t ;
    $L_k := \{c \in C_k \mid s(c)_T \geq \sigma\}$;
    $B_k := \{c \in C_k \mid s(c)_T < \sigma\}$;
    $k := k + 1$;
end
    $L := \cup_k L_k$;
    $B := \cup_k B_k$;

gen_candidate_itemsets *with the given* L_{k-1} as follows:

$C_k = \emptyset$
for all itemsets $l_1 \in L_{k-1}$ ***do***
for all itemsets $l_2 \in L_{k-1}$ ***do***
    if $l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \ldots \wedge l_1[k-1] < l_2[k-1]$
    then $c = l_1[1], l_1[2] \ldots l_1[k-1], l_2[k-1]$
    $C_k = C_k \cup \{c\}$

prune(C_k)

for all $c \in C_k$
for all $(k-1)$-subsets d of c ***do***
    if $d \notin L_{k-1}$
    then $C_k = C_k \setminus \{c\}$

Boarder Algorithm

- The border algorithm maintain support counter for all frequent sets and all border sets
- Count the supports of the itemsets of $L_{old} \cup B_{old}$
  - This requires one pass over the incremental portion of the database
  - Let, F contains the itemset of L_{old} which become frequent set in T_{whole}
  - Let, B contains all the border sets whose support count reach the min support level and hence, are promoted boarder set
    - If there is no promoted border, then F contains all the frequent sets of T_{whole}
    - But if there is at least one promoted border, then the algorithm generates candidate sets which are superset of promoted border sets (one more pass of the dataset)

Association Rules with Item Constraints

- In practice, the users are often interested only in a subset of associations
  - For instance, those containing at least one item from a user-defined subset of items
  - Straightforward of doing this is to carry out a filtering as a post-processing step
  - However, it would be desirable to incorporate these constraints within the steps of the association rule mining algorithm

Association Rules with Item Constraints

- Let B be a Boolean expression over the set of itemsets A
- Lets assume that the B is in disjunctive normal form $D_1 \vee D_2 \vee \cdots \vee D_m$, where each of the D_i is of the form $a_{i1} \wedge a_{i2} \wedge \cdots \wedge a_{in}$
- Each of the a refers either the presence of an item, I_{ij} or the absence of an item, $\neg I_{ij}$
- The problem of the mining association rule with the item-constraint is to discover all the rules that satisfy B and have support and confidence greater than user defined minimum support and minimum confidence.

Association Rules with Item Constraints

- Phase 1
  - Find all frequent itemsets that satisfy the Boolean expression B
  - Two different operation
    - Candidate set generation
    - Counting support : The technique for counting support will remain unchanged
  - Candidate set generation
    - Generate a set of selected items S, such that any itemset that satisfies B will contain at least one selected item
    - Modify the candidate-generation procedure to generate only candidates that contain selected items
    - Discard frequent itemsets that do not satisfy B

Association Rules with Item Constraints

- Phase 2
  - To generate rules from these frequent itemsets, we need to find the support of all subsets of frequent itemsets that do not satisfy B
  - For example
    - To generate a rule ABCD, we need the support of AB to find the confidence of the rule.
    - However, AB may not satisfy B and hence may not have been counted in Phase 1.
    - So we generate all subsets of the frequent itemsets found in phase 1, then make an extra pass over the data set to count the support of those subsets that are not present in the output of phase 1

Association Rules with Item Constraints

- Phase 3
  - To generate the rules from the frequent itemsets found in phase !, using the frequent itemsets found in Phase 1 and 2 to compute confidence