Partition (Savasere, Omiecinski and Navathe, *VLDB'95) 1/3*

- The reason the *DB* needs to be scanned multiple times with Apriori is because the number of possible itemsets to be tested for support is exponential in the number of items, if it must be done in a single scan of the database.
- Idea: If we had a small set of potentially frequent itemsets, we could test their support in a single scan of the database to discover the final real frequent itemsets.
  - ➔partition the database into smaller partitions that fit in main memory

Partition (Savasere, Omiecinski and Navathe, *VLDB'95) 2/3*

- Partition the *DB* into *n* non-overlapping partitions: DB_1, DB_2, …, DB_n
- Apply Apriori in each partition $DB_i \rightarrow$ extract *local frequent itemsets*
  - local absolute *minSupport* threshold in DB_i: relative *minSupport* * $|DB_i|$

Scan 1

- Idea: Any itemset that is potentially frequent in DB must be frequent in at least one of the partitions of DB!
- The set of local frequent itemsets forms the *global candidate itemsets*
  - This is a superset of the actual global frequent itemsets, i.e., it may contain **false positives**.
- Find the actual support of each candidate $\rightarrow$ *global frequent itemsets* **Scan 2**

Partition (Savasere, Omiecinski and Navathe, *VLDB'95) 3/3*

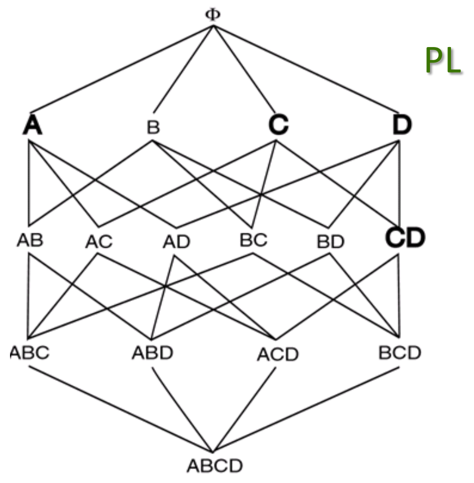
- Pseudocode

```
1.  Divide D into partitions $D^1, D^2, \ldots, D^p$;
2.  For i = 1 to p do
3.      $L^i$ = Apriori($D^i$);               // 1st pass
4.  C = $L^1 \cup \ldots \cup L^p$;
5.  Count C on D to generate L;           // 2nd pass
```

- Advantages:
  - DB partitions to be scanned adapted to fit in main memory size
  - parallel execution of DB scan enabled
  - 2 scans in DB in total to retrieve all frequent itemsets
- Disadvantages:
  - # candidates in 2nd scan might be large

Sampling (Toinoven, VLDB'96) 1/4

- Idea: Pick a random sample, find the frequent itemsets in the sample and verify the results with the rest of the database.



- Select a random sample *S* of *DB* (that fits in memory)
- Apply Apriori to the sample S

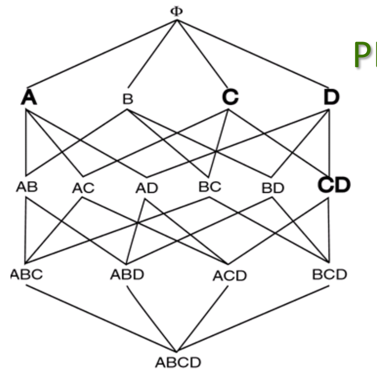
  → *PL (potential frequent itemsets from sample)*

- But, since we search only in *S*, we might miss some global frequent itemsets
- Idea: Expand PL by applying the **negative border**
  - negative border: minimal set of itemsets which are not in *PL*, but whose subsets are all in *PL*.

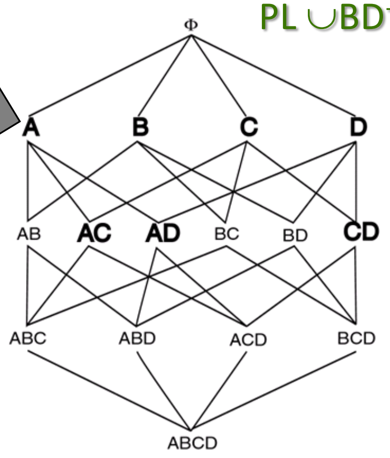
Sampling (Toinoven, VLDB'96) 2/4

- PL from sample
  - Select a random sample *S* of *DB* (that fits in memory)
  - Apply Apriori to the sample → *PL* *(potential frequent itemsets from sample)*
- Expand PL by applying the negative border
  - Candidate set *C = PL* ∪ *BD⁻(PL)*:
    - *BD⁻*(*PL*): negative border (minimal set of itemsets which are not in *PL*, but whose subsets are all in *PL*.)
  - Count the support of *C* itemsets in entire *DB*
  - If there are frequent itemsets in *BD⁻* (by *minSupport*),

    expand *C* by repeatedly applying *BD⁻*
  - Finally, count *C* in *DB*

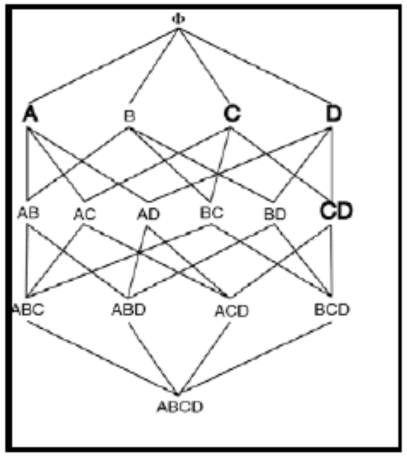


PL

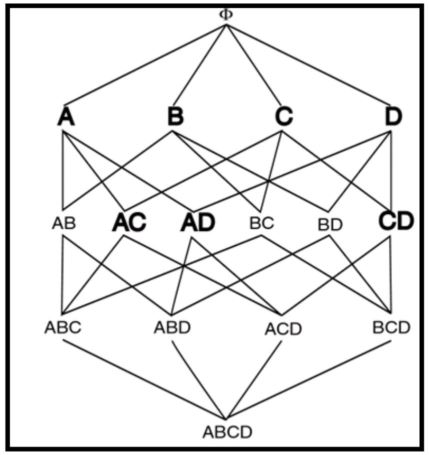


PL ∪BD⁻(PL)

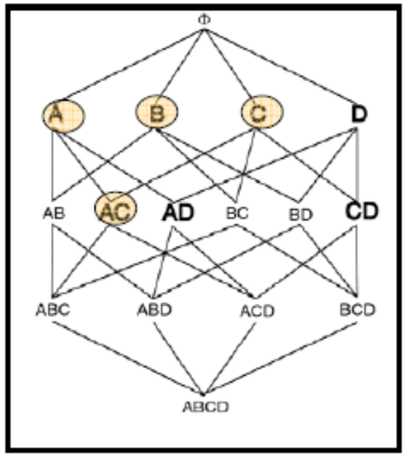
Sampling (Toinoven, VLDB'96) 3/4



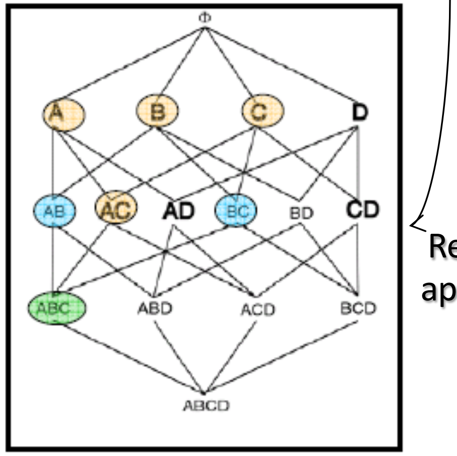
PL



C= PL ∪BD$^-$(PL)



ML:frequent itemsets



Repeated application of BD$^-$

Sampling (Toinoven, VLDB'96) 4/4

- Pseudocode

```
1.   $D_s$ = sample of Database D;
2.   PL = Frequent itemsets in $D_s$ using small $|D_s|$;
3.   C = PL ∪ $BD^-$(PL);
4.   Count C in Database D; (database scan)
5.   ML = Frequent itemsets in C;
6.   repeat
7.      ML = ML ∪ $BD^-$(ML); (repeated application of $BD^-$)
8.   until ML does not grow;
9.   Count sets (ML-C) in Database; (database scan)
```

- Advantages:
  - Scales better than Apriori and Partition.
  - Reduces the number of database scans to 1 in the best case and 2 in worst.
- Disadvantage:
  - Provides approximate result (with error bound guaranty)

Vertical vs horizontal transaction representation

Horizontal layout: (*TID, item set*)

Vertical layout: *(item, TID set)*

Horizontal Data Layout

TID	Items
1	A,B,E
2	B,C,D
3	C,E
4	A,C,D
5	A,B,C,D
6	A,E
7	A,B
8	A,B,C
9	A,C,D
10	B

Vertical Data Layout

A	B	C	D	E
1	1	2	2	1
4	2	3	4	3
5	5	4	5	6
6	7	8	9	
7	8	9		
8	10			
9				

TID-list

Eclat (Zaki, TKDE'00)

- Vertical data layout
- For each itemset *X*, a list of the transaction *ids* that contain *X* is maintained:
  - *X*.tidlist ={t_1, t_4, t_5, t_6, t_7, t_8, t_9}; Y.tidilist={t_1, t_2, t_5, t_7, t_8,t_{10}}
- To find the support of *X Y*, we use their lists intersection:
  - *X*.tidlist $\cap$Y.tidilist={t_1, t_5, t_7, t_8}
  - Support(XY)=| *X*.tidlist $\cap$ Y.tidilist |=4

X
1
4
5
6
7
8
9

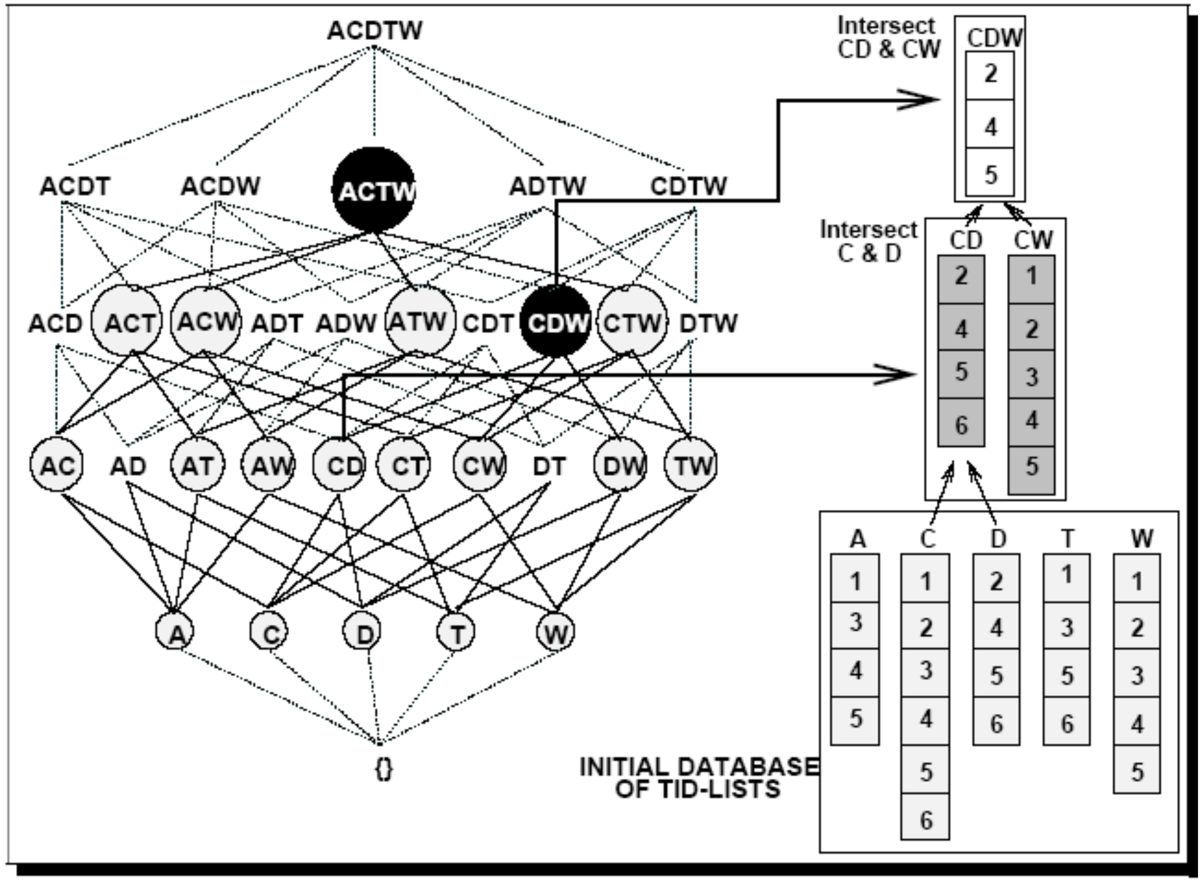
$\wedge$

Y
1
2
5
7
8
10

$\rightarrow$

XY
1
5
7
8

Example ECLAT



Eclat (Zaki, TKDE’00)

- Advantage:
  - No need to access the DB (use instead lists intersection)
  - Very fast support counting (using the lists intersection)
    - As we proceed, the size of the lists decreases, so intersection computation is faster
- Disadvantage:
  - intermediate tid-lists may become too large for memory

Outline

- Apriori improvements
- Closed frequent itemsets (CFI) & Maximal frequent itemsets (MFI)
- Beyond FIM for binary data