ASSOCIATION ANALYSIS

◆ ◆ ◆

FREQUENT ITEMSETS MINING

Alexandre Termier, LIG

M1 MIAGE - Option RIM
2011 S2

Market basket analysis

- Analyse supermarket's transaction data
- Transaction = « market basket » of a customer
- Find which items are often bought together
  - Ex: *{bread, chocolate, butter}*
  - Ex: *{hamburger bread, tomato}* → *{steak}*
- Applications
  - Product placement
  - Cross selling (suggestion of other products)
  - Promotions

Funny example

- Most famous itemset : *{beer, diapers}*
- Found in a chain of American supermarkets
- Further study :
  - Mostly bought on Friday evenings
  - Who ? …

Example

Transactions	Items (products bought)
1	bread, butter, chocolate, vine, pencil
2	bread,butter, chocolate, pencil
3	chocolate
4	butter,chocolate
5	bread, butter, chocolate, vine
6	bread,butter, chocolate

- *{bread, butter, chocolate}* sold together in 4/6 = 66% of transactions
- *{butter, chocolate}* → *{bread}* is true in 4/5 = 80% of cases
- *{chocolate}* → *{bread, butter}* is true in 4/6 = 66% of cases

Definitions

- $A = \{a_1, \dots, a_n\}$: **items**, A : item base
- Any $I \subseteq A$: **itemset**
  - *k*-itemset: itemset with *k* items
- $T = (t_1, \dots, t_m)$, $\forall i\ t_i \subseteq A$: **transaction database**
- $tid(t_j) = j$: transaction index
- **support**(X ∈ I) = number of transactions containing itemset X
- **tidlist**(X ∈ I) = list of tids of transactions containing itemset X

An itemset X is **frequent** if *support(X)* ≥ *minsup*

- Confidence of association rule $X \rightarrow Y$: $c = \dfrac{\text{support}(X \cup Y)}{\text{support}(X)}$

  $(X \cap Y = \varnothing)$

An association rule with confidence *c* **holds** if *c* ≥ *minconf*

Example, rewritten

Transactions	Items (products bought)
1	bread, butter, chocolate, vine, pencil
2	bread,butter, chocolate, pencil
3	chocolate
4	butter,chocolate
5	bread, butter, chocolate, vine
6	bread,butter, chocolate

- *{bread, butter, chocolate}*
  support = *(absolute)*
  = *(relative)*
- *{chocolate} → {bread, butter}*
  confidence =
- *{butter, chocolate} → {bread}*
  confidence =

Computing association rules

- Two steps:
  1. Compute frequent itemsets
     - Discover itemsets with support ≥ *minsup*
     - **Very expensive computationally !**

  2. Compute which association rules hold
     - Partition each itemset and discover rules with confidence ≥ *minconf*
     - Much faster than discovering itemsets

How to compute frequent itemsets ?

- Brute force approach
  - **Generate and Test** method
  - Generate all possible itemsets randomly
  - Compute their support
  - But highly combinatorial problem :
    - How many possible itemsets for 1000 items ?
  - *Infeasible in practice*

The Apriori algorithm

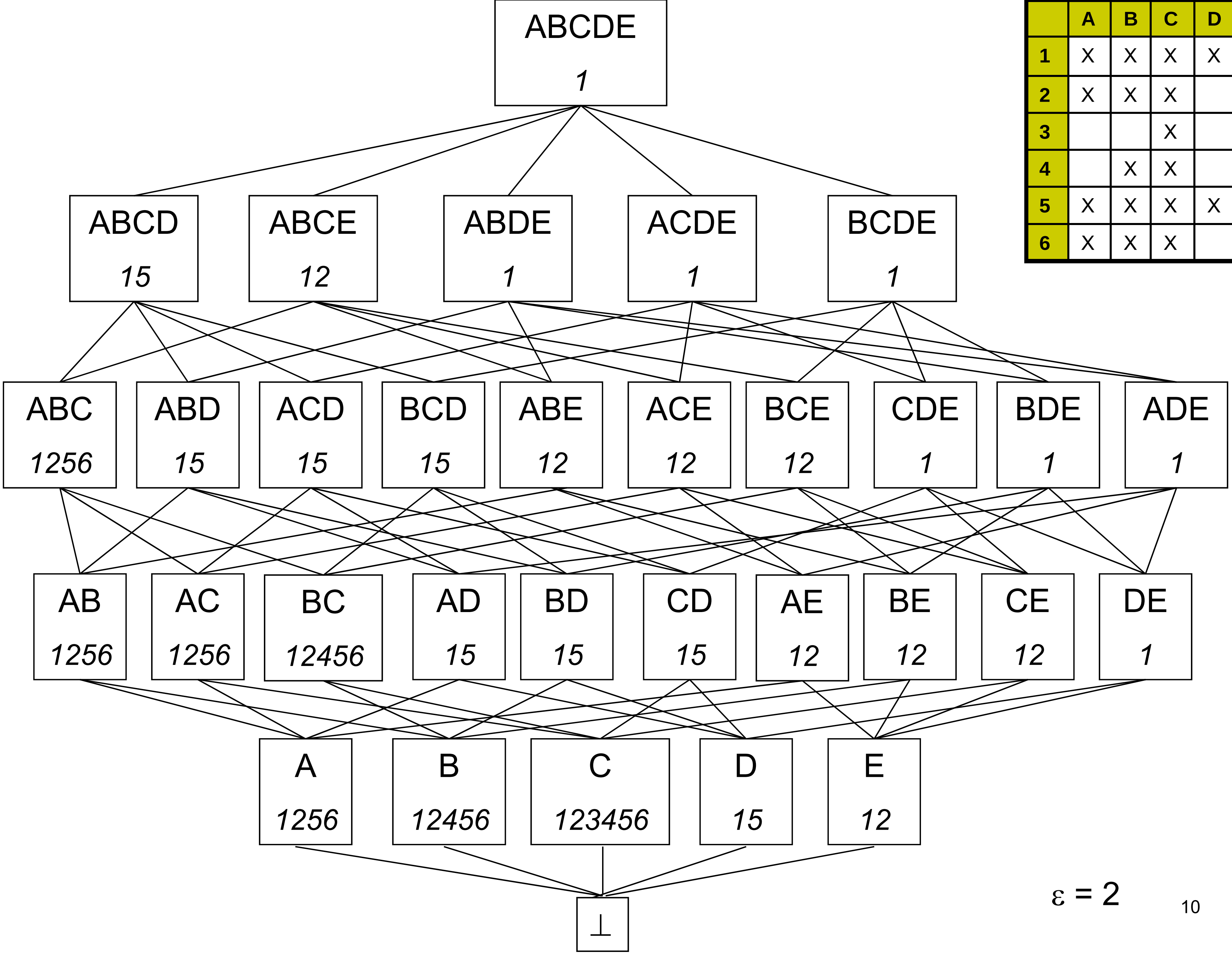
[Agrawal *et al.*, 93]

- Levelwise search
  - Discover frequent 1-itemsets, 2-itemsets,…

Apriori property *:*

If an itemset is not frequent, then all its supersets are not frequent

- Ex: If *{vine, pencil}* is not frequent, then of course *{vine, pencil, chocolate}* will not be frequent
- *Downward closure property*
- *Anti-monotonicity property*



ABCDE *1*

ABCD *15* | ABCE *12* | ABDE *1* | ACDE *1* | BCDE *1*

ABC *1256* | ABD *15* | ACD *15* | BCD *15* | ABE *12* | ACE *12* | BCE *12* | CDE *1* | BDE *1* | ADE *1*

AB *1256* | AC *1256* | BC *12456* | AD *15* | BD *15* | CD *15* | AE *12* | BE *12* | CE *12* | DE *1*

A *1256* | B *12456* | C *123456* | D *15* | E *12*

⊥

	A	B	C	D	E
1	X	X	X	X	X
2	X	X	X		X
3			X		
4		X	X		
5	X	X	X	X	
6	X	X	X		

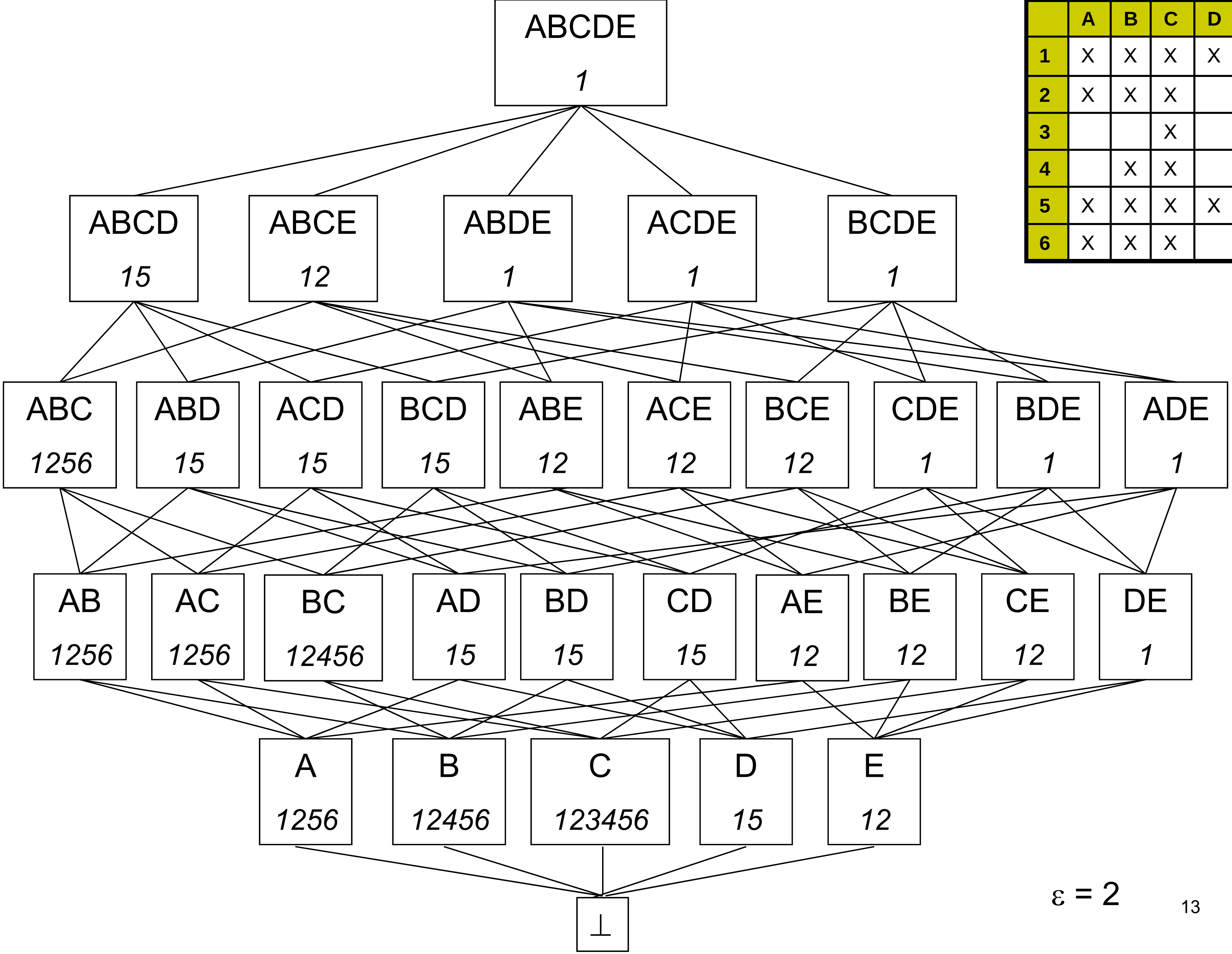
$\varepsilon = 2$

Apriori algorithm

```
Input: T, minsup

$F_1$ = {Frequent 1-itemsets} ;
for (k=2 ; $F_{k-1} \neq \emptyset$ ; k++) do begin
   $C_k$ = apriori-gen($F_{k-1}$) ; // Candidates generation
   foreach transaction t $\in$ T do begin
      $C_t$ = subset($C_k$, t) ; // Support counting
      foreach candidate c $\in$ $C_t$ do
         c.count++ ;
   end
   $F_k$ = { c $\in$ $C_k$ | c.count $\geq$ minsup } ;
end
return $\cup_k F_k$ ;
```

Candidate generation

- `apriori-gen`: generates candidates *k*-itemsets from frequent *(k-1)*-itemsets
- c (size k) = merge of $p, q \in F_{k-1}$ (both have size k-1)
- How many combinations of such p,q to build c ?


ABCDE 1

ABCD 15
ABCE 12
ABDE 1
ACDE 1
BCDE 1

ABC 1256
ABD 15
ACD 15
BCD 15
ABE 12
ACE 12
BCE 12
CDE 1
BDE 1
ADE 1

AB 1256
AC 1256
BC 12456
AD 15
BD 15
CD 15
AE 12
BE 12
CE 12
DE 1

A 1256
B 12456
C 123456
D 15
E 12

⊥

	A	B	C	D	E
1	X	X	X	X	X
2	X	X	X		X
3			X		
4		X	X		
5	X	X	X	X	
6	X	X	X		

$\varepsilon = 2$

apriori-gen

```
Input: $F_{k-1}$

// Join step
insert into $C_k$
select p.$item_1$,p.$item_2$,…,p.$item_{k-1}$,q.$item_{k-1}$
from p,q $\in$ $F_{k-1}$
where p.$item_1$ = q.$item_1$,…,p.$item_{k-2}$=q.$item_{k-2}$,
                            p.$item_{k-1}$<q.$item_{k-1}$
// Prune step
foreach itemset c $\in$ $C_k$ do
  foreach (k-1)-subset s of c do
      if (s $\notin$ $F_{k-1}$) then
         delete c from $C_k$ ;

return $C_k$
```

Here use of anti-monotony property !

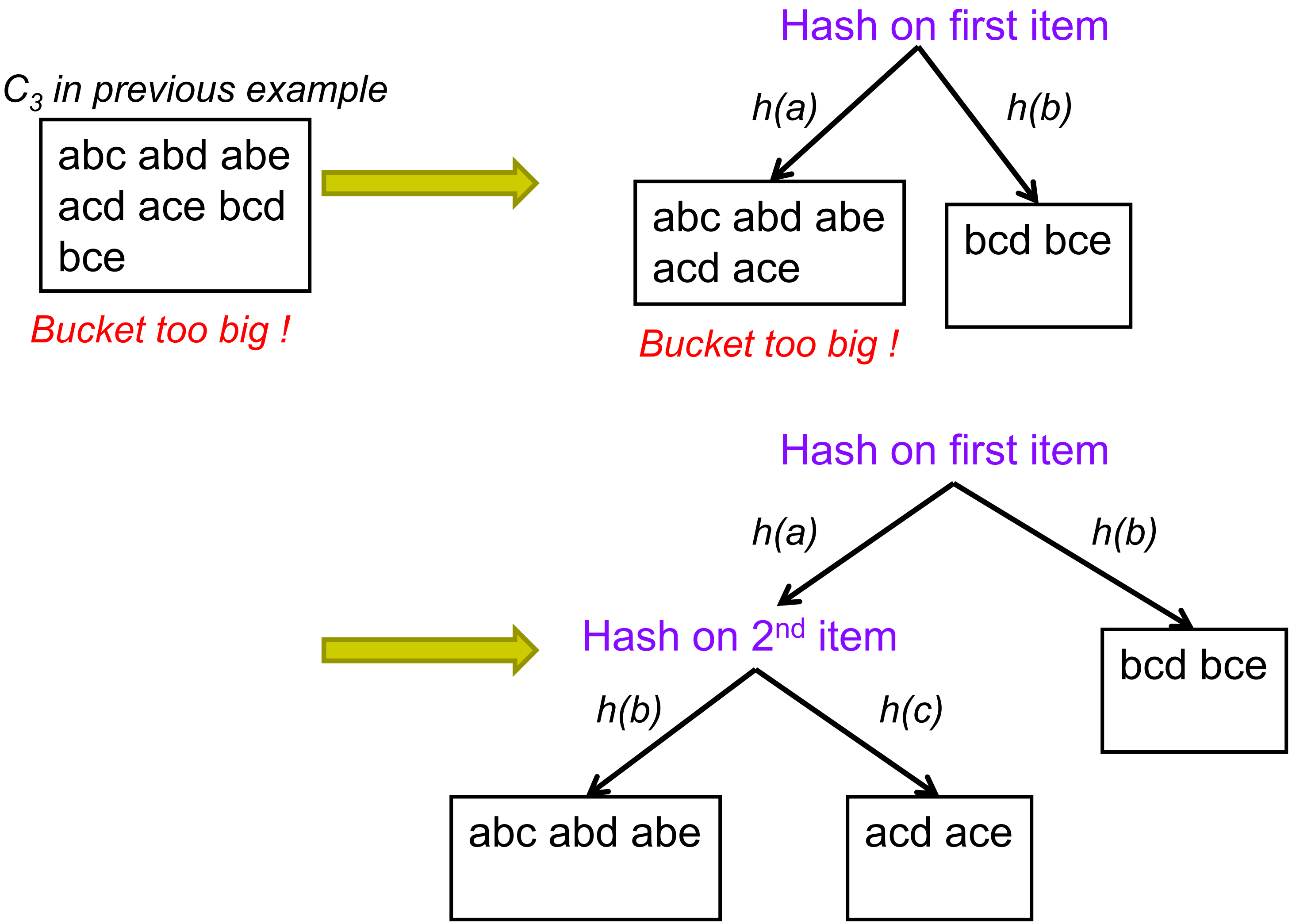
Subset

- For each transaction *t*, find all the itemsets of C_k that are in *t*
- Brute force :

  -
    ```
    foreach $t \in T$
       foreach $c \in C_k$
            compute if $c \subseteq t$
    ```
- Too much computation !
- The Apriori solution:
  - Partition candidates into different buckets of limited size
  - Store buckets in leaves of a hash tree
  - Find candidates subset of a transaction by traversing hash tree

Hash tree construction

Max bucket size = 3



C_3 in previous example

abc abd abe acd ace bcd bce

Bucket too big !

Hash on first item

h(a)

abc abd abe acd ace

Bucket too big !

h(b)

bcd bce

Hash on first item

h(a)

Hash on 2nd item

h(b)

abc abd abe

h(c)

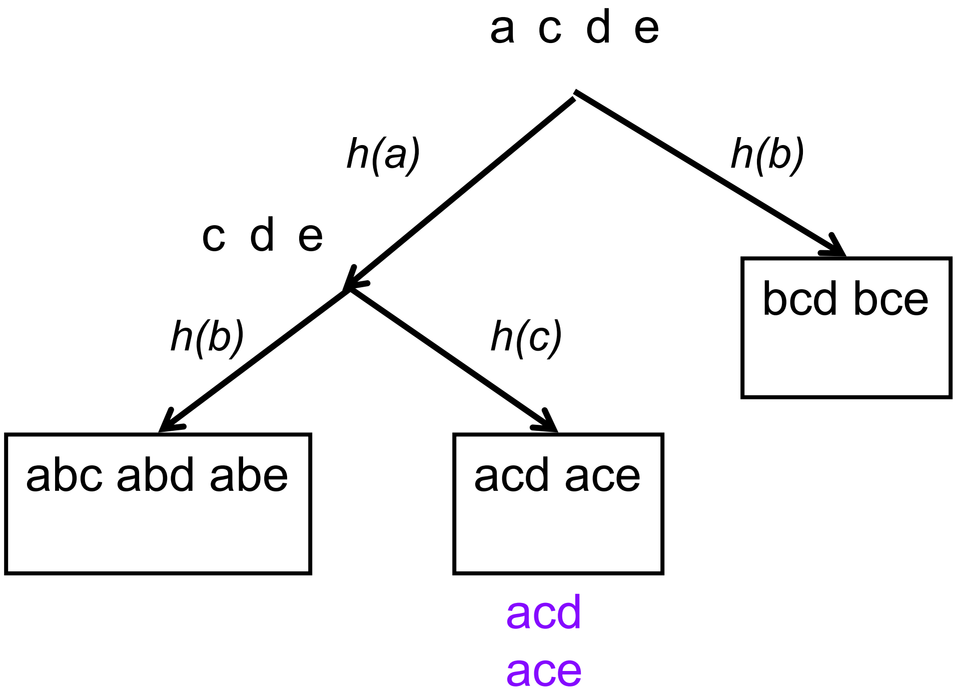
acd ace

h(b)

bcd bce

Hash tree utilisation for subset

Transaction **2'** : a c d e



Complexity

- apriori_gen, step k
  - Dominated by prune step ≈ $O(k.|C_k|)$
- support counting
  - ≈ $O(m.|C_k|)$ with $m = |T|$ (database size)
- → one iteration is ≈ $O(m.|C_k|)$
- Total complexity :
  - ≈ $O(m.\Sigma_k|C_k|)$
  - Worst case : candidates are all possible itemsets
    - ≈ $O(m.2^n)$ with n = number of items
- ⇒ Linear in database size
- ⇒ Exponential in number of items
- Influence of transaction width (database density) on number of traversal of hash tree

Association rules computation

- Once we have the frequent itemsets, we want the association rules.
- Reminder: we are only interested in rules that have a high confidence value

Confidence of X → Y: $c = \dfrac{\text{support}(X \cup Y)}{\text{support}(X)}$

- Let *F* be an itemset, with |*F*| = *k*.
  How many possible rules ?
- What is a naive solution to compute them ?
- Is it efficient ?

Monotony of confidence ?

Transactions	Items (products bought)
1	bread, butter, chocolate, vine, pencil
2	bread,butter, chocolate, pencil
3	chocolate
4	butter,chocolate
5	bread, butter, chocolate, vine
6	bread,butter, chocolate
7	bu
8	bu
9	bu

- *{chocolate} → {bread, butter}* confidence = 4/6 = 66%

CONFIDENCE IS NOT MONOTONE / ANTI-MONOTONE

More on monotony of confidence

- For rules coming from the same itemset, confidence is anti-monotone
  - e.g., L = {A,B,C,D}:

$$c(ABC \rightarrow D) \geq c(AB \rightarrow CD) \geq c(A \rightarrow BCD)$$

    - Confidence is anti-monotone w.r.t. number of items on the RHS of the rule

- → some pruning is possible

Association rule generation algorithm

```
Input: T, minsup, minconf, $F_{all}$ = union of $F_1..F_n$

$H_1 = \emptyset$
foreach $f_k \in F_{all}$, $k \geq 2$ do begin
   A = (k-1)-itemsets $a_{k-1}$ such that $a_{k-1} \subset f_k$ ;
   foreach $a_{k-1} \in A$ do begin
       conf = support($f_k$)/support($a_{k-1}$) ;
       if conf $\geq$ minconf do begin
           output rule $a_{k-1} \rightarrow (f_k - a_{k-1})$ ;
           add $(f_k - a_{k-1})$ to $H_1$ ;
       end
   end
   ap-genrules($f_k$, $H_1$) ;
end
```

ap-genrules

```
Input: $f_k$, $H_m$ : set of m-item consequents

if (k>m+1) then begin
   $H_{m+1}$ = apriori-gen($H_m$) ; // Generate all possible m+1
                                    itemsets
   foreach $h_{m+1} \in H_{m+1}$ do begin
        conf = support($f_k$)/support($f_k$-$h_{m+1}$) ;
        if conf $\geq$ minconf then
            output rule $f_k$ - $h_{m+1} \rightarrow h_{m+1}$ ;
         else
            delete $h_{m+1}$ from $H_{m+1}$ ;   } Pruning by anti-monotony
   end
   ap-genrules($f_k$, $H_{m+1}$) ;
end
```

First improvements of Apriori

- End of 90’s :
  - Main memory: 64-256 MB
  - Databases: can go over 1 GB
  - Apriori : several passes over database…
  - ⇒ need algorithms that can handle database in memory
- Partition [Savasere et al. 1995]
  - Cut the database in pieces fitting into memory, compute results for each piece and join them
- Sampling [Toivonen 1996]
  - Compute frequent itemsets on a sample of the database
- DIC [Brin et al. 1997]
  - Improves number of passes on database

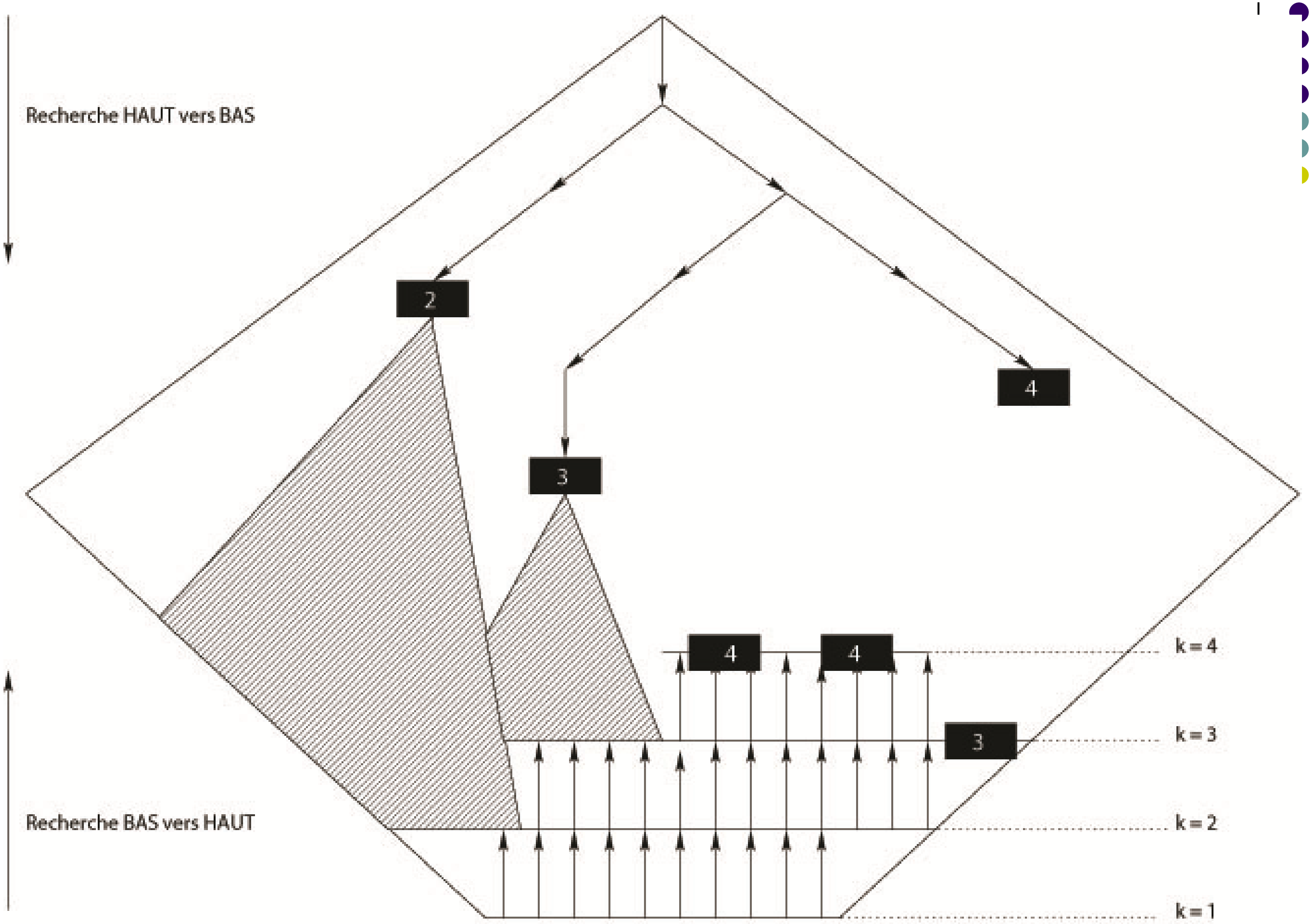
Maximal frequent itemsets

Set of maximal frequent itemsets :

$MFI = \{ I \in FI \mid \forall I' \supset I \;\; I' \notin FI \}$

with FI set of frequent itemsets

- Several orders of magnitudes less MFI than FI
- Can be searched both **bottom-up** and **top-down**
- Pincer-Search [Lin & Kedem 1998]
  Max-Miner [Bayardo et al. 1998]
- BUT loss of information



k itemset fréquent maximal + numéro de l'itération où il est trouvé

▨ partie de l'espace de recherche qu'il n'est pas nécessaire de considerer

The Eclat algorithm

[Zaki *et al.*, 97]

- Apriori : DB is in **horizontal** format
- Eclat introduces the **vertical** format
  - Itemset x → tid-list(x)

	A	B	C	D	E
1	X	X	X	X	X
2	X	X	X		X
3			X		
4		X	X		
5	X	X	X	X	
6	X	X	X		

Horizontal format

A	B	C	D	E
1	1	1	1	1
2	2	2	5	2
5	4	3		
6	5	4		
	6	5		
		6		

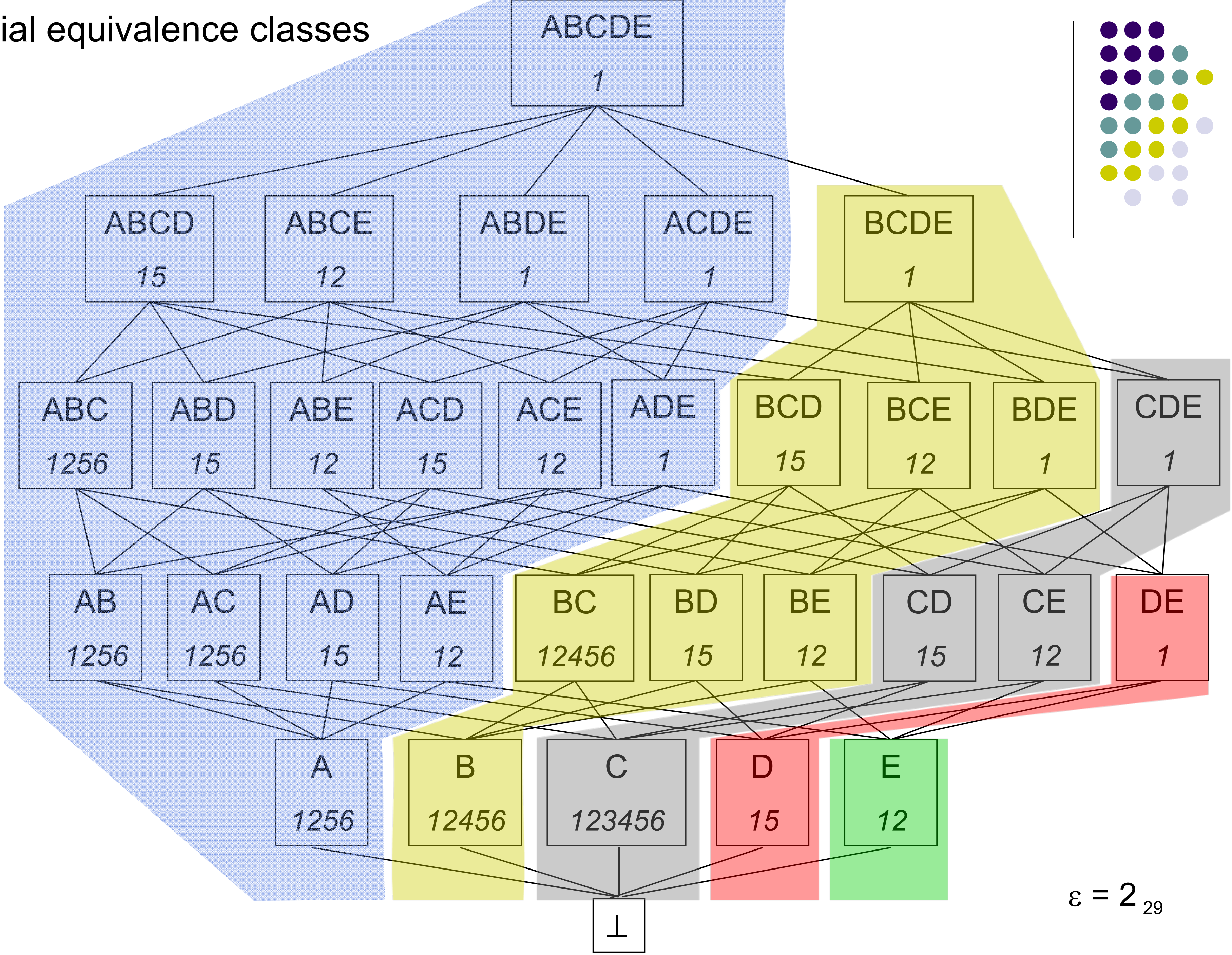
Vertical format

Vertical format

- Support counting can be done with tid-list intersections
  - $\forall I,J$ itemsets : $tidlist(I \cup J) = tidlist(I) \cap tidlist(J)$
  - No need for costly subset tests, hash tree generation…

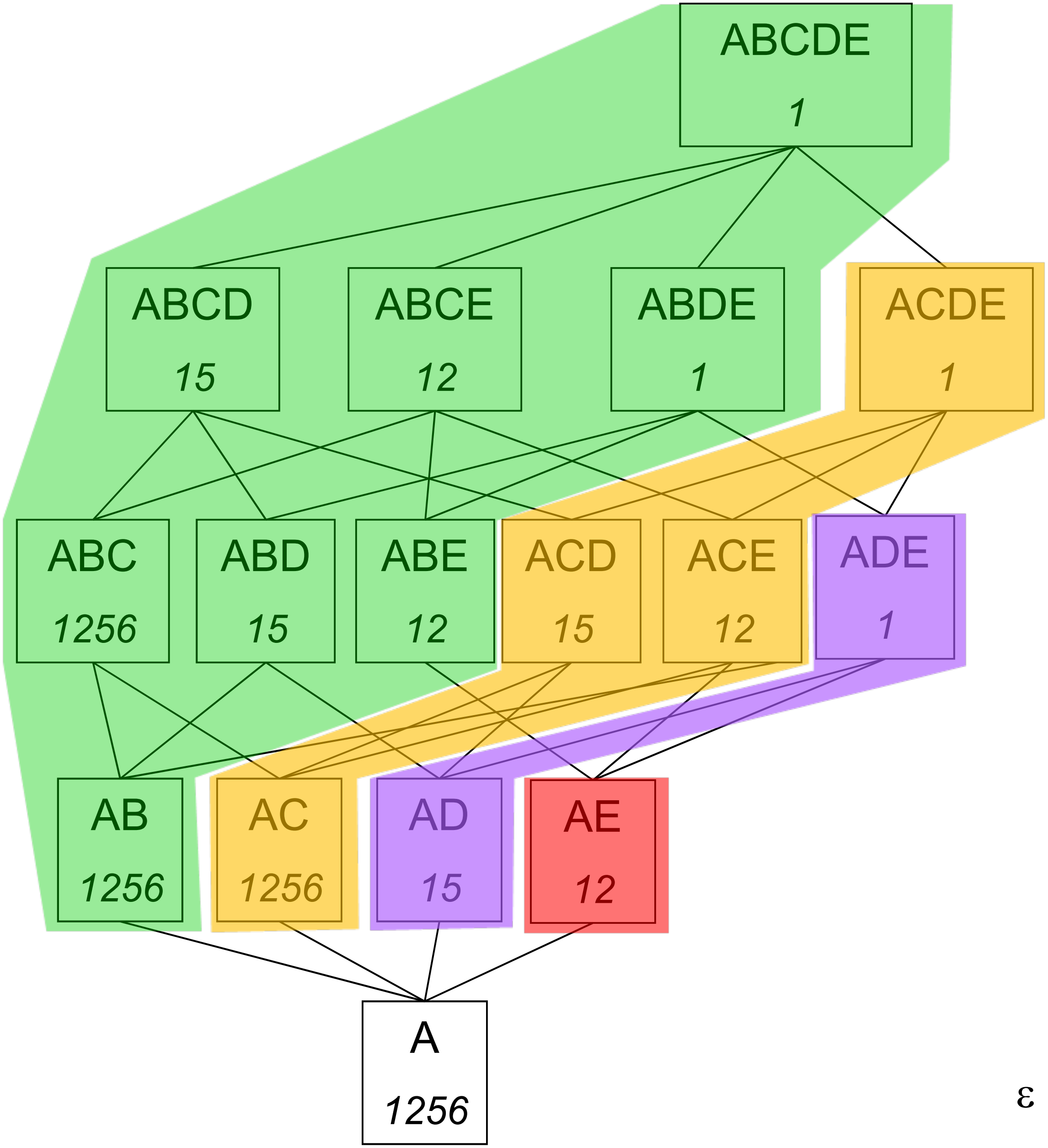
- Problem
  - If database is big, tidlists of the many candidates created will be big also, and will not hold in memory
- Solution
  - Partition the lattice into equivalence classes
  - In Eclat : equivalence relation = **sharing the same prefix**

Initial equivalence classes



$\varepsilon = 2$

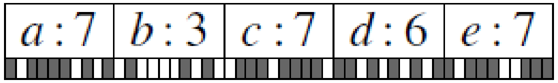
Equivalence classes inside [A] class



ε = 2

Eclat: Depth-First Search

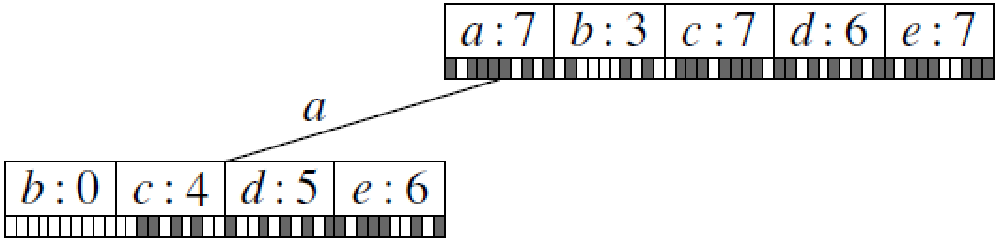
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Form a transaction list for each item. Here: bit vector representation.
  - grey: item is contained in transaction
  - white: item is not contained in transaction
- Transaction database is needed only once (for the single item transaction lists).

Eclat: Depth-First Search

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

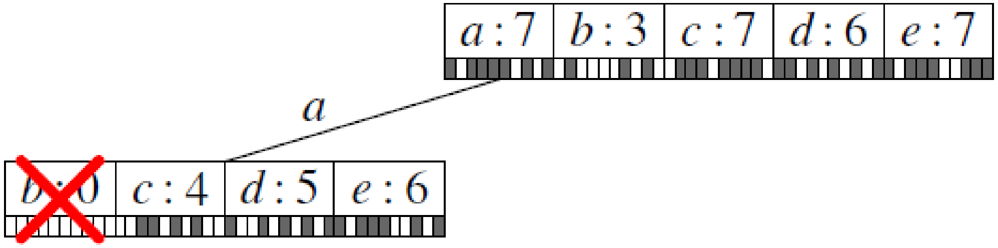


- Intersect the transaction list for item a with the transaction lists of all other items (*conditional database* for item a).

- Count the number of bits that are set (number of containing transactions). This yields the support of all item sets with the prefix a.

Eclat: Depth-First Search

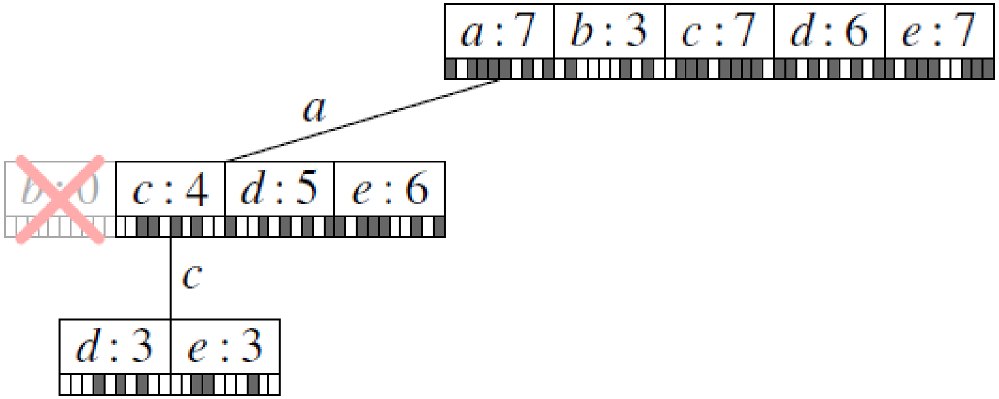
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The item set $\{a, b\}$ is infrequent and can be pruned.
- All other item sets with the prefix a are frequent and are therefore kept and processed recursively.

Eclat: Depth-First Search

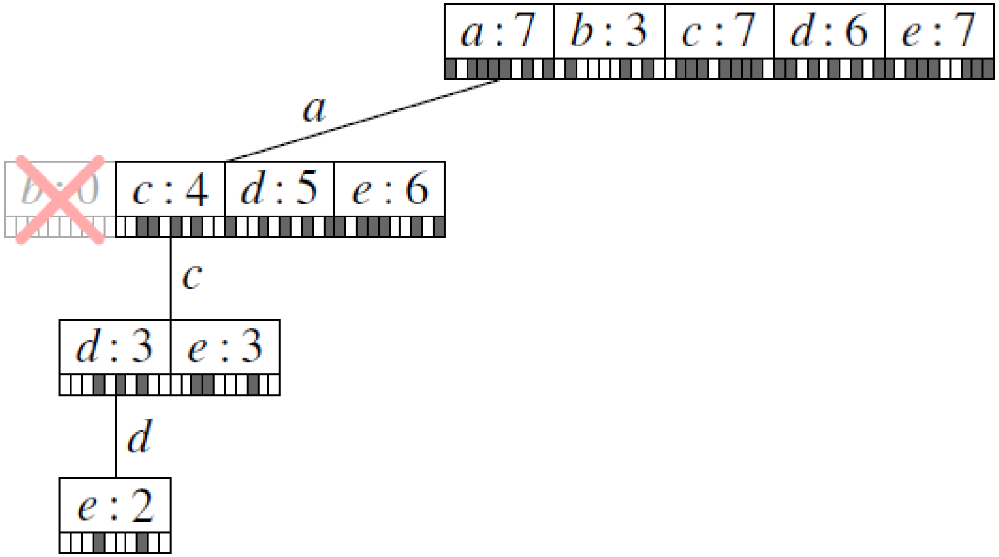
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Intersect the transaction list for the item set $\{a, c\}$ with the transaction lists of the item sets $\{a, x\}$, $x \in \{d, e\}$.
- Result: Transaction lists for the item sets $\{a, c, d\}$ and $\{a, c, e\}$.
- Count the number of bits that are set (number of containing transactions). This yields the support of all item sets with the prefix *ac*.

Eclat: Depth-First Search

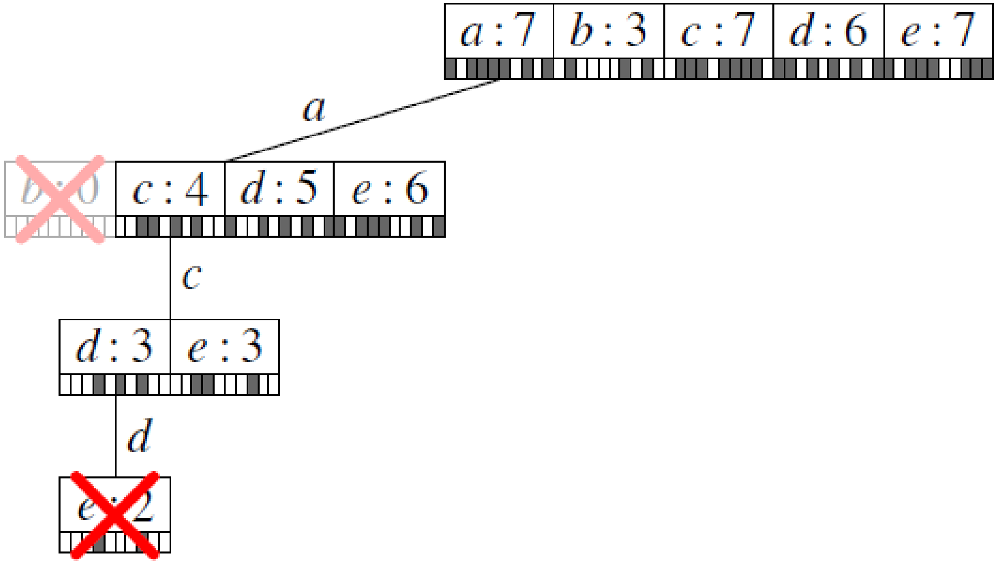
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Intersect the transaction lists for the item sets $\{a, c, d\}$ and $\{a, c, e\}$.
- Result: Transaction list for the item set $\{a, c, d, e\}$.
- With Apriori this item set could be pruned before counting, because it was known that $\{c, d, e\}$ is infrequent.

Eclat: Depth-First Search

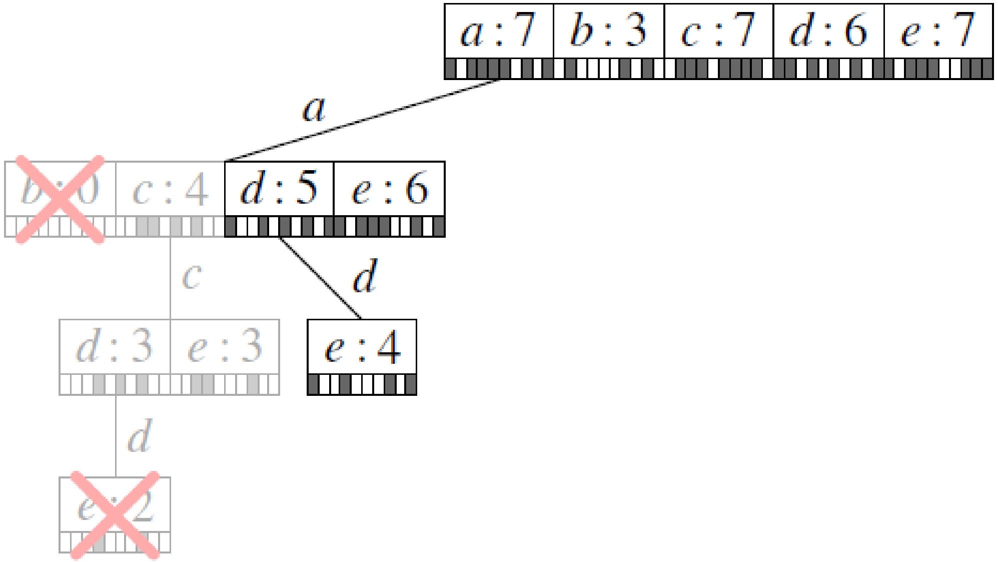
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The item set $\{a, c, d, e\}$ is not frequent (support 2/20%) and therefore pruned.
- Since there is no transaction list left (and thus no intersection possible), the recursion is terminated and the search backtracks.

Eclat: Depth-First Search

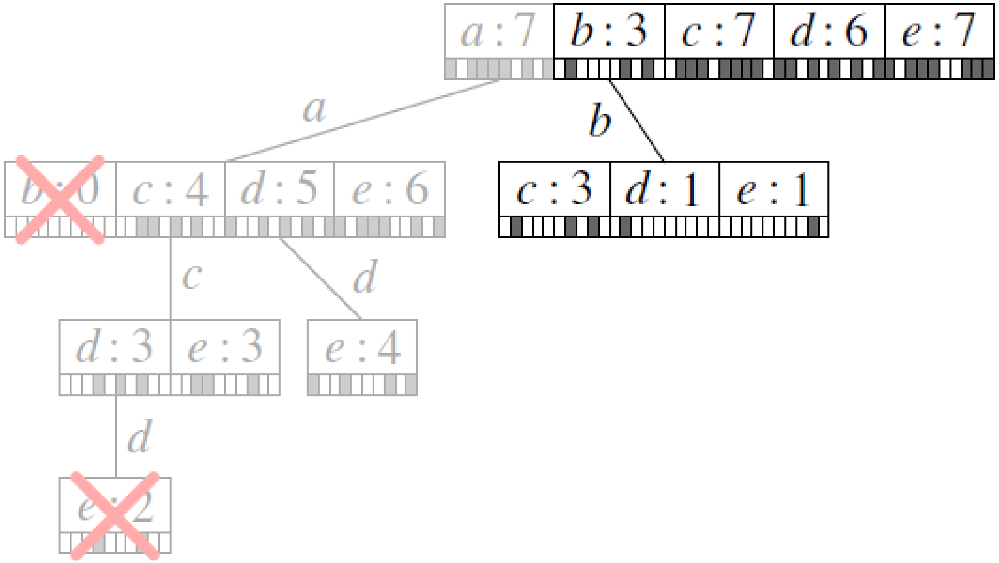
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The search backtracks to the second level of the search tree and intersect the transaction list for the item sets $\{a, d\}$ and $\{a, e\}$.
- Result: Transaction list for the item set $\{a, d, e\}$.
- Since there is only one transaction list left (and thus no intersection possible), the recursion is terminated and the search backtracks again.

Eclat: Depth-First Search

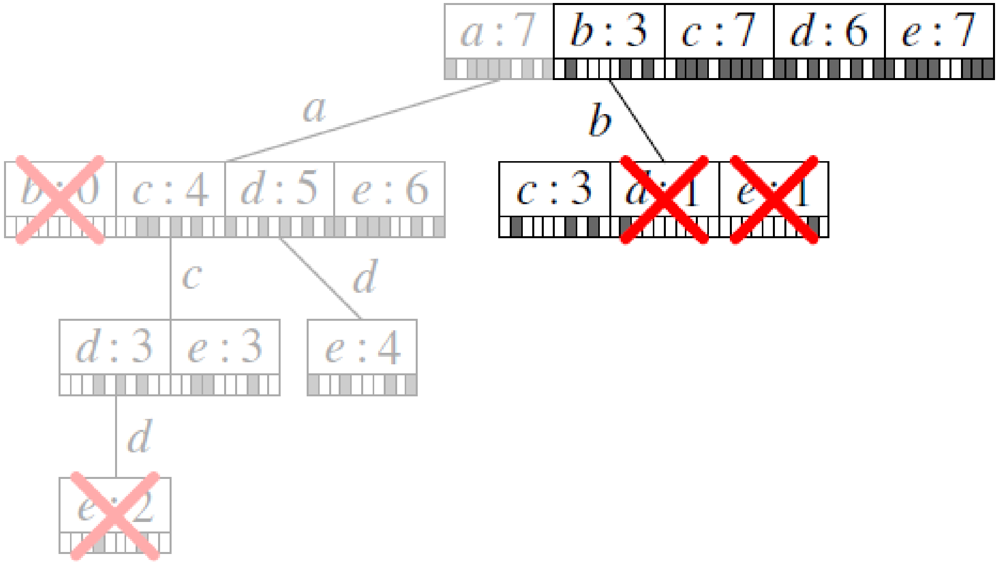
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The search backtracks to the first level of the search tree and intersect the transaction list for b with the transaction lists for c, d, and e.
- Result: Transaction lists for the item sets $\{b, c\}$, $\{b, d\}$, and $\{b, e\}$.

Eclat: Depth-First Search

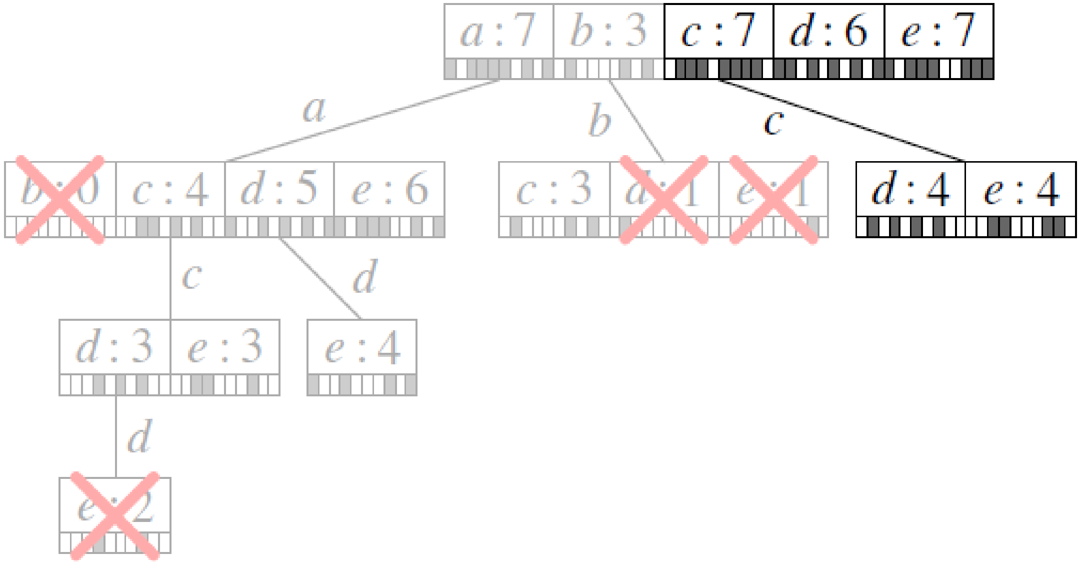
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Only one item set has sufficient support → prune all subtrees.
- Since there is only one transaction list left (and thus no intersection possible), the recursion is terminated and the search backtracks again.

Eclat: Depth-First Search

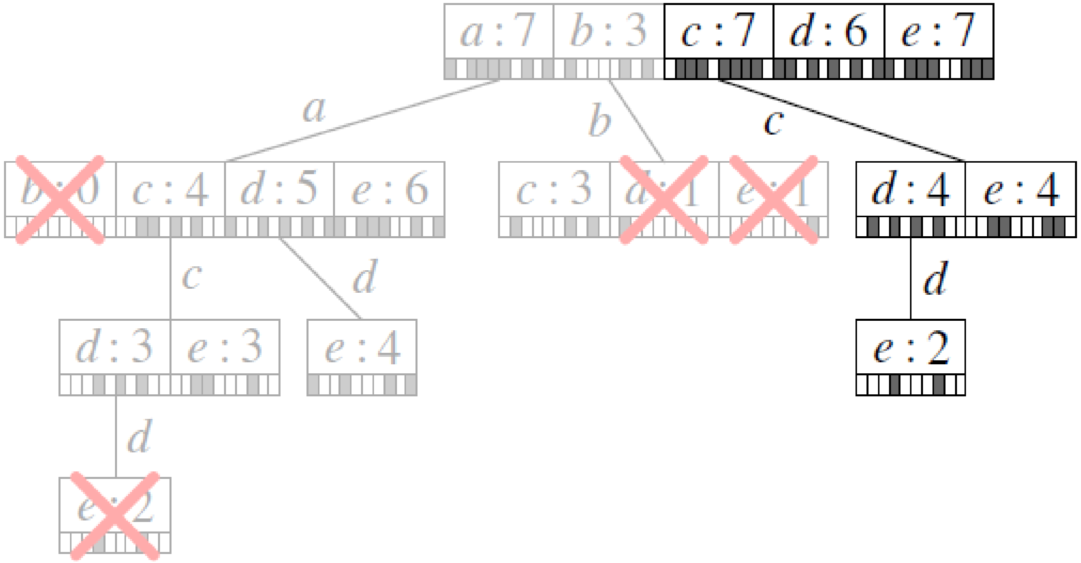
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Backtrack to the first level of the search tree and intersect the transaction list for c with the transaction lists for d and e.
- Result: Transaction lists for the item sets $\{c, d\}$ and $\{c, e\}$.

Eclat: Depth-First Search

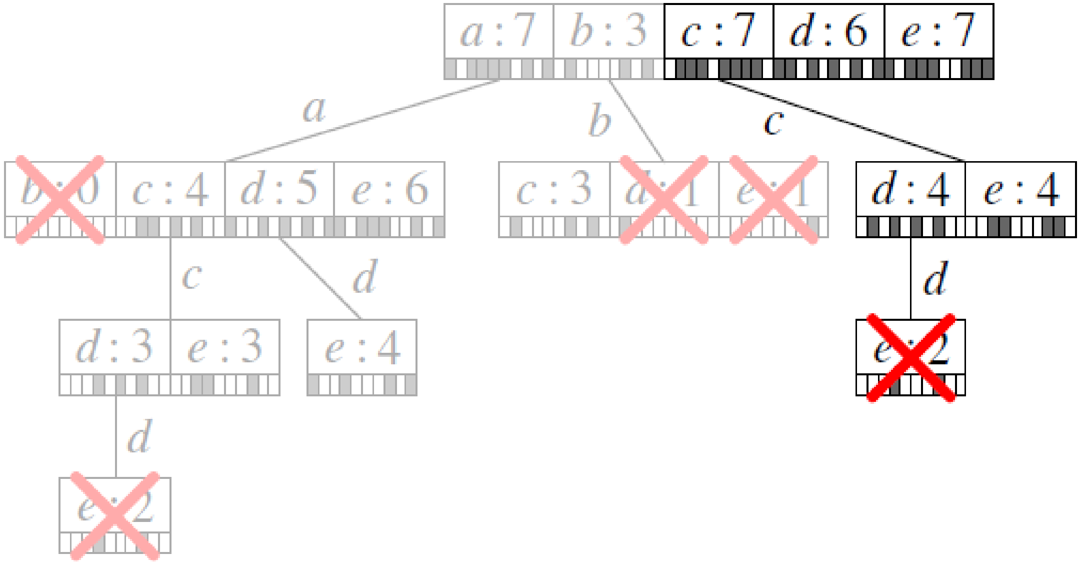
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Intersect the transaction list for the item sets $\{c, d\}$ and $\{c, e\}$.
- Result: Transaction list for the item set $\{c, d, e\}$.

Eclat: Depth-First Search

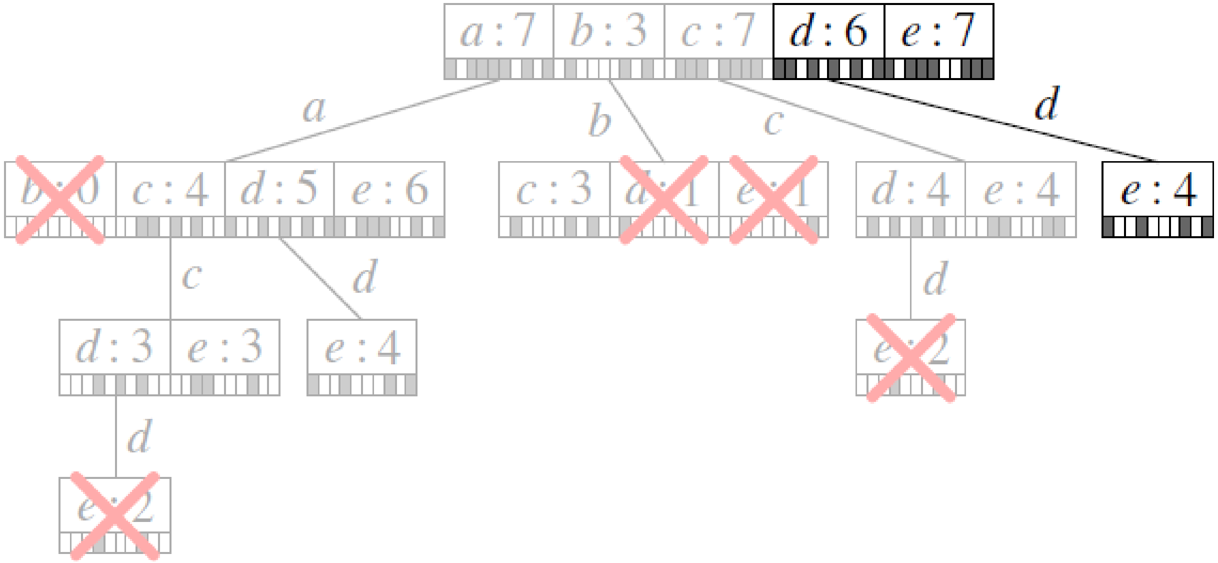
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The item set $\{c, d, e\}$ is not frequent (support 2/20%) and therefore pruned.
- Since there is no transaction list left (and thus no intersection possible), the recursion is terminated and the search backtracks.

Eclat: Depth-First Search

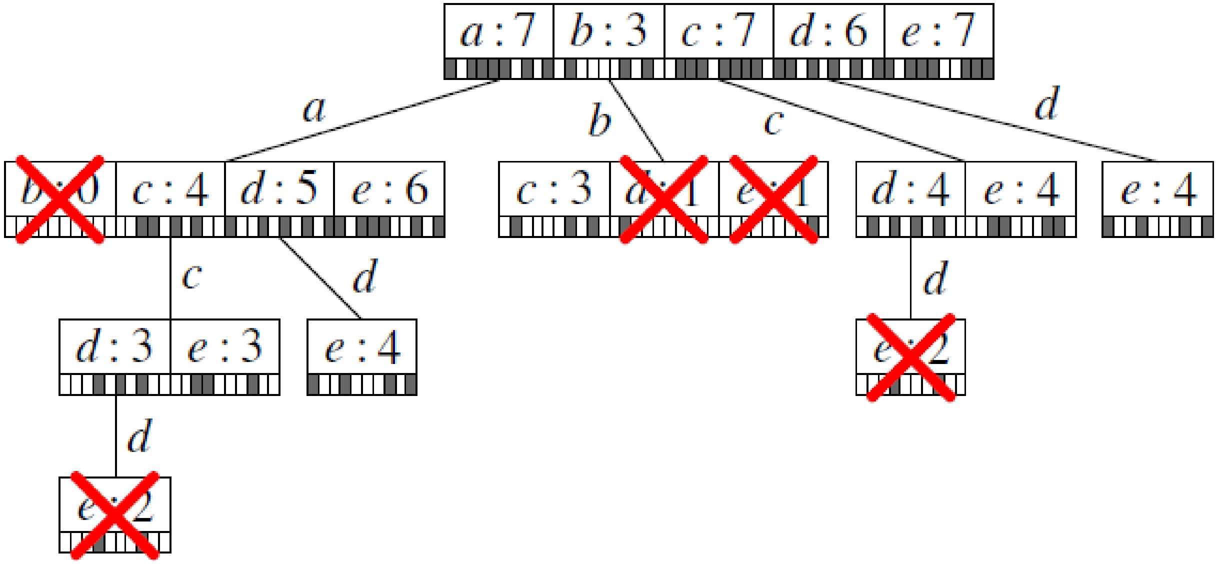
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The search backtracks to the first level of the search tree and intersect the transaction list for d with the transaction list for e.
- Result: Transaction list for the item set $\{d, e\}$.
- With this step the search is finished.

Eclat: Depth-First Search

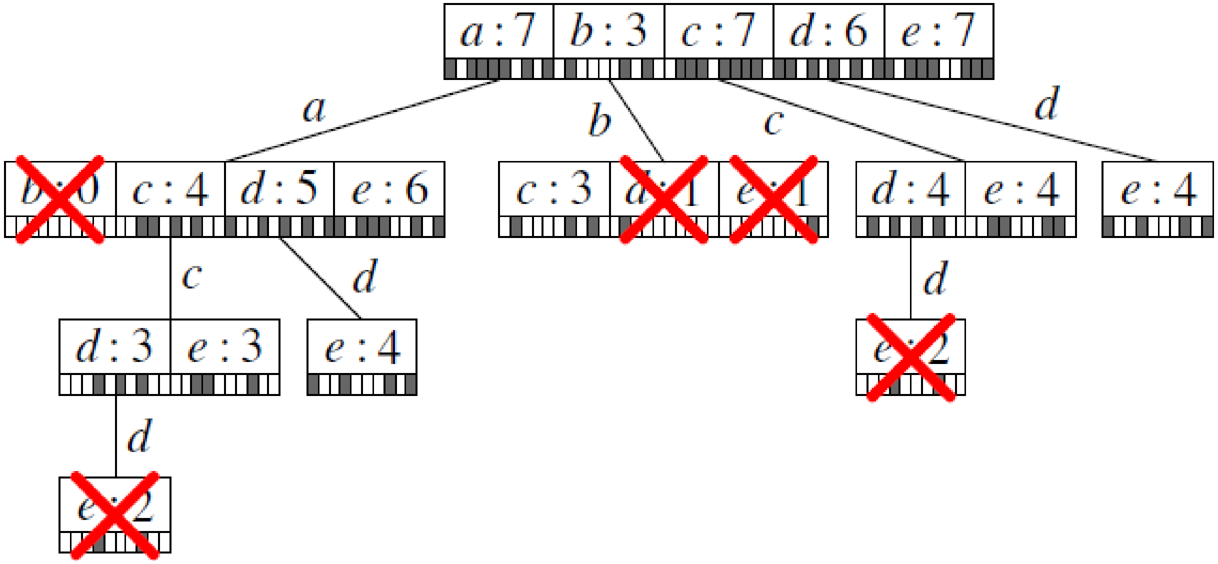
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The found frequent item sets coincide, of course, with those found by the Apriori algorithm.
- However, a fundamental difference is that Eclat usually only writes found frequent item sets to an output file, while Apriori keeps the whole search tree in main memory.

Eclat: Depth-First Search

1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Note that the item set $\{a, c, d, e\}$ could be pruned by Apriori without computing its support, because the item set $\{c, d, e\}$ is infrequent.
- The same can be achieved with Eclat if the depth-first traversal of the prefix tree is carried out from right to left *and* computed support values are stored.
  It is debatable whether the expected gains justify the memory requirement.

Eclat algorithm

```
Input: T, minsup

compute $L_1$ and $L_2$ // like apriori
Transform T in vertical representation
$CE_2$ = Decompose $L_2$ in equivalence classes
forall $E_2 \in CE_2$ do
   compute_frequent($E_2$)
end forall

return $\cup_k F_k$ ;
```

compute_frequent(E_{k-1})

```
forall itemsets $I_1$ and $I_2$ in $E_{k-1}$ do
   if $|tidlist(I_1) \cap tidlist(I_2)| \geq minsup$ then
      $L_k \leftarrow L_k \cup \{I_1 \cup I_2\}$
   end if
end forall

$CE_k$ = Decompose $L_k$ in equivalence classes
forall $E_k \in CE_k$ do
   compute_frequent($E_k$)
end forall
```

The FP-growth approach

- FP-Growth : Frequent Pattern Growth
- No candidate generation
- Compress transaction database into FP-tree (Frequent Pattern Tree)
  - Extended prefix-tree
- Recursive processing of *conditional databases*
- Can be one order of magnitude faster than Apriori

FP-tree

- Compact structure for representing DB and frequent itemsets

1. Composed of :
    - root
    - item-prefix subtrees
    - frequent-item-header array
2. Node =
    - item-name
    - count *// number of transactions containing path reaching this node*
    - node-link *// next node having same item-name*
3. Entry in frequent-item-header array =
    - item-name
    - head of node-link *// pointer to first node having item-name*

- Both an **horizontal** (prefix-tree) and a **vertical** (node links) structure

FP-tree example (1/2)

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

Original transaction database

C : 6
B : 5
A : 4
D : 2
E : 2
F : 1

Items ordered by frequency

C B A D E
C B A E
C
C B
C B A D
C B A

*Transactions reordered by item frequency (infrequent item **F** pruned)*

C
C B
C B A
C B A D
C B A D E
C B A E

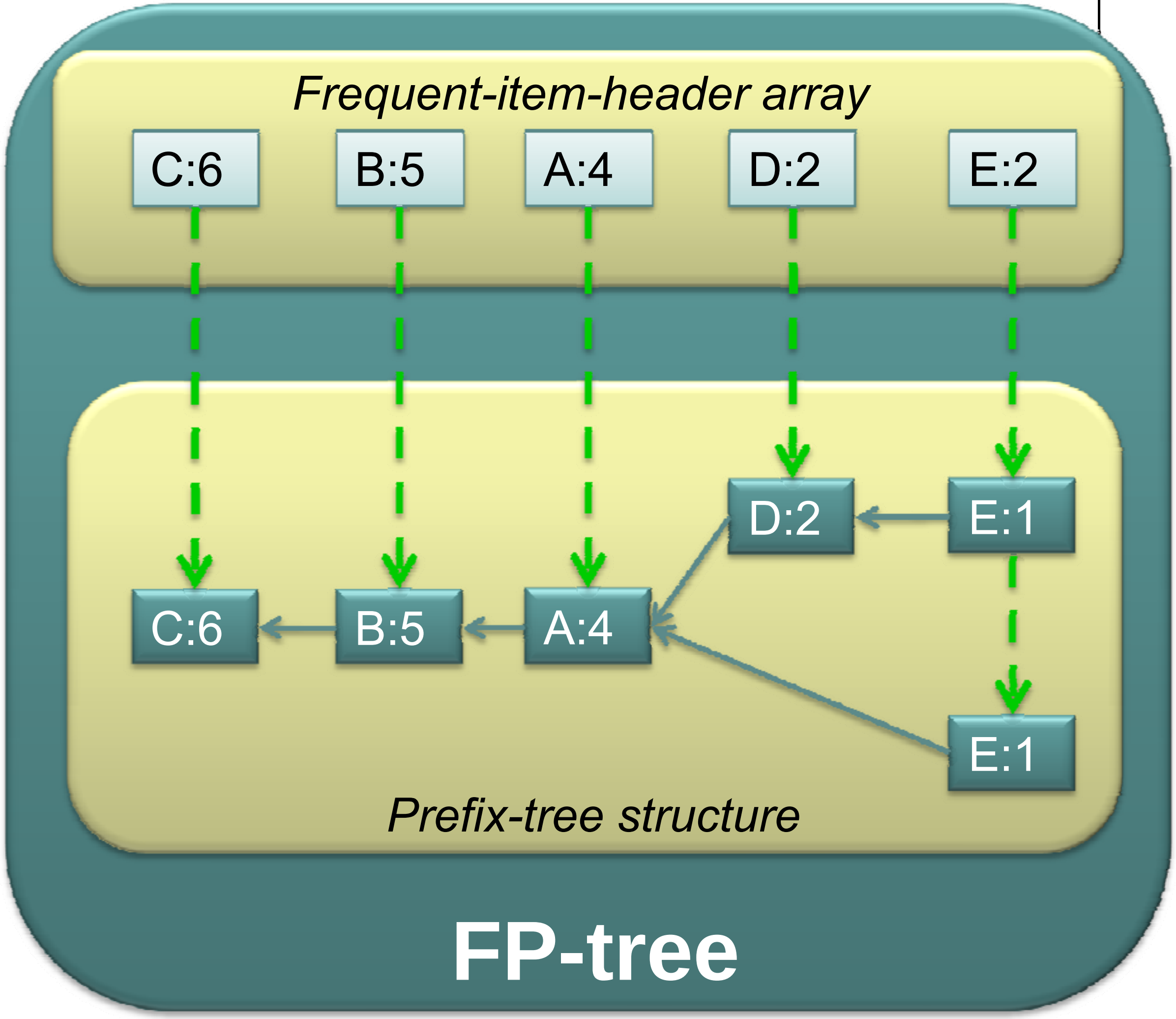
Transactions sorted lexicographically

minsup = 2

FP-tree example (2/2)

C
C B
C B A
C B A D
C B A D E
C B A E

Transactions sorted lexicographically



Exercise

- Draw the FP-tree for the following DB : *(minsup = 3)*

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

FP-Growth: Preprocessing the Transaction Database

①	②	③	④	⑤
a d f	*d*: 8	*d a*	*d b*	
a c d e	*b*: 7	*d c a e*	*d b c*	
b d	*c*: 5	*d b*	*d b a*	FP-tree (see next slide)
b c d	*a*: 4	*d b c*	*d b a*	
b c	*e*: 3	*b c*	*d b e*	
a b d	*f*: 2	*d b a*	*d c*	
b d e	*g*: 1	*d b e*	*d c a e*	
b c e g		*b c e*	*d a*	
c d f		*d c*	*b c*	
a b d	$s_{\min} = 3$	*d b a*	*b c e*	

1. Original transaction database.
2. Frequency of individual items.
3. Items in transactions sorted descendingly w.r.t. their frequency and infrequent items removed.
4. Transactions sorted lexicographically in ascending order (comparison of items is the same as in preceding step).
5. Data structure used by the algorithm (details on next slide).

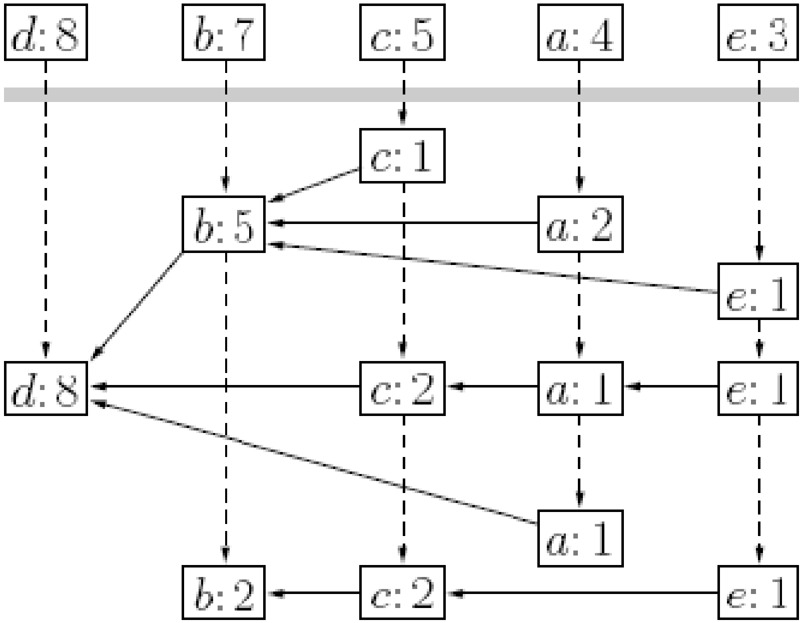
Transaction Representation: FP-Tree

- Build a **frequent pattern tree (FP-tree)** from the transactions (basically a prefix tree with links between branches for items).
- Frequent single item sets can be read directly from the FP-tree.

Simple Example Database

① $a\ d\ f$
$a\ c\ d\ e$
$b\ d$
$b\ c\ d$
$b\ c$
$a\ b\ d$
$b\ d\ e$
$b\ c\ e\ g$
$c\ d\ f$
$a\ b\ d$

④ $d\ b$
$d\ b\ c$
$d\ b\ a$
$d\ b\ a$
$d\ b\ e$
$d\ c$
$d\ c\ a\ e$
$d\ a$
$b\ c$
$b\ c\ e$



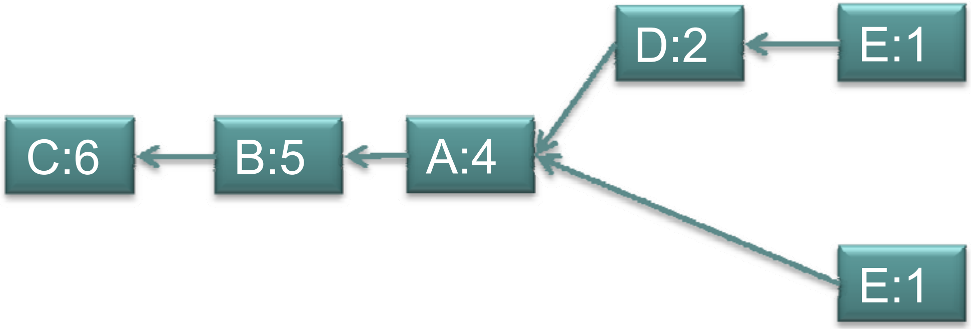
FP-tree

FP-Growth

```
FP-growth(FP, prefix)

foreach frequent item x in increasing order of frequency do
   prefix = prefix ∪ x
   Dx = ∅
   $count_x$ = 0
   foreach node-link $nl_x$ of x do
        $D_x$ = $D_x$ ∪ {transaction of path reaching x, with
                     count for each item = $nl_x$.count, without x}
        $count_x$ += $nl_x$.count
   end
   if $count_x$ ≥ minsup then
        output (prefix ∪ x)
        $FP_x$ = FP-tree constructed from $D_x$
        FP-growth($FP_x$, prefix)
   end if
end
```

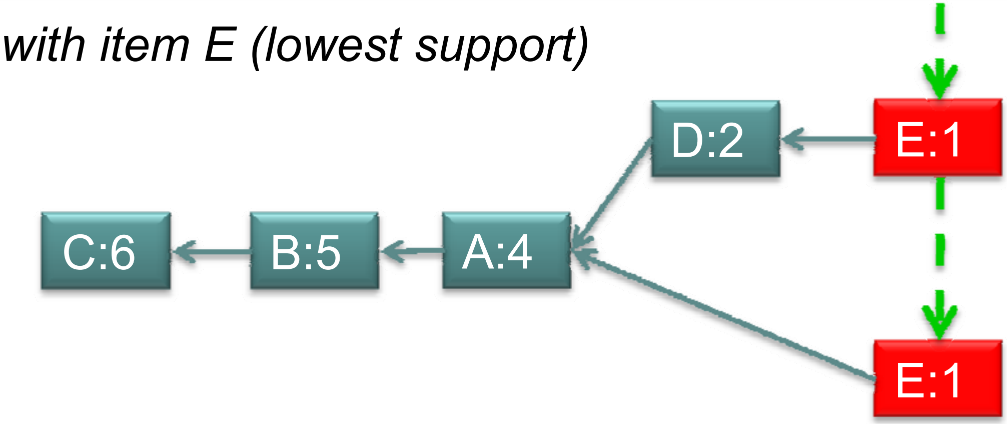
FP-Growth example

C : 6
B : 5
A : 4
D : 2
E : 2



Original FP-tree

Start with item E (lowest support)

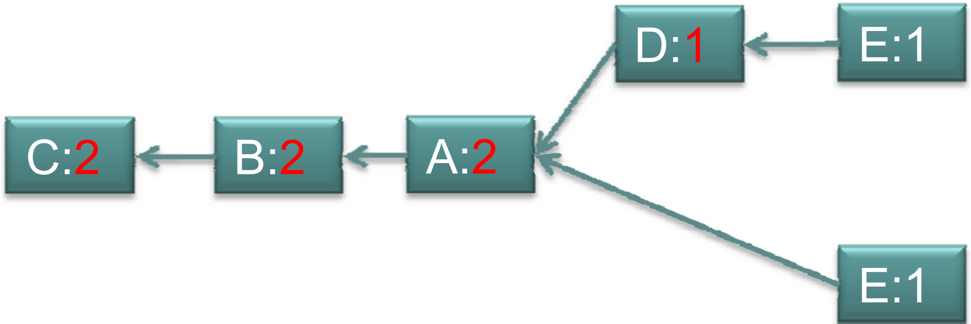


$\text{count}_E = 1+1 = 2$

⇒ E is frequent

⇒ Output E

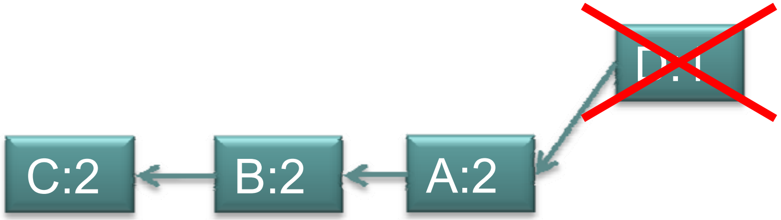
Conditional FP-tree for E



- update counts
  → only transactions containing E
- drop E

FP-Growth example (cont.)

Conditional FP-tree for E

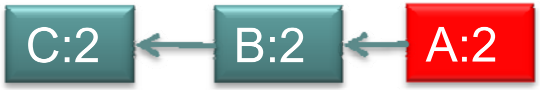


D not frequent here
→ do not consider DE

Loop on ~~AE~~, BE, CE

The rest is left as exercise…

For AE :

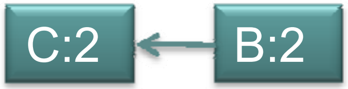


count$_{AE}$= 2

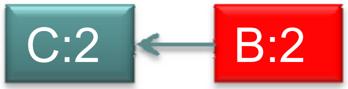
⇒ AE is frequent

⇒ Output AE

Conditional FP-tree for AE:



For BAE :



count$_{BAE}$= 2

⇒ BAE is frequent

⇒ Output BAE

Conditional FP-tree for BAE:

C:2

For CBAE :

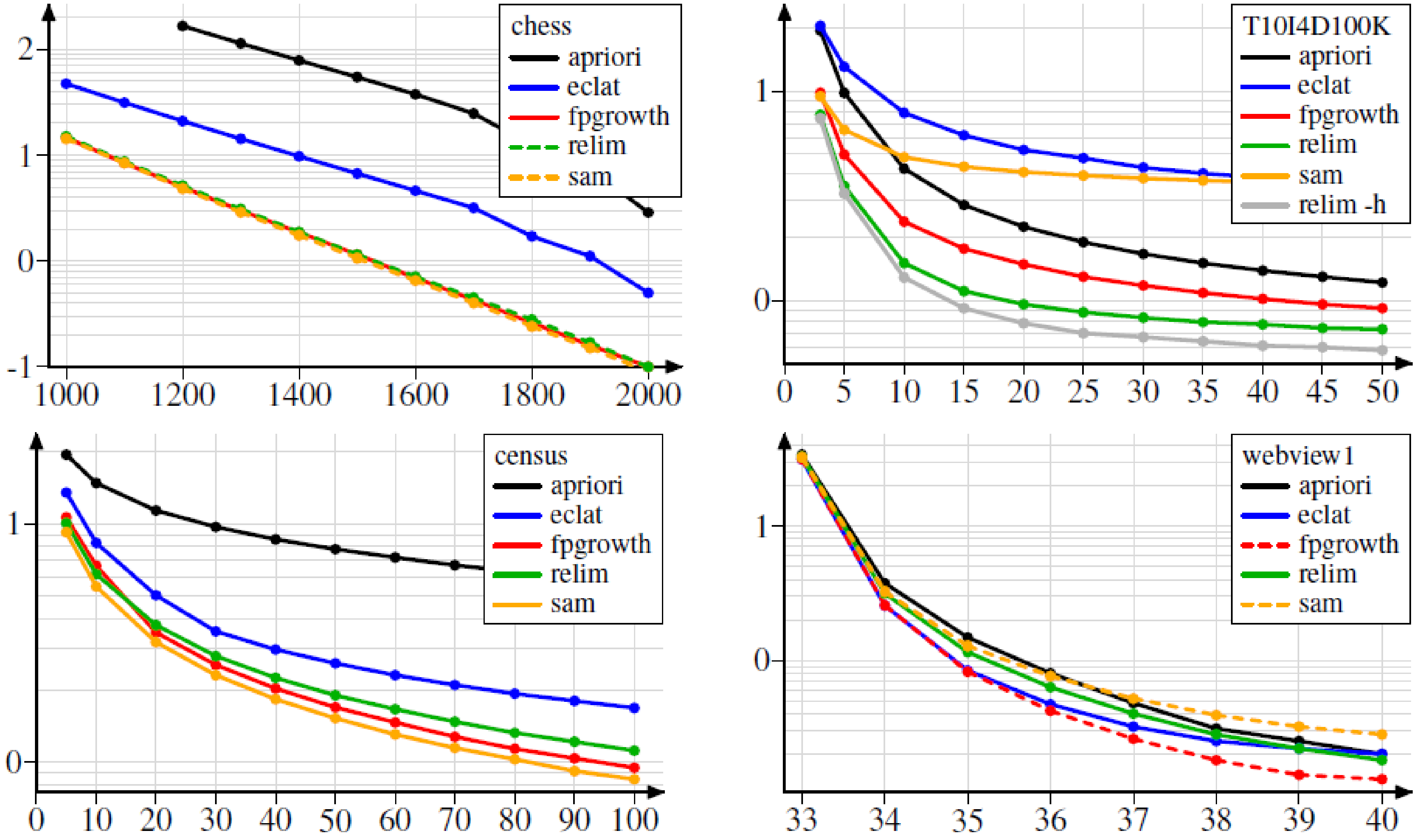
C:2

count$_{CBAE}$= 2

⇒ CBAE is frequent

⇒ Output CBAE

Experiments: Execution Times



Decimal logarithm of execution time in seconds over absolute minimum support.

Problems of frequent itemsets

- Large computation time
- For low support values, huge number of frequent itemsets
- Lots of redundant information

Simple example:

Tid	Transaction
1	A B C D
2	A B C
3	A B C E

minsup = 2

FIS	Support	Tidlist
A B C	3	{1,2,3}
A B	3	{1,2,3}
A C	3	{1,2,3}
B C	3	{1,2,3}
A	3	{1,2,3}
B	3	{1,2,3}
C	3	{1,2,3}

REDUNDANT

Closed frequent itemsets

[Pasquier *et al.*, 99]

- We have seen that there is **loss of information** with maximal frequent itemsets

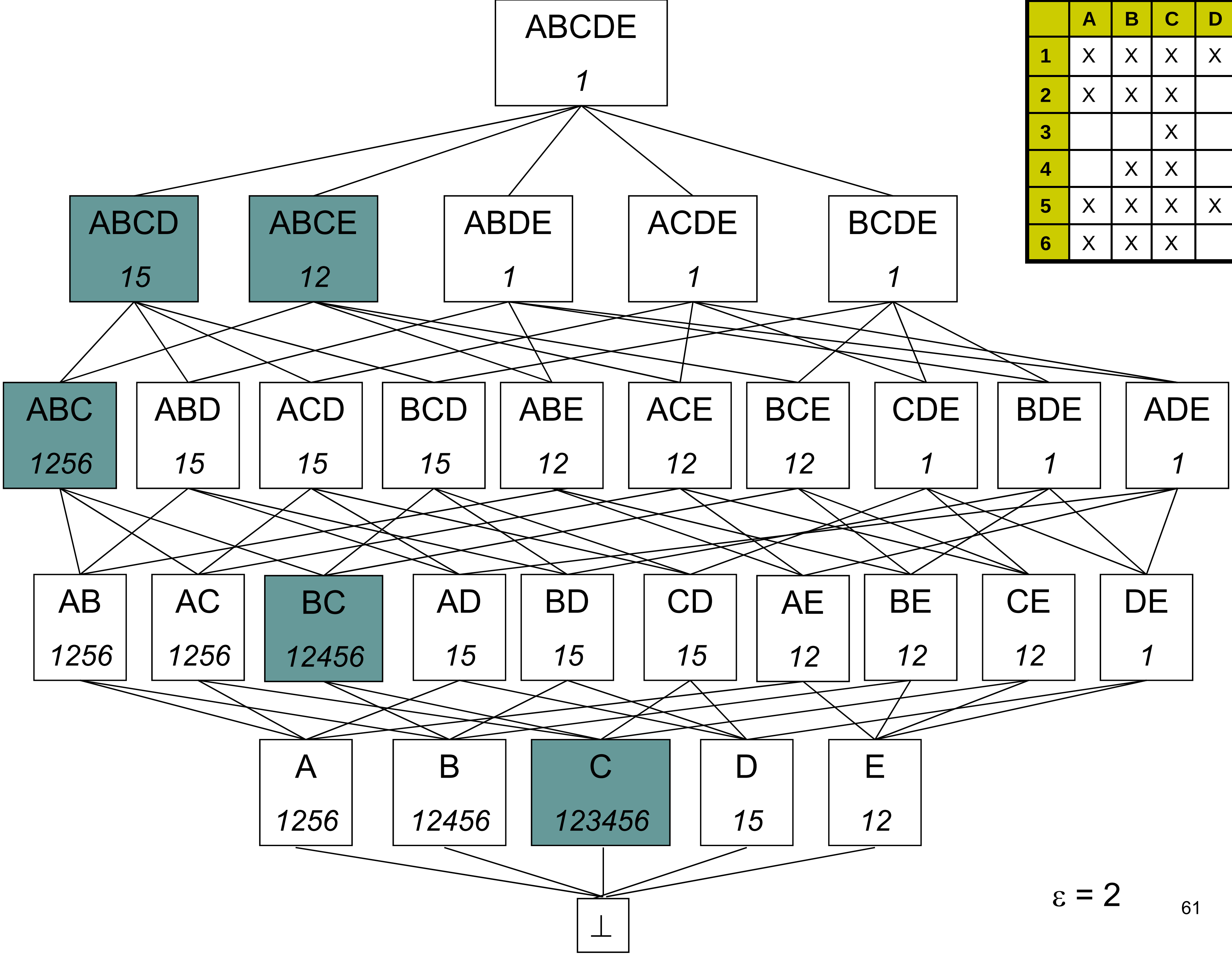
- Lets consider equivalence classes for frequent itemsets sharing the same tidlist
- The closed frequent itemsets are the maximums of these equivalence classes

Set of closed frequent itemsets :

$$CFI = \{\, I \in FI \mid \forall I' \in FI \text{ tq } tidlist(I')=tidlist(I) \quad I' \subseteq I \,\}$$

with FI set of frequent itemsets

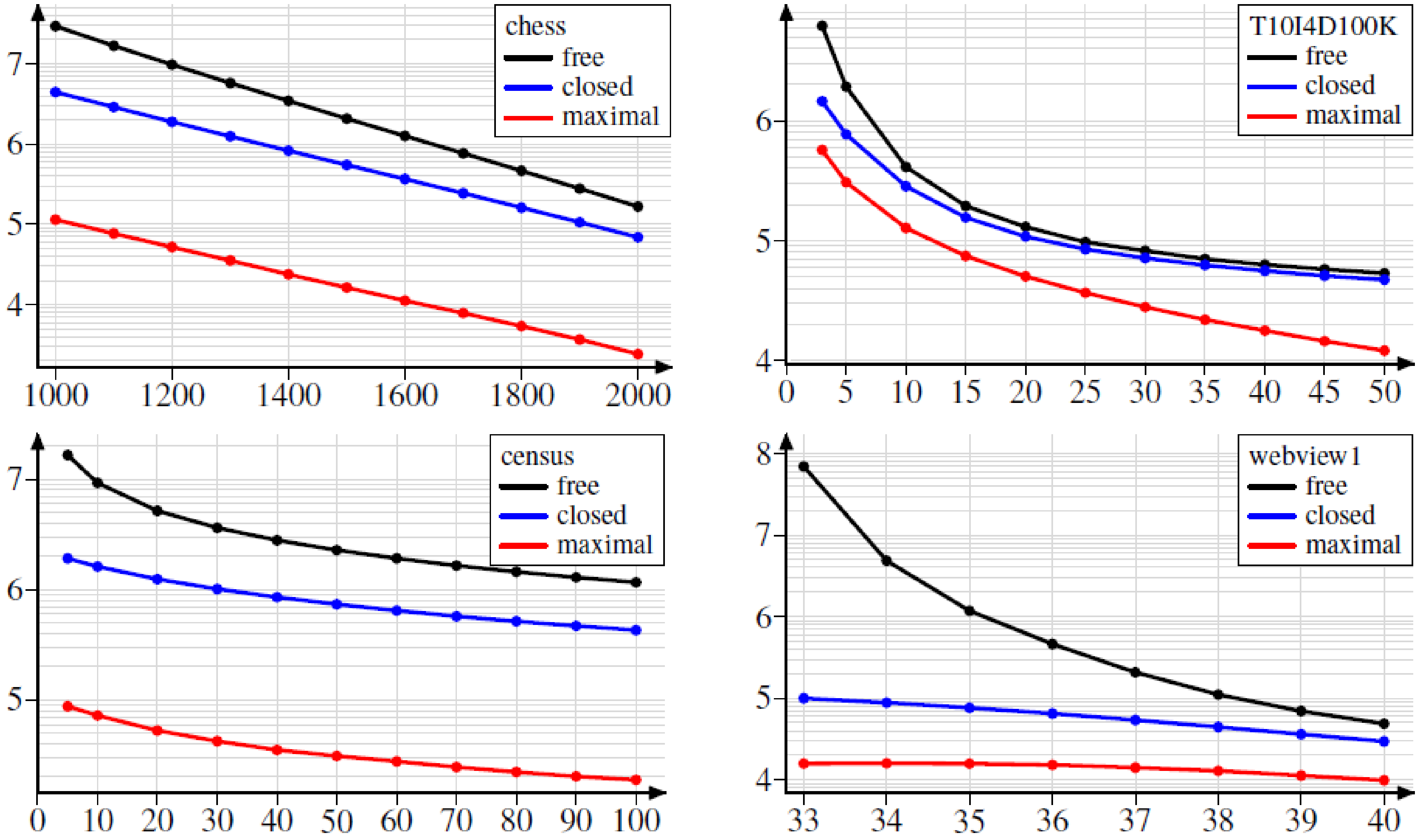
$\Rightarrow$ Sets are ordered by inclusion: $MFI \subseteq CFI \subseteq FI$



	A	B	C	D	E
1	X	X	X	X	X
2	X	X	X		X
3			X		
4		X	X		
5	X	X	X	X	
6	X	X	X		

$\varepsilon = 2$

Types of Frequent Item Sets: Experiments



Decimal logarithm of the number of item sets over absolute minimum support.

Computing closed frequent itemsets

- Brute force *(frequent pattern base)*
  - Enumerate all the frequent patterns
  - Output only closed ones
  - *Most of the time : inefficient*
    - *Exception: if |FI| is very small*
- Closure base
  - Compute only closed patterns with closure operations
  - *Can be very efficient*

Efficient computation

- First algorithms (Closet, Charm,…)
  - Candidate-based method
  - **Try** to compute as many non-closed frequent itemsets as possible
  - OR Closure Extension: **add an item to an existing closed frequent itemset, and take closure**
  - Keep in memory all closed frequent itemsets found so far
  - → *Need a lot of memory during execution*

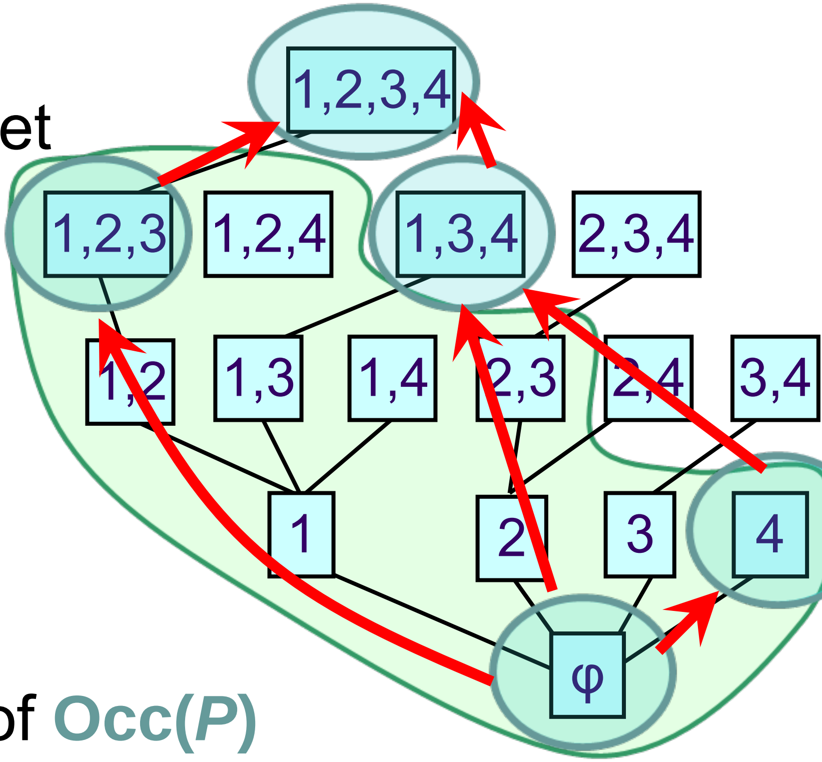
- Reverse search (LCM)
  - Depth First Search algorithm so no global memory needed
  - *Fast computation time, Little memory usage*

Closure Extension of Itemset

- Usual backtracking does not work for closed itemsets, because there are possibly big gap between closed itemsets

- On the other hand, any closed itemset is obtained from another by **"add an item and take closure (maximal)"**

  - closure of *P* is the closed itemset having the same denotation to *P*, and computed by taking intersection of **Occ(*P*)**



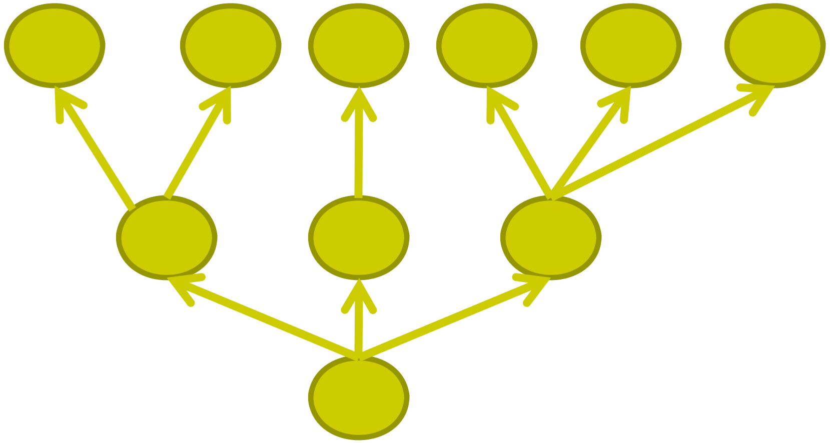
This is an adjacency structure defined on closed itemsets, thus we can perform graph search on it, with using memory

Reverse Search

- Uno and Arimura found that the closed frequent itemsets are organized in a **directed spanning tree**

● : Closed frequent itemsets

↑ : transition function



- ⇒ they can be visited by DFS
- ⇒ from a node of the tree, need of a *transition function* to compute its children

Parent–Child Relation

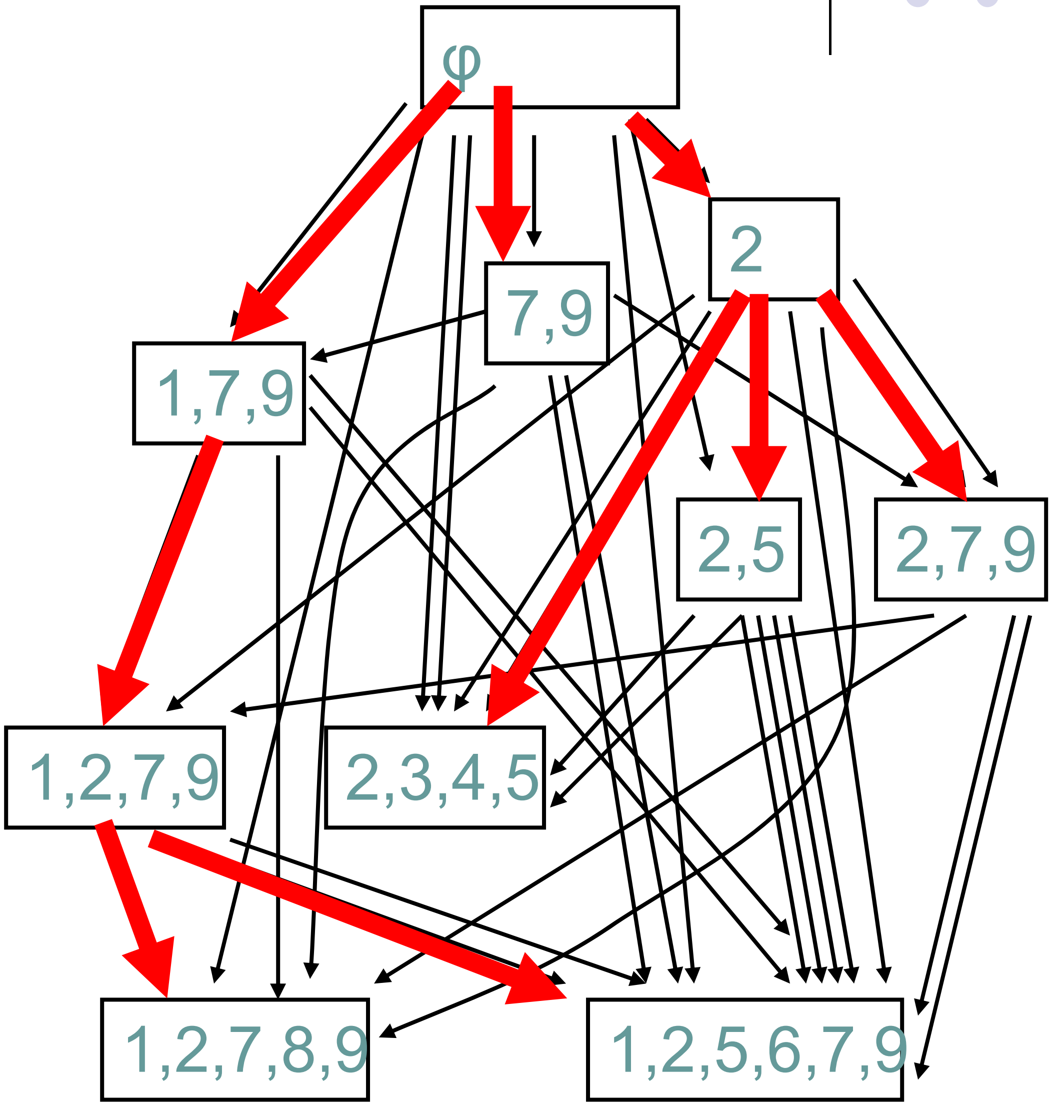
- All closed itemsets and parent-child relation

D =

1,2,5,6,7,9
2,3,4,5
1,2,7,8,9
1,7,9
2,7,9
2

← Adjacency by adding one item

← Parent-child

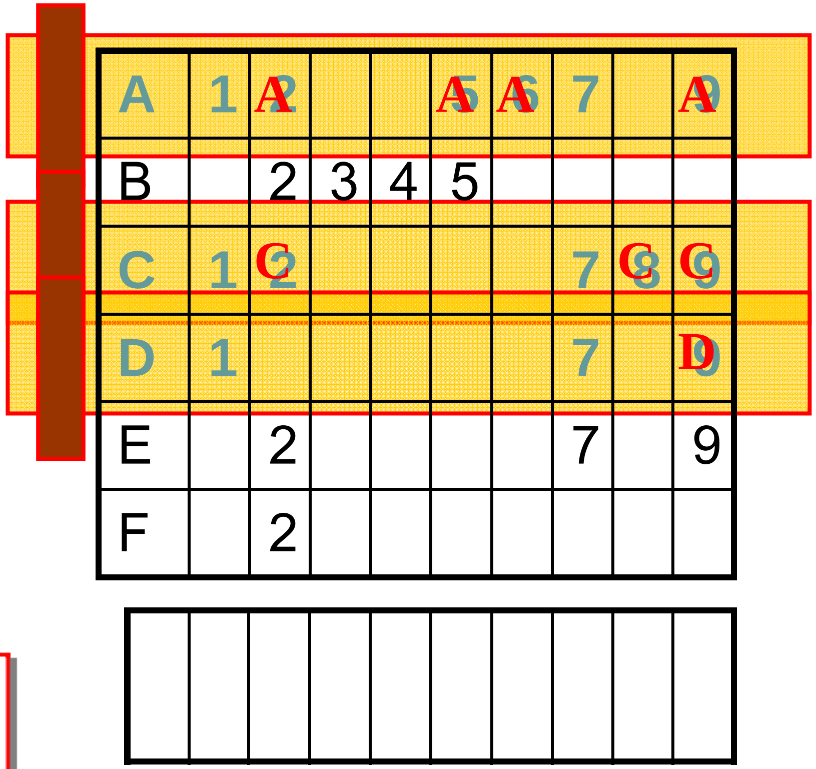


Occurrence Deliver

- Compute the denotations of *P* ∪{e} for **all** e’s at once, by scanning each occurrence

D= 1,2,5,6,7,9
2,3,4,5
1,2,7,8,9
1,7,9
2,7,9
2

P = {1,7}



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

Check the frequency for all items to be added in linear time of the database size

frequency of item = reliability of rule
Computed in short time

Database Reductions

Conditional database is to reduce database by unnecessary items and transactions, for deeper levels

1,3,5
1,3,5
1,3,5
1,2,5,6
1,4,6
1,2,6

$\theta = 3$

filtering

Remove infrequent items, items included in all

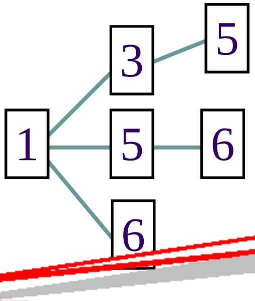
3,5
3,5
3,5
5,6
6
6

filtering

Unify same transactions

Linear time

3,5 ×3
5,6
6 ×2

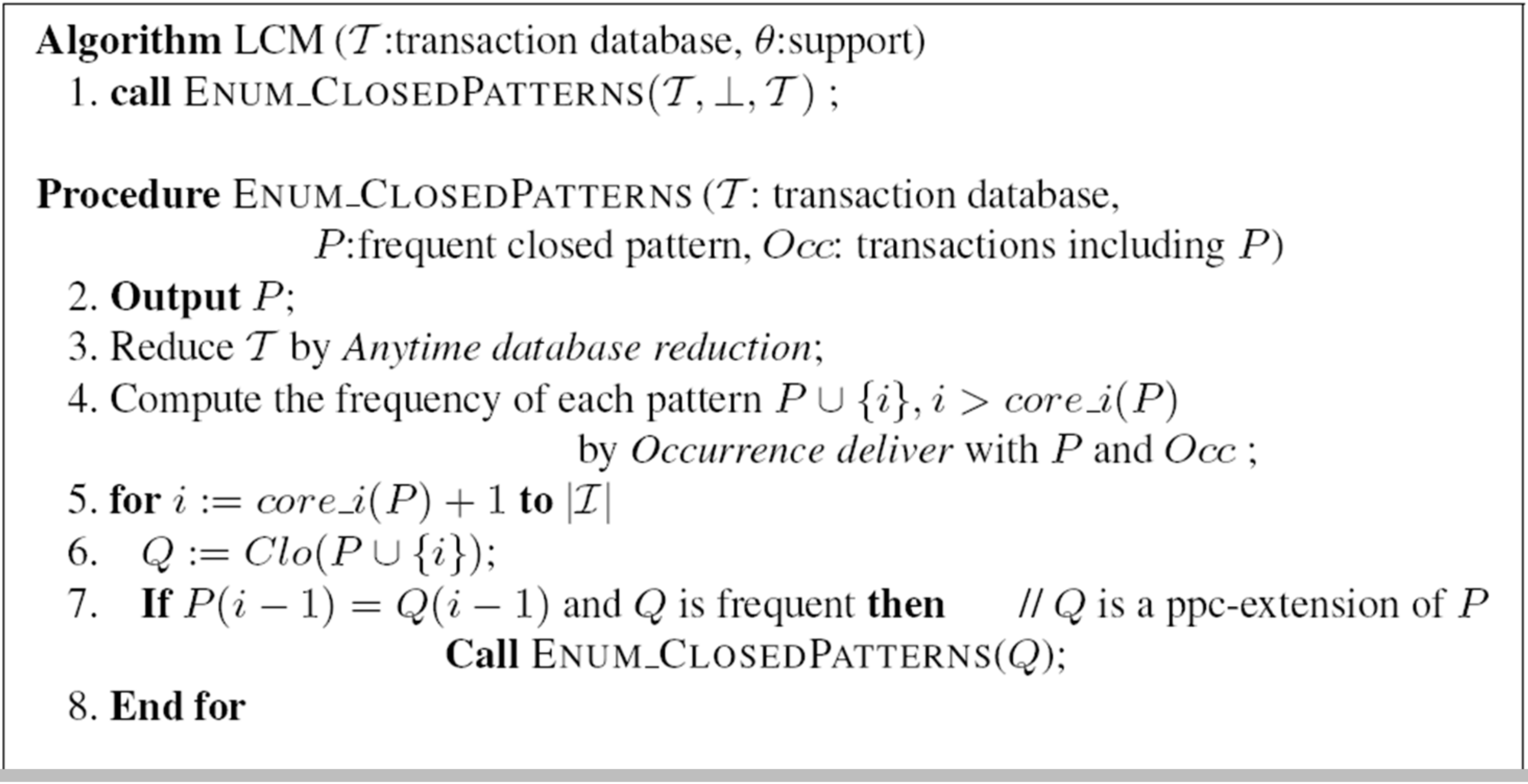


FP-tree, prefix tree

Remove infrequent items, automatically unified

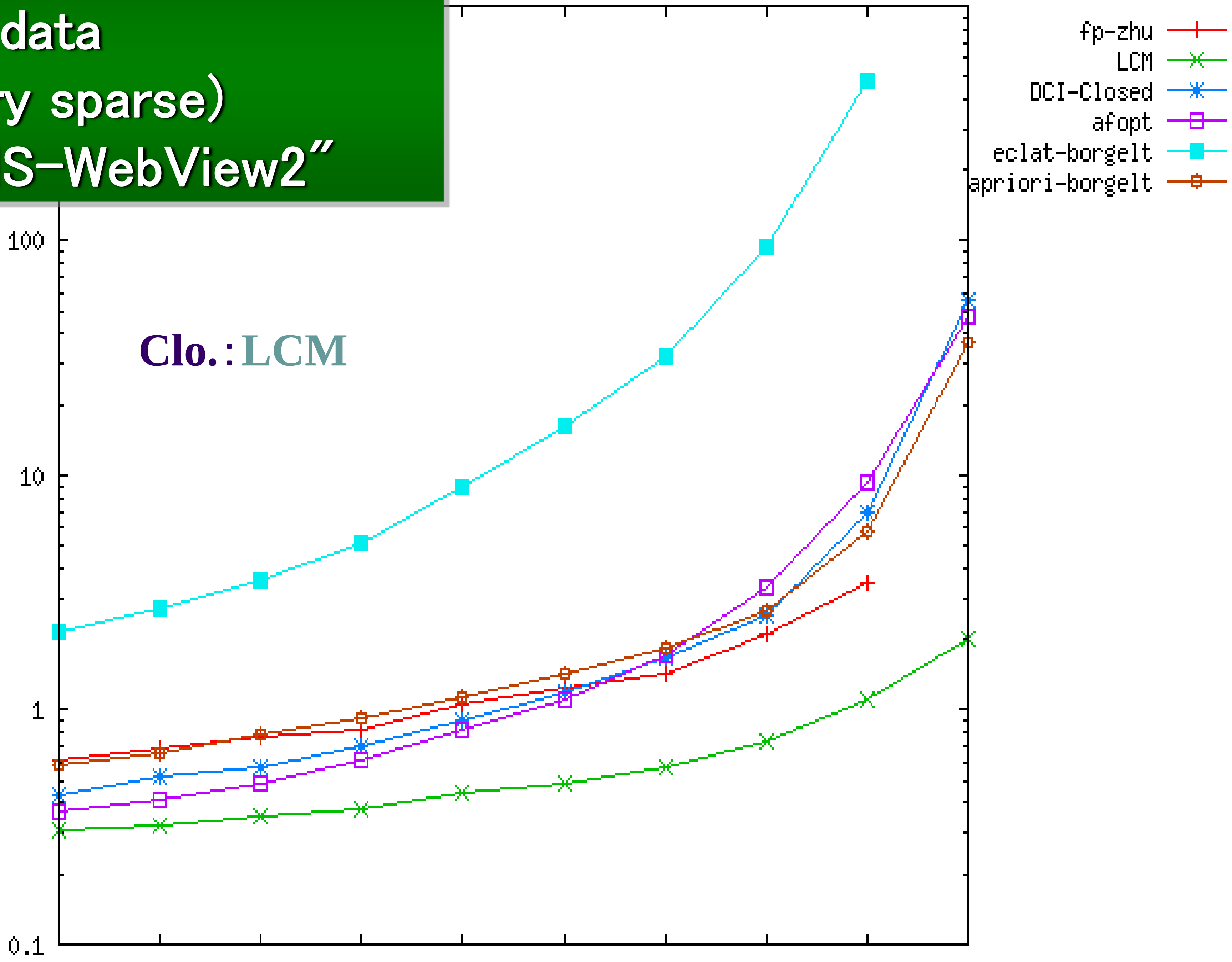
$O(||D|| \log ||D||)$ time

Compact if database is dense and large



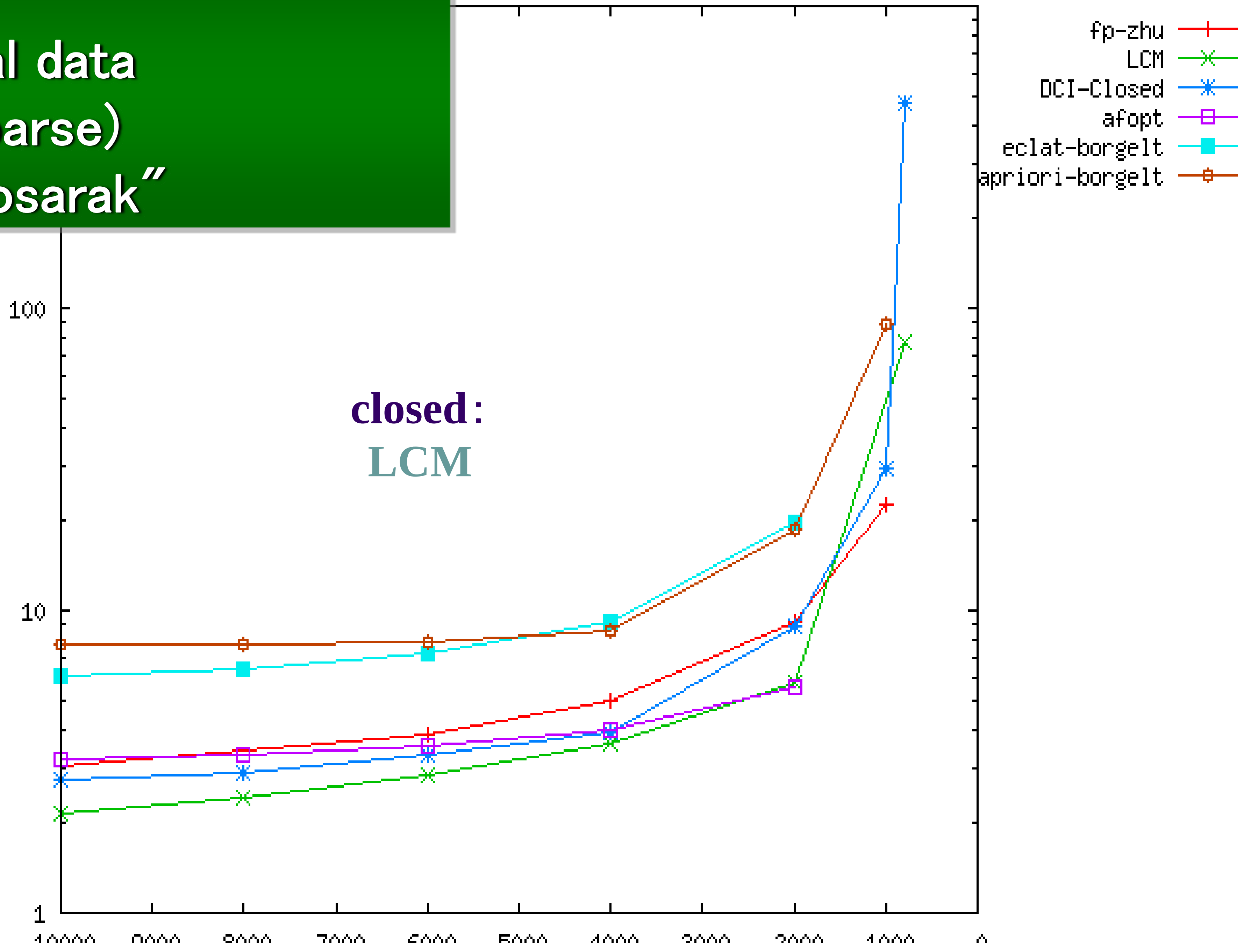
```
**Algorithm** LCM ($\mathcal{T}$:transaction database, $\theta$:support)
    **call** ENUM_CLOSEDPATTERNS($\mathcal{T}, \bot, \mathcal{T}$) ;

**Procedure** ENUM_CLOSEDPATTERNS ($\mathcal{T}$: transaction database,
                        $P$:frequent closed pattern, $Occ$: transactions including $P$)
    **Output** $P$;
    Reduce $\mathcal{T}$ by *Anytime database reduction*;
    Compute the frequency of each pattern $P \cup \{i\}, i > core\_i(P)$
                                        by *Occurrence deliver* with $P$ and $Occ$ ;
    **for** $i := core\_i(P) + 1$ **to** $|\mathcal{I}|$
        $Q := Clo(P \cup \{i\})$;
        **If** $P(i-1) = Q(i-1)$ and $Q$ is frequent **then**      // $Q$ is a ppc-extension of $P$
                                **Call** ENUM_CLOSEDPATTERNS($Q$);
    **End for**
```

real data
(very sparse)
"BMS-WebView2"



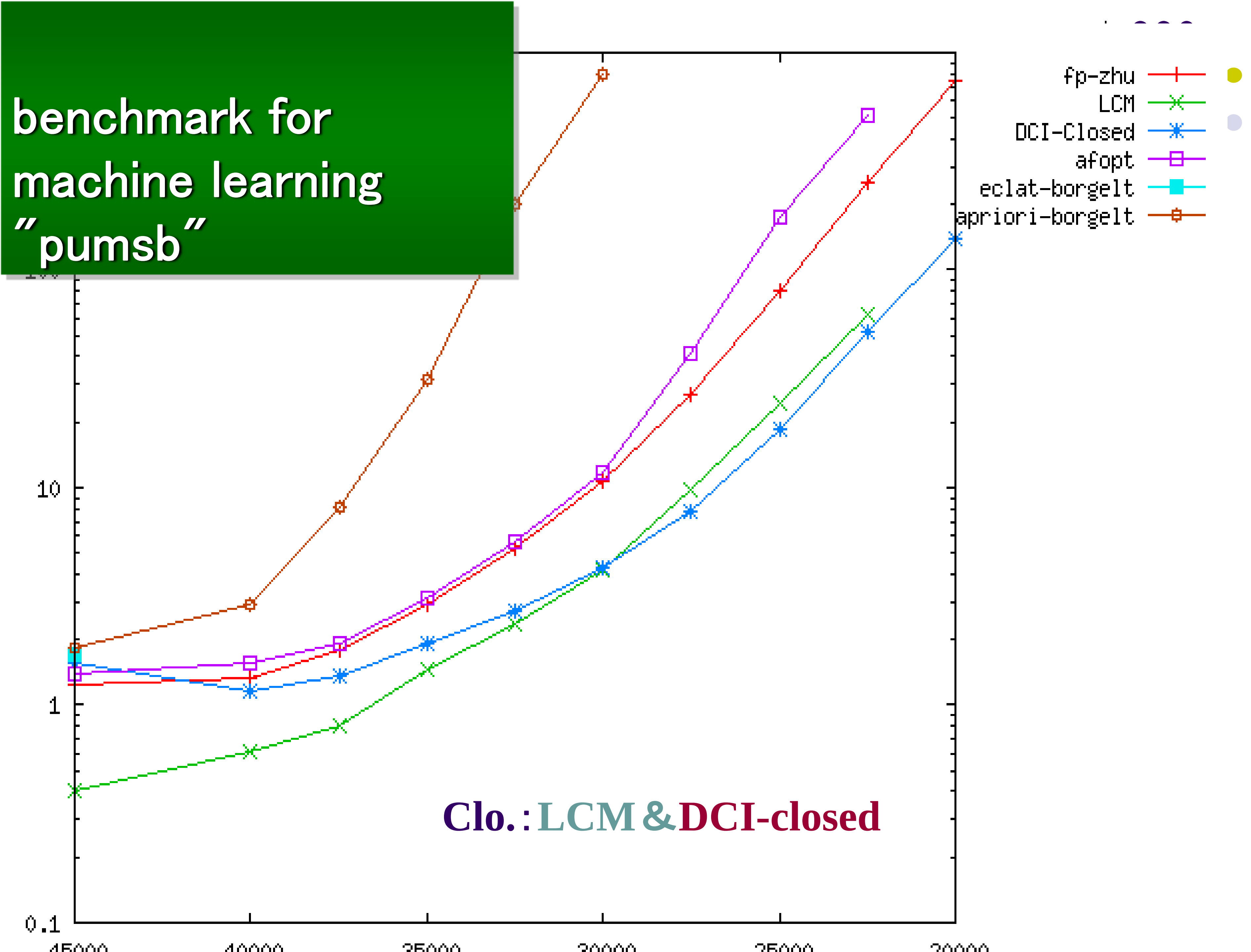
Clo.: LCM

real data (sparse) "kosarak"



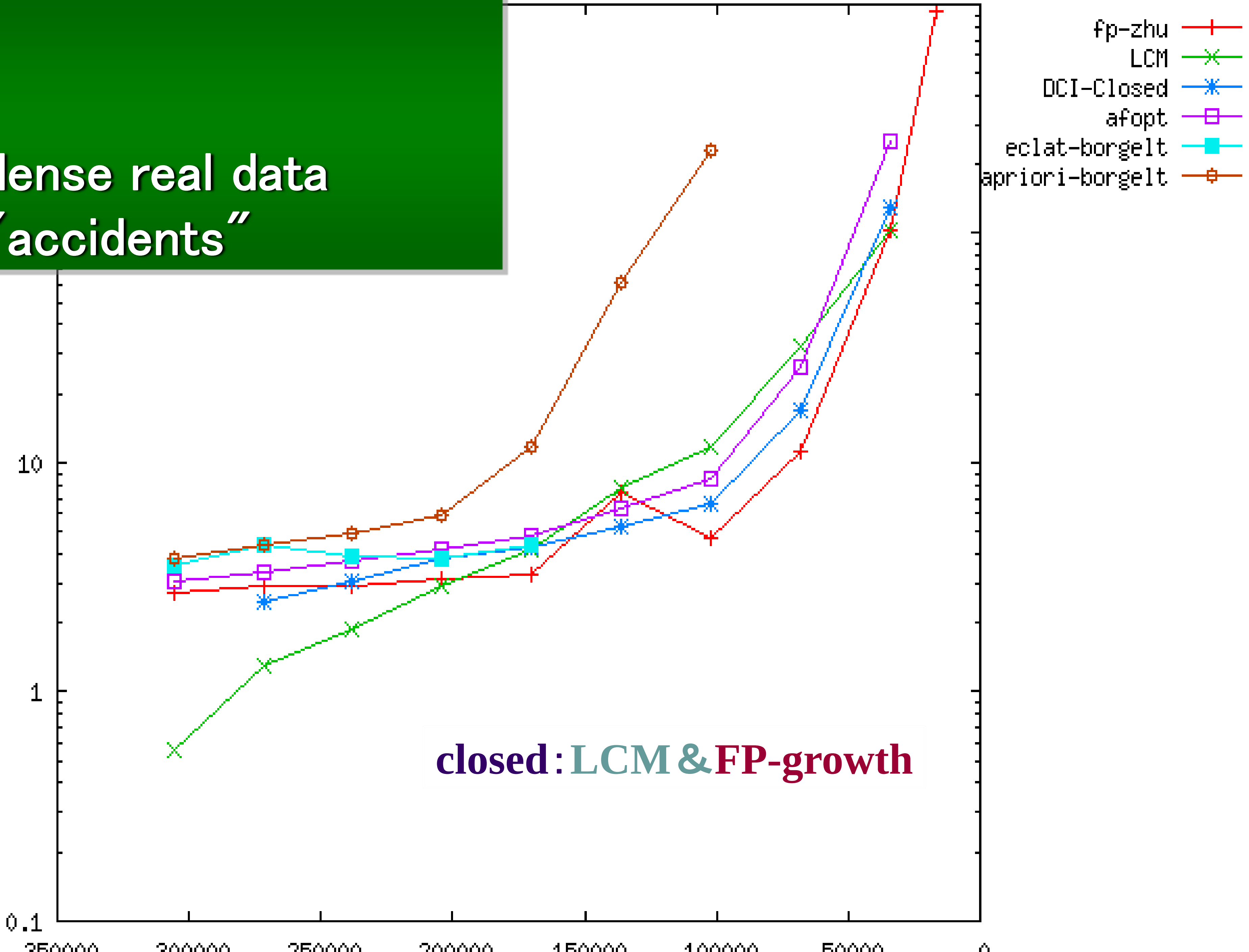
closed: LCM

benchmark for machine learning "pumsb"



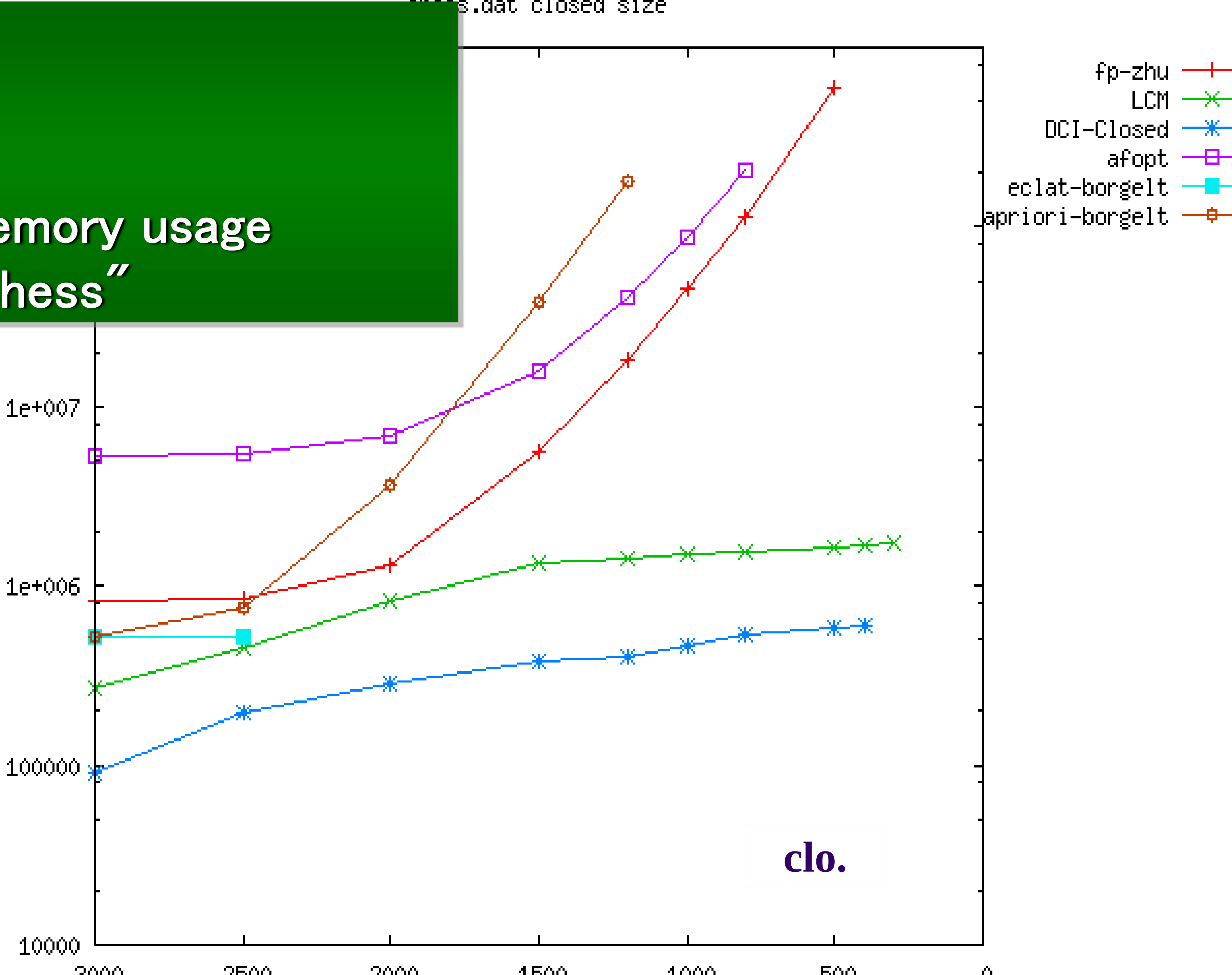
Clo.:LCM&DCI-closed

dense real data ”accidents”



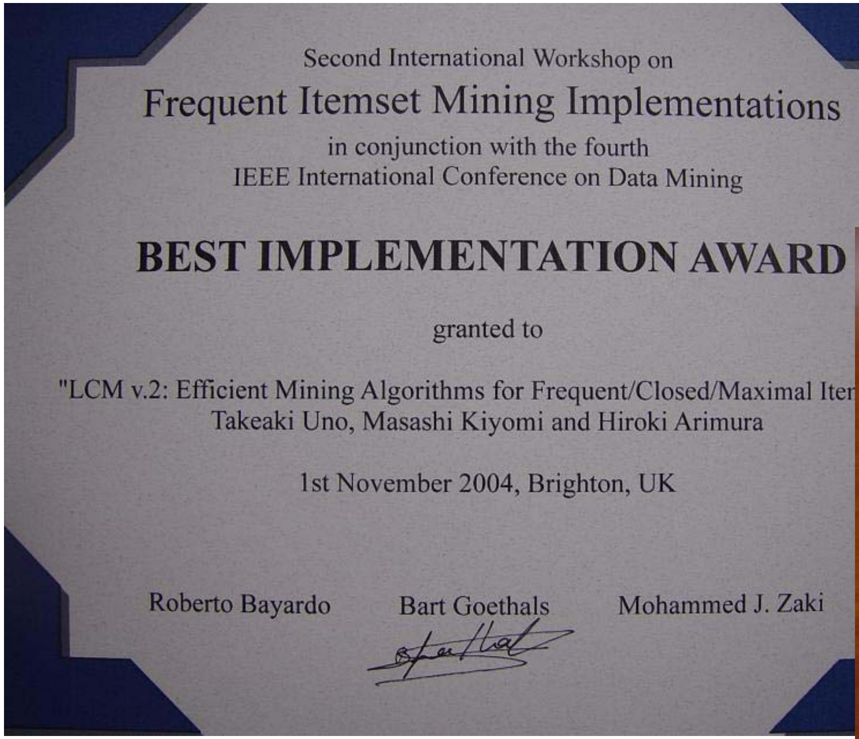
closed : LCM & FP-growth

memory usage
“chess”



clo.

Prize for the Award



Second International Workshop on

Frequent Itemset Mining Implementations

in conjunction with the fourth
IEEE International Conference on Data Mining

BEST IMPLEMENTATION AWARD

granted to

"LCM v.2: Efficient Mining Algorithms for Frequent/Closed/Maximal Iter
Takeaki Uno, Masashi Kiyomi and Hiroki Arimura

1st November 2004, Brighton, UK

Roberto Bayardo | Bart Goethals | Mohammed J. Zaki

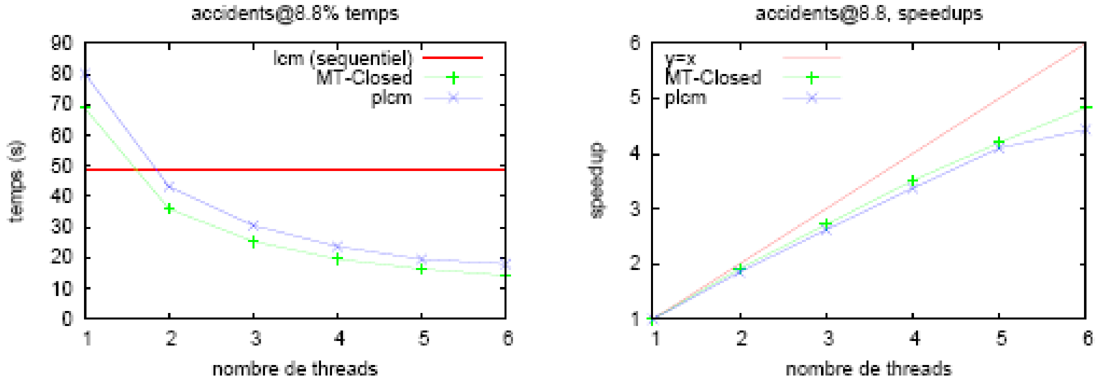
Prize is {beer, diapers}
"Most Frequent Itemset"

Implémentations de LCM

- LCM5.3
  - Implémentation en C de Takeaki Uno (NII)
  - Très rapide mais n'exploite qu'un coeur du processeur
  - http://research.nii.ac.jp/~uno/codes.htm

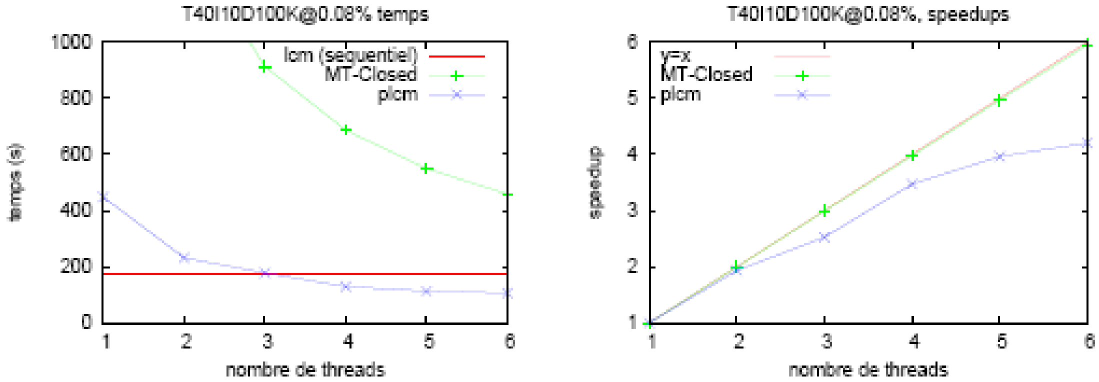
- PLCM
  - Implémentation parallèle en C++ de Benjamin Négrevergne (LIG)
  - Le plus rapide grâce à l'exploitation des processeurs multi-coeurs
  - http://melindaplcm.ligforge.imag.fr/

- HLCM
  - Implémentation parallèle en Haskell, A. Termier (LIG)
  - Le plus lent, mais peut exploiter les processeurs multi-coeurs
  - Résultats directement utilisables dans un programme Haskell
  - http://hackage.haskell.org/package/hlcm



(a) Execution time (b) Speedup

FIG. 1 – *Résultats pour accidents à 8.8% (dense)*

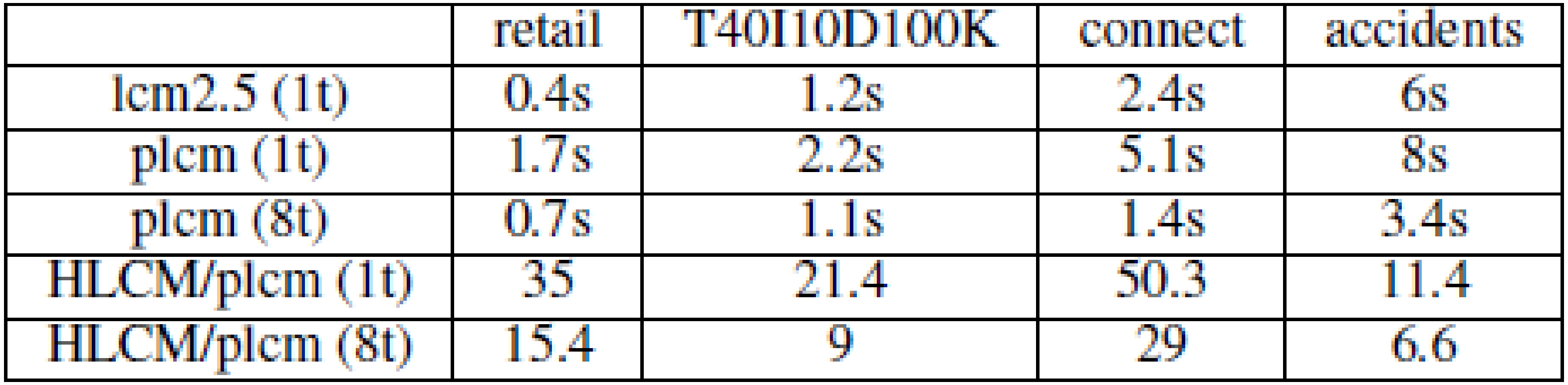


(a) Execution time (b) Speedup

FIG. 2 – *Résultats pour T10I4D100K à 0.08% (creux)*

Performances PLCM

Performances HLCM



	retail	T40I10D100K	connect	accidents
lcm2.5 (1t)	0.4s	1.2s	2.4s	6s
plcm (1t)	1.7s	2.2s	5.1s	8s
plcm (8t)	0.7s	1.1s	1.4s	3.4s
HLCM/plcm (1t)	35	21.4	50.3	11.4
HLCM/plcm (8t)	15.4	9	29	6.6