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 $\rightarrow$ Y: $$c = \frac{\text{support}(X \cup Y)}{\text{support}(X)}$$

- 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	butter

- *{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

- $\rightarrow$ some pruning is possible

Association rule generation algorithm

```
Input: T, minsup, minconf, F_all = union of F_1..F_n

H_1 = ∅
foreach f_k ∈ F_all, k≥2 do begin
   A = (k-1)-itemsets a_{k-1}  such that a_{k-1} ⊂ f_k ;
   foreach a_{k-1} ∈ A do begin
        conf = support(f_k)/support(a_{k-1}) ;
        if conf ≥ minconf do begin
             output rule a_{k-1} → (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 ∈ H_m+1 do begin
       conf = support(f_k)/support(f_k-h_m+1) ;
       if conf ≥ minconf then
           output rule f_k - h_m+1 → h_m+1 ;
       else
           delete h_m+1 from H_m+1 ;
   end
   ap-genrules(f_k, H_m+1) ;
end
```

Pruning by anti-monotony

The Eclat algorithm

[Zaki *et al.*, 97]

- Apriori : DB is in **horizontal** format
- Eclat introduces the **vertical** format
  - Itemset x $\rightarrow$ 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		

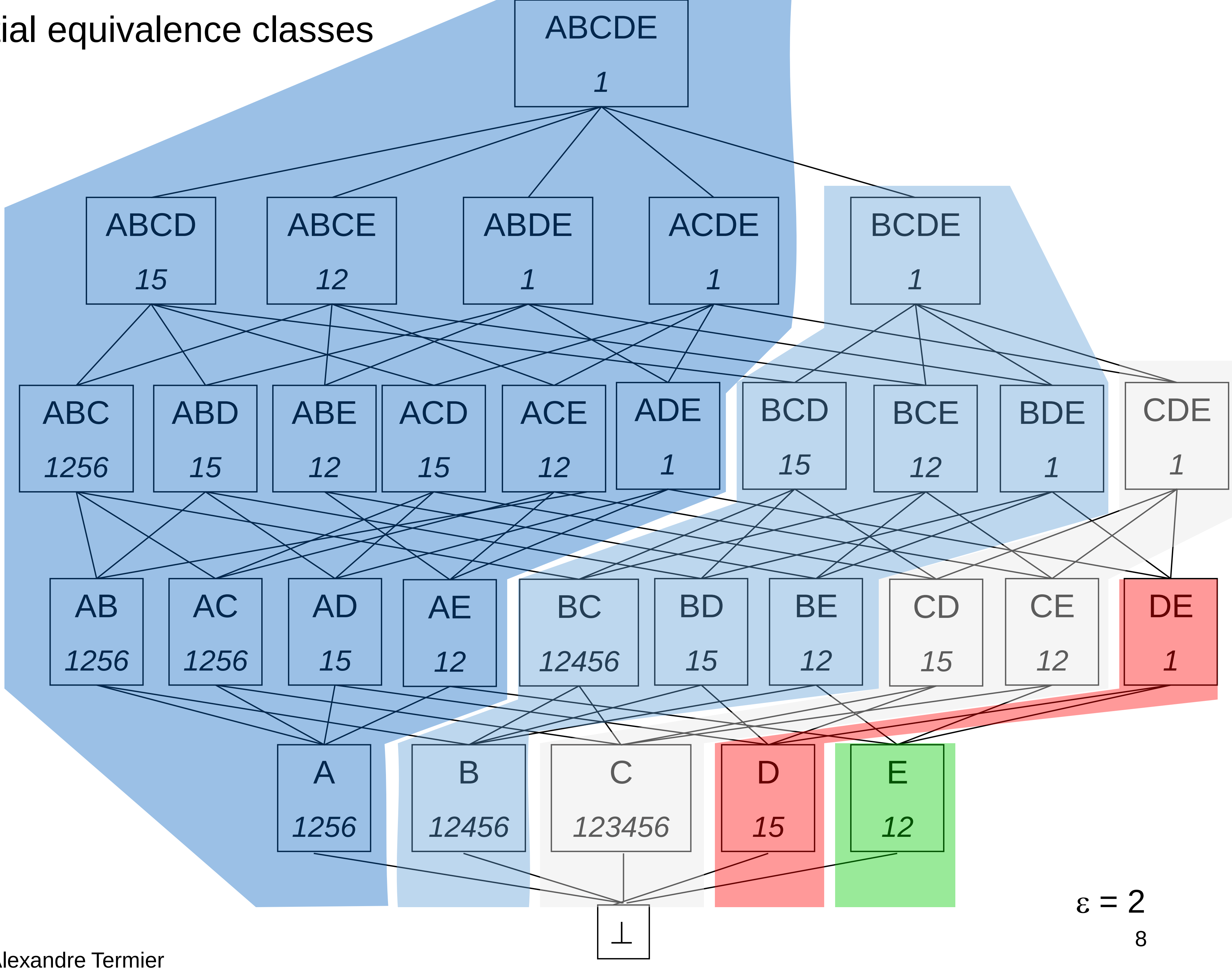
Vertical format

Vertical format

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

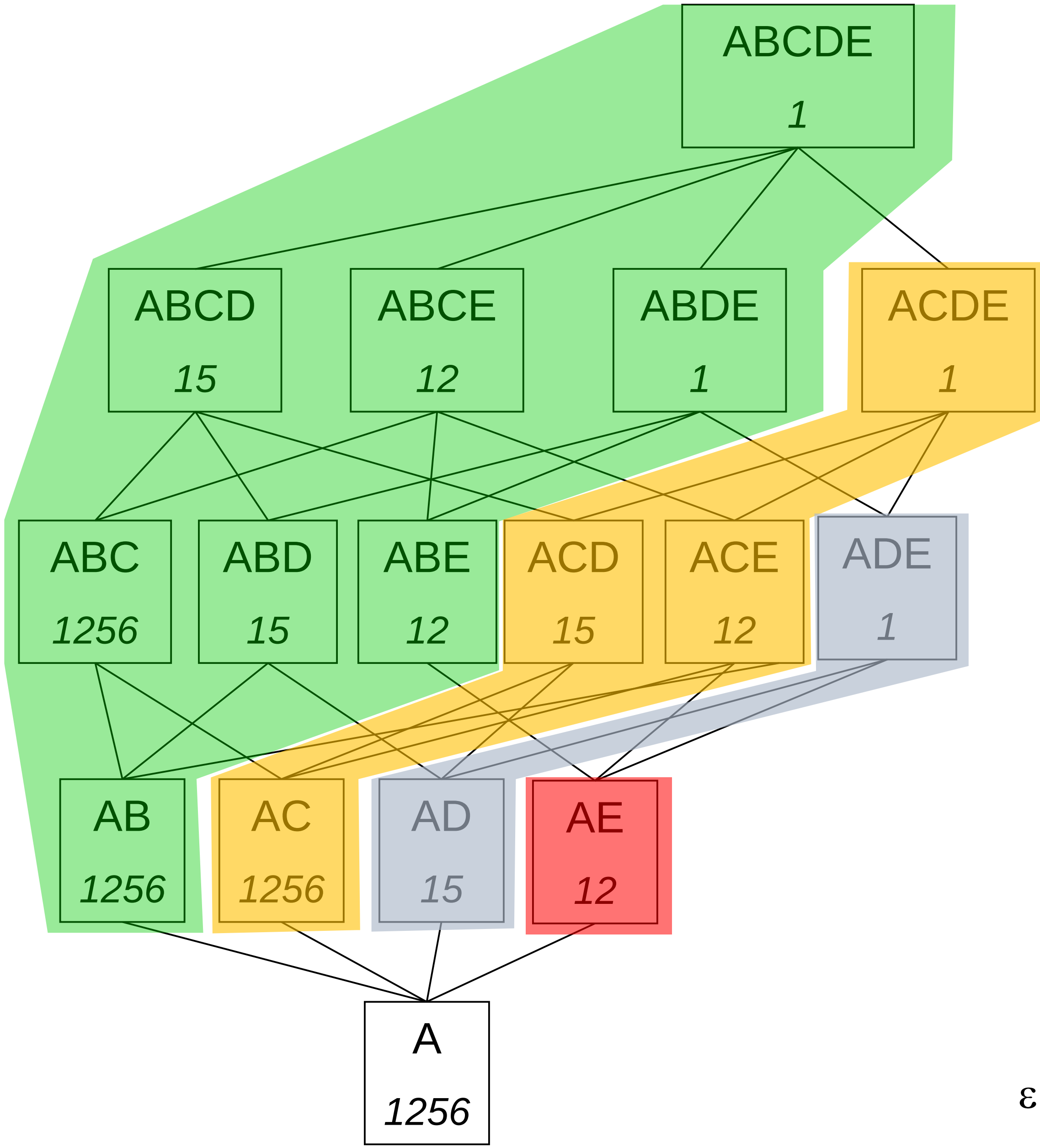
- Problem
  - If database is big, tidlists of the many candidates created will be big also, and will not hold in memory
- Solution
  - Partition the lattice into equivalence classes
  - In Eclat : equivalence relation = **sharing the same prefix**

Initial equivalence classes



ε = 2

Equivalence classes inside [A] class



ε = 2

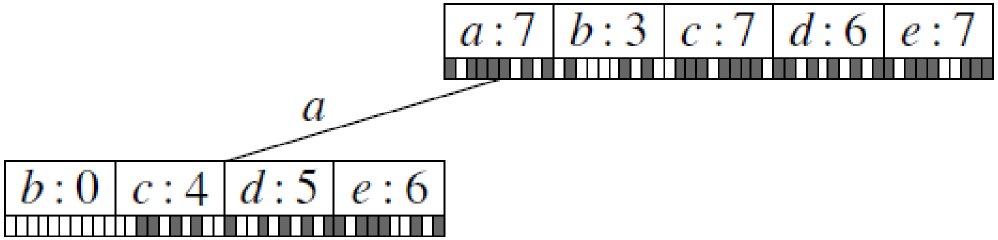
Eclat: Depth-First Search

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

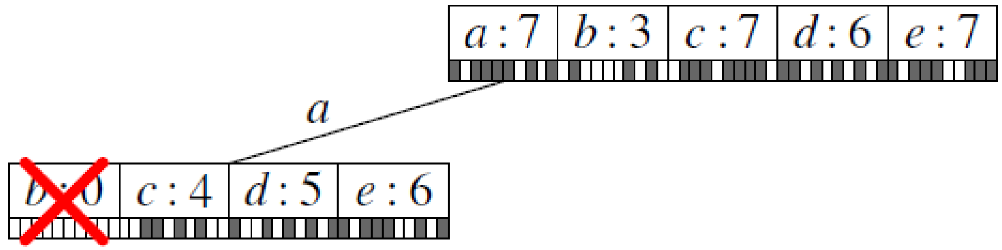
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

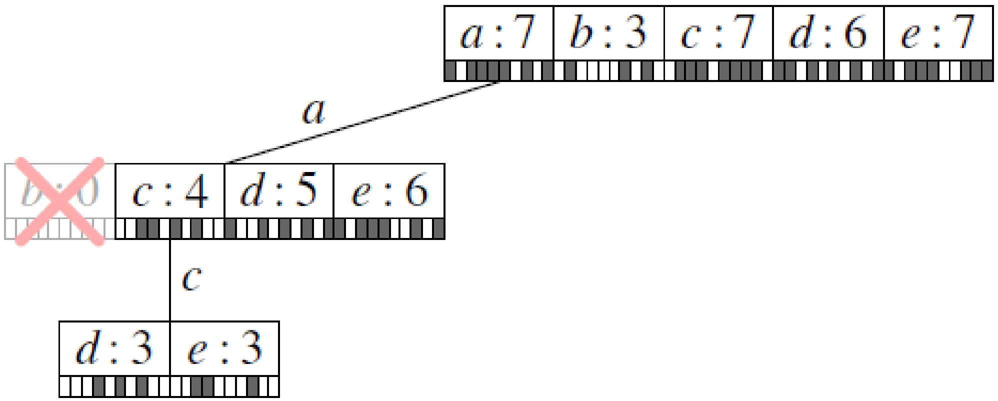
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

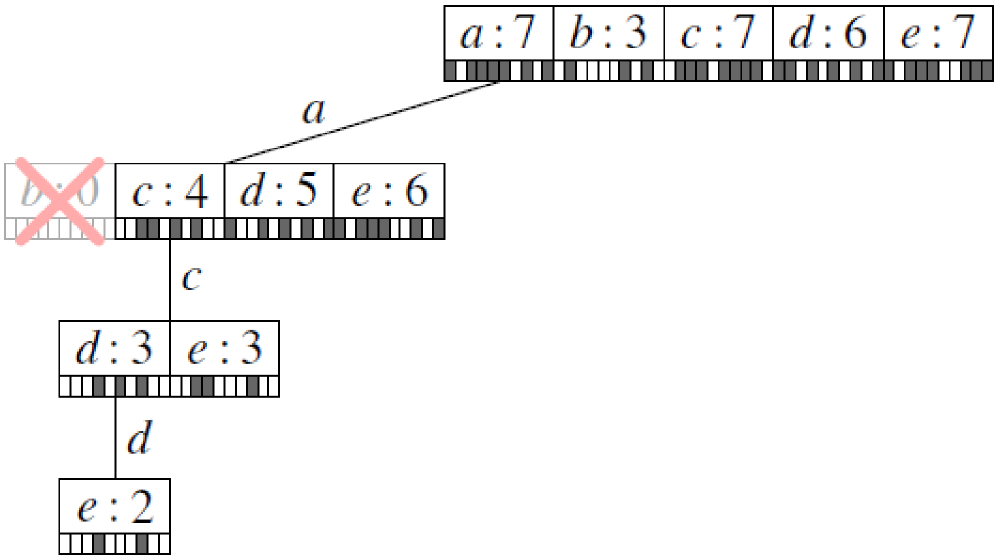
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

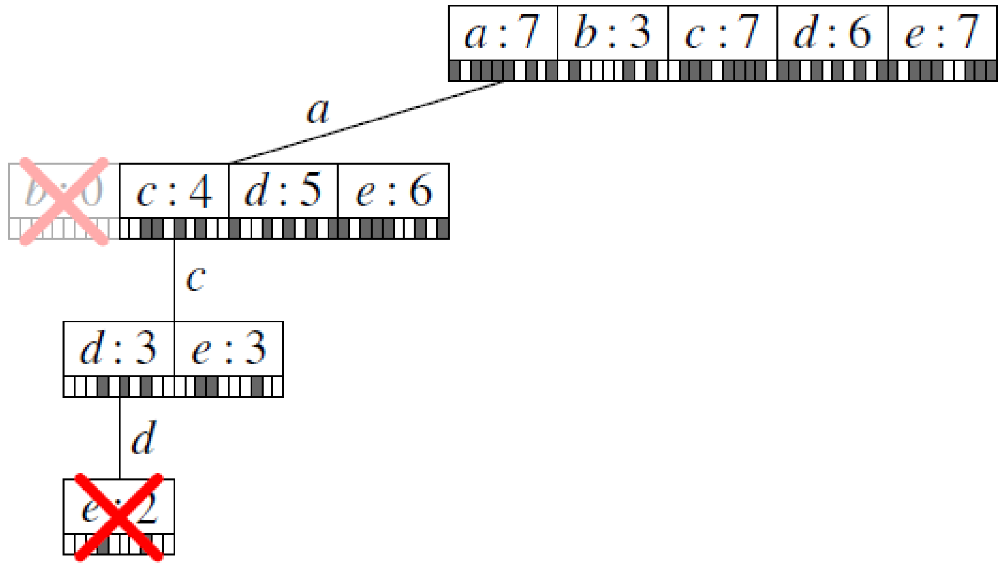
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

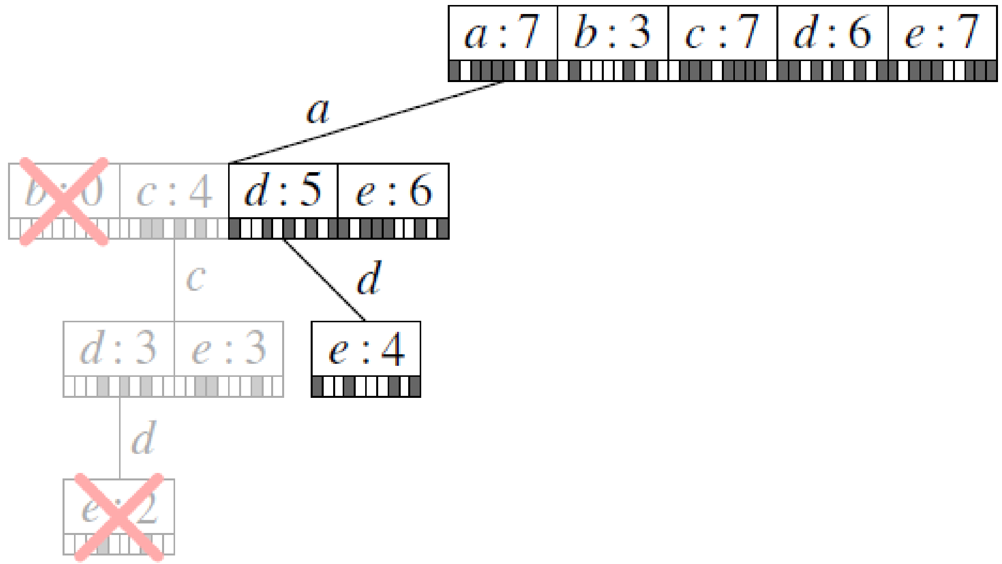
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

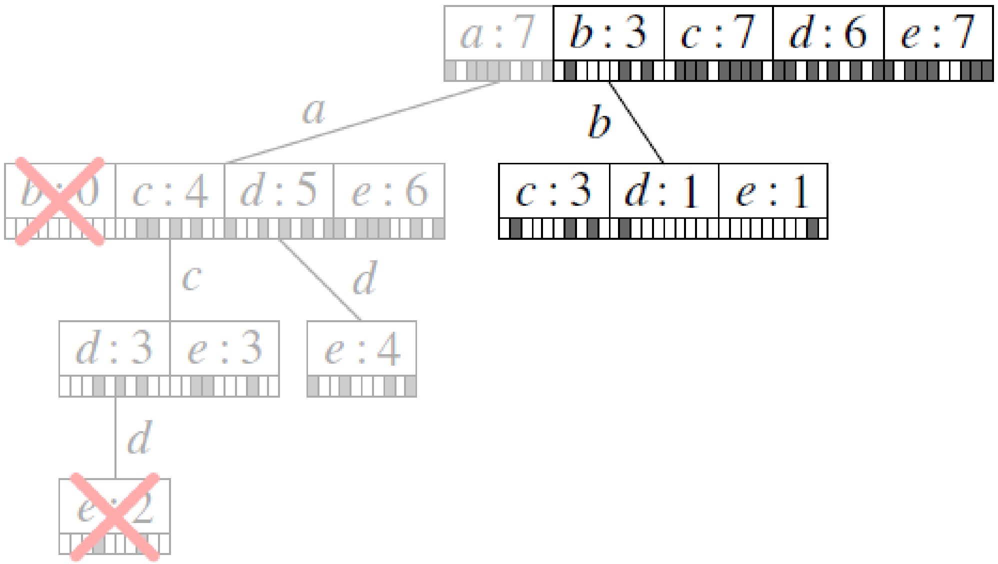
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

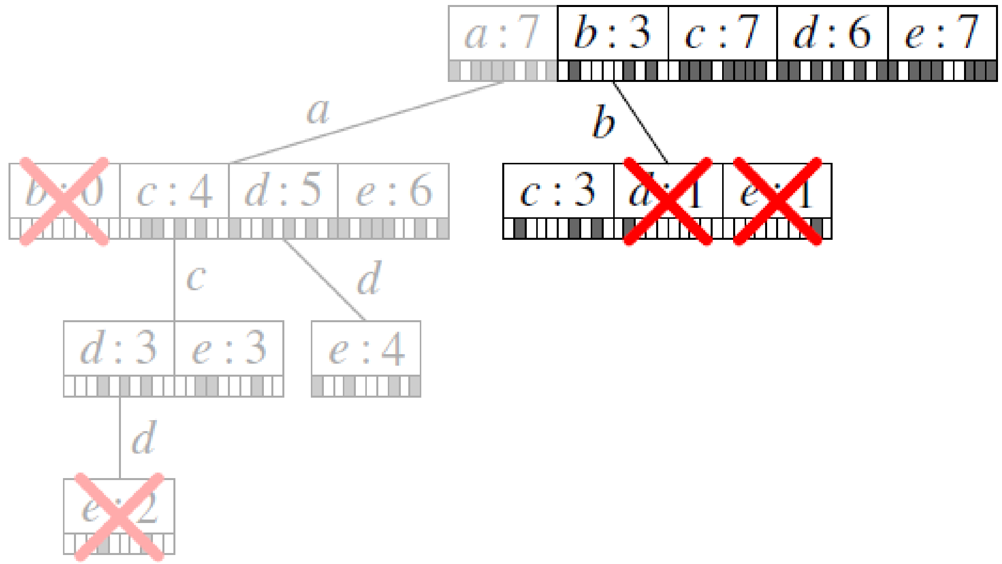
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

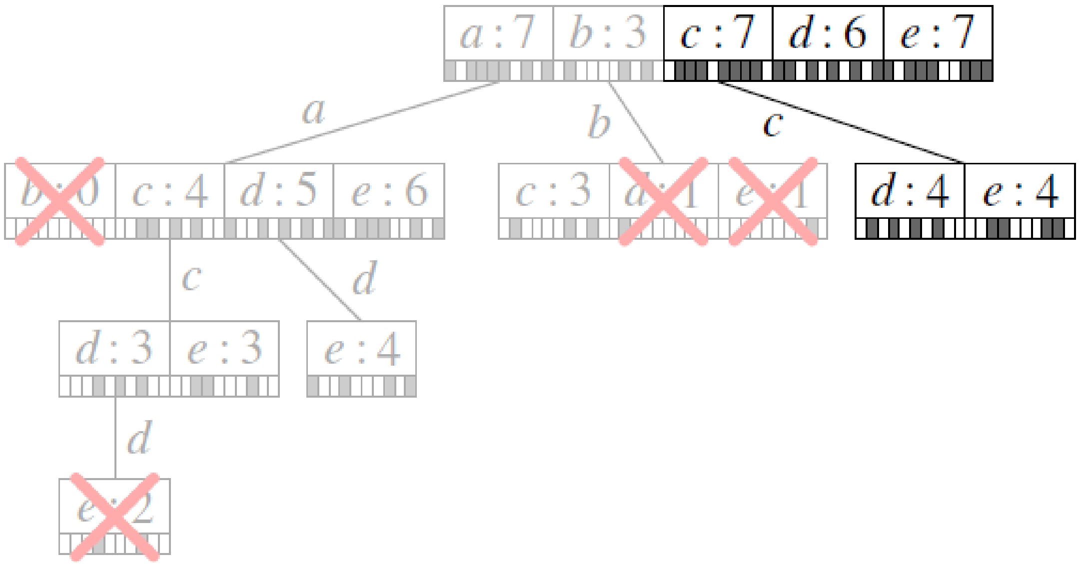
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

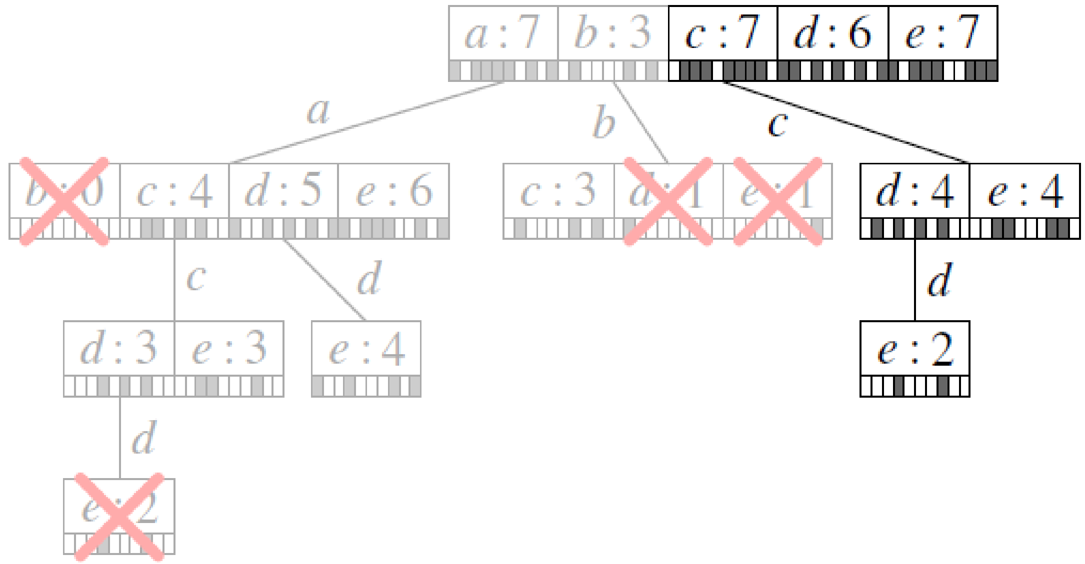
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

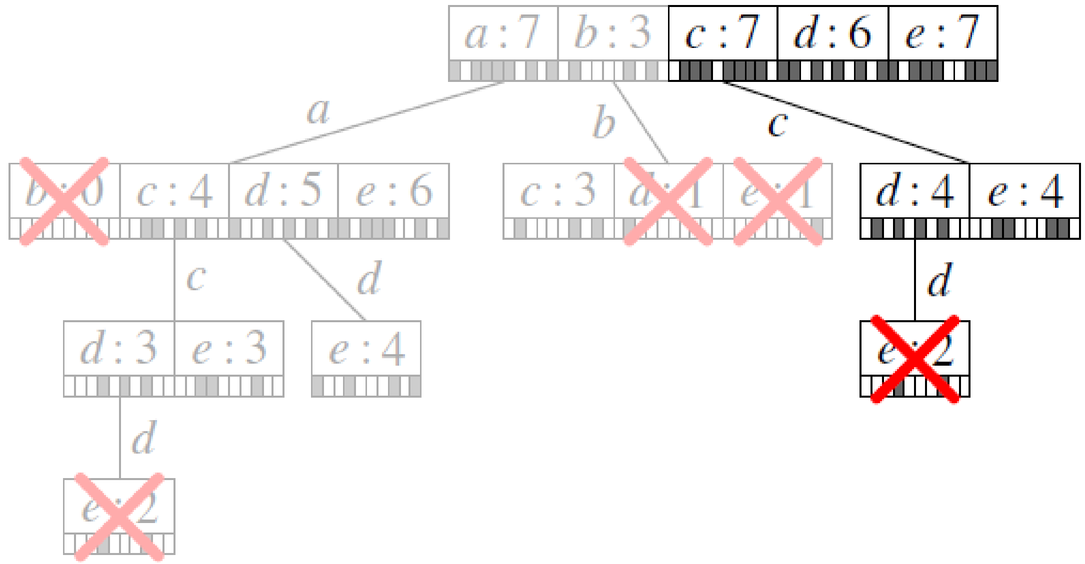
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

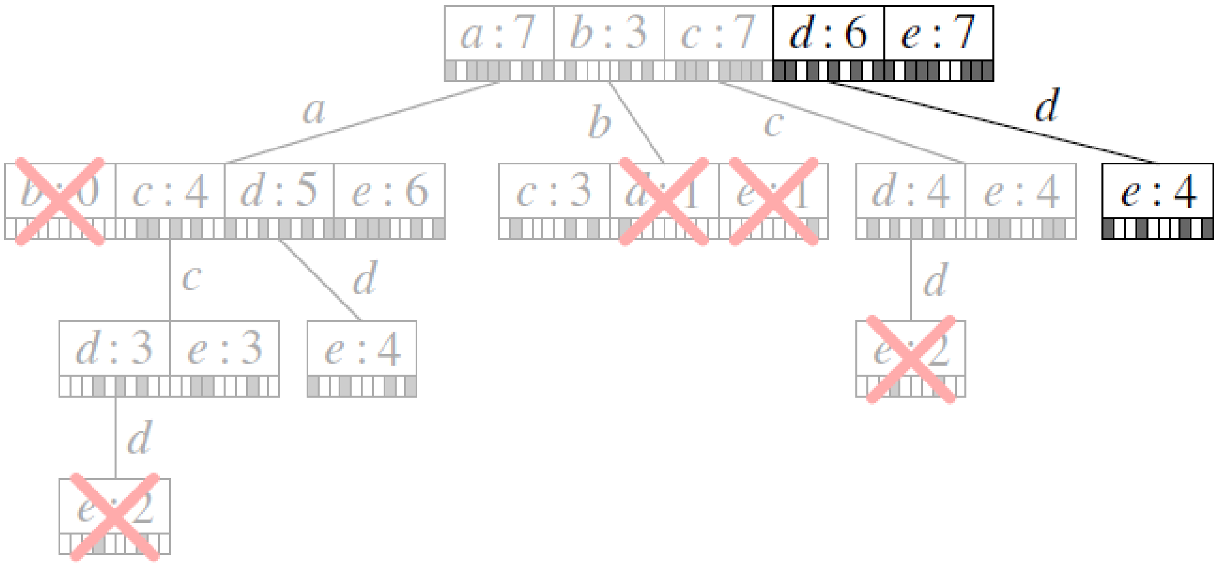
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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.

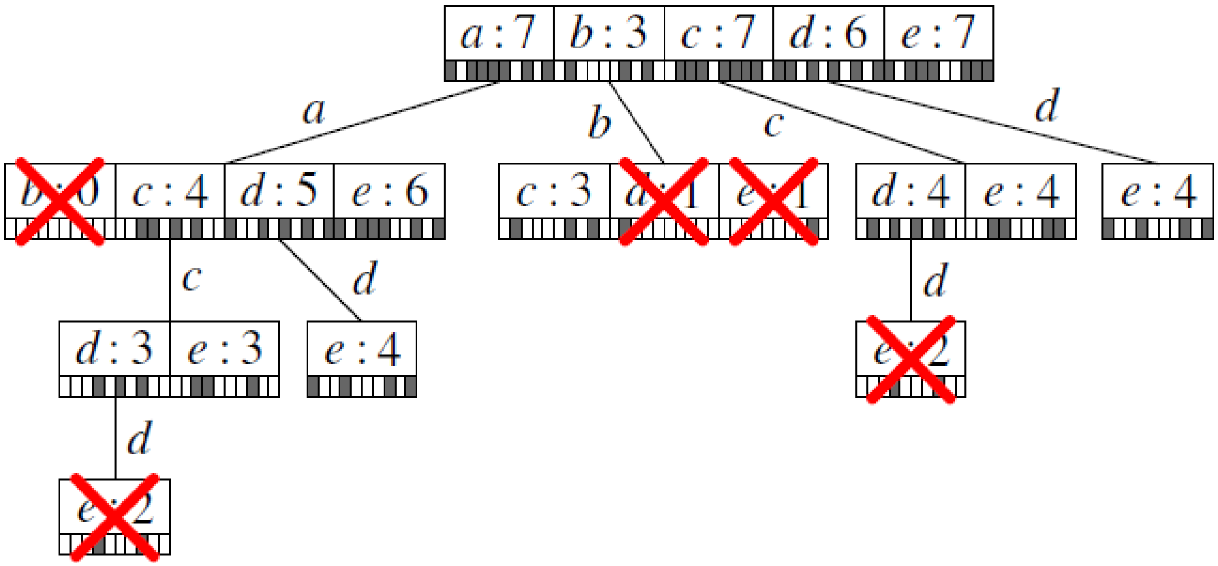
Eclat: Depth-First Search

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.

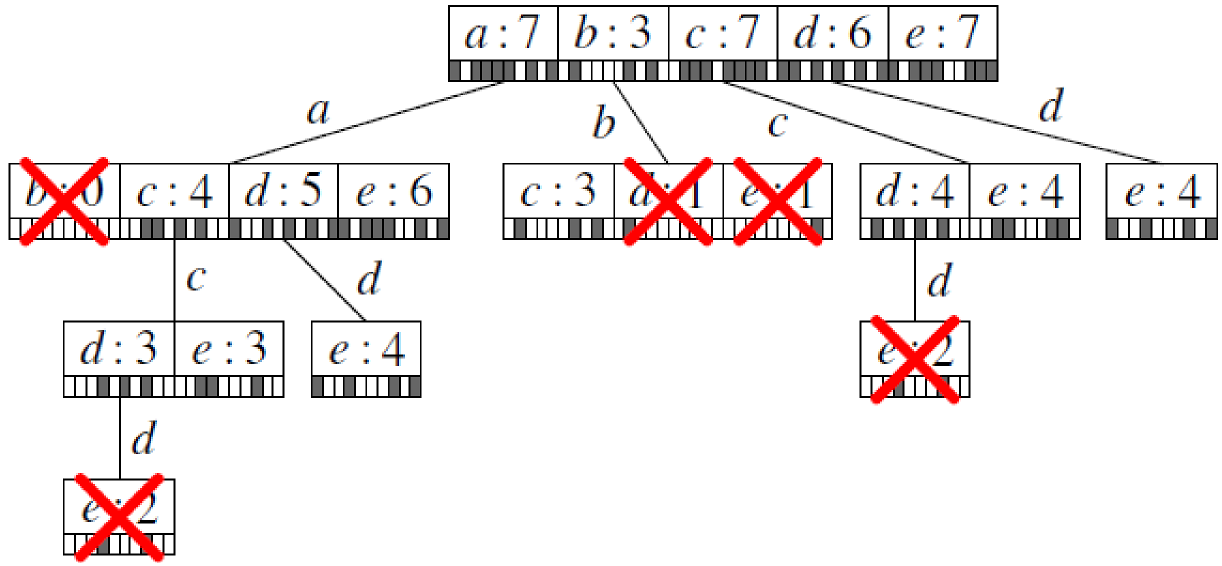
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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 L1 and L2 // like apriori
Transform T in vertical representation
CE2 = Decompose L2 in equivalence classes
forall E2∈CE2 do
  compute_frequent(E2)
end forall

return ∪kFk ;
```

compute_frequent(E_{k-1})

```
forall itemsets I1 and I2 in Ek-1 do
  if |tidlist(I1)∩tidlist(I2)| ≥ minsup then
      Lk ← Lk∪{I1∪I2}
  end if
end forall

CEk = Decompose Lk in equivalence classes
forall Ek∈CEk do
  compute_frequent(Ek)
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

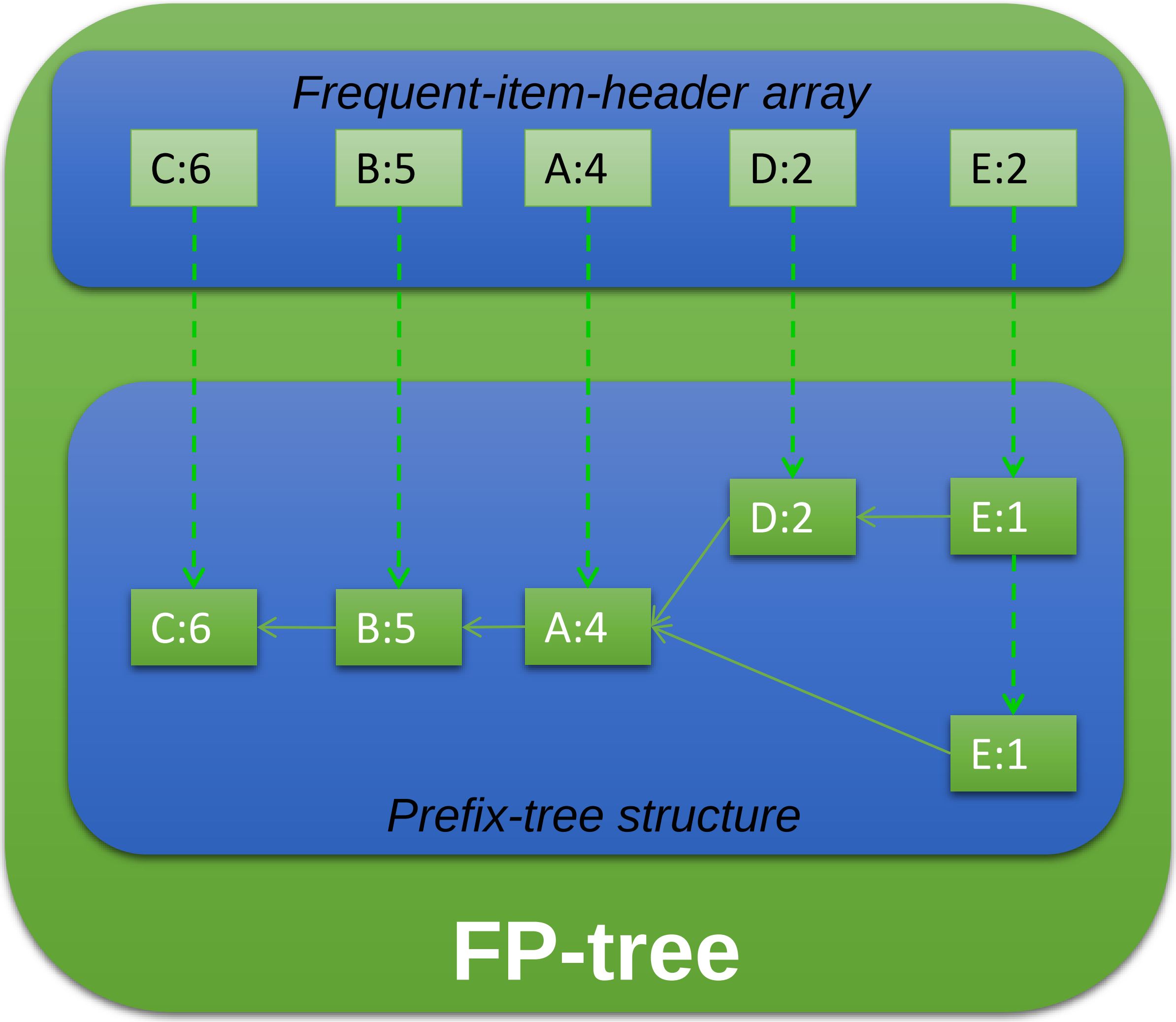
Transactions sorted lexicographically

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
$b\ c\ d$	a: 4	$d\ b\ c$	$d\ b\ a$	(see next slide)
$b\ c$	e: 3	$b\ c$	$d\ b\ e$	
$a\ b\ d$	f: 2	$d\ b\ a$	$d\ c$	
$b\ d\ e$	g: 1	$d\ b\ e$	$d\ c\ a\ e$	
$b\ c\ e\ g$		$b\ c\ e$	$d\ a$	
$c\ d\ f$		$d\ c$	$b\ c$	
$a\ b\ d$	$s_{\min} = 3$	$d\ b\ a$	$b\ c\ e$	

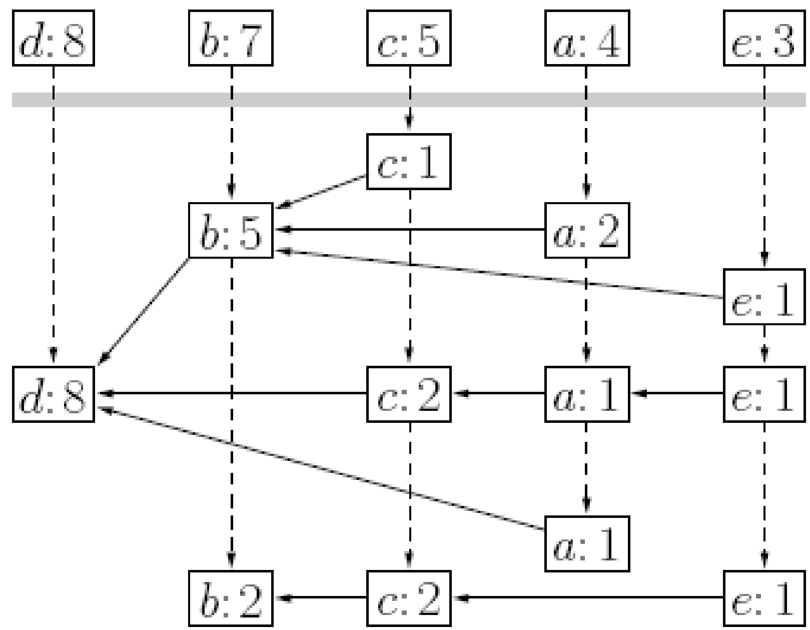
1. Original transaction database.
2. Frequency of individual items.
3. Items in transactions sorted descendingly w.r.t. their frequency and infrequent items removed.
4. Transactions sorted lexicographically in ascending order (comparison of items is the same as in preceding step).
5. Data structure used 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	$a\ d\ f$	4	$d\ b$
	$a\ c\ d\ e$		$d\ b\ c$
	$b\ d$		$d\ b\ a$
	$b\ c\ d$		$d\ b\ a$
	$b\ c$		$d\ b\ e$
	$a\ b\ d$		$d\ c$
	$b\ d\ e$		$d\ c\ a\ e$
	$b\ c\ e\ g$		$d\ a$
	$c\ d\ f$		$b\ c$
	$a\ b\ d$		$b\ c\ e$



FP-tree

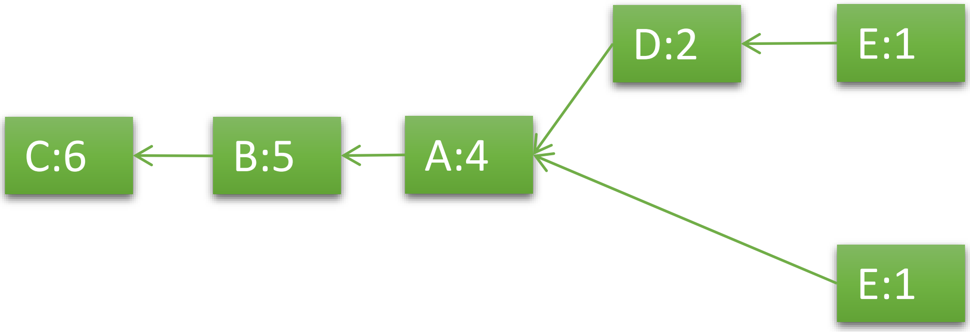
FP-Growth

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 $count_x \geq$ minsup then
    output (prefix ∪ x)
    FP_x = FP-tree constructed from D_x
    FP-growth(FP_x, prefix)
  end if
end

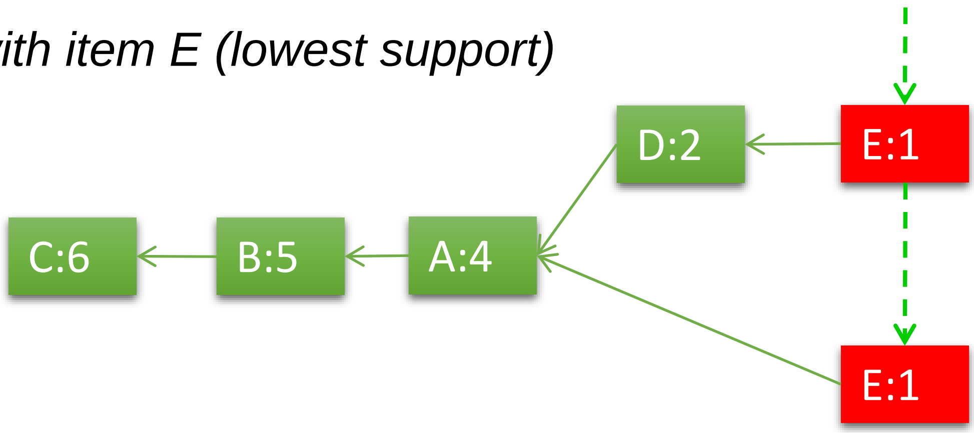
FP-Growth example

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



Original FP-tree

Start with item E (lowest support)

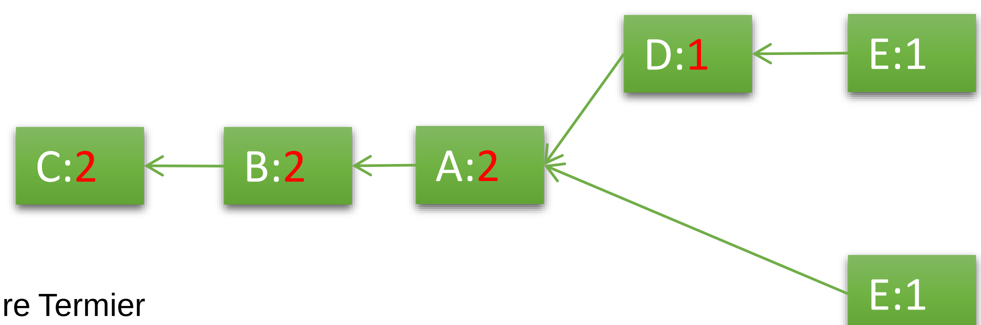


$count_E = 1+1 = 2$

$\Rightarrow$ E is frequent

$\Rightarrow$ Output E

Conditional FP-tree for E



- update counts
  $\rightarrow$ 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…

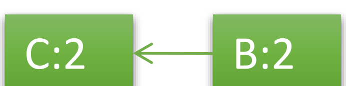
For AE :

count$_{AE}$= 2

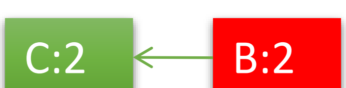
⇒ AE is frequent

⇒ Output AE

Conditional FP-tree for AE:



For BAE :



count$_{BAE}$= 2

⇒ BAE is frequent

⇒ Output BAE

Conditional FP-tree for BAE:

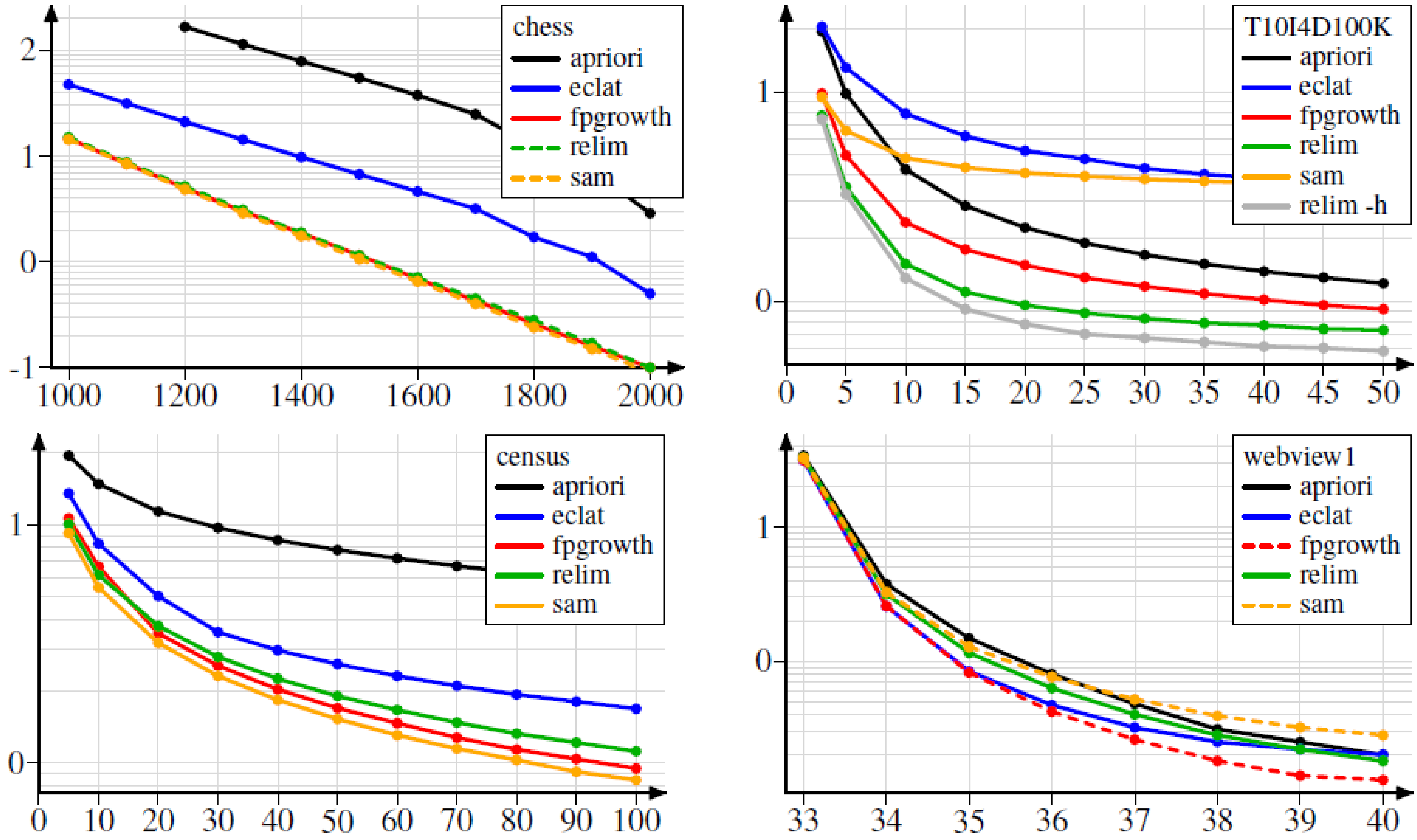
For CBAE :

count$_{CBAE}$= 2

⇒ CBAE is frequent

⇒ Output CBAE

Experiments: Execution Times



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