Some Data mining techniques

Overview

Frequent Pattern Mining comprises

- Frequent Item Set Mining and Association Rule Induction
- Frequent Sequence Mining
- Frequent Tree Mining
- Frequent Graph Mining

Application Areas of Frequent Pattern Mining include

- Market Basket Analysis
- Click Stream Analysis
- Web Link Analysis
- Genome Analysis
- Drug Design (Molecular Fragment Mining)

Frequent Item Set Mining

Frequent Item Set Mining: Motivation

- Frequent Item Set Mining is a method for **market basket analysis**.
- It aims at finding regularities in the shopping behavior of customers of supermarkets, mail-order companies, on-line shops etc.
- More specifically:
  Find sets of products that are frequently bought together.
- Possible applications of found frequent item sets:
  - Improve arrangement of products in shelves, on a catalog's pages etc.
  - Support cross-selling (suggestion of other products), product bundling.
  - Fraud detection, technical dependence analysis etc.
- Often found patterns are expressed as **association rules**, for example:

  If a customer buys **bread** and **wine**,
  then she/he will probably also buy **cheese**.

Frequent Item Set Mining: Basic Notions

- Let $B = \{i_1, \ldots, i_m\}$ be a set of **items**. This set is called the **item base**.

  Items may be products, special equipment items, service options etc.

- Any subset $I \subseteq B$ is called an **item set**.

  An item set may be any set of products that can be bought (together).

- Let $T = (t_1, \ldots, t_n)$ with $\forall k, 1 \leq k \leq n : t_k \subseteq B$ be a tuple of **transactions** over B. This tuple is called the **transaction database**.

  A transaction database can list, for example, the sets of products bought by the customers of a supermarket in a given period of time.

  Every transaction is an item set, but some item sets may not appear in T.

  Transactions need not be pairwise different: it may be $t_j = t_k$ for $j \neq k$.

  T may also be defined as a *bag* or *multiset* of transactions.

  The item base B may not be given explicitly, but only implicitly as $B = \bigcup_{k=1}^{n} t_k$.

Frequent Item Set Mining: Basic Notions

Let $I \subseteq B$ be an item set and T a transaction database over B.

- A transaction $t \in T$ **covers** the item set I or
  the item set I is **contained in** a transaction $t \in T$ iff $I \subseteq t$.

- The set $K_T(I) = \{k \in \{1, \ldots, n\} \mid I \subseteq t_k\}$ is called the **cover** of I w.r.t. T.

  The cover of an item set is the index set of the transactions that cover it.

  It may also be defined as a tuple of all transactions that cover it
  (which, however, is complicated to write in a formally correct way).

- The value $s_T(I) = |K_T(I)|$ is called the **(absolute) support** of I w.r.t. T.
  The value $\sigma_T(I) = \frac{1}{n}|K_T(I)|$ is called the **relative support** of I w.r.t. T.

  The support of I is the number or fraction of transactions that contain it.

  Sometimes $\sigma_T(I)$ is also called the *(relative) frequency* of I w.r.t. T.

Frequent Item Set Mining: Basic Notions

Alternative Definition of Transactions

- A **transaction** over an item base B is a pair $t = (\text{tid}, J)$, where
  - tid is a unique **transaction identifier** and
  - $J \subseteq B$ is an item set.
- A **transaction database** $T = \{t_1, \ldots, t_n\}$ is a *set* of transactions.

  A simple set can be used, because transactions differ at least in their identifier.
- A transaction $t = (\text{tid}, J)$ **covers** an item set I iff $I \subseteq J$.
- The set $K_T(I) = \{\text{tid} \mid \exists J \subseteq B : \exists t \in T : t = (\text{tid}, J) \wedge I \subseteq J\}$ is the **cover** of I w.r.t. T.

Remark: If the transaction database is defined as a tuple, there is an implicit transaction identifier, namely the position/index of the transaction in the tuple.

Frequent Item Set Mining: Formal Definition

Given:

- a set $B = \{i_1, \ldots, i_m\}$ of items, the **item base**,
- a tuple $T = (t_1, \ldots, t_n)$ of transactions over B, the **transaction database**,
- a number $s_{\min} \in \mathbb{N}$, $1 \leq s_{\min} \leq n$, or (equivalently)
  a number $\sigma_{\min} \in \mathbb{R}$, $0 < \sigma_{\min} \leq 1$, the **minimum support**.

Desired:

- the set of **frequent item sets**, that is,
  the set $F_T(s_{\min}) = \{I \subseteq B \mid s_T(I) \geq s_{\min}\}$ or (equivalently)
  the set $\Phi_T(\sigma_{\min}) = \{I \subseteq B \mid \sigma_T(I) \geq \sigma_{\min}\}$.

Note that with the relations $s_{\min} = \lceil n\sigma_{\min} \rceil$ and $\sigma_{\min} = \frac{1}{n} s_{\min}$
the two versions can easily be transformed into each other.

Frequent Item Sets: Example

transaction database

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: $\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

- In this example, the minimum support is $s_{\min} = 3$ or $\sigma_{\min} = 0.3 = 30\%$.
- There are $2^5 = 32$ possible item sets over $B = \{a, b, c, d, e\}$.
- There are 16 frequent item sets (but only 10 transactions).

Searching for Frequent Item Sets

Properties of the Support of Item Sets

- A **brute force approach** that traverses all possible item sets, determines their support, and discards infrequent item sets is usually **infeasible**:

  The number of possible item sets grows exponentially with the number of items. A typical supermarket offers (tens of) thousands of different products.

- **Idea:** Consider the properties of an item set's cover and support, in particular:

$$\forall I : \forall J \supseteq I : \quad K_T(J) \subseteq K_T(I).$$

  This property holds, since $\forall t : \forall I : \forall J \supseteq I : \quad J \subseteq t \Rightarrow I \subseteq t$.

  Each additional item is another condition that a transaction has to satisfy. Transactions that do not satisfy this condition are removed from the cover.

- It follows: $\forall I : \forall J \supseteq I : \quad s_T(J) \leq s_T(I).$

  That is: **If an item set is extended, its support cannot increase.**

  One also says that support is **anti-monotone** or **downward closed**.

Properties of the Support of Item Sets

- From $\forall I : \forall J \supseteq I : s_T(J) \leq s_T(I)$ it follows immediately

$$\forall s_{\min} : \forall I : \forall J \supseteq I : \quad s_T(I) < s_{\min} \;\Rightarrow\; s_T(J) < s_{\min}.$$

  That is: **No superset of an infrequent item set can be frequent.**

- This property is often referred to as the **Apriori Property**.

  Rationale: Sometimes we can know *a priori*, that is, before checking its support by accessing the given transaction database, that an item set cannot be frequent.

- Of course, the contraposition of this implication also holds:

$$\forall s_{\min} : \forall J : \forall I \subseteq J : \quad s_T(J) \geq s_{\min} \;\Rightarrow\; s_T(I) \geq s_{\min}.$$

  That is: **All subsets of a frequent item set are frequent.**

- This suggests a compressed representation of the set of frequent item sets (which will be explored later: maximal and closed frequent item sets).

Reminder: Partially Ordered Sets

- A **partial order** is a binary relation $\leq$ over a set S which satisfies $\forall a, b, c \in S$:
  - $a \leq a$ (reflexivity)
  - $a \leq b \wedge b \leq a \;\Rightarrow\; a = b$ (anti-symmetry)
  - $a \leq b \wedge b \leq c \;\Rightarrow\; a \leq c$ (transitivity)
- A set with a partial order is called a **partially ordered set** (or **poset** for short).
- Let a and b be two distinct elements of a partially ordered set $(S, \leq)$.
  - if $a \leq b$ or $b \leq a$, then a and b are called **comparable**.
  - if neither $a \leq b$ nor $b \leq a$, then a and b are called **incomparable**.
- If all pairs of elements of the underlying set S are comparable, the order $\leq$ is called a **total order** or a **linear order**.
- In a total order the reflexivity axiom is replaced by the stronger axiom:
  - $a \leq b \vee b \leq a$ (totality)

Properties of the Support of Item Sets

Monotonicity in Calculus and Mathematical Analysis

- A function $f : \mathbb{R} \to \mathbb{R}$ is called **monotonically non-decreasing** if $\forall x, y: \quad x \leq y \;\Rightarrow\; f(x) \leq f(y)$.
- A function $f : \mathbb{R} \to \mathbb{R}$ is called **monotonically non-increasing** if $\forall x, y: \quad x \leq y \;\Rightarrow\; f(x) \geq f(y)$.

Monotonicity in Order Theory

- Order theory is concerned with arbitrary (partially) ordered sets.
  The terms *increasing* and *decreasing* are avoided, because they lose their pictorial motivation as soon as sets are considered that are not totally ordered.
- A function $f : S \to R$, where S and R are two partially ordered sets, is called **monotone** or **order-preserving** if $\forall x, y \in S: \; x \leq_S y \;\Rightarrow f(x) \leq_R f(y)$.
- A function $f : S \to R$ is called **anti-monotone** or **order-reversing** if $\forall x, y \in S: \; x \leq_S y \;\Rightarrow f(x) \geq_R f(y)$.
- In this sense the **support** of item sets **is anti-monotone**.

Properties of Frequent Item Sets

- A subset R of a partially ordered set $(S, \leq)$ is called **downward closed** if for any element of the set all smaller elements are also in it:

  $\forall x \in R\colon\ \forall y \in S\colon \quad y \leq x \quad \Rightarrow \quad y \in R$

  In this case the subset R is also called a **lower set**.

- The notions of **upward closed** and **upper set** are defined analogously.

- For every $s_{\min}$ the set of frequent item sets $F_T(s_{\min})$ is downward closed w.r.t. the partially ordered set $(2^B, \subseteq)$, where 2^B denotes the powerset of B:

  $\forall s_{\min}\colon\ \forall X \in F_T(s_{\min})\colon\ \forall Y \subseteq B\colon \quad Y \subseteq X \quad \Rightarrow \quad Y \in F_T(s_{\min}).$

- Since the set of frequent item sets is induced by the support function, the notions of **up-** or **downward closed** are transferred to the support function:

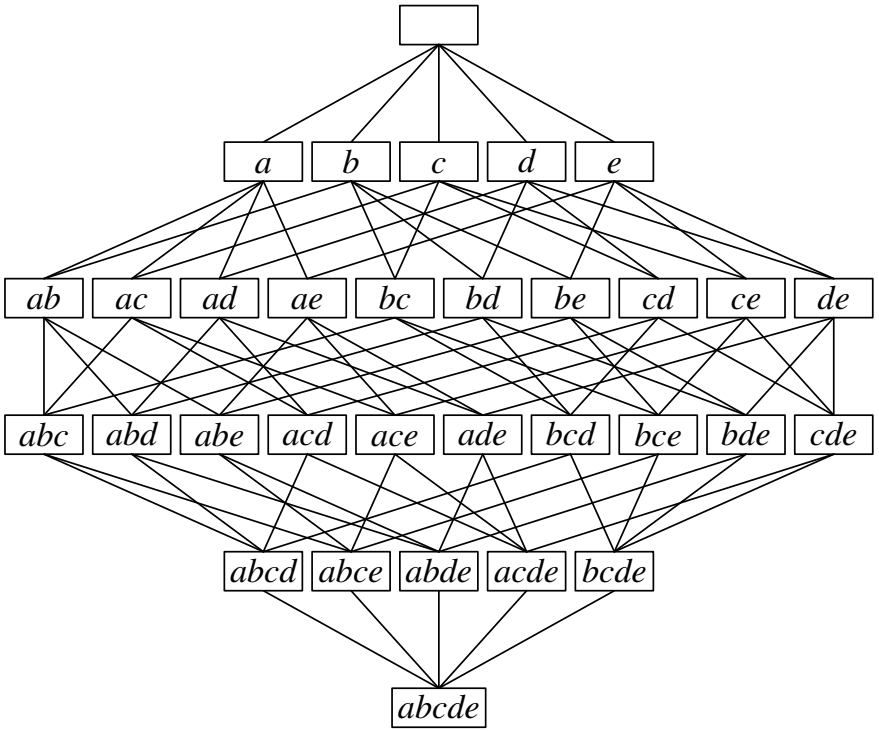
  Any set of item sets induced by a support threshold $s_{\min}$ is up- or downward closed.

  $F_T(s_{\min}) = \{S \subseteq B \mid s_T(S) \geq s_{\min}\}$ (frequent item sets) is downward closed,
  $G_T(s_{\min}) = \{S \subseteq B \mid s_T(S) < s_{\min}\}$ (infrequent item sets) is upward closed.

Reminder: Partially Ordered Sets and Hasse Diagrams

- A finite partially ordered set $(S, \leq)$ can be depicted as a (directed) acyclic graph G, which is called **Hasse diagram**.
- G has the elements of S as vertices. The edges are selected according to:

  If x and y are elements of S with $x < y$ (that is, $x \leq y$ and not $x = y$) and there is no element between x and y (that is, no $z \in S$ with $x < z < y$), then there is an edge from x to y.
- Since the graph is acyclic (there is no directed cycle), the graph can always be depicted such that all edges lead downward.
- The Hasse diagram of a total order (or linear order) is a chain.



Hasse diagram of $(2^{\{a,b,c,d,e\}}, \subseteq)$.

(Edge directions are omitted; all edges lead downward.)

Searching for Frequent Item Sets

- The standard search procedure is an **enumeration approach**,
  that enumerates candidate item sets and checks their support.
- It improves over the brute force approach by exploiting the **apriori property**
  to skip item sets that cannot be frequent because they have an infrequent subset.
- The **search space** is the **partially ordered set** $(2^B, \subseteq)$.
- The structure of the partially ordered set $(2^B, \subseteq)$ helps to identify
  those item sets that can be skipped due to the apriori property.
  $\Rightarrow$ **top-down search** (from empty set/one-element sets to larger sets)
- Since a partially ordered set can conveniently be depicted by a **Hasse diagram**,
  we will use such diagrams to illustrate the search.
- Note that the search may have to visit an exponential number of item sets.
  In practice, however, the search times are often bearable,
  at least if the minimum support is not chosen too low.

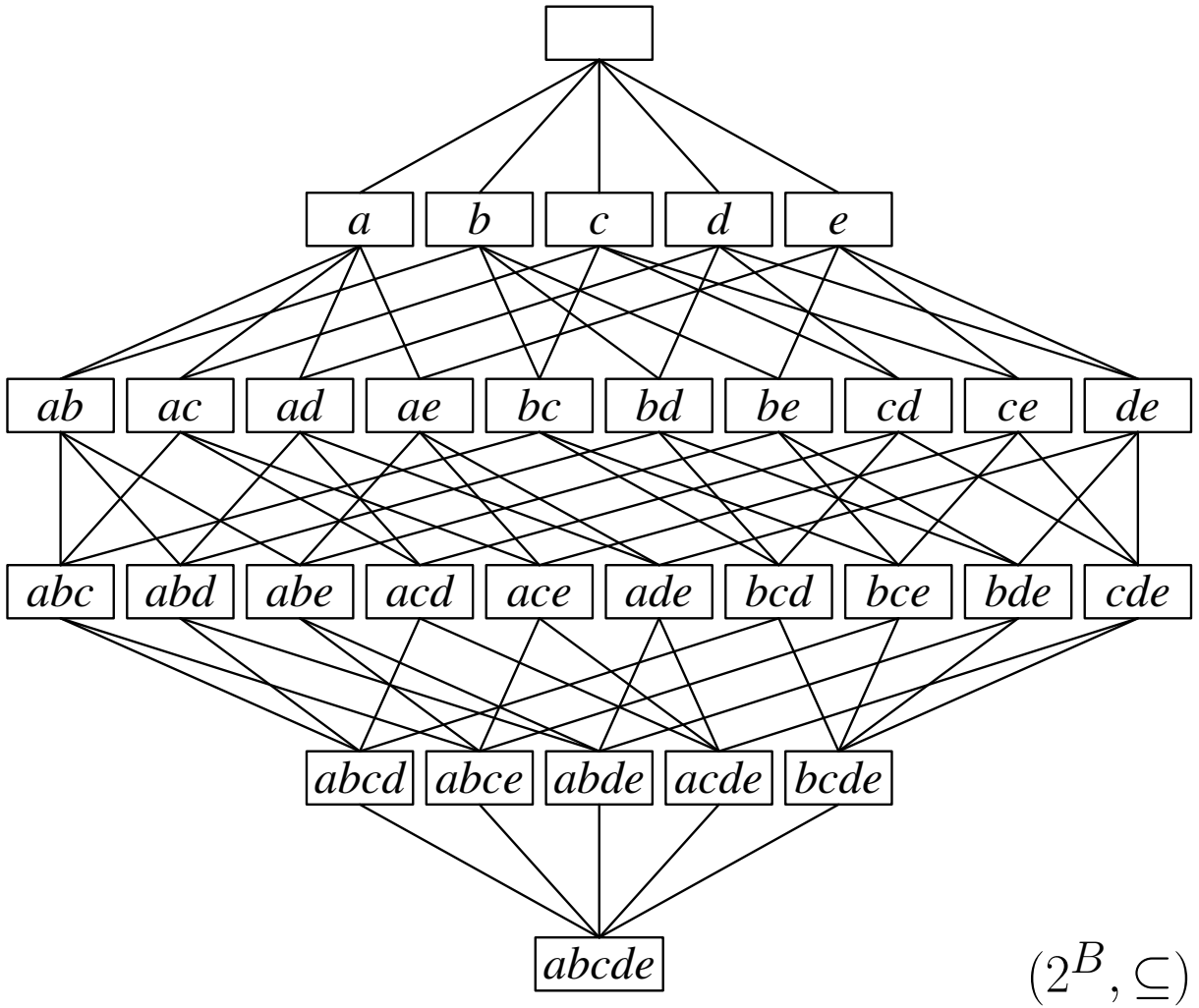
Searching for Frequent Item Sets

Idea: Use the properties of the support to organize the search for all frequent item sets, especially the **apriori property**:

$$\forall I : \forall J \supset I : \\ s_T(I) < s_{\min} \\ \Rightarrow \quad s_T(J) < s_{\min}.$$

Since these properties relate the support of an item set to the support of its **subsets** and **supersets**, it is reasonable to organize the search based on the structure of the **partially ordered set** $(2^B, \subseteq)$.

Hasse diagram for five items $\{a, b, c, d, e\} = B$:



$(2^B, \subseteq)$

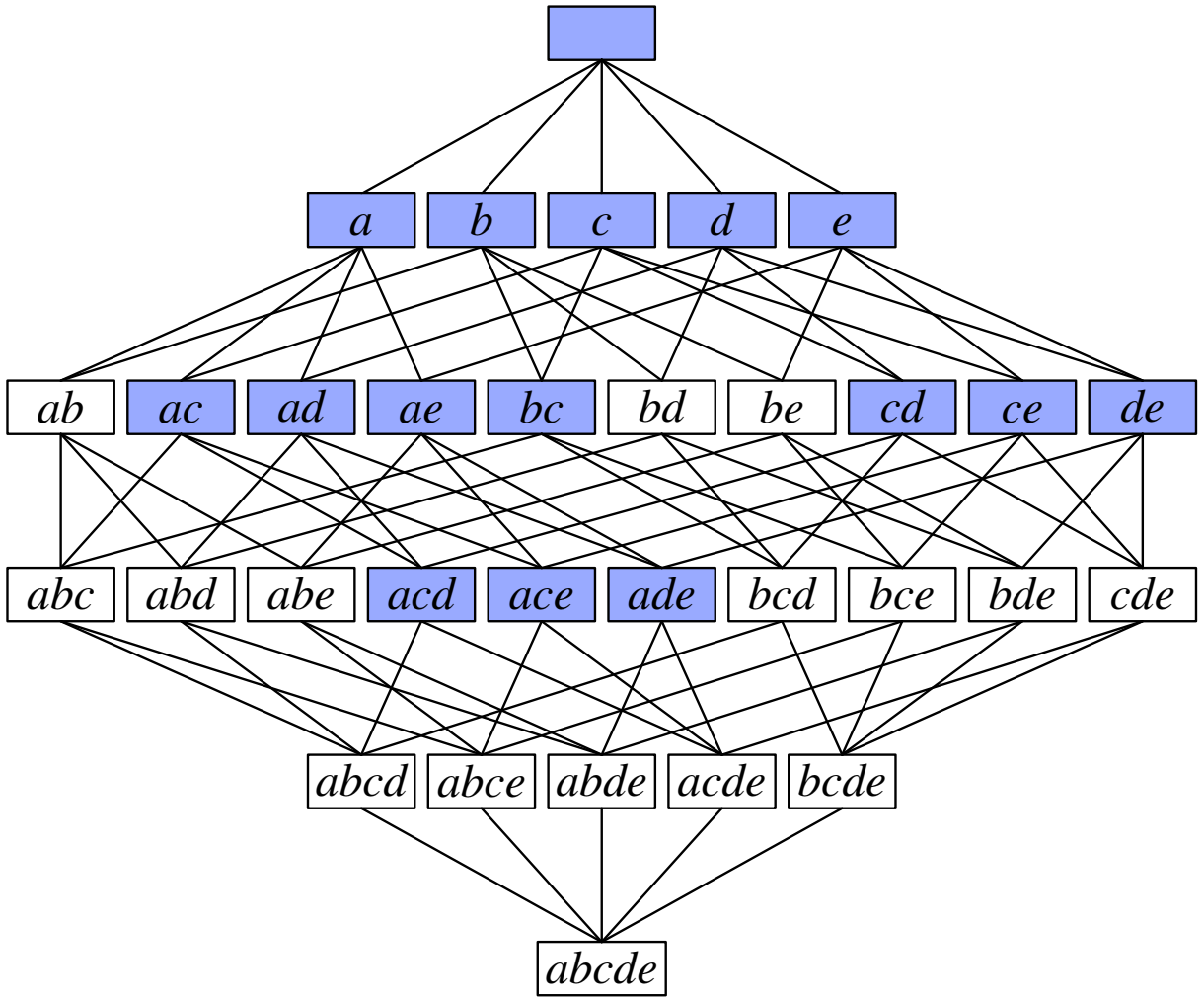
Hasse Diagrams and Frequent Item Sets

transaction database

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: $\{a, d, e\}$

Blue boxes are frequent item sets, white boxes infrequent item sets.

Hasse diagram with frequent item sets ($s_{\min} = 3$):



The Apriori Algorithm

[Agrawal and Srikant 1994]

Searching for Frequent Item Sets

Possible scheme for the search:

- Determine the support of the one-element item sets (a.k.a. singletons) and discard the infrequent items / item sets.
- Form candidate item sets with two items (both items must be frequent), determine their support, and discard the infrequent item sets.
- Form candidate item sets with three items (all contained pairs must be frequent), determine their support, and discard the infrequent item sets.
- Continue by forming candidate item sets with four, five etc. items until no candidate item set is frequent.

This is the general scheme of the **Apriori Algorithm**.

It is based on two main steps: **candidate generation** and **pruning**.

All enumeration algorithms are based on these two steps in some form.

The Apriori Algorithm 1

function apriori $(B, T, s_{\min})$
begin $(*$ — Apriori algorithm $*)$
$k := 1;$ $(*$ initialize the item set size $*)$
$E_k := \bigcup_{i \in B} \{\{i\}\};$ $(*$ start with single element sets $*)$
$F_k := \text{prune}(E_k, T, s_{\min});$ $(*$ and determine the frequent ones $*)$
while $F_k \neq \emptyset$ **do begin** $(*$ while there are frequent item sets $*)$
$E_{k+1} := \text{candidates}(F_k);$ $(*$ create candidates with one item more $*)$
$F_{k+1} := \text{prune}(E_{k+1}, T, s_{\min});$ $(*$ and determine the frequent item sets $*)$
$k := k + 1;$ $(*$ increment the item counter $*)$
end;
return $\bigcup_{j=1}^{k} F_j;$ $(*$ return the frequent item sets $*)$
end $(*$ apriori $*)$

E_j: candidate item sets of size j, F_j: frequent item sets of size j.

The Apriori Algorithm 2

function candidates (F_k)
begin (∗ — generate candidates with $k+1$ items ∗)
  $E := \emptyset$; (∗ initialize the set of candidates ∗)
  forall $f_1, f_2 \in F_k$ (∗ traverse all pairs of frequent item sets ∗)
  with $f_1 = \{i_1, \ldots, i_{k-1}, i_k\}$ (∗ that differ only in one item and ∗)
  and $f_2 = \{i_1, \ldots, i_{k-1}, i'_k\}$ (∗ are in a lexicographic order ∗)
  and $i_k < i'_k$ **do begin** (∗ (this order is arbitrary, but fixed) ∗)
    $f := f_1 \cup f_2 = \{i_1, \ldots, i_{k-1}, i_k, i'_k\}$; (∗ union has $k+1$ items ∗)
    if $\forall i \in f : \; f - \{i\} \in F_k$ (∗ if all subsets with k items are frequent, ∗)
    then $E := E \cup \{f\}$; (∗ add the new item set to the candidates ∗)
  end; (∗ (otherwise it cannot be frequent) ∗)
  return E; (∗ return the generated candidates ∗)
end (∗ candidates ∗)

The Apriori Algorithm 3

function prune $(E, T, s_{\min})$
begin (∗ — prune infrequent candidates ∗)
  forall $e \in E$ **do** (∗ initialize the support counters ∗)
    $s_T(e) := 0$; (∗ of all candidates to be checked ∗)
  forall $t \in T$ **do** (∗ traverse the transactions ∗)
    forall $e \in E$ **do** (∗ traverse the candidates ∗)
      if $e \subseteq t$ (∗ if the transaction contains the candidate, ∗)
      then $s_T(e) := s_T(e) + 1$; (∗ increment the support counter ∗)
  $F := \emptyset$; (∗ initialize the set of frequent candidates ∗)
  forall $e \in E$ **do** (∗ traverse the candidates ∗)
    if $s_T(e) \geq s_{\min}$ (∗ if a candidate is frequent, ∗)
    then $F := F \cup \{e\}$; (∗ add it to the set of frequent item sets ∗)
  return F; (∗ return the pruned set of candidates ∗)
end (∗ prune ∗)

Improving the Candidate Generation

Searching for Frequent Item Sets

- The Apriori algorithm searches the partial order top-down level by level.
- Collecting the frequent item sets of size k in a *set* F_k has drawbacks:
  A frequent item set of size $k+1$ can be formed in

$$j = \frac{k(k+1)}{2}$$

  possible ways. (For infrequent item sets the number may be smaller.)

  As a consequence, the candidate generation step may carry out a lot of redundant work, since it suffices to generate each candidate item set once.

- **Question:** Can we reduce or even eliminate this redundant work?

  More generally:
  How can we make sure that any candidate item set is generated at most once?

- **Idea:** Assign to each item set a unique parent item set, from which this item set is to be generated.

Searching for Frequent Item Sets

- A core problem is that an item set of size k (that is, with k items) can be generated in $k!$ different ways (on $k!$ paths in the Hasse diagram), because in principle the items may be added in any order.

- If we consider an item by item process of building an item set (which can be imagined as a levelwise traversal of the partial order), there are k possible ways of forming an item set of size k from item sets of size $k-1$ by adding the remaining item.

- It is obvious that it suffices to consider each item set at most once in order to find the frequent ones (infrequent item sets need not be generated at all).

- **Question:** Can we reduce or even eliminate this variety?

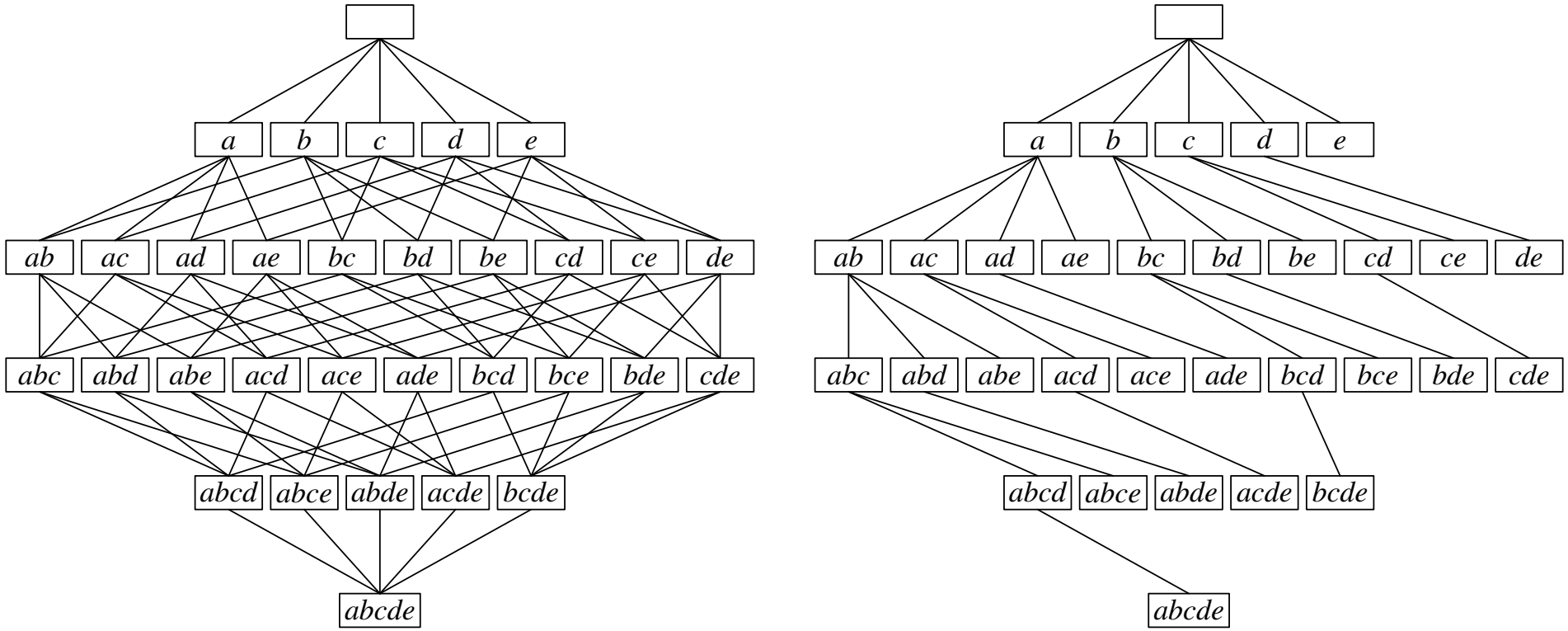
  More generally:
  How can we make sure that any candidate item set is generated at most once?

- **Idea:** Assign to each item set a unique parent item set, from which this item set is to be generated.

Searching for Frequent Item Sets

- We have to search the partially ordered set $(2^B, \subseteq)$ or its Hasse diagram.
- Assigning unique parents turns the Hasse diagram into a tree.
- Traversing the resulting tree explores each item set exactly once.

Hasse diagram and a possible tree for five items:



Searching with Unique Parents

Principle of a Search Algorithm based on Unique Parents:

- **Base Loop:**
  - Traverse all one-element item sets (their unique parent is the empty set).
  - Recursively process all one-element item sets that are frequent.
- **Recursive Processing:**

  For a given frequent item set I:
  - Generate all extensions J of I by one item (that is, $J \supset I$, $|J| = |I| + 1$) for which the item set I is the chosen unique parent.
  - For all J: if J is frequent, process J recursively, otherwise discard J.
- **Questions:**
  - How can we formally assign unique parents?
  - How can we make sure that we generate only those extensions for which the item set that is extended is the chosen unique parent?

Assigning Unique Parents

- Formally, the set of all **possible/candidate parents** of an item set I is

$$\Pi(I) = \{J \subset I \mid \nexists K : J \subset K \subset I\}.$$

  In other words, the possible parents of I are its *maximal proper subsets*.

- In order to single out one element of $\Pi(I)$, the **canonical parent** $\pi_c(I)$, we can simply define an (arbitrary, but fixed) global order of the items:

$$i_1 < i_2 < i_3 < \cdots < i_n.$$

  Then the canonical parent of an item set I can be defined as the item set

$$\pi_c(I) = I - \{\max_{i \in I} i\} \qquad \text{(or} \quad \pi_c(I) = I - \{\min_{i \in I} i\}),$$

  where the maximum (or minimum) is taken w.r.t. the chosen order of the items.

- Even though this approach is straightforward and simple, we reformulate it now in terms of a **canonical form** of an item set, in order to lay the foundations for the study of frequent (sub)graph mining.

Canonical Forms of Item Sets

Canonical Forms

The meaning of the word "canonical":

(source: Oxford Advanced Learner's Dictionary — Encyclopedic Edition)

canon /ˈkænən/ *n* **1** general rule, standard or principle, by which sth is judged: *This film offends against all the canons of good taste. ...*

canonical /kəˈnɒnɪkl/ *adj* ... **3** standard; accepted. ...

- A **canonical form** of something is a standard representation of it.
- The canonical form must be unique (otherwise it could not be standard).

  Nevertheless there are often several possible choices for a canonical form. However, one must fix one of them for a given application.
- In the following we will define a standard representation of an item set, and later standard representations of a graph, a sequence, a tree etc.
- This canonical form will be used to assign unique parents to all item sets.

A Canonical Form for Item Sets

- An item set is represented by a **code word**; each letter represents an item.

  The code word is a word over the alphabet B, the item base.

- There are $k!$ possible code words for an item set of size k,
  because the items may be listed in any order.

- By introducing an (arbitrary, but fixed) **order of the items**,
  and by comparing code words lexicographically w.r.t. this order,
  we can define an order on these code words.

  Example: $abc < bac < bca < cab$ etc. for the item set $\{a, b, c\}$ and $a < b < c$.

- The lexicographically smallest (or, alternatively, greatest) code word
  for an item set is defined to be its **canonical code word**.

  Obviously the canonical code word lists the items in the chosen, fixed order.

Remark: These explanations may appear obfuscated, since the core idea and the result are very simple. However, the view developed here will help us a lot when we turn to frequent (sub)graph mining.

Canonical Forms and Canonical Parents

- Let I be an item set and $w_c(I)$ its canonical code word.

  The **canonical parent** $\pi_c(I)$ of the item set I is the item set described by the **longest proper prefix** of the code word $w_c(I)$.

- Since the canonical code word of an item set lists its items in the chosen order, this definition is equivalent to

$$\pi_c(I) = I - \{\max_{i \in I} i\}.$$

- **General Recursive Processing with Canonical Forms:**

  For a given frequent item set I:

  - Generate all possible extensions J of I by one item ($J \supset I$, $|J| = |I| + 1$).
  - Form the canonical code word $w_c(J)$ of each extended item set J.
  - For each J: if the last letter of $w_c(J)$ is the item added to I to form J and J is frequent, process J recursively, otherwise discard J.

The Prefix Property

- Note that the considered item set coding scheme has the **prefix property**:

  The longest proper prefix of the canonical code word of any item set is a canonical code word itself.

$\Rightarrow$ With the longest proper prefix of the canonical code word of an item set I we not only know the canonical parent of I, but also its canonical code word.

- Example: Consider the item set $I = \{a, b, d, e\}$:
  - The canonical code word of I is $abde$.
  - The longest proper prefix of $abde$ is abd.
  - The code word abd is the canonical code word of $\pi_c(I) = \{a, b, d\}$.

- Note that the prefix property immediately implies:

  Every prefix of a canonical code word is a canonical code word itself.

(In the following both statements are called the **prefix property**, since they are obviously equivalent.)

Searching with the Prefix Property

The prefix property allows us to **simplify the search scheme**:

- The general recursive processing scheme with canonical forms requires to construct the **canonical code word** of each created item set in order to decide whether it has to be processed recursively or not.

$\Rightarrow$ We know the canonical code word of every item set that is processed recursively.

- With this code word we know, due to the **prefix property**, the canonical code words of all child item sets that have to be explored in the recursion *with the exception of the last letter* (that is, the added item).

$\Rightarrow$ We only have to check whether the code word that results from appending the added item to the given canonical code word is canonical or not.

- **Advantage:**
  Checking whether a given code word is canonical can be simpler/faster than constructing a canonical code word from scratch.

Searching with the Prefix Property

Principle of a Search Algorithm based on the Prefix Property:

- **Base Loop:**
  - Traverse all possible items, that is,
    the canonical code words of all one-element item sets.
  - Recursively process each code word that describes a frequent item set.
- **Recursive Processing:**

  For a given (canonical) code word of a frequent item set:
  - Generate all possible extensions by one item.
    This is done by simply **appending the item** to the code word.
  - Check whether the extended code word is the **canonical code word**
    of the item set that is described by the extended code word
    (and, of course, whether the described item set is frequent).

    If it is, process the extended code word recursively, otherwise discard it.

Searching with the Prefix Property: Examples

- Suppose the item base is $B = \{a, b, c, d, e\}$ and let us assume that we simply use the alphabetical order to define a canonical form (as before).

- Consider the recursive processing of the code word acd (this code word is canonical, because its letters are in alphabetical order):
  - Since acd contains neither b nor e, its extensions are $acdb$ and $acde$.
  - The code word $acdb$ is not canonical and thus it is discarded (because $d > b$ — note that it suffices to compare the last two letters)
  - The code word $acde$ is canonical and therefore it is processed recursively.

- Consider the recursive processing of the code word bc:
  - The extended code words are bca, bcd and bce.
  - bca is not canonical and thus discarded.
    bcd and bce are canonical and therefore processed recursively.

Searching with the Prefix Property

Exhaustive Search

- The **prefix property** is a necessary condition for ensuring that all canonical code words can be constructed in the search by appending extensions (items) to visited canonical code words.

- Suppose the prefix property would not hold. Then:
  - There exist a canonical code word w and a (proper) prefix v of w, such that v is not a canonical code word.
  - Forming w by repeatedly appending items must form v first (otherwise the prefix would differ).
  - When v is constructed in the search, it is discarded, because it is not canonical.
  - As a consequence, the canonical code word w can never be reached.

$\Rightarrow$ The simplified search scheme can be exhaustive only if the prefix property holds.

Searching with Canonical Forms

Straightforward Improvement of the Extension Step:

- The considered canonical form lists the items in the chosen item order.

$\Rightarrow$ If the added item succeeds all already present items in the chosen order, the result is in canonical form.

$\wedge$ If the added item precedes any of the already present items in the chosen order, the result is not in canonical form.

- As a consequence, we have a very simple **canonical extension rule** (that is, a rule that generates all children and only canonical code words).

- Applied to the Apriori algorithm, this means that we generate candidates of size $k+1$ by combining two frequent item sets $f_1 = \{i_1, \ldots, i_{k-1}, i_k\}$ and $f_2 = \{i_1, \ldots, i_{k-1}, i'_k\}$ only if $i_k < i'_k$ and $\forall j, 1 \leq j < k : i_j < i_{j+1}$.

  Note that it suffices to compare the last letters/items i_k and i'_k if all frequent item sets are represented by canonical code words.

Searching with Canonical Forms

Final Search Algorithm based on Canonical Forms:

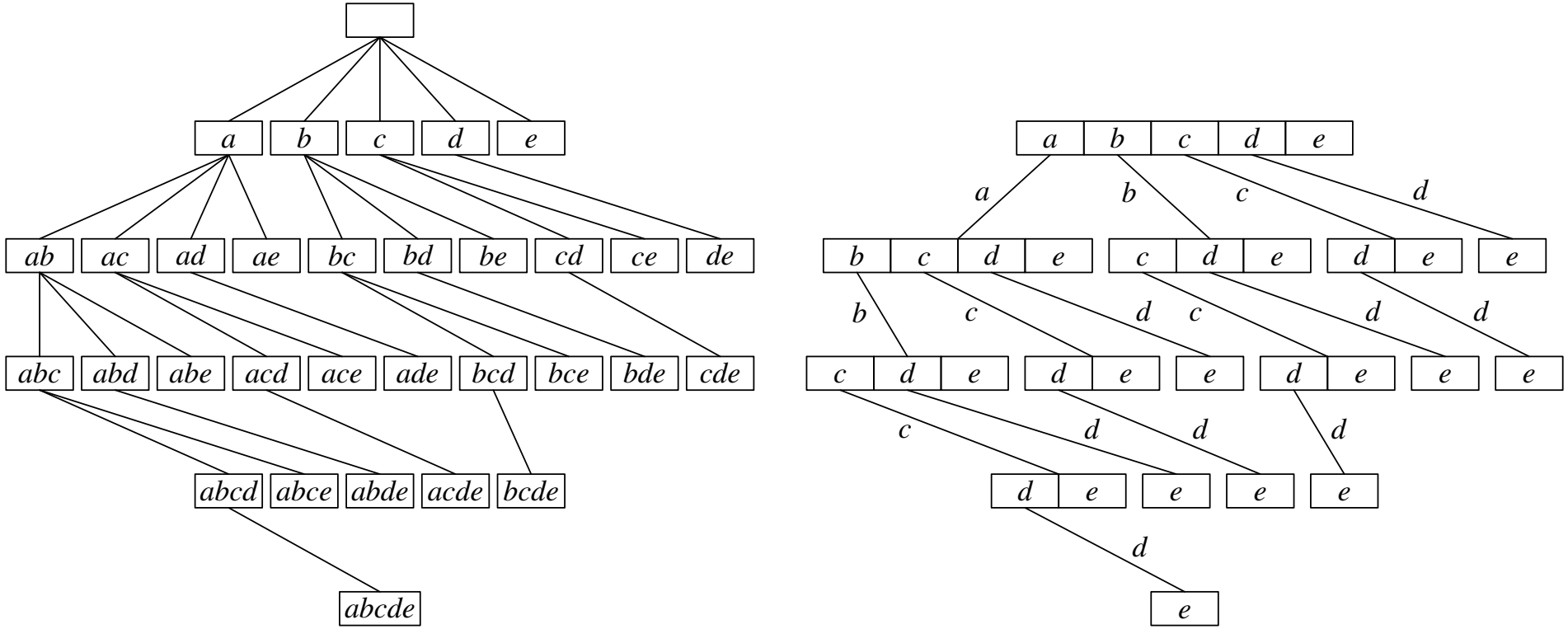
- **Base Loop:**
  - Traverse all possible items, that is,
    the canonical code words of all one-element item sets.
  - Recursively process each code word that describes a frequent item set.
- **Recursive Processing:**

  For a given (canonical) code word of a frequent item set:
  - Generate all possible extensions by a single item,
    where this item succeeds the last letter (item) of the given code word.
    This is done by simply **appending the item** to the code word.
  - If the item set described by the resulting extended code word is frequent,
    process the code word recursively, otherwise discard it.
- This search scheme generates each candidate item set **at most once**.

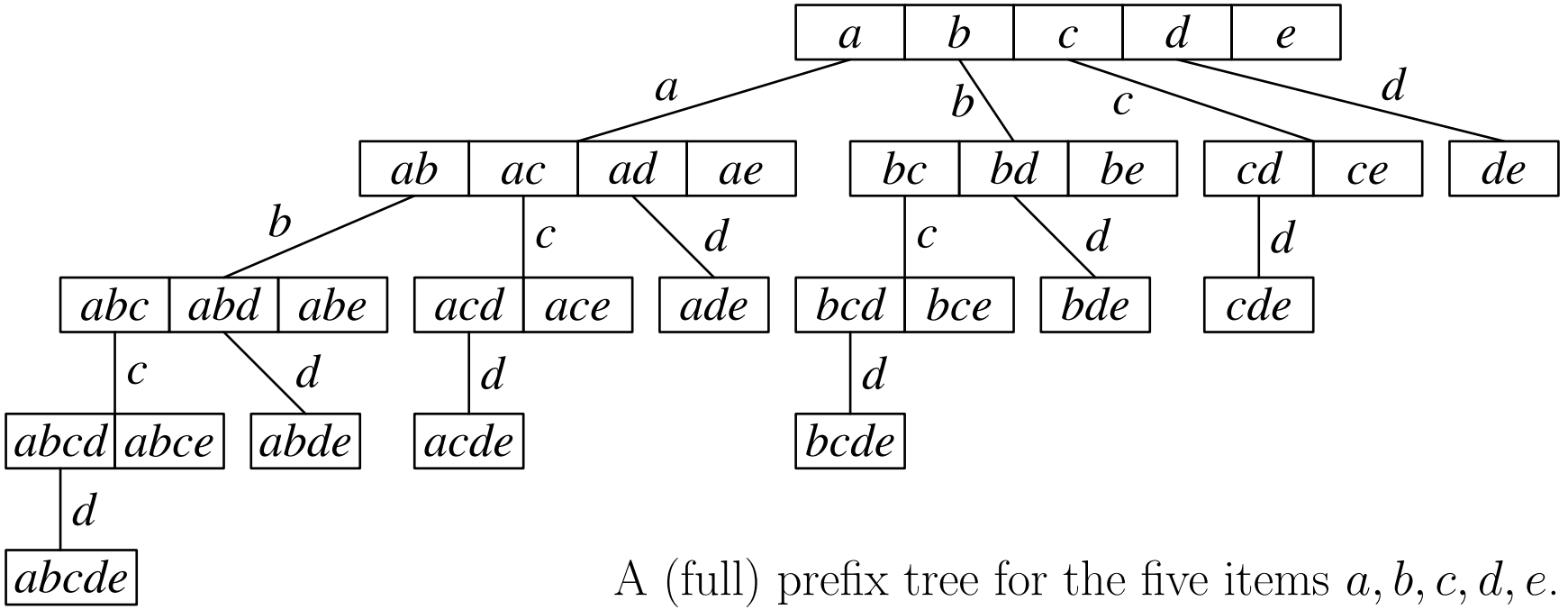
Canonical Parents and Prefix Trees

- Item sets, whose canonical code words share the same longest proper prefix are siblings, because they have (by definition) the same canonical parent.
- This allows us to represent the canonical parent tree as a **prefix tree** or **trie**.

Canonical parent tree/prefix tree and prefix tree with merged siblings for five items:



Canonical Parents and Prefix Trees



A (full) prefix tree for the five items a, b, c, d, e.

- Based on a global order of the items (which can be arbitrary).
- The item sets counted in a node consist of
  - all items labeling the edges to the node (common prefix) and
  - one item following the last edge label in the item order.

Search Tree Pruning

In applications the search tree tends to get very large, so pruning is needed.

- **Structural Pruning:**
  - Extensions based on canonical code words remove superfluous paths.
  - Explains the unbalanced structure of the full prefix tree.
- **Support Based Pruning:**
  - **No superset of an infrequent item set can be frequent.** (*apriori property*)
  - No counters for item sets having an infrequent subset are needed.
- **Size Based Pruning:**
  - Prune the tree if a certain depth (a certain size of the item sets) is reached.
  - Idea: Sets with too many items can be difficult to interpret.

The Order of the Items

- The structure of the (structurally pruned) prefix tree obviously depends on the chosen order of the items.

- In principle, the order is arbitrary (that is, any order can be used).

  However, the number and the size of the nodes that are visited in the search differs considerably depending on the order.

  As a consequence, the execution times of frequent item set mining algorithms can differ considerably depending on the item order.

- Which order of the items is best (leads to the fastest search) can depend on the frequent item set mining algorithm used.

  Advanced methods even adapt the order of the items during the search (that is, use different, but "compatible" orders in different branches).

- Heuristics for choosing an item order are usually based on (conditional) independence assumptions.

The Order of the Items

Heuristics for Choosing the Item Order

- **Basic Idea: independence assumption**

  It is plausible that frequent item sets consist of frequent items.

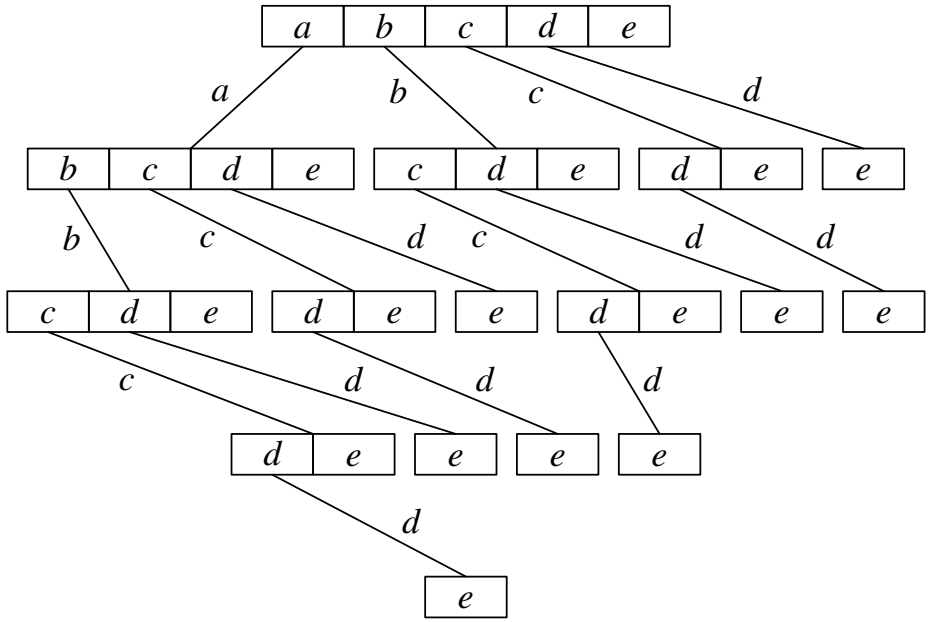
  - Sort the items w.r.t. their support (frequency of occurrence).
  - Sort descendingly: Prefix tree has fewer, but larger nodes.
  - Sort ascendingly: Prefix tree has more, but smaller nodes.

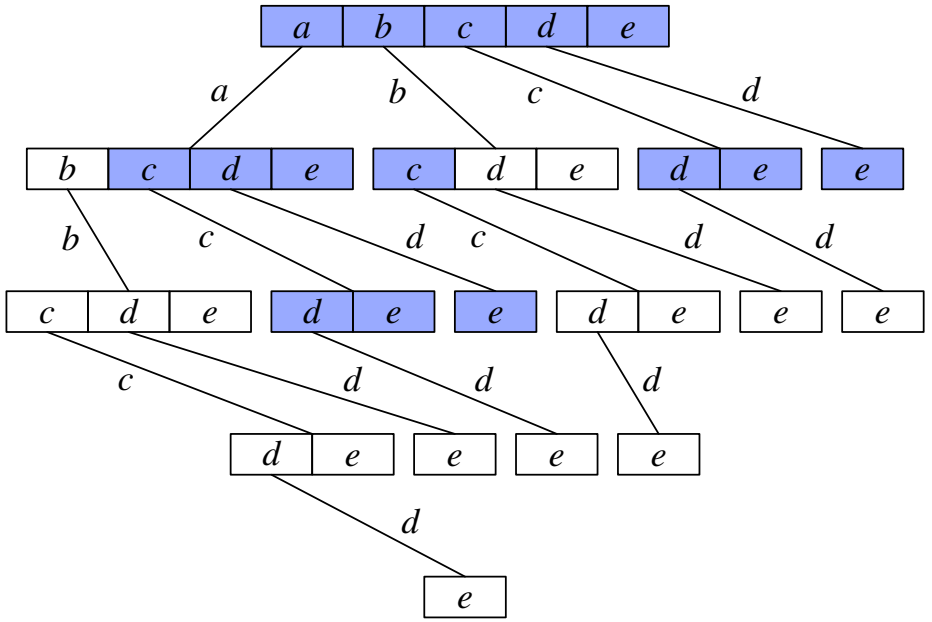
- **Extension of this Idea:**

  Sort items w.r.t. the sum of the sizes of the transactions that cover them.

  - Idea: the sum of transaction sizes also captures implicitly the frequency of pairs, triplets etc. (though, of course, only to some degree).
  - Empirical evidence: better performance than simple frequency sorting.

Searching the Prefix Tree





- **Apriori**
  - Breadth-first/levelwise search (item sets of same size).
  - Subset tests on transactions to find the support of item sets.
- **Eclat**
  - Depth-first search (item sets with same prefix).
  - Intersection of transaction lists to find the support of item sets.

Searching the Prefix Tree Levelwise

(Apriori Algorithm Revisited)

Apriori: Basic Ideas

- The item sets are checked in the **order of increasing size** (**breadth-first/levelwise traversal** of the prefix tree).
- The canonical form of item sets and the induced prefix tree are used to ensure that each candidate item set is generated at most once.
- The already generated levels are used to execute *a priori* pruning of the candidate item sets (using the **apriori property**).

  (*a priori:* before accessing the transaction database to determine the support)
- Transactions are represented as simple arrays of items (so-called **horizontal transaction representation**, see also below).
- The support of a candidate item set is computed by checking whether they are subsets of a transaction or by generating subsets of a transaction and finding them among the candidates.

Apriori: Levelwise Search

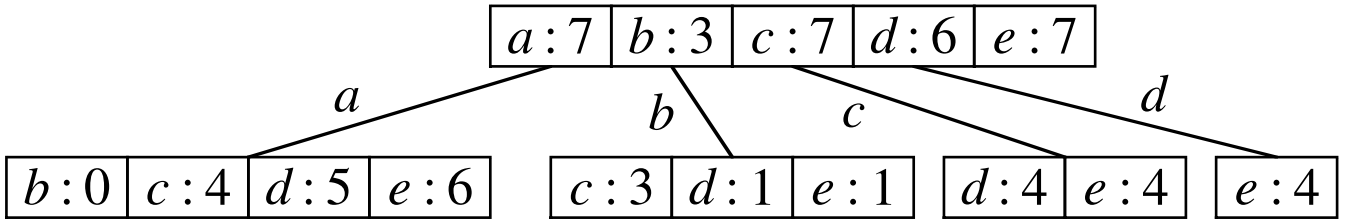
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

$a:7$	$b:3$	$c:7$	$d:6$	$e:7$

- Example transaction database with 5 items and 10 transactions.
- Minimum support: 30%, that is, at least 3 transactions must contain the item set.
- All sets with one item (singletons) are frequent $\Rightarrow$ full second level is needed.

Apriori: Levelwise Search

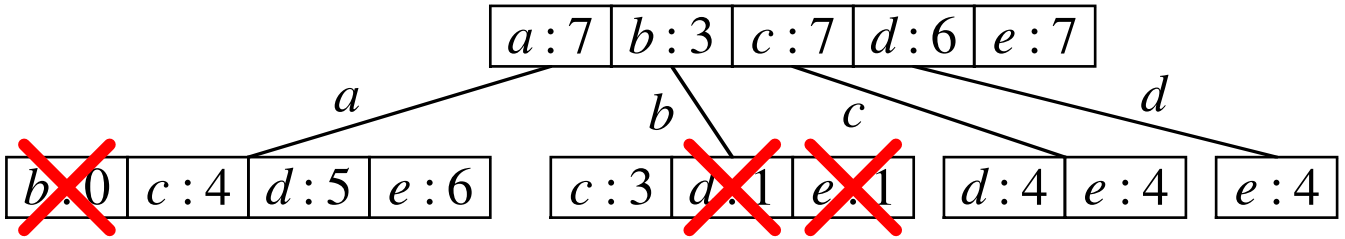
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Determining the support of item sets: For each item set traverse the database and count the transactions that contain it (highly inefficient).
- Better: Traverse the tree for each transaction and find the item sets it contains (efficient: can be implemented as a simple (doubly) recursive procedure).

Apriori: Levelwise Search

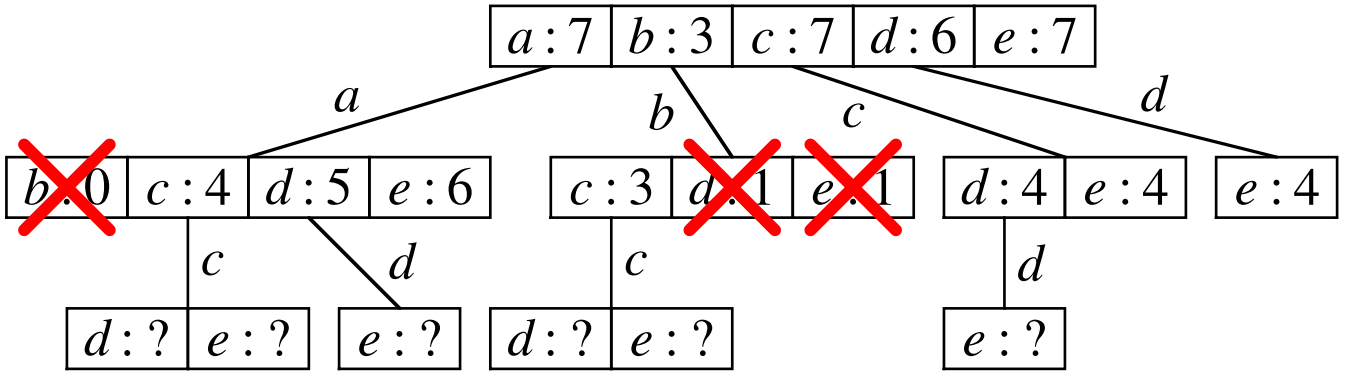
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Minimum support: 30%, that is, at least 3 transactions must contain the item set.
- Infrequent item sets: $\{a, b\}$, $\{b, d\}$, $\{b, e\}$.
- The subtrees starting at these item sets can be pruned.
  (*a posteriori*: after accessing the transaction database to determine the support)

Apriori: Levelwise Search

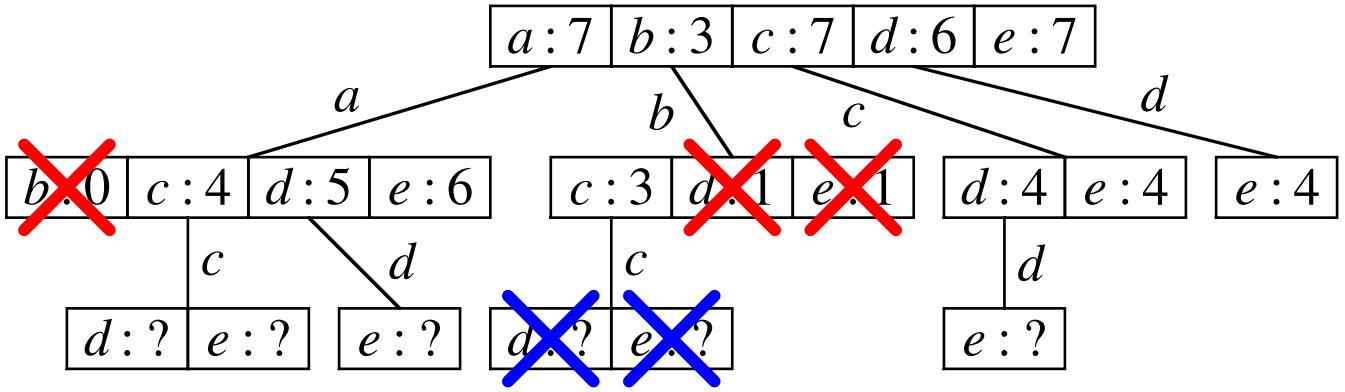
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Generate candidate item sets with 3 items (parents must be frequent).
- Before counting, check whether the candidates contain an infrequent item set.
  - An item set with k items has k subsets of size $k - 1$.
  - The parent item set is only one of these subsets.

Apriori: Levelwise Search

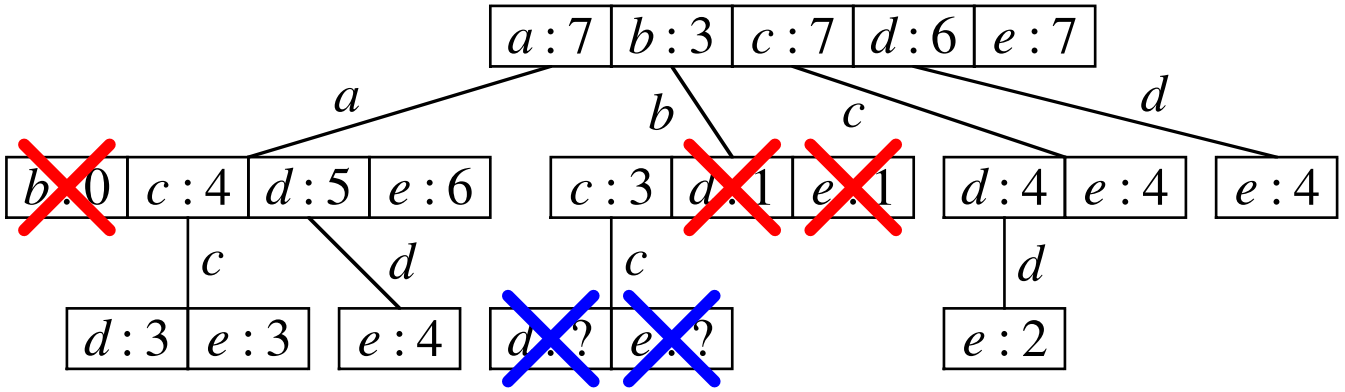
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The item sets $\{b, c, d\}$ and $\{b, c, e\}$ can be pruned, because
  - $\{b, c, d\}$ contains the infrequent item set $\{b, d\}$ and
  - $\{b, c, e\}$ contains the infrequent item set $\{b, e\}$.
- *a priori*: before accessing the transaction database to determine the support

Apriori: Levelwise Search

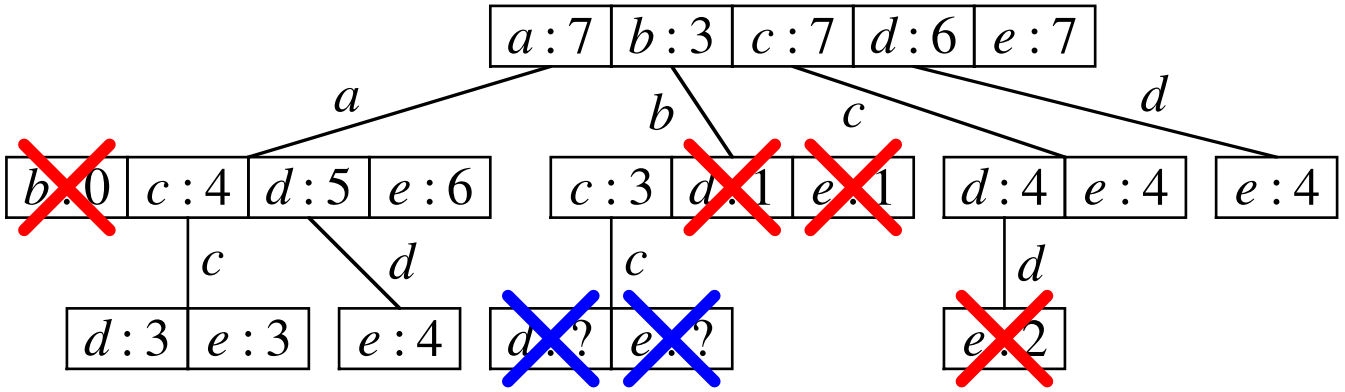
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Only the remaining four item sets of size 3 are evaluated.
- No other item sets of size 3 can be frequent.
- The transaction database is accessed to determine the support.

Apriori: Levelwise Search

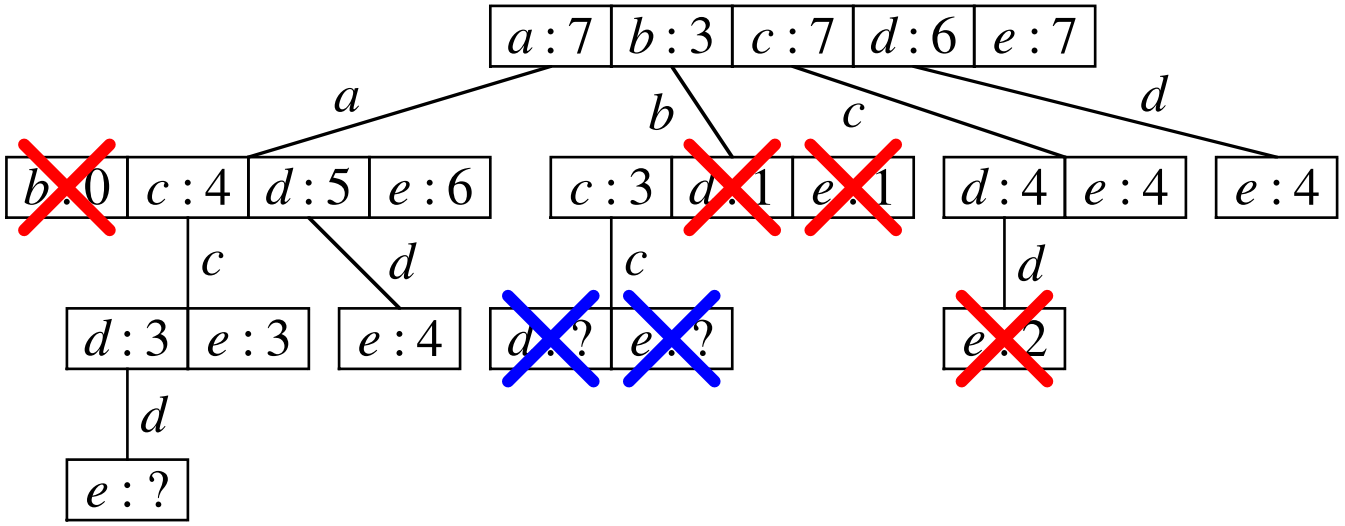
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Minimum support: 30%, that is, at least 3 transactions must contain the item set.
- The infrequent item set $\{c, d, e\}$ is pruned.
  (*a posteriori*: after accessing the transaction database to determine the support)
- Blue: *a priori* pruning, Red: *a posteriori* pruning.

Apriori: Levelwise Search

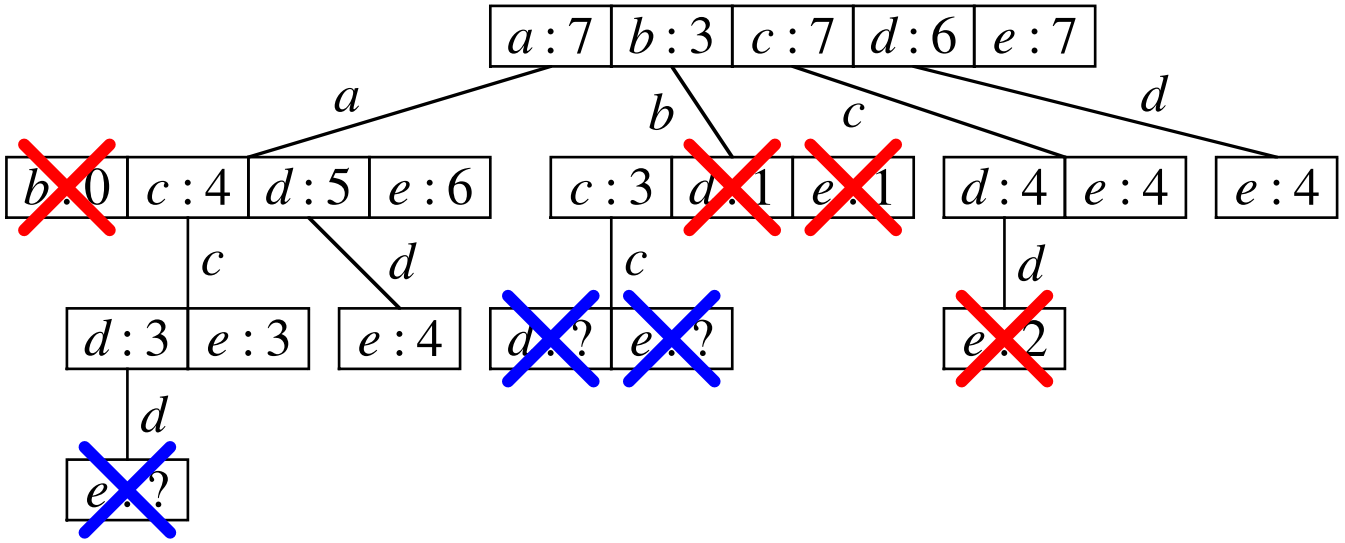
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Generate candidate item sets with 4 items (parents must be frequent).
- Before counting, check whether the candidates contain an infrequent item set. (*a priori* pruning)

Apriori: Levelwise 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 $\{a, c, d, e\}$ can be pruned,
  because it contains the infrequent item set $\{c, d, e\}$.
- Consequence: No candidate item sets with four items.
- Fourth access to the transaction database is not necessary.

Apriori: Node Organization 1

Idea: Optimize the organization of the counters and the child pointers.

Direct Indexing:

- Each node is a simple array of counters.
- An item is used as a direct index to find the counter.
- Advantage: Counter access is extremely fast.
- Disadvantage: Memory usage can be high due to "gaps" in the index space.

Sorted Vectors:

- Each node is a (sorted) array of item/counter pairs.
- A binary search is necessary to find the counter for an item.
- Advantage: Memory usage may be smaller, no unnecessary counters.
- Disadvantage: Counter access is slower due to the binary search.

Apriori: Node Organization 2

Hash Tables:

- Each node is a array of item/counter pairs (closed hashing).
- The index of a counter is computed from the item code.
- Advantage: Faster counter access than with binary search.
- Disadvantage: Higher memory usage than sorted arrays (pairs, fill rate).
  The order of the items cannot be exploited.

Child Pointers:

- The deepest level of the item set tree does not need child pointers.
- Fewer child pointers than counters are needed.

  $\Rightarrow$ It pays to represent the child pointers in a separate array.
- The sorted array of item/counter pairs can be reused for a binary search.

Apriori: Item Coding

- Items are coded as consecutive integers starting with 0
  (needed for the direct indexing approach).
- The size and the number of the "gaps" in the index space
  depend on how the items are coded.
- Idea: It is plausible that frequent item sets consist of frequent items.
  - Sort the items w.r.t. their frequency (group frequent items).
  - Sort descendingly: prefix tree has fewer nodes.
  - Sort ascendingly: there are fewer and smaller index "gaps".
  - Empirical evidence: sorting ascendingly is better.
- Extension: Sort items w.r.t. the sum of the sizes
  of the transactions that cover them.
  - Empirical evidence: better than simple item frequencies.

Apriori: Recursive Counting

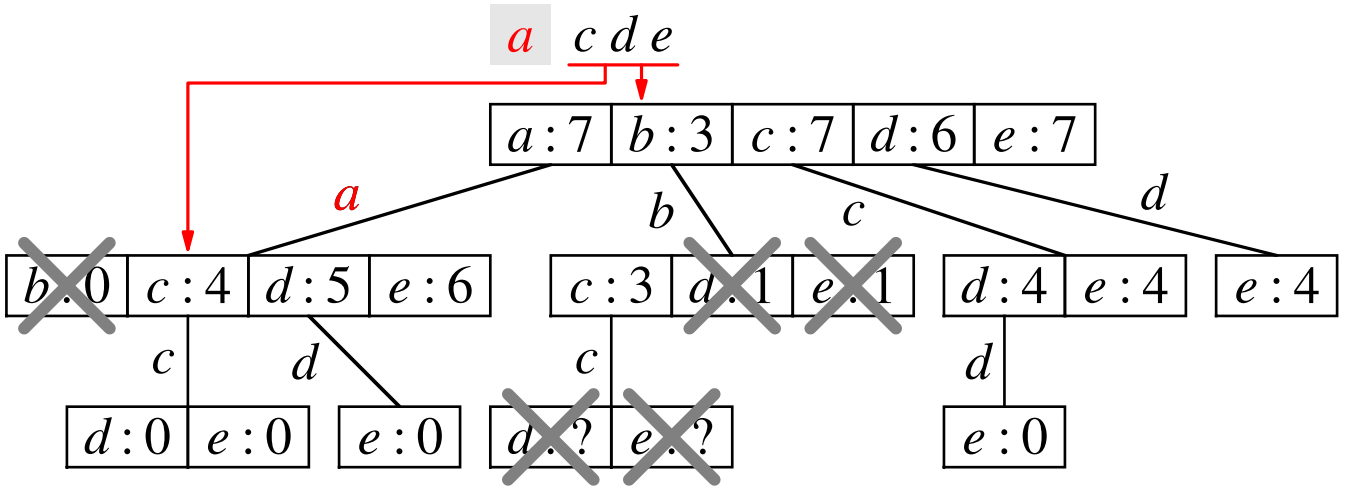
- The items in a transaction are sorted (ascending item codes).
- Processing a transaction is a **(doubly) recursive procedure**.
  To process a transaction for a node of the item set tree:
  - Go to the child corresponding to the first item in the transaction and count the suffix of the transaction recursively for that child.

    (In the currently deepest level of the tree we increment the counter corresponding to the item instead of going to the child node.)
  - Discard the first item of the transaction and process the remaining suffix recursively for the node itself.
- Optimizations:
  - Directly skip all items preceding the first item in the node.
  - Abort the recursion if the first item is beyond the last one in the node.
  - Abort the recursion if a transaction is too short to reach the deepest level.

Apriori: Recursive Counting

transaction
to count:
$\{a, c, d, e\}$

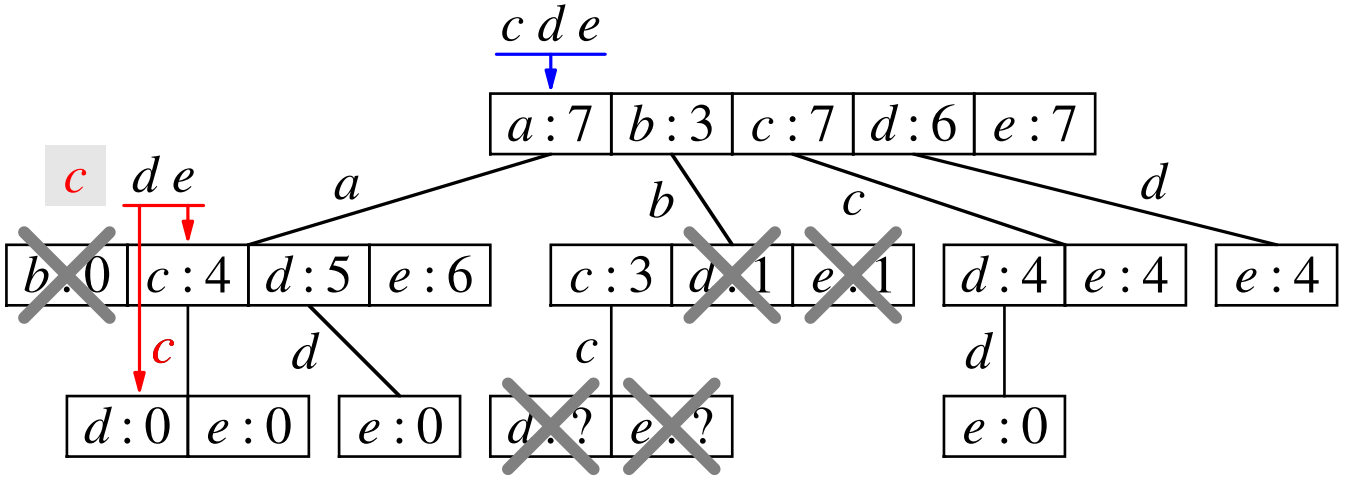
current
item set size: 3



---

processing: a

processing: c

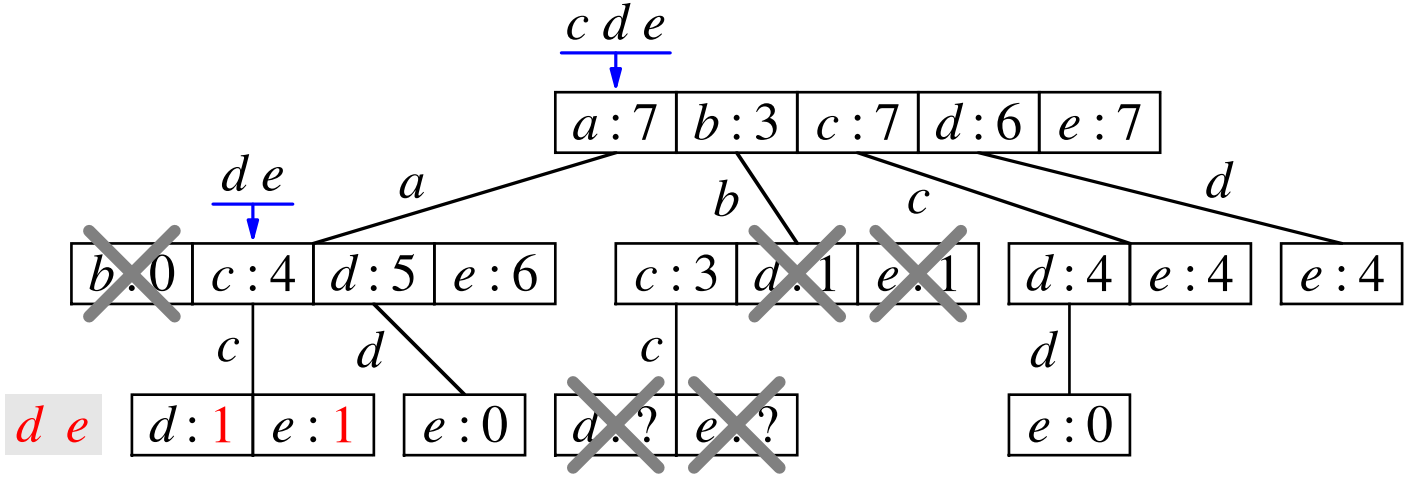


Apriori: Recursive Counting

processing: a

processing: c

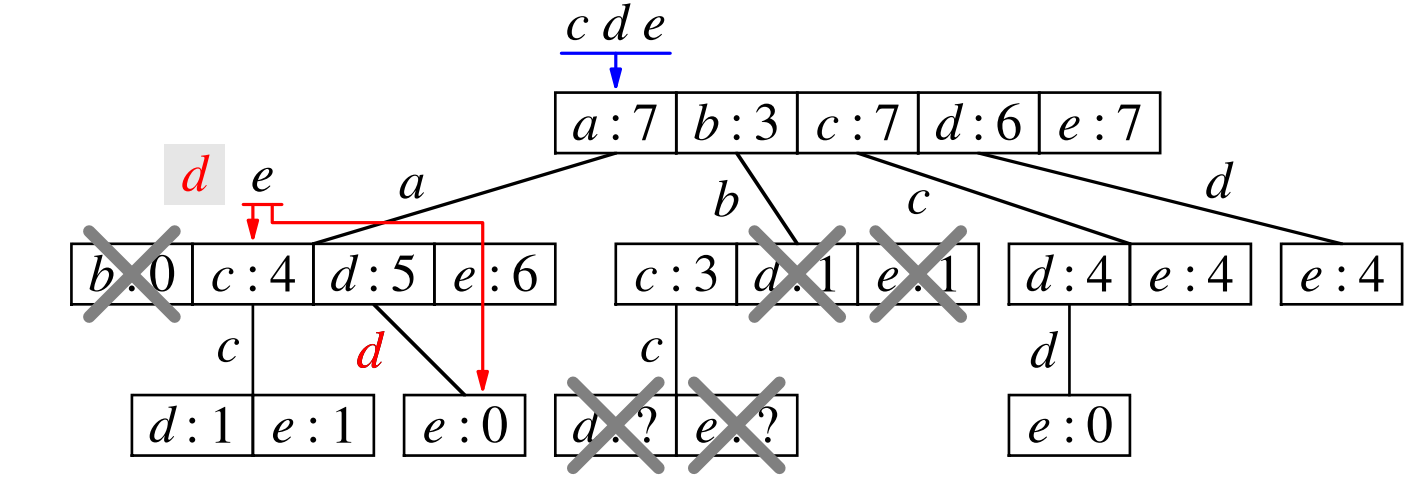
processing: $d\ e$



---

processing: a

processing: d

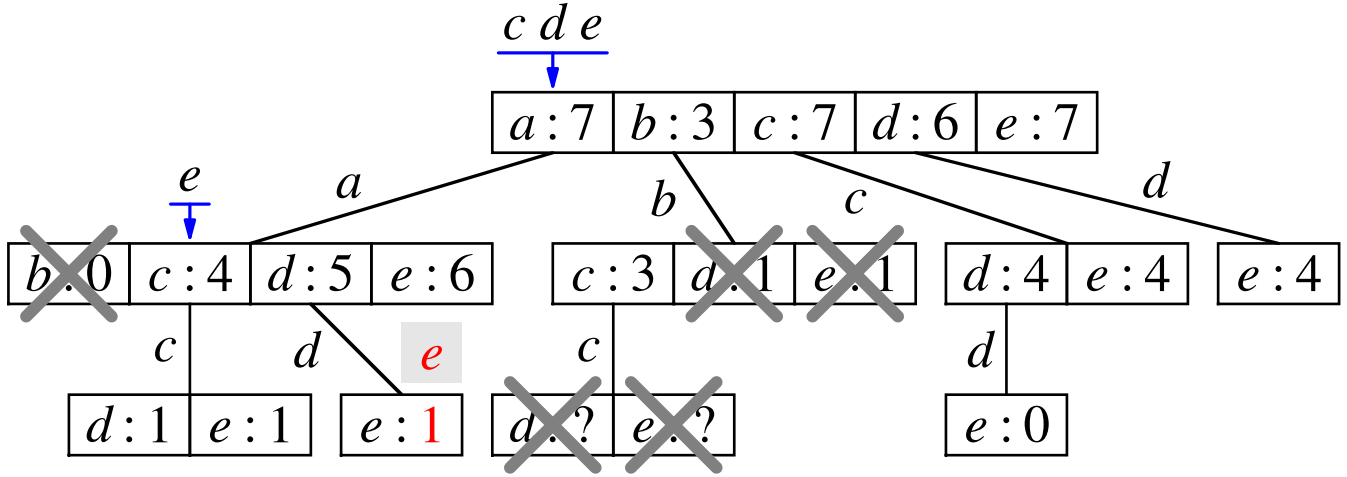


Apriori: Recursive Counting

processing: a

processing: d

processing: e

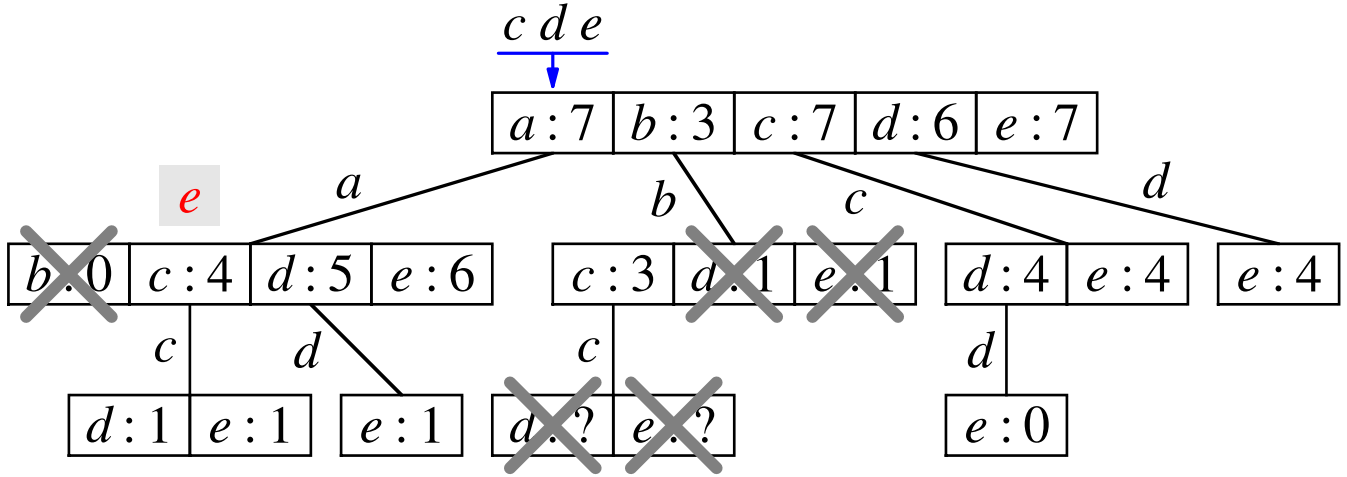


---

processing: a

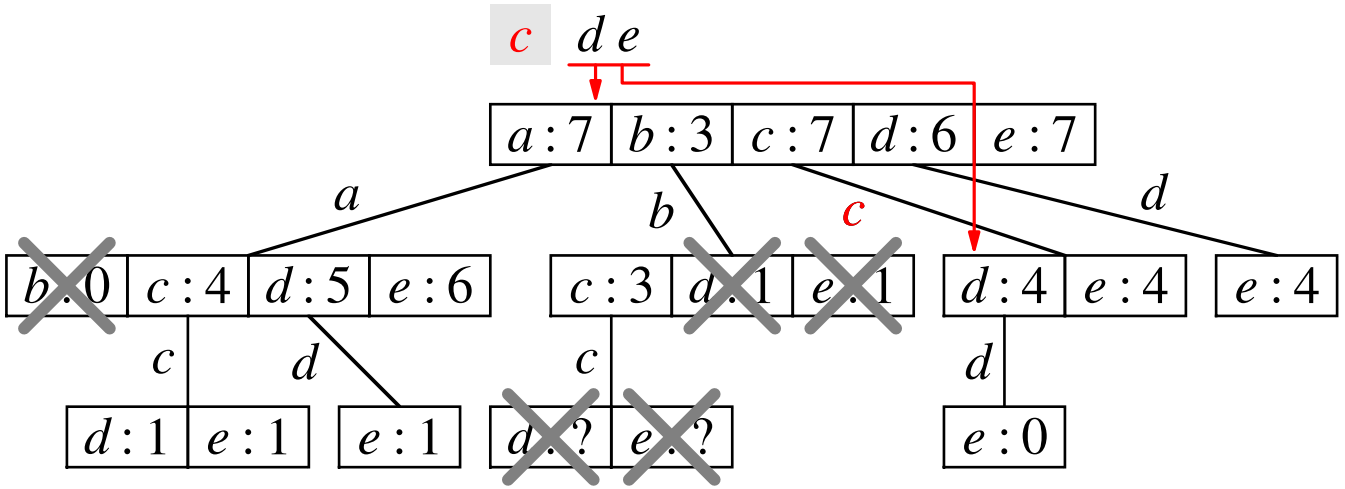
processing: e

(skipped:
too few items)



Apriori: Recursive Counting

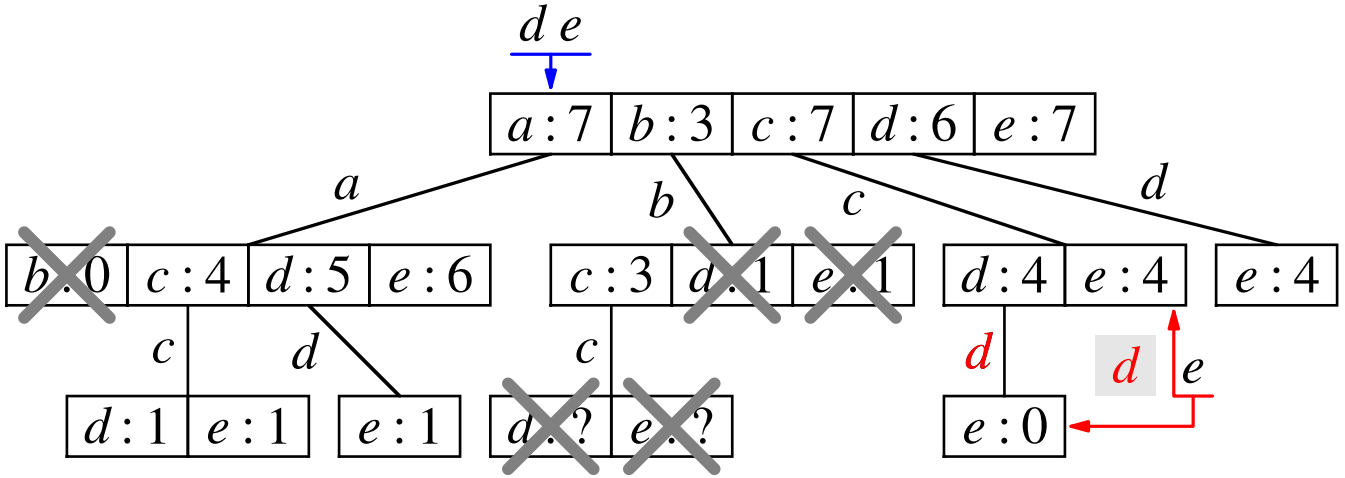
processing: c



---

processing: c

processing: d

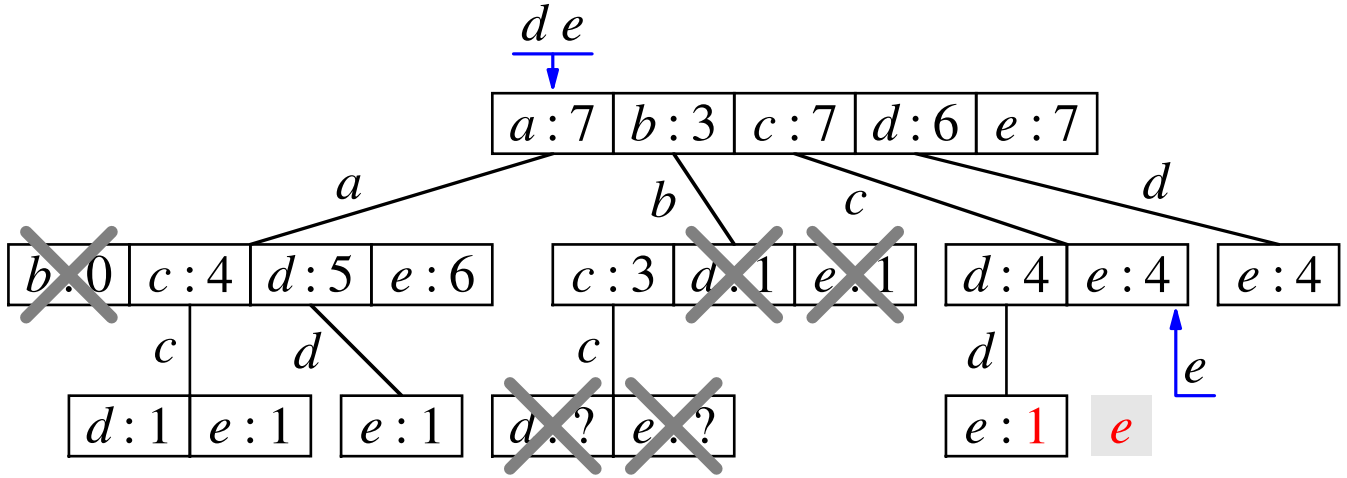


Apriori: Recursive Counting

processing: c

processing: d

processing: e

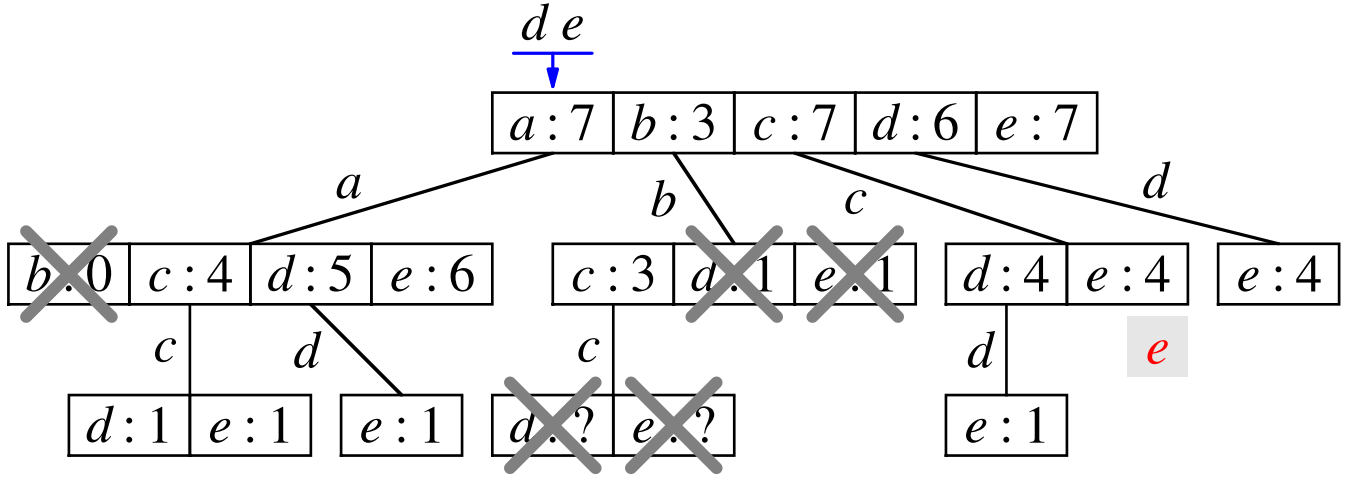


---

processing: c

processing: e

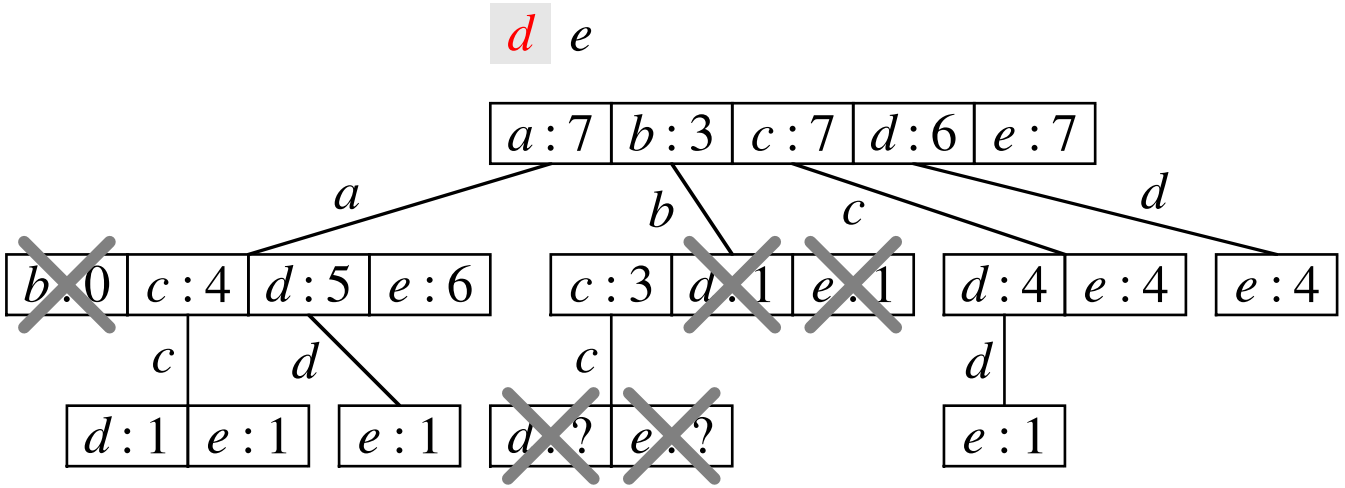
(skipped:
too few items)



Apriori: Recursive Counting

processing: d

(skipped:
too few items)



- Processing a transaction (suffix) in a node is easily implemented as a simple loop.
- For each item the remaining suffix is processed in the corresponding child.
- If the (currently) deepest tree level is reached,
  counters are incremented for each item in the transaction (suffix).
- If the remaining transaction (suffix) is too short to reach
  the (currently) deepest level, the recursion is terminated.

Apriori: Transaction Representation

Direct Representation:

- Each transaction is represented as an array of items.
- The transactions are stored in a simple list or array.

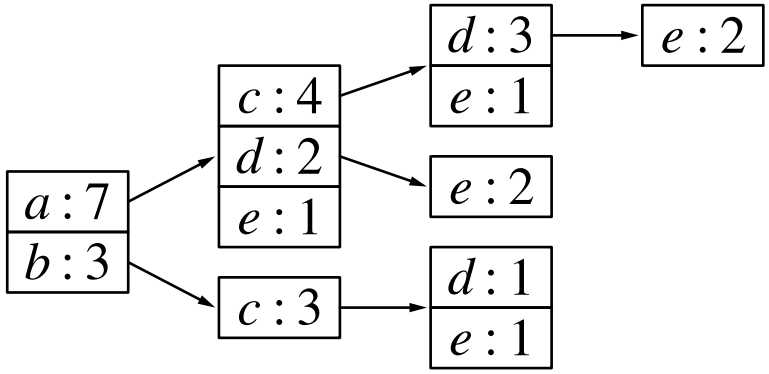
Organization as a Prefix Tree:

- The items in each transaction are sorted (arbitrary, but fixed order).
- Transactions with the same prefix are grouped together.
- Advantage: a common prefix is processed only once in the support counting.
- Gains from this organization depend on how the items are coded:
  - Common transaction prefixes are more likely
    if the items are sorted with descending frequency.
  - However: an ascending order is better for the search and
    this dominates the execution time (empirical evidence).

Apriori: Transactions as a Prefix Tree

transaction database	lexicographically sorted
a, d, e	a, c, d
b, c, d	a, c, d, e
a, c, e	a, c, d, e
a, c, d, e	a, c, e
a, e	a, d, e
a, c, d	a, d, e
b, c	a, e
a, c, d, e	b, c
b, c, e	b, c, d
a, d, e	b, c, e

prefix tree representation



- Items in transactions are sorted w.r.t. some arbitrary order, transactions are sorted lexicographically, then a prefix tree is constructed.
- **Advantage:** identical transaction prefixes are processed only once.

Summary Apriori

Basic Processing Scheme

- Breadth-first/levelwise traversal of the partially ordered set $(2^B, \subseteq)$.
- Candidates are formed by merging item sets that differ in only one item.
- Support counting can be done with a (doubly) recursive procedure.

Advantages

- "Perfect" pruning of infrequent candidate item sets (with infrequent subsets).

Disadvantages

- Can require a lot of memory (since all frequent item sets are represented).
- Support counting takes very long for large transactions.

Searching the Prefix Tree Depth-First

(Eclat, FP-growth and other algorithms)

Depth-First Search and Conditional Databases

- A depth-first search can also be seen as a **divide-and-conquer scheme**:

  First find all frequent item sets that contain a chosen item,
  then all frequent item sets that do not contain it.

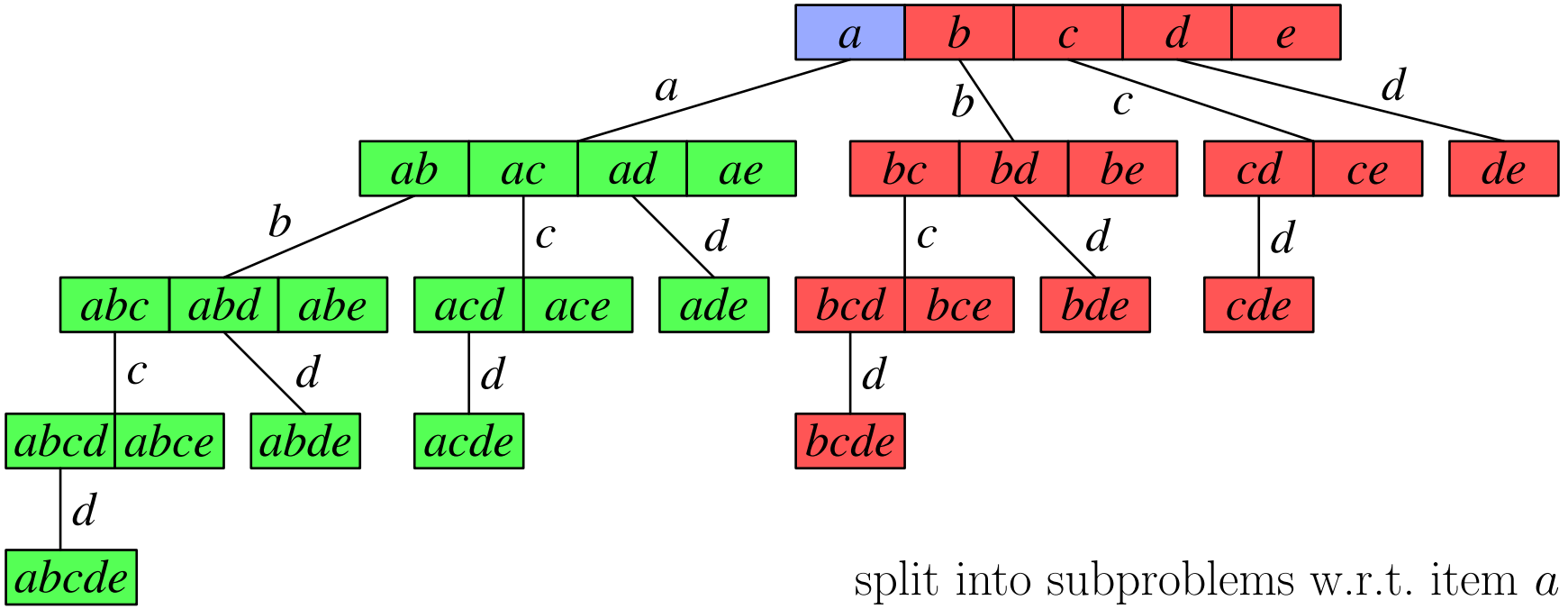
- General search procedure:
  - Let the item order be $a < b < c < \cdots$.
  - Restrict the transaction database to those transactions that contain a.
    This is the **conditional database for the prefix** a.

    Recursively search this conditional database for frequent item sets
    and add the prefix a to all frequent item sets found in the recursion.
  - Remove the item a from the transactions in the *full* transaction database.
    This is the **conditional database for item sets without** a.

    Recursively search this conditional database for frequent item sets.

- With this scheme only frequent one-element item sets have to be determined.
  Larger item sets result from adding possible prefixes.

Depth-First Search and Conditional Databases

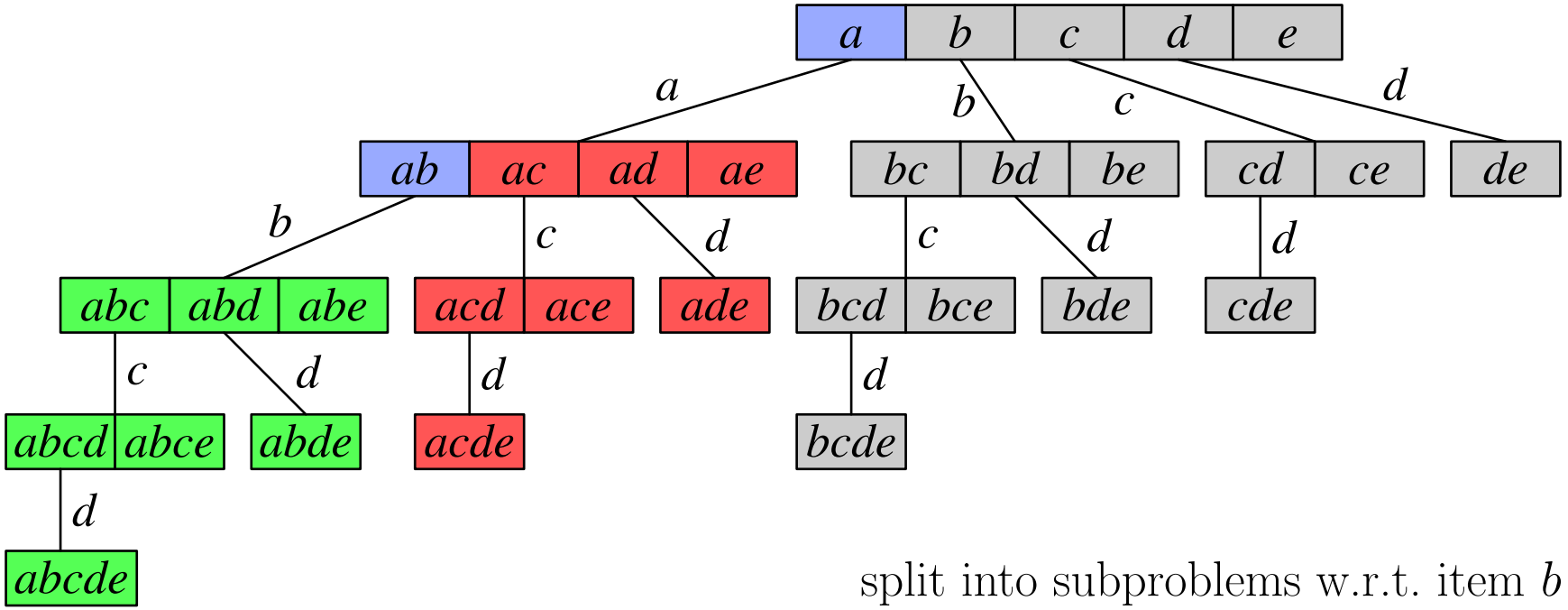


split into subproblems w.r.t. item a

- blue : item set containing *only* item a.
  green: item sets containing item a (and at least one other item).
  red : item sets not containing item a (but at least one other item).

- green: needs cond. database with transactions containing item a.
  red : needs cond. database with *all* transactions, but with item a removed.

Depth-First Search and Conditional Databases

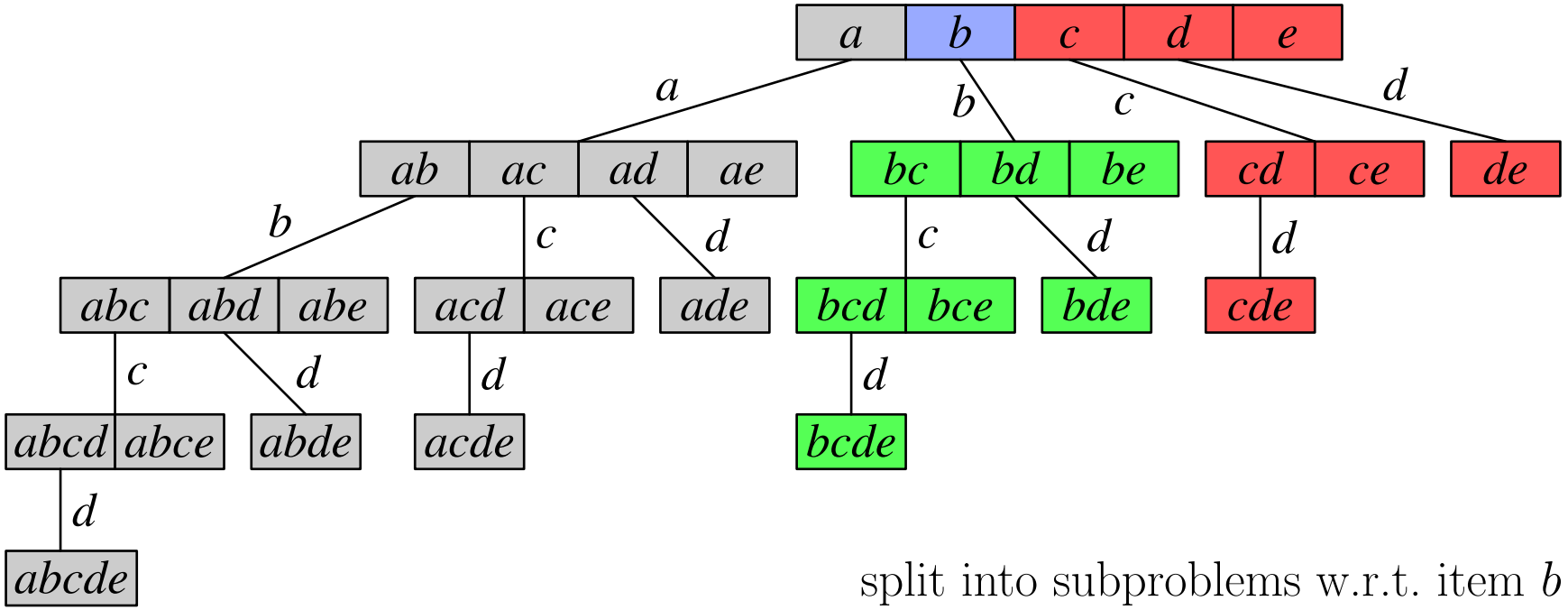


split into subproblems w.r.t. item b

- blue : item sets $\{a\}$ and $\{a, b\}$.
  green: item sets containing both items a and b (and at least one other item).
  red : item sets containing item a (and at least one other item), but not item b.

- green: needs database with trans. containing both items a and b.
  red : needs database with trans. containing item a, but with item b removed.

Depth-First Search and Conditional Databases



split into subproblems w.r.t. item b

- blue : item set containing *only* item b.
  green: item sets containing item b (and at least one other item), but not item a.
  red : item sets containing neither item a nor b (but at least one other item).

- green: needs database with trans. containing item b, but with item a removed.
  red : needs database with *all* trans., but with both items a and b removed.

Formal Description of the Divide-and-Conquer Scheme

- Generally, a divide-and-conquer scheme can be described as a set of (sub)problems.
  - The initial (sub)problem is the actual problem to solve.
  - A subproblem is processed by splitting it into smaller subproblems, which are then processed recursively.
- All subproblems that occur in frequent item set mining can be defined by
  - a **conditional transaction database** and
  - a **prefix** (of items).

  The prefix is a set of items that has to be added to all frequent item sets that are discovered in the conditional transaction database.
- Formally, all subproblems are tuples $S = (T_*, P)$, where T_* is a conditional transaction database and $P \subseteq B$ is a prefix.
- The initial problem, with which the recursion is started, is $S = (T, \emptyset)$, where T is the transaction database to mine and the prefix is empty.

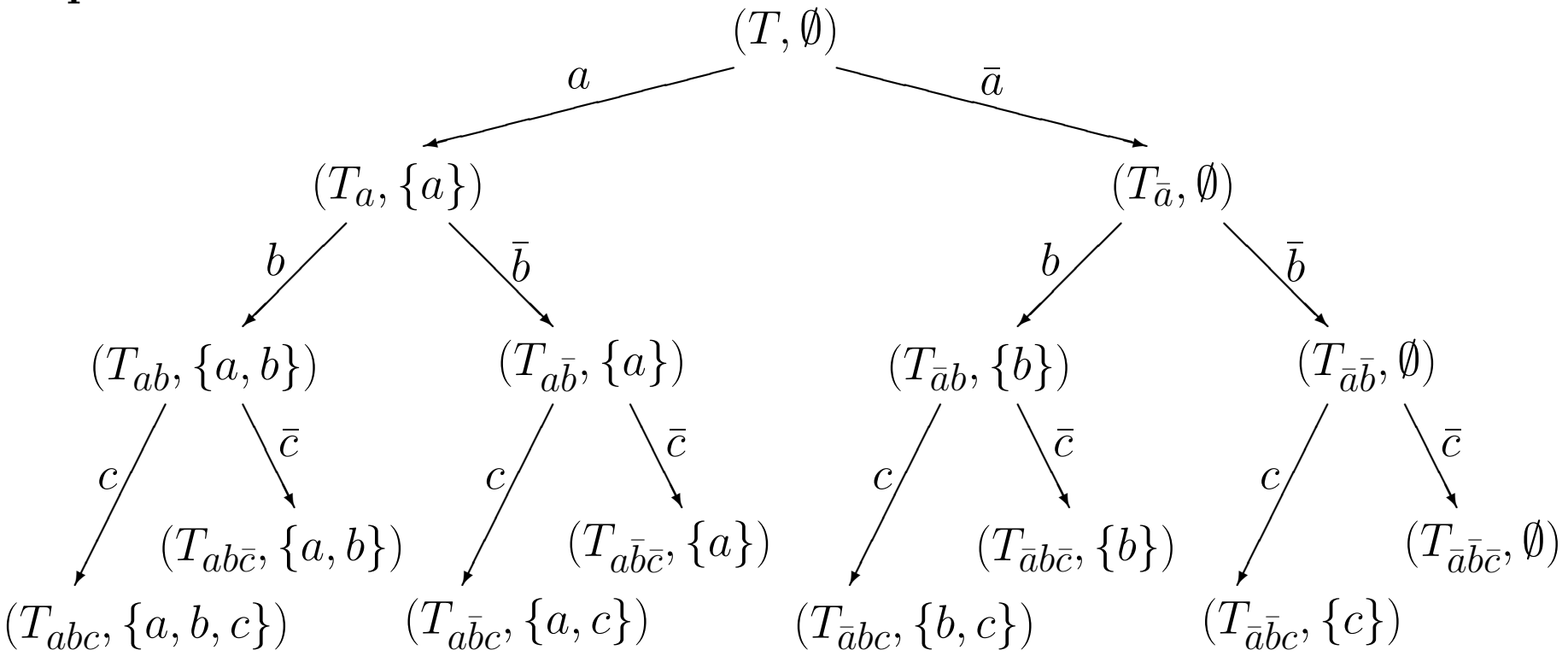
Formal Description of the Divide-and-Conquer Scheme

A subproblem $S_0 = (T_0, P_0)$ is processed as follows:

- Choose an item $i \in B_0$, where B_0 is the set of items occurring in T_0.
- If $s_{T_0}(i) \geq s_{\min}$ (where $s_{T_0}(i)$ is the support of the item i in T_0):
  - Report the item set $P_0 \cup \{i\}$ as frequent with the support $s_{T_0}(i)$.
  - Form the subproblem $S_1 = (T_1, P_1)$ with $P_1 = P_0 \cup \{i\}$.
    T_1 comprises all transactions in T_0 that contain the item i,
    but with the item i removed (and empty transactions removed).
  - If T_1 is not empty, process S_1 recursively.
- In any case (that is, regardless of whether $s_{T_0}(i) \geq s_{\min}$ or not):
  - Form the subproblem $S_2 = (T_2, P_2)$, where $P_2 = P_0$.
    T_2 comprises all transactions in T_0 (whether they contain i or not),
    but again with the item i removed (and empty transactions removed).
  - If T_2 is not empty, process S_2 recursively.

Divide-and-Conquer Recursion

Subproblem Tree



- Branch to the left: include an item (first subproblem)
- Branch to the right: exclude an item (second subproblem)

(Items in the indices of the conditional transaction databases T have been removed from them.)

Reminder: Searching with the Prefix Property

Principle of a Search Algorithm based on the Prefix Property:

- **Base Loop:**
  - Traverse all possible items, that is,
    the canonical code words of all one-element item sets.
  - Recursively process each code word that describes a frequent item set.
- **Recursive Processing:**

  For a given (canonical) code word of a frequent item set:
  - Generate all possible extensions by one item.
    This is done by simply **appending the item** to the code word.
  - Check whether the extended code word is the **canonical code word**
    of the item set that is described by the extended code word
    (and, of course, whether the described item set is frequent).

    If it is, process the extended code word recursively, otherwise discard it.

Perfect Extensions

The search can easily be improved with so-called **perfect extension pruning**.

- Let T be a transaction database over an item base B.
  Given an item set I, an item $i \notin I$ is called a **perfect extension** of I w.r.t. T,
  iff the item sets I and $I \cup \{i\}$ have the same support: $s_T(I) = s_T(I \cup \{i\})$
  (that is, if all transactions containing the item set I also contain the item i).

- Perfect extensions have the following properties:

  - If the item i is a perfect extension of an item set I,
    then i is also a perfect extension of any item set $J \supseteq I$ (provided $i \notin J$).

    This can most easily be seen by considering that $K_T(I) \subseteq K_T(\{i\})$
    and hence $K_T(J) \subseteq K_T(\{i\})$, since $K_T(J) \subseteq K_T(I)$.

  - If $X_T(I)$ is the set of all perfect extensions of an item set I w.r.t. T
    (that is, if $X_T(I) = \{i \in B - I \mid s_T(I \cup \{i\}) = s_T(I)\}$),
    then all sets $I \cup J$ with $J \in 2^{X_T(I)}$ have the same support as I
    (where 2^M denotes the power set of a set M).

Perfect Extensions: Examples

transaction database

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: $\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

- c is a perfect extension of $\{b\}$ since $\{b\}$ and $\{b, c\}$ both have support 3.
- a is a perfect extension of $\{d, e\}$ since $\{d, e\}$ and $\{a, d, e\}$ both have support 4.
- There are no other perfect extensions in this example for a minimum support of $s_{\text{min}} = 3$.

Perfect Extension Pruning

- Consider again the original **divide-and-conquer scheme**:
  A subproblem $S_0 = (T_0, P_0)$ is split into
  - a subproblem $S_1 = (T_1, P_1)$ to find all frequent item sets that *do* contain an item $i \in B_0$ and
  - a subproblem $S_2 = (T_2, P_2)$ to find all frequent item sets that *do not* contain the item i.

- Suppose the item i is a **perfect extension** of the prefix P_0.
  - Let F_1 and F_2 be the sets of frequent item sets that are reported when processing S_1 and S_2, respectively.
  - It is $I \cup \{i\} \in F_1 \quad \Leftrightarrow \quad I \in F_2$.
  - The reason is that generally $P_1 = P_2 \cup \{i\}$ and in this case $T_1 = T_2$, because all transactions in T_0 contain item i (as i is a perfect extension).

- Therefore it suffices to solve one subproblem (namely S_2).
  The solution of the other subproblem (S_1) is constructed by adding item i.

Perfect Extension Pruning

- Perfect extensions can be exploited by collecting these items in the recursion, in a third element of a subproblem description.
- Formally, a subproblem is a triplet $S = (T_*, P, X)$, where
  - T_* is a **conditional transaction database**,
  - P is the set of **prefix items** for T_*,
  - X is the set of **perfect extension items**.
- Once identified, perfect extension items are no longer processed in the recursion, but are only used to generate all supersets of the prefix having the same support. Consequently, they are removed from the conditional transaction databases. This technique is also known as **hypercube decomposition**.
- The divide-and-conquer scheme has basically the same structure as without perfect extension pruning.

  However, the exact way in which perfect extensions are collected can depend on the specific algorithm used.

Reporting Frequent Item Sets

- With the described divide-and-conquer scheme, item sets are reported in **lexicographic order**.
- This can be exploited for **efficient item set reporting**:
  - The prefix P is a string, which is extended when an item is added to P.
  - Thus only one item needs to be formatted per reported frequent item set, the prefix is already formatted in the string.
  - Backtracking the search (return from recursion) removes an item from the prefix string.
  - This scheme can speed up the output considerably.

Example:

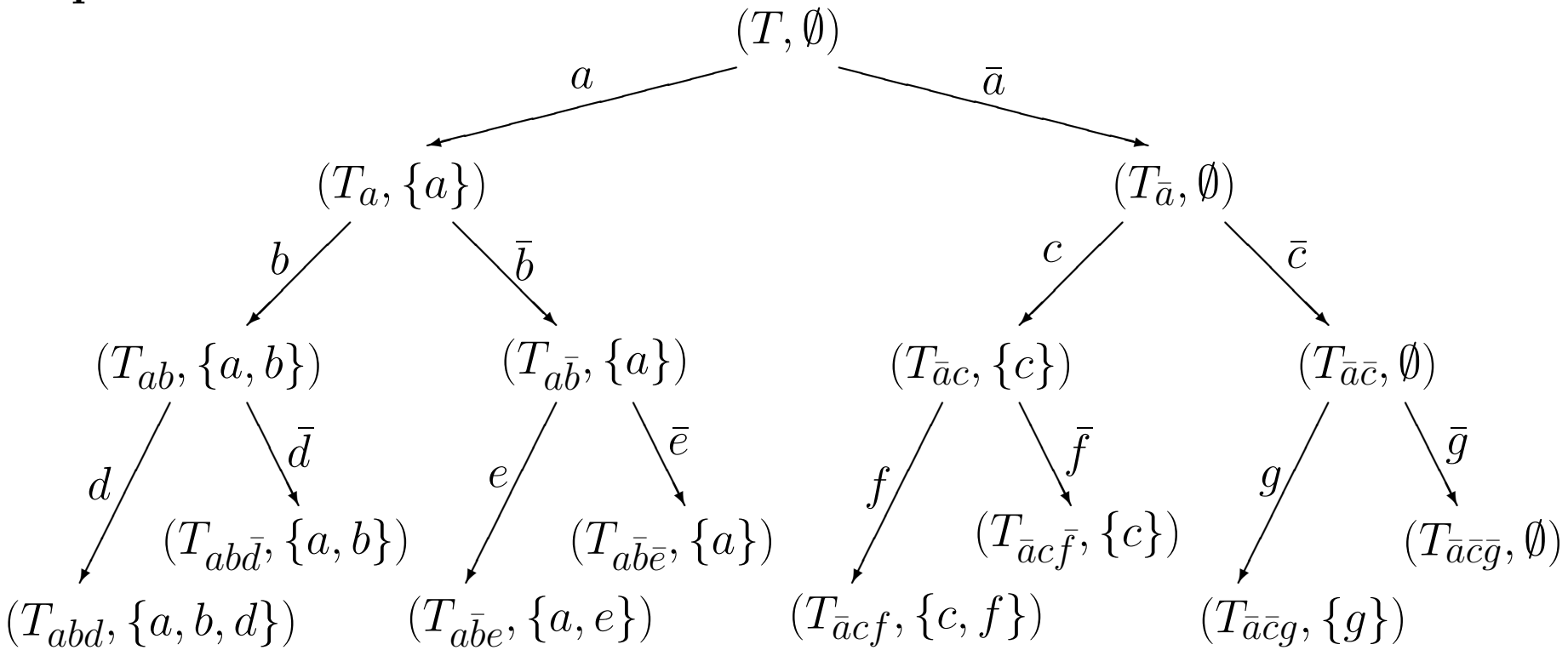
a	(7)	$a\ d\ e$	(4)	$c\ d$	(4)
$a\ c$	(4)	$a\ e$	(6)	$c\ e$	(4)
$a\ c\ d$	(3)	b	(3)	d	(6)
$a\ c\ e$	(3)	$b\ c$	(3)	$d\ e$	(4)
$a\ d$	(5)	c	(7)	e	(7)

Global and Local Item Order

- Up to now we assumed that the item order is (globally) fixed, and determined at the very beginning based on heuristics.
- However, the described divide-and-conquer scheme shows that a globally fixed item order is more restrictive than necessary:
  - The item used to split the current subproblem can be any item that occurs in the conditional transaction database of the subproblem.
  - There is no need to choose the same item for splitting sibling subproblems (as a global item order would require us to do).
  - The same heuristics used for determining a global item order suggest that the split item for a given subproblem should be selected from the (conditionally) least frequent item(s).
- As a consequence, the item orders may differ for every branch of the search tree.
  - However, two subproblems must share the item order that is fixed by the common part of their paths from the root (initial subproblem).

Item Order: Divide-and-Conquer Recursion

Subproblem Tree



- All local item orders start with $a < \ldots$
- All subproblems on the left share $a < b < \ldots$,
  All subproblems on the right share $a < c < \ldots$.

Global and Local Item Order

Local item orders have advantages and disadvantages:

- **Advantage**
  - In some data sets the order of the conditional item frequencies differs considerably from the global order.
  - Such data sets can sometimes be processed significantly faster with local item orders (depending on the algorithm).
- **Disadvantage**
  - The data structure of the conditional databases must allow us to determine conditional item frequencies quickly.
  - Not having a globally fixed item order can make it more difficult to determine conditional transaction databases w.r.t. split items (depending on the employed data structure).
  - The gains from the better item order may be lost again due to the more complex processing / conditioning scheme.

Transaction Database Representation

Transaction Database Representation

- Eclat, FP-growth and several other frequent item set mining algorithms rely on the described basic divide-and-conquer scheme.

  They differ mainly in how they represent the conditional transaction databases.

- The main approaches are horizontal and vertical representations:
  - In a **horizontal representation**, the database is stored as a list (or array) of transactions, each of which is a list (or array) of the items contained in it.
  - In a **vertical representation**, a database is represented by first referring with a list (or array) to the different items. For each item a list (or array) of identifiers is stored, which indicate the transactions that contain the item.

- However, this distinction is not pure, since there are many algorithms that use a combination of the two forms of representing a transaction database.

- Frequent item set mining algorithms also differ in how they construct new conditional transaction databases from a given one.

Transaction Database Representation

- The Apriori algorithm uses a **horizontal transaction representation**: each transaction is an array of the contained items.
  - Note that the alternative prefix tree organization is still an essentially *horizontal* representation.
- The alternative is a **vertical transaction representation**:
  - For each item a **transaction (index/identifier) list** is created.
  - The transaction list of an item i indicates the transactions that contain it, that is, it represents its **cover** $K_T(\{i\})$.
  - Advantage: the transaction list for a pair of items can be computed by intersecting the transaction lists of the individual items.
  - Generally, a vertical transaction representation can exploit

$$\forall I, J \subseteq B: \quad K_T(I \cup J) = K_T(I) \cap K_T(J).$$

- A combined representation is the **frequent pattern tree** (to be discussed later).

Transaction Database Representation

- **Horizontal Representation:** List items for each transaction
- **Vertical Representation:** List transactions for each item

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

horizontal representation

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

vertical representation

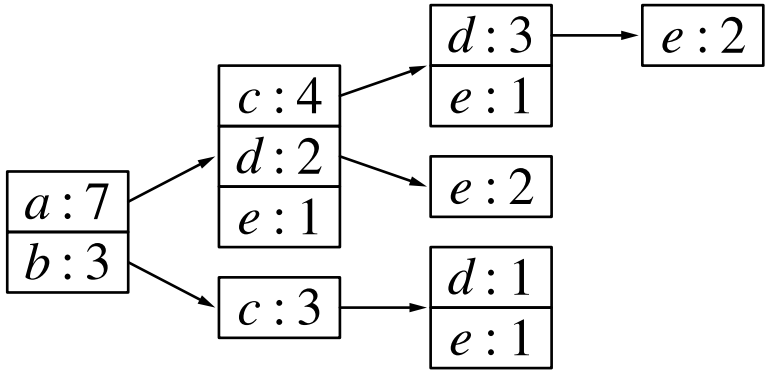
	a	b	c	d	e
1:	**1**	0	0	**1**	**1**
2:	0	**1**	**1**	**1**	0
3:	**1**	0	**1**	0	**1**
4:	**1**	0	**1**	**1**	**1**
5:	**1**	0	0	0	**1**
6:	**1**	0	**1**	**1**	0
7:	0	**1**	**1**	0	0
8:	**1**	0	**1**	**1**	**1**
9:	0	**1**	**1**	0	**1**
10:	**1**	0	0	**1**	**1**

matrix representation

Transaction Database Representation

transaction database	lexicographically sorted
a, d, e	a, c, d
b, c, d	a, c, d, e
a, c, e	a, c, d, e
a, c, d, e	a, c, e
a, e	a, d, e
a, c, d	a, d, e
b, c	a, e
a, c, d, e	b, c
b, c, e	b, c, d
a, d, e	b, c, e

prefix tree representation



- Note that a prefix tree representation is a compressed horizontal representation.
- Principle: **equal prefixes of transactions are merged.**
- This is most effective if the items are sorted descendingly w.r.t. their support.

The Eclat Algorithm

[Zaki, Parthasarathy, Ogihara, and Li 1997]

Eclat: Basic Ideas

- The item sets are checked in **lexicographic order** (**depth-first traversal** of the prefix tree).
- The search scheme is the same as the general scheme for searching with canonical forms having the prefix property and possessing a perfect extension rule (generate only canonical extensions).
- Eclat generates more candidate item sets than Apriori, because it (usually) does not store the support of all visited item sets.*

  As a consequence it cannot fully exploit the Apriori property for pruning.
- Eclat uses a purely **vertical transaction representation**.
- No subset tests and no subset generation are needed to compute the support.

  The support of item sets is rather determined by intersecting transaction lists.

* Note that Eclat cannot fully exploit the Apriori property, because it does not *store* the support of all explored item sets, not because it cannot *know* it. If all computed support values were stored, it could be implemented in such a way that all support values needed for full *a priori* pruning are available.

Eclat: Subproblem Split

a
7
1
3
4
5
6
8
10

b
3
2
7
9

c
7
2
3
4
6
7
8
9

d
6
1
2
4
6
8
10

e
7
1
3
4
5
8
9
10

b
0

c
4
3
4
6
8

d
5
1
4
6
8
10

e
6
1
3
4
5
8
10

↑ Conditional database for prefix a (1st subproblem)

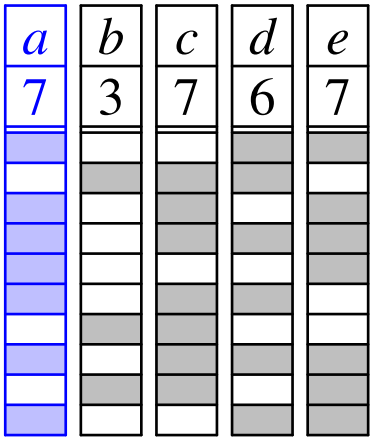
b
3
2
7
9

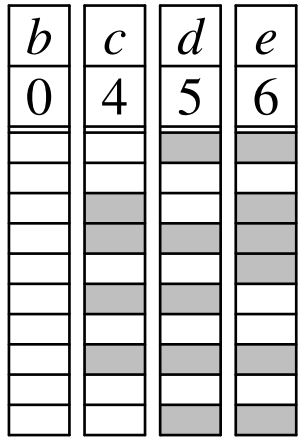
c
7
2
3
4
6
7
8
9

d
6
1
2
4
6
8
10

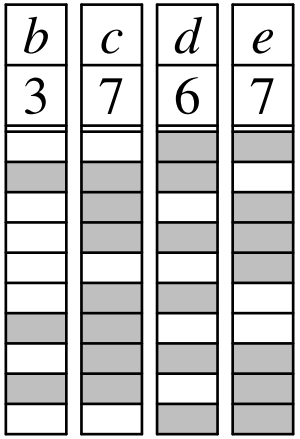
e
7
1
3
4
5
8
9
10

← Conditional database with item a removed (2nd subproblem)





↑ Conditional database for prefix a (1st subproblem)



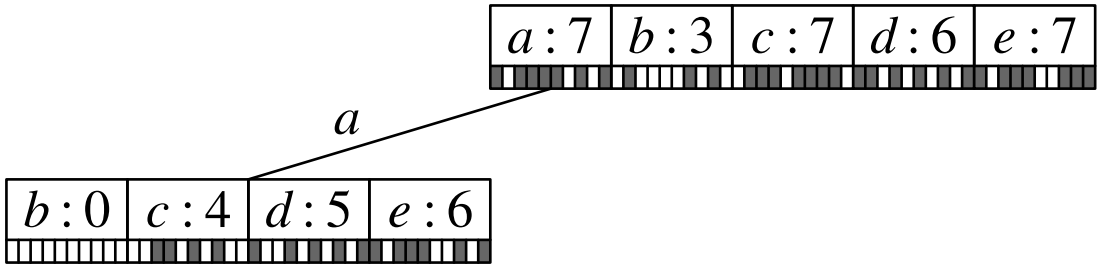
← Conditional database with item a removed (2nd subproblem)

Eclat: Depth-First Search

- Form a transaction list for each item. Here: bit array representation.
  - gray: 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

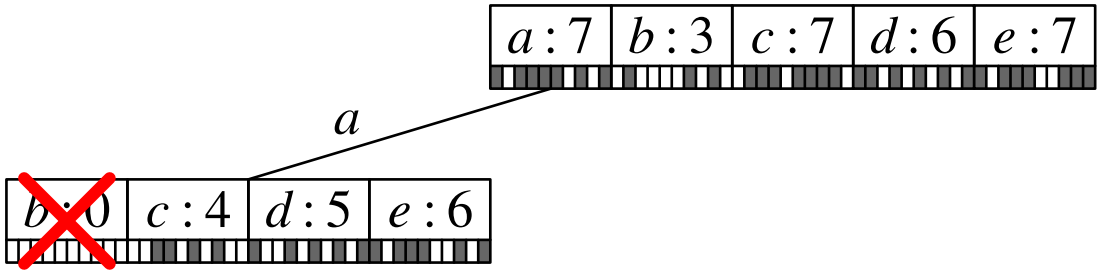
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

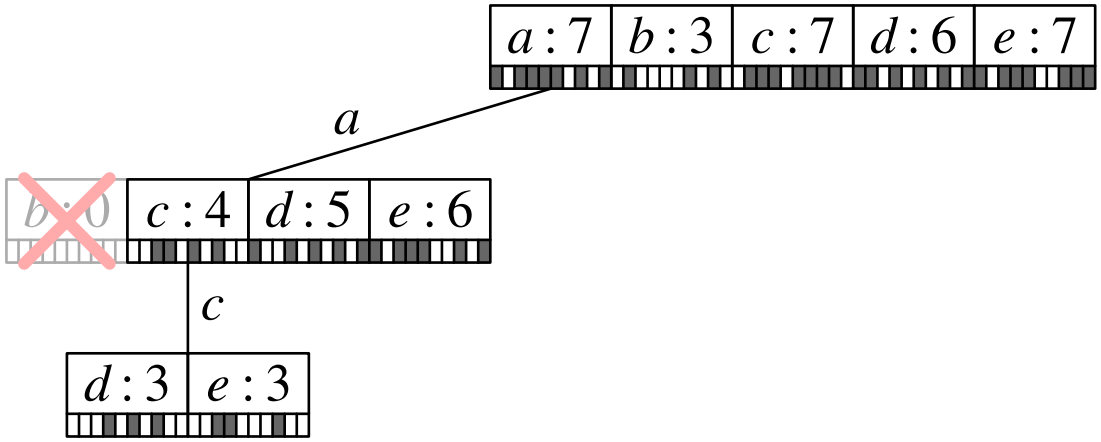
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

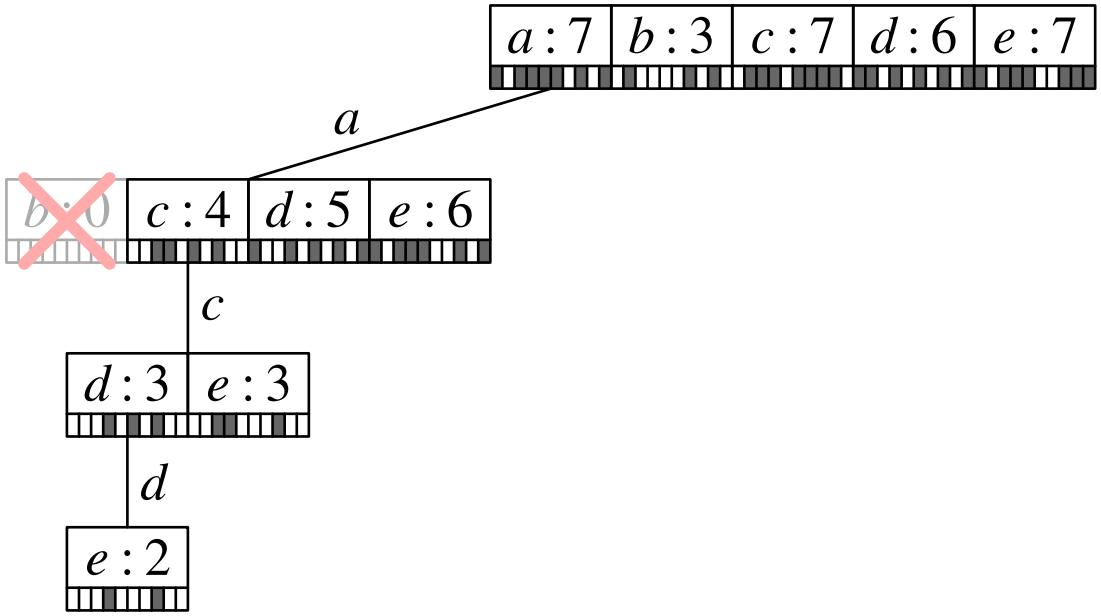
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

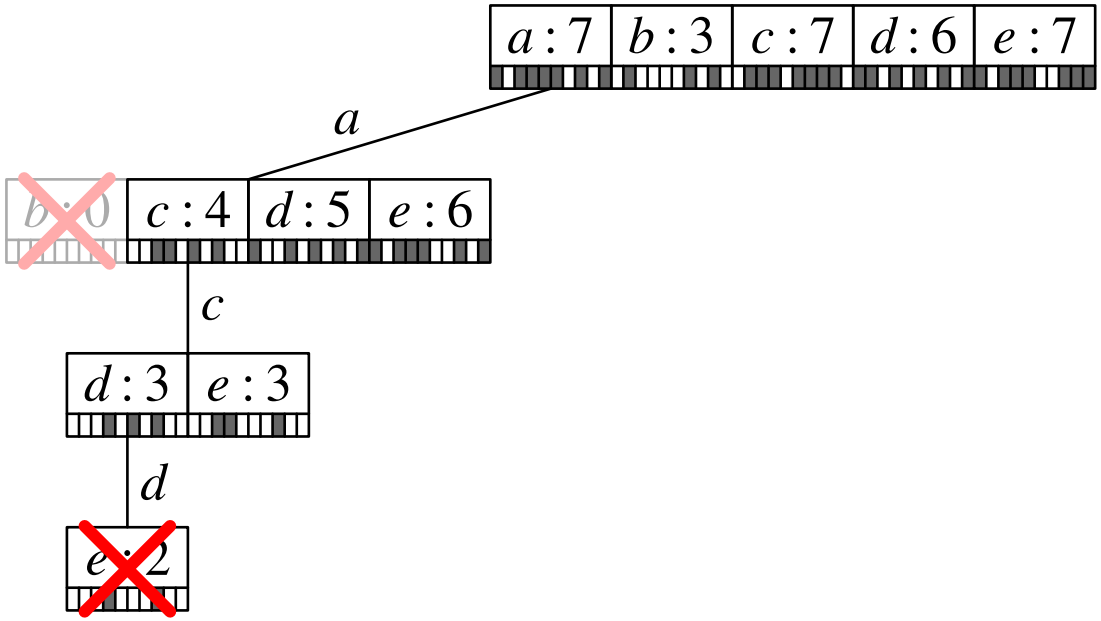
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

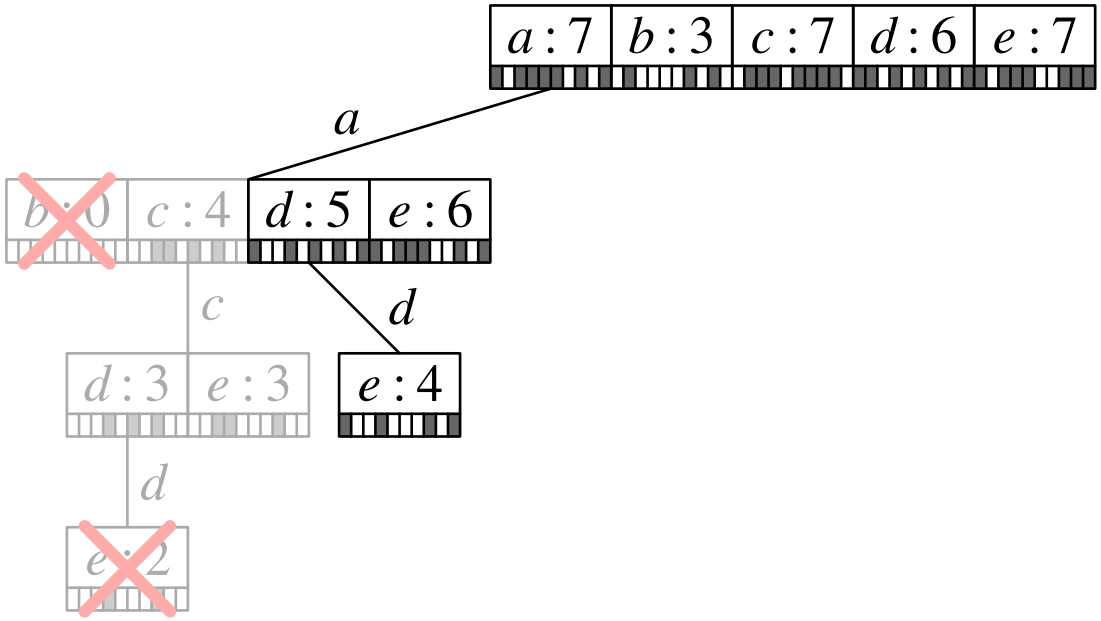
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

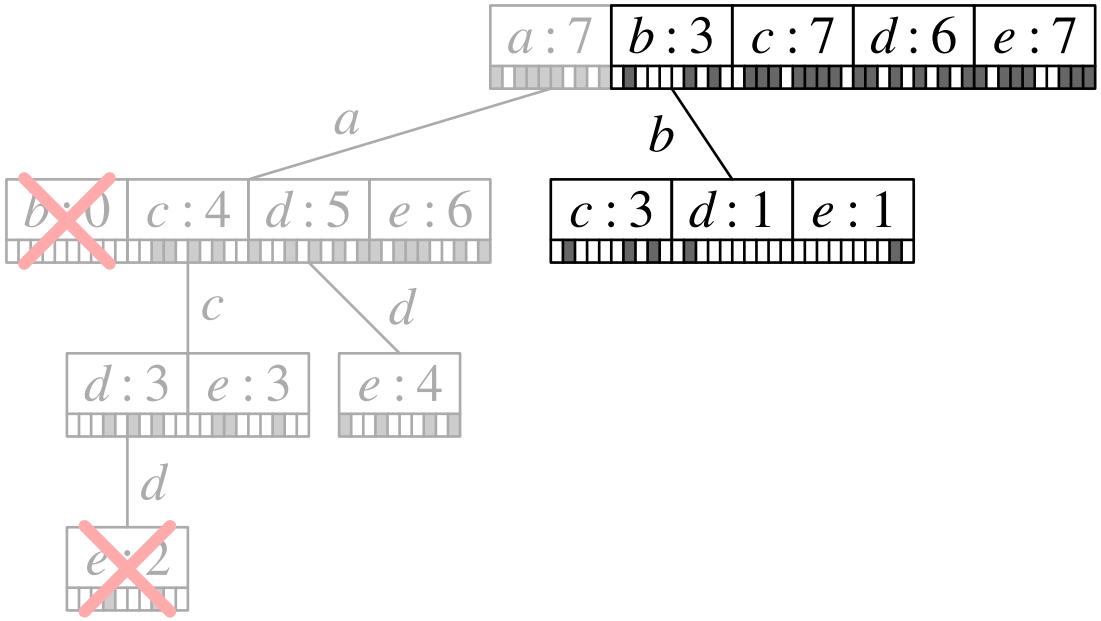
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The search backtracks to the second level of the search tree and intersects 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

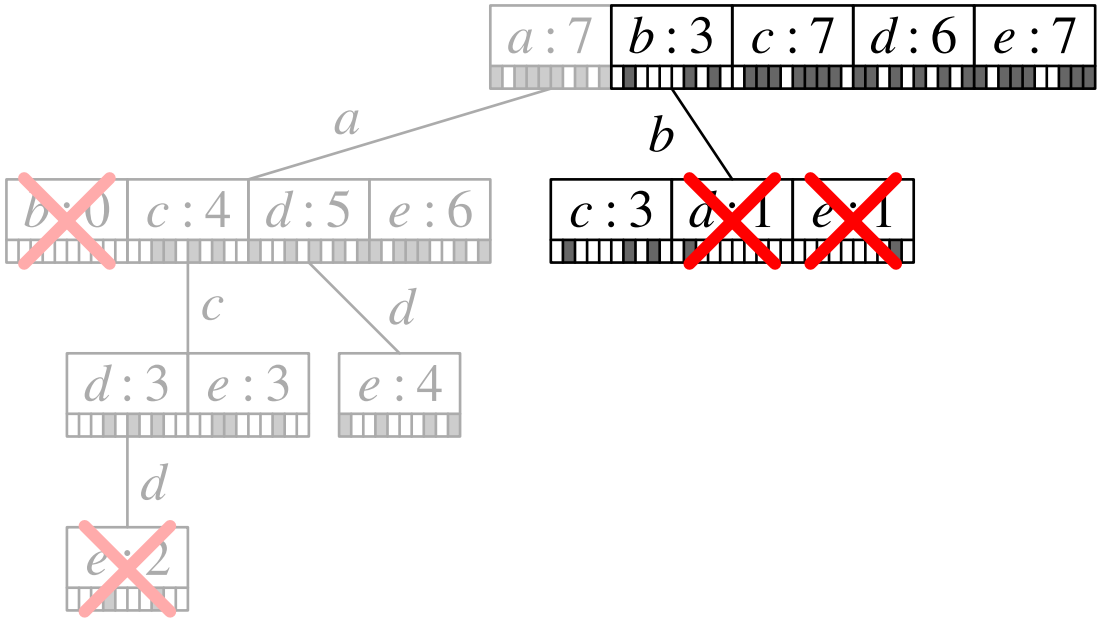
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The search backtracks to the first level of the search tree and intersects 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

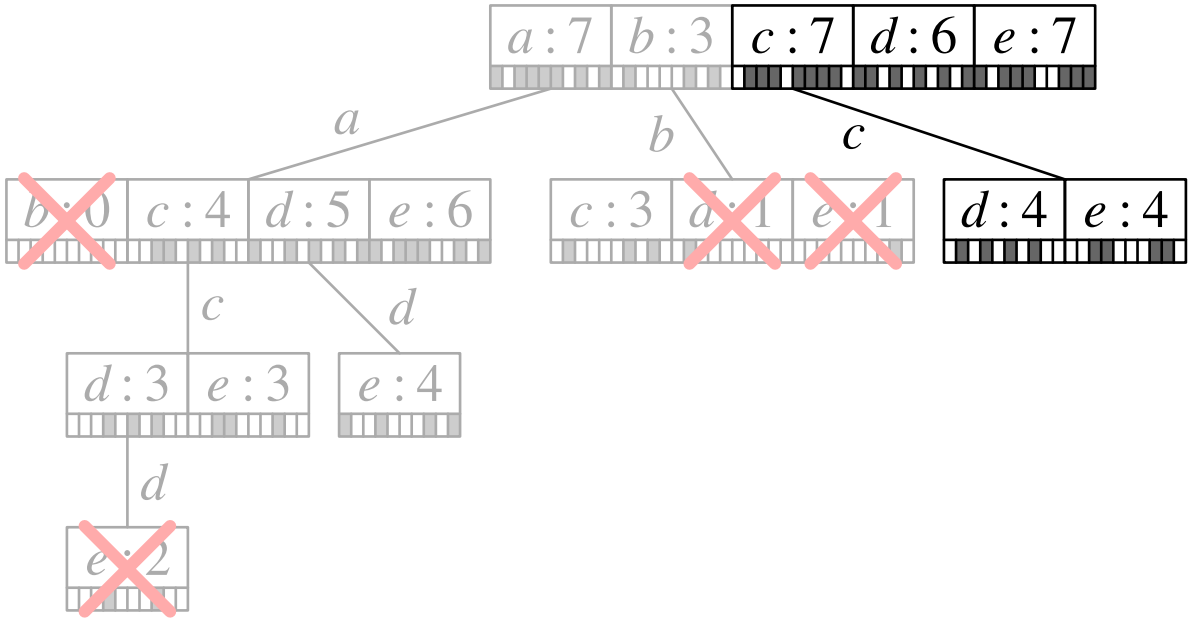
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

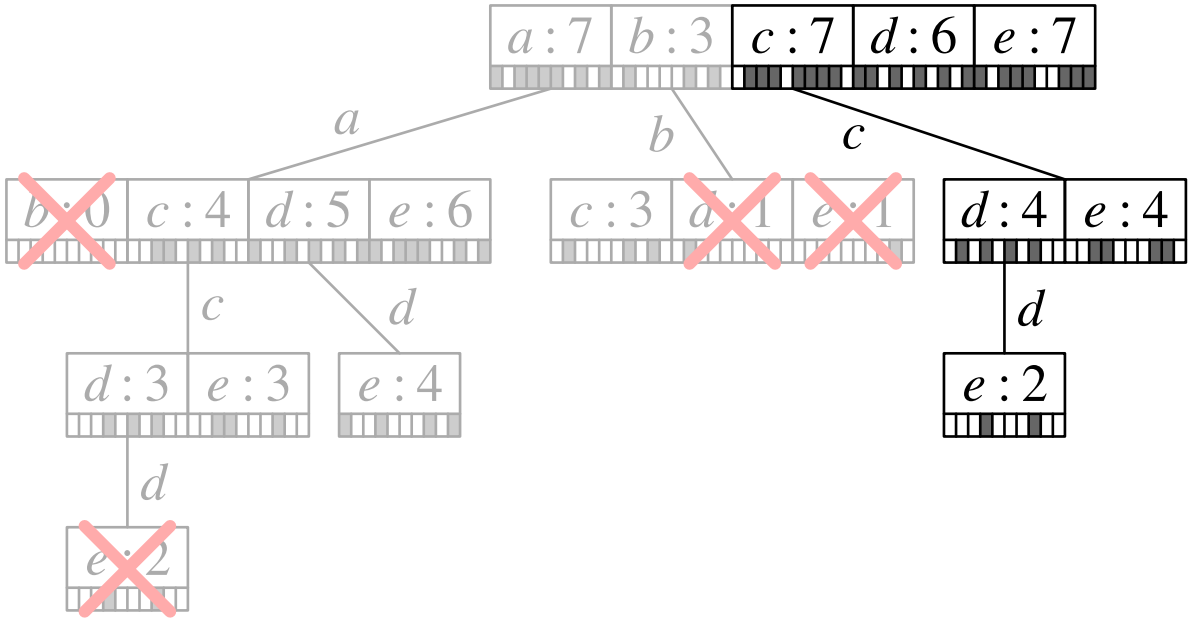
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

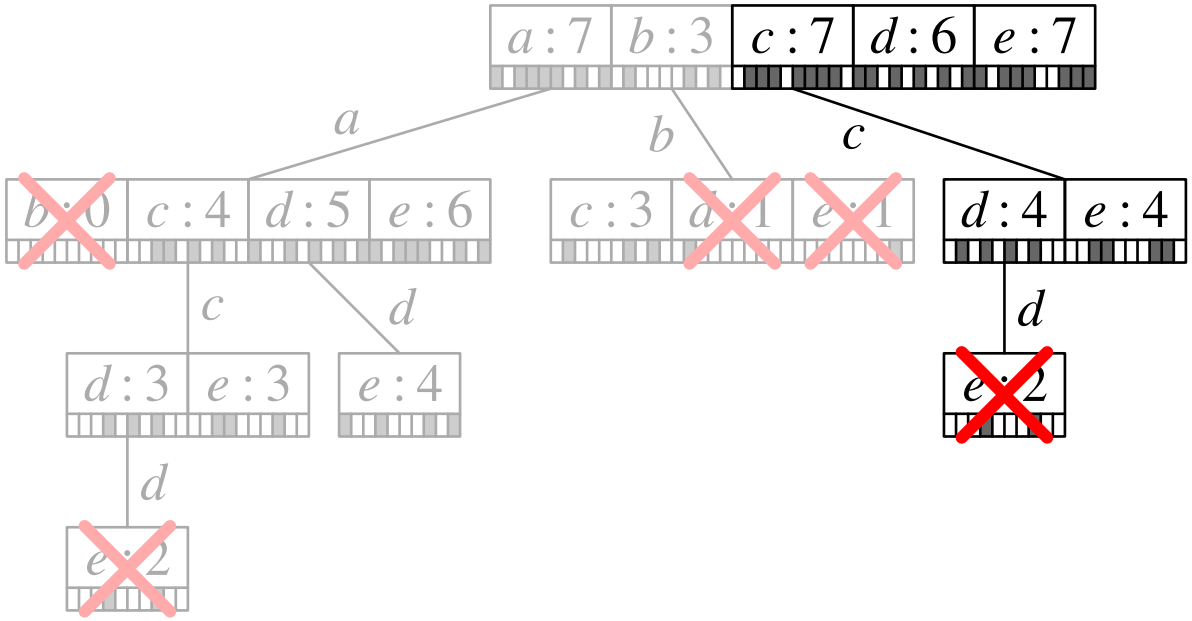
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

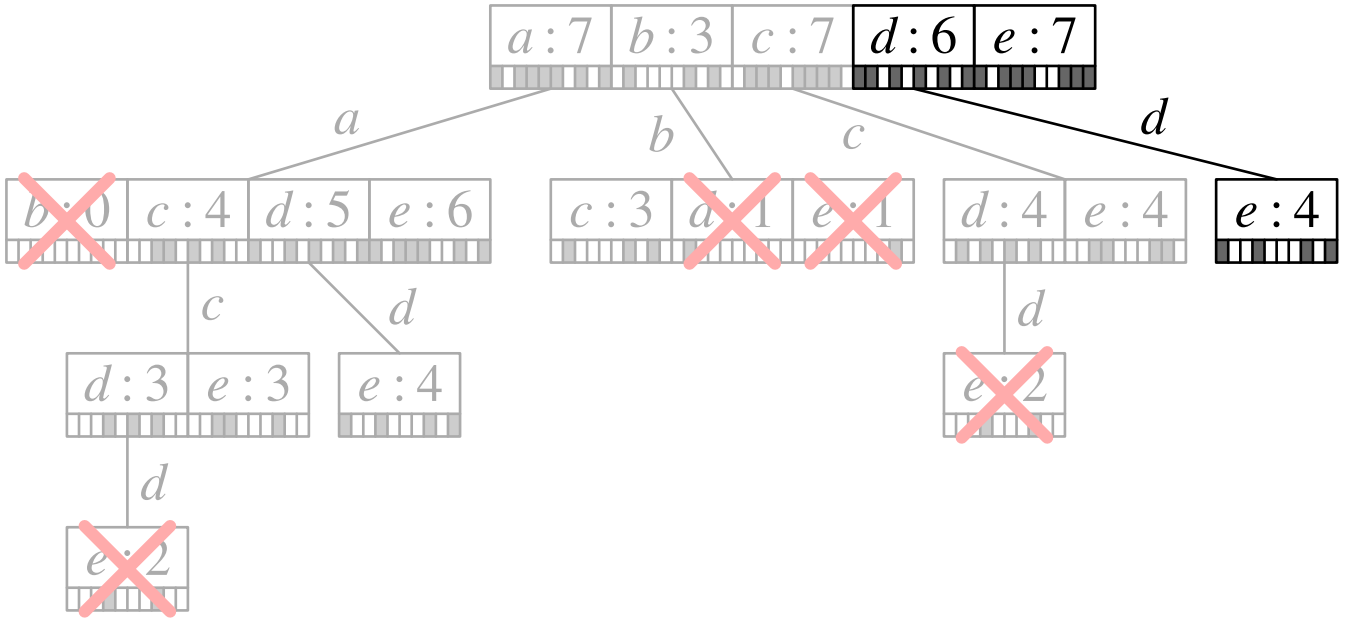
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

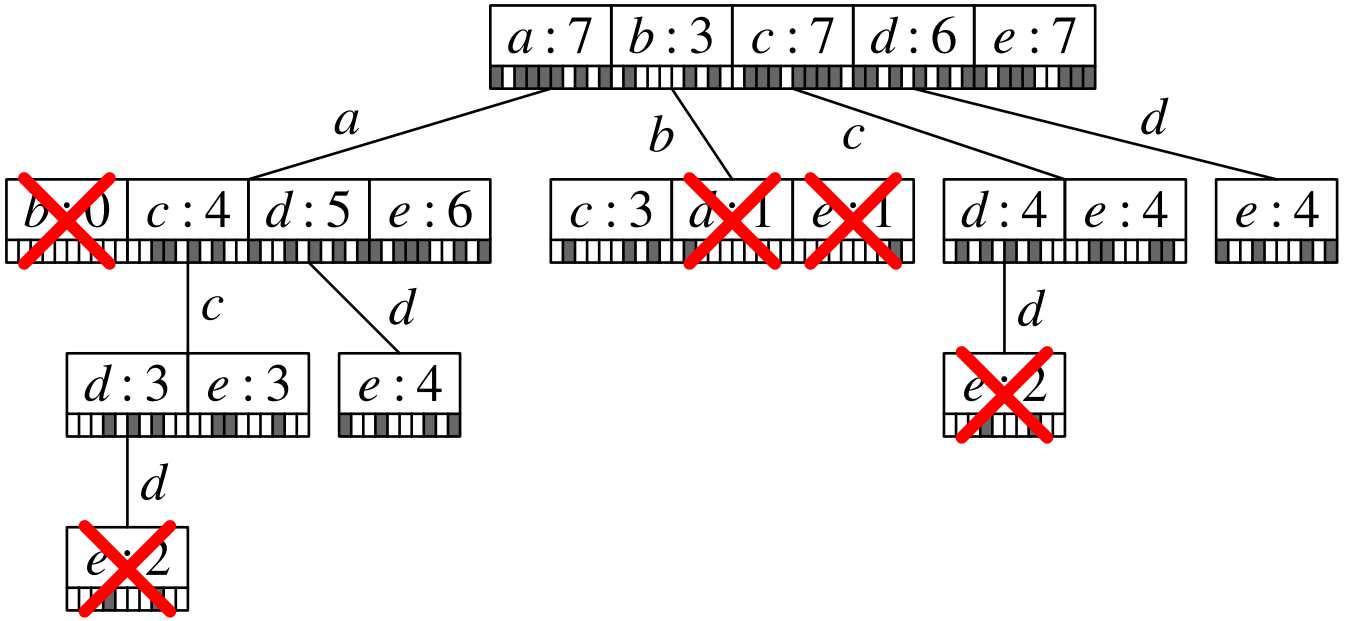
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- The search backtracks to the first level of the search tree and intersects 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 completed.

Eclat: Depth-First Search

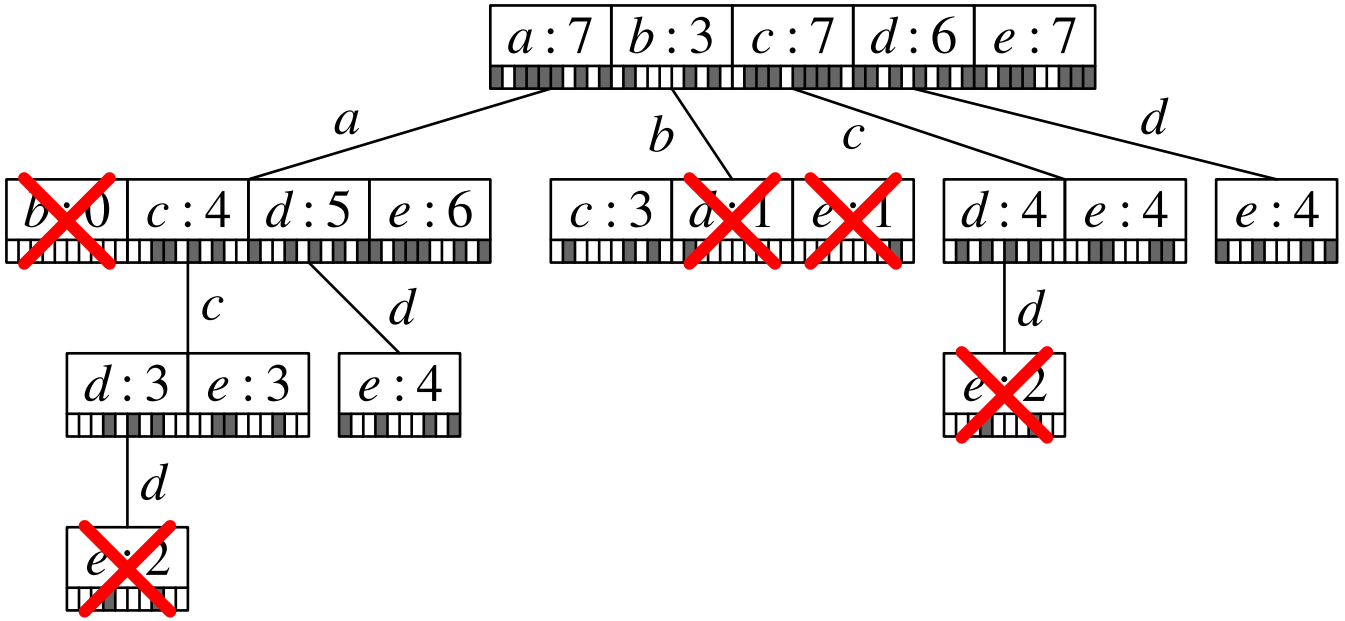
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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 potential gains justify the memory requirement.

Eclat: Representing Transaction Identifier Lists

Bit Matrix Representations

- Represent transactions as a bit matrix:
  - Each column corresponds to an item.
  - Each row corresponds to a transaction.
- Normal and sparse representation of bit matrices:
  - Normal: one memory bit per matrix bit
    (zeros are represented).
  - Sparse : lists of row indices of set bits (transaction identifier lists).
    (zeros are not represented)
- Which representation is preferable depends on
  the ratio of set bits to cleared bits.
- In most cases a sparse representation is preferable,
  because the intersections clear more and more bits.

Eclat: Intersecting Transaction Lists

```
function isect (src1, src2 : tidlist)
begin                                   (* — intersect two transaction id lists *)
   var dst : tidlist;                   (* created intersection *)
   while both src1 and src2 are not empty do begin
      if      head(src1) < head(src2)   (* skip transaction identifiers that are *)
      then  src1 = tail(src1);          (* unique to the first source list *)
      elseif head(src1) > head(src2)    (* skip transaction identifiers that are *)
      then  src2 = tail(src2);          (* unique to the second source list *)
      else begin                        (* if transaction id is in both sources *)
         dst.append(head(src1));        (* append it to the output list *)
         src1 = tail(src1); src2 = tail(src2);
      end;                              (* remove the transferred transaction id *)
   end;                                 (* from both source lists *)
   return dst;                          (* return the created intersection *)
end;     (* function isect() *)
```

Eclat: Filtering Transaction Lists

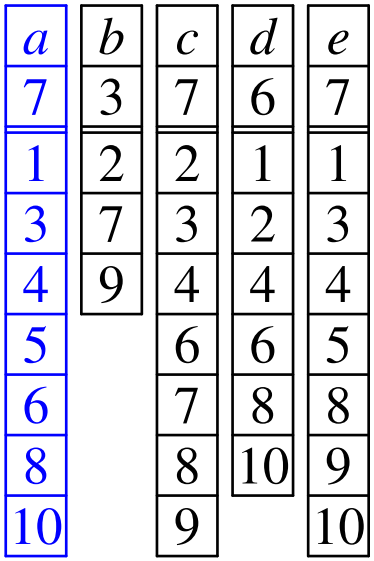
```
function filter (transdb : list of tidlist)
begin                                    (* — filter a transaction database *)
    var condb : list of tidlist;         (* created conditional transaction database *)
        out : tidlist;                   (* filtered tidlist of other item *)
    for tid in head(transdb) do          (* traverse the tidlist of the split item *)
        contained[tid] := true;          (* and set flags for contained tids *)
    for inp in tail(transdb) do begin    (* traverse tidlists of the other items *)
        out := new tidlist;              (* create an output tidlist and *)
        condb.append(out);               (* append it to the conditional database *)
        for tid in inp do                (* collect tids shared with split item *)
            if contained[tid] then out.append(tid);
    end                                  (* ("contained" is a global boolean array) *)
    for tid in head(transdb) do          (* traverse the tidlist of the split item *)
        contained[tid] := false;         (* and clear flags for contained tids *)
    return condb;                        (* return the created conditional database *)
end;     (* function filter() *)
```

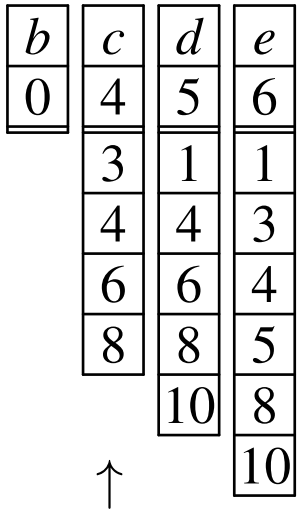
Eclat: Item Order

Consider **Eclat with transaction identifier lists** (sparse representation):

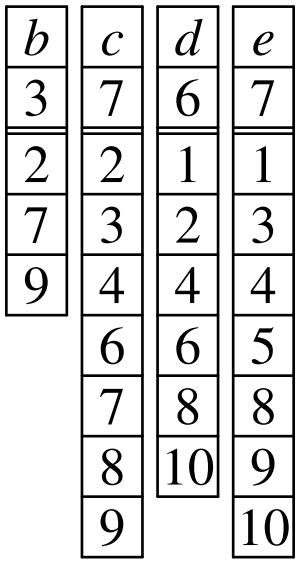
- Each computation of a conditional transaction database intersects the transaction list for an item (let this be list L) with all transaction lists for items following in the item order.
- The lists resulting from the intersections cannot be longer than the list L. (This is another form of the fact that support is anti-monotone.)
- If the **items are processed in the order of increasing frequency** (that is, if they are chosen as split items in this order):
  - Short lists (less frequent items) are intersected with many other lists, creating a conditional transaction database with many short lists.
  - Longer lists (more frequent items) are intersected with few other lists, creating a conditional transaction database with few long lists.
- Consequence: The average size of conditional transaction databases is reduced, which leads to **faster processing / search**.

Eclat: Item Order

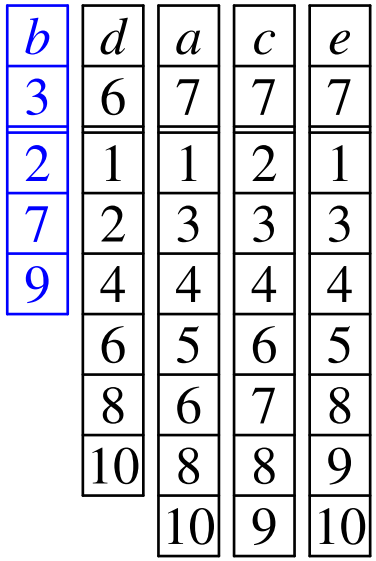


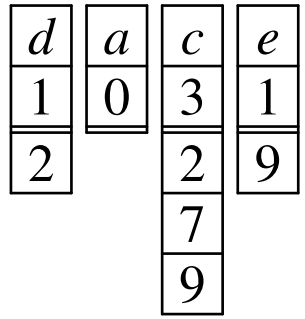


↑ Conditional database for prefix a (1st subproblem)

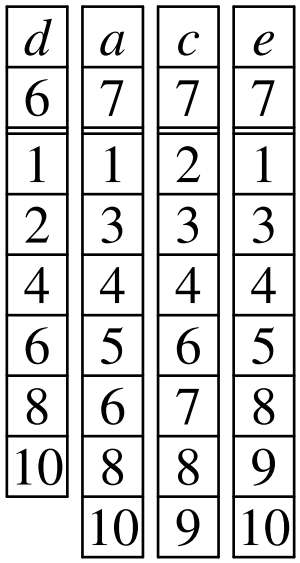


← Conditional database with item a removed (2nd subproblem)





↑ Conditional database for prefix b (1st subproblem)

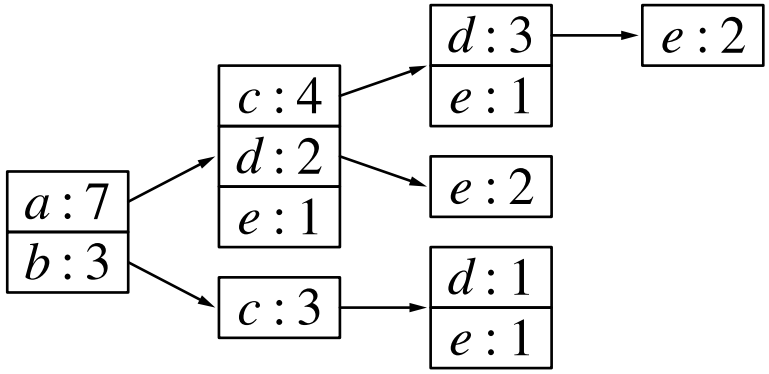


← Conditional database with item b removed (2nd subproblem)

Reminder (Apriori): Transactions as a Prefix Tree

transaction database	lexicographically sorted
a, d, e	a, c, d
b, c, d	a, c, d, e
a, c, e	a, c, d, e
a, c, d, e	a, c, e
a, e	a, d, e
a, c, d	a, d, e
b, c	a, e
a, c, d, e	b, c
b, c, e	b, c, d
a, d, e	b, c, e

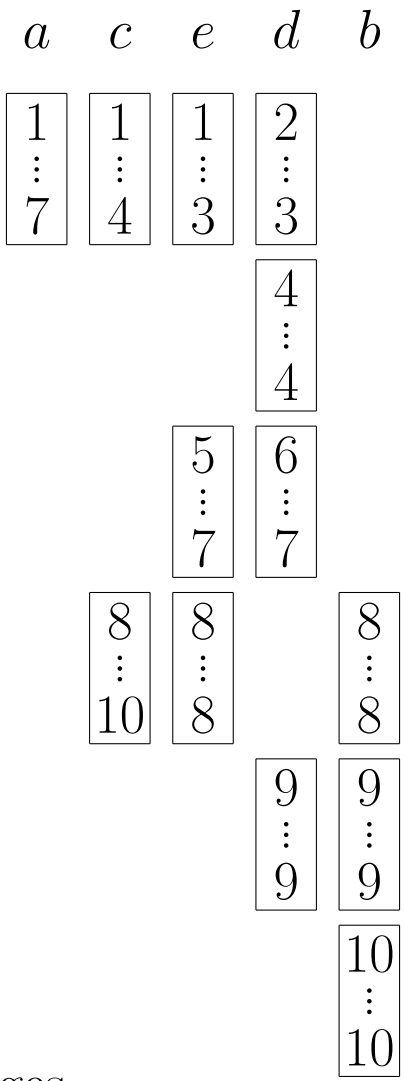
prefix tree representation



- Items in transactions are sorted w.r.t. some arbitrary order, transactions are sorted lexicographically, then a prefix tree is constructed.
- **Advantage:** identical transaction prefixes are processed only once.

Eclat: Transaction Ranges

transaction database	item frequencies	sorted by frequency	lexicographically sorted
a, d, e	a: 7	a, e, d	1: a, c, e
b, c, d	b: 3	c, d, b	2: a, c, e, d
a, c, e	c: 7	a, c, e	3: a, c, e, d
a, c, d, e	d: 6	a, c, e, d	4: a, c, d
a, e	e: 7	a, e	5: a, e
a, c, d		a, c, d	6: a, e, d
b, c		c, b	7: a, e, d
a, c, d, e		a, c, e, d	8: c, e, b
b, c, e		c, e, b	9: c, d, b
a, d, e		a, e, d	10: c, b

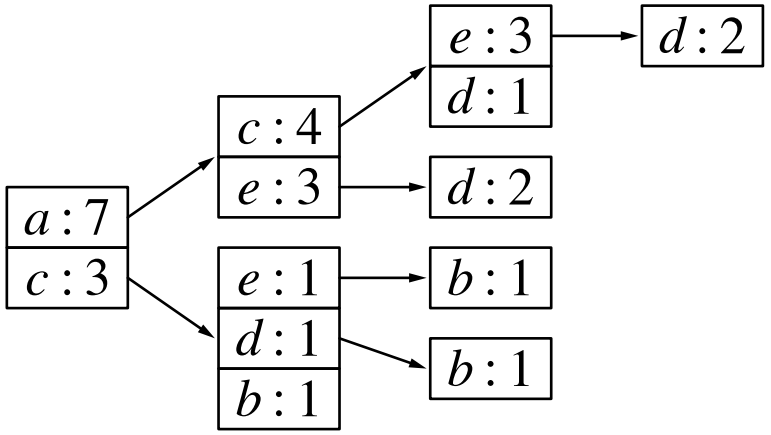


- The transaction lists can be compressed by combining consecutive transaction identifiers into ranges.
- Exploit item frequencies and ensure subset relations between ranges from lower to higher frequencies, so that intersecting the lists is easy.

Eclat: Transaction Ranges / Prefix Tree

transaction database	sorted by frequency	lexicographically sorted
a, d, e	a, e, d	1: a, c, e
b, c, d	c, d, b	2: a, c, e, d
a, c, e	a, c, e	3: a, c, e, d
a, c, d, e	a, c, e, d	4: a, c, d
a, e	a, e	5: a, e
a, c, d	a, c, d	6: a, e, d
b, c	c, b	7: a, e, d
a, c, d, e	a, c, e, d	8: c, e, b
b, c, e	c, e, b	9: c, d, b
a, d, e	a, e, d	10: c, b

prefix tree representation



- Items in transactions are sorted by frequency,
  transactions are sorted lexicographically, then a prefix tree is constructed.
- The transaction ranges reflect the structure of this prefix tree.

Eclat: Difference sets (Diffsets)

- In a conditional database, all transaction lists are "filtered" by the prefix: Only transactions contained in the transaction identifier list for the prefix can be in the transaction identifier lists of the conditional database.

- This suggests the idea to use **diffsets** to represent conditional databases:

$$\forall I : \forall a \notin I : \quad D_T(a \mid I) = K_T(I) - K_T(I \cup \{a\})$$

  $D_T(a \mid I)$ contains the identifiers of the transactions that contain I but not a.

- The support of direct supersets of I can now be computed as

$$\forall I : \forall a \notin I : \quad s_T(I \cup \{a\}) = s_T(I) - |D_T(a \mid I)|.$$

  The diffsets for the next level can be computed by

$$\forall I : \forall a, b \notin I, a \neq b : \quad D_T(b \mid I \cup \{a\}) = D_T(b \mid I) - D_T(a \mid I)$$

- For some transaction databases, using diffsets speeds up the search considerably.

Eclat: Diffsets

Proof of the Formula for the Next Level:

$$
\begin{aligned}
D_T(b \mid I \cup \{a\}) &= K_T(I \cup \{a\}) - K_T(I \cup \{a,b\}) \\
&= \{k \mid I \cup \{a\} \subseteq t_k\} - \{k \mid I \cup \{a,b\} \subseteq t_k\} \\
&= \{k \mid I \subseteq t_k \wedge a \in t_k\} \\
&\quad -\{k \mid I \subseteq t_k \wedge a \in t_k \wedge b \in t_k\} \\
&= \{k \mid I \subseteq t_k \wedge a \in t_k \wedge b \notin t_k\} \\
&= \{k \mid I \subseteq t_k \wedge b \notin t_k\} \\
&\quad -\{k \mid I \subseteq t_k \wedge b \notin t_k \wedge a \notin t_k\} \\
&= \{k \mid I \subseteq t_k \wedge b \notin t_k\} \\
&\quad -\{k \mid I \subseteq t_k \wedge a \notin t_k\} \\
&= (\{k \mid I \subseteq t_k\} - \{k \mid I \cup \{b\} \subseteq t_k\}) \\
&\quad -(\{k \mid I \subseteq t_k\} - \{k \mid I \cup \{a\} \subseteq t_k\}) \\
&= (K_T(I) - K_T(I \cup \{b\}) \\
&\quad -(K_T(I) - K_T(I \cup \{a\}) \\
&= D(b \mid I) - D(a \mid I)
\end{aligned}
$$

Summary Eclat

Basic Processing Scheme

- Depth-first traversal of the prefix tree (divide-and-conquer scheme).
- Data is represented as lists of transaction identifiers (one per item).
- Support counting is done by intersecting lists of transaction identifiers.

Advantages

- Depth-first search reduces memory requirements.
- Usually (considerably) faster than Apriori.

Disadvantages

- With a sparse transaction list representation (row indices)
  intersections are difficult to execute for modern processors (branch prediction).

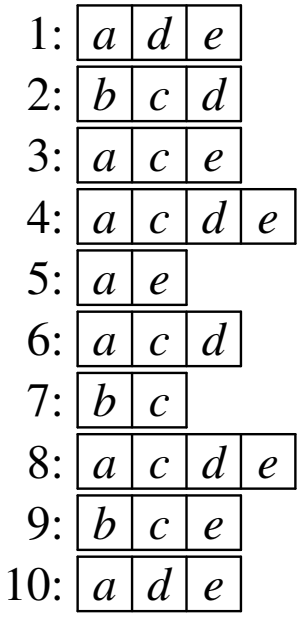
The LCM Algorithm

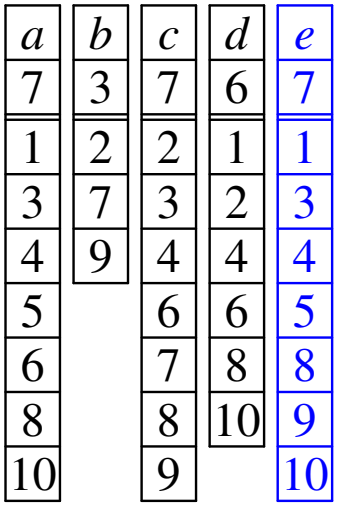
Linear Closed Item Set Miner
[Uno, Asai, Uchida, and Arimura 2003] (version 1)
[Uno, Kiyomi and Arimura 2004, 2005] (versions 2 & 3)

LCM: Basic Ideas

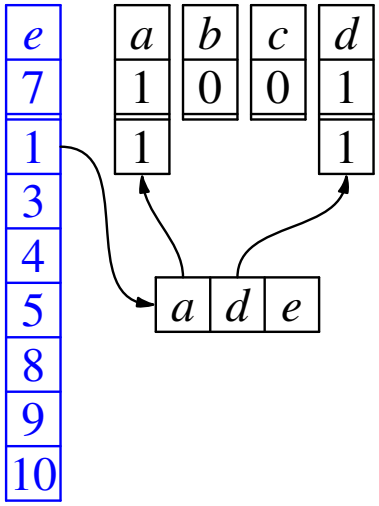
- The item sets are checked in **lexicographic order**
  (**depth-first traversal** of the prefix tree).
- Standard divide-and-conquer scheme (include/exclude items);
  recursive processing of the conditional transaction databases.
- Closely related to the Eclat algorithm.
- Maintains **both a horizontal and a vertical representation**
  of the transaction database in parallel.
  - Uses the vertical representation to filter the transactions
    with the chosen split item.
  - Uses the horizontal representation to fill the vertical representation
    for the next recursion step (no intersection as in Eclat).
- Usually traverses the search tree from **right to left**
  in order to reuse the memory for the vertical representation
  (fixed memory requirement, proportional to database size).

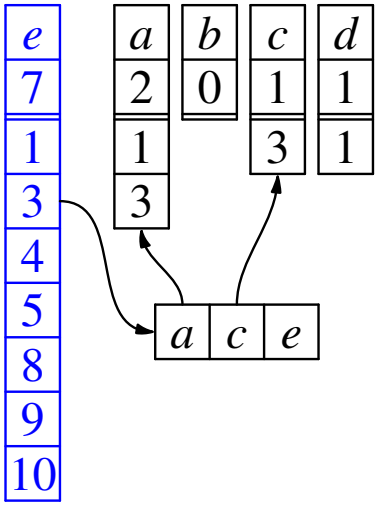
LCM: Occurrence Deliver

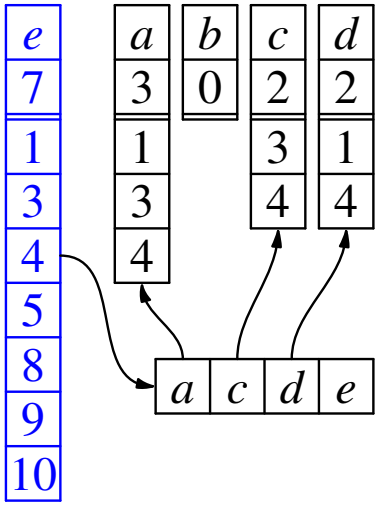




Occurrence deliver scheme used by LCM to find the conditional transaction database for the first subproblem (needs a horizontal representation in parallel).

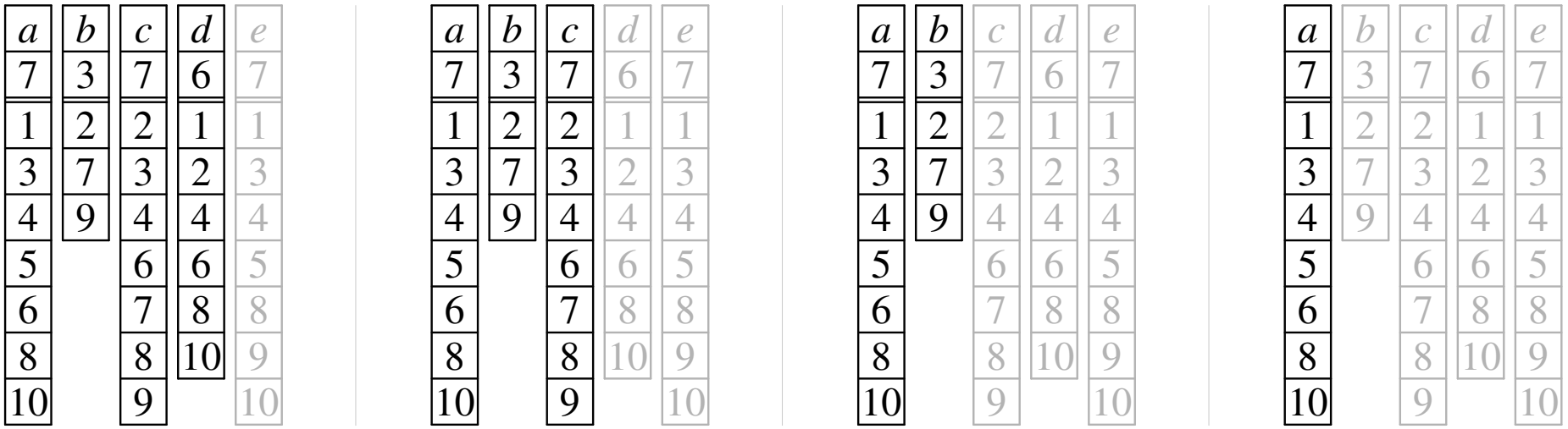






etc.

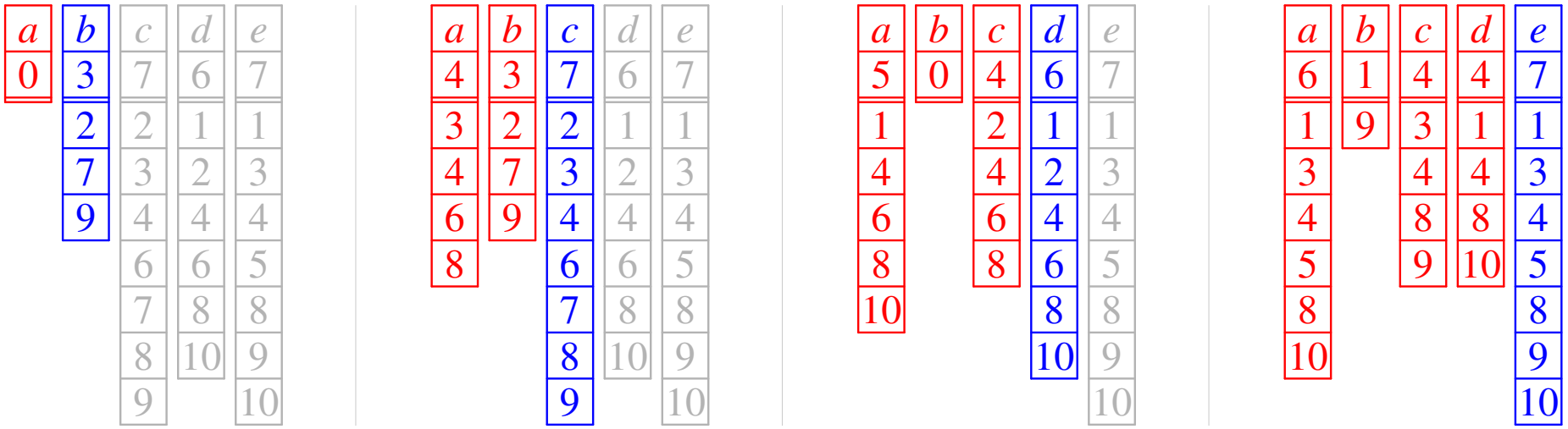
LCM: Solve 2nd Subproblem before 1st



gray: excluded item (2nd subproblem first) black: data needed for 2nd subproblem

- The second subproblem (exclude split item) is solved before the first subproblem (include split item).
- The algorithm is executed only on the memory that stores the initial vertical representation (plus the horizontal representation).
- If the transaction database can be loaded, the frequent item sets can be found.

LCM: Solve 2nd Subproblem before 1st



gray: unprocessed part blue: split item red: conditional database

- The second subproblem (exclude split item) is solved before the first subproblem (include split item).
- The algorithm is executed only on the memory that stores the initial vertical representation (plus the horizontal representation).
- If the transaction database can be loaded, the frequent item sets can be found.

Summary LCM

Basic Processing Scheme

- Depth-first traversal of the prefix tree (divide-and-conquer scheme).
- Parallel horizontal and vertical transaction representation.
- Support counting is done during the occurrence deliver process.

Advantages

- Fairly simple data structure and processing scheme.
- Very fast if implemented properly (and with additional tricks).

Disadvantages

- Simple, straightforward implementation is relatively slow.

The SaM Algorithm

Split and Merge Algorithm [Borgelt 2008]

SaM: Basic Ideas

- The item sets are checked in **lexicographic order** (**depth-first traversal** of the prefix tree).
- Standard divide-and-conquer scheme (include/exclude items).
- Recursive processing of the conditional transaction databases.
- While Eclat uses a purely vertical transaction representation, SaM uses a purely **horizontal transaction representation**.

  This demonstrates that the traversal order for the prefix tree and the representation form of the transaction database can be combined freely.
- The data structure used is a simply array of transactions.
- The two conditional databases for the two subproblems formed in each step are created with a **split step** and a **merge step**.

  Due to these steps the algorithm is called Split and Merge (SaM).

SaM: Preprocessing the Transaction Database

① *a d*
a c d e
b d
b c d g
b c f
a b d
b d e
b c d e
b c
a b d f

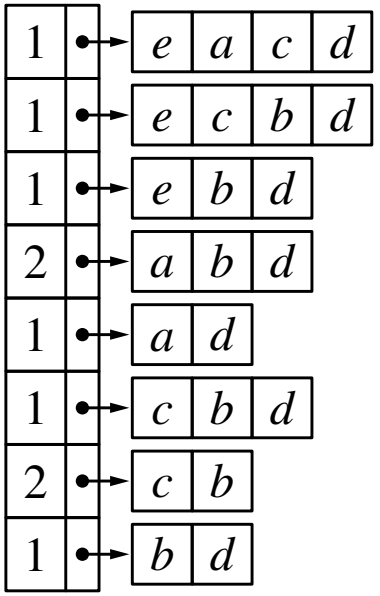
② *g*: 1
f: 2
e: 3
a: 4
c: 5
b: 8
d: 8

$s_{\min} = 3$

③ *a d*
e a c d
b d
c b d
c b
a b d
e b d
e c b d
c b
a b d

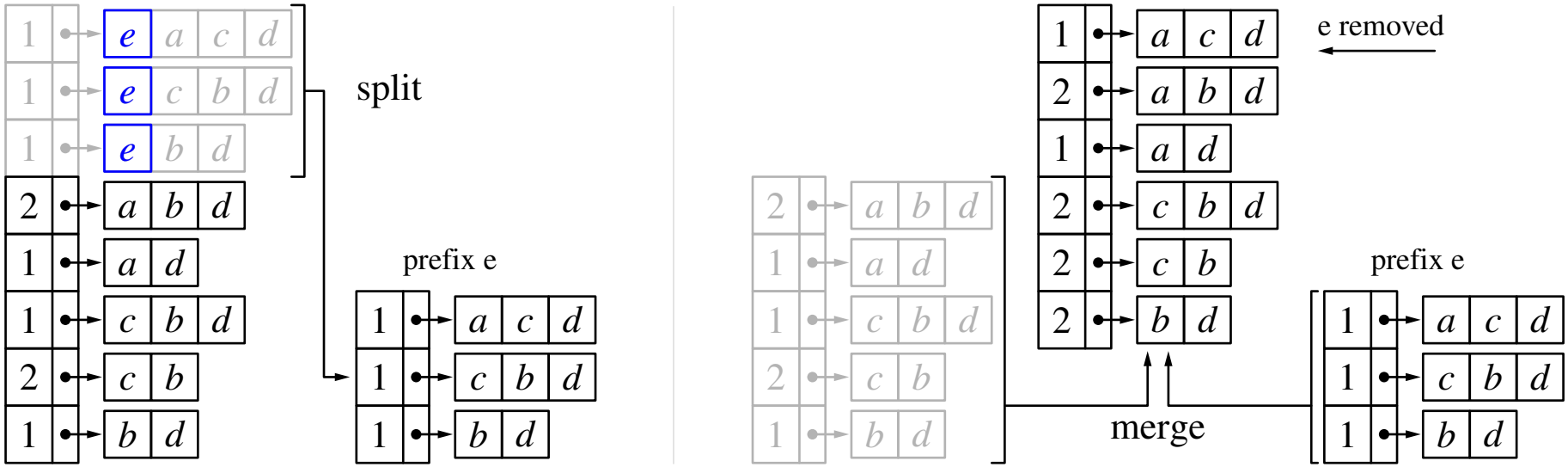
④ *e a c d*
e c b d
e b d
a b d
a b d
a d
c b d
c b
c b
b d

⑤



1. Original transaction database.
2. Frequency of individual items.
3. Items in transactions sorted ascendingly w.r.t. their frequency.
4. Transactions sorted lexicographically in descending order (comparison of items inverted w.r.t. preceding step).
5. Data structure used by the algorithm.

SaM: Basic Operations



- **Split Step:** (on the left; for first subproblem)
  - Move all transactions starting with the same item to a new array.
  - Remove the common leading item (advance pointer into transaction).
- **Merge Step:** (on the right; for second subproblem)
  - Merge the remainder of the transaction array and the copied transactions.
  - The merge operation is similar to a *mergesort* phase.

SaM: Pseudo-Code

```
function SaM (a: array of transactions,     (* conditional database to process *)
              p: set of items,              (* prefix of the conditional database a *)
              s_min: int)                   (* minimum support of an item set *)
var i: item;                                (* buffer for the split item *)
    b: array of transactions;               (* split result *)
begin                                       (* --- split and merge recursion --- *)
  while a is not empty do                   (* while the database is not empty *)
    i := a[0].items[0];                     (* get leading item of first transaction *)
    move transactions starting with i to b; (* split step: first subproblem *)
    merge b and the remainder of a into a;  (* merge step: second subproblem *)
    if s(i) ≥ s_min then                    (* if the split item is frequent: *)
      p := p ∪ {i};                         (* extend the prefix item set and *)
      report p with support s(i);           (* report the found frequent item set *)
      SaM(b, p, s_min);                     (* process the split result recursively, *)
      p := p − {i};                         (* then restore the original prefix *)
    end;
  end;
end;   (* function SaM() *)
```

SaM: Pseudo-Code — Split Step

```
var i: item;                                  (* buffer for the split item *)
    s: int;                                   (* support of the split item *)
    b: array of transactions;                 (* split result *)
begin                                         (* — split step *)
  b := empty; s := 0;                         (* initialize split result and item support *)
  i := a[0].items[0];                         (* get leading item of first transaction *)
  while a is not empty                        (* while database is not empty and *)
  and   a[0].items[0] = i do                  (* next transaction starts with same item *)
    s := s + a[0].wgt;                        (* sum occurrences (compute support) *)
    remove i from a[0].items;                 (* remove split item from transaction *)
    if      a[0].items is not empty           (* if transaction has not become empty *)
    then remove a[0] from a and append it to b;
    else  remove a[0] from a; end;            (* move it to the conditional database, *)
  end;                                        (* otherwise simply remove it: *)
end;                                          (* empty transactions are eliminated *)
```

- Note that the split step also determines the support of the item i.

SaM: Pseudo-Code — Merge Step

```
var c: array of transactions;                (* buffer for remainder of source array *)
begin                                        (* — merge step *)
  c := a; a := empty;                        (* initialize the output array *)
  while b and c are both not empty do        (* merge split and remainder of database *)
    if      c[0].items > b[0].items          (* copy lex. smaller transaction from c *)
    then    remove c[0] from c and append it to a;
    else if c[0].items < b[0].items          (* copy lex. smaller transaction from b *)
    then    remove b[0] from b and append it to a;
    else    b[0].wgt := b[0].wgt +c[0].wgt;  (* sum the occurrences/weights *)
            remove b[0] from b and append it to a;
            remove c[0] from c;              (* move combined transaction and *)
    end;                                     (* delete the other, equal transaction: *)
  end;                                       (* keep only one copy per transaction *)
  while c is not empty do                    (* copy remaining transactions in c *)
    remove c[0] from c and append it to a; end;
  while b is not empty do                    (* copy remaining transactions in b *)
    remove b[0] from b and append it to a; end;
end;                                         (* second recursion: executed by loop *)
```

SaM: Optimization

- If the transaction database is sparse,
  the two transaction arrays to merge can differ substantially in size.

- In this case SaM can become fairly slow,
  because the merge step processes many more transactions than the split step.

- Intuitive explanation (extreme case):
  - Suppose *mergesort* always merged a single element
    with the recursively sorted remainder of the array (or list).
  - This version of mergesort would be equivalent to *insertion sort*.
  - As a consequence the time complexity worsens from $O(n \log n)$ to $O(n^2)$.

- Possible optimization:
  - Modify the merge step if the arrays to merge differ significantly in size.
  - Idea: use the same optimization as in **binary search** based *insertion sort*.

SaM: Pseudo-Code — Binary Search Based Merge

```
function merge (a, b: array of transactions) : array of transactions
var l, m, r: int;                          (* binary search variables *)
    c: array of transactions;              (* output transaction array *)
begin                                      (* — binary search based merge — *)
  c := empty;                              (* initialize the output array *)
  while a and b are both not empty do      (* merge the two transaction arrays *)
    l := 0; r := length(a);                (* initialize the binary search range *)
    while l < r do                         (* while the search range is not empty *)
      m := ⌊(l+r)/2⌋;                      (* compute the middle index *)
      if      a[m] < b[0]                  (* compare the transaction to insert *)
      then l := m + 1; else r := m;        (* and adapt the binary search range *)
    end;                                   (* according to the comparison result *)
    while l > 0 do                         (* while still before insertion position *)
      remove a[0] from a and append it to c;
      l := l − 1;                          (* copy lex. larger transaction and *)
    end;                                   (* decrement the transaction counter *)
    ...
```

SaM: Pseudo-Code — Binary Search Based Merge

```
  ...
  remove b[0] from b and append it to c;  (* copy the transaction to insert and *)
  i := length(c) − 1;                     (* get its index in the output array *)
  if      a is not empty and a[0].items = c[i].items
  then c[i].wgt = c[i].wgt +a[0].wgt;     (* if there is another transaction *)
       remove a[0] from a;                (* that is equal to the one just copied, *)
  end;                                    (* then sum the transaction weights *)
end;                                      (* and remove trans. from the array *)
while a is not empty do                   (* copy remainder of transactions in a *)
  remove a[0] from a and append it to c; end;
while b is not empty do                   (* copy remainder of transactions in b *)
  remove b[0] from b and append it to c; end;
return c;                                 (* return the merge result *)
end;   (* function merge() *)
```

- Applying this merge procedure if the length ratio of the transaction arrays exceeds 16:1 accelerates the execution on sparse data sets.

SaM: Optimization and External Storage

- Accepting a slightly more complicated processing scheme,
  one may work with **double source buffering**:
  - Initially, one source is the input database and the other source is empty.
  - A split result, which has to be created by moving and merging transactions from both sources, is always merged to the smaller source.
  - If both sources have become large,
    they may be merged in order to empty one source.

- Note that SaM can easily be implemented to work on **external storage**:
  - In principle, the transactions need not be loaded into main memory.
  - Even the transaction array can easily be stored on external storage
    or as a relational database table.
  - The fact that the transaction array is processed linearly
    is advantageous for external storage operations.

Summary SaM

Basic Processing Scheme

- Depth-first traversal of the prefix tree (divide-and-conquer scheme).
- Data is represented as an array of transactions (purely horizontal representation).
- Support counting is done implicitly in the split step.

Advantages

- Very simple data structure and processing scheme.
- Easy to implement for operation on external storage / relational databases.

Disadvantages

- Can be slow on sparse transaction databases due to the merge step.

The RElim Algorithm

Recursive Elimination Algorithm [Borgelt 2005]

Recursive Elimination: Basic Ideas

- The item sets are checked in **lexicographic order**
  (**depth-first traversal** of the prefix tree).
- Standard divide-and-conquer scheme (include/exclude items).
- Recursive processing of the conditional transaction databases.
- Avoids the main problem of the SaM algorithm:
  does not use a merge operation to group transactions with the same leading item.
- RElim rather maintains **one list of transactions per item**,
  thus employing the core idea of *radix sort*.

  However, only transactions starting with an item are in the corresponding list.
- After an item has been processed, transactions are reassigned to other lists
  (based on the next item in the transaction).
- RElim is in several respects similar to the LCM algorithm (as discussed before)
  and closely related to the H-mine algorithm (not covered in this lecture).

RElim: Preprocessing the Transaction Database

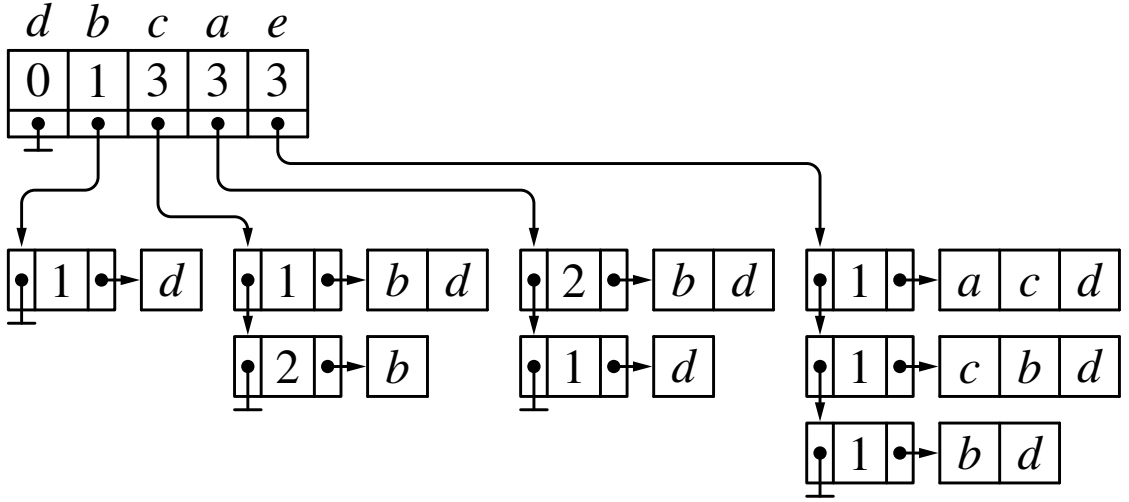
① ··· ③

same
as for
SaM

④

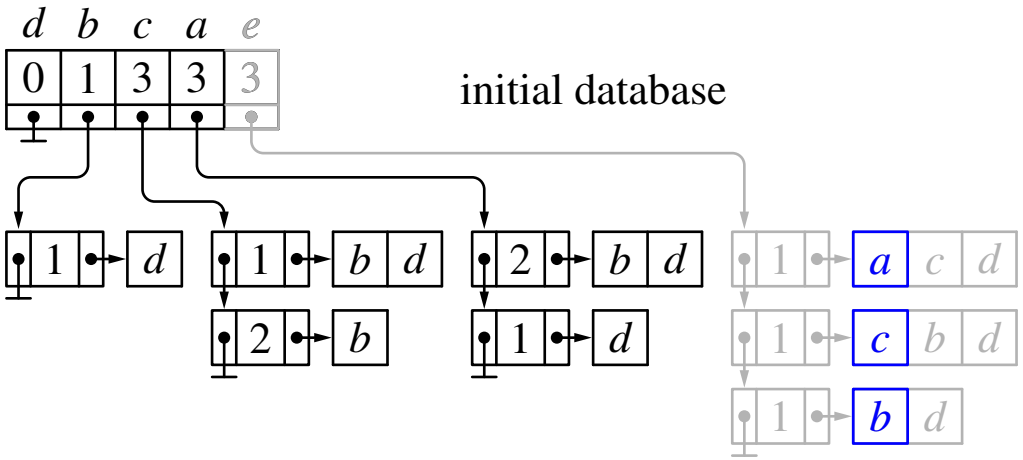
e a c d
e c b d
e b d
a b d
a b d
a d
c b d
c b
c b
b d

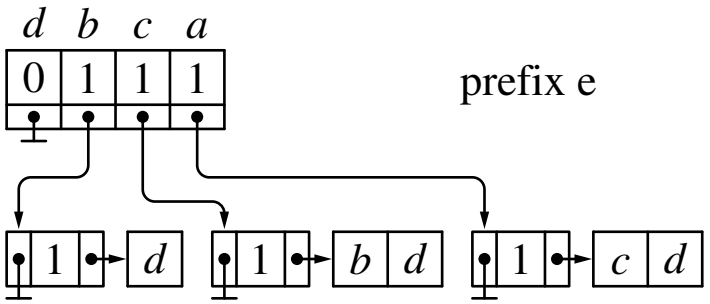
⑤

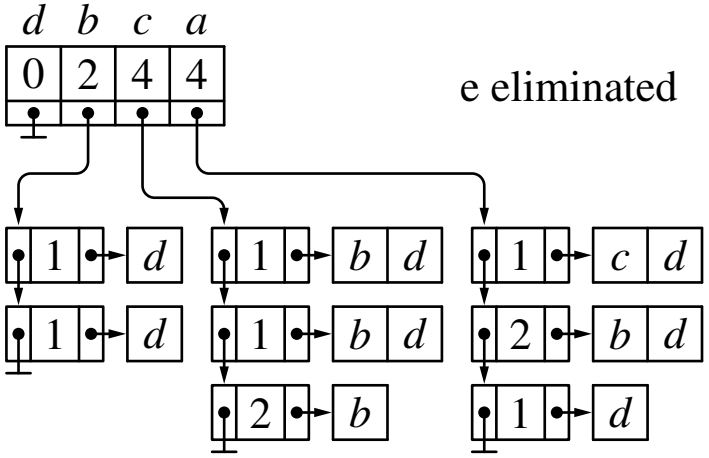


1. Original transaction database.
2. Frequency of individual items.
3. Items in transactions sorted ascendingly w.r.t. their frequency.
4. Transactions sorted lexicographically in descending order (comparison of items inverted w.r.t. preceding step).
5. Data structure used by the algorithm (leading items implicit in list).

RElim: Subproblem Split







The subproblem split of the RElim algorithm. The rightmost list is traversed and reassigned: once to an initially empty list array (conditional database for the prefix e, see top right) and once to the original list array (eliminating item e, see bottom left). These two databases are then both processed recursively.

- Note that after a simple reassignment there may be duplicate list elements.

RElim: Pseudo-Code

```
function RElim (a: array of transaction lists,    (* cond. database to process *)
                p: set of items,                  (* prefix of the conditional database a *)
                s_min: int) : int                 (* minimum support of an item set *)
var i, k: item;                                   (* buffer for the current item *)
    s: int;                                       (* support of the current item *)
    n: int;                                       (* number of found frequent item sets *)
    b: array of transaction lists;                (* conditional database for current item *)
    t, u: transaction list element;               (* to traverse the transaction lists *)
begin                                             (* — recursive elimination — *)
  n := 0;                                         (* initialize the number of found item sets *)
  while a is not empty do                         (* while conditional database is not empty *)
    i := last item of a; s := a[i].wgt;           (* get the next item to process *)
    if s ≥ s_min then                             (* if the current item is frequent: *)
      p := p ∪ {i};                               (* extend the prefix item set and *)
      report p with support s;                    (* report the found frequent item set *)
      ...                                         (* create conditional database for i *)
      p := p − {i};                               (* and process it recursively, *)
    end;                                          (* then restore the original prefix *)
```

RElim: Pseudo-Code

```
if s ≥ s_min then                    (* if the current item is frequent: *)
  ...                                (* report the found frequent item set *)
  b := array of transaction lists;   (* create an empty list array *)
  t := a[i].head;                    (* get the list associated with the item *)
  while t ≠ nil do                   (* while not at the end of the list *)
    u := copy of t; t := t.succ;     (* copy the transaction list element, *)
    k := u.items[0];                 (* go to the next list element, and *)
    remove k from u.items;           (* remove the leading item from the copy *)
    if u.items is not empty          (* add the copy to the conditional database *)
    then u.succ = b[k].head; b[k].head = u; end;
    b[k].wgt := b[k].wgt +u.wgt;     (* sum the transaction weight *)
  end;                               (* in the list weight/transaction counter *)
  n := n + 1 + RElim(b, p, s_min);   (* process the created database recursively *)
  ...                                (* and sum the found frequent item sets, *)
end;                                 (* then restore the original item set prefix *)
...                                  (* go on by reassigning *)
                                     (* the processed transactions *)
```

RElim: Pseudo-Code

```
    ...
    t := a[i].head;                    (* get the list associated with the item *)
    while t ≠ nil do                   (* while not at the end of the list *)
      u := t; t := t.succ;             (* note the current list element, *)
      k := u.items[0];                 (* go to the next list element, and *)
      remove k from u.items;           (* remove the leading item from current *)
      if u.items is not empty          (* reassign the noted list element *)
      then u.succ = a[k].head; a[k].head = u; end;
      a[k].wgt := a[k].wgt +u.wgt;     (* sum the transaction weight *)
    end;                               (* in the list weight/transaction counter *)
    remove a[i] from a;                (* remove the processed list *)
  end;
  return n;                            (* return the number of frequent item sets *)
end;   (* function RElim() *)
```

- In order to remove duplicate elements, it is usually advisable to sort and compress the next transaction list before it is processed.

The k-Items Machine

- Introduced with LCM algorithm (see above) to combine equal transaction suffixes.
- Idea: If the number of items is small, a *bucket/bin sort scheme* can be used to perfectly combine equal transaction suffixes.
- This scheme leads to the **k-items machine** (for small k).
  - All possible transaction suffixes are represented as bit patterns; one bucket/bin is created for each possible bit pattern.
  - A RElim-like processing scheme is employed (on a fixed data structure).
  - Leading items are extracted with a table that is indexed with the bit pattern.
  - Items are eliminated with a bit mask.

Table of highest set bits for a 4-items machine (special instructions: `bsr` / `lzcount`):

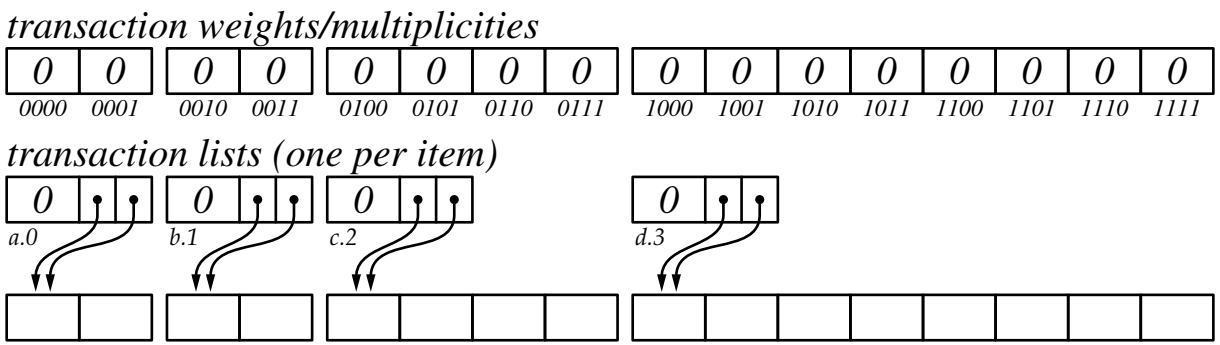
highest items/set bits of transactions (constant)

.	a.0	b.1	b.1	c.2	c.2	c.2	c.2	d.3	d.3	d.3	d.3	d.3	d.3	d.3	d.3
0000 ____	0001 ___a	0010 __b_	0011 __ba	0100 _c__	0101 _c_a	0110 _cb_	0111 _cba	1000 d___	1001 d__a	1010 d_b_	1011 d_ba	1100 dc__	1101 dc_a	1110 dcb_	1111 dcba

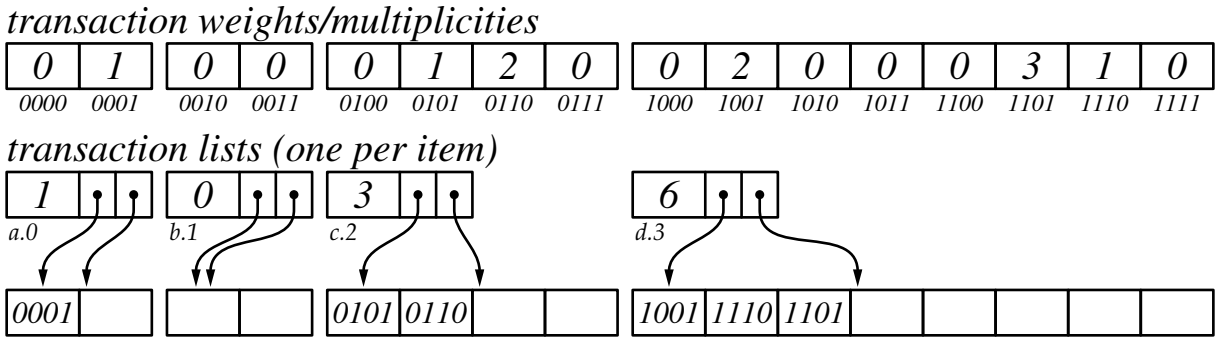
The k-items Machine

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: $\{a, d, e\}$

Empty 4-items machine (no transactions)



4-items machine after inserting the transactions

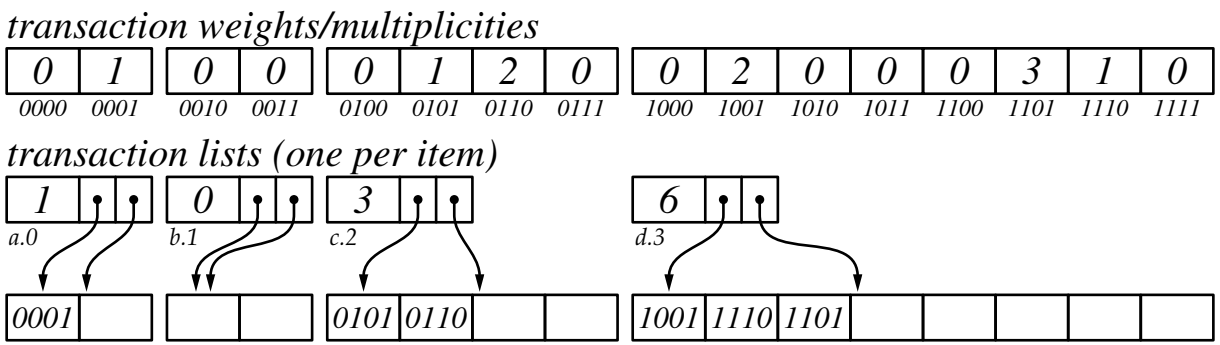


- In this state the 4-items machine represents a special form of the initial transaction database of the RElim algorithm.

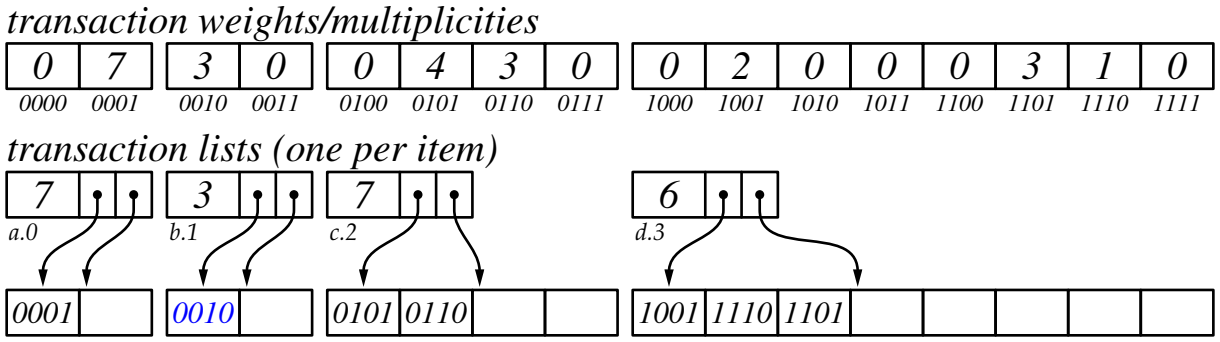
The k-items Machine

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: $\{a, d, e\}$

4-items machine after inserting the transactions



After propagating the transaction lists



- Propagating the transactions lists is equivalent to occurrence deliver.
- Conditional transaction databases are created as in RElim plus propagation.

Summary RElim

Basic Processing Scheme

- Depth-first traversal of the prefix tree (divide-and-conquer scheme).
- Data is represented as lists of transactions (one per item).
- Support counting is implicit in the (re)assignment step.

Advantages

- Fairly simple data structures and processing scheme.
- Competitive with the fastest algorithms despite this simplicity.

Disadvantages

- RElim is usually outperformed by LCM and FP-growth (discussed later).

The FP-Growth Algorithm

Frequent Pattern Growth Algorithm [Han, Pei, and Yin 2000]

FP-Growth: Basic Ideas

- FP-Growth means **Frequent Pattern Growth**.
- The item sets are checked in **lexicographic order**
  (**depth-first traversal** of the prefix tree).
- Standard divide-and-conquer scheme (include/exclude items).
- Recursive processing of the conditional transaction databases.
- The transaction database is represented as an **FP-tree**.
  An FP-tree is basically a **prefix tree** with additional structure:
  nodes of this tree that correspond to the same item are linked into lists.

  This **combines a horizontal and a vertical database representation**.
- This data structure is used to compute conditional databases efficiently.

  All transactions containing a given item can easily be found
  by the links between the nodes corresponding to this item.

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	$d\ b\ e$	$d\ c\ a\ e$	
$b\ c\ e\ g$		$b\ c\ e$	$d\ a$	
$c\ d\ f$		$d\ c$	$b\ c$	
$a\ b\ d$	$s_{\min} = 3$	$d\ b\ a$	$b\ c\ e$	

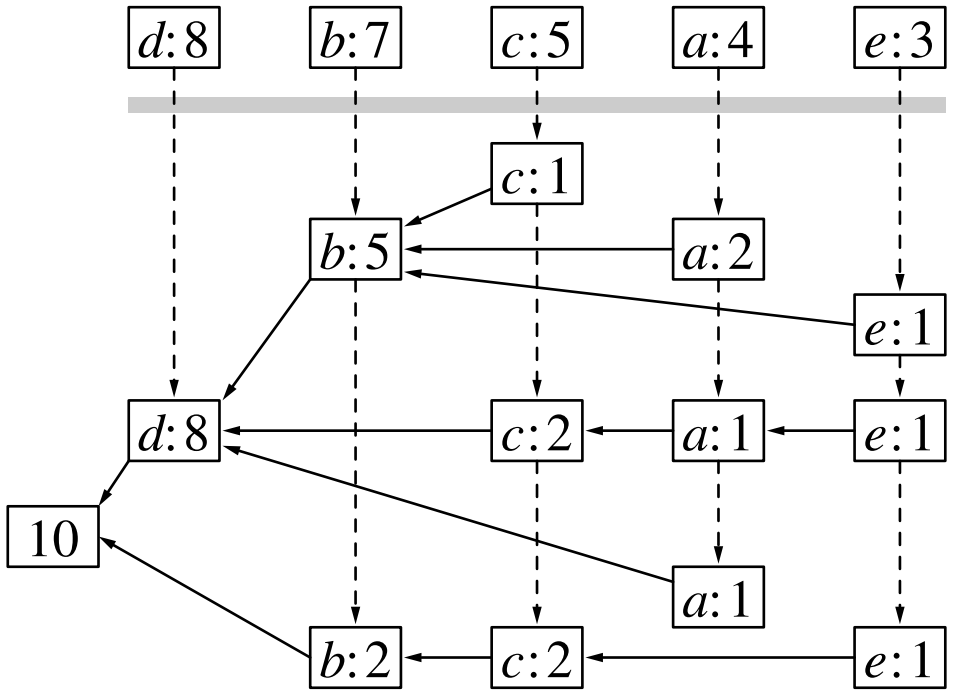
1. Original transaction database.
2. Frequency of individual items.
3. Items in transactions sorted descendingly w.r.t. their frequency and infrequent items removed.
4. Transactions sorted lexicographically in ascending order (comparison of items is the same as in preceding step).
5. Data structure used 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 the branches* that link nodes with the same item and a *header table* for the resulting item lists).
- 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*



frequent pattern tree

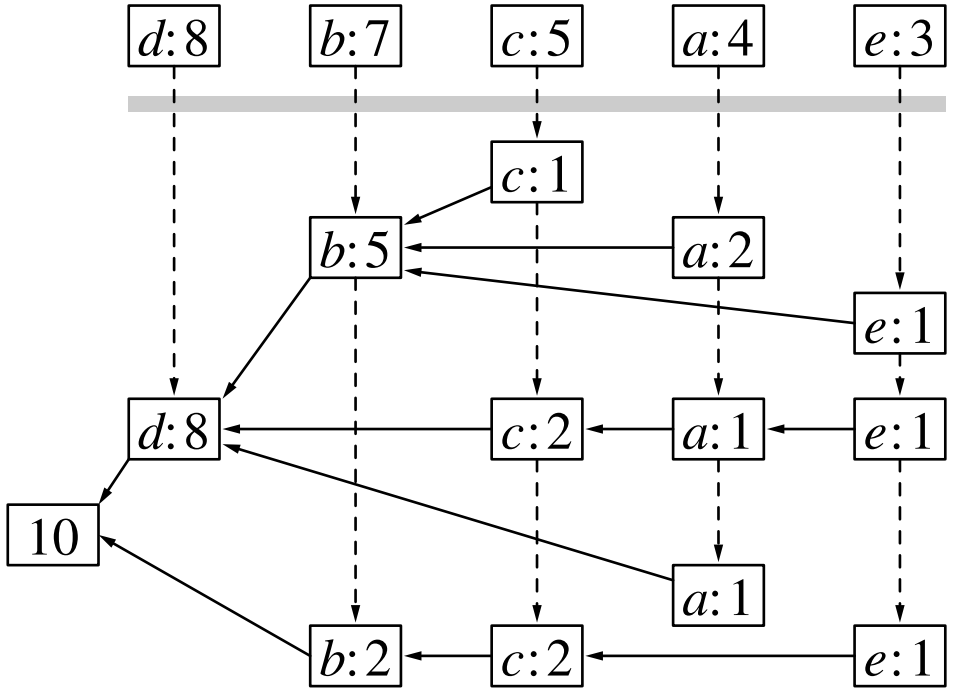
Transaction Representation: FP-Tree

- An FP-tree combines a horizontal and a vertical transaction representation.
- **Horizontal Representation:** prefix tree of transactions
  Vertical Representation: links between the prefix tree branches

Note: the prefix tree is inverted,
i.e. there are only parent pointers.

Child pointers are not needed
due to the processing scheme
(to be discussed).

In principle, all nodes referring
to the same item can be stored
in an array rather than a list.

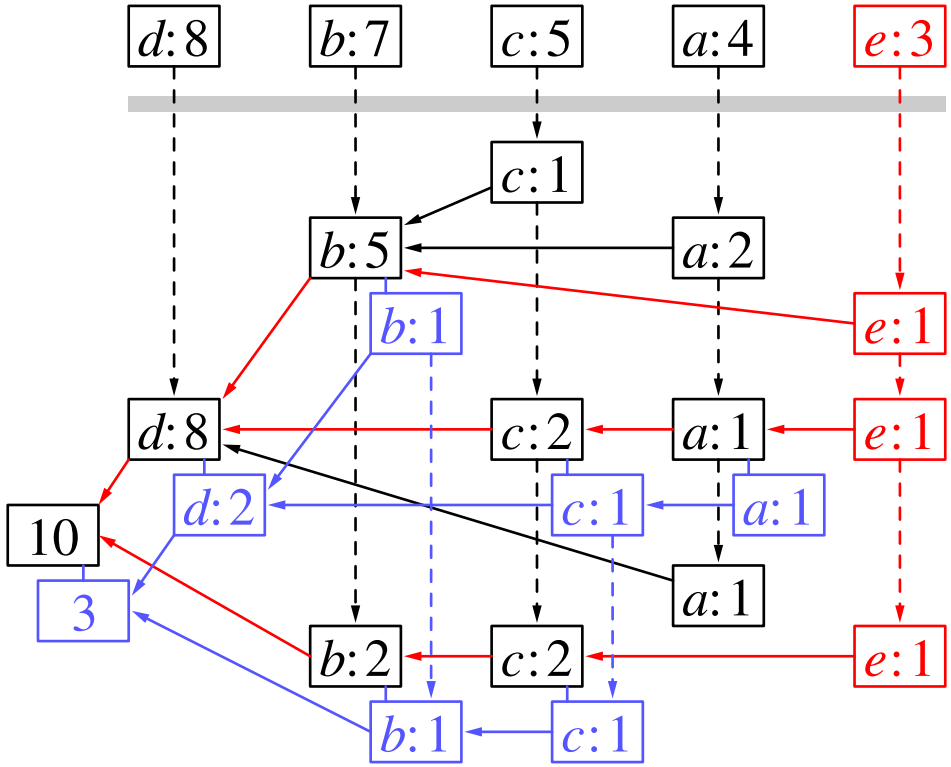


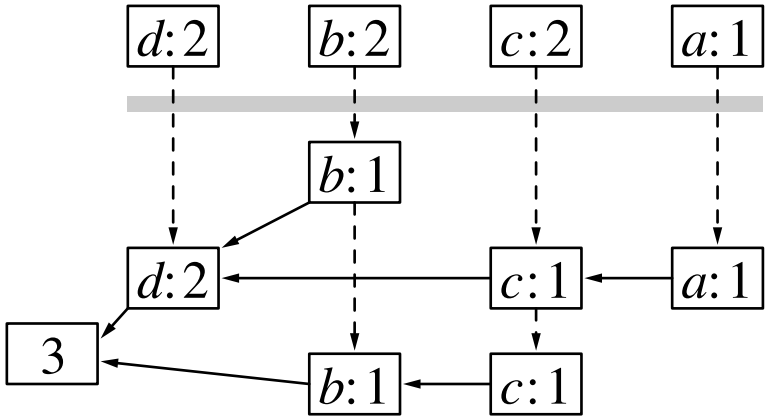
frequent pattern tree

Recursive Processing

- The initial FP-tree is **projected** w.r.t. the item corresponding to the rightmost level in the tree (let this item be i).
- This yields an FP-tree of the **conditional transaction database** (database of transactions containing the item i, but with this item removed — it is implicit in the FP-tree and recorded as a common prefix).
- From the projected FP-tree the frequent item sets containing item i can be read directly.
- The **rightmost level** of the original (unprojected) FP-tree is **removed** (the item i is removed from the database — exclude split item).
- The projected FP-tree is processed recursively; the item i is noted as a prefix that is to be added in deeper levels of the recursion.
- Afterward the reduced original FP-tree is further processed by working on the next level leftward.

Projecting an FP-Tree



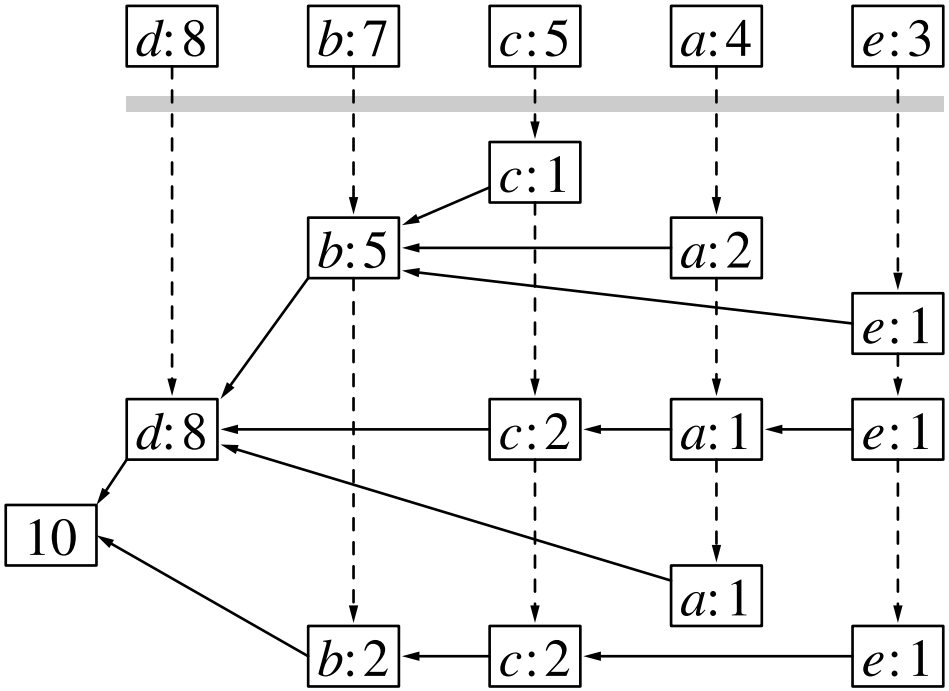


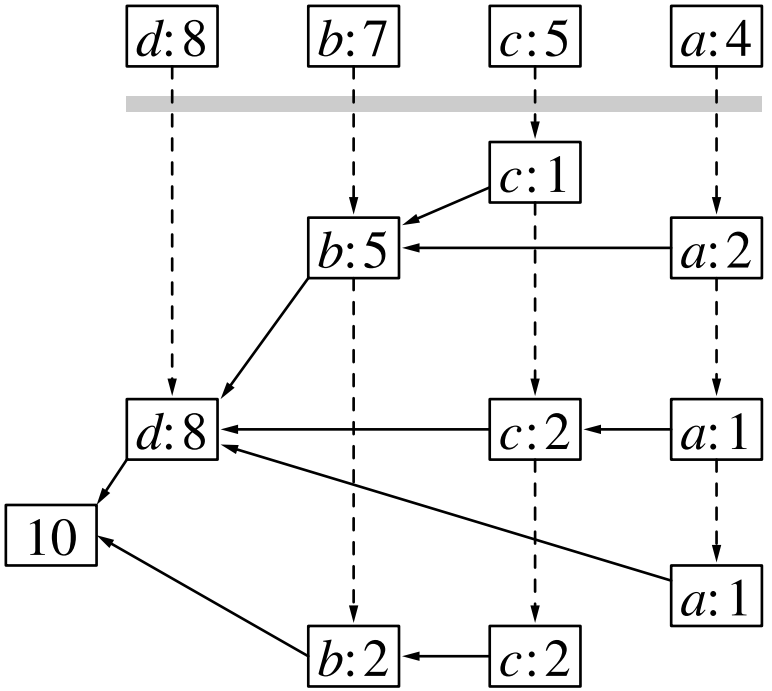
↑
detached projection (prefix e)

← FP-tree with attached projection

- By traversing the node list for the rightmost item, all transactions containing this item can be found.
- The FP-tree of the conditional database for this item is created by copying the nodes on the paths to the root.

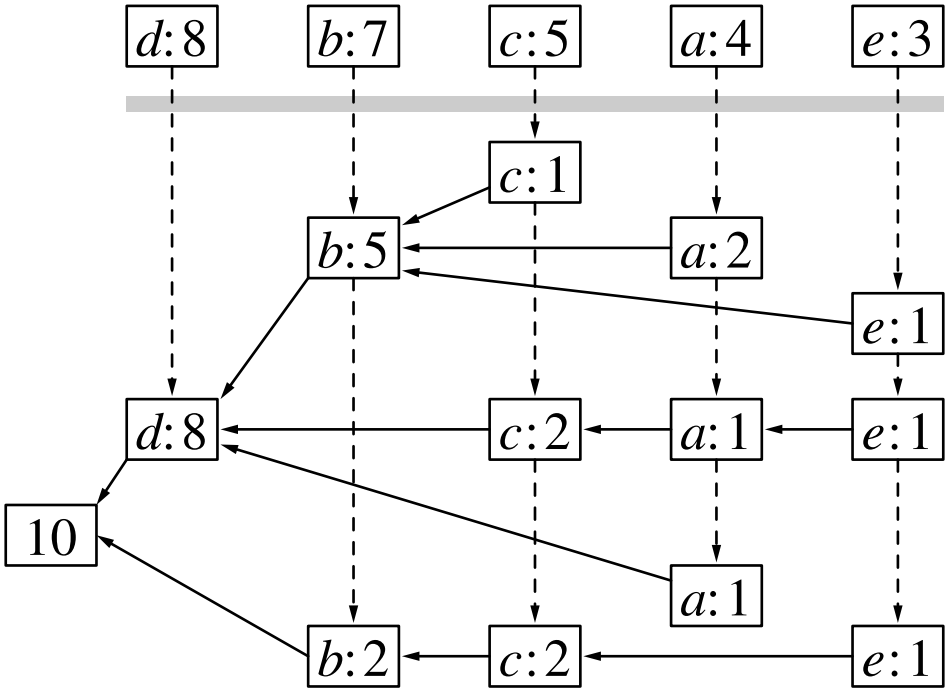
Reducing the Original FP-Tree

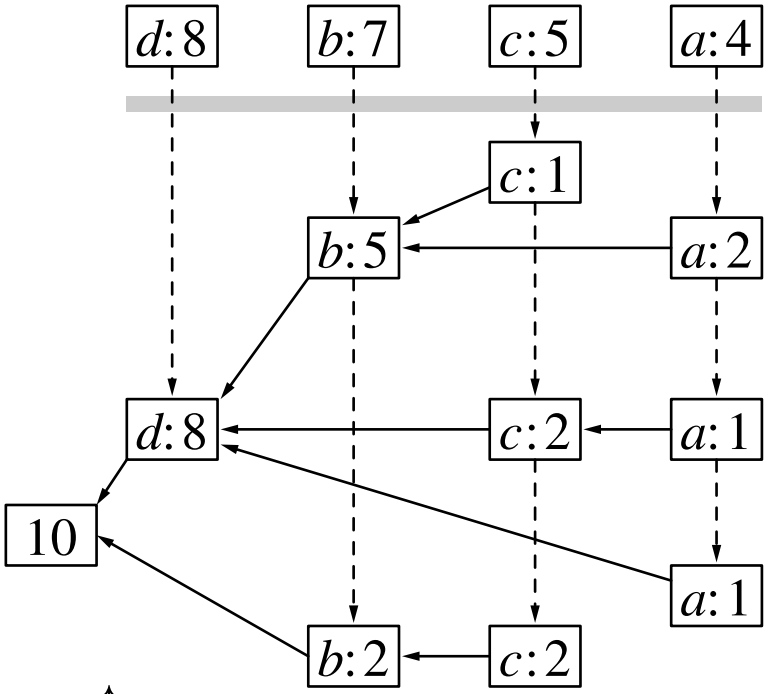




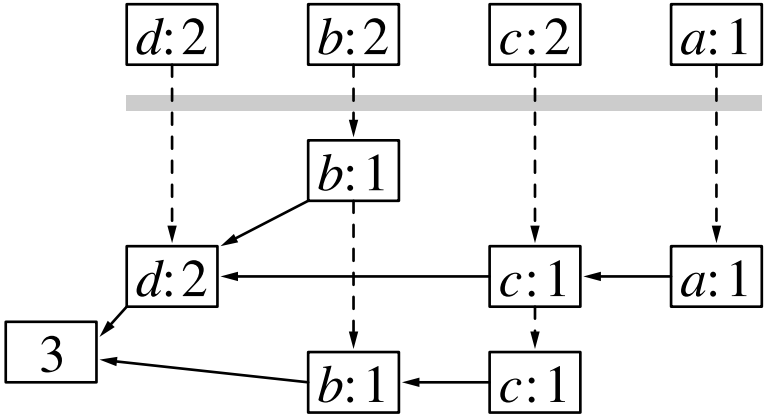
- The original FP-tree is reduced by removing the rightmost level.
- This yields the conditional database for item sets *not* containing the item corresponding to the rightmost level.

FP-growth: Divide-and-Conquer





↑
Conditional database
with item e removed
(second subproblem)



← Conditional database for prefix e
(first subproblem)

Projecting an FP-Tree

- A simpler, but equally efficient projection scheme (compared to node copying) is to extract a path to the root as a (reduced) transaction (into a global buffer) and to insert this transaction into a new, initially empty FP-tree.

- For the insertion into the new FP-tree, there are two approaches:

  - Apart from a parent pointer (which is needed for the path extraction), each node possesses a pointer to its **first child** and **right sibling**. These pointers allow to insert a new transaction top-down.

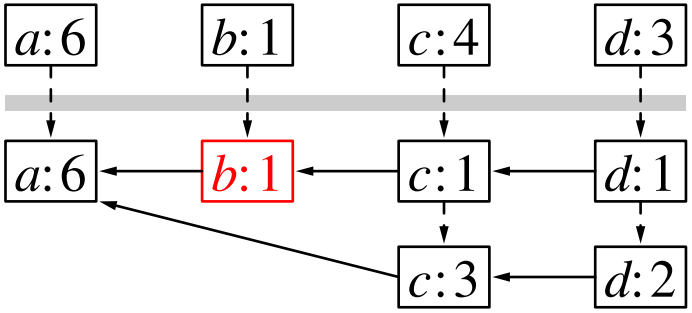
  - If the initial FP-tree has been built from a lexicographically sorted transaction database, the traversal of the item lists yields the (reduced) transactions in lexicographical order.
    This can be exploited to insert a transaction using only the **header table**.

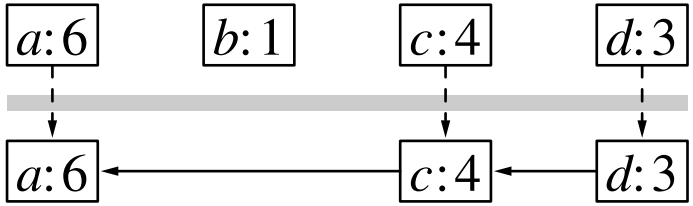
- By processing an FP-tree from **left to right** (or from **top to bottom** w.r.t. the prefix tree), the projection may even reuse the already present nodes and the already processed part of the header table (**top-down FP-growth**). In this way the algorithm can be executed on a fixed amount of memory.

Pruning a Projected FP-Tree

- **Trivial case:** If the item corresponding to the rightmost level is infrequent, the item and the FP-tree level are removed without projection.
- **More interesting case:** An item corresponding to a middle level is infrequent, but an item on a level further to the right is frequent.

Example FP-Tree with an infrequent item on a middle level:





- So-called α-pruning or Bonsai pruning of a (projected) FP-tree.
- Implemented by left-to-right levelwise merging of nodes with same parents.
- Not needed if projection works by extraction, support filtering, and insertion.

FP-growth: Implementation Issues

- **Rebuilding the FP-tree:**

  An FP-tree may be projected by extracting the (reduced) transactions described by the paths to the root and inserting them into a new FP-tree.

  The transaction extraction uses a single global buffer of sufficient size.

  This makes it possible to change the item order, with the following **advantages**:

  - No need for α- or Bonsai pruning, since the items can be reordered so that all conditionally frequent items appear on the left.
  - No need for perfect extension pruning, because the perfect extensions can be moved to the left and are processed at the end with chain optimization.
    (Chain optimization is explained on the next slide.)

  However, there are also **disadvantages**:

  - Either the FP-tree has to be traversed twice or pair frequencies have to be determined to reorder the items according to their conditional frequency (for this the resulting item frequencies need to be known.)

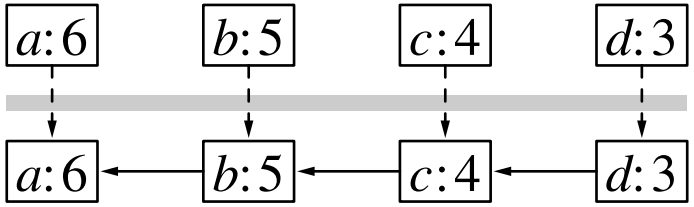
FP-growth: Implementation Issues

- **Chains:**

  If an FP-tree has been reduced to a chain, no projections are computed anymore. Rather all subsets of the set of items in the chain are formed and reported.

- Example of chain processing, exploiting **hypercube decomposition**.

  Suppose we have the following conditional database with prefix P:



  - $P \cup \{d\}$ has support 3 and c, b and d as perfect extensions.
  - $P \cup \{c\}$ has support 4 and b and d as perfect extensions.
  - $P \cup \{b\}$ has support 5 and d as a perfect extension.
  - $P \cup \{a\}$ has support 6.

- Local item order and chain processing implicitly do perfect extension pruning.

FP-growth: Implementation Issues

- The initial FP-tree is built from an array-based main memory representation of the transaction database (eliminates the need for child pointers).
- This has the disadvantage that the memory savings often resulting from an FP-tree representation cannot be fully exploited.
- However, it has the advantage that no child and sibling pointers are needed and the transactions can be inserted in lexicographic order.
- Each FP-tree node has a constant size of 16/24 bytes (2 integers, 2 pointers). Allocating these through the standard memory management is wasteful. (Allocating many small memory objects is highly inefficient.)
- Solution: The nodes are allocated in one large array per FP-tree.
- As a consequence, each FP-tree resides in a single memory block. There is no allocation and deallocation of individual nodes. (This may waste some memory, but is highly efficient.)

FP-growth: Implementation Issues

- An FP-tree can be implemented with only **two integer arrays** [Rasz 2004]:
  - one array contains the transaction counters (support values) and
  - one array contains the parent pointers (as the indices of array elements).

  This reduces the memory requirements to 8 bytes per node.
- Such a memory structure has **advantages**
  due the way in which modern processors access the main memory:

  Linear memory accesses are faster than random accesses.
  - Main memory is organized as a "table" with rows and columns.
  - First the row is addressed and then, after some delay, the column.
  - Accesses to different columns in the same row can skip the row addressing.
- However, there are also **disadvantages**:
  - Programming projection and α- or Bonsai pruning becomes more complex, because less structure is available.
  - Reordering the items is virtually ruled out.

Summary FP-Growth

Basic Processing Scheme

- The transaction database is represented as a frequent pattern tree.
- An FP-tree is projected to obtain a conditional database.
- Recursive processing of the conditional database.

Advantages

- Often the fastest algorithm or among the fastest algorithms.

Disadvantages

- More difficult to implement than other approaches, complex data structure.
- An FP-tree can need more memory than a list or array of transactions.

Experimental Comparison

Experiments: Data Sets

- **Chess**
  A data set listing chess end game positions for king vs. king and rook.
  This data set is part of the UCI machine learning repository.

  75 items, 3196 transactions
  (average) transaction size: 37, density: ≈ 0.5

- **Census** (a.k.a. **Adult**)
  A data set derived from an extract of the US census bureau data of 1994, which was preprocessed by discretizing numeric attributes.
  This data set is part of the UCI machine learning repository.

  135 items, 48842 transactions
  (average) transaction size: 14, density: ≈ 0.1

The **density** of a transaction database is the average fraction of all items occurring per transaction: density = average transaction size / number of items.

Experiments: Data Sets

- **T10I4D100K**
  An artificial data set generated with IBM's data generator.
  The name is formed from the parameters given to the generator
  (for example: 100K = 100000 transactions, T10 = 10 items per transaction).

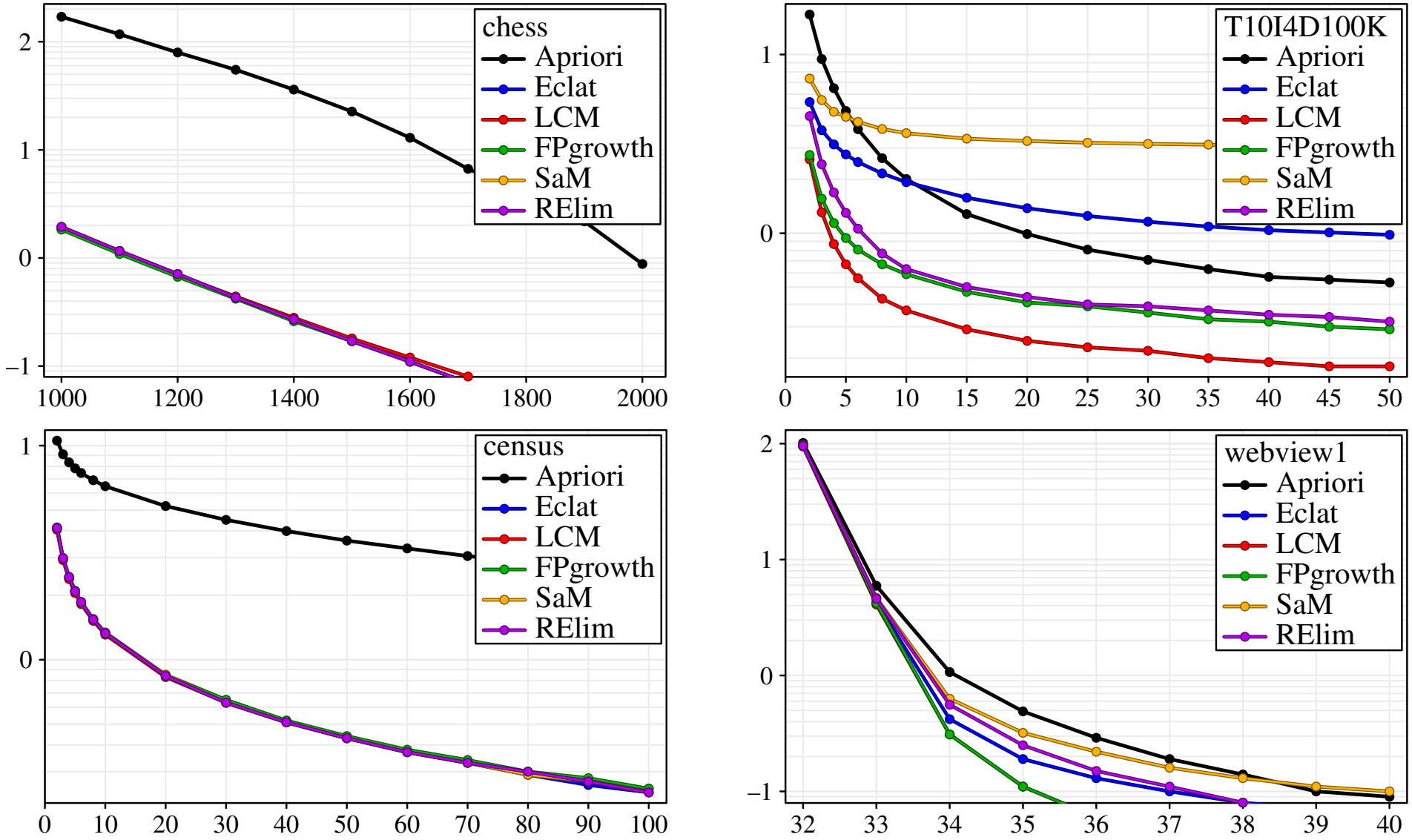
  870 items, 100000 transactions
  average transaction size: ≈ 10.1, density: ≈ 0.012

- **BMS-Webview-1**
  A web click stream from a leg-care company that no longer exists.
  It has been used in the KDD cup 2000 and is a popular benchmark.

  497 items, 59602 transactions
  average transaction size: ≈ 2.5, density: ≈ 0.005

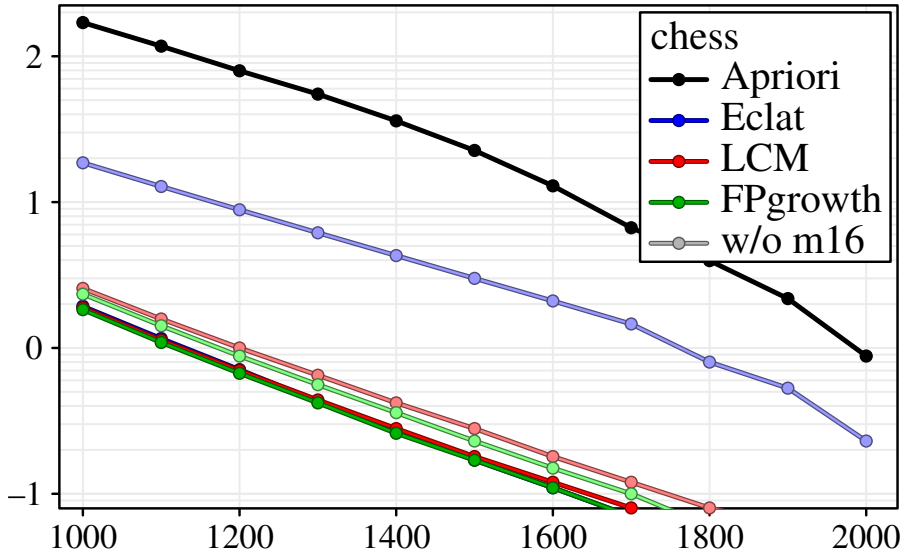
The **density** of a transaction database is the average fraction of all items occurring per transaction: density = average transaction size / number of items

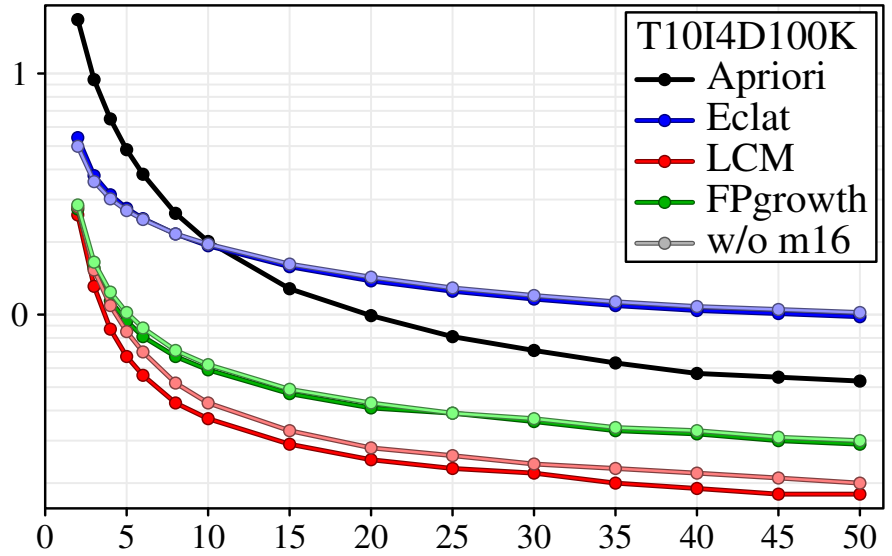
Experiments: Execution Times

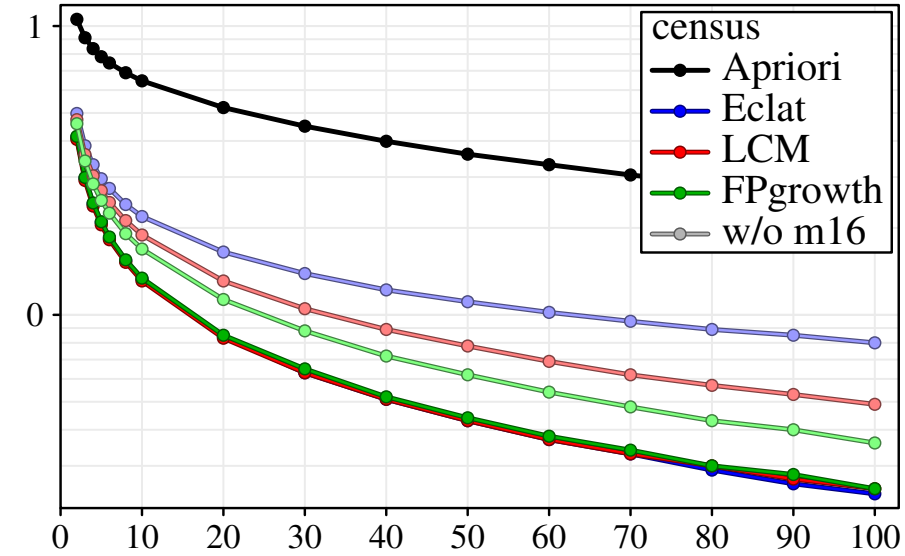


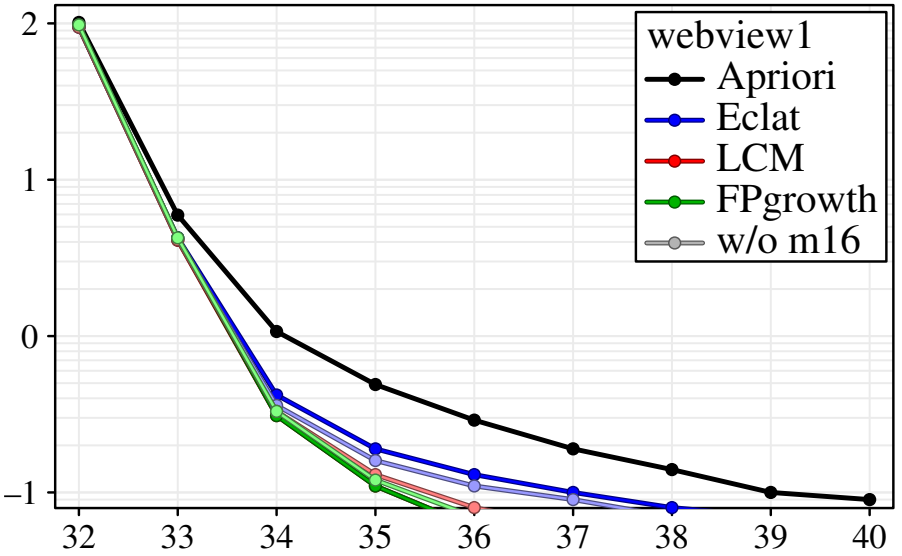
Decimal logarithm of execution time in seconds over absolute minimum support.

Experiments: k-items Machine (here: $k = 16$)







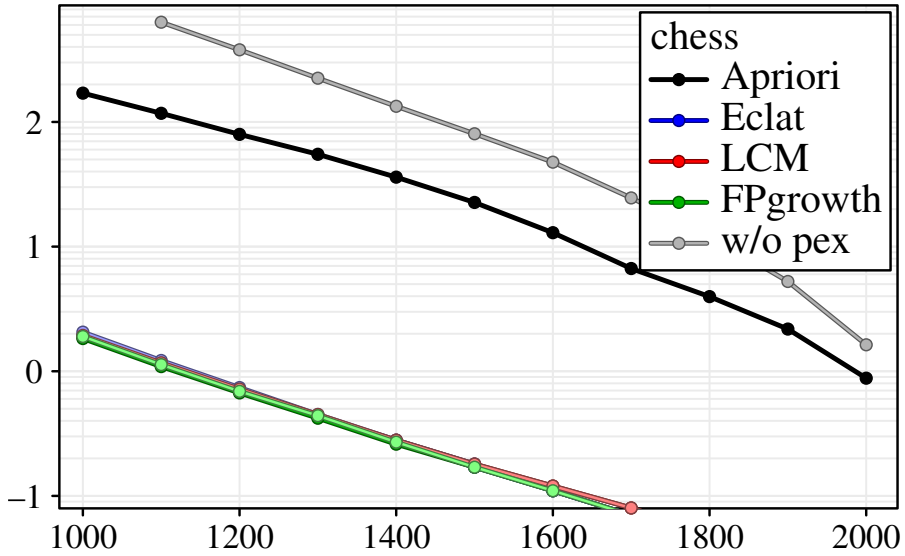


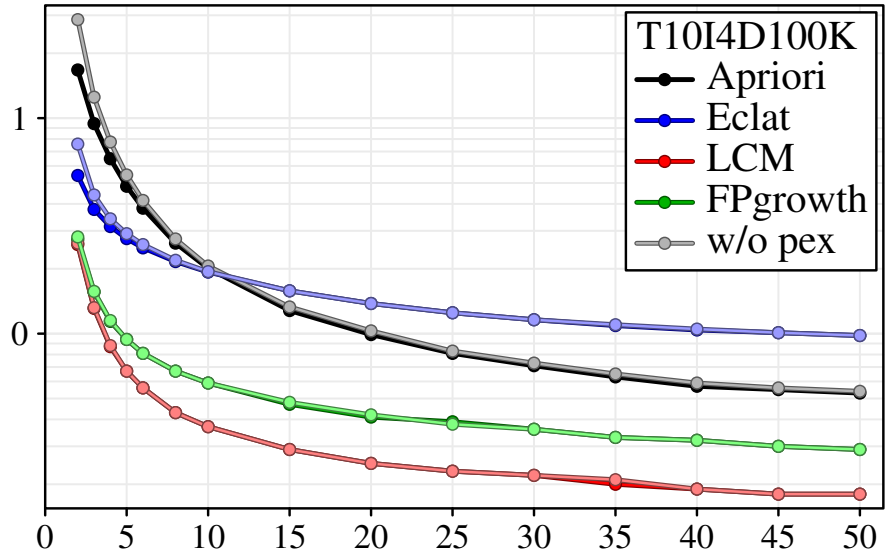
Decimal logarithm of execution time in seconds over absolute minimum support.

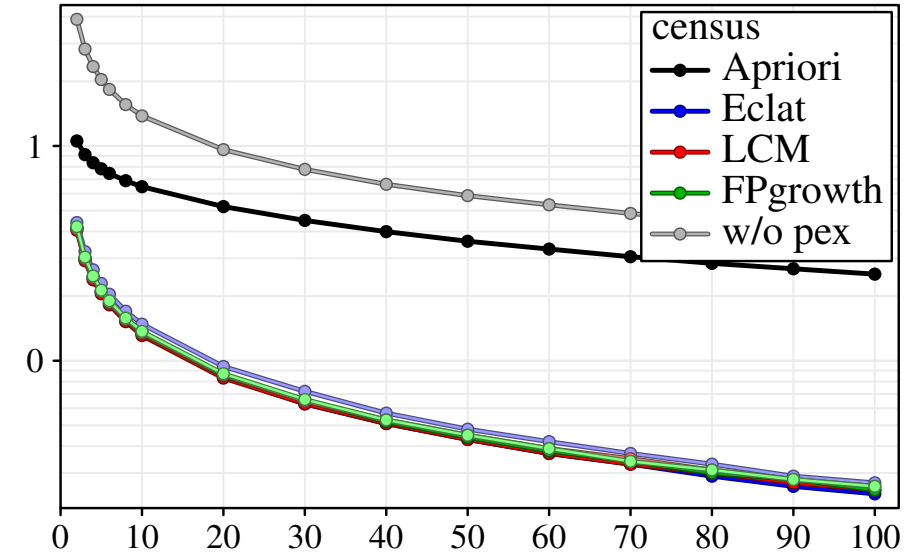
Reminder: Perfect Extensions

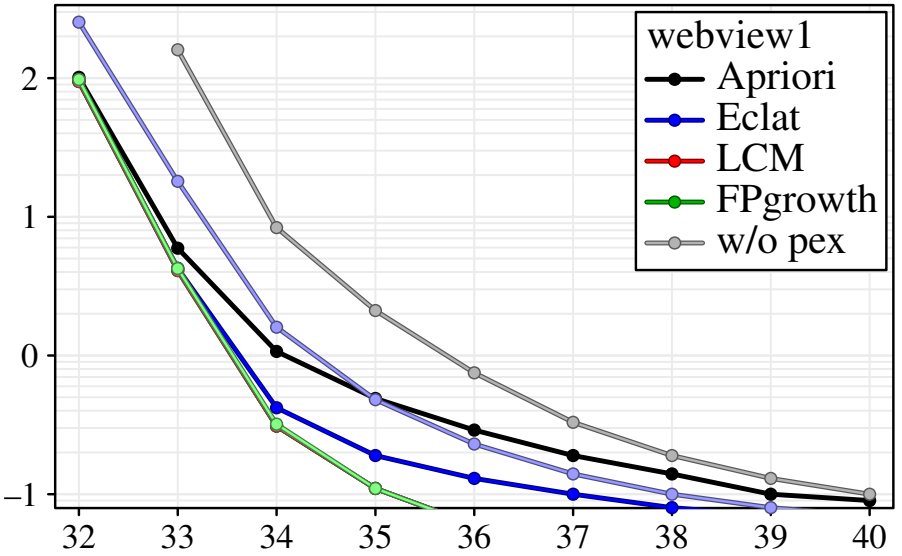
- The search can be improved with so-called **perfect extension pruning**.
- Given an item set I, an item $i \notin I$ is called a **perfect extension** of I, iff I and $I \cup \{i\}$ have the same support (all transactions containing I contain i).
- Perfect extensions have the following properties:
  - If the item i is a perfect extension of an item set I, then i is also a perfect extension of any item set $J \supseteq I$ (as long as $i \notin J$).
  - If I is a frequent item set and X is the set of all perfect extensions of I, then all sets $I \cup J$ with $J \in 2^X$ (where 2^X denotes the power set of X) are also frequent and have the same support as I.
- This can be exploited by collecting perfect extension items in the recursion, in a third element of a subproblem description: $S = (T_*, P, X)$.
- Once identified, perfect extension items are no longer processed in the recursion, but are only used to generate all supersets of the prefix having the same support.

Experiments: Perfect Extension Pruning (with m16)



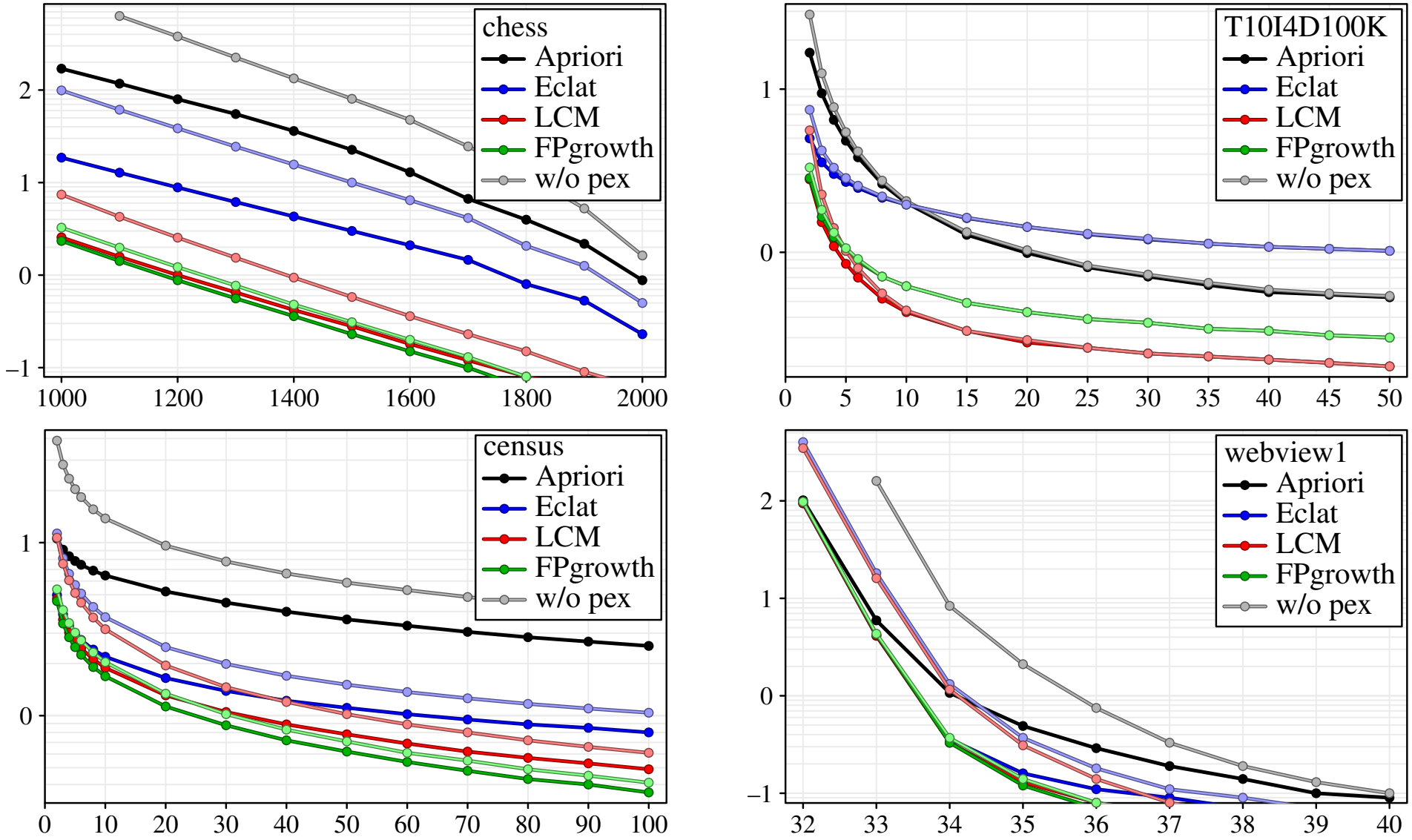






Decimal logarithm of execution time in seconds over absolute minimum support.

Experiments: Perfect Extension Pruning (w/o m16)



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

Reducing the Output:

Closed and Maximal Item Sets

Maximal Item Sets

- Consider the set of **maximal (frequent) item sets**:

$$M_T(s_{\min}) = \{I \subseteq B \mid s_T(I) \geq s_{\min} \wedge \forall J \supset I : s_T(J) < s_{\min}\}.$$

That is: **An item set is maximal if it is frequent, but none of its proper supersets is frequent.**

- Since with this definition we know that

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad I \in M_T(s_{\min}) \ \vee \ \exists J \supset I : s_T(J) \geq s_{\min}$$

it follows (can easily be proven by successively extending the item set I)

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \exists J \in M_T(s_{\min}) : \quad I \subseteq J.$$

That is: **Every frequent item set has a maximal superset.**

- Therefore:

$$\forall s_{\min} : \quad F_T(s_{\min}) = \bigcup_{I \in M_T(s_{\min})} 2^I$$

Mathematical Excursion: Maximal Elements

- Let R be a subset of a partially ordered set $(S, \leq)$.
  An element $x \in R$ is called **maximal** or a **maximal element** of R if
  $$\forall y \in R: \quad y \geq x \quad \Rightarrow \quad y = x.$$
- The notions **minimal** and **minimal element** are defined analogously.
- Maximal elements need not be unique,
  because there may be elements $x, y \in R$ with neither $x \leq y$ nor $y \leq x$.
- Infinite partially ordered sets need not possess a maximal/minimal element.
- Here we consider the set $F_T(s_{\min})$ as a subset of the partially ordered set $(2^B, \subseteq)$:
  The **maximal (frequent) item sets** are the maximal elements of $F_T(s_{\min})$:
  $$M_T(s_{\min}) = \{I \in F_T(s_{\min}) \mid \forall J \in F_T(s_{\min}): \; J \supseteq I \; \Rightarrow \; J = I\}.$$
  That is, no superset of a maximal (frequent) item set is frequent.

Maximal Item Sets: Example

transaction database

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: $\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

- The maximal item sets are:

$$\{b, c\}, \quad \{a, c, d\}, \quad \{a, c, e\}, \quad \{a, d, e\}.$$

- Every frequent item set is a subset of at least one of these sets.

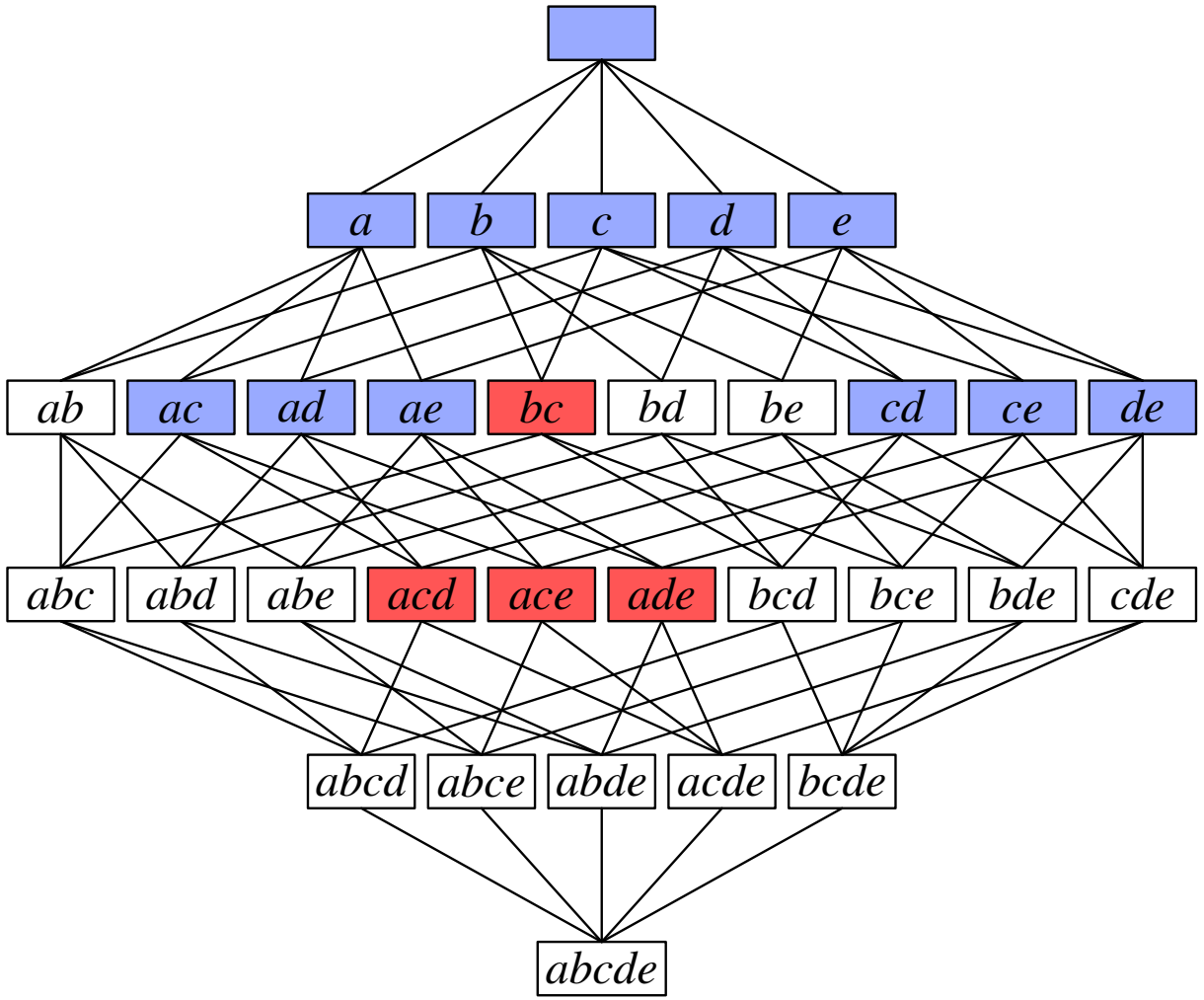
Hasse Diagram and Maximal Item Sets

transaction database

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: $\{a, d, e\}$

Red boxes are maximal item sets, white boxes infrequent item sets.

Hasse diagram with maximal item sets ($s_{\min} = 3$):



Limits of Maximal Item Sets

- The set of maximal item sets captures the set of all frequent item sets, but then we know at most the support of the maximal item sets exactly.

- About the support of a non-maximal frequent item set we only know:

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) - M_T(s_{\min}) : \quad s_T(I) \geq \max_{J \in M_T(s_{\min}), J \supset I} s_T(J).$$

  This relation follows immediately from $\forall I : \forall J \supseteq I : \; s_T(I) \geq s_T(J)$, that is, an item set cannot have a lower support than any of its supersets.

- Note that we have generally

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad s_T(I) \geq \max_{J \in M_T(s_{\min}), J \supseteq I} s_T(J).$$

- **Question:** Can we find a subset of the set of all frequent item sets, which also preserves knowledge of all support values?

Closed Item Sets

- Consider the set of **closed (frequent) item sets**:

$$C_T(s_{\min}) = \{I \subseteq B \mid s_T(I) \geq s_{\min} \wedge \forall J \supset I : s_T(J) < s_T(I)\}.$$

That is: **An item set is closed if it is frequent, but none of its proper supersets has the same support.**

- Since with this definition we know that

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad I \in C_T(s_{\min}) \ \vee \ \exists J \supset I : s_T(J) = s_T(I)$$

it follows (can easily be proven by successively extending the item set I)

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \exists J \in C_T(s_{\min}) : \quad I \subseteq J.$$

That is: **Every frequent item set has a closed superset.**

- Therefore:

$$\forall s_{\min} : \quad F_T(s_{\min}) = \bigcup_{I \in C_T(s_{\min})} 2^I$$

Closed Item Sets

- However, not only has every frequent item set a closed superset, but it has a **closed superset with the same support**:

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \exists J \supseteq I : \quad J \in C_T(s_{\min}) \ \wedge \ s_T(J) = s_T(I).$$

(Proof: see (also) the considerations on the next slide)

- The set of all closed item sets preserves knowledge of all support values:

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad s_T(I) = \max_{J \in C_T(s_{\min}), J \supseteq I} s_T(J).$$

- Note that the weaker statement

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad s_T(I) \geq \max_{J \in C_T(s_{\min}), J \supseteq I} s_T(J)$$

follows immediately from $\forall I : \forall J \supseteq I : s_T(I) \geq s_T(J)$, that is, an item set cannot have a lower support than any of its supersets.

Closed Item Sets

- **Alternative characterization of closed (frequent) item sets:**

$$I \text{ is closed} \quad \Leftrightarrow \quad s_T(I) \geq s_{\min} \quad \wedge \quad I = \bigcap_{k \in K_T(I)} t_k.$$

Reminder: $K_T(I) = \{k \in \{1, \ldots, n\} \mid I \subseteq t_k\}$ is the *cover* of I w.r.t. T.

- This is derived as follows: since $\forall k \in K_T(I) : I \subseteq t_k$, it is obvious that

$$\forall s_{\min} : \forall I \in F_T(s_{\min}) : \quad I \subseteq \bigcap_{k \in K_T(I)} t_k,$$

If $I \subset \bigcap_{k \in K_T(I)} t_k$, it is not closed, since $\bigcap_{k \in K_T(I)} t_k$ has the same support. On the other hand, no superset of $\bigcap_{k \in K_T(I)} t_k$ has the cover $K_T(I)$.

- Note that the above characterization allows us to construct for any item set the (uniquely determined) closed superset that has the same support.

Closed Item Sets: Example

transaction database

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: $\{a, d, e\}$

frequent item sets

0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

- All frequent item sets are closed with the exception of $\{b\}$ and $\{d, e\}$.
- $\{b\}$ is a subset of $\{b, c\}$, both have a support of $3 \hat{=} 30\%$.
  $\{d, e\}$ is a subset of $\{a, d, e\}$, both have a support of $4 \hat{=} 40\%$.

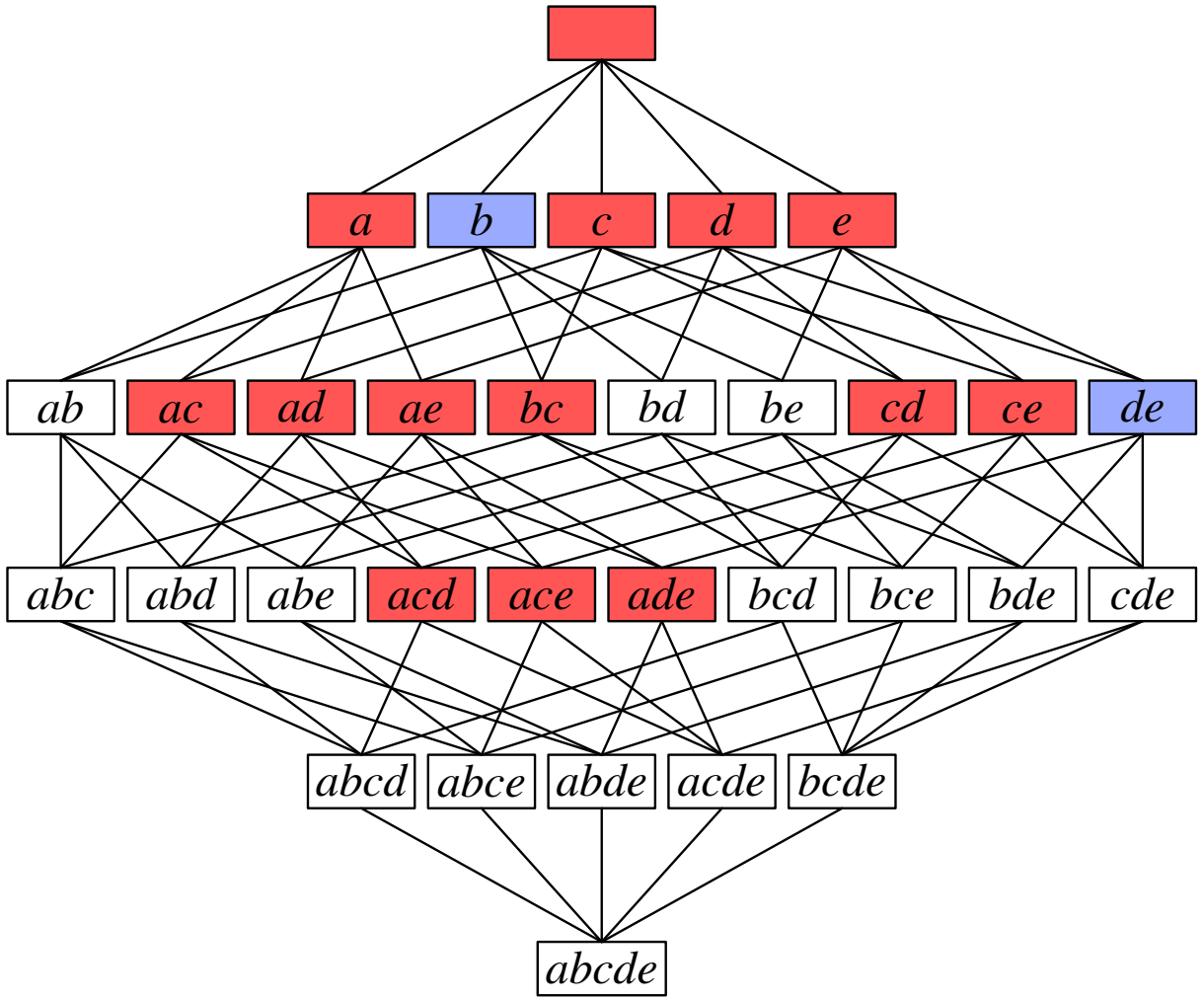
Hasse diagram and Closed Item Sets

transaction database

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: $\{a, d, e\}$

Red boxes are closed item sets, white boxes infrequent item sets.

Hasse diagram with closed item sets ($s_{\min} = 3$):



Reminder: Perfect Extensions

- The search can be improved with so-called **perfect extension pruning**.
- Given an item set I, an item $i \notin I$ is called a **perfect extension** of I, iff I and $I \cup \{i\}$ have the same support (all transactions containing I contain i).
- Perfect extensions have the following properties:
  - If the item i is a perfect extension of an item set I, then i is also a perfect extension of any item set $J \supseteq I$ (as long as $i \notin J$).
  - If I is a frequent item set and X is the set of all perfect extensions of I, then all sets $I \cup J$ with $J \in 2^X$ (where 2^X denotes the power set of X) are also frequent and have the same support as I.
- This can be exploited by collecting perfect extension items in the recursion, in a third element of a subproblem description: $S = (T_*, P, X)$.
- Once identified, perfect extension items are no longer processed in the recursion, but are only used to generate all supersets of the prefix having the same support.

Closed Item Sets and Perfect Extensions

transaction database

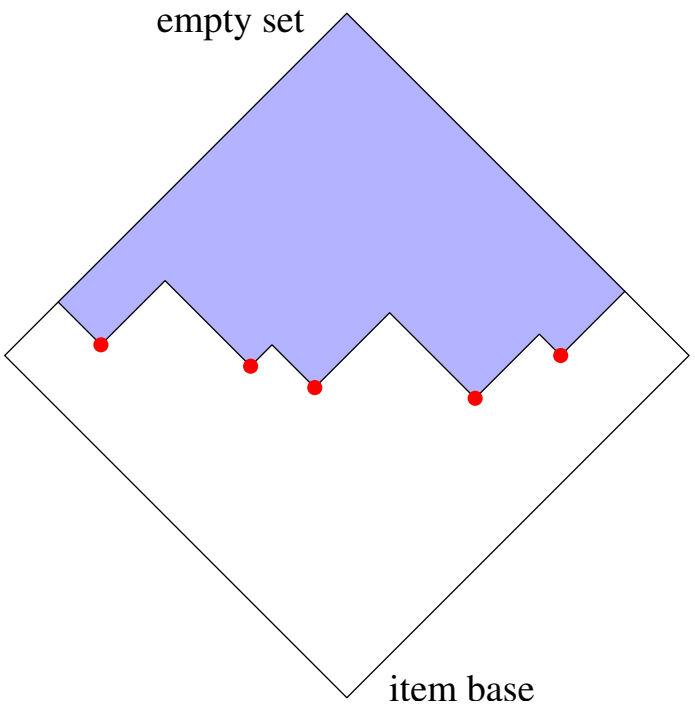
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

frequent item sets

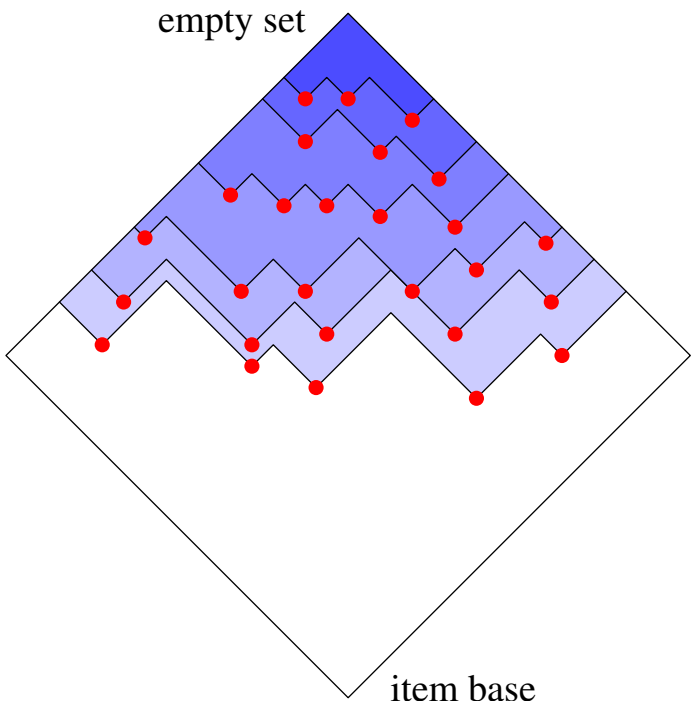
0 items	1 item	2 items	3 items
$\emptyset$: 10	$\{a\}$: 7	$\{a, c\}$: 4	$\{a, c, d\}$: 3
	$\{b\}$: 3	$\{a, d\}$: 5	$\{a, c, e\}$: 3
	$\{c\}$: 7	$\{a, e\}$: 6	$\{a, d, e\}$: 4
	$\{d\}$: 6	$\{b, c\}$: 3	
	$\{e\}$: 7	$\{c, d\}$: 4	
		$\{c, e\}$: 4	
		$\{d, e\}$: 4	

- c is a perfect extension of $\{b\}$ as $\{b\}$ and $\{b, c\}$ both have support 3.
- a is a perfect extension of $\{d, e\}$ as $\{d, e\}$ and $\{a, d, e\}$ both have support 4.
- Non-closed item sets possess at least one perfect extension,
  closed item sets do not possess any perfect extension.

Relation of Maximal and Closed Item Sets



maximal (frequent) item sets



closed (frequent) item sets

- The set of closed item sets is the union of the sets of maximal item sets for all minimum support values at least as large as $s_{\min}$:

$$C_T(s_{\min}) = \bigcup_{s \in \{s_{\min}, s_{\min}+1, \ldots, n-1, n\}} M_T(s)$$

Mathematical Excursion: Closure Operators

- A **closure operator** on a set S is a function $cl : 2^S \to 2^S$ that satisfies the following conditions $\forall X, Y \subseteq S$:
  - $X \subseteq cl(X)$ (cl is *extensive*)
  - $X \subseteq Y \Rightarrow cl(X) \subseteq cl(Y)$ (cl is *increasing or monotone*)
  - $cl(cl(X)) = cl(X)$ (cl is *idempotent*)
- A set $R \subseteq S$ is called **closed** if it is equal to its closure:

$$R \text{ is closed} \quad \Leftrightarrow \quad R = cl(R).$$

- The **closed (frequent) item sets** are induced by the closure operator

$$cl(I) = \bigcap_{k \in K_T(I)} t_k.$$

  restricted to the set of frequent item sets:

$$C_T(s_{\min}) = \{I \in F_T(s_{\min}) \mid I = cl(I)\}$$

Mathematical Excursion: Galois Connections

- Let $(X, \preceq_X)$ and $(Y, \preceq_Y)$ be two partially ordered sets.
- A function pair (f_1, f_2) with $f_1 : X \to Y$ and $f_2 : Y \to X$ is called a **(monotone) Galois connection** iff
  - $\forall A_1, A_2 \in X : \quad A_1 \preceq_X A_2 \quad \Rightarrow \quad f_1(A_1) \preceq_Y f_1(A_2),$
  - $\forall B_1, B_2 \in Y : \quad B_1 \preceq_Y B_2 \quad \Rightarrow \quad f_2(B_1) \preceq_X f_2(B_2),$
  - $\forall A \in X : \forall B \in Y : \quad A \preceq_X f_2(B) \quad \Leftrightarrow \quad B \preceq_Y f_1(A).$
- A function pair (f_1, f_2) with $f_1 : X \to Y$ and $f_2 : Y \to X$ is called an **anti-monotone Galois connection** iff
  - $\forall A_1, A_2 \in X : \quad A_1 \preceq_X A_2 \quad \Rightarrow \quad f_1(A_1) \succeq_Y f_1(A_2),$
  - $\forall B_1, B_2 \in Y : \quad B_1 \preceq_Y B_2 \quad \Rightarrow \quad f_2(B_1) \succeq_X f_2(B_2),$
  - $\forall A \in X : \forall B \in Y : \quad A \preceq_X f_2(B) \quad \Leftrightarrow \quad B \preceq_Y f_1(A).$
- In a monotone Galois connection, both f_1 and f_2 are monotone, in an anti-monotone Galois connection, both f_1 and f_2 are anti-monotone.

Mathematical Excursion: Galois Connections

- Let the two sets X and Y be power sets of some sets U and V, respectively, and let the partial orders be the subset relations on these power sets, that is, let

$$(X, \preceq_X) = (2^U, \subseteq) \qquad \text{and} \qquad (Y, \preceq_Y) = (2^V, \subseteq).$$

- Then the combination $f_1 \circ f_2 : X \to X$ of the functions of a Galois connection is a **closure operator** (as well as the combination $f_2 \circ f_1 : Y \to Y$).

(i) $\forall A \subseteq U : \quad A \subseteq f_2(f_1(A))$ (a closure operator is **extensive**):

- Since (f_1, f_2) is a Galois connection, we know

$$\forall A \subseteq U : \forall B \subseteq V : \quad A \subseteq f_2(B) \Leftrightarrow B \subseteq f_1(A).$$

- Choose $B = f_1(A)$:

$$\forall A \subseteq U : \quad A \subseteq f_2(f_1(A)) \Leftrightarrow \underbrace{f_1(A) \subseteq f_1(A)}_{=\text{true}}.$$

- Choose $A = f_2(B)$:

$$\forall B \subseteq V : \quad \underbrace{f_2(B) \subseteq f_2(B)}_{=\text{true}} \Leftrightarrow B \subseteq f_1(f_2(B)).$$

Mathematical Excursion: Galois Connections

(ii) $\forall A_1, A_2 \subseteq U: \quad A_1 \subseteq A_2 \Rightarrow f_2(f_1(A_1)) \subseteq f_2(f_1(A_2))$

(a closure operator is **increasing** or **monotone**):

- This property follows immediately from the fact that the functions f_1 and f_2 are both (anti-)monotone.

- If f_1 and f_2 are both monotone, we have

$$\begin{aligned} & \forall A_1, A_2 \subseteq U: \quad A_1 \subseteq A_2 \\ \Rightarrow \quad & \forall A_1, A_2 \subseteq U: \quad f_1(A_1) \subseteq f_1(A_2) \\ \Rightarrow \quad & \forall A_1, A_2 \subseteq U: \quad f_2(f_1(A_1)) \subseteq f_2(f_1(A_2)). \end{aligned}$$

- If f_1 and f_2 are both anti-monotone, we have

$$\begin{aligned} & \forall A_1, A_2 \subseteq U: \quad A_1 \subseteq A_2 \\ \Rightarrow \quad & \forall A_1, A_2 \subseteq U: \quad f_1(A_1) \supseteq f_1(A_2) \\ \Rightarrow \quad & \forall A_1, A_2 \subseteq U: \quad f_2(f_1(A_1)) \subseteq f_2(f_1(A_2)). \end{aligned}$$

Mathematical Excursion: Galois Connections

(iii) $\forall A \subseteq U: \; f_2(f_1(f_2(f_1(A)))) = f_2(f_1(A))$ (a closure operator is **idempotent**):

- Since both $f_1 \circ f_2$ and $f_2 \circ f_1$ are extensive (see above), we know

$$\begin{aligned} \forall A \subseteq V: &\quad A \subseteq f_2(f_1(A)) \subseteq f_2(f_1(f_2(f_1(A)))) \\ \forall B \subseteq V: &\quad B \subseteq f_1(f_2(B)) \subseteq f_1(f_2(f_1(f_2(B)))) \end{aligned}$$

- Choosing $B = f_1(A')$ with $A' \subseteq U$, we obtain

$$\forall A' \subseteq U: \quad f_1(A') \subseteq f_1(f_2(f_1(f_2(f_1(A'))))).$$

- Since (f_1, f_2) is a Galois connection, we know

$$\forall A \subseteq U: \forall B \subseteq V: \quad A \subseteq f_2(B) \Leftrightarrow B \subseteq f_1(A).$$

- Choosing $A = f_2(f_1(f_2(f_1(A'))))$ and $B = f_1(A')$, we obtain

$$\begin{aligned} \forall A' \subseteq U: &\quad f_2(f_1(f_2(f_1(A')))) \subseteq f_2(f_1(A')) \\ &\Leftrightarrow \underbrace{f_1(A') \subseteq f_1(f_2(f_1(f_2(f_1(A')))))}_{\text{=true (see above)}}. \end{aligned}$$

Galois Connections in Frequent Item Set Mining

- Consider the partially ordered sets $(2^B, \subseteq)$ and $(2^{\{1,\ldots,n\}}, \subseteq)$.

  Let $f_1 : 2^B \to 2^{\{1,\ldots,n\}}, \quad I \mapsto K_T(I) = \{k \in \{1,\ldots,n\} \mid I \subseteq t_k\}$

  and $f_2 : 2^{\{1,\ldots,n\}} \to 2^B, \quad J \mapsto \bigcap_{j \in J} t_j = \{i \in B \mid \forall j \in J : i \in t_j\}$.

- The function pair (f_1, f_2) is an **anti-monotone Galois connection**:
  - $\forall I_1, I_2 \in 2^B$:

    $I_1 \subseteq I_2 \quad \Rightarrow \quad f_1(I_1) = K_T(I_1) \supseteq K_T(I_2) = f_1(I_2),$
  - $\forall J_1, J_2 \in 2^{\{1,\ldots,n\}}$:

    $J_1 \subseteq J_2 \quad \Rightarrow \quad f_2(J_1) = \bigcap_{k \in J_1} t_k \supseteq \bigcap_{k \in J_2} t_k = f_2(J_2),$
  - $\forall I \in 2^B : \forall J \in 2^{\{1,\ldots,n\}}$:

    $I \subseteq f_2(J) = \bigcap_{j \in J} t_j \quad \Leftrightarrow \quad J \subseteq f_1(I) = K_T(I).$

- As a consequence $f_1 \circ f_2 : 2^B \to 2^B$, $I \mapsto \bigcap_{k \in K_T(I)} t_k$ is a **closure operator**.

Galois Connections in Frequent Item Set Mining

- Likewise $f_2 \circ f_1 : 2^{\{1,\ldots,n\}} \to 2^{\{1,\ldots,n\}}$, $J \mapsto K_T(\bigcap_{j \in J} t_j)$ is also a **closure operator**.

- Furthermore, if we restrict our considerations to the respective sets of closed sets in both domains, that is, to the sets

  $\mathcal{C}_B = \{I \subseteq B \mid I = f_2(f_1(I)) = \bigcap_{k \in K_T(I)} t_k\}$ and

  $\mathcal{C}_T = \{J \subseteq \{1, \ldots, n\} \mid J = f_1(f_2(J)) = K_T(\bigcap_{j \in J} t_j)\}$,

  there exists a **1-to-1 relationship** between these two sets, which is described by the Galois connection:

  $f'_1 = f_1|_{\mathcal{C}_B}$ is a **bijection** with $f'^{-1}_1 = f'_2 = f_2|_{\mathcal{C}_T}$.

  (This follows immediately from the facts that the Galois connection describes closure operators and that a closure operator is idempotent.)

- Therefore finding closed item sets with a given **minimum support** is equivalent to finding closed sets of transaction indices of a given **minimum size**.

Closed Item Sets / Transaction Index Sets

- Finding closed item sets with a given **minimum support** is equivalent to finding closed sets of transaction indices of a given **minimum size**.

Closed in the item set domain 2^B: an item set I is closed if

- adding an item to I reduces the support compared to I;
- adding an item to I loses at least one trans. in $K_T(I) = \{k \in \{1, \ldots, n\} | I \subseteq t_k\}$;
- there is no perfect extension, that is, no (other) item that is contained in all transactions t_k, $k \in K_T(I)$.

Closed in the transaction index set domain $2^{\{1,\ldots,n\}}$:
a transaction index set K is closed if

- adding a transaction index to K reduces the size of the transaction intersection $I_K = \bigcap_{k \in K} t_k$ compared to K;
- adding a transaction index to K loses at least one item in $I_K = \bigcap_{k \in K} t_k$;
- there is no perfect extension, that is, no (other) transaction that contains all items in $I_K = \bigcap_{k \in K} t_k$.

Types of Frequent Item Sets: Summary

- **Frequent Item Set**

  Any frequent item set (support is higher than the minimal support):

  I frequent $\quad\Leftrightarrow\quad s_T(I) \geq s_{\min}$

- **Closed (Frequent) Item Set**

  A frequent item set is called *closed* if no superset has the same support:

  I closed $\quad\Leftrightarrow\quad s_T(I) \geq s_{\min} \quad\wedge\quad \forall J \supset I : s_T(J) < s_T(I)$

- **Maximal (Frequent) Item Set**

  A frequent item set is called *maximal* if no superset is frequent:

  I maximal $\quad\Leftrightarrow\quad s_T(I) \geq s_{\min} \quad\wedge\quad \forall J \supset I : s_T(J) < s_{\min}$

- Obvious relations between these types of item sets:
  - All maximal item sets and all closed item sets are frequent.
  - All maximal item sets are closed.

Types of Frequent Item Sets: Summary

0 items	1 item	2 items	3 items
$\emptyset^{+}$: 10	$\{a\}^{+}$: 7	$\{a, c\}^{+}$: 4	$\{a, c, d\}^{+*}$: 3
	$\{b\}$: 3	$\{a, d\}^{+}$: 5	$\{a, c, e\}^{+*}$: 3
	$\{c\}^{+}$: 7	$\{a, e\}^{+}$: 6	$\{a, d, e\}^{+*}$: 4
	$\{d\}^{+}$: 6	$\{b, c\}^{+*}$: 3	
	$\{e\}^{+}$: 7	$\{c, d\}^{+}$: 4	
		$\{c, e\}^{+}$: 4	
		$\{d, e\}$: 4	

- **Frequent Item Set**
  Any frequent item set (support is higher than the minimal support).

- **Closed (Frequent) Item Set** (marked with $^{+}$)
  A frequent item set is called *closed* if no superset has the same support.

- **Maximal (Frequent) Item Set** (marked with *)
  A frequent item set is called *maximal* if no superset is frequent.

Searching for Closed and Maximal Item Sets

Searching for Closed Frequent Item Sets

- We know that it suffices to find the closed item sets together with their support: from them all frequent item sets and their support can be retrieved.

- The characterization of closed item sets by

$$I \text{ closed} \quad \Leftrightarrow \quad s_T(I) \geq s_{\min} \quad \wedge \quad I = \bigcap_{k \in K_T(I)} t_k$$

  suggests to find them by **forming all possible intersections** of the transactions (of at least $s_{\min}$ transactions).

- However, on standard data sets, approaches using this idea are rarely competitive with other methods.

- Special cases in which they are competitive are domains with few transactions and very many items.
  Examples of such a domains are **gene expression analysis** and the **analysis of document collections**.

Carpenter

[Pan, Cong, Tung, Yang, and Zaki 2003]

Carpenter: Enumerating Transaction Sets

- The **Carpenter** algorithm implements the intersection approach by enumerating sets of transactions (or, equivalently, sets of transaction indices), intersecting them, and removing/pruning possible duplicates (ensuring closed transaction index sets).
- This is done with basically the same **divide-and-conquer scheme** as for the item set enumeration approaches, only that it is applied to transactions (that is, items and transactions exchange their meaning [Rioult *et al.* 2003]).
- The task to enumerate all transaction index sets is split into two sub-tasks:
  - enumerate all transaction index sets that contain the index 1
  - enumerate all transaction index sets that do *not* contain the index 1.
- These sub-tasks are then further divided w.r.t. the transaction index 2: enumerate all transaction index sets containing
  - both indices 1 and 2,
  - index 1, but not index 2,
  - index 2, but not index 1,
  - neither index 1 nor index 2,

  and so on recursively.

Carpenter: Enumerating Transaction Sets

- All subproblems in the recursion can be described by triplets $S = (I, K, k)$.
  - $K \subseteq \{1, \ldots, n\}$ is a set of transaction indices,
  - $I = \bigcap_{k \in K} t_k$ is their intersection, and
  - k is a transaction index, namely the index of the next transaction to consider.
- The initial problem, with which the recursion is started, is $S = (B, \emptyset, 1)$, where B is the item base and no transactions have been intersected yet.
- A subproblem $S_0 = (I_0, K_0, k_0)$ is processed as follows:
  - Let $K_1 = K_0 \cup \{k_0\}$ and form the intersection $I_1 = I_0 \cap t_{k_0}$.
  - If $I_1 = \emptyset$, do nothing (return from recursion).
  - If $|K_1| \geq s_{\min}$, and there is no transaction t_j with $j \in \{1, \ldots, n\} - K_1$ such that $I_1 \subseteq t_j$ (that is, K_1 is closed), report I_1 with support $s_T(I_1) = |K_1|$.
  - Let $k_1 = k_0 + 1$. If $k_1 \leq n$, then form the subproblems $S_1 = (I_1, K_1, k_1)$ and $S_2 = (I_0, K_0, k_1)$ and process them recursively.

Carpenter: List-based Implementation

- **Transaction identifier lists** are used to represent the current item set I (vertical transaction representation, as in the Eclat algorithm).
- The intersection consists in collecting all lists with the next transaction index k.
- Example:

transaction database

t_1	a	b	c	
t_2	a	d	e	
t_3	b	c	d	
t_4	a	b	c	d
t_5	b	c		
t_6	a	b	d	
t_7	d	e		
t_8	c	d	e	

transaction identifier lists

a	b	c	d	e
1	1	1	2	2
2	3	3	3	7
4	4	4	4	8
6	5	5	6	
	6	8	7	
			8	

collection for $K = \{1\}$

a	b	c
2	3	3
4	4	4
6	5	5
	6	8

for $K = \{1, 2\}, \{1, 3\}$

a
4
6

b	c
4	4
5	5
6	8

Carpenter: Table-/Matrix-based Implementation

- Represent the data set by a $n \times |B|$ matrix M as follows [Borgelt *et al.* 2011]

$$m_{ki} = \begin{cases} 0, & \text{if item } i \notin t_k, \\ |\{j \in \{k, \ldots, n\} \mid i \in t_j\}|, & \text{otherwise.} \end{cases}$$

- Example:

transaction database

t_1	a b c
t_2	a d e
t_3	b c d
t_4	a b c d
t_5	b c
t_6	a b d
t_7	d e
t_8	c d e

matrix representation

	a	b	c	d	e
t_1	4	5	5	0	0
t_2	3	0	0	6	3
t_3	0	4	4	5	0
t_4	2	3	3	4	0
t_5	0	2	2	0	0
t_6	1	1	0	3	0
t_7	0	0	0	2	2
t_8	0	0	1	1	1

- The current item set I is simply represented by the contained items.
  An intersection collects all items $i \in I$ with $m_{ki} > \max\{0, s_{\min} - |K| - 1\}$.

Carpenter: Duplicate Removal/Closedness Check

- The intersection of several transaction index sets can yield the same item set.
- The **support** of the item set is the size of the **largest transaction index set** that yields the item set; smaller transaction index sets can be skipped/ignored.

  This is the reason for the check whether there exists a transaction t_j with $j \in \{1, \ldots, n\} - K_1$ such that $I_1 \subseteq t_j$.
- This check is split into the two checks whether there exists such a transaction t_j
  - with $j > k_0$ and
  - with $j \in \{1, \ldots, k_0 - 1\} - K_0$.
- The **first check** is easy, because such transactions are considered in the **recursive processing** which can return whether one exists.
- The problematic **second check** is solved by maintaining a **repository of already found closed frequent item sets**.
- In order to make the look-up in the repository efficient, it is laid out as a **prefix tree** with a flat array top level.

Summary Carpenter

Basic Processing Scheme

- Enumeration of transactions sets (transaction identifier sets).
- Intersection of the transactions in any set yields a closed item set.
- Duplicate removal/closedness check is done with a repository (prefix tree).

Advantages

- Effectively linear in the number of items.
- Very fast for transaction databases with many more items than transactions.

Disadvantages

- Exponential in the number of transactions.
- Very slow for transaction databases with many more transactions than items.

IsTa

Intersecting Transactions
[Mielikäinen 2003] (simple repository, no prefix tree)
[Borgelt, Yang, Nogales-Cadenas, Carmona-Saez, and Pascual-Montano 2011]

Ista: Cumulative Transaction Intersections

- Alternative approach: maintain a repository of all closed item sets, which is updated by intersecting it with the next transaction [Mielikainen 2003].
- To justify this approach formally, we consider the set of all closed frequent item sets for $s_{\min} = 1$, that is, the set

$$\mathcal{C}_T(1) = \{I \subseteq B \mid \exists S \subseteq T : S \neq \emptyset \wedge I = \textstyle\bigcap_{t \in S} t\}.$$

- The set $\mathcal{C}_T(1)$ satisfies the following simple recursive relation:

$$\begin{aligned} \mathcal{C}_\emptyset(1) &= \emptyset, \\ \mathcal{C}_{T \cup \{t\}}(1) &= \mathcal{C}_T(1) \cup \{t\} \cup \{I \mid \exists s \in \mathcal{C}_T(1) : I = s \cap t\}. \end{aligned}$$

- Therefore we can start the procedure with an empty set of closed item sets and then process the transactions one by one.
- In each step update the set of closed item sets by adding the new transaction t and the additional closed item sets that result from intersecting it with $\mathcal{C}_T(1)$.
- In addition, the support of already known closed item sets may have to be updated.

Ista: Cumulative Transaction Intersections

- The core implementation problem is to find a **data structure** for storing the closed item sets that allows to quickly compute the intersections with a new transaction and to merge the result with the already stored closed item sets.
- For this we rely on a **prefix tree**, each node of which represents an item set.
- The algorithm works on the prefix tree as follows:
  - At the beginning an empty tree is created (dummy root node);
    then the transactions are processed one by one.
  - Each new transaction is first simply added to the prefix tree.
    Any new nodes created in this step are initialized with a support of zero.
  - In the next step we compute the intersections of the new transaction
    with all item sets represented by the current prefix tree.
  - A recursive procedure traverses the prefix tree selectively (depth-first) and matches the items in the tree nodes with the items of the transaction.
- **Intersecting with and inserting into the tree can be combined.**

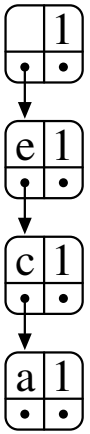
Ista: Cumulative Transaction Intersections

transaction database

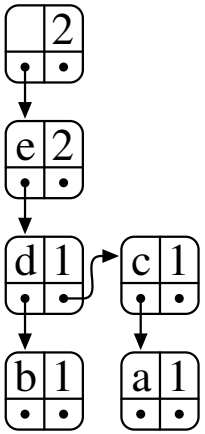
t_1	$e\ c\ a$
t_2	$e\ d\ b$
t_3	$d\ c\ b\ a$

0:

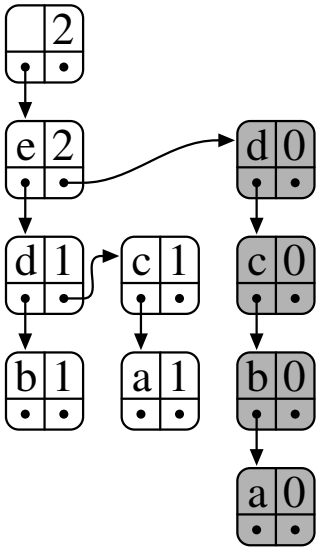
1:



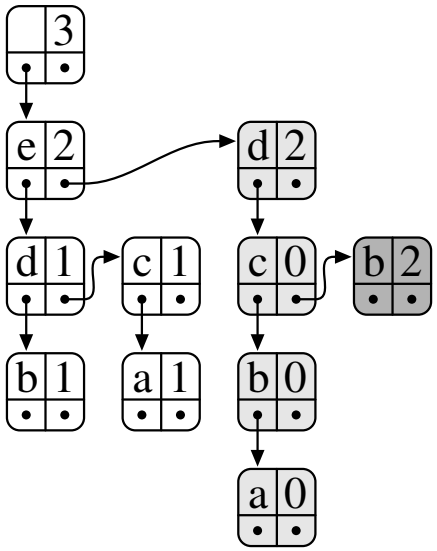
2:



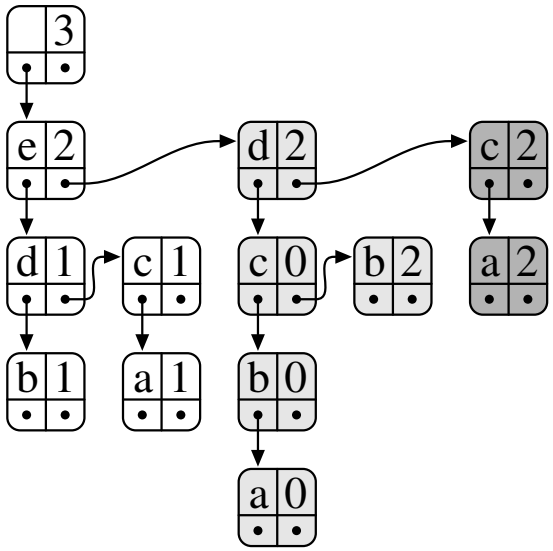
3.1:



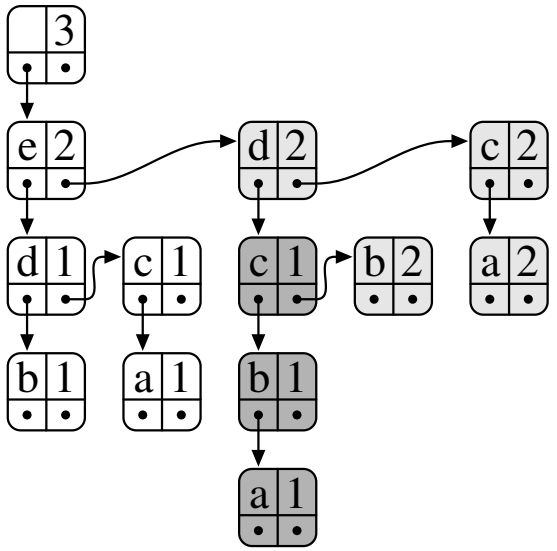
3.2:



3.3:



3.4:



Ista: Data Structure

```
typedef struct _node {          /* a prefix tree node */
  int step;                     /* most recent update step */
  int item;                     /* assoc. item (last in set) */
  int supp;                     /* support of item set */
  struct _node *sibling;        /* successor in sibling list */
  struct _node *children;       /* list of child nodes */
} NODE;
```

- Standard first child / right sibling node structure.
  - Fixed size of each node allows for optimized allocation.
  - Flexible structure that can easily be extended
- The "step" field indicates whether the support field was already updated.
- The step field is an "incremental marker", so that it need not be cleared in a separate traversal of the prefix tree.

Ista: Pseudo-Code

```
void isect (NODE* node, NODE **ins)
{                                       /* intersect with transaction */
  int  i;                               /* buffer for current item */
  NODE *d;                              /* to allocate new nodes */
  while (node) {                        /* traverse the sibling list */
    i = node->item;                     /* get the current item */
    if (trans[i]) {                     /* if item is in intersection */
      while ((d = *ins) && (d→item > i))
        ins = &d->sibling;              /* find the insertion position */
      if (d                             /* if an intersection node with */
      && (d->item == i)) {              /* the item already exists */
        if (d->step >= step) d->supp--;
        if (d->supp < node->supp)
          d->supp = node->supp;
        d->supp++;                      /* update intersection support */
        d->step = step; }               /* and set current update step */
```

Ista: Pseudo-Code

```
      else {                          /* if there is no corresp. node */
        d = malloc(sizeof(NODE));
        d->step = step;               /* create a new node and */
        d->item = i;                  /* set item and support */
        d->supp = node->supp+1;
        d->sibling = *ins; *ins = d;
        d->children = NULL;
      }                               /* insert node into the tree */
      if (i <= imin) return;          /* if beyond last item, abort */
      isect(node->children, &d->children); }
    else {                            /* if item is not in intersection */
      if (i <= imin) return;          /* if beyond last item, abort */
      isect(node->children, ins);
    }                                 /* intersect with subtree */
    node = node->sibling;             /* go to the next sibling */
  } /* end of while (node) */
} /* isect() */
```

Ista: Keeping the Repository Small

- In practice we will not work with a minimum support $s_{\min} = 1$.
- Removing intersections early, because they do not reach the minimum support is difficult: in principle, enough of the transactions to be processed in the future could contain the item set under consideration.
- Improved processing with item occurrence counters:
  - In an initial pass the frequency of the individual items is determined.
  - The obtained counters are updated with each processed transaction. They always represent the item occurrences in the unprocessed transactions.
- Based on these counters, we can apply the following pruning scheme:
  - Suppose that after having processed k of a total of n transactions the support of a closed item set I is $s_{T_k}(I) = x$.
  - Let y be the minimum of the counter values for the items contained in I.
  - If $x + y < s_{\min}$, then I can be discarded, because it cannot reach $s_{\min}$.

Ista: Keeping the Repository Small

- One has to be careful, though, because I may be needed in order to form subsets, namely those that result from intersections of it with new transactions.

  These subsets may still be frequent, even though I is not.

- As a consequence, an item set I is not simply removed, but those **items are selectively removed** from it that do not occur frequently enough in the remaining transactions.

- Although in this way non-closed item sets may be constructed, no problems for the final output are created:

  - either the reduced item set also occurs as the intersection of enough transactions and thus is closed,

  - or it will not reach the minimum support threshold and then it will not be reported.

Summary Ista

Basic Processing Scheme

- Cumulative intersection of transactions (incremental/on-line/stream mining).
- Combined intersection and repository extensions (one traversal).
- Additional pruning is possible for batch processing.

Advantages

- Effectively linear in the number of items.
- Very fast for transaction databases with many more items than transactions.

Disadvantages

- Exponential in the number of transactions.
- Very slow for transaction databases with many more transactions than items

Experimental Comparison: Data Sets

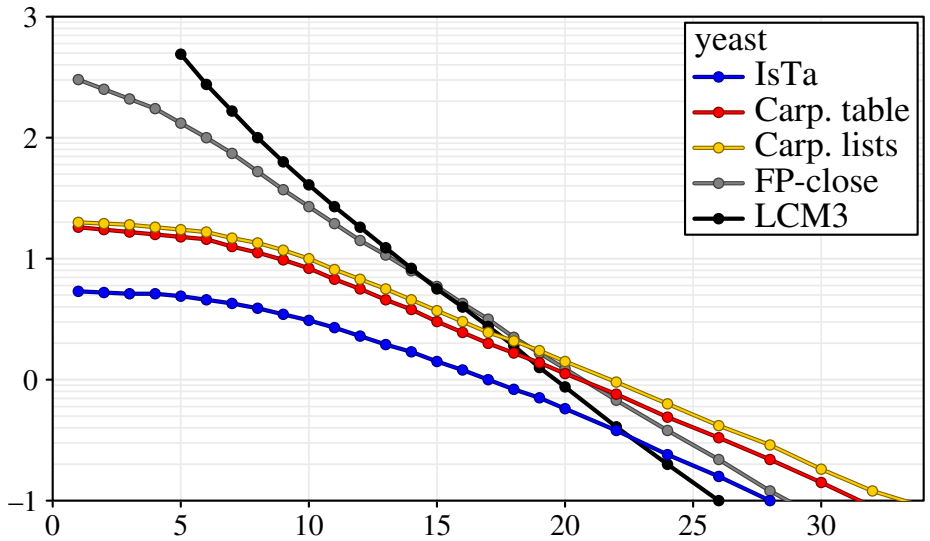
- **Yeast**
  Gene expression data for baker's yeast *(saccharomyces cerevisiae)*.
  300 transactions (experimental conditions), about 10,000 items (genes)

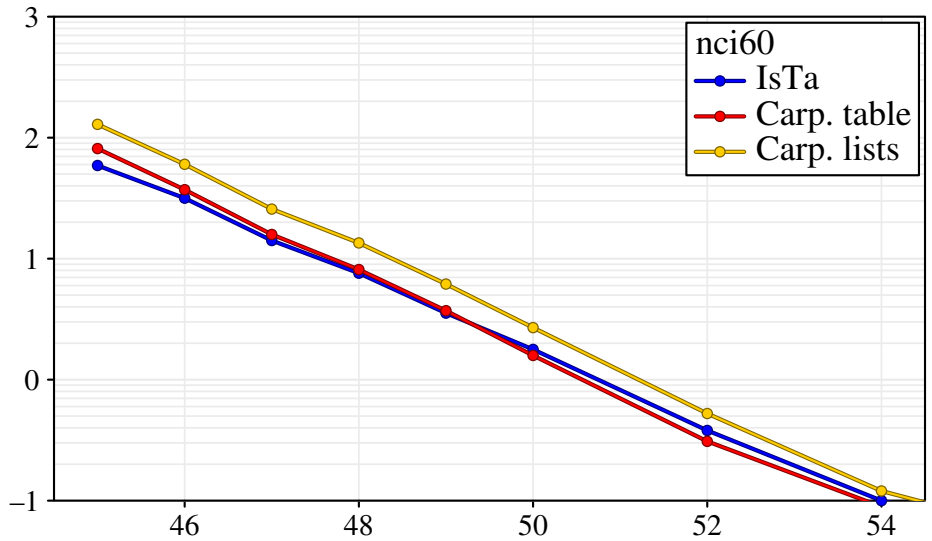
- **NCI 60**
  Gene expression data from the Stanford NCI60 Cancer Microarray Project.
  64 transactions (experimental conditions), about 10,000 items (genes)

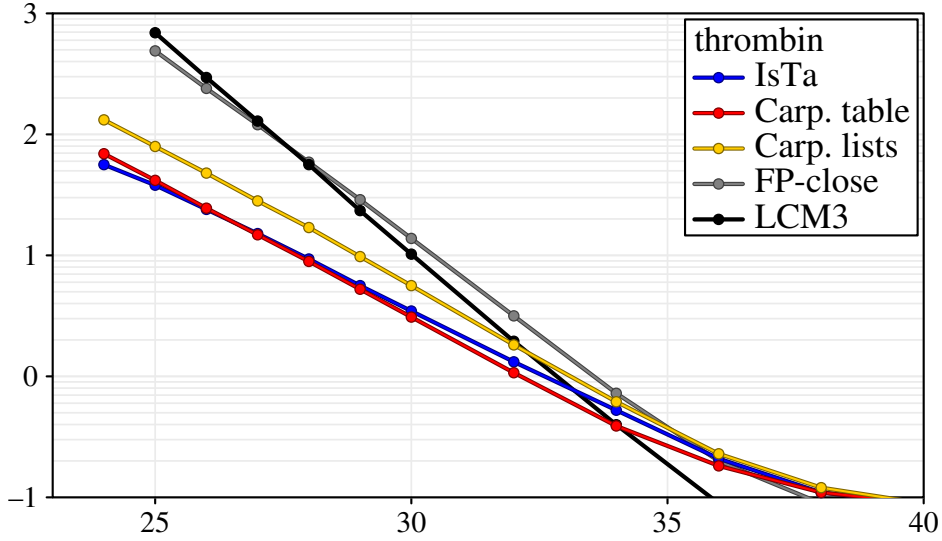
- **Thrombin**
  Chemical fingerprints of compounds (not) binding to Thrombin
  (a.k.a. fibrinogenase, (activated) blood-coagulation factor II etc.).
  1909 transactions (compounds), 139,351 items (binary features)

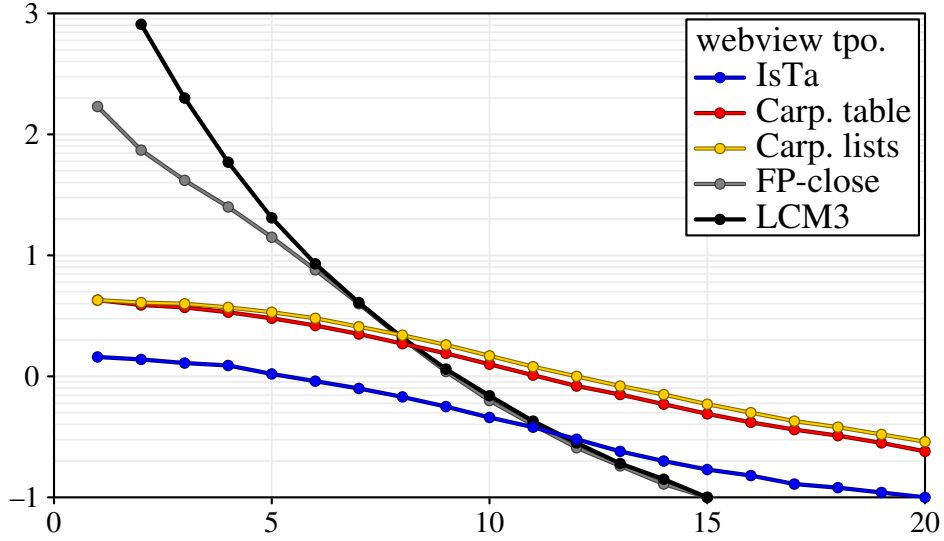
- **BMS-Webview-1 transposed**
  A web click stream from a leg-care company that no longer exists.
  497 transactions (originally items), 59602 items (originally transactions).

Experimental Comparison: Execution Times









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

Searching for Closed and Maximal Item Sets with Item Set Enumeration

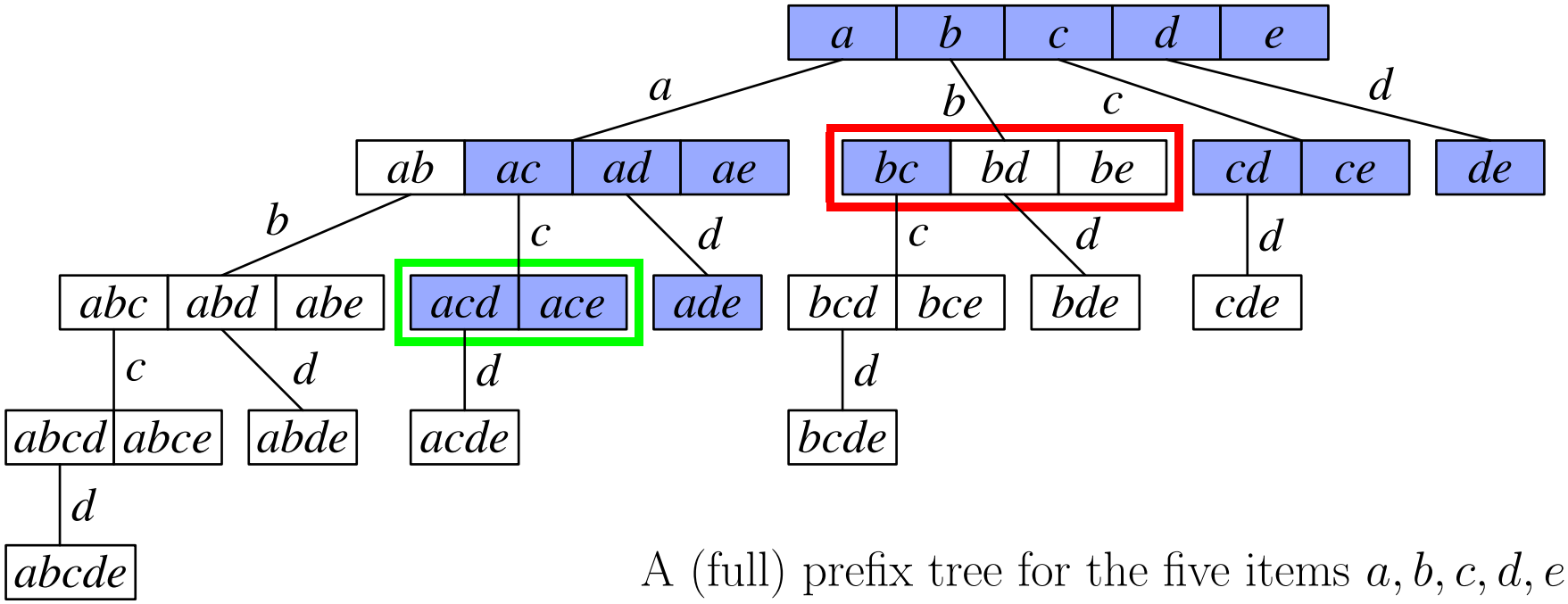
Filtering Frequent Item Sets

- If only closed item sets or only maximal item sets are to be found with item set enumeration approaches, the found frequent item sets have to be filtered.

- Some useful notions for filtering and pruning:
  - The **head** $H \subseteq B$ of a search tree node is the set of items on the path leading to it. It is the prefix of the conditional database for this node.
  - The **tail** $L \subseteq B$ of a search tree node is the set of items that are frequent in its conditional database. They are the possible extensions of H.
  - Note that $\forall h \in H : \forall l \in L : \quad h < l$
    (provided the split items are chosen according to a fixed order).
  - $E = \{i \in B - H \mid \exists h \in H : h > i\}$ is the set of **excluded items**.
    These items are not considered anymore in the corresponding subtree.

- Note that the items in the tail and their support in the conditional database are known, at least after the search returns from the recursive processing.

Head, Tail and Excluded Items



A (full) prefix tree for the five items a, b, c, d, e.

- The blue boxes are the frequent item sets.
- For the encircled search tree nodes we have:

  red: head $H = \{b\}$, tail $L = \{c\}$, excluded items $E = \{a\}$

  green: head $H = \{a, c\}$, tail $L = \{d, e\}$, excluded items $E = \{b\}$

Closed and Maximal Item Sets

- When filtering frequent item sets for closed and maximal item sets the following conditions are easy and efficient to check:
  - If the tail of a search tree node is *not* empty, its head is *not* a maximal item set.
  - If an item in the tail of a search tree node has the same support as the head, the head is *not* a closed item set.
- However, the inverse implications need not hold:
  - If the tail of a search tree node is empty, its head is not necessarily a maximal item set.
  - If no item in the tail of a search tree node has the same support as the head, the head is not necessarily a closed item set.
- The problem are the **excluded items**, which can still render the head non-closed or non-maximal.

Closed and Maximal Item Sets

Check the Defining Condition Directly:

- **Closed Item Sets**:

  Check whether $\exists i \in E: \quad K_T(H) \subseteq K_T(i)$

  or check whether $\bigcap_{k \in K_T(H)} (t_k - H) \neq \emptyset.$

  If either is the case, H is not closed, otherwise it is.

  Note that the intersection can be computed transaction by transaction.
  It can be concluded that H is closed as soon as the intersection becomes empty.

- **Maximal Item Sets:**

  Check whether $\exists i \in E: \quad s_T(H \cup \{i\}) \geq s_{\min}.$

  If this is the case, H is not maximal, otherwise it is.

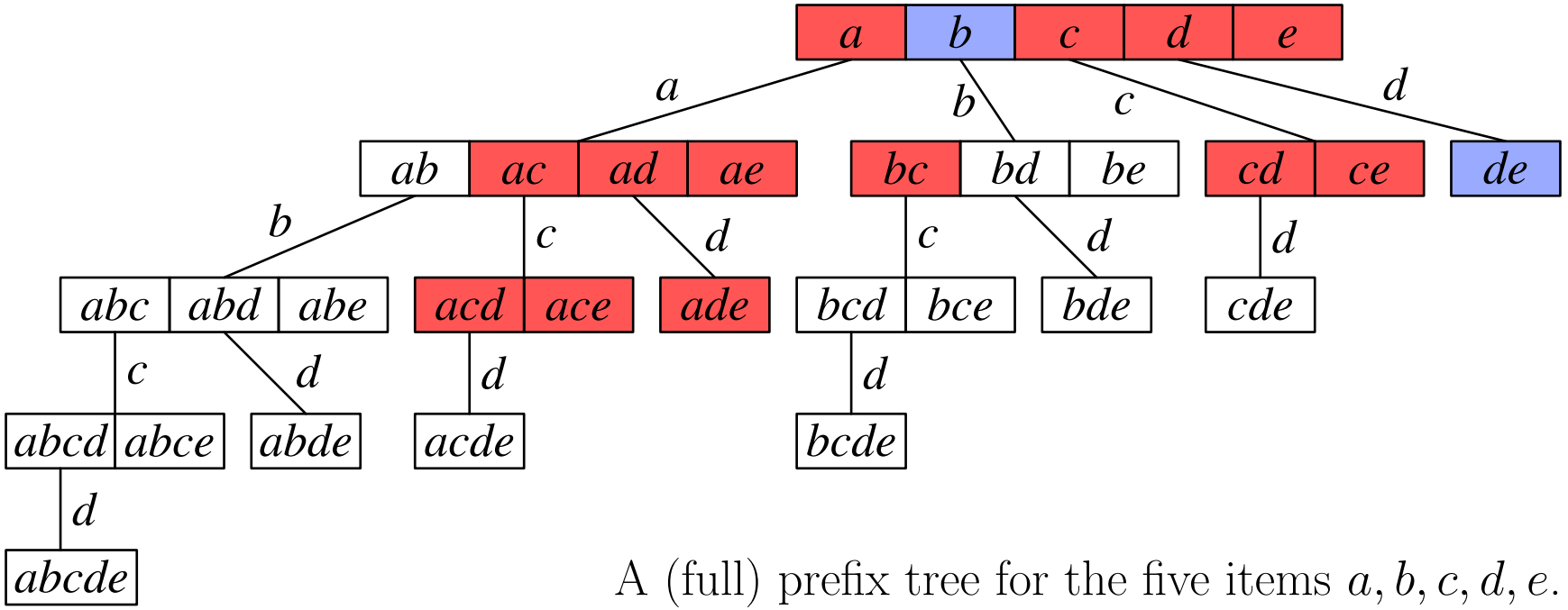
Closed and Maximal Item Sets

- Checking the defining condition directly is trivial for the tail items, as their support values are available from the conditional transaction databases.
- As a consequence, all item set enumeration approaches for closed and maximal item sets check the defining condition for the tail items.
- However, checking the defining condition can be difficult for the excluded items, since additional data (beyond the conditional transaction database) is needed to determine their occurrences in the transactions or their support values.
- It can depend on the database structure used whether a check of the defining condition is efficient for the excluded items or not.
- As a consequence, some item set enumeration algorithms do not check the defining condition for the excluded items, but rely on a repository of already found closed or maximal item sets.
- With such a repository it can be checked in an indirect way whether an item set is closed or maximal.

Checking the Excluded Items: Repository

- Each found maximal or closed item set is stored in a repository.
  (Preferred data structure for the repository: prefix tree)
- It is checked whether a superset of the head H with the same support has already been found. If yes, the head H is neither closed nor maximal.
- Even more: the head H need not be processed recursively, because the recursion cannot yield any closed or maximal item sets. Therefore the current subtree of the search tree can be pruned.
- Note that with a repository the depth-first search has to proceed from left to right.
  - We need the repository to check for possibly existing closed or maximal supersets that contain one or more excluded item(s).
  - Item sets containing excluded items are considered only in search tree branches to the left of the considered node.
  - Therefore these branches must already have been processed in order to ensure that possible supersets have already been recorded.

Checking the Excluded Items: Repository



A (full) prefix tree for the five items a, b, c, d, e.

- Suppose the prefix tree would be traversed from right to left.
- For none of the frequent item sets $\{d, e\}$, $\{c, d\}$ and $\{c, e\}$ it could be determined with the help of a repository that they are not maximal, because the maximal item sets $\{a, c, d\}$, $\{a, c, e\}$, $\{a, d, e\}$ have not been processed then.

Checking the Excluded Items: Repository

- If a superset of the current head H with the same support has already been found, the head H need not be processed, because it cannot yield any maximal or closed item sets.

- The reason is that a found proper superset $I \supset H$ with $s_T(I) = s_T(H)$ contains at least one item $i \in I - H$ that is a perfect extension of H.

- The item i is an excluded item, that is, $i \notin L$ (item i is not in the tail). (If i were in L, the set I would not be in the repository already.)

- If the item i is a perfect extension of the head H, it is a perfect extension of all supersets $J \supseteq H$ with $i \notin J$.

- All item sets explored from the search tree node with head H and tail L are subsets of $H \cup L$ (because only the items in L are conditionally frequent).

- Consequently, the item i is a perfect extension of all item sets explored from the search tree node with head H and tail L, and therefore none of them can be closed.

Checking the Excluded Items: Repository

- It is usually advantageous to use not just a single, global repository, but to create conditional repositories for each recursive call, which contain only the found closed item sets that contain H.

- With conditional repositories the check for a known superset reduces to the check whether the conditional repository contains an item set with the next split item and the same support as the current head.

  (Note that the check is executed before going into recursion, that is, before constructing the extended head of a child node. If the check finds a superset, the child node is pruned.)

- The conditional repositories are obtained by basically the same operation as the conditional transaction databases (projecting/conditioning on the split item).

- A popular structure for the repository is an FP-tree, because it allows for simple and efficient projection/conditioning.

  However, a simple prefix tree that is projected top-down may also be used.

Closed and Maximal Item Sets: Pruning

- If only closed item sets or only maximal item sets are to be found, additional pruning of the search tree becomes possible.

- **Perfect Extension Pruning / Parent Equivalence Pruning (PEP)**

  - Given an item set I, an item $i \notin I$ is called a **perfect extension** of I, iff the item sets I and $I \cup \{i\}$ have the same support: $s_T(I) = s_T(I \cup \{i\})$ (that is, if all transactions containing I also contain the item i).

    Then we know: $\quad \forall J \supseteq I: \quad s_T(J \cup \{i\}) = s_T(J)$.

  - As a consequence, no superset $J \supseteq I$ with $i \notin J$ can be closed. Hence i can be added directly to the prefix of the conditional database.

- Let $X_T(I) = \{i \mid i \notin I \wedge s_T(I \cup \{i\}) = s_T(I)\}$ be the set of all perfect extension items. Then the whole set $X_T(I)$ can be added to the prefix.

- Perfect extension / parent equivalence pruning can be applied for both closed and maximal item sets, since all maximal item sets are closed.

Head Union Tail Pruning

- If only maximal item sets are to be found,
  even more additional pruning of the search tree becomes possible.

- **General Idea:** All frequent item sets in the subtree rooted at a node
  with head H and tail L are subsets of $H \cup L$.

- **Maximal Item Set Contains Head $\cup$ Tail Pruning (MFIHUT)**
  - If we find out that $H \cup L$ is a subset of an already found
    maximal item set, the whole subtree can be pruned.
  - This pruning method requires a left to right traversal of the prefix tree.

- **Frequent Head $\cup$ Tail Pruning (FHUT)**
  - If $H \cup L$ is not a subset of an already found maximal item set
    and by some clever means we discover that $H \cup L$ is frequent,
    $H \cup L$ can immediately be recorded as a maximal item set.

Alternative Description of Closed Item Set Mining

- In order to avoid redundant search in the partially ordered set $(2^B, \subseteq)$, we assigned a unique parent item set to each item set (except the empty set).
- Analogously, we may structure the set of closed item sets by assigning **unique closed parent item sets**. [Uno *et al.* 2003]
- Let $\leq$ be an item order and let I be a closed item set with $I \neq \bigcap_{1 \leq k \leq n} t_k$. Let $i_* \in I$ be the (uniquely determined) item satisfying

$$s_T(\{i \in I \mid i < i_*\}) > s_T(I) \qquad \text{and} \qquad s_T(\{i \in I \mid i \leq i_*\}) = s_T(I).$$

  Intuitively, the item i_* is the greatest item in I that is not a perfect extension. (All items greater than i_* can be removed without affecting the support.)
  Let $I_* = \{i \in I \mid i < i_*\}$ and $X_T(I) = \{i \in B - I \mid s_T(I \cup \{i\}) = s_T(I)\}$.
  Then the canonical parent $\pi_C(I)$ of I is the item set

$$\pi_C(I) = I_* \cup \{i \in X_T(I_*) \mid i > i_*\}.$$

  Intuitively, to find the canonical parent of the item set I, the reduced item set I_* is enhanced by all perfect extension items following the item i_*.

Alternative Description of Closed Item Set Mining

- Note that $\bigcap_{1 \leq k \leq n} t_k$ is the smallest closed item set for a given database T.
- Note also that the set $\{i \in X_T(I_*) \mid i > i_*\}$ need not contain all items $i > i_*$, because a perfect extension of $I_* \cup \{i_*\}$ need not be a perfect extension of I_*, since $K_T(I_*) \supset K_T(I_* \cup \{i_*\})$.
- For the recursive search, the following formulation is useful:

  Let $I \subseteq B$ be a closed item set. The **canonical children** of I (that is, the closed item sets that have I as their canonical parent) are the item sets

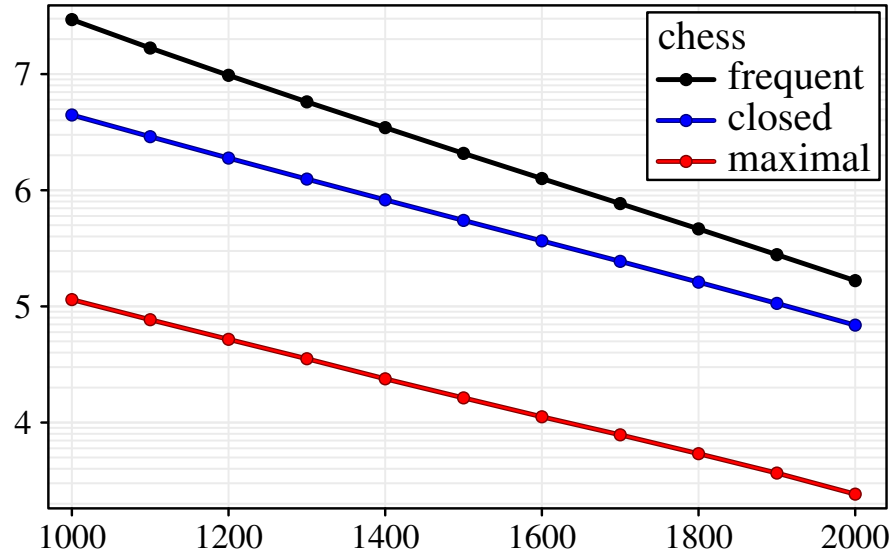
  $$J = I \cup \{i\} \cup \{j \in X_T(I \cup \{i\}) \mid j > i\}$$

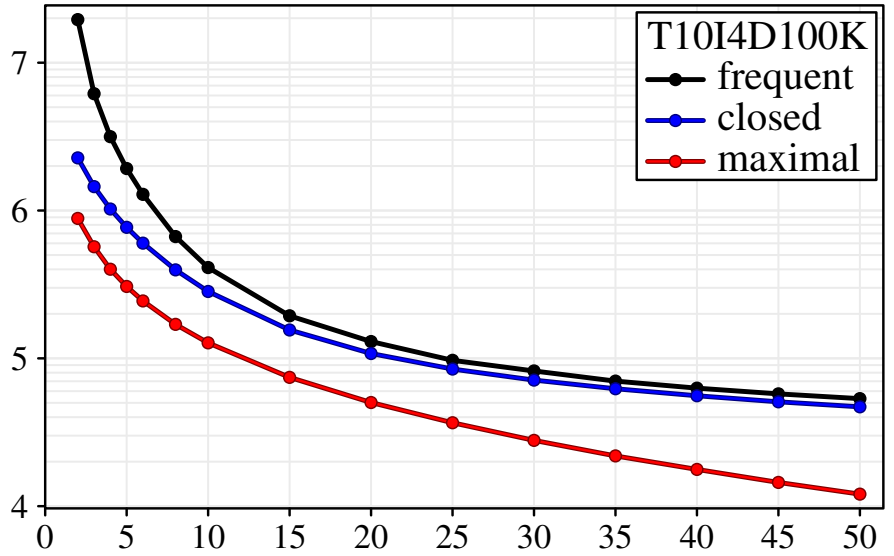
  with $\forall j \in I : i > j$ and $\{j \in X_T(I \cup \{i\}) \mid j < i\} = X_T(J) = \emptyset$.
- The union with $\{j \in X_T(I \cup \{i\}) \mid j > i\}$ represents perfect extension or parent equivalence pruning: all perfect extensions in the tail of $I \cup \{i\}$ are immediately added.
- The condition $\{j \in X_T(I \cup \{i\}) \mid j < i\} = \emptyset$ expresses that there must not be any perfect extensions among the excluded items.

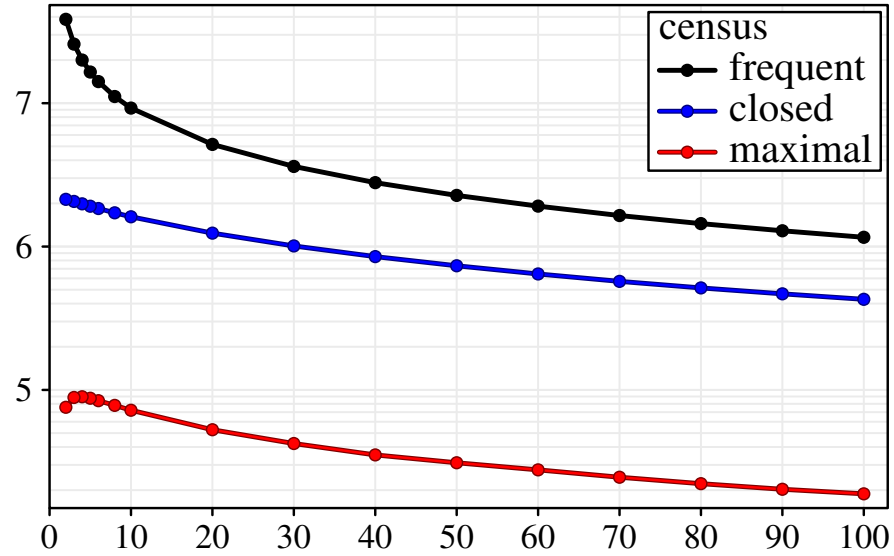
Experiments: Reminder

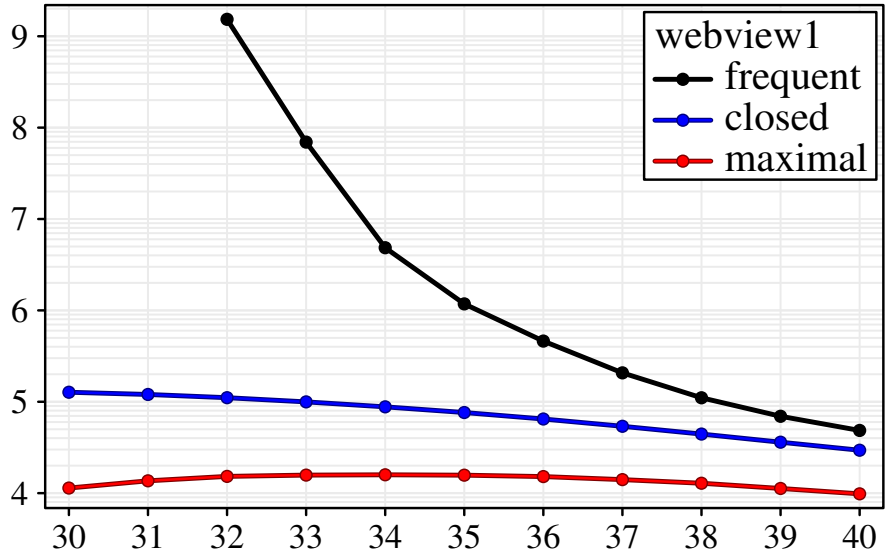
- **Chess**
  A data set listing chess end game positions for king vs. king and rook.
  This data set is part of the UCI machine learning repository.
- **Census**
  A data set derived from an extract of the US census bureau data of 1994,
  which was preprocessed by discretizing numeric attributes.
  This data set is part of the UCI machine learning repository.
- **T10I4D100K**
  An artificial data set generated with IBM's data generator.
  The name is formed from the parameters given to the generator
  (for example: 100K = 100000 transactions).
- **BMS-Webview-1**
  A web click stream from a leg-care company that no longer exists.
  It has been used in the KDD cup 2000 and is a popular benchmark.
- All tests were run on an Intel Core2 Quad Q9650@3GHz with 8GB memory
  running Ubuntu Linux 14.04 LTS (64 bit); programs compiled with GCC 4.8.2.

Types of Frequent Item Sets



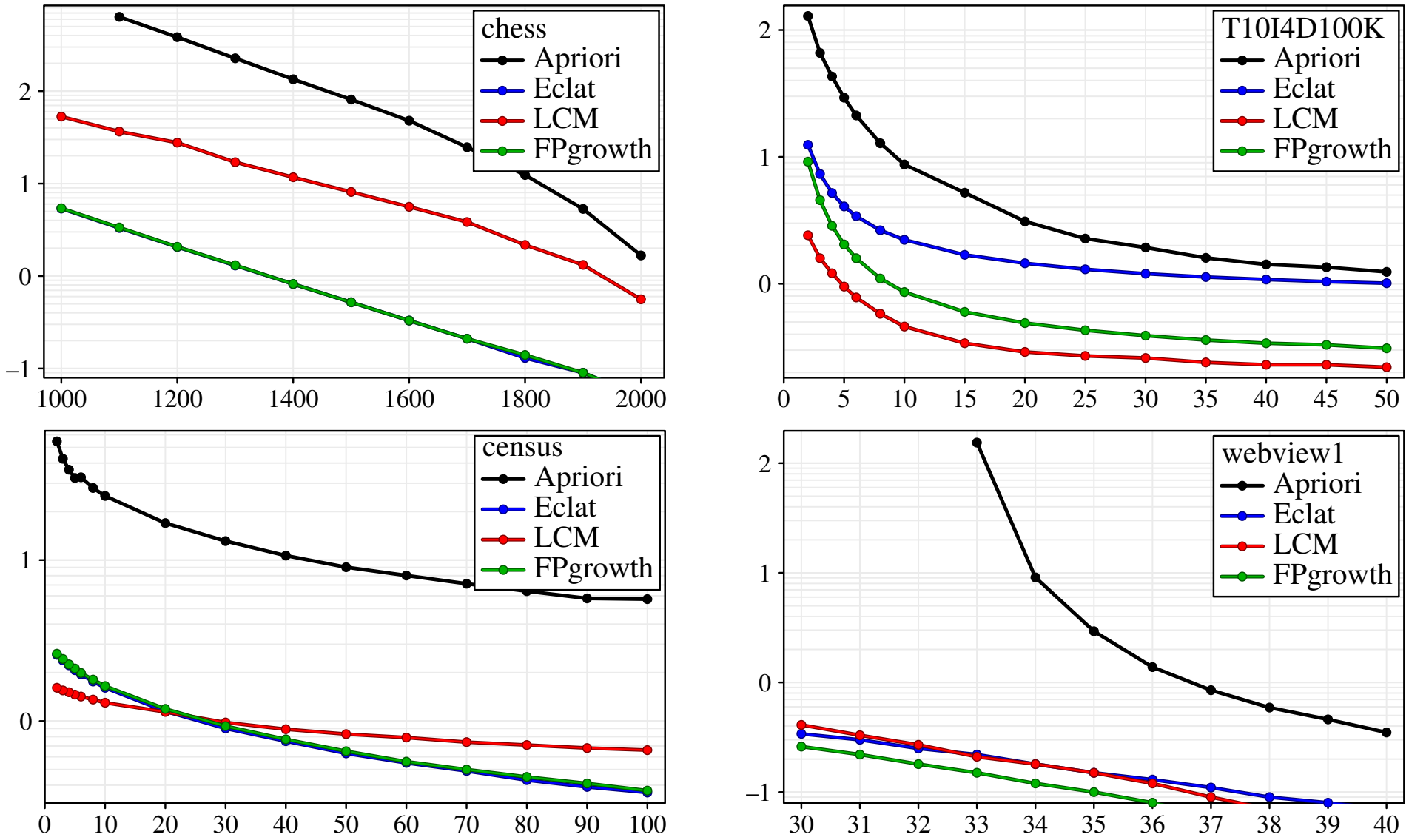






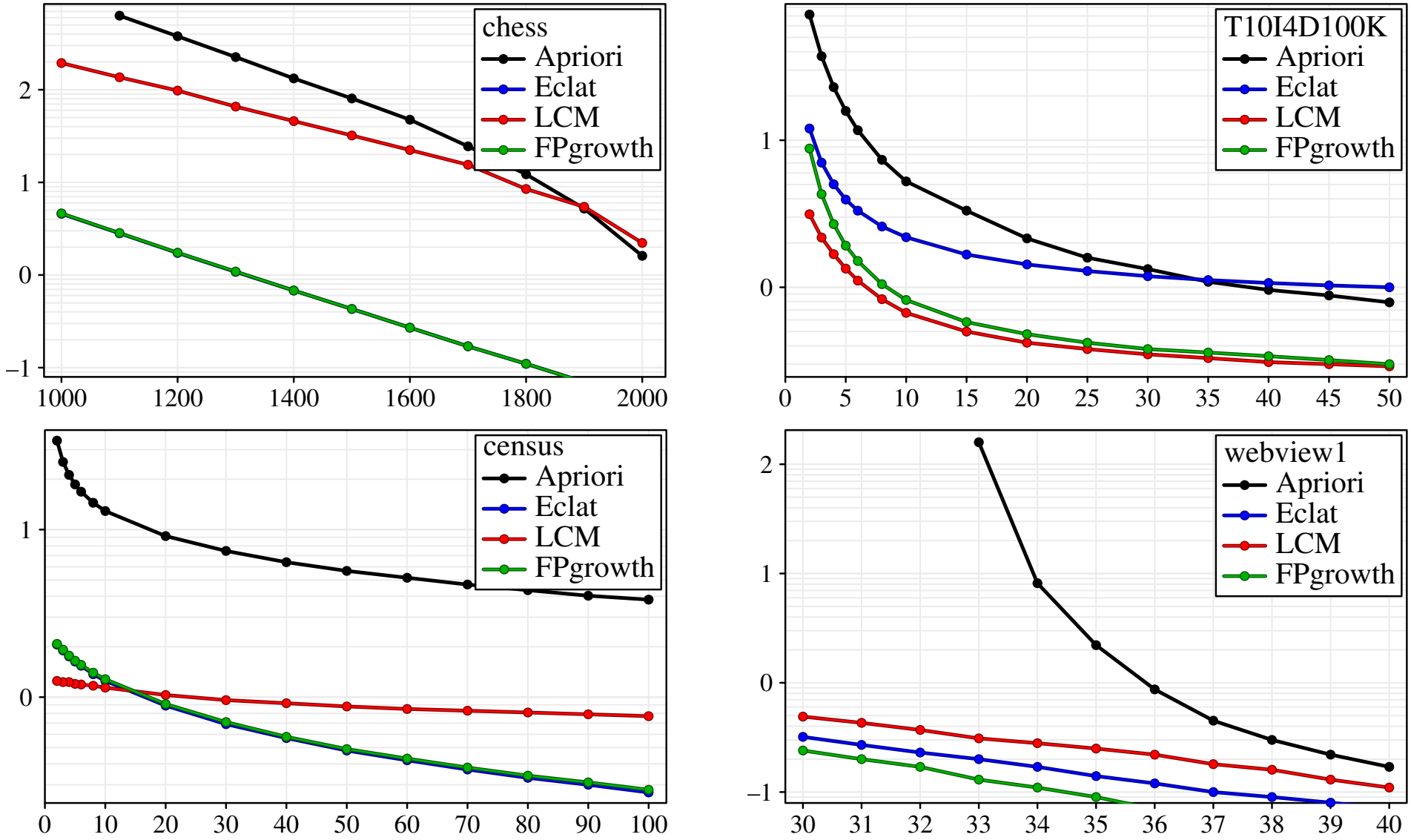
Decimal logarithm of the number of item sets over absolute minimum support.

Experiments: Mining Closed Item Sets



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

Experiments: Mining Maximal Item Sets



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

Additional Frequent Item Set Filtering

Additional Frequent Item Set Filtering

- **General problem of frequent item set mining:**

  The number of frequent item sets, even the number of closed or maximal item sets, can exceed the number of transactions in the database by far.

- Therefore: Additional filtering is necessary to find
  the "relevant" or "interesting" frequent item sets.

- General idea: **Compare support to expectation.**
  - Item sets consisting of items that appear frequently
    are likely to have a high support.
  - However, this is not surprising:
    we expect this even if the occurrence of the items is independent.
  - Additional filtering should remove item sets with a support
    close to the support expected from an independent occurrence.

Additional Frequent Item Set Filtering

Full Independence

- Evaluate item sets with

$$\varrho_{\mathrm{fi}}(I) \;=\; \frac{s_T(I) \cdot n^{|I|-1}}{\prod_{i \in I} s_T(\{i\})} \;=\; \frac{\hat{p}_T(I)}{\prod_{i \in I} \hat{p}_T(\{i\})}.$$

  and require a minimum value for this measure.
  ($\hat{p}_T$ is the probability estimate based on T.)

- Assumes full independence of the items in order to form an expectation about the support of an item set.

- Advantage: Can be computed from only the support of the item set and the support values of the individual items.

- Disadvantage: If some item set I scores high on this measure, then all $J \supset I$ are also likely to score high, even if the items in $J - I$ are independent of I.

Additional Frequent Item Set Filtering

Incremental Independence

- Evaluate item sets with

$$\varrho_{\text{ii}}(I) \;=\; \min_{i \in I} \frac{n\, s_T(I)}{s_T(I - \{i\}) \cdot s_T(\{i\})} \;=\; \min_{i \in I} \frac{\hat{p}_T(I)}{\hat{p}_T(I - \{i\}) \cdot \hat{p}_T(\{i\})}.$$

  and require a minimum value for this measure.
  ($\hat{p}_T$ is the probability estimate based on T.)

- Advantage: If I contains independent items, the minimum ensures a low value.

- Disadvantages: We need to know the support values of all subsets $I - \{i\}$.

  If there exist high scoring independent subsets I_1 and I_2 with $|I_1| > 1$, $|I_2| > 1$, $I_1 \cap I_2 = \emptyset$ and $I_1 \cup I_2 = I$, the item set I still receives a high evaluation.

Additional Frequent Item Set Filtering

Subset Independence

- Evaluate item sets with

$$\varrho_{\mathrm{si}}(I) \;=\; \min_{J \subset I, J \neq \emptyset} \frac{n\, s_T(I)}{s_T(I-J) \cdot s_T(J)} \;=\; \min_{J \subset I, J \neq \emptyset} \frac{\hat{p}_T(I)}{\hat{p}_T(I-J) \cdot \hat{p}_T(J)}.$$

  and require a minimum value for this measure.
  ($\hat{p}_T$ is the probability estimate based on T.)

- Advantage: Detects all cases where a decomposition is possible and evaluates them with a low value.

- Disadvantages: We need to know the support values of all proper subsets J.

- Improvement: Use incremental independence and in the minimum consider only items $\{i\}$ for which $I - \{i\}$ has been evaluated high.

  This captures subset independence "incrementally".

Summary Frequent Item Set Mining

- With a **canonical form** of an item set the Hasse diagram
  can be turned into a much simpler **prefix tree**
  ($\Rightarrow$ divide-and-conquer scheme using conditional databases).

- **Item set enumeration** algorithms differ in:
  - the **traversal order** of the prefix tree:
    (breadth-first/levelwise versus depth-first traversal)
  - the **transaction representation**:
    horizontal (item arrays) versus *vertical* (transaction lists)
    versus *specialized data structures* like FP-trees
  - the **types of frequent item sets** found:
    frequent versus *closed* versus *maximal item sets*
    (additional pruning methods for closed and maximal item sets)

- An alternative are **transaction set enumeration** or **intersection** algorithms.

- **Additional filtering** is necessary to reduce the size of the output.

Mining More Complex Patterns

Mining More Complex Patterns

- The search scheme in Frequent Graph/Tree/Sequence mining is the same, namely the general scheme of searching with a canonical form.

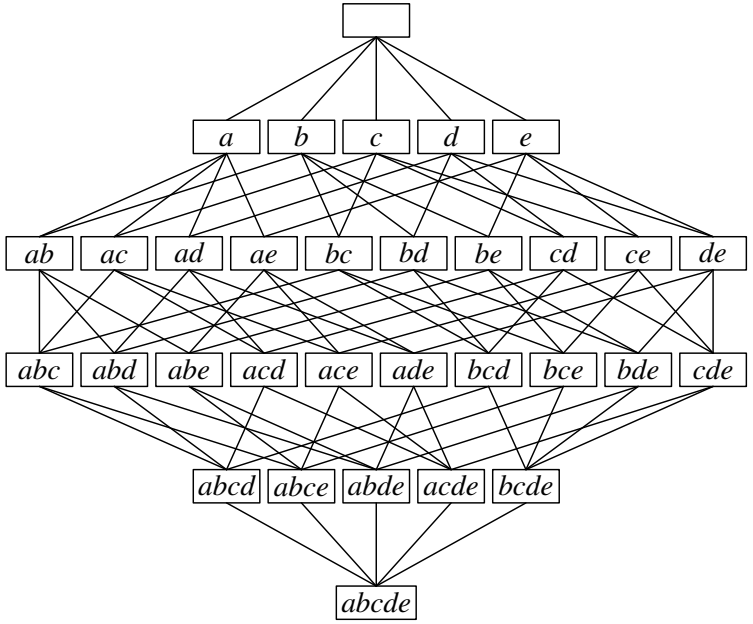
- **Frequent (Sub)Graph Mining** comprises the other areas:
  - Trees are special graphs, namely graphs that are singly connected.
  - Sequences can be seen as special trees, namely chains (only one or two branches — depending on the choice of the root).

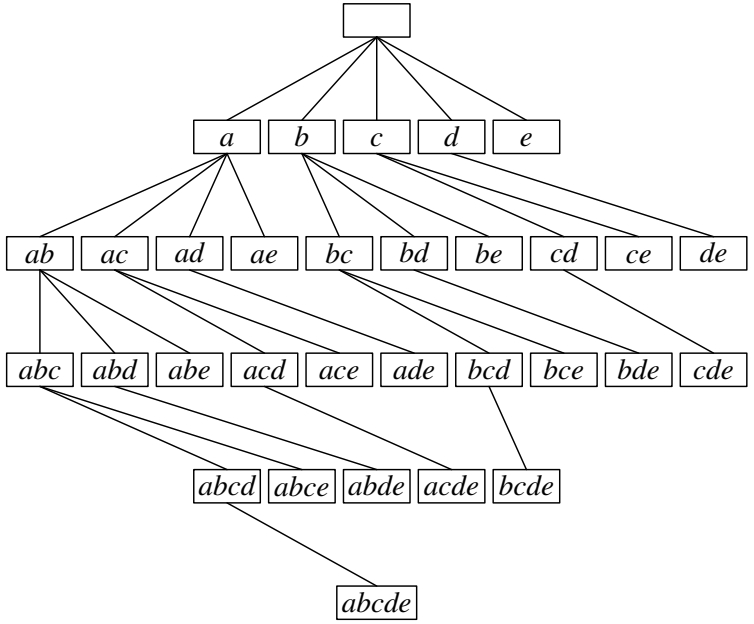
- **Frequent Sequence Mining** and **Frequent Tree Mining** can exploit:
  - Specialized canonical forms that allow for more efficient checks.
  - Special data structures to represent the database to mine, so that support counting becomes more efficient.

- We will treat *Frequent (Sub)Graph Mining* first and will discuss optimizations for the other areas later.

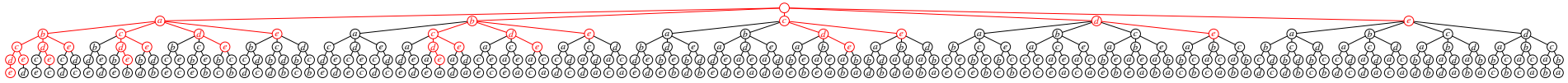
Search Space Comparison

Search space for sets: (5 items)





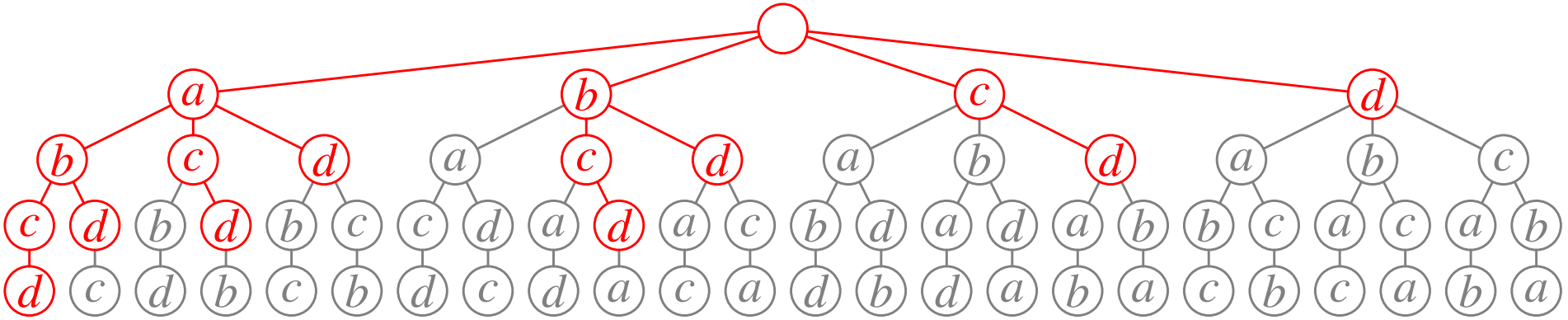
Search space for sequences: (5 items, no repetitions)



- Red part corresponds to search space for sets (top right).

Search Space Comparison

Search space for sequences: (4 items, no repetitions)



- Red part corresponds to search space for sets.
- The search space for (sub)sequences is considerably larger than the one for sets.
- However: support of (sub)sequences reduces much faster with increasing length.
  - Out of k items only one set can be formed,
    but $k!$ sequences (every order yields a different sequences).
  - All $k!$ sequences cover the set (tendency towards higher support).
  - To cover a specific sequence, a specific order is required.
    (tendency towards lower support).

Summary Frequent Pattern Mining

Summary Frequent Pattern Mining

- Possible types of patterns: **item sets**, **sequences**, **trees**, and **graphs**.
- A core ingredient of the search is a **canonical form** of the type of pattern.
  - Purpose: ensure that each possible pattern is processed at most once. (Discard non-canonical code words, process only canonical ones.)
  - It is desirable that the canonical form possesses the **prefix property**.
  - Except for general graphs there exist **canonical extension rules**.
  - For general graphs, **restricted extensions** allow to reduce the number of actual canonical form tests considerably.
- Frequent pattern mining algorithms prune with the **Apriori property**:

$$\forall P : \forall S \supset P : \quad s_{\mathcal{D}}(P) < s_{\min} \rightarrow s_{\mathcal{D}}(S) < s_{\min}.$$

  That is: **No super-pattern of an infrequent pattern is frequent.**
- **Additional filtering** is important to single out the relevant patterns.