Frequent Pattern Mining

Christian Borgelt

School of Computer Science
Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany

`christian@borgelt.net`
`http://www.borgelt.net/`
`http://www.borgelt.net/teach/fpm/`

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: 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 $s_{\min} \in \mathbb{N}$, $0 < s_{\min} \leq n$, 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 a transaction has to satisfy. Transactions that do not satisfy this condition are removed from the cover.

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

  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:

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

  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)
  - $a \leq b \land b \leq a \;\Rightarrow\; a = b$ (anti-symmetry)
  - $a \leq b \land b \leq c \;\Rightarrow\; a \leq c$ (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:
  - $a \leq b \lor b \leq a$ (totality)

Reminder: Partially Ordered Sets and Hasse Diagrams

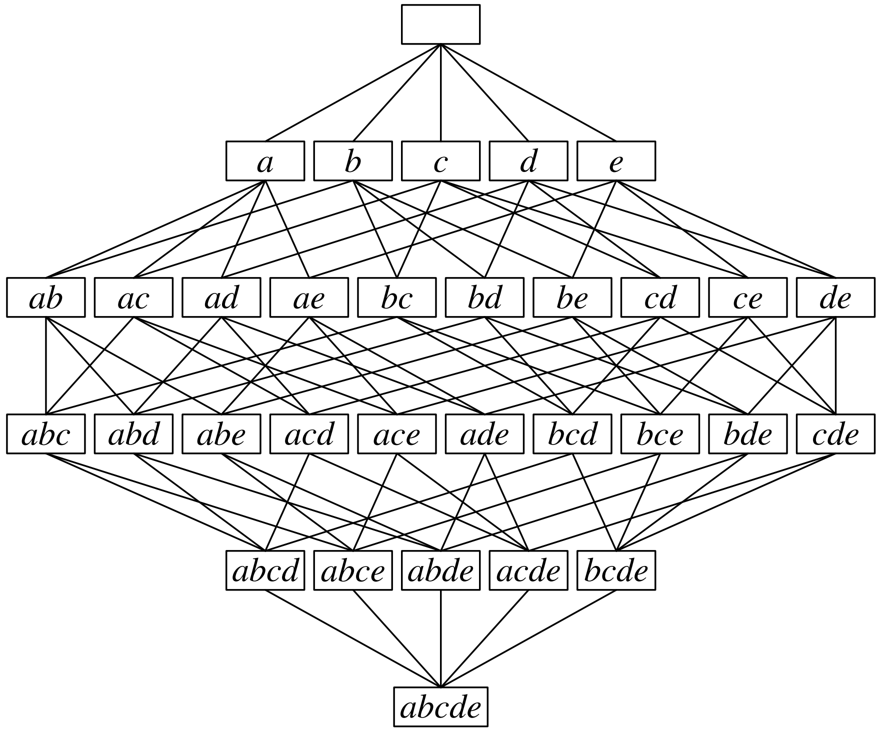
- A finite partially ordered set $(S, \leq)$ can be depicted as a (directed) acyclic graph G, which is called **Hasse diagram**.

- 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.

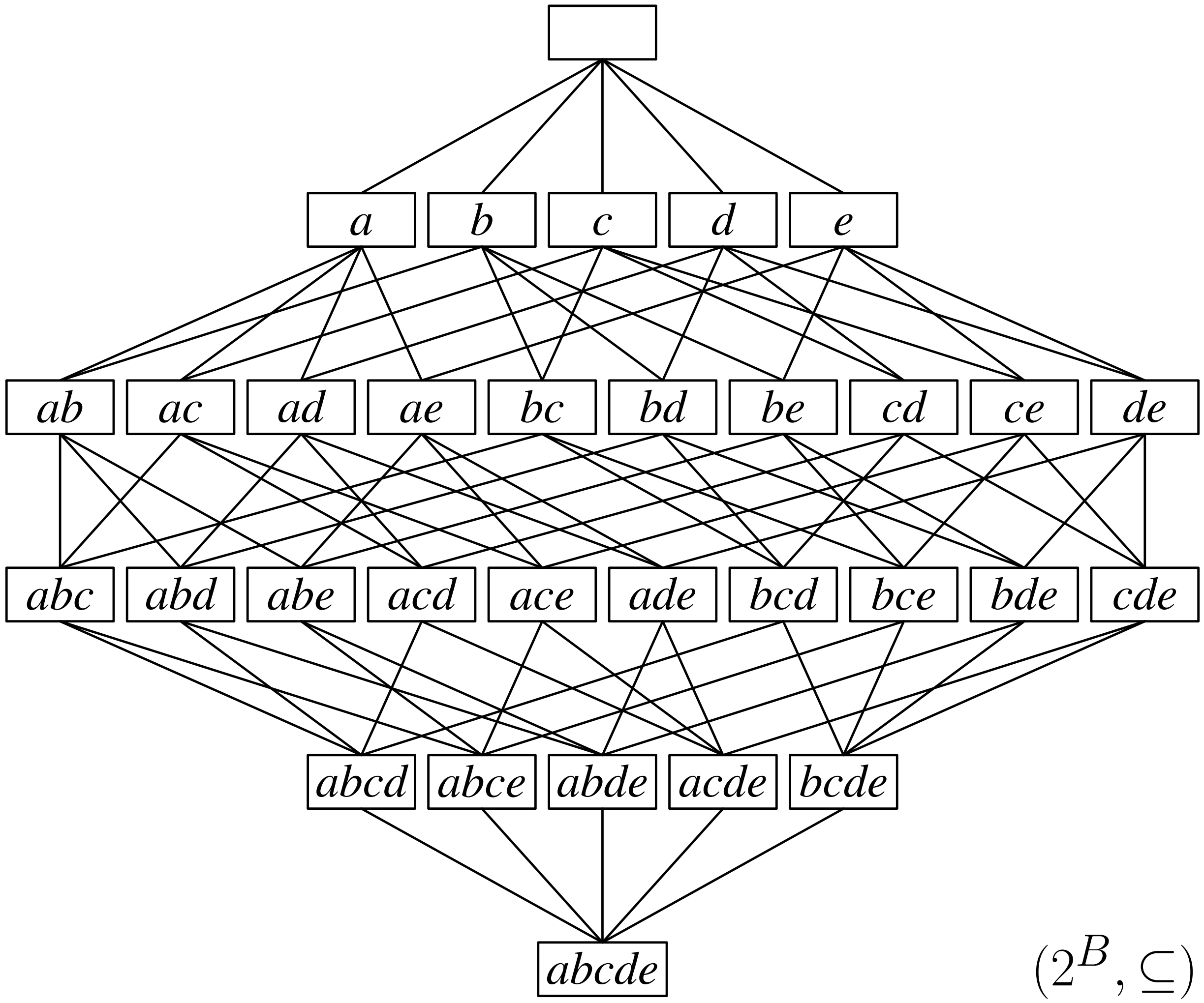
Searching for Frequent Item Sets

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

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

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 $\{a, b, c, d, e\} = B$:



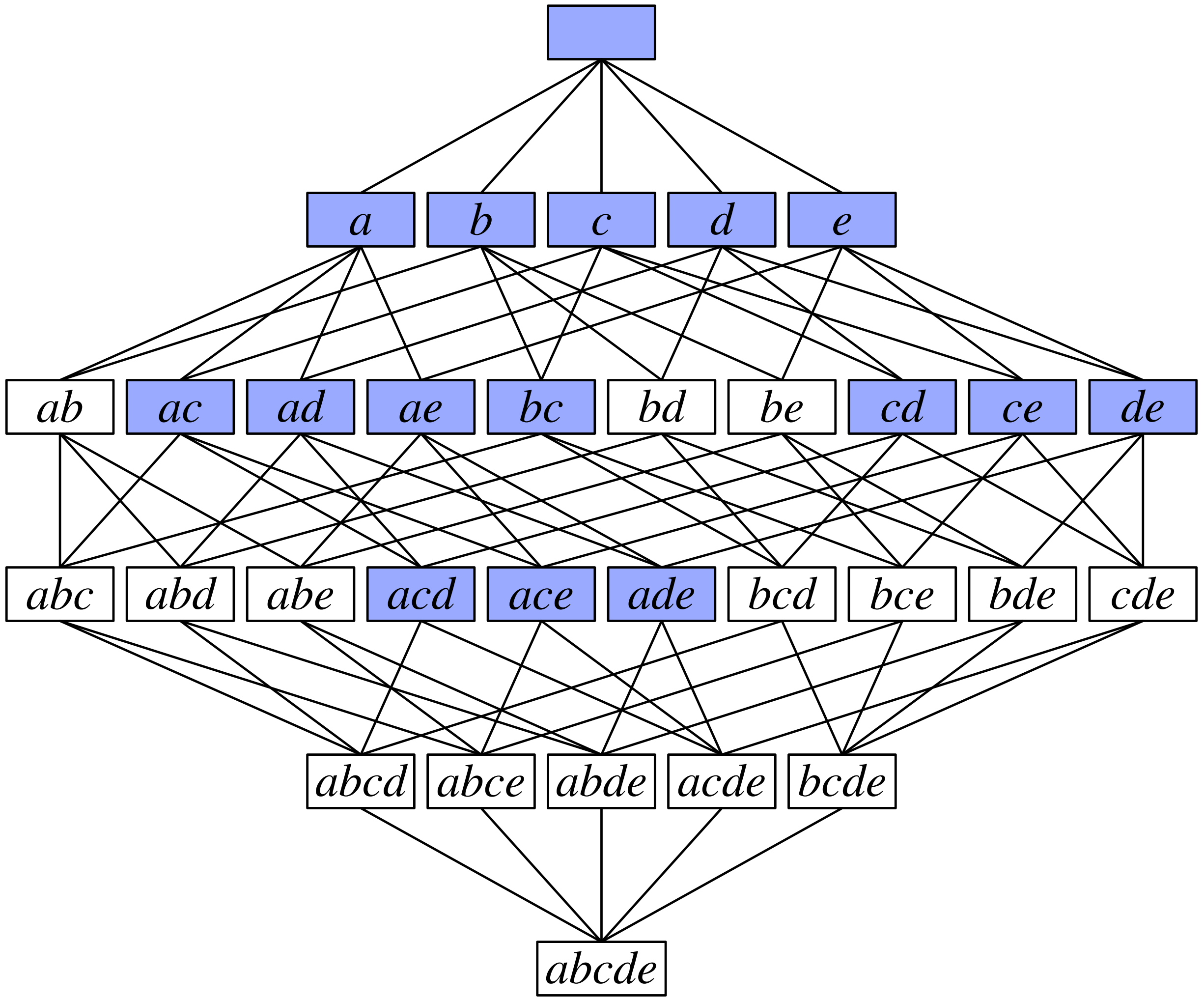
$(2^B, \subseteq)$

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

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



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

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 ∗)
    $F_k$ := prune($E_k, T, s_{\min}$);         (∗ and determine the frequent ones ∗)
    while $F_k \neq \emptyset$ do begin         (∗ while there are frequent item sets ∗)
        $E_{k+1}$ := candidates($F_k$);         (∗ create candidates with one item more ∗)
        $F_{k+1}$ := 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?

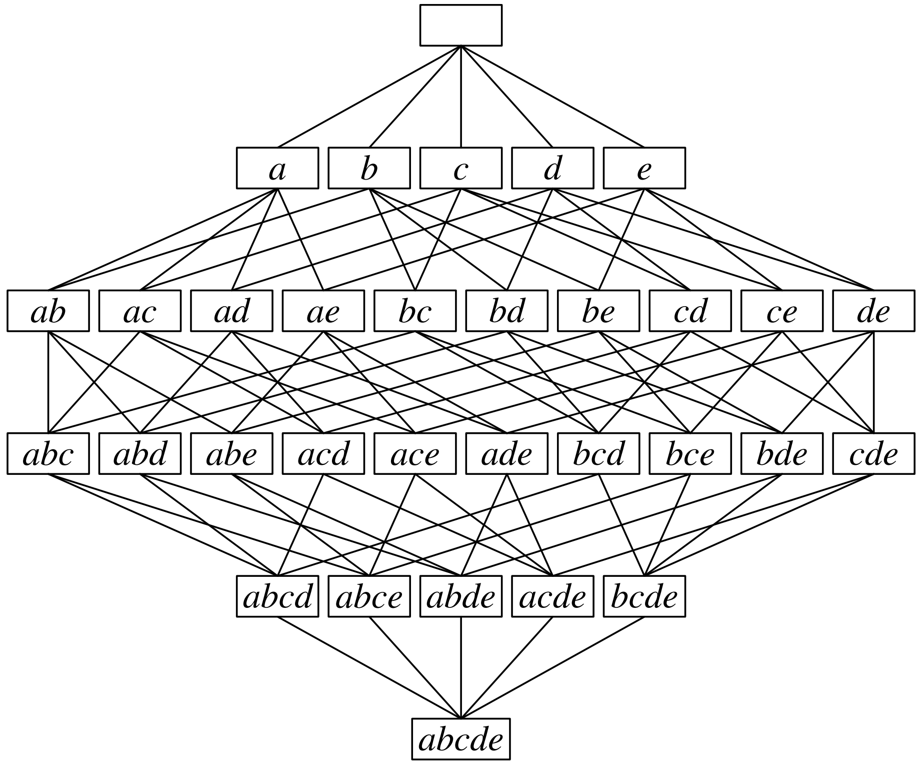
  More generally:
  How can we make sure that any candidate item set is generated at most once?

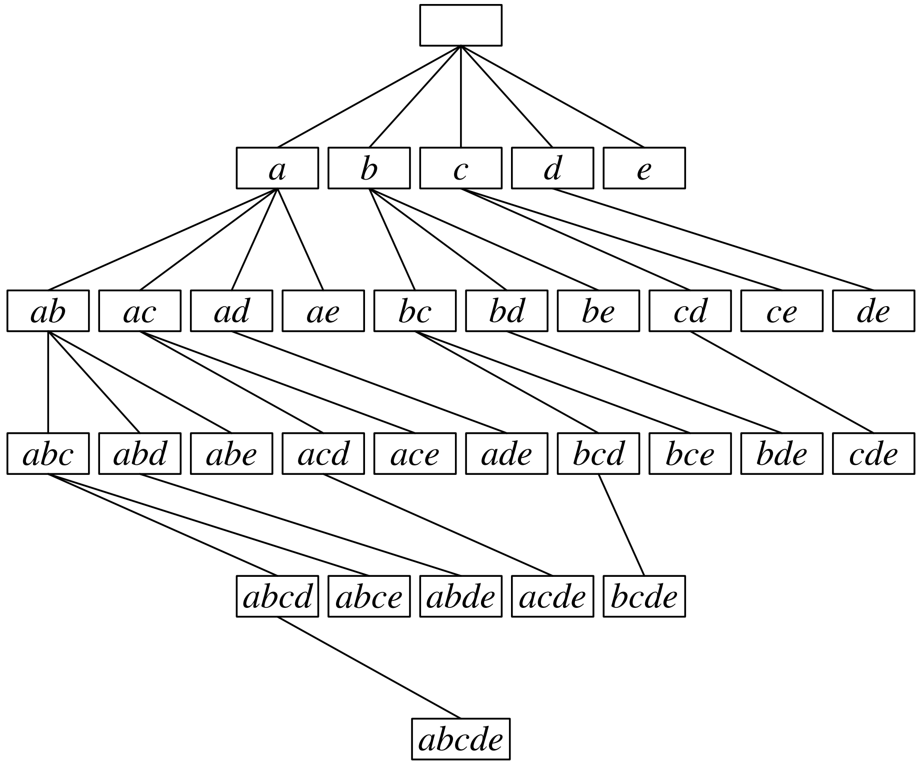
- **Idea:** Assign to each item set a unique parent item set, from which this item set is to be generated.

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 parents** of an item set I is

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

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

- In order to single out one element of $P(I)$, the **canonical parent** $p_c(I)$, 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

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

  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** $p_c(I)$ 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

$$p_c(I) = I - \{\max_{a \in I} a\}.$$

- **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 $p_c(I) = \{a, b, d\}$.

- 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.

⇒ 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).

⇒ 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 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:**

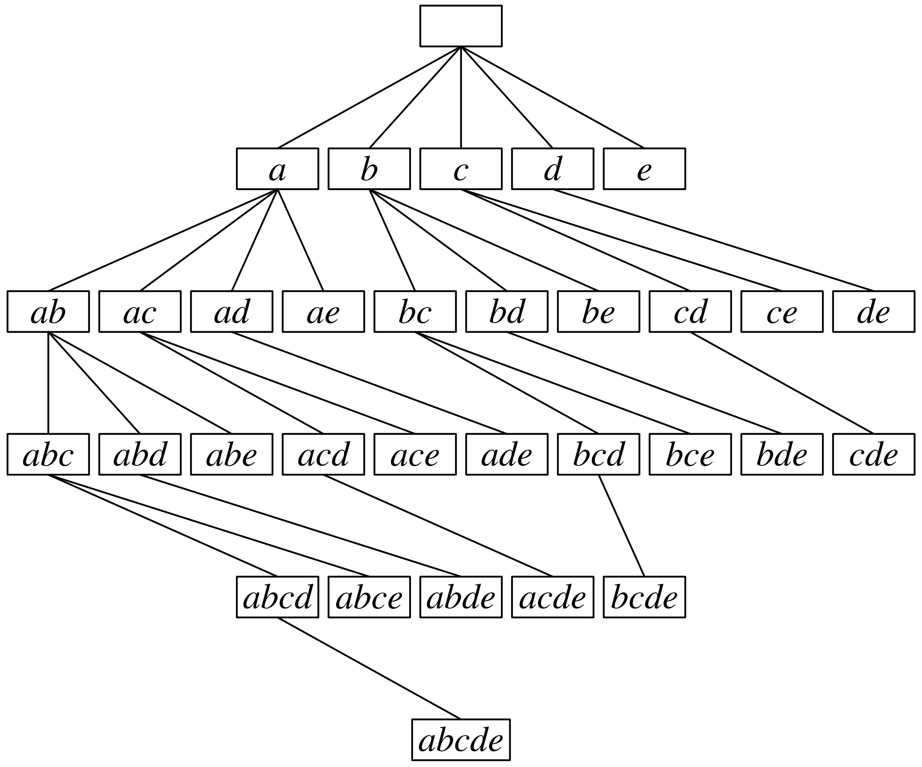
  For a given (canonical) code word of a frequent item set:
  - Generate all possible extensions by a single item,
    where this item succeeds the last letter (item) of the given code word.
    This is done by simply **appending the item** to the code word.
  - If the item set described by the resulting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

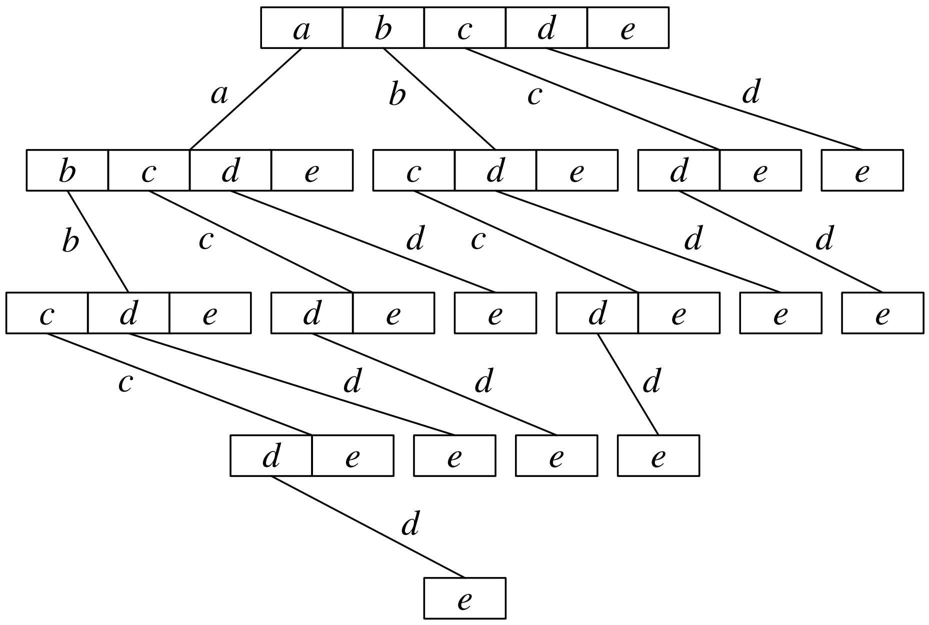
- This search scheme generates each candidate item set **at most once**.

Canonical Parents and Prefix Trees

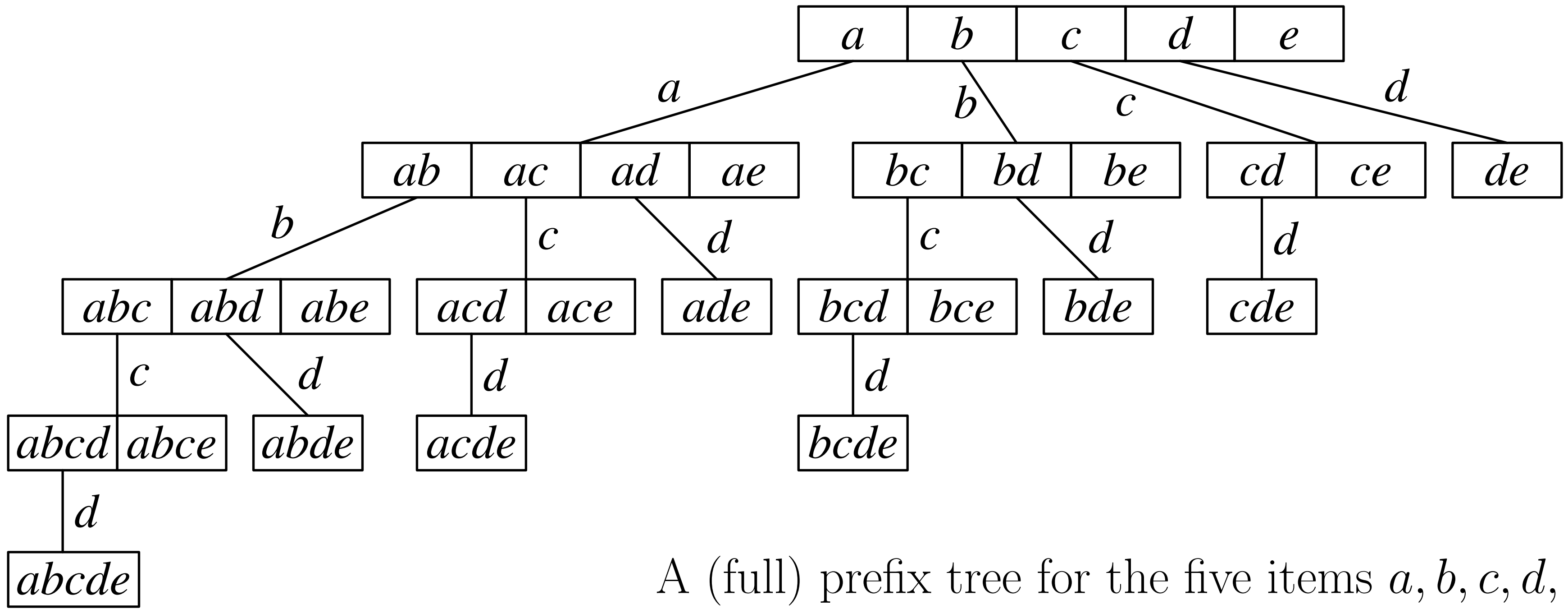
- Item sets, whose canonical code words share the same longest proper prefix, are siblings, because they have (by definition) the same canonical parent.
- This allows us to represent the canonical parent tree as a **prefix tree** or **trie**.

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.

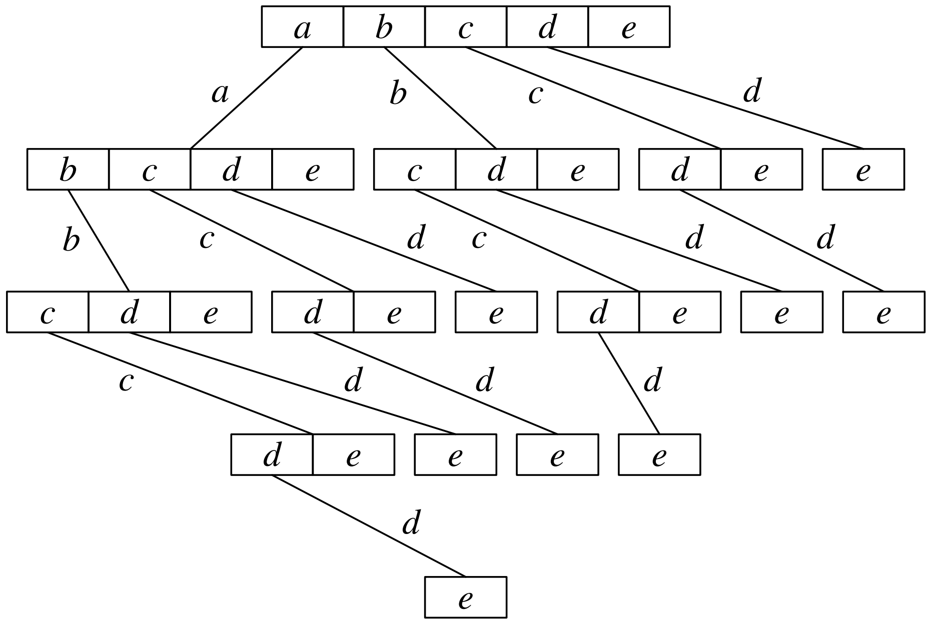
  - Sort the items w.r.t. their support (frequency of occurrence).
  - Sort descendingly: Prefix tree has fewer, but larger nodes.
  - Sort ascendingly: Prefix tree has more, but smaller nodes.

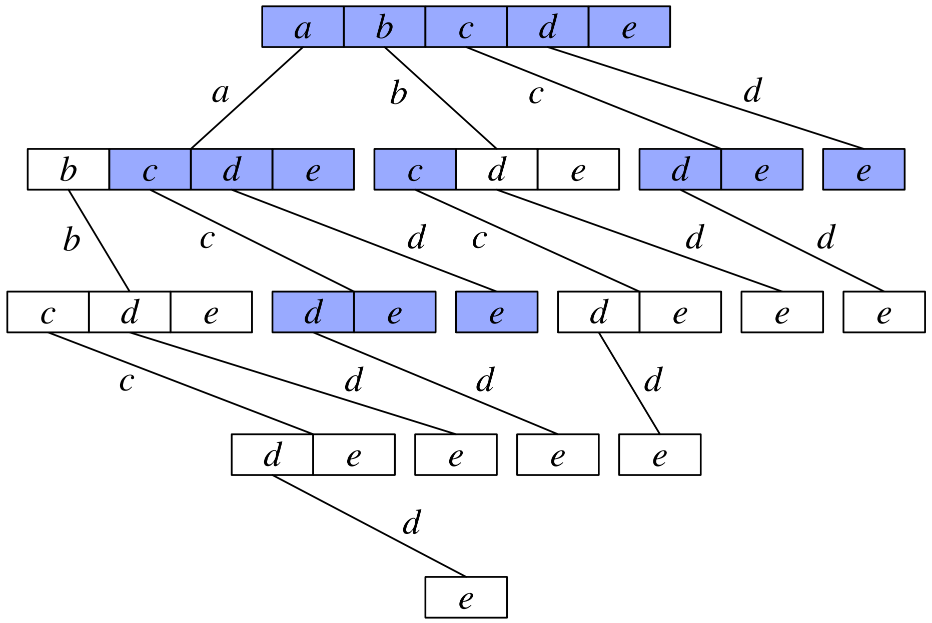
- **Extension of this Idea:**

  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

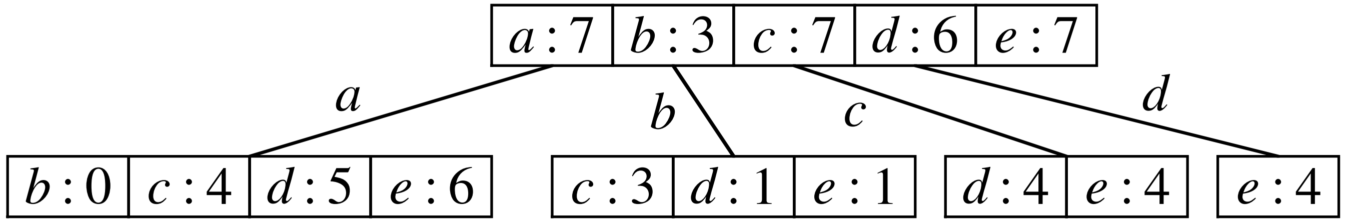
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$

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

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

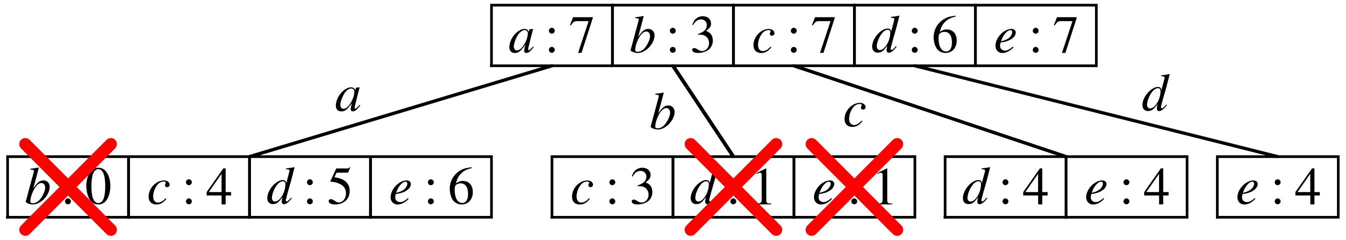


a : 7 | b : 3 | c : 7 | d : 6 | e : 7
a
b
c
d
b : 0 | c : 4 | d : 5 | e : 6
c : 3 | d : 1 | e : 1
d : 4 | e : 4
e : 4

- 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 recursive procedure).

Apriori: Levelwise Search

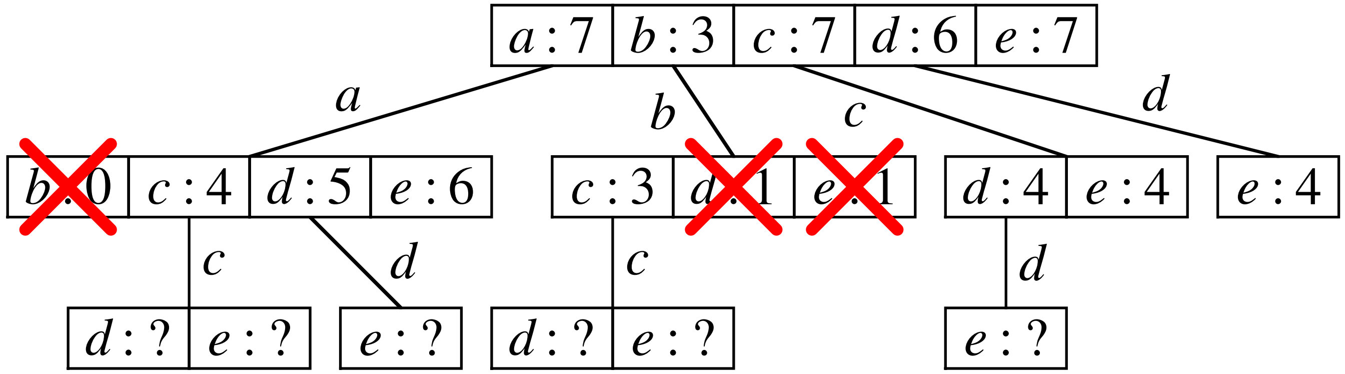
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

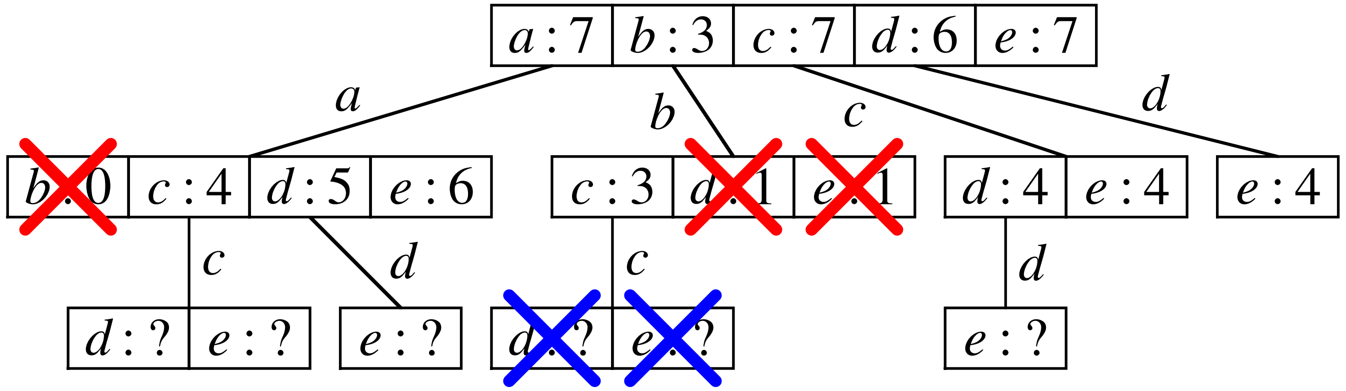
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

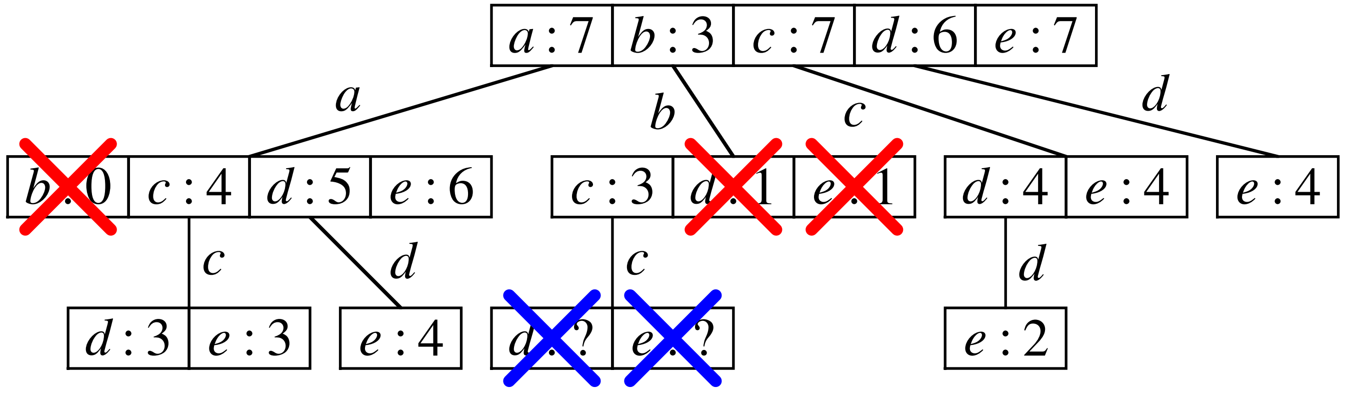
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

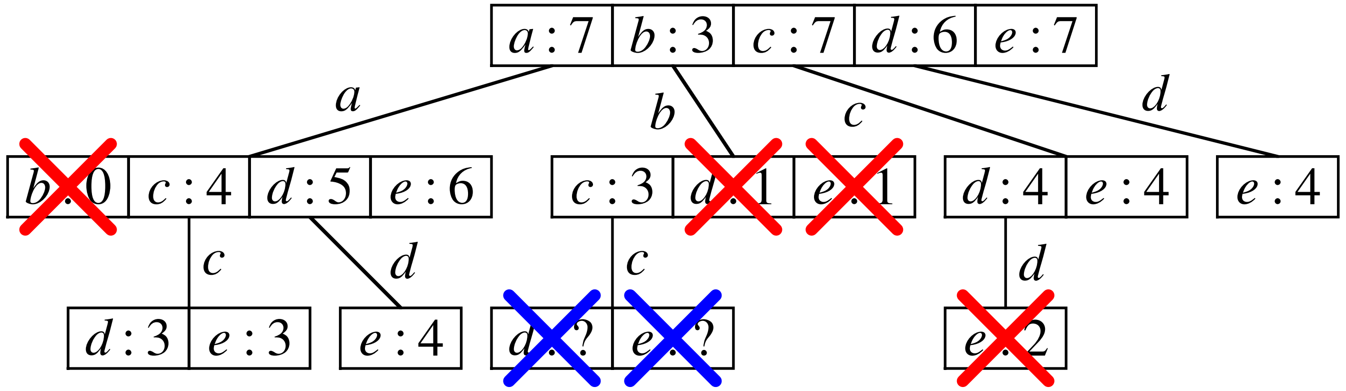
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Only the remaining four item sets of size 3 are evaluated.
- No other item sets of size 3 can be frequent.

Apriori: Levelwise Search

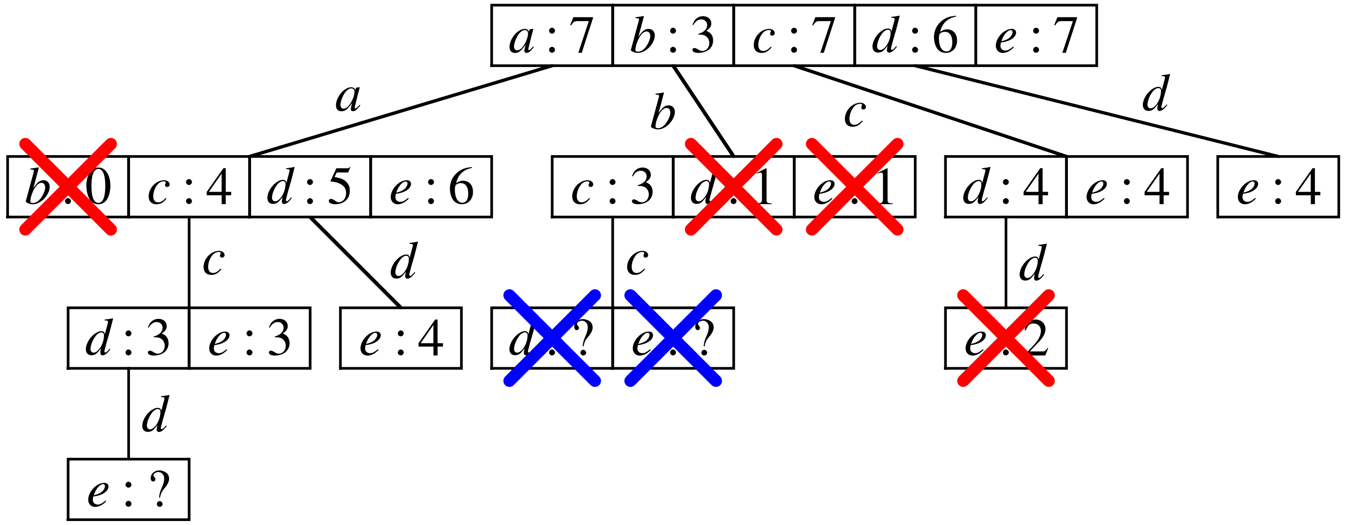
1:	$\{a, d, e\}$
2:	$\{b, c, d\}$
3:	$\{a, c, e\}$
4:	$\{a, c, d, e\}$
5:	$\{a, e\}$
6:	$\{a, c, d\}$
7:	$\{b, c\}$
8:	$\{a, c, d, e\}$
9:	$\{b, c, e\}$
10:	$\{a, d, e\}$



- 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)

Apriori: Levelwise Search

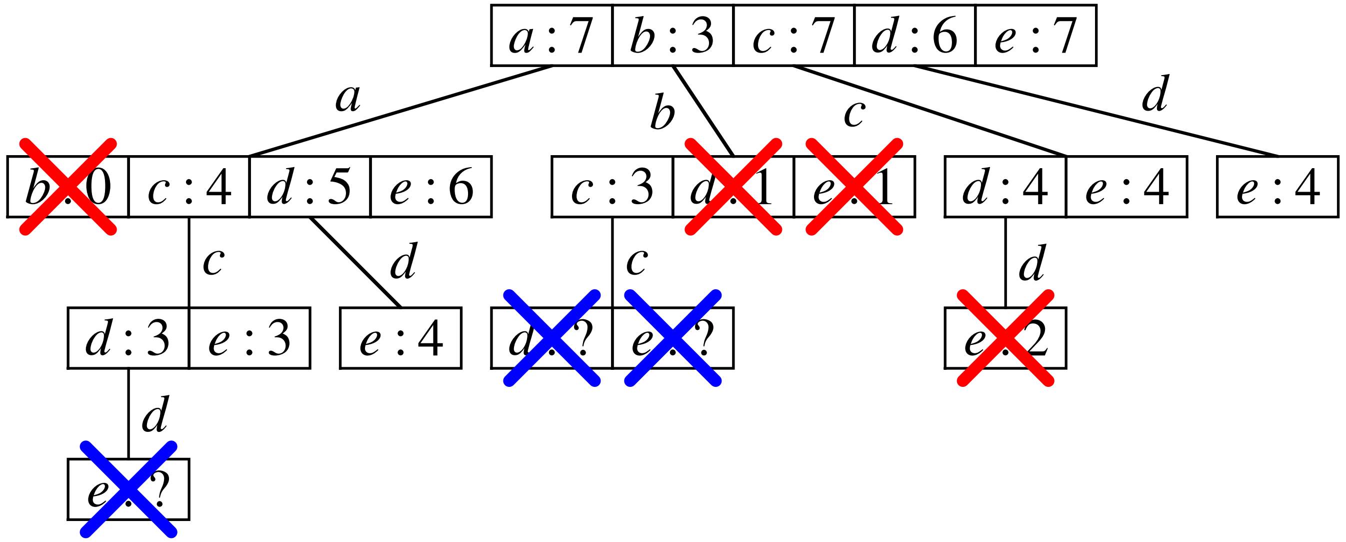
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



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

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 depends 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.

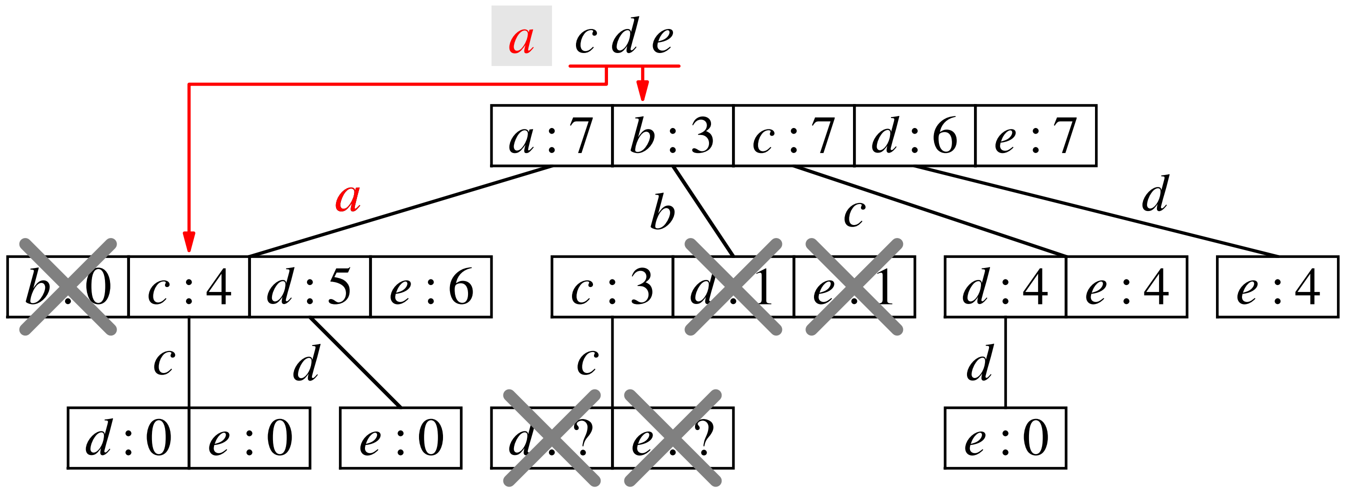
Apriori: Recursive Counting

- 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 rest 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 it 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:
$\{a, c, d, e\}$

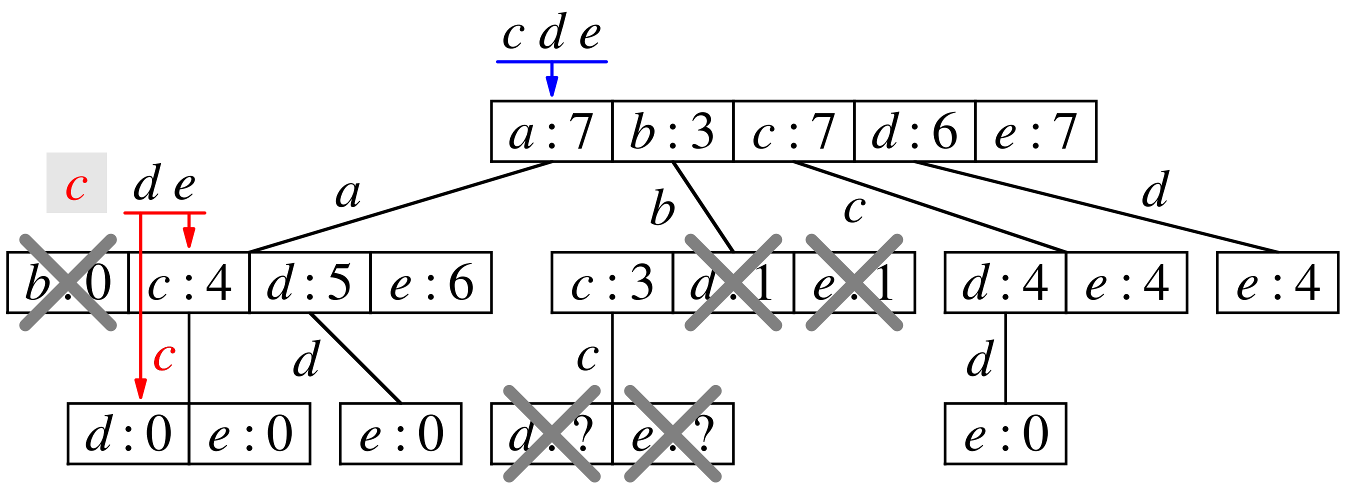


current
item set size: 3

---

processing: a

processing: c

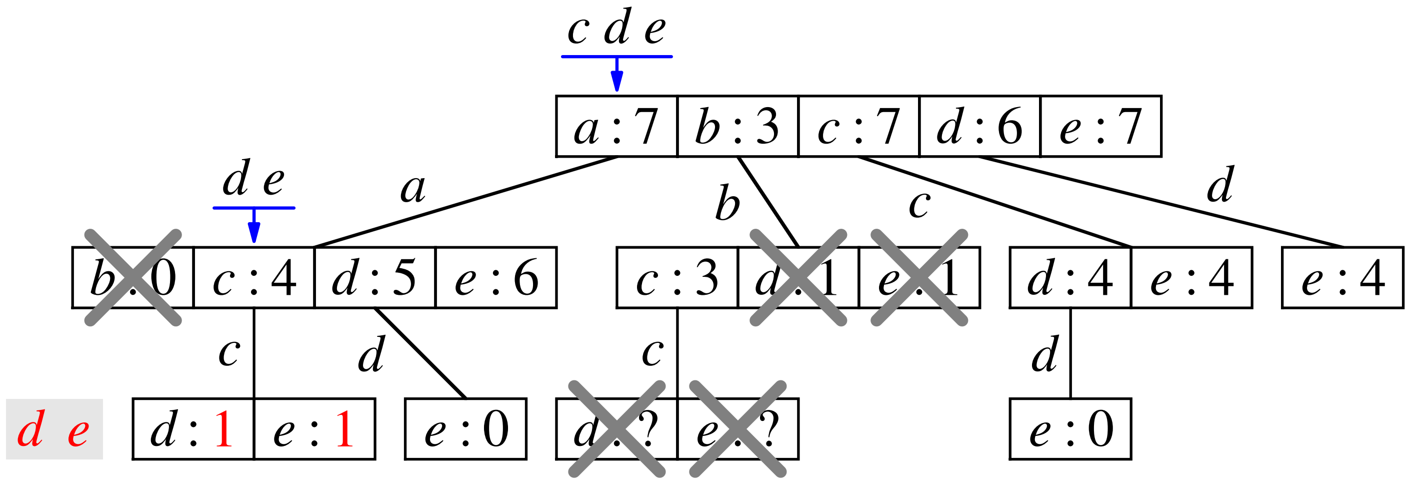


Apriori: Recursive Counting

processing: a

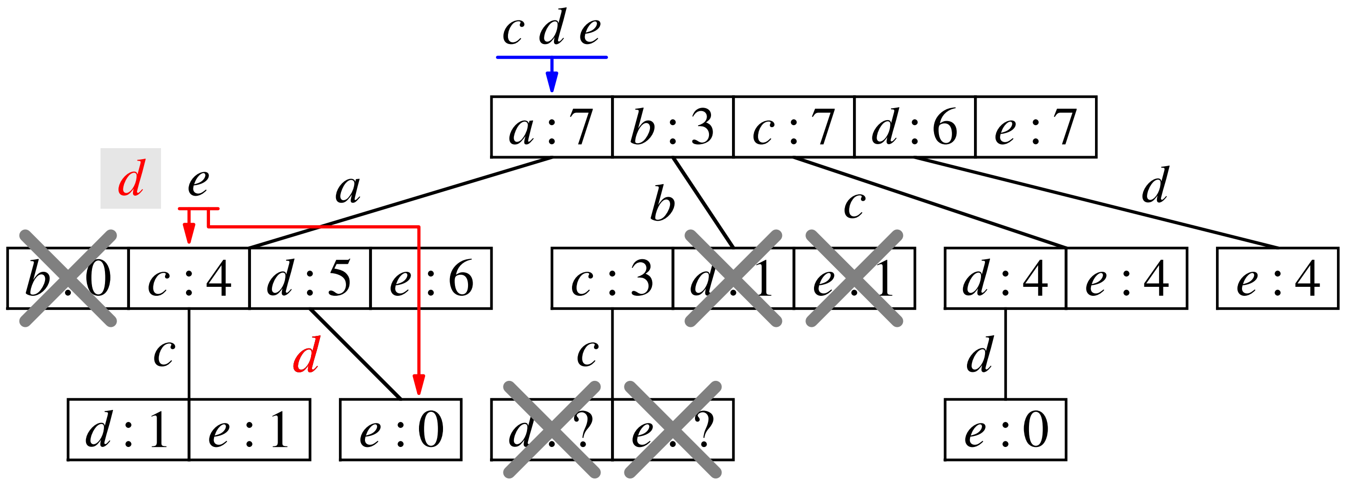
processing: c

processing: $d\ e$



processing: a

processing: d

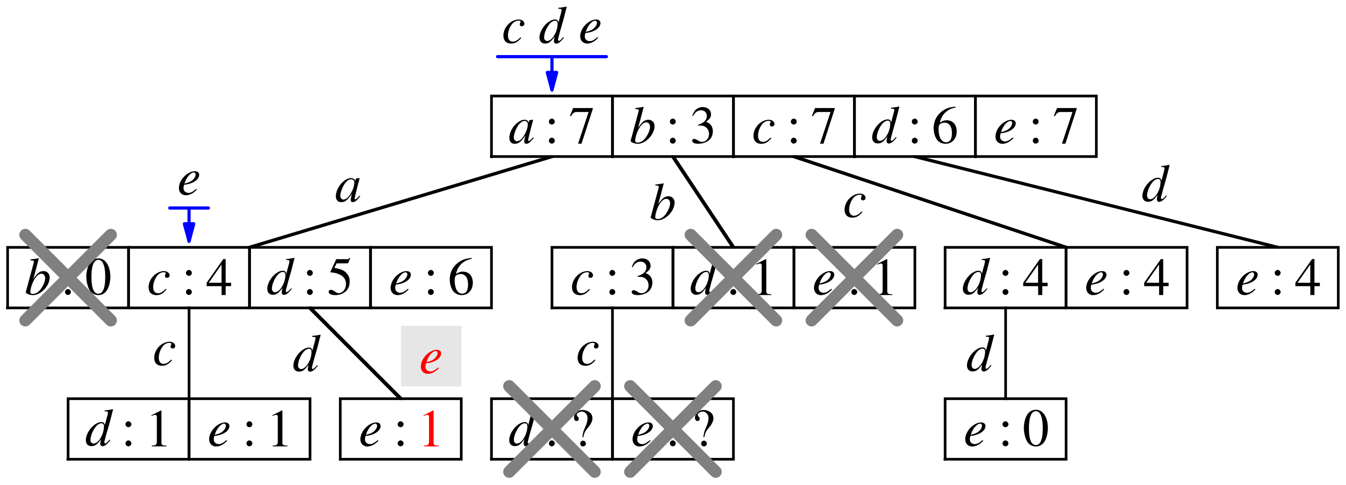


Apriori: Recursive Counting

processing: a

processing: d

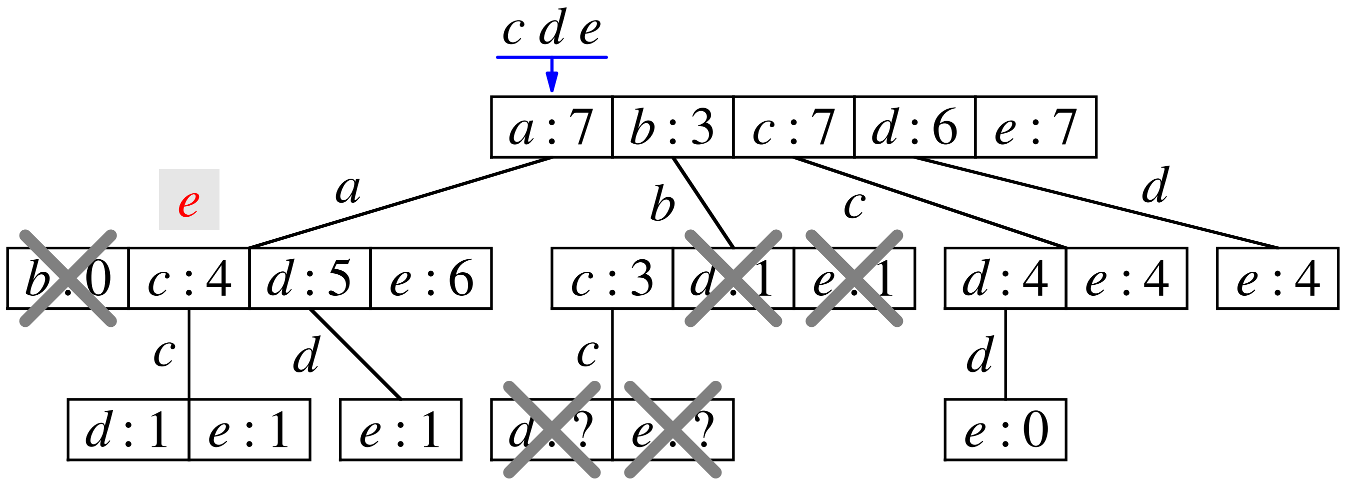
processing: e



processing: a

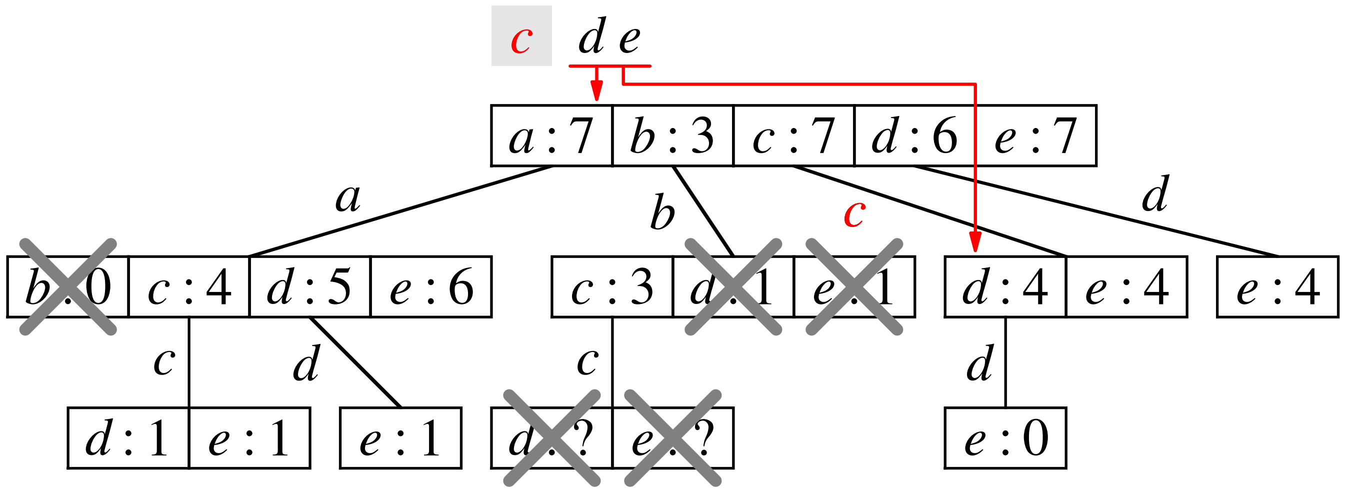
processing: e

(skipped:
too few items)



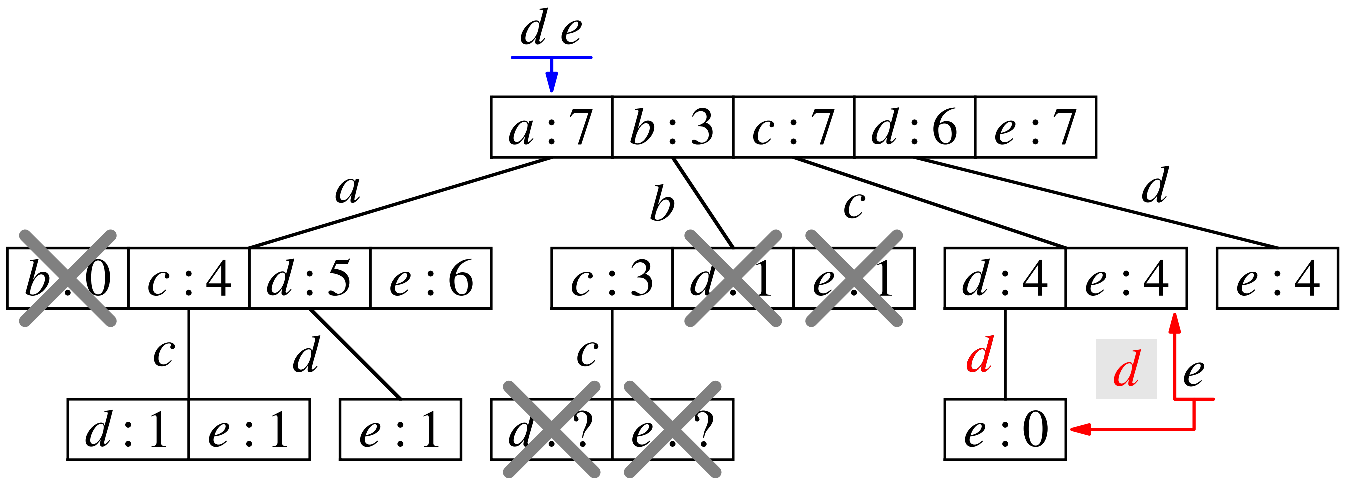
Apriori: Recursive Counting

processing: c



processing: c

processing: d

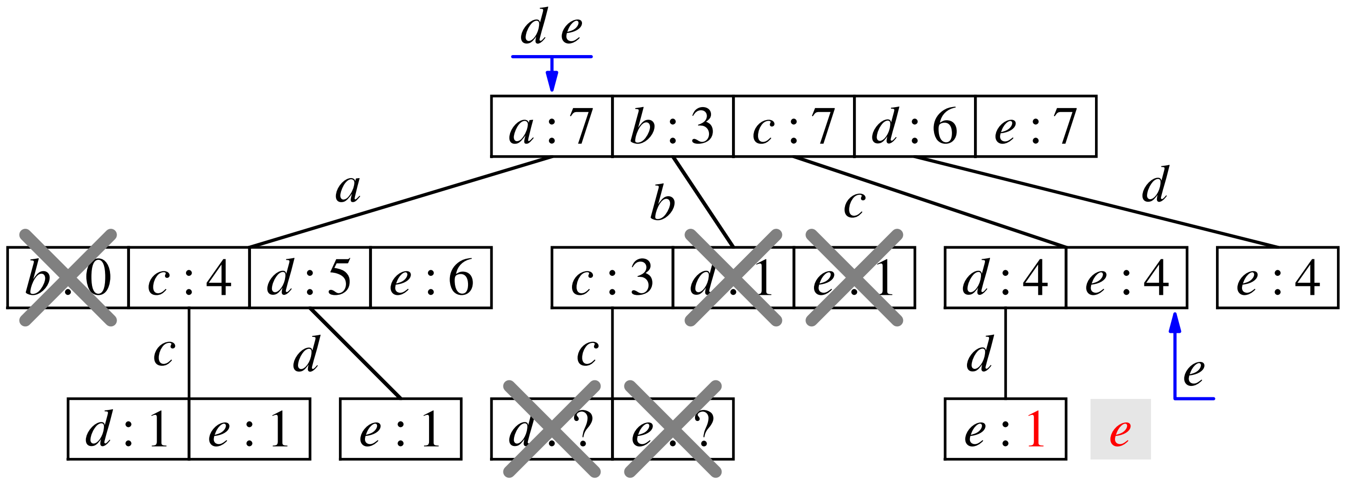


Apriori: Recursive Counting

processing: c

processing: d

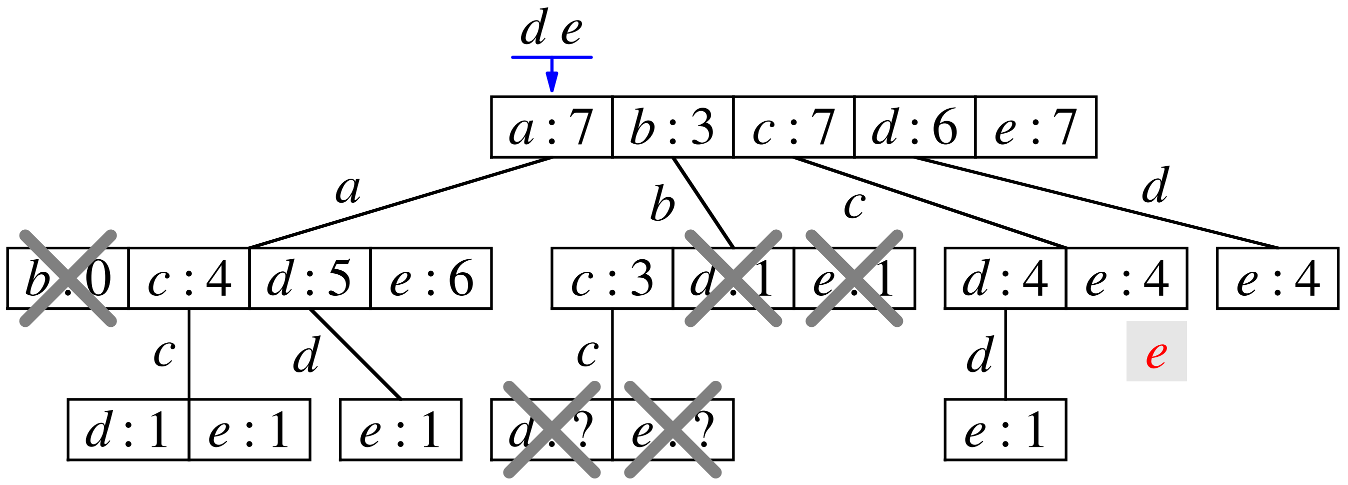
processing: e



processing: c

processing: e

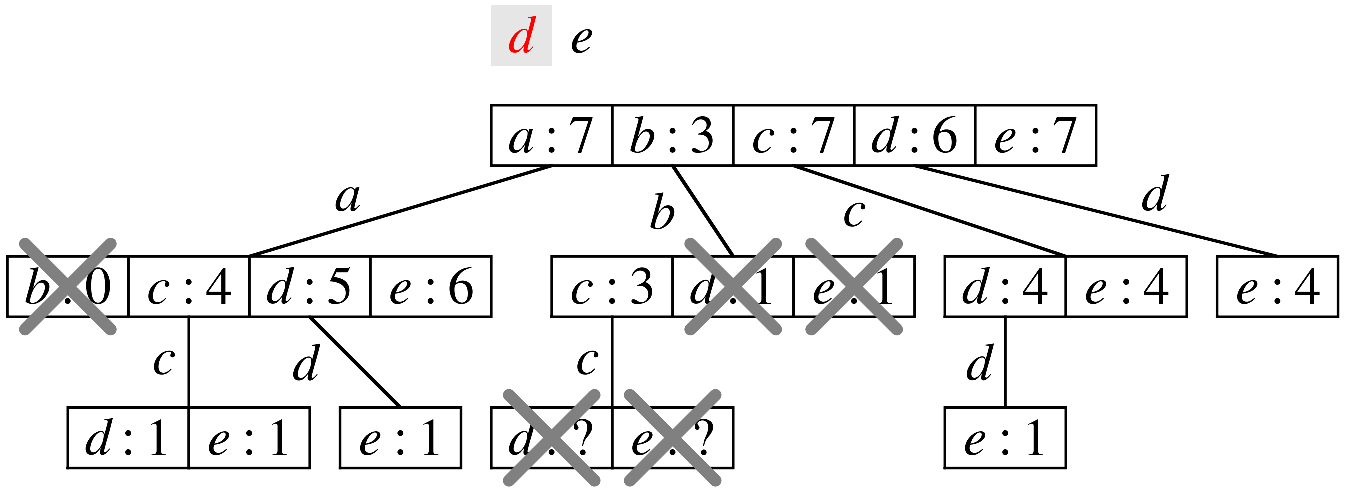
(skipped:
too few items)



Apriori: Recursive Counting

processing: d

(skipped:
too few items)



- Processing an item set 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.

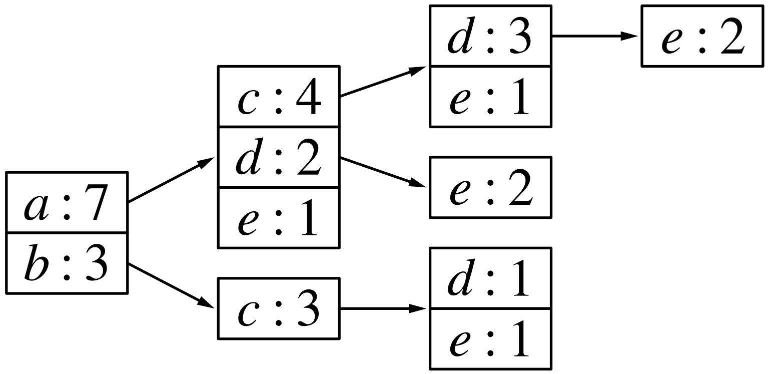
Organization as a Prefix Tree:

- 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.

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.

Software

- `http://www.borgelt.net/apriori.html`

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.

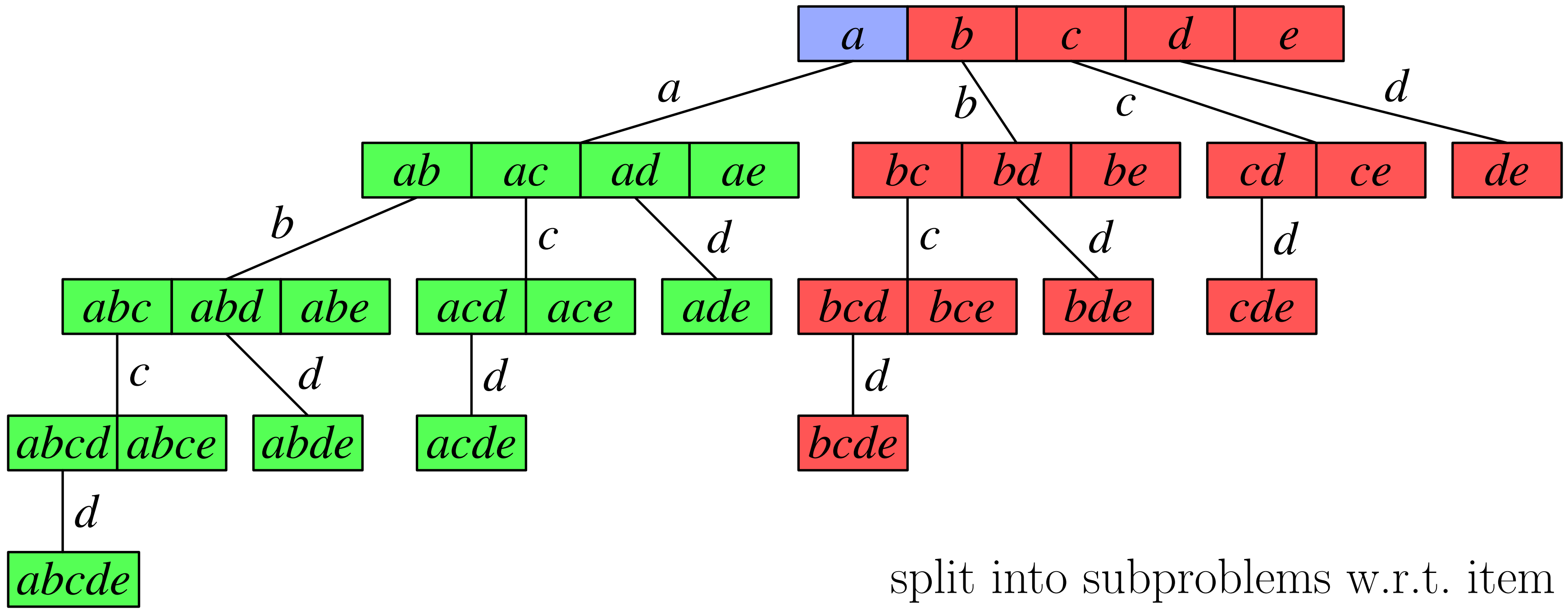
- General search procedure:
  - Let the item order be $a < b < c < \cdots$.
  - Restrict the transaction database to those transactions that contain a.
    This is the **conditional database for the prefix** a.

    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

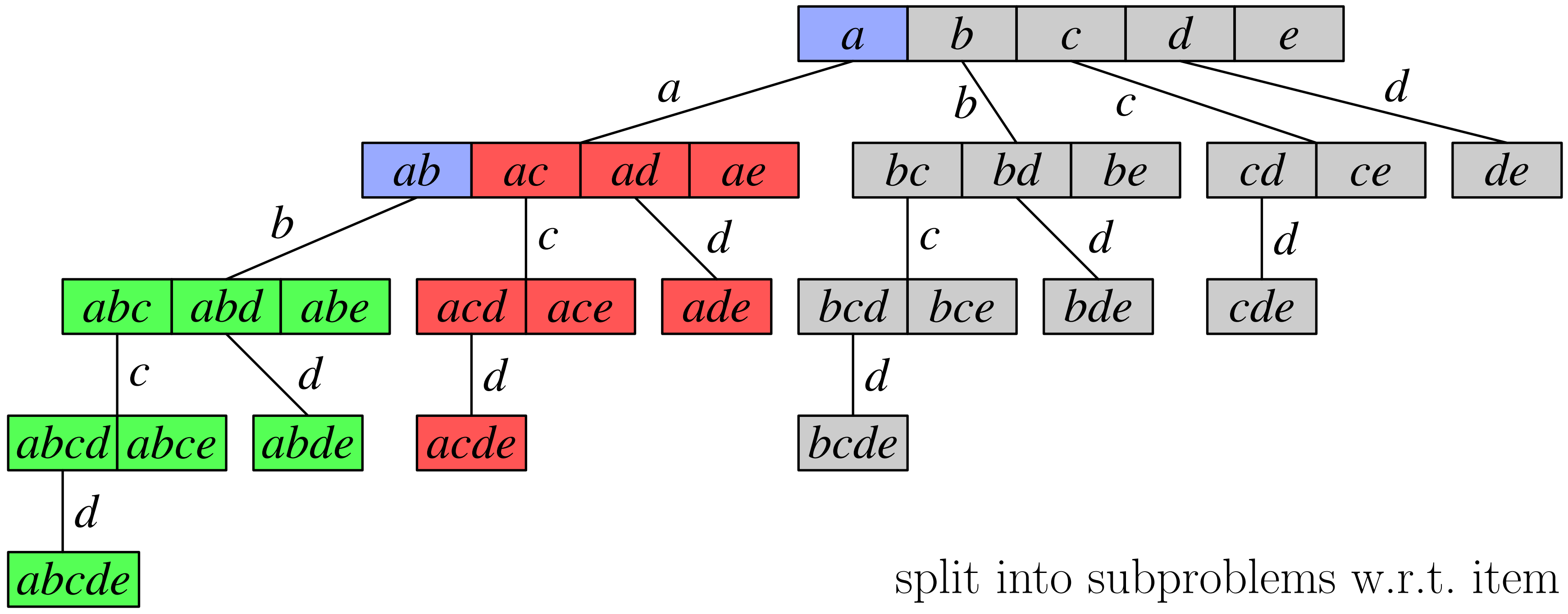


split into subproblems w.r.t. item a

- 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

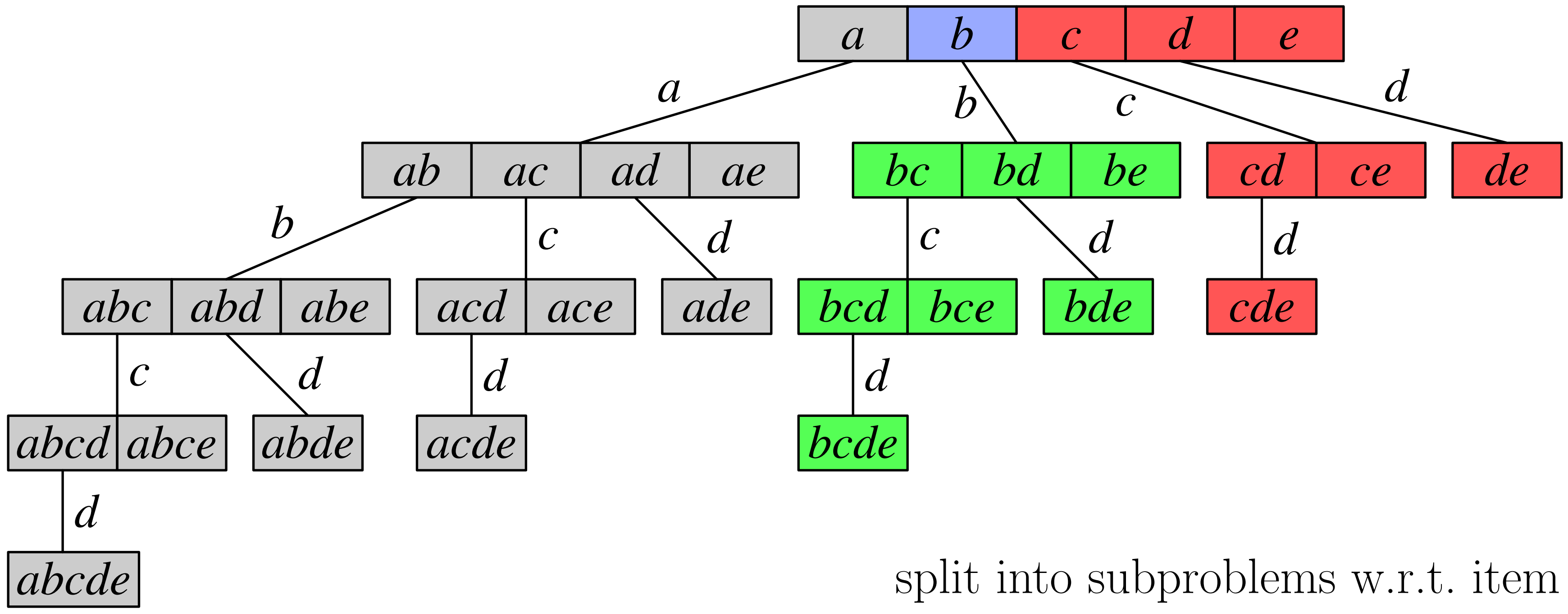


split into subproblems w.r.t. item b

- blue : item sets $\{a\}$ and $\{a, b\}$.
  green: item sets containing 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 not item a.
  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 $S = (D, P)$, where D 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

$$\begin{tikzcd}
 & & & (T, \emptyset) \arrow[dll, "a"'] \arrow[drr, "\bar{a}"] & & & & \\
 & (T_a, \{a\}) \arrow[dl, "b"'] \arrow[dr, "\bar{b}"] & & & & (T_{\bar{a}}, \emptyset) \arrow[dl, "b"'] \arrow[dr, "\bar{b}"] & & \\
(T_{ab}, \{a, b\}) \arrow[d, "c"'] \arrow[dr, "\bar{c}"] & & (T_{a\bar{b}}, \{a\}) \arrow[d, "c"'] \arrow[dr, "\bar{c}"] & & (T_{\bar{a}b}, \{b\}) \arrow[d, "c"'] \arrow[dr, "\bar{c}"] & & (T_{\bar{a}\bar{b}}, \emptyset) \arrow[d, "c"'] \arrow[dr, "\bar{c}"] & \\
(T_{abc}, \{a, b, c\}) & (T_{ab\bar{c}}, \{a, b\}) & (T_{a\bar{b}c}, \{a, c\}) & (T_{a\bar{b}\bar{c}}, \{a\}) & (T_{\bar{a}bc}, \{b, c\}) & (T_{\bar{a}b\bar{c}}, \{b\}) & (T_{\bar{a}\bar{b}c}, \{c\}) & (T_{\bar{a}\bar{b}\bar{c}}, \emptyset)
\end{tikzcd}$$

- 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 $\{b\}$ 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.
- There are no other perfect extensions in this example for a minimum support of $s_{\text{min}} = 3$.

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 database.

- Frequent item set mining algorithms also differ in how they construct new conditional 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 list** is created.
  - The transaction list of item a indicates the transactions that contain it, that is, it represents its **cover** $K_T(\{a\})$.
  - 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

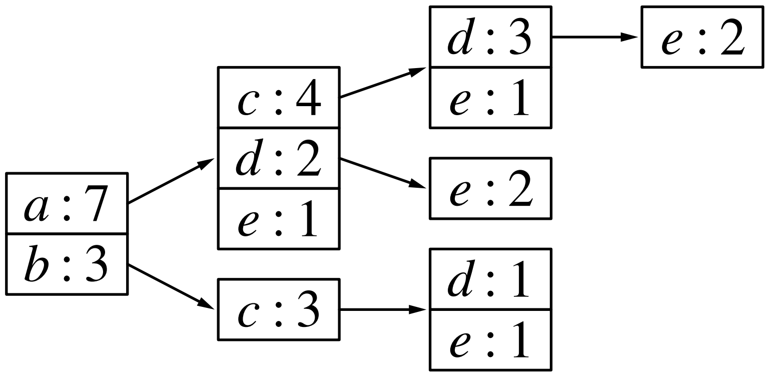
	a	b	c	d	e
1:	**1**	0	0	**1**	**1**
2:	0	**1**	**1**	**1**	0
3:	**1**	0	**1**	0	**1**
4:	**1**	0	**1**	**1**	**1**
5:	**1**	0	0	0	**1**
6:	**1**	0	**1**	**1**	0
7:	0	**1**	**1**	0	0
8:	**1**	0	**1**	**1**	**1**
9:	0	**1**	**1**	0	**1**
10:	**1**	0	0	**1**	**1**

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

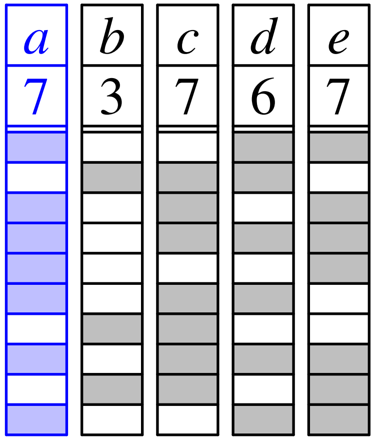
a	*b*	*c*	*d*	*e*
7	3	7	6	7
1	2	2	1	1
3	7	3	2	3
4	9	4	4	4
5		6	6	5
6		7	8	8
8		8	10	9
10		9		10

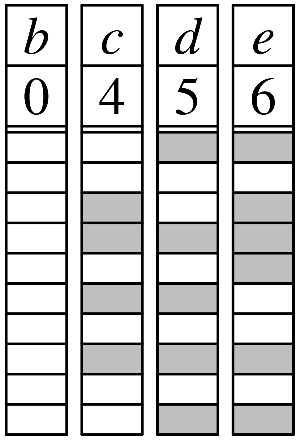
b	*c*	*d*	*e*
0	4	5	6
	3	1	1
	4	4	3
	6	6	4
	8	8	5
		10	8
			10

↑ Conditional database for prefix a (1st subproblem)

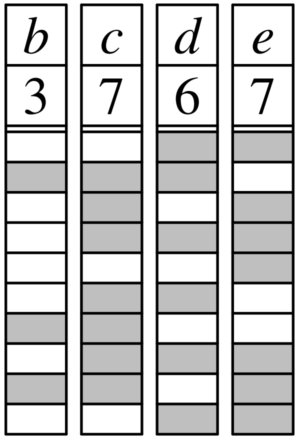
b	*c*	*d*	*e*
3	7	6	7
2	2	1	1
7	3	2	3
9	4	4	4
	6	6	5
	7	8	8
	8	10	9
	9		10

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





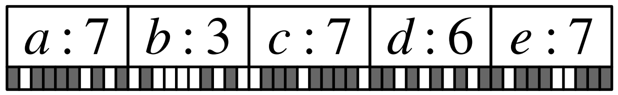
↑ Conditional database for prefix a (1st subproblem)



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

Eclat: Depth-First Search

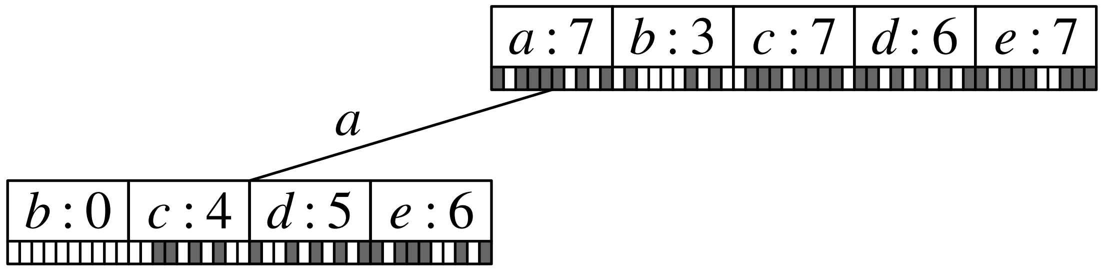
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



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

Eclat: Depth-First Search

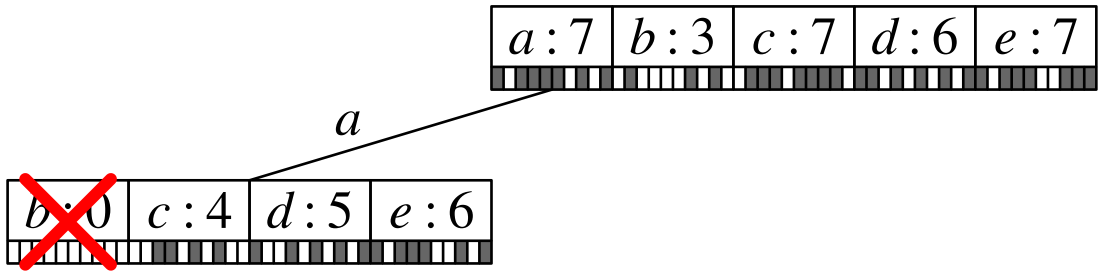
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- Intersect the transaction list for item a with the transaction lists of all other items (*conditional database* for item a).
- Count the number of bits that are set (number of containing transactions). This yields the support of all item sets with the prefix a.

Eclat: Depth-First Search

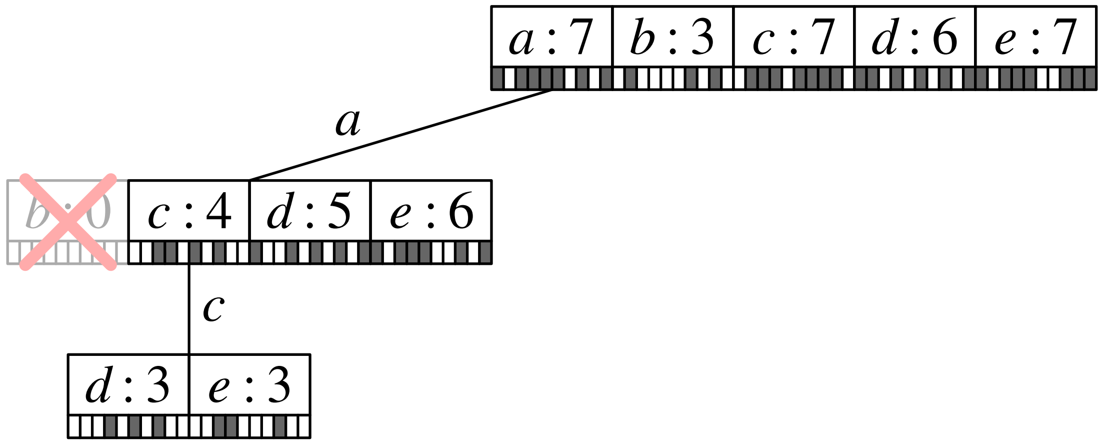
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



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

Eclat: Depth-First Search

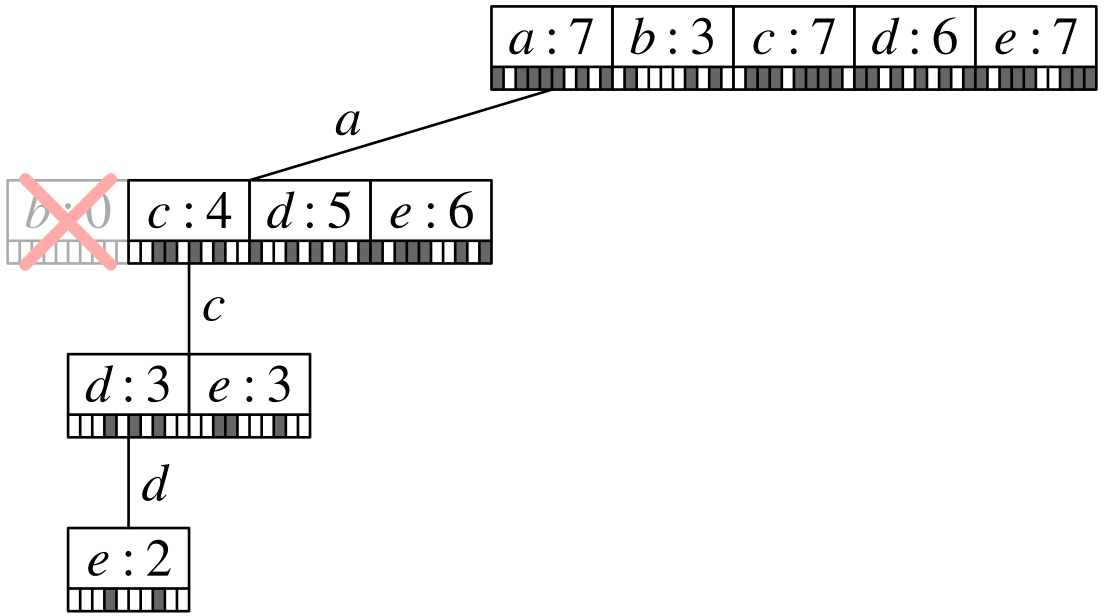
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



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

Eclat: Depth-First Search

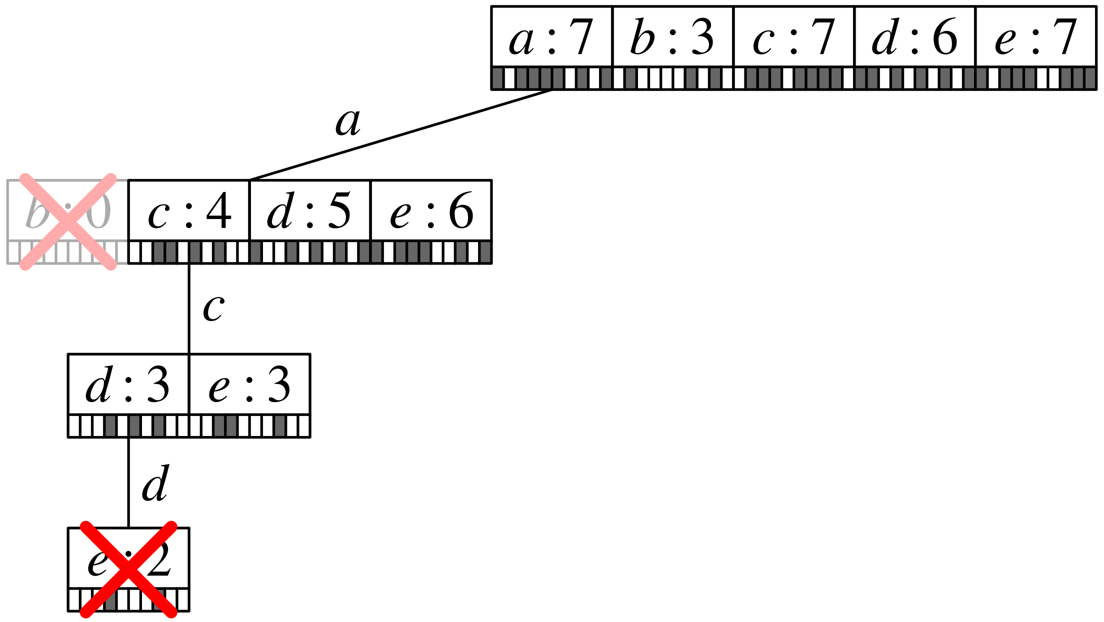
1:	$\{a, d, e\}$
2:	$\{b, c, d\}$
3:	$\{a, c, e\}$
4:	$\{a, c, d, e\}$
5:	$\{a, e\}$
6:	$\{a, c, d\}$
7:	$\{b, c\}$
8:	$\{a, c, d, e\}$
9:	$\{b, c, e\}$
10:	$\{a, d, e\}$



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

Eclat: Depth-First Search

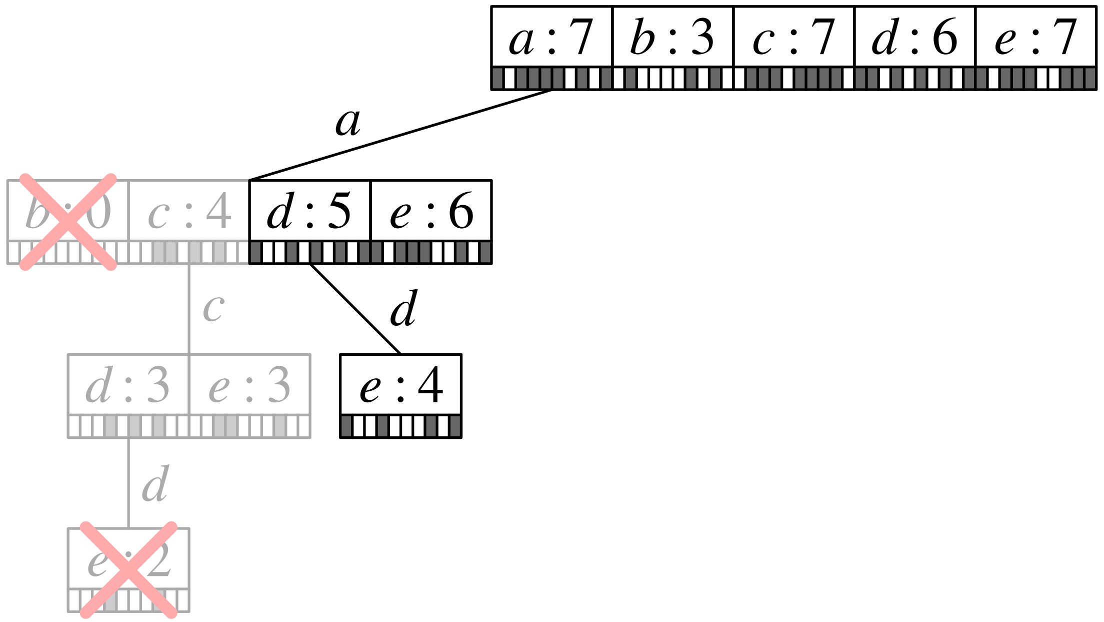
1:	$\{a, d, e\}$
2:	$\{b, c, d\}$
3:	$\{a, c, e\}$
4:	$\{a, c, d, e\}$
5:	$\{a, e\}$
6:	$\{a, c, d\}$
7:	$\{b, c\}$
8:	$\{a, c, d, e\}$
9:	$\{b, c, e\}$
10:	$\{a, d, e\}$



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

Eclat: Depth-First Search

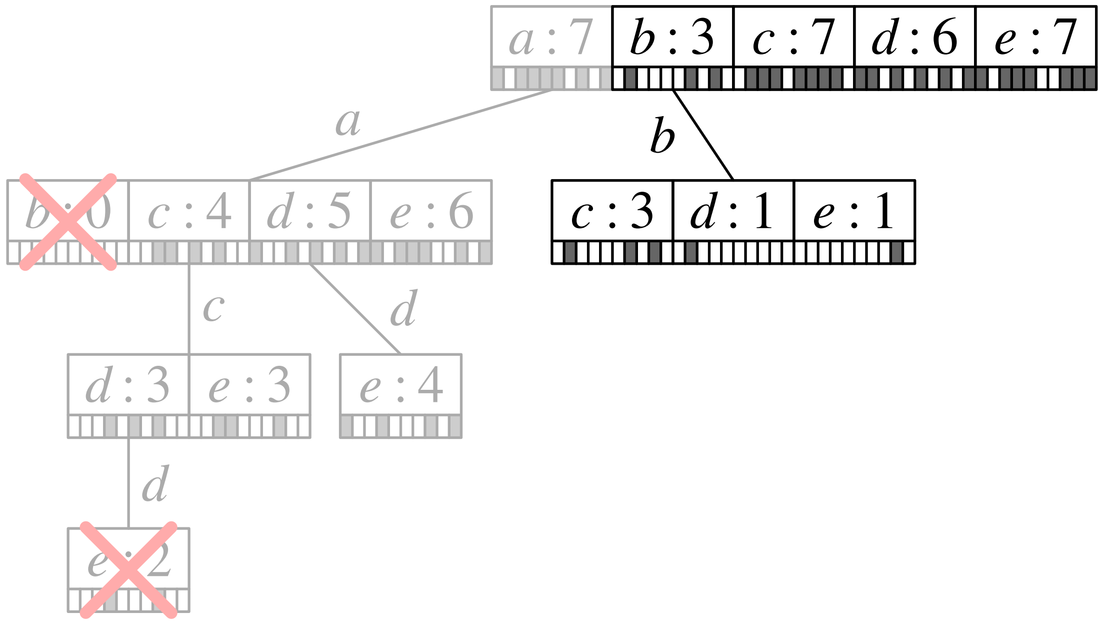
1:	$\{a, d, e\}$
2:	$\{b, c, d\}$
3:	$\{a, c, e\}$
4:	$\{a, c, d, e\}$
5:	$\{a, e\}$
6:	$\{a, c, d\}$
7:	$\{b, c\}$
8:	$\{a, c, d, e\}$
9:	$\{b, c, e\}$
10:	$\{a, d, e\}$



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

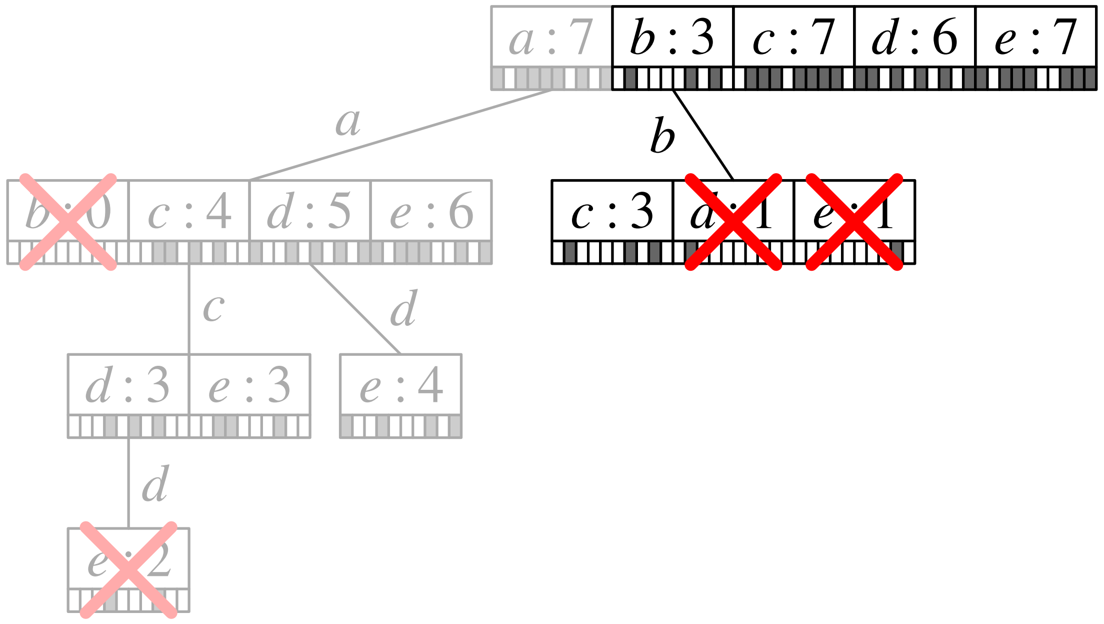
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

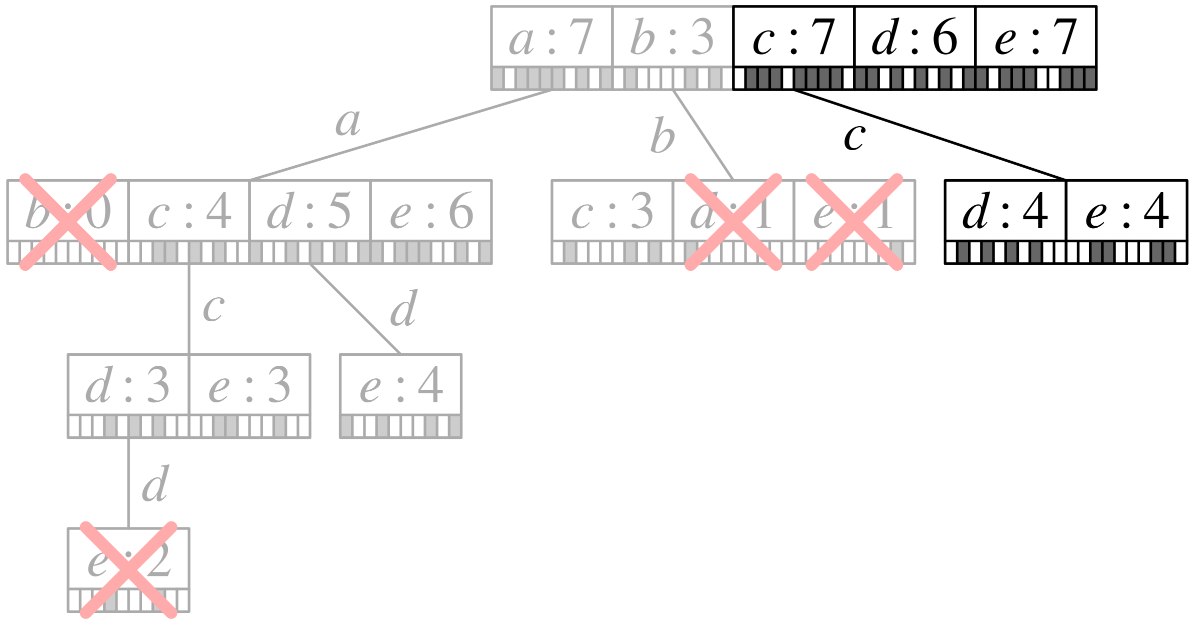
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



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

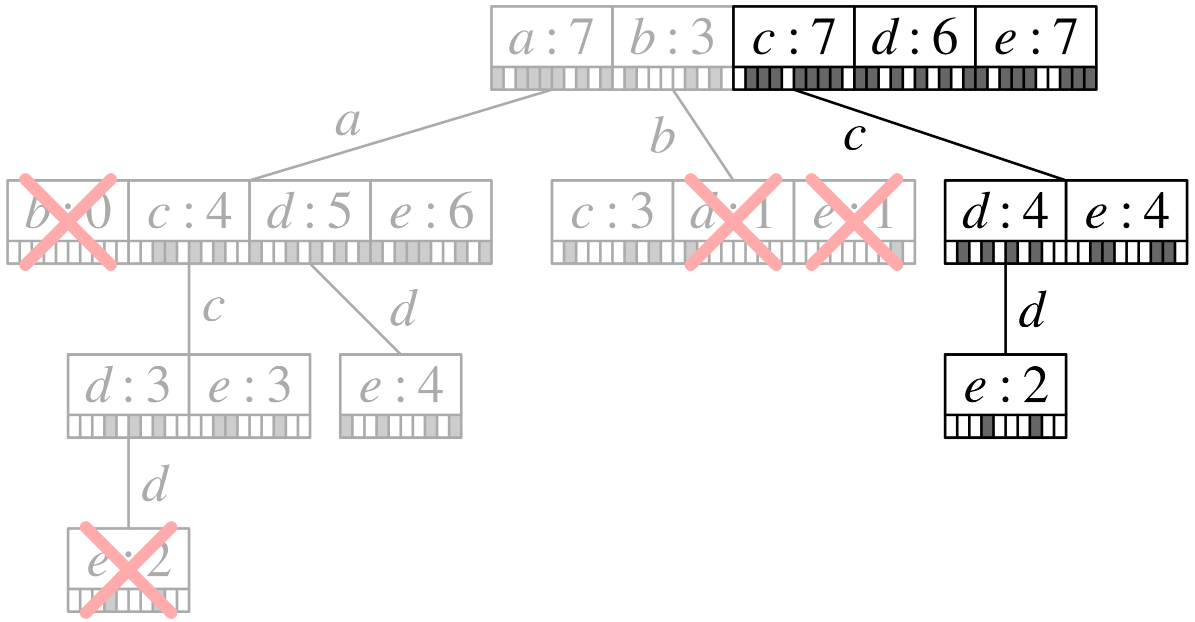


- Backtrack to the first level of the search tree and intersect the transaction list for c with the transaction lists for d and e.

- Result: Transaction lists for the item sets $\{c, d\}$ and $\{c, e\}$.

Eclat: Depth-First Search

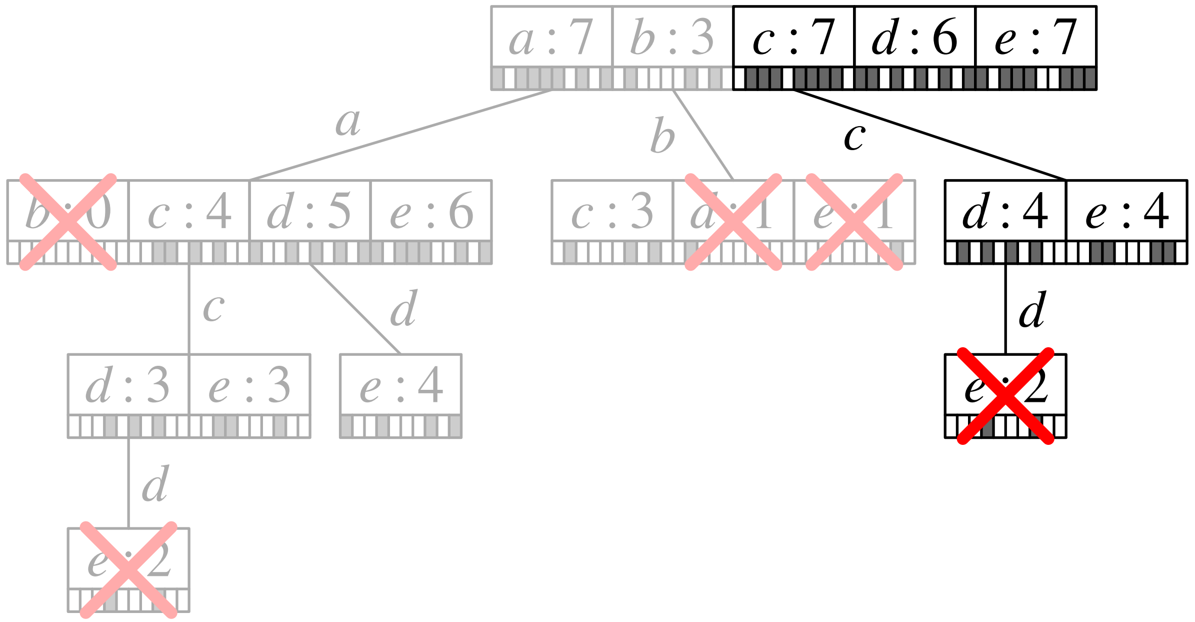
1:	$\{a, d, e\}$
2:	$\{b, c, d\}$
3:	$\{a, c, e\}$
4:	$\{a, c, d, e\}$
5:	$\{a, e\}$
6:	$\{a, c, d\}$
7:	$\{b, c\}$
8:	$\{a, c, d, e\}$
9:	$\{b, c, e\}$
10:	$\{a, d, e\}$



- 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

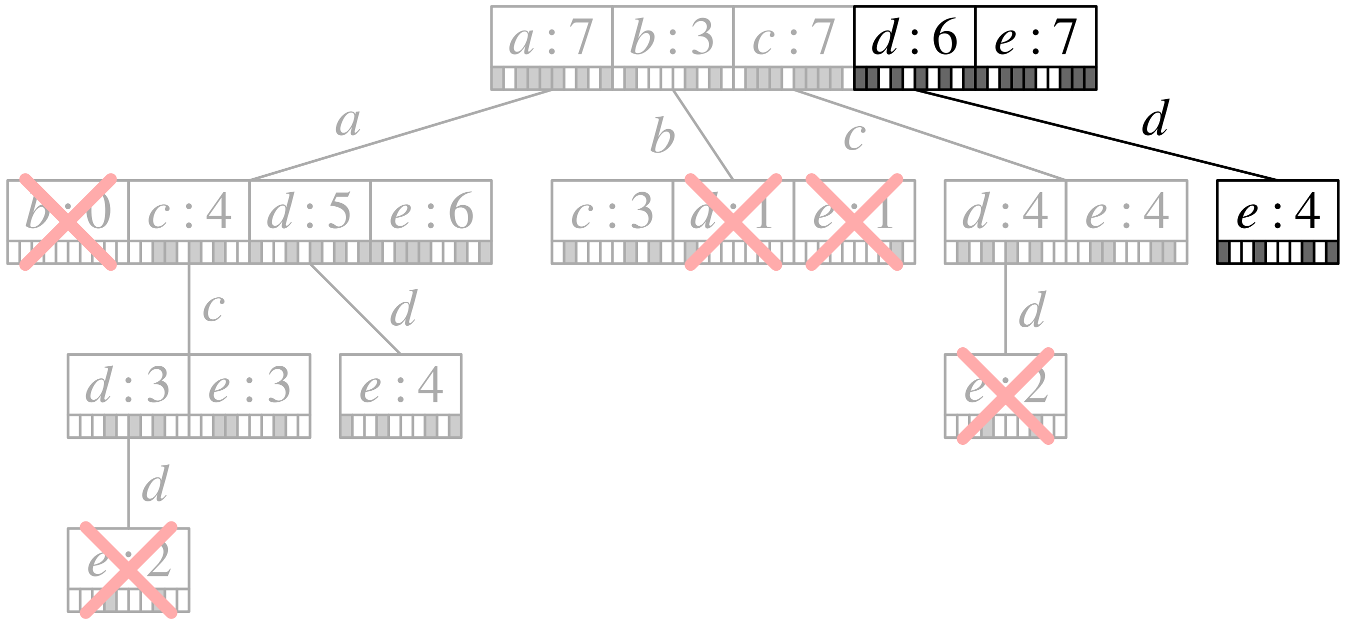
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



- 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

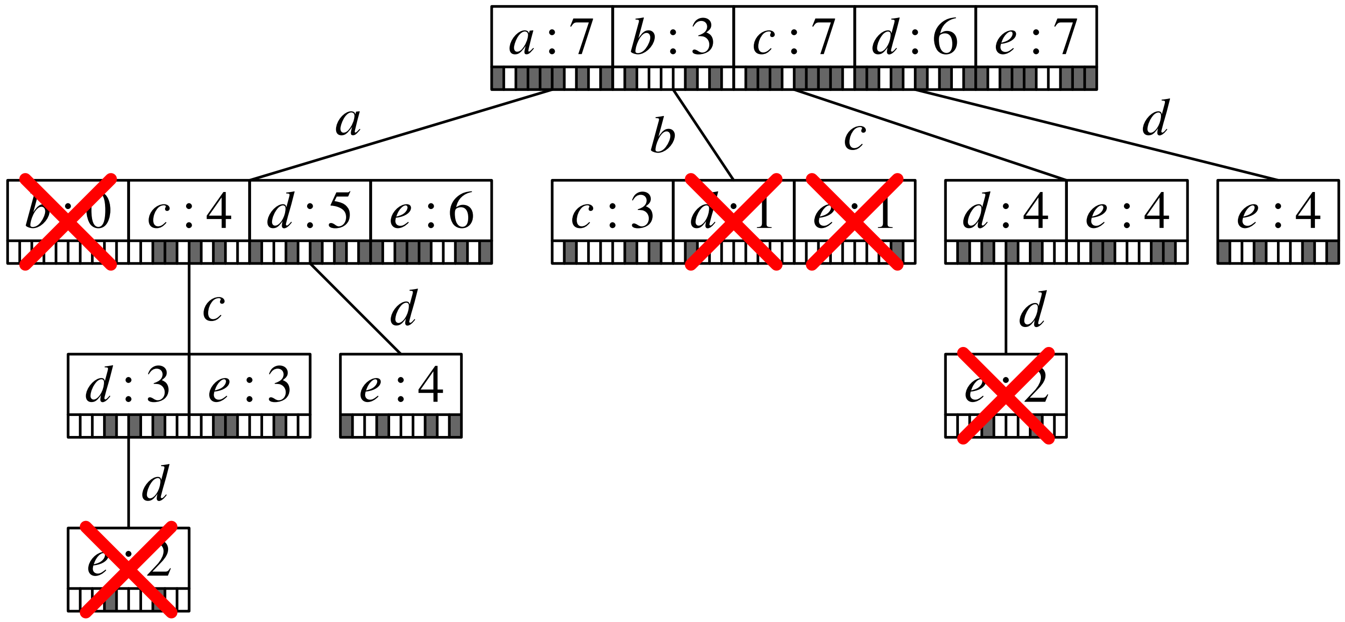
1: $\{a, d, e\}$
2: $\{b, c, d\}$
3: $\{a, c, e\}$
4: $\{a, c, d, e\}$
5: $\{a, e\}$
6: $\{a, c, d\}$
7: $\{b, c\}$
8: $\{a, c, d, e\}$
9: $\{b, c, e\}$
10: $\{a, d, e\}$



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

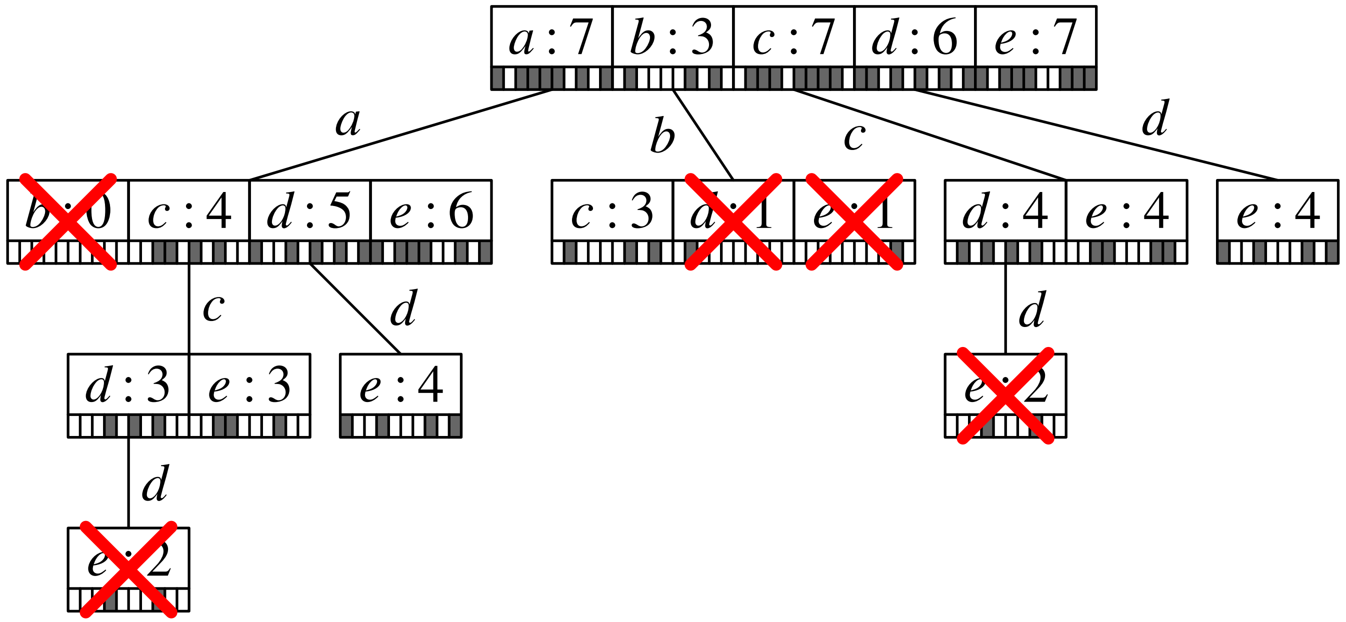


- The found frequent item sets coincide, of course, with those found by the Apriori algorithm.

- 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.

Summary Eclat

Basic Processing Scheme

- Depth-first traversal of the prefix tree.
- 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)
  Eclat is difficult to execute for modern processors (branch prediction).

Software

- `http://www.borgelt.net/eclat.html`

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 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).

  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: Programs and Test System

- All programs are my own implementations.

  All use the same code for reading the transaction database
  and for writing the found frequent item sets.

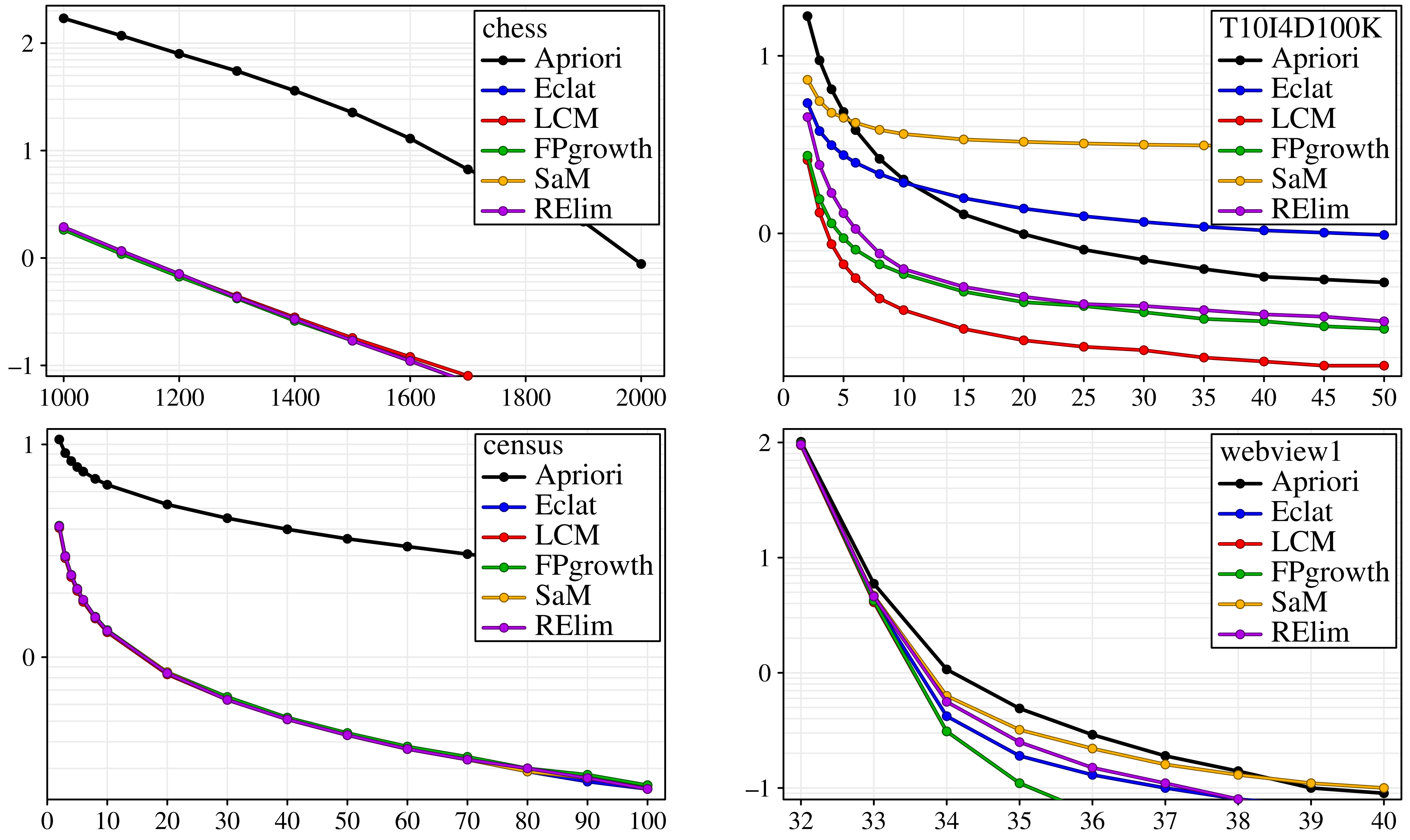
  Therefore differences in speed can only be the effect of the processing schemes.

- These programs and their source code can be found on my web site:
  `http://www.borgelt.net/fpm.html`

○ Apriori	`http://www.borgelt.net/apriori.html`
○ Eclat & LCM	`http://www.borgelt.net/eclat.html`
○ FP-Growth	`http://www.borgelt.net/fpgrowth.html`
○ RElim	`http://www.borgelt.net/relim.html`
○ SaM	`http://www.borgelt.net/sam.html`

- All tests were run on an Intel Core2 Quad Q9650@3GHz with 8GB memory
  running Ubuntu Linux 14.04 LTS (64 bit);
  programs were compiled with GCC 4.8.2.

Experiments: Execution Times



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

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.

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** (⇒ 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.

Association Rules

Association Rules: Basic Notions

- 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**.

- Formally, we consider rules of the form $X \to Y$,
  with $X, Y \subseteq B$ and $X \cap Y = \emptyset$.

- **Support of a Rule** $X \to Y$:

  Either: $\varsigma_T(X \to Y) = \sigma_T(X \cup Y)$ (more common: rule is correct)

  Or: $\varsigma_T(X \to Y) = \sigma_T(X)$ (more plausible: rule is applicable)

- **Confidence of a Rule** $X \to Y$:

$$c_T(X \to Y) = \frac{\sigma_T(X \cup Y)}{\sigma_T(X)} = \frac{s_T(X \cup Y)}{s_T(X)} = \frac{s_T(I)}{s_T(X)}$$

  The confidence can be seen as an estimate of $P(Y \mid X)$.

Association Rules: Formal Definition

Given:

- a set $B = \{i_1, \dots, i_m\}$ of items,
- a tuple $T = (t_1, \dots, t_n)$ of transactions over B,
- a real number $\varsigma_{\min}$, $0 < \varsigma_{\min} \leq 1$, the **minimum support**,
- a real number $c_{\min}$, $0 < c_{\min} \leq 1$, the **minimum confidence**.

Desired:

- the set of all **association rules**, that is, the set

$$\mathcal{R} = \{R : X \to Y \mid \varsigma_T(R) \geq \varsigma_{\min} \wedge c_T(R) \geq c_{\min}\}.$$

General Procedure:

- Find the frequent item sets.
- Construct rules and filter them w.r.t. $\varsigma_{\min}$ and $c_{\min}$.

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:	$\{c, b, e\}$
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 minimum support is $s_{\min} = 3$ or $\sigma_{\min} = 0.3 = 30\%$ in this example.
- 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).

Generating Association Rules

Example: $I = \{a, c, e\}$, $X = \{c, e\}$, $Y = \{a\}$.

$$c_T(c, e \to a) = \frac{s_T(\{a, c, e\})}{s_T(\{c, e\})} = \frac{3}{4} = 75\%$$

Minimum confidence: 80%

association rule	support of all items	support of antecedent	confidence
$b \to c$	3 (30%)	3 (30%)	100%
$d \to a$	5 (50%)	6 (60%)	83.3%
$e \to a$	6 (60%)	7 (70%)	85.7%
$a \to e$	6 (60%)	7 (70%)	85.7%
$d, e \to a$	4 (40%)	4 (40%)	100%
$a, d \to e$	4 (40%)	5 (50%)	80%

Support of an Association Rule

The two rule support definitions are not equivalent:

transaction database

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

two association rules

association rule	support of all items	support of antecedent	confidence
$a \to c$	3 (37.5%)	5 (62.5%)	60.0%
$b \to d$	4 (50.0%)	4 (50.0%)	100.0%

Let the minimum confidence be $c_{\min} = 60\%$.

- For $\varsigma_T(R) = \sigma(X \cup Y)$ and $3 < \varsigma_{\min} \leq 4$ only the rule $b \to d$ is generated, but not the rule $a \to c$.

- For $\varsigma_T(R) = \sigma(X)$ there is no value $\varsigma_{\min}$ that generates only the rule $b \to d$, but not at the same time also the rule $a \to c$.

Rules with Multiple Items in the Consequent?

- The general definition of association rules $X \to Y$ allows for multiple items in the consequent (i.e. $|Y| \geq 1$).

- However: If $a \to b, c$ is an association rule, then $a \to b$ and $a \to c$ are also association rules.

  Because: (regardless of the rule support definition)

  $$\begin{aligned} \varsigma_T(a \to b) &\geq \varsigma_T(a \to b, c), \qquad & c_T(a \to b) &\geq c_T(a \to b, c), \\ \varsigma_T(a \to c) &\geq \varsigma_T(a \to b, c), \qquad & c_T(a \to c) &\geq c_T(a \to b, c). \end{aligned}$$

- The two simpler rules are often sufficient (e.g. for product suggestions), even though they contain less information.

  - $a \to b, c$ provides information about the *joint* conditional occurence of b and c (condition a).

  - $a \to b$ and $a \to c$ only provide information about the *individual* conditional occurrences of b and c (condition a).

  In most applications this additional information does not yield any additional benefit.

Additional Rule Filtering: Simple Measures

- General idea: Compare $\hat{P}_T(Y \mid X) = c_T(X \to Y)$ and $\hat{P}_T(Y) = c_T(\emptyset \to Y) = \sigma_T(Y)$.

- (Absolute) confidence difference to prior:

$$d_T(R) = |c_T(X \to Y) - \sigma_T(Y)|$$

- Lift value:

$$l_T(R) = \frac{c_T(X \to Y)}{\sigma_T(Y)}$$

- (Absolute) difference of lift value to 1:

$$q_T(R) = \left| \frac{c_T(X \to Y)}{\sigma_T(Y)} - 1 \right|$$

- (Absolute) difference of lift quotient to 1:

$$r_T(R) = \left| 1 - \min \left\{ \frac{c_T(X \to Y)}{\sigma_T(Y)}, \frac{\sigma_T(Y)}{c_T(X \to Y)} \right\} \right|$$

Additional Rule Filtering: More Sophisticated Measures

- Consider the 2×2 contingency table or the estimated probability table:

	$X \not\subseteq t$	$X \subseteq t$	
$Y \not\subseteq t$	n_{00}	n_{01}	$n_{0.}$
$Y \subseteq t$	n_{10}	n_{11}	$n_{1.}$
	$n_{.0}$	$n_{.1}$	$n_{..}$

	$X \not\subseteq t$	$X \subseteq t$	
$Y \not\subseteq t$	p_{00}	p_{01}	$p_{0.}$
$Y \subseteq t$	p_{10}	p_{11}	$p_{1.}$
	$p_{.0}$	$p_{.1}$	1

- $n_{..}$ is the total number of transactions.
  $n_{.1}$ is the number of transactions to which the rule is applicable.
  n_{11} is the number of transactions for which the rule is correct.

  It is $\quad p_{ij} = \frac{n_{ij}}{n_{..}}, \quad p_{i.} = \frac{n_{i.}}{n_{..}}, \quad p_{.j} = \frac{n_{.j}}{n_{..}} \quad$ for $i, j = 1, 2$.

- General idea: Use measures for the strength of dependence of X and Y.

- There is a large number of such measures of dependence
  originating from statistics, decision tree induction etc.

An Information-theoretic Evaluation Measure

Information Gain (Kullback and Leibler 1951, Quinlan 1986)

Based on Shannon Entropy $H = -\sum_{i=1}^{n} p_i \log_2 p_i$ (Shannon 1948)

$$\begin{aligned} I_{\text{gain}}(X,Y) &= \overbrace{H(Y)} \quad - \quad \overbrace{H(Y|X)} \\ &= -\sum_{i=1}^{k_Y} p_{i.} \log_2 p_{i.} - \sum_{j=1}^{k_X} p_{.j} \left(-\sum_{i=1}^{k_Y} p_{i|j} \log_2 p_{i|j} \right) \end{aligned}$$

$H(Y)$	Entropy of the distribution of Y
$H(Y\|X)$	*Expected entropy* of the distribution of Y if the value of the X becomes known
$H(Y) - H(Y\|X)$	Expected entropy reduction or *information gain*

Interpretation of Shannon Entropy

- Let $S = \{s_1, \ldots, s_n\}$ be a finite set of alternatives having positive probabilities $P(s_i)$, $i = 1, \ldots, n$, satisfying $\sum_{i=1}^{n} P(s_i) = 1$.

- **Shannon Entropy:**

$$H(S) = -\sum_{i=1}^{n} P(s_i) \log_2 P(s_i)$$

- Intuitively: **Expected number of yes/no questions that have to be asked in order to determine the obtaining alternative.**
  - Suppose there is an oracle, which knows the obtaining alternative, but responds only if the question can be answered with "yes" or "no".
  - A better question scheme than asking for one alternative after the other can easily be found: Divide the set into two subsets of about equal size.
  - Ask for containment in an arbitrarily chosen subset.
  - Apply this scheme recursively $\rightarrow$ number of questions bounded by $\lceil \log_2 n \rceil$.

A Statistical Evaluation Measure

χ^2 Measure

- Compares the actual joint distribution
  with a **hypothetical independent distribution**.

- Uses absolute comparison.

- Can be interpreted as a difference measure.

$$\chi^2(X,Y) = \sum_{i=1}^{k_X} \sum_{j=1}^{k_Y} n_{..} \frac{(p_{i.}p_{.j} - p_{ij})^2}{p_{i.}p_{.j}}$$

- Side remark: Information gain can also be interpreted as a difference measure.

$$I_{\text{gain}}(X,Y) = \sum_{j=1}^{k_X} \sum_{i=1}^{k_Y} p_{ij} \log_2 \frac{p_{ij}}{p_{i.}p_{.j}}$$

A Statistical Evaluation Measure

χ^2 **Measure**

- Compares the actual joint distribution
  with a **hypothetical independent distribution**.

- Uses absolute comparison.

- Can be interpreted as a difference measure.

$$\chi^2(X,Y) = \sum_{i=1}^{k_X} \sum_{j=1}^{k_Y} n_{..} \frac{(p_{i.}p_{.j} - p_{ij})^2}{p_{i.}p_{.j}}$$

- For $k_X = k_Y = 2$ (as for rule evaluation) the χ^2 measure simplifies to

$$\chi^2(X,Y) \;=\; n_{..}\frac{(p_{1.}\,p_{.1} - p_{11})^2}{p_{1.}(1-p_{1.})p_{.1}(1-p_{.1})} \;=\; n_{..}\frac{(n_{1.}n_{.1} - n_{..}n_{11})^2}{n_{1.}(n_{..}-n_{1.})n_{.1}(n_{..}-n_{.1})}.$$

Examples from the Census Data

All rules are stated as

```
consequent <- antecedent (support%, confidence%, lift)
```

where the support of a rule is the support of the antecedent.

Trivial/Obvious Rules

```
edu_num=13 <- education=Bachelors  (16.4, 100.0, 6.09)
sex=Male <- relationship=Husband  (40.4, 99.99, 1.50)
sex=Female <- relationship=Wife  (4.8, 99.9, 3.01)
```

Interesting Comparisons

```
marital=Never-married <- age=young sex=Female  (12.3, 80.8, 2.45)
marital=Never-married <- age=young sex=Male  (17.4, 69.9, 2.12)

salary>50K <- occupation=Exec-managerial sex=Male  (8.9, 57.3, 2.40)
salary>50K <- occupation=Exec-managerial  (12.5, 47.8, 2.00)

salary>50K <- education=Masters  (5.4, 54.9, 2.29)
hours=overtime <- education=Masters  (5.4, 41.0, 1.58)
```

Examples from the Census Data

```
salary>50K <- education=Masters  (5.4, 54.9, 2.29)
salary>50K <- occupation=Exec-managerial  (12.5, 47.8, 2.00)
salary>50K <- relationship=Wife  (4.8, 46.9, 1.96)
salary>50K <- occupation=Prof-specialty  (12.6, 45.1, 1.89)
salary>50K <- relationship=Husband  (40.4, 44.9, 1.88)
salary>50K <- marital=Married-civ-spouse  (45.8, 44.6, 1.86)
salary>50K <- education=Bachelors  (16.4, 41.3, 1.73)
salary>50K <- hours=overtime  (26.0, 40.6, 1.70)

salary>50K <- occupation=Exec-managerial hours=overtime
              (5.5, 60.1, 2.51)
salary>50K <- occupation=Prof-specialty hours=overtime
              (4.4, 57.3, 2.39)
salary>50K <- education=Bachelors hours=overtime
              (6.0, 54.8, 2.29)
```

Examples from the Census Data

```
salary>50K <- occupation=Prof-specialty marital=Married-civ-spouse
              (6.5, 70.8, 2.96)
salary>50K <- occupation=Exec-managerial marital=Married-civ-spouse
              (7.4, 68.1, 2.85)
salary>50K <- education=Bachelors marital=Married-civ-spouse
              (8.5, 67.2, 2.81)
salary>50K <- hours=overtime marital=Married-civ-spouse
              (15.6, 56.4, 2.36)

marital=Married-civ-spouse <- salary>50K  (23.9, 85.4, 1.86)
```

Examples from the Census Data

```
hours=half-time <- occupation=Other-service age=young
                   (4.4, 37.2, 3.08)

hours=overtime <- salary>50K  (23.9, 44.0, 1.70)
hours=overtime <- occupation=Exec-managerial  (12.5, 43.8, 1.69)
hours=overtime <- occupation=Exec-managerial salary>50K
                  (6.0, 55.1, 2.12)
hours=overtime <- education=Masters  (5.4, 40.9, 1.58)

education=Bachelors <- occupation=Prof-specialty  (12.6, 36.2, 2.20)
education=Bachelors <- occupation=Exec-managerial  (12.5, 33.3, 2.03)
education=HS-grad <- occupation=Transport-moving  (4.8, 51.9, 1.61)
education=HS-grad <- occupation=Machine-op-inspct  (6.2, 50.7, 1.6)
```

Examples from the Census Data

```
occupation=Prof-specialty <- education=Masters  (5.4, 49.0, 3.88)
occupation=Prof-specialty <- education=Bachelors sex=Female
                             (5.1, 34.7, 2.74)
occupation=Adm-clerical <- education=Some-college sex=Female
                           (8.6, 31.1, 2.71)

sex=Female <- occupation=Adm-clerical  (11.5, 67.2, 2.03)
sex=Female <- occupation=Other-service  (10.1, 54.8, 1.65)
sex=Female <- hours=half-time  (12.1, 53.7, 1.62)

age=young <- hours=half-time  (12.1, 53.3, 1.79)
age=young <- occupation=Handlers-cleaners  (4.2, 50.6, 1.70)
age=senior <- workclass=Self-emp-not-inc  (7.9, 31.1, 1.57)
```

Summary Association Rules

- **Association Rule Induction is a Two Step Process**
  - Find the frequent item sets (minimum support).
  - Form the relevant association rules (minimum confidence).
- **Generating the Association Rules**
  - Form all possible association rules from the frequent item sets.
  - Filter "interesting" association rules
    based on minimum support and minimum confidence.
- **Filtering the Association Rules**
  - Compare rule confidence and consequent support.
  - Information gain, χ^2 measure
  - In principle: other measures used for decision tree induction.

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.

Software

Software for frequent pattern mining can be found at

- my web site: `http://www.borgelt.net/fpm.html`

◦ Apriori	`http://www.borgelt.net/apriori.html`
◦ Eclat	`http://www.borgelt.net/eclat.html`
◦ FP-Growth	`http://www.borgelt.net/fpgrowth.html`
◦ RElim	`http://www.borgelt.net/relim.html`
◦ SaM	`http://www.borgelt.net/sam.html`
◦ MoSS	`http://www.borgelt.net/moss.html`

- the Frequent Item Set Mining Implementations (FIMI) Repository
  `http://fimi.cs.helsinki.fi/`

  This repository was set up with the contributions to the FIMI workshops in 2003 and 2004, where each submission had to be accompanied by the source code of an implementation. The web site offers all source code, several data sets, and the results of the competition.