Mining Association Rules in Large Databases

Association rules

- Given a set of transactions **D**, find rules that will predict the occurrence of an item (or a set of items) based on the occurrences of other items in the transaction

Market-Basket transactions

TID	*Items*
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Examples of association rules

{Diaper} $\rightarrow$ {Beer},
{Milk, Bread} $\rightarrow$ {Diaper,Coke},
{Beer, Bread} $\rightarrow$ {Milk},

An even simpler concept: frequent itemsets

- Given a set of transactions **D**, find combination of items that occur frequently

Market-Basket transactions

TID	*Items*
1	**Bread, Milk**
2	**Bread, Diaper, Beer, Eggs**
3	**Milk, Diaper, Beer, Coke**
4	**Bread, Milk, Diaper, Beer**
5	**Bread, Milk, Diaper, Coke**

Examples of frequent itemsets

{Diaper, Beer},
{Milk, Bread}
{Beer, Bread, Milk},

Lecture outline

- **Task 1:** Methods for finding all frequent itemsets efficiently
- **Task 2:** Methods for finding association rules efficiently

Definition: Frequent Itemset

- **Itemset**
  - A set of one or more items
    - E.g.: {Milk, Bread, Diaper}
  - k-itemset
    - An itemset that contains k items
- **Support count (σ)**
  - Frequency of occurrence of an itemset (number of transactions it appears)
  - E.g. **σ({Milk, Bread,Diaper}) = 2**
- **Support**
  - Fraction of the transactions in which an itemset appears
  - E.g. **s({Milk, Bread, Diaper}) = 2/5**
- **Frequent Itemset**
  - An itemset whose support is greater than or equal to a ***minsup*** threshold

TID	*Items*
1	**Bread, Milk**
2	**Bread, Diaper, Beer, Eggs**
3	**Milk, Diaper, Beer, Coke**
4	**Bread, Milk, Diaper, Beer**
5	**Bread, Milk, Diaper, Coke**

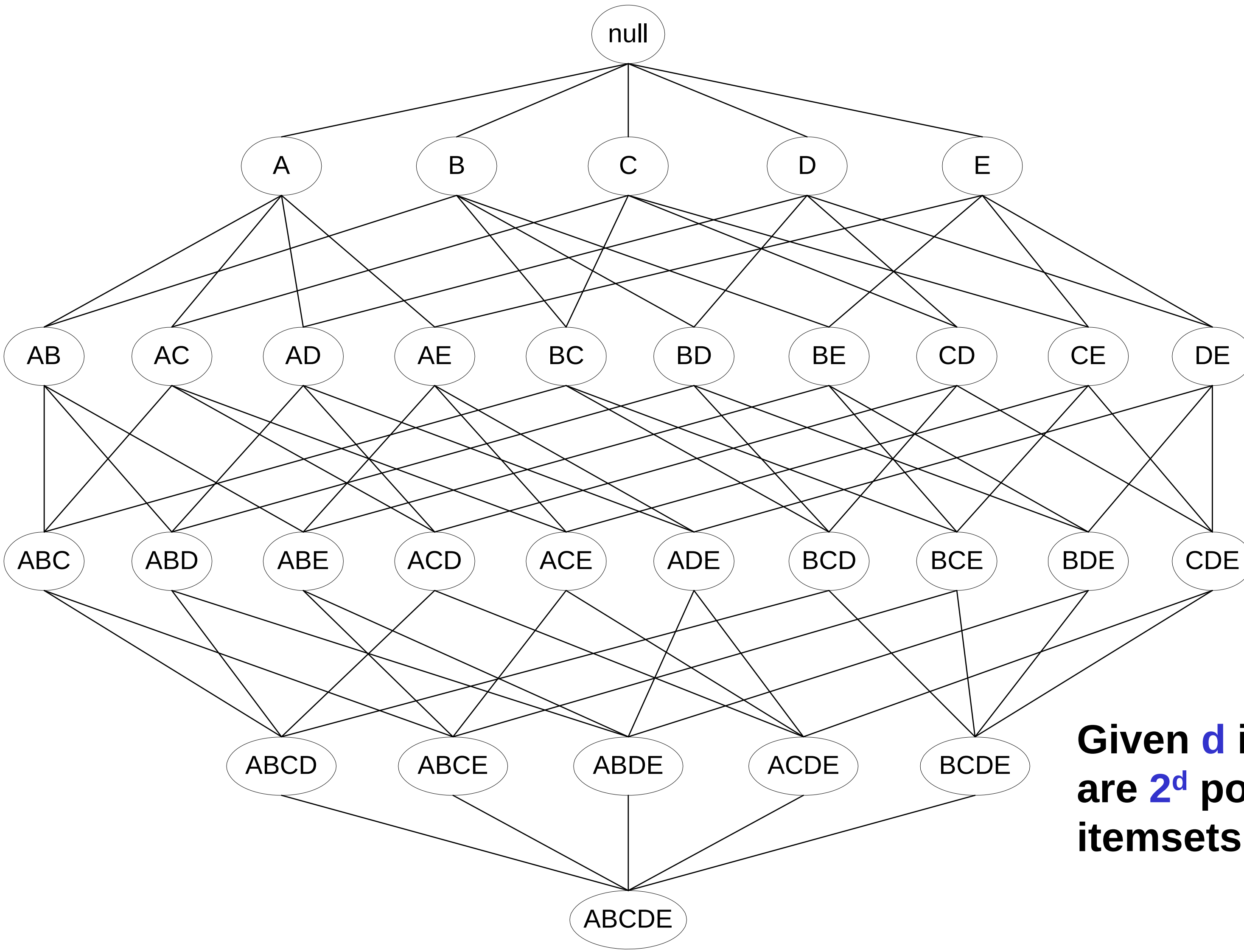
Why do we want to find frequent itemsets?

- Find all combinations of items that occur together
- They might be interesting (e.g., in placement of items in a store ☺)
- Frequent itemsets are only positive combinations (we do not report combinations that do not occur frequently together)
- Frequent itemsets aims at providing a summary for the data

Finding frequent sets

- **Task:** Given a transaction database **D** and a **minsup** threshold find all frequent itemsets and the frequency of each set in this collection
- **Stated differently:** Count the number of times combinations of attributes occur in the data. If the count of a combination is above **minsup** report it.
- **Recall:** The input is a transaction database **D** where every transaction consists of a subset of items from some universe ***I***

How many itemsets are there?

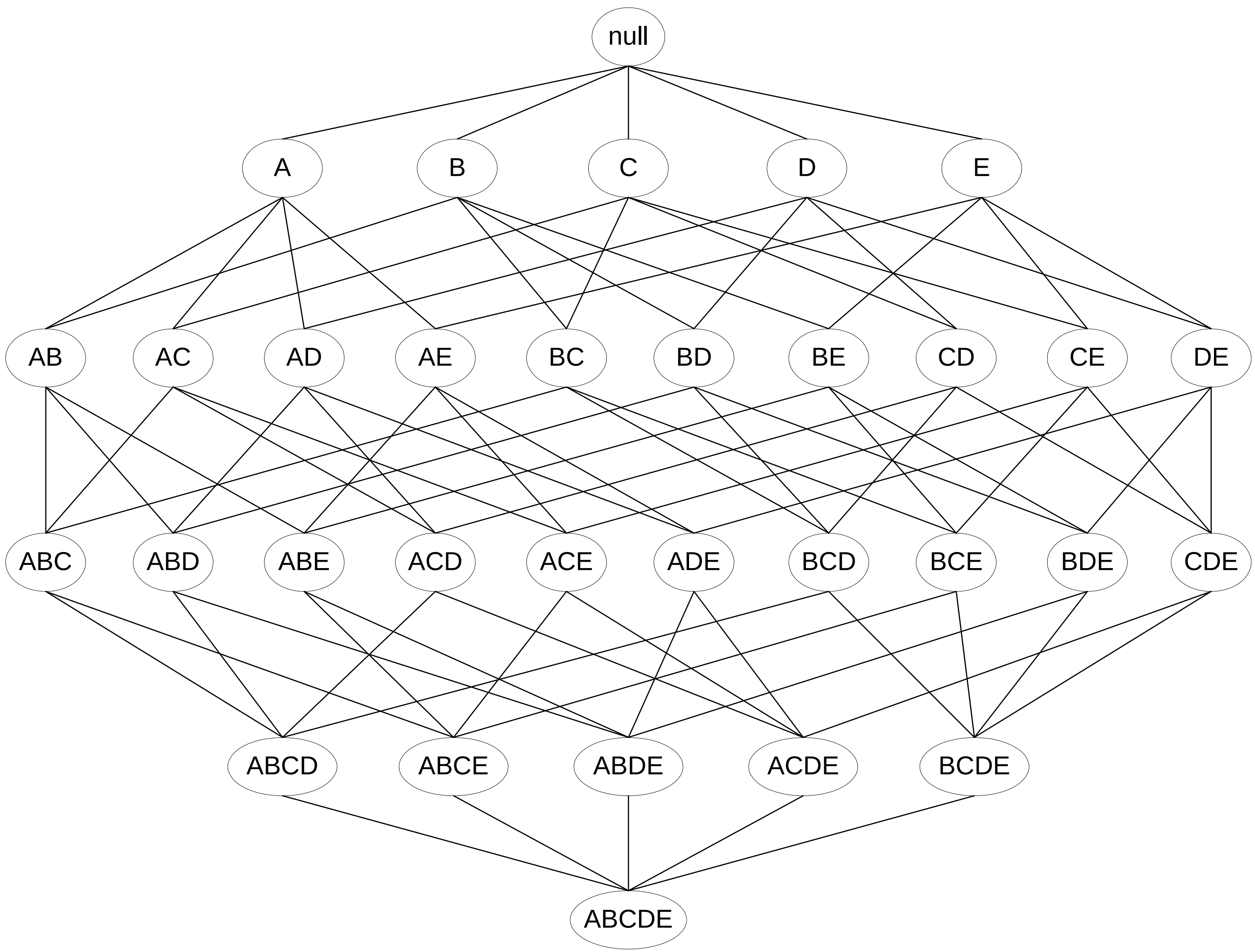


Given d items, there are 2^d possible itemsets

When is the task sensible and feasible?

- If **minsup = 0**, then all subsets of ***I*** will be frequent and thus the size of the collection will be very large
- This summary is very large (maybe larger than the original input) and thus not interesting
- The task of finding all frequent sets is interesting typically only for relatively large values of **minsup**

A simple algorithm for finding all frequent itemsets ??



Brute-force algorithm for finding all frequent itemsets?

- Generate all possible itemsets (lattice of itemsets)
  - Start with 1-itemsets, 2-itemsets,...,d-itemsets
- Compute the frequency of each itemset from the data
  - Count in how many transactions each itemset occurs
- If the support of an itemset is above **minsup** report it as a frequent itemset

Brute-force approach for finding all frequent itemsets

- Complexity?
  - Match every candidate against each transaction
  - For **M** candidates and **N** transactions, the complexity is~ **O(NMw)** => Expensive since $M = 2^d$!!!

Speeding-up the brute-force algorithm

- Reduce the number of candidates (M)
  - Complete search: $M=2^d$
  - Use pruning techniques to reduce M
- Reduce the number of transactions (N)
  - Reduce size of N as the size of itemset increases
  - Use vertical-partitioning of the data to apply the mining algorithms
- Reduce the number of comparisons (NM)
  - Use efficient data structures to store the candidates or transactions
  - No need to match every candidate against every transaction

Reduce the number of candidates

- Apriori principle (Main observation):
  - If an itemset is frequent, then all of its subsets must also be frequent

- Apriori principle holds due to the following property of the support measure:

$$\forall X, Y : (X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

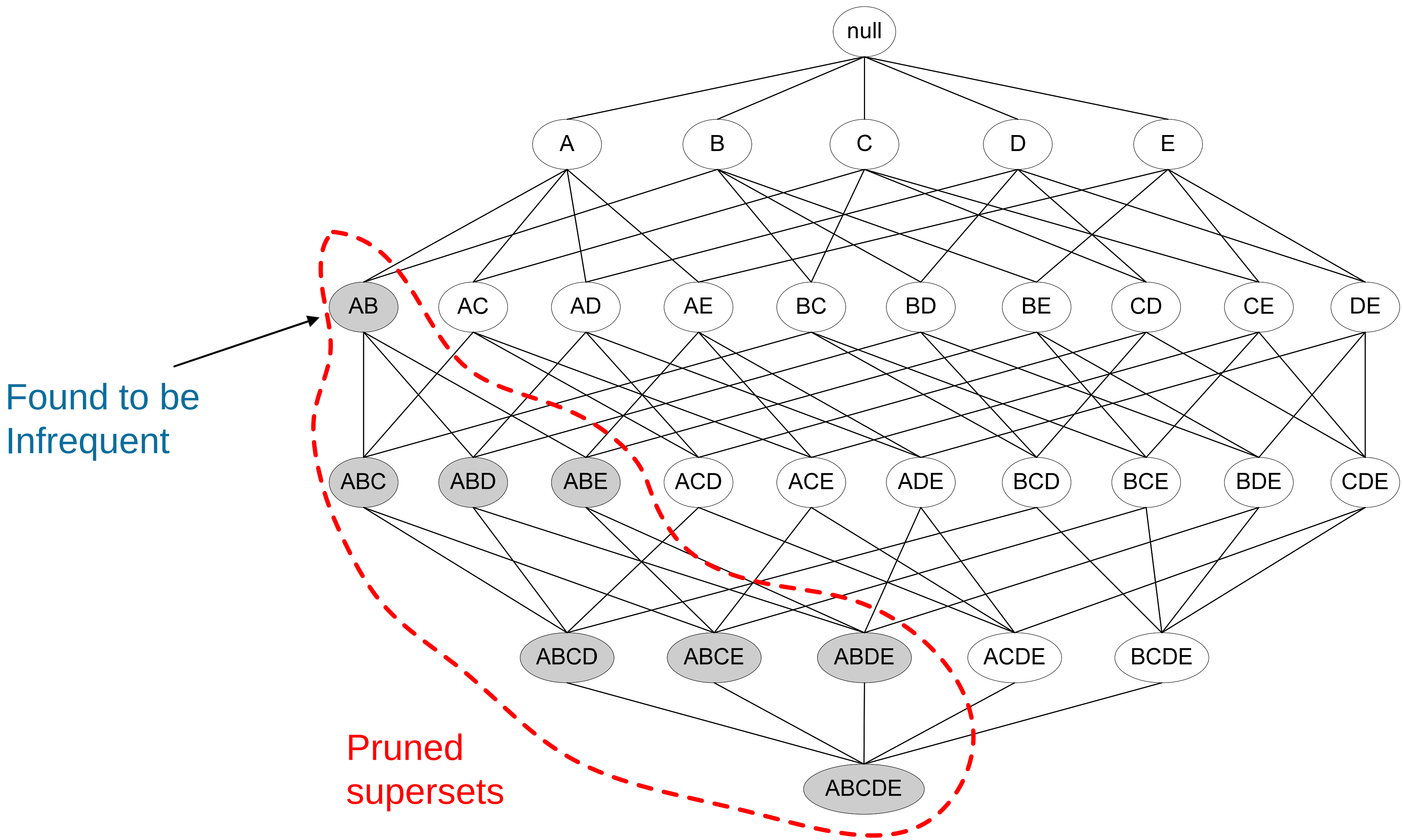
  - The support of an itemset ***never exceeds*** the support of its subsets
  - This is known as the ***anti-monotone*** property of support

Example

TID	*Items*
1	**Bread, Milk**
2	**Bread, Diaper, Beer, Eggs**
3	**Milk, Diaper, Beer, Coke**
4	**Bread, Milk, Diaper, Beer**
5	**Bread, Milk, Diaper, Coke**

s(Bread) > s(Bread, Beer)
s(Milk) > s(Bread, Milk)
s(Diaper, Beer) > s(Diaper, Beer, Coke)

Illustrating the Apriori principle



Illustrating the Apriori principle

Item	Count
Bread	**4**
Coke	**2**
Milk	**4**
Beer	**3**
Diaper	**4**
Eggs	**1**

Items (1-itemsets)

Itemset	Count
{Bread,Milk}	**3**
{Bread,Beer}	**2**
{Bread,Diaper}	**3**
{Milk,Beer}	**2**
{Milk,Diaper}	**3**
{Beer,Diaper}	**3**

Pairs (2-itemsets)

(No need to generate candidates involving Coke or Eggs)

minsup = 3/5

Triplets (3-itemsets)

Itemset	Count
{Bread,Milk,Diaper}	**3**

If every subset is considered,
$^6C_1 + {}^6C_2 + {}^6C_3 = 41$
With support-based pruning,
6 + 6 + 1 = 13

Exploiting the Apriori principle

1. Find frequent 1-items and put them to L_k ($k=1$)
2. Use L_k to generate a collection of *candidate* itemsets C_{k+1} with size ($k+1$)
3. Scan the database to find which itemsets in C_{k+1} are frequent and put them into L_{k+1}
4. If L_{k+1} is not empty
   - $k=k+1$
   - Goto step 2

R. Agrawal, R. Srikant: "Fast Algorithms for Mining Association Rules", *Proc. of the 20th Int'l Conference on Very Large Databases*, 1994.

The Apriori algorithm

C_k: Candidate itemsets of size k

L_k : frequent itemsets of size k

L_1 = {frequent 1-itemsets};

for (k = 2; L_k !=$\emptyset$; k++)

$\quad C_{k+1}$ = GenerateCandidates(L_k)

$\quad$ **for** each transaction t in database do

$\quad\quad$ increment count of candidates in C_{k+1} that are contained in **t**

$\quad$ **endfor**

$\quad L_{k+1}$ = candidates in C_{k+1} with support $\geq$***min_sup***

endfor

return $\cup_k L_k$;

GenerateCandidates

- Assume the items in L_k are listed in an order (e.g., alphabetical)
- **Step 1:** ***self-joining*** L_k *(IN SQL)*

  insert into C_{k+1}

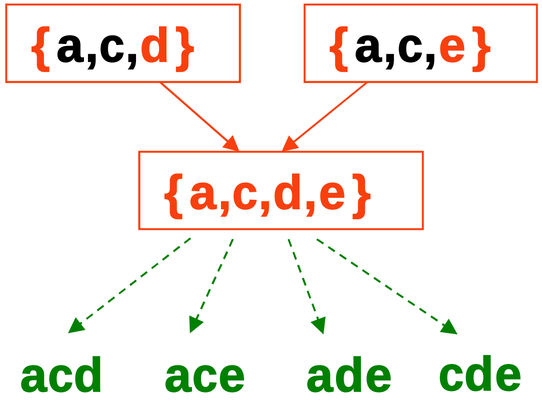
  select $p.item_1, p.item_2, \ldots, p.item_k, q.item_k$

  from $L_k\ p, L_k\ q$

  where $p.item_1 = q.item_1, \ldots, p.item_{k-1} = q.item_{k-1}, p.item_k < q.item_k$

Example of Candidates Generation

- L_3={*abc, abd, acd, ace, bcd*}
- ***Self-joining***: $L_3 * L_3$
  - ***abcd*** from ***abc*** and ***abd***
  - ***acde*** from ***acd*** and ***ace***



GenerateCandidates

- Assume the items in L_k are listed in an order (e.g., alphabetical)
- **Step 1:** ***self-joining*** L_k *(IN SQL)*

  insert into C_{k+1}

  select $p.item_1, p.item_2, \ldots, p.item_k, q.item_k$

  from $L_k\ p, L_k\ q$

  where $p.item_1=q.item_1, \ldots, p.item_{k-1}=q.item_{k-1}, p.item_k < q.item_k$
- **Step 2:** ***pruning***

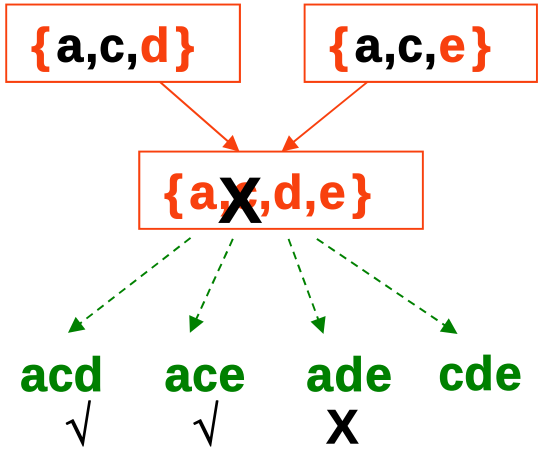
  forall ***itemsets c in*** C_{k+1} do

  forall ***k-subsets s of c*** do

  if *(**s** is **not** in L_k)* **then delete *c* from** C_{k+1}

Example of Candidates Generation

- L_3={*abc, abd, acd, ace, bcd*}
- ***Self-joining***: $L_3 * L_3$
  - ***abcd*** from ***abc*** and ***abd***
  - ***acde*** from ***acd*** and ***ace***
- ***Pruning:***
  - ***acde*** is removed because ***ade*** is not in L_3
- C_4={*abcd*}



The Apriori algorithm

C_k: Candidate itemsets of size k

L_k : frequent itemsets of size k

L_1 = {frequent items};

for (k = 1; L_k !=$\emptyset$; k++)

$\boxed{C_{k+1} = \text{GenerateCandidates}(L_k)}$

for each transaction t in database do

$\boxed{\text{increment count of candidates in } C_{k+1} \text{ that are contained in } \mathbf{t}}$

endfor

L_{k+1} = candidates in C_{k+1} with support $\geq$ ***min_sup***

endfor

return $\cup_k L_k$;

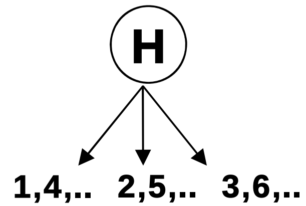
How to Count Supports of Candidates?

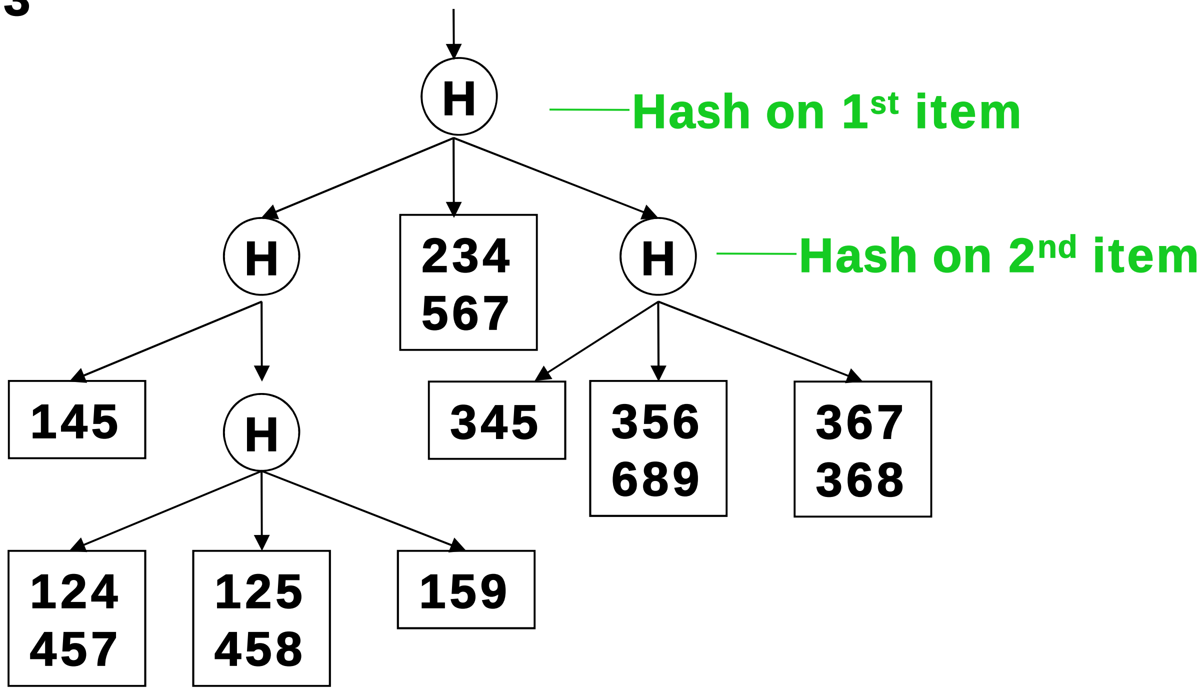
- Naive algorithm?

– Method:
  - Candidate itemsets are stored in a *hash-tree*
  - *Leaf* node of hash-tree contains a list of itemsets and counts
  - *Interior* node contains a hash table
  - *Subset function*: finds all the candidates contained in a transaction

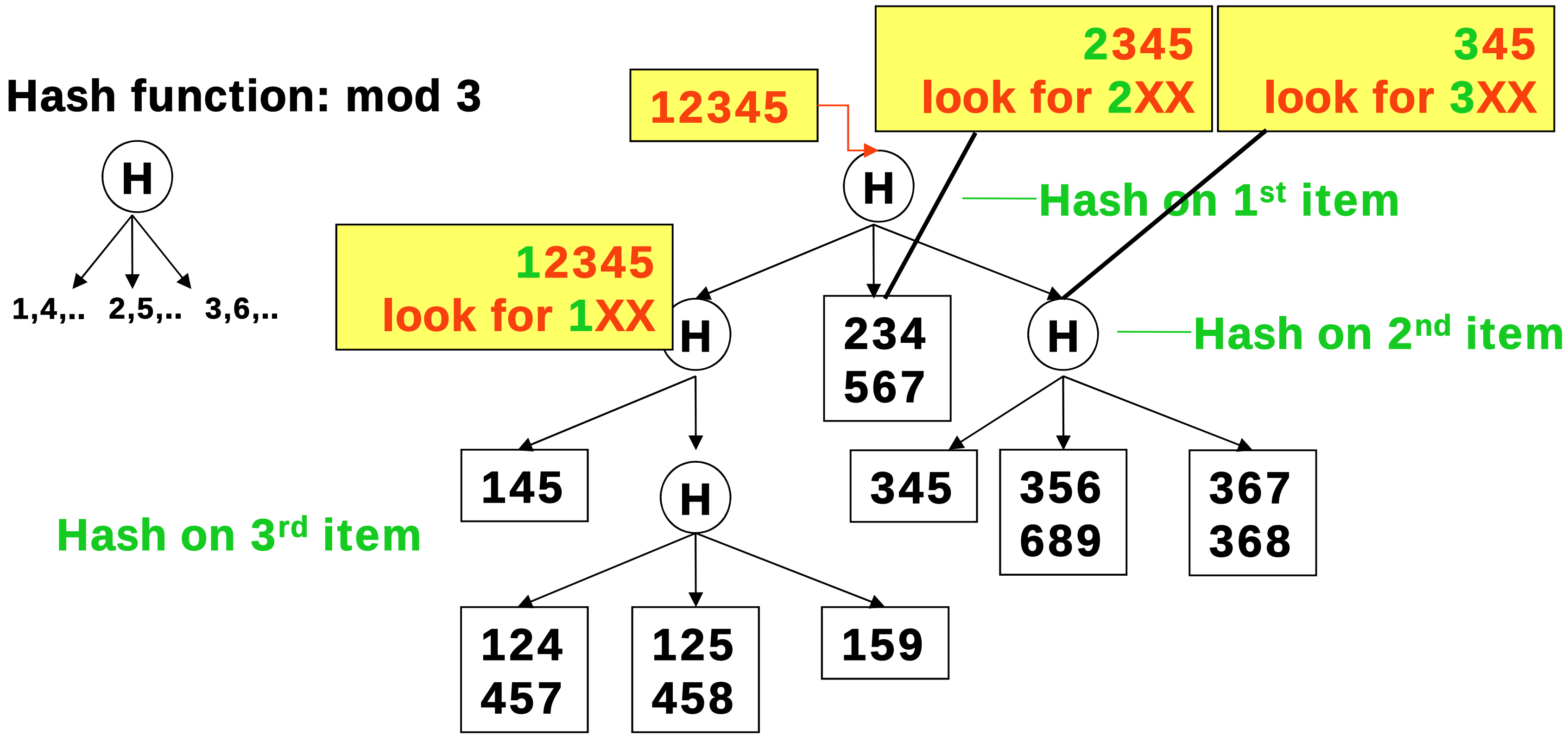
Example of the hash-tree for C_3

Hash function: mod 3

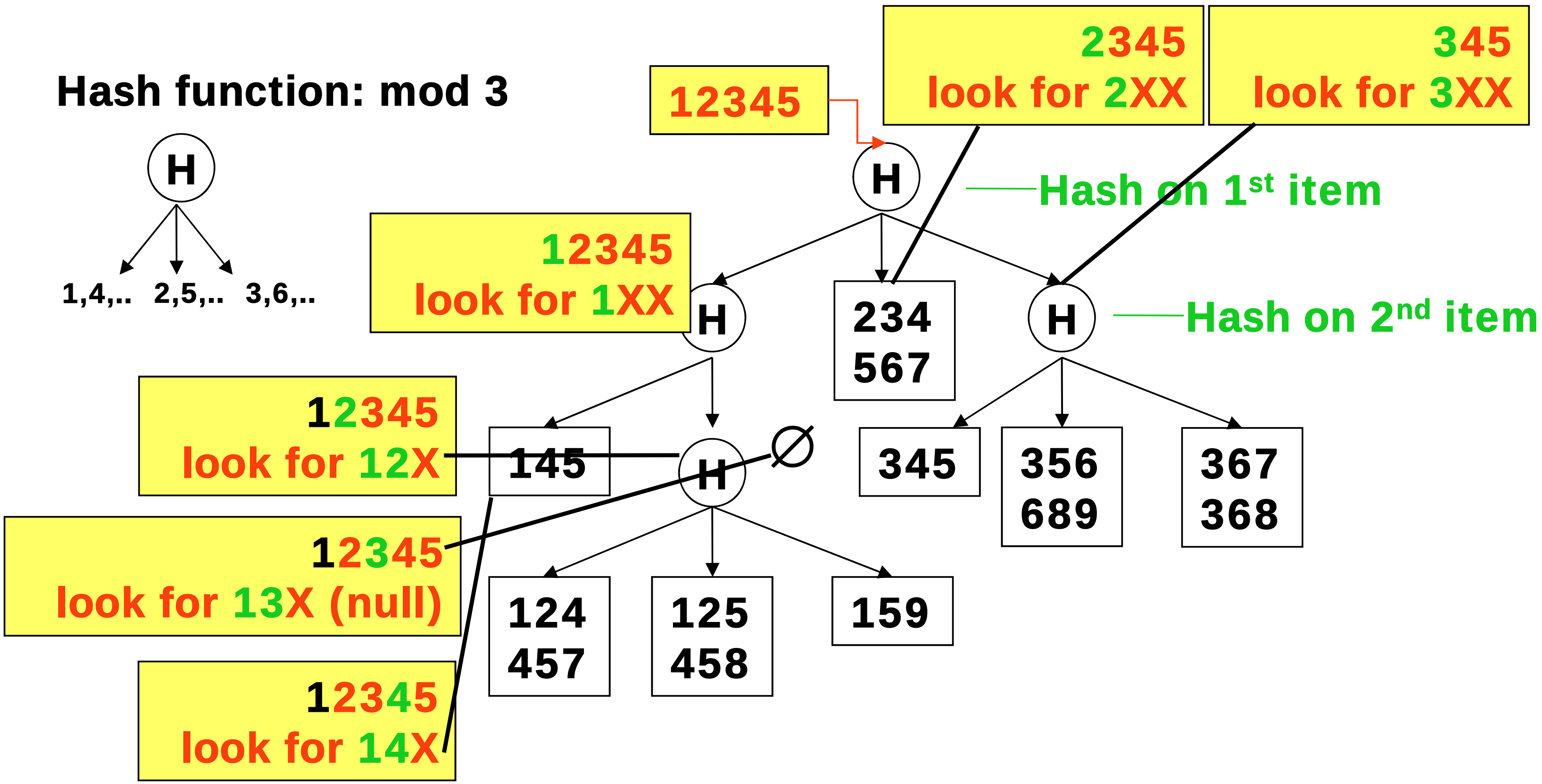




Example of the hash-tree for C_3



Example of the hash-tree for C_3



The subset function finds all the candidates contained in a transaction:

- At the root level it hashes on all items in the transaction
- At level i it hashes on all items in the transaction that come after item the i-th item

Discussion of the Apriori algorithm

- Much faster than the Brute-force algorithm
  - It avoids checking all elements in the lattice
- The running time is in the worst case **$O(2^d)$**
  - Pruning really prunes in practice
- It makes multiple passes over the dataset
  - One pass for every level **k**
- Multiple passes over the dataset is inefficient when we have thousands of candidates and millions of transactions

Making a single pass over the data: the AprioriTid algorithm

- The database is **not** used for counting support after the 1st pass!

- Instead information in data structure C_k' is used for counting support in every step

  - **$C_k' = \{$<TID, $\{X_k\}$> | X_k is a potentially frequent k-itemset in transaction with id=TID$\}$**

  - **C_1':** corresponds to the original database (every item **i** is replaced by itemset **{i}**)

  - The member C_k' corresponding to transaction **t** is **<t.TID, $\{c \in C_k \mid$ c is contained in t$\}$>**

The AprioriTID algorithm

- L_1 = {frequent 1-itemsets}
- C_1' = database **D**
- **for** (k=2, $L_{k-1}' \neq$ empty; k++)

  C_k = GenerateCandidates(L_{k-1})

  C_k' = {}

  for all entries **t** ϵ C_{k-1}'

  $C_t = \{c \epsilon C_k \mid t[c-c[k]]=1 \text{ and } t[c-c[k-1]]=1\}$

  for all $c \epsilon C_t$ {c.count++}

  if ($C_t \neq \{\}$)

  append C_t to C_k'

  endif

  endfor

  $L_k = \{c \epsilon C_k \mid c.count >= minsup\}$

  endfor
- **return** $\bigcup_k L_k$

AprioriTid Example (minsup=2)

Database D

TID	Items
100	1 3 4
200	2 3 5
300	1 2 3 5
400	2 5

C_1'

TID	Sets of itemsets
100	{ {1},{3},{4} }
200	{ {2},{3},{5} }
300	{ {1},{2},{3},{5} }
400	{ {2},{5} }

L_1

itemset	sup.
{1}	2
{2}	3
{3}	3
{5}	3

C_2

itemset
{1 2}
{1 3}
{1 5}
{2 3}
{2 5}
{3 5}

C_2'

TID	Sets of itemsets
100	{ {1 3} }
200	{ {2 3},{2 5},{3 5} }
300	{ {1 2},{1 3},{1 5}, {2 3},{2 5},{3 5} }
400	{ {2 5} }

L_2

itemset	sup
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

C_3

itemset
{2 3 5}

C_3'

TID	Sets of itemsets
200	{ {2 3 5} }
300	{ {2 3 5} }

L_3

itemset	sup
{2 3 5}	2

Discussion on the AprioriTID algorithm

- L_1 = {frequent 1-itemsets}
- C_1' = database **D**
- **for** (k=2, $L_{k-1}' \neq$ empty; k++)

    C_k = GenerateCandidates(L_{k-1})

    C_k' = {}

    for all entries **t** ϵ C_{k-1}'

        $C_t = \{c \in C_k \mid t[c-c[k]]=1 \text{ and } t[c-c[k-1]]=1\}$

        for all $c \in C_t$ {c.count++}

        if ($C_t \neq \{\}$)

            append C_t to C_k'

        endif

    endfor

    $L_k = \{c \in C_k \mid c.count >= minsup\}$

  endfor
- **return** $\cup_k L_k$

- One single pass over the data
- C_k' is generated from C_{k-1}'
- For small values of **k**, C_k' could be larger than the database!
- For large values of **k**, C_k' can be very small

Apriori vs. AprioriTID

- *Apriori* makes multiple passes over the data while *AprioriTID* makes a single pass over the data
- *AprioriTID* needs to store additional data structures that may require more space than *Apriori*
- Both algorithms need to check all candidates' frequencies in every step

Implementations

- Lots of them around
- See, for example, the web page of Bart Goethals: http://www.adrem.ua.ac.be/~goethals/software/
- Typical input format: each row lists the items (using item id's) that appear in every row

Lecture outline

- **Task 1:** Methods for finding all frequent itemsets efficiently
- **Task 2:** Methods for finding association rules efficiently

Definition: Association Rule

Let **D** be database of transactions

– e.g.:

Transaction ID	Items
2000	A, B, C
1000	A, C
4000	A, D
5000	B, E, F

- Let ***I*** be the set of items that appear in the database, e.g., ***I={A,B,C,D,E,F}***
- A **rule** is defined by $X \rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \varnothing$
  – e.g.: $\{B,C\} \rightarrow \{A\}$ is a rule

Definition: Association Rule

TID	*Items*
1	**Bread, Milk**
2	**Bread, Diaper, Beer, Eggs**
3	**Milk, Diaper, Beer, Coke**
4	**Bread, Milk, Diaper, Beer**
5	**Bread, Milk, Diaper, Coke**

- **Association Rule**
  - An implication expression of the form **X** $\rightarrow$ **Y**, where **X** and **Y** are non-overlapping itemsets
  - Example:
    {Milk, Diaper} $\rightarrow$ ***{Beer}***

- **Rule Evaluation Metrics**
  - **Support (s)**
    - Fraction of transactions that contain both **X** and **Y**
  - **Confidence (c)**
    - Measures how often items in **Y** appear in transactions that contain **X**

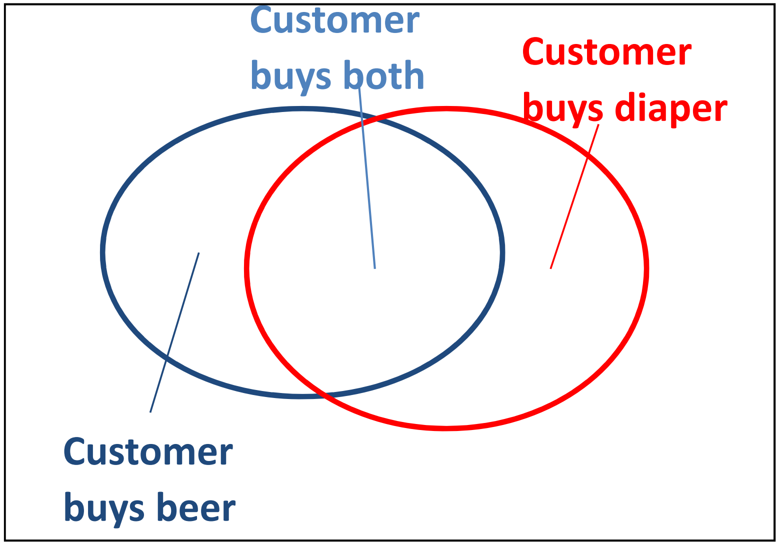
Example:

$$\{\text{Milk}, \text{Diaper}\} \rightarrow \text{Beer}$$

$$s = \frac{\sigma(\text{Milk}, \text{Diaper}, \text{Beer})}{|\text{T}|} = \frac{2}{5} = 0.4$$

$$c = \frac{\sigma(\text{Milk}, \text{Diaper}, \text{Beer})}{\sigma(\text{Milk}, \text{Diaper})} = \frac{2}{3} = 0.67$$

Rule Measures: Support and Confidence



Find all the rules ***X → Y*** with minimum confidence and support

- support, *s*, probability that a transaction contains **{X ∪ Y}**
- confidence, *c*, ***conditional probability*** that a transaction having **X** also contains **Y**

TID	Items
100	A,B,C
200	A,C
300	A,D
400	B,E,F

Let minimum support 50%, and minimum confidence 50%, we have

- *A → C* (50%, 66.6%)
- *C → A* (50%, 100%)

Example

TID	date	items bought
100	10/10/99	{F,A,D,B}
200	15/10/99	{D,A,C,E,B}
300	19/10/99	{C,A,B,E}
400	20/10/99	{B,A,D}

What is the ***support*** and ***confidence*** of the rule: {B,D} → {A}

- Support:
  - percentage of tuples that contain {A,B,D} = 75%
- Confidence:

$$\frac{\text{number of tuples that contain }\{A, B, D\}}{\text{number of tuples that contain }\{B, D\}} = 100\%$$

Association-rule mining task

- Given a set of transactions **D**, the goal of association rule mining is to find **all** rules having
  - support ≥ ***minsup*** threshold
  - confidence ≥ ***minconf*** threshold

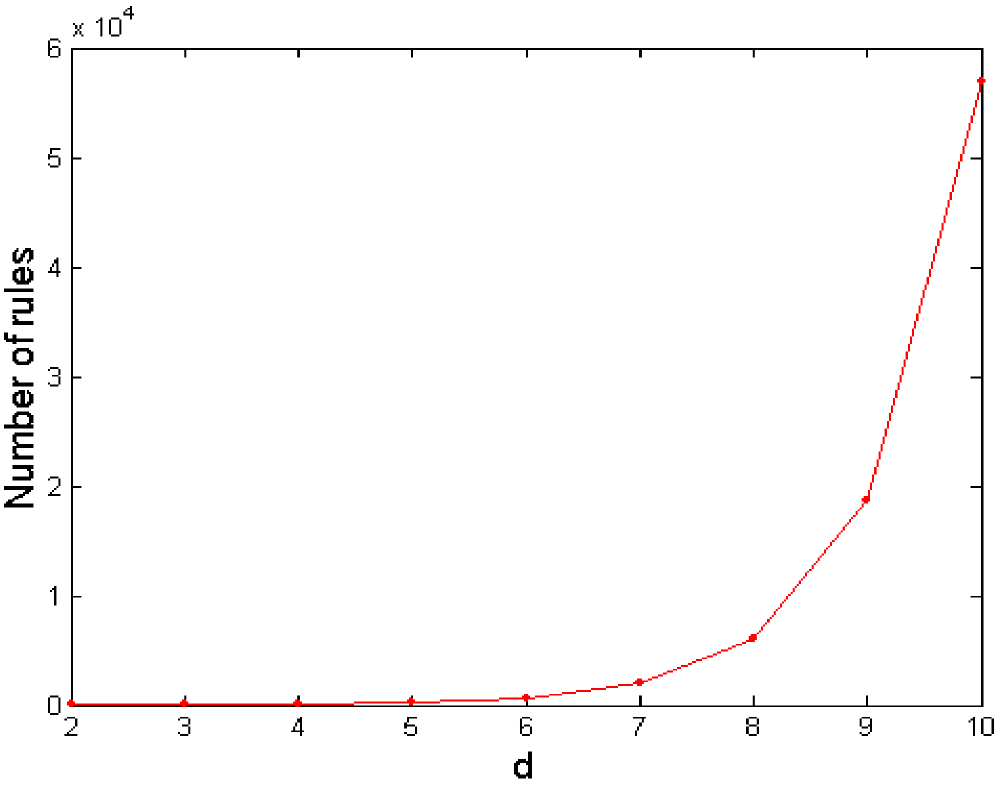
Brute-force algorithm for association-rule mining

- List all possible association rules
- Compute the support and confidence for each rule
- Prune rules that fail the *minsup* and *minconf* thresholds

- $\Rightarrow$ Computationally prohibitive!

Computational Complexity

- Given **d** unique items in ***I***:
  - Total number of itemsets = $\mathbf{2^d}$
  - Total number of possible association rules:



$$R = \sum_{k=1}^{d-1}\left[\binom{d}{k} \times \sum_{j=1}^{d-k}\binom{d-k}{j}\right]$$

$$= 3^d - 2^{d+1} + 1$$

Mining Association Rules

TID	*Items*
1	**Bread, Milk**
2	**Bread, Diaper, Beer, Eggs**
3	**Milk, Diaper, Beer, Coke**
4	**Bread, Milk, Diaper, Beer**
5	**Bread, Milk, Diaper, Coke**

Example of Rules:

{Milk,Diaper} $\rightarrow$ {Beer} (s=0.4, c=0.67)
{Milk,Beer} $\rightarrow$ {Diaper} (s=0.4, c=1.0)
{Diaper,Beer} $\rightarrow$ {Milk} (s=0.4, c=0.67)
{Beer} $\rightarrow$ {Milk,Diaper} (s=0.4, c=0.67)
{Diaper} $\rightarrow$ {Milk,Beer} (s=0.4, c=0.5)
{Milk} $\rightarrow$ {Diaper,Beer} (s=0.4, c=0.5)

Observations:

- All the above rules are binary partitions of the same itemset: {Milk, Diaper, Beer}
- Rules originating from the same itemset have identical support but can have different confidence
- Thus, we may decouple the support and confidence requirements

Mining Association Rules

- Two-step approach:
  - Frequent Itemset Generation
    - Generate all itemsets whose support $\geq$ minsup
  - Rule Generation
    - Generate high confidence rules from each frequent itemset, where each rule is a binary partition of a frequent itemset

Rule Generation – Naive algorithm

- Given a frequent itemset **X**, find all non-empty subsets $y \subset X$ such that $y \rightarrow X - y$ satisfies the minimum confidence requirement
  - If **{A,B,C,D}** is a frequent itemset, candidate rules:

    $ABC \rightarrow D$, $ABD \rightarrow C$, $ACD \rightarrow B$, $BCD \rightarrow A$,
    $A \rightarrow BCD$, $B \rightarrow ACD$, $C \rightarrow ABD$, $D \rightarrow ABC$
    $AB \rightarrow CD$, $AC \rightarrow BD$, $AD \rightarrow BC$, $BC \rightarrow AD$,
    $BD \rightarrow AC$, $CD \rightarrow AB$,

- If $|X| = k$, then there are $2^k - 2$ candidate association rules (ignoring $L \rightarrow \varnothing$ and $\varnothing \rightarrow L$)

Efficient rule generation

- How to efficiently generate rules from frequent itemsets?
  - In general, confidence does not have an anti-monotone property

    $c(ABC \rightarrow D)$ can be larger or smaller than **$c(AB \rightarrow D)$**

  - ***But confidence of rules generated from the same itemset has an anti-monotone property***
  - Example: $X = \{A,B,C,D\}$:

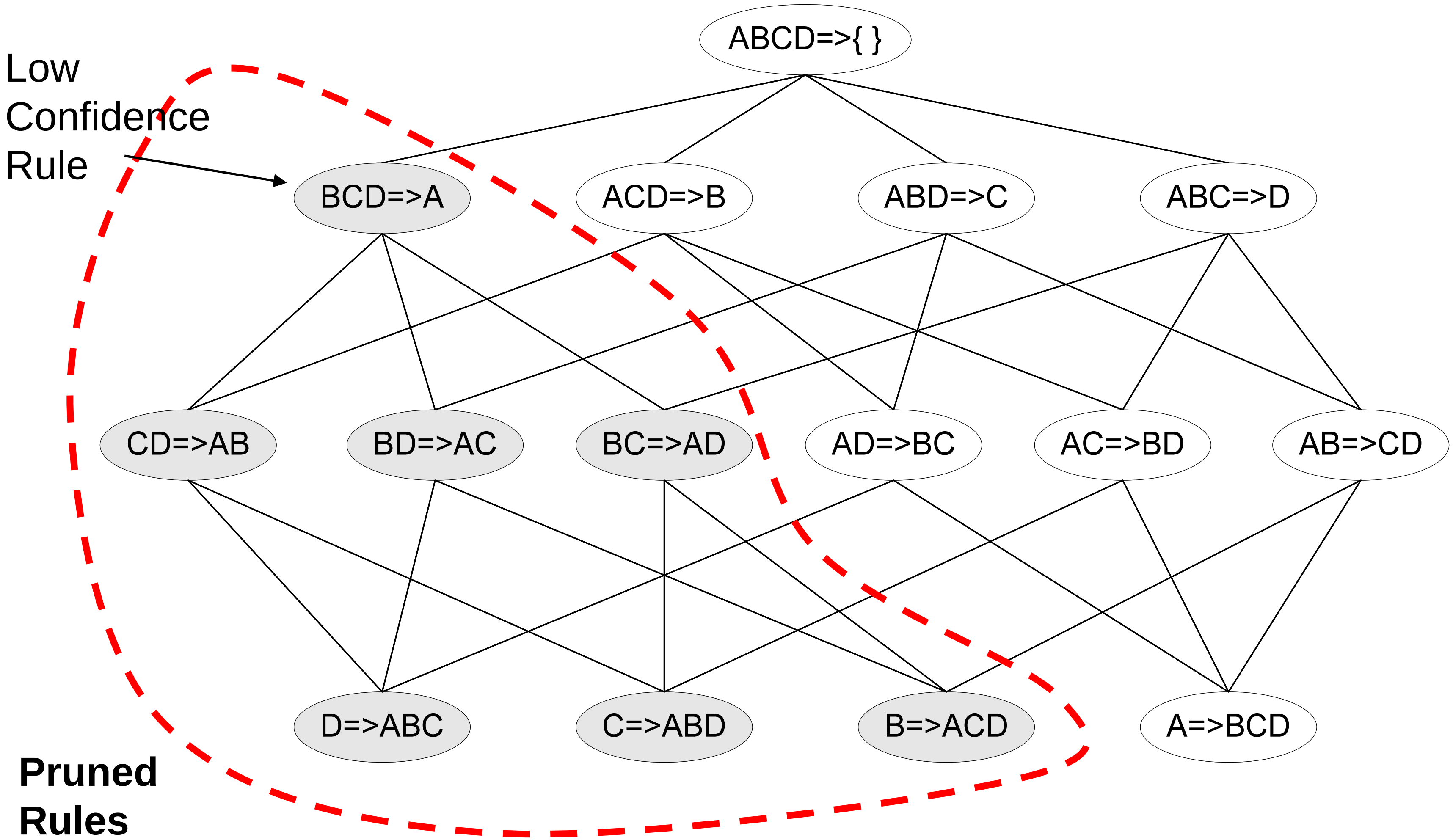
$$c(ABC \rightarrow D) \geq c(AB \rightarrow CD) \geq c(A \rightarrow BCD)$$

  - **Why?**

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

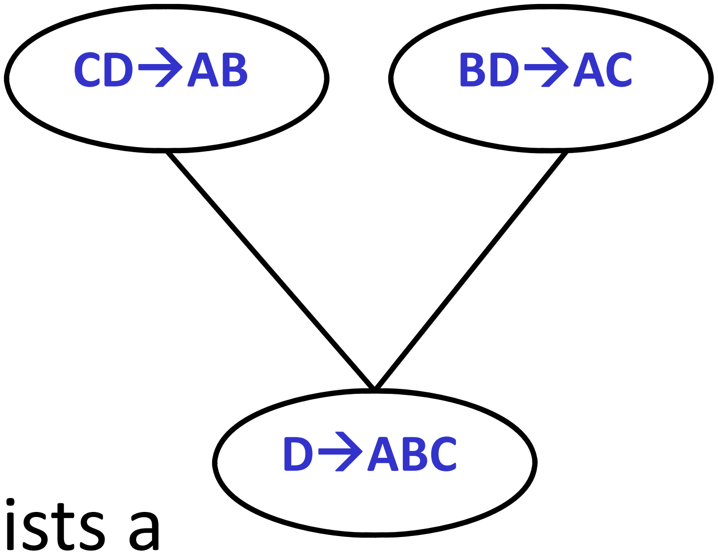
Rule Generation for Apriori Algorithm

Lattice of rules



Apriori algorithm for rule generation

- Candidate rule is generated by merging two rules that share the same prefix in the rule consequent



- **join(CD→AB,BD—>AC)** would produce the candidate rule **D →ABC**

- **Prune** rule **D→ABC** if there exists a subset (e.g., **AD→BC**) that does not have high confidence