Data Mining

Part 3. Associations Rules

3.2 Efficient Frequent Itemset Mining Methods

Fall 2009

Instructor: Dr. Masoud Yaghini

Outline

- Apriori Algorithm
- Generating Association Rules from Frequent Itemsets
- FP-growth Algorithm
- References

Apriori Algorithm

Apriori Algorithm

- Complete search approach:
  - List all possible itemsets $M = 2^d - 1$
  - Count the support of each itemset by scanning the database
  - Eliminate itemsets that fail the *min_sup*

  $\Rightarrow$ Computationally prohibitive!

- Reduce the number of candidates (M)
  - Use pruning techniques to reduce M
  - e.g. Apriori algorithm

Apriori Algorithm

- Finding frequent itemsets using candidate generation
- Proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for **Boolean association rules**.
- The name of the algorithm is based on the fact that the algorithm uses ***prior knowledge*** of frequent itemset properties.
- Apriori employs an iterative approach, where k-itemsets are used to explore (k+1)-itemsets.

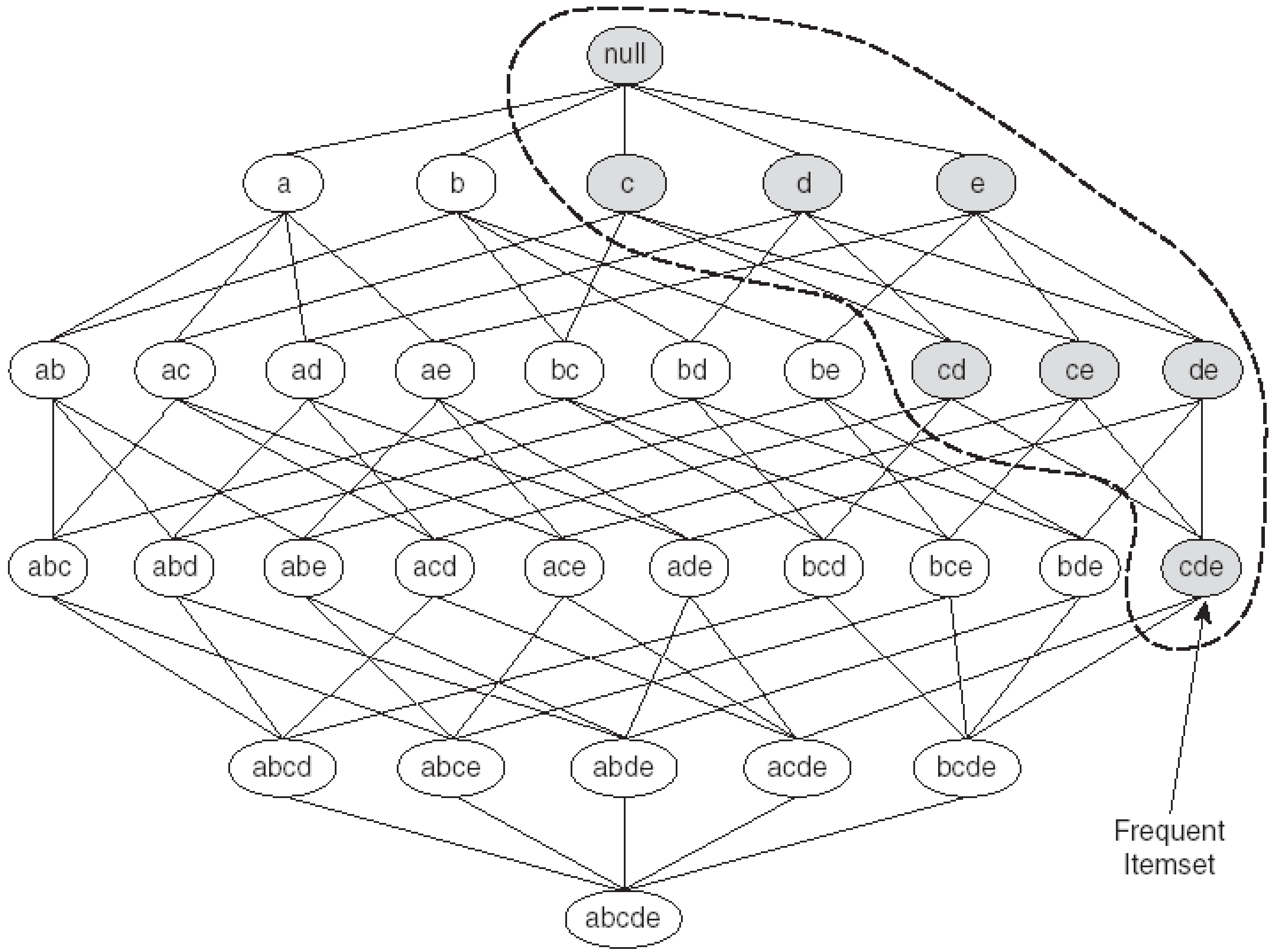
Apriori Algorithm

- Let k=1
- Scan DB once to get frequent k-itemset
- Repeat until no new frequent or candidate itemsets are identified
  - Generate length (k+1) candidate itemsets from length k frequent itemsets
  - Prune candidate itemsets containing subsets of length k that are infrequent
  - Scan DB to count the support of each candidate
  - Eliminate candidates that are infrequent, leaving only those that are frequent

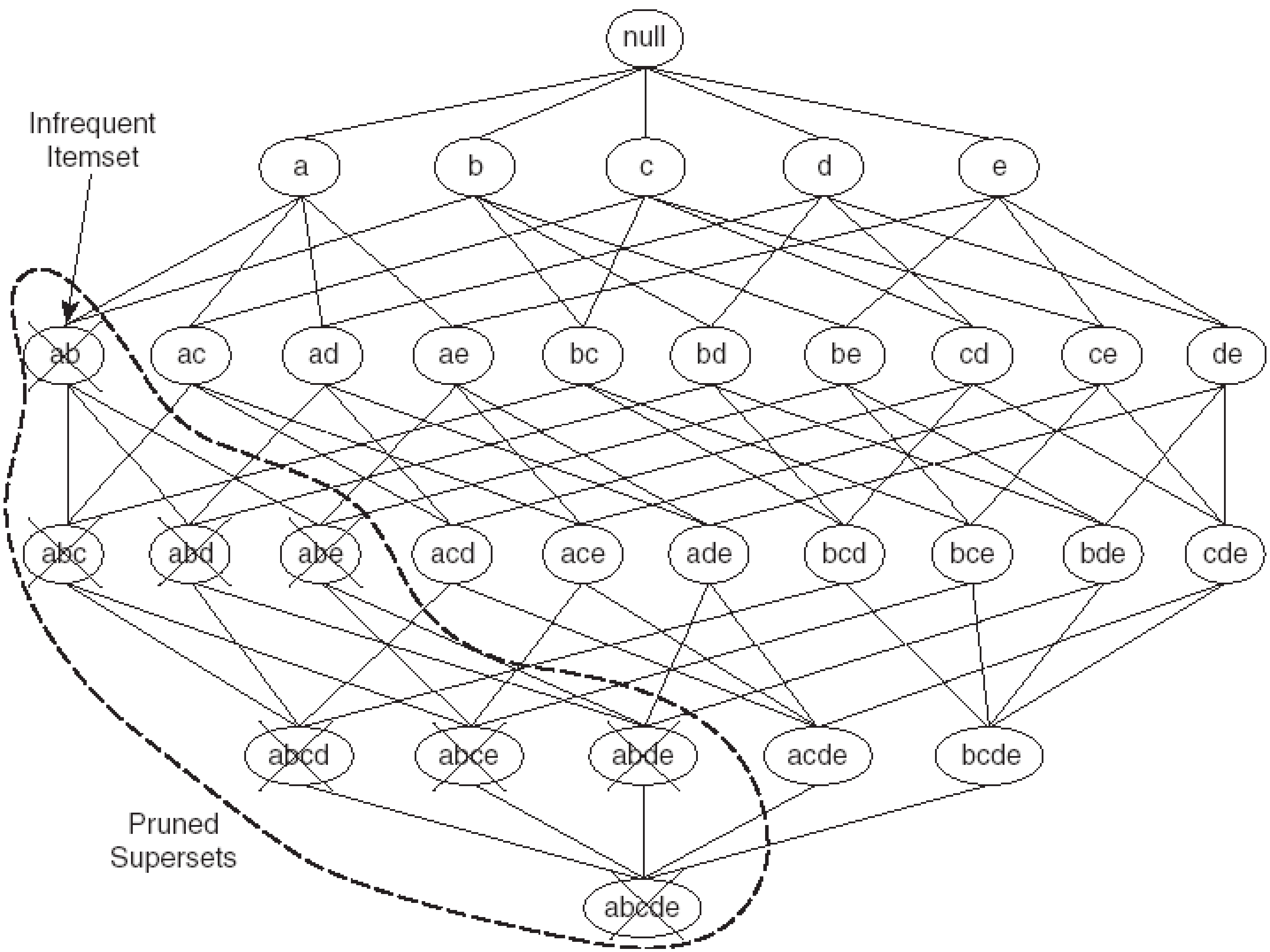
Apriori Algorithm

- Apriori property:
  - All nonempty subsets of a frequent itemset must also be frequent.
- Apriori pruning principle:
  - If there is any itemset which is infrequent, its superset is not frequent either and it should not be generated as a candidate

Apriori Principle



Apriori Principle



Example: Apriori Algorithm

- Transactional data for an *AllElectronics*
- the minimum support count required is 2, that is, *min_sup = 2.*

TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Example: Apriori Algorithm

- Step 1:
  - generate a candidate set of 1-itemsets, C1.
  - scans all of the transactions in order to count the number of occurrences of each item.

Scan *D* for count of each candidate

C_1

Itemset	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

Example: Apriori Algorithm

- Step 2:
  - Determine the set of frequent 1-itemsets, L1.
  - all of the candidates in C1 satisfy minimum support.

Compare candidate support count with minimum support count

L_1

Itemset	Sup. count
{I1}	6
{I2}	7
{I3}	6
{I4}	2
{I5}	2

Example: Apriori Algorithm

- Step 3:
  - generate a candidate set of 2-itemsets, *C2*

Generate C_2 candidates from L_1 →

C_2

Itemset
{I1, I2}
{I1, I3}
{I1, I4}
{I1, I5}
{I2, I3}
{I2, I4}
{I2, I5}
{I3, I4}
{I3, I5}
{I4, I5}

Example: Apriori Algorithm

- Step 4:
  - scans all of the transactions in order to count the number of occurrences of each item in *C2.*

Scan D for count of each candidate

C_2

Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I4}	1
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2
{I3, I4}	0
{I3, I5}	1
{I4, I5}	0

Example: Apriori Algorithm

- Step 5:
  - Determine the set of frequent 2-itemsets, L2.

Compare candidate support count with minimum support count

L_2

Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2

Example: Apriori Algorithm

- Step 6: generate a candidate set of 3-itemsets, *C3*

(a) Join: $C_3 = L_2 \bowtie L_2 = \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\} \bowtie$
$\{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$= \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$.

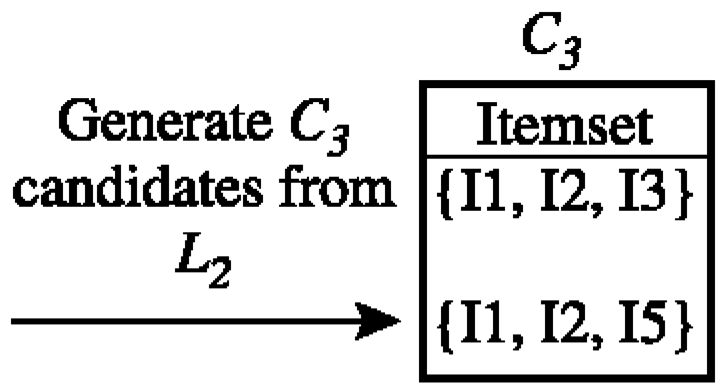
(b) Prune using the Apriori property: All nonempty subsets of a frequent itemset must also be frequent. Do any of the candidates have a subset that is not frequent?

- The 2-item subsets of {I1, I2, I3} are {I1, I2}, {I1, I3}, and {I2, I3}. All 2-item subsets of {I1, I2, I3} are members of L_2. Therefore, keep {I1, I2, I3} in C_3.
- The 2-item subsets of {I1, I2, I5} are {I1, I2}, {I1, I5}, and {I2, I5}. All 2-item subsets of {I1, I2, I5} are members of L_2. Therefore, keep {I1, I2, I5} in C_3.
- The 2-item subsets of {I1, I3, I5} are {I1, I3}, {I1, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I1, I3, I5} from C_3.
- The 2-item subsets of {I2, I3, I4} are {I2, I3}, {I2, I4}, and {I3, I4}. {I3, I4} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I4} from C_3.
- The 2-item subsets of {I2, I3, I5} are {I2, I3}, {I2, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I5} from C_3.
- The 2-item subsets of {I2, I4, I5} are {I2, I4}, {I2, I5}, and {I4, I5}. {I4, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I4, I5} from C_3.

(c) Therefore, $C_3 = \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$ after pruning.

Example: Apriori Algorithm

- Step 6: (cont.)
  - The resulting pruned version of C3



The Apriori Algorithm

- Step 7:
  - scans all of the transactions in order to count the number of occurrences of each item in *C3.*

Scan *D* for count of each candidate →

C_3

Itemset	Sup. count
{I1, I2, I3}	2
{I1, I2, I5}	2

The Apriori Algorithm

- Step 8:
  - Determine the set of frequent 3-itemsets, L3.

Compare candidate support count with minimum support count

L_3

Itemset	Sup. count
{I1, I2, I3}	2
{I1, I2, I5}	2

The Apriori Algorithm

- Step 9:
  - Generate a candidate set of 4-itemsets, C4.
  - Although the join results in {{I1, I2, I3, I5}}, this itemset is pruned because its subset {{I2, I3, I5}} is not frequent.
  - Thus, C4 = $\emptyset$, and the algorithm terminates, having found all of the frequent itemsets.

The Apriori Algorithm

- <u>Pseudo-code</u>:

  C_k: Candidate itemset of size k
  L_k : frequent itemset of size k

```
L_1 = {frequent items};
for (k = 1; L_k !=∅; k++) do begin
    C_{k+1} = candidates generated from L_k;
   for each transaction t in database do
        increment the count of all candidates in C_{k+1}
        that are contained in t
   L_{k+1} = candidates in C_{k+1} with min_support
   end
return ∪_k L_k;
```

Important Details of Apriori

- How to generate candidates?
  - Step 1: self-joining L_k
  - Step 2: pruning
- Example of Candidate-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*}

Generating Association Rules from Frequent Itemsets

Generating Association Rules

- Once the **frequent itemsets** from transactions in a database D have been found, it is straightforward to generate **strong association rules** from them
- Where strong association rules satisfy both minimum support and minimum confidence.
- Confidence:

$$confidence(A \Rightarrow B) = P(B|A) = \frac{support\_count(A \cup B)}{support\_count(A)}$$

Generating Association Rules

- Association rules can be generated as follows:
  - For each frequent itemset *l*, generate all nonempty subsets of *l*.
  - For every nonempty subset *s* of *l*, output the rule

$$\text{“}s \Rightarrow (l - s)\text{” if } \frac{support\_count(l)}{support\_count(s)} \geq min\_conf,$$

  - where *min_conf* is the minimum confidence threshold.

- Because the rules are generated from frequent itemsets, each one automatically satisfies minimum support.

Example: Generating Association Rules

- Suppose the data contain the frequent itemset I = {I1, I2, I5}.
- What are the association rules that can be generated from I?
- The nonempty subsets of I are
  - {I1, I2}, {I1, I5}, {I2, I5}, {I1}, {I2}, and {I5}.

Example: Generating Association Rules

- The resulting association rules are as shown below, each listed with its confidence:

$$
\begin{array}{ll}
I1 \wedge I2 \Rightarrow I5, & confidence = 2/4 = 50\% \\
I1 \wedge I5 \Rightarrow I2, & confidence = 2/2 = 100\% \\
I2 \wedge I5 \Rightarrow I1, & confidence = 2/2 = 100\% \\
I1 \Rightarrow I2 \wedge I5, & confidence = 2/6 = 33\% \\
I2 \Rightarrow I1 \wedge I5, & confidence = 2/7 = 29\% \\
I5 \Rightarrow I1 \wedge I2, & confidence = 2/2 = 100\%
\end{array}
$$

- If the minimum confidence threshold is, say, 70%, then only the second, third, and last rules above are output,

Factors Affecting Complexity

- Choice of minimum support threshold
  - lowering support threshold results in more frequent itemsets
  - this may increase number of candidates and max length of frequent itemsets
- Dimensionality (number of items) of the data set
  - more space is needed to store support count of each item
  - if number of frequent items also increases, both computation and I/O costs may also increase
- Size of database
  - since Apriori makes multiple passes, run time of algorithm may increase with number of transactions
- Average transaction width
  - transaction width increases with denser data sets
  - This may increase max length of frequent itemsets and traversals of hash tree (number of subsets in a transaction increases with its width)

FP-growth Algorithm

Bottleneck of Apriori Algorithm

- Apriori has two shortcoming:
  - Generating a huge number of candidate sets
  - Scanning lots of candidates
- To find frequent itemset $i_1 i_2 \ldots i_{100}$
  - # of scans: 100
  - # of Candidates: $\binom{100}{1} + \binom{100}{2} + \ldots + \binom{100}{100} = 2^{100}-1 = 1.27 * 10^{30}$!
- Bottleneck: candidate-generation-and-test
- Can we avoid candidate generation?
  - An interesting method in this attempt is called **frequent-pattern growth**

FP-growth

- **FP-growth**
  - finding frequent itemsets without candidate generation
  - adopts a divide-and-conquer strategy.
- **First**, FP-growth compresses the database representing frequent items into a frequent-pattern tree, or **FP-tree**,
  - which retains the itemset association information.
- **Then**, It divides the compressed database into a set of conditional databases (a special kind of projected database),
  - each associated with one frequent item or "pattern fragment," and mines each such database separately.

Example: FP-growth

- Transactional data for an *AllElectronics*
- the minimum support count required is 2, that is, *min_sup = 2.*

TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Example: FP-growth

1. Scan DB once, find frequent 1-itemset (same as Apriori)
2. Sort frequent items in frequency descending order, denoted *L.*
   – L = {{I2: 7}, {I1: 6}, {I3: 6}, {I4: 2}, {I5: 2}}.
3. Construct FP-tree

Constructing FP-tree

- Constructing FP-tree
  - First, create the root of the tree, labeled with “null.”
  - Scan database *D* a second time.
  - The items in each transaction are processed in *L* order, and a branch is created for each transaction.

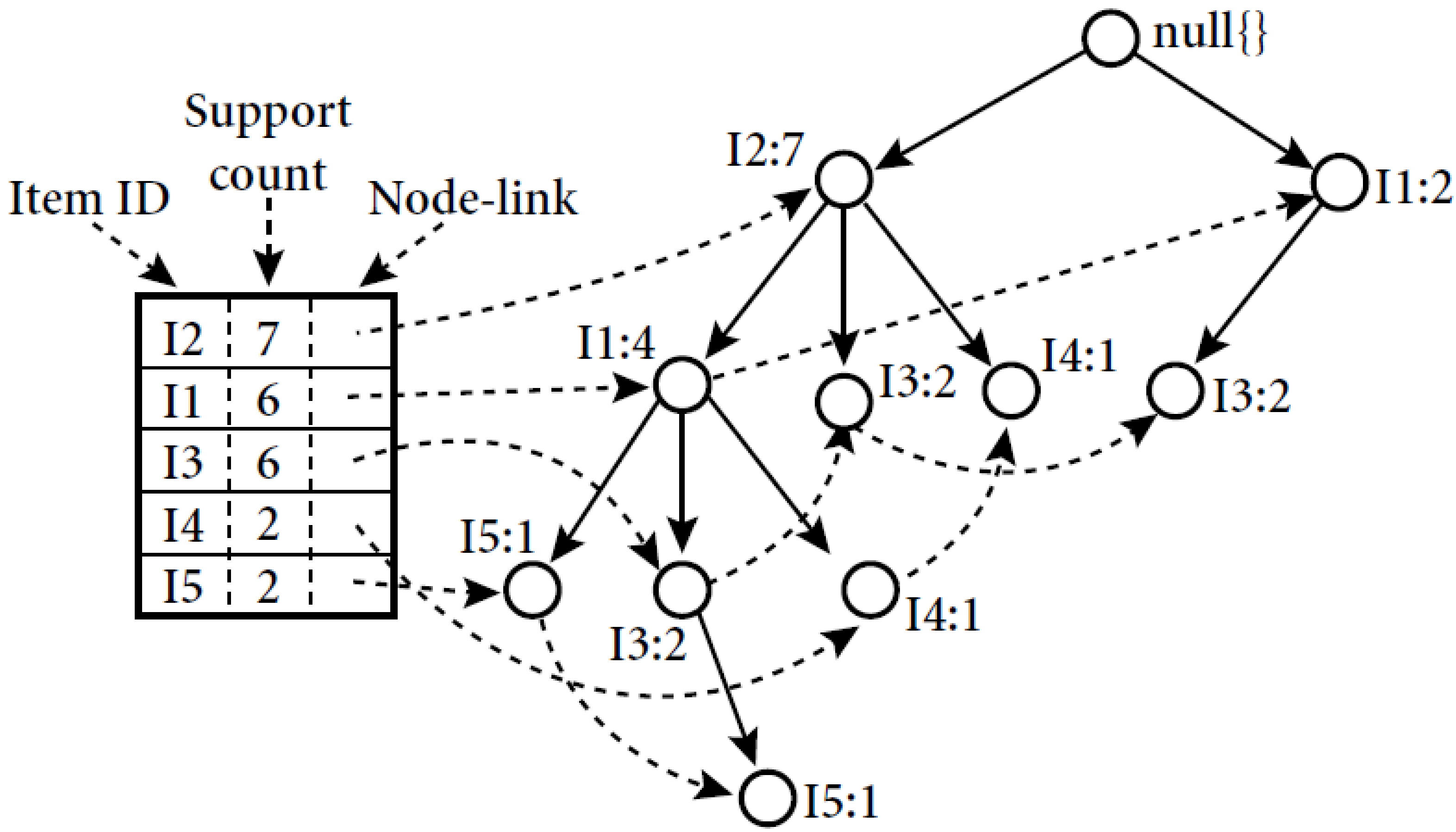
Constructing FP-tree

- Scanning the transactions:
  - The first transaction,
    - "T100: I1, I2, I5," which contains three items (I2, I1, I5 in *L order),* leads to the construction of the first branch of the tree with three nodes, (I2: 1), (I1:1), and (I5: 1), where I2 is linked as a child of the root, I1 is linked to I2, and I5 is linked to I1.
  - The second transaction,
    - T200, contains the items I2 and I4 in *L* order, which would result in a branch where *I2* is linked to the root and I4 is linked to I2.
    - This branch would share a common prefix, I2, with the existing path for T100.
    - We increment the count of the I2 node by 1, and create a new node, (I4: 1), which is linked as a child of (I2: 2).

Constructing FP-tree

- When considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly.
- To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of **node-links**.

FP-tree



Item ID
Support count
Node-link
I2 7
I1 6
I3 6
I4 2
I5 2
null{}
I2:7
I1:2
I1:4
I3:2
I4:1
I3:2
I5:1
I3:2
I4:1
I5:1

Benefits of the FP-tree Structure

- Completeness
  - Preserve complete information for frequent pattern mining
  - Never break a long pattern of any transaction
- Compactness
  - Reduce irrelevant info—infrequent items are gone
  - Items in frequency descending order: the more frequently occurring, the more likely to be shared
  - Never be larger than the original database (not count node-links and the *count* field)

Mining FP-tree

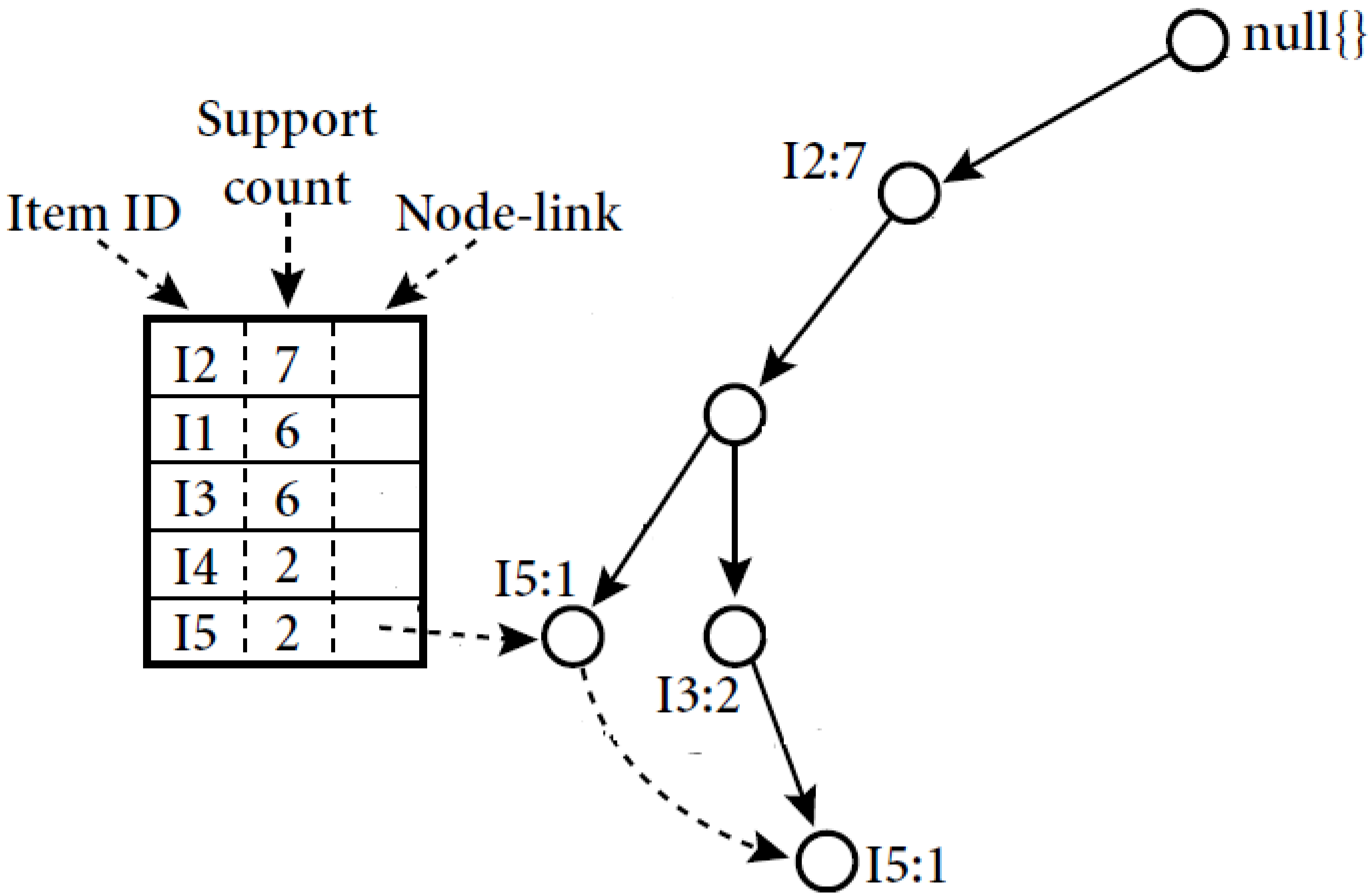
- Now we should mine the FP-tree.
  - Start from each frequent length-1 pattern as an initial **suffix pattern**,
  - construct its **conditional pattern base** (which consists of the set of prefix paths in the FP-tree co-occurring with the suffix pattern),
  - then construct its **conditional FP-tree**, and perform mining recursively on such a tree.
  - The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Example: Mining FP-tree

- First consider I5, which is the last item in *L*
  - I5 occurs in two branches of the FP-tree, the paths (I2, I1, I5: 1) and (I2, I1, I3, I5: 1).
  - Considering I5 as a suffix, its corresponding two prefix paths are (I2, I1: 1) and (I2, I1, I3: 1), which form its **conditional pattern base**
  - Its **conditional FP-tree** contains only a single path, (I2: 2, I1: 2); I3 is not included because its support count of 1 is less than the minimum support count 2.
  - The single path generates all the combinations of frequent patterns:
    - {I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}.

Example: Mining FP-tree

- the paths in which I5 occurs



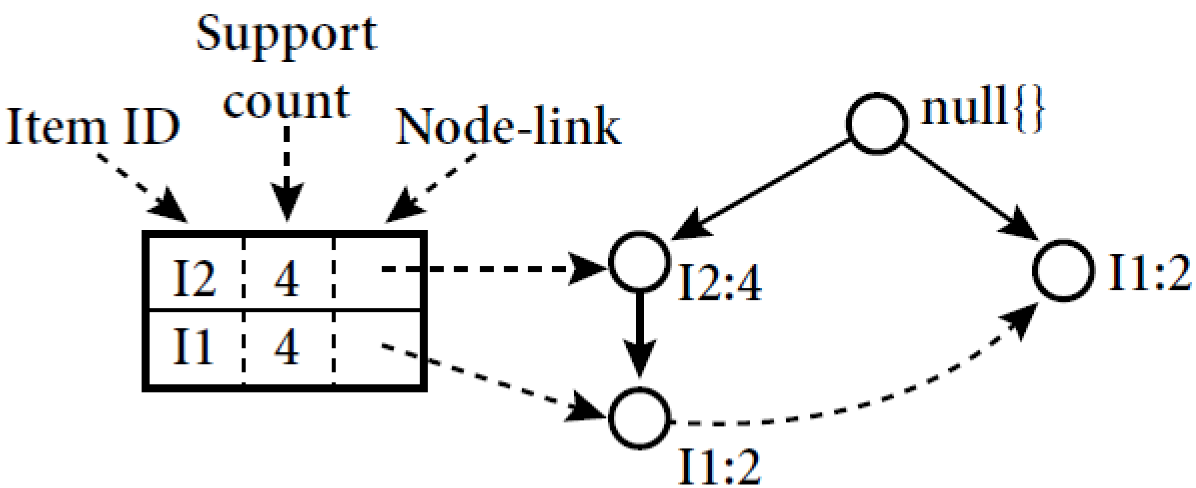
Mining FP-tree

- For I4,
  - its two prefix paths form the conditional pattern base, {{I2 I1: 1}, {I2: 1}},
  - which generates a single-node conditional FP-tree, (I2: 2),
  - and derives one frequent pattern, {I2, I1: 2}.

Mining FP-tree

- For I3
  - conditional pattern base is {{I2, I1: 2}, {I2: 2}, {I1: 2}}.
  - conditional FP-tree has two branches, (I2: 4, I1: 2) and (I1: 2), as shown in the following Figure
  - which generates the set of patterns:

    {{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}}.



Mining FP-tree

- For I1
  - conditional pattern base is {{I2: 4}},
  - whose FP-tree contains only one node, (I2: 4),
  - which generates one frequent pattern, {I2, I1: 4}.

Mining FP-tree

- Summarization of the FP-tree:

Item	*Conditional Pattern Base*	*Conditional FP-tree*	*Frequent Patterns Generated*
I5	{{I2, I1: 1}, {I2, I1, I3: 1}}	⟨I2: 2, I1: 2⟩	{I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}
I4	{{I2, I1: 1}, {I2: 1}}	⟨I2: 2⟩	{I2, I4: 2}
I3	{{I2, I1: 2}, {I2: 2}, {I1: 2}}	⟨I2: 4, I1: 2⟩, ⟨I1: 2⟩	{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}
I1	{{I2: 4}}	⟨I2: 4⟩	{I2, I1: 4}

Why Is FP-Growth the Winner?

- Divide-and-conquer:
  - decompose both the mining task and DB according to the frequent patterns obtained so far
  - leads to focused search of smaller databases
- Other factors
  - no candidate generation, no candidate test
  - compressed database: FP-tree structure
  - no repeated scan of entire database
  - basic ops—counting local freq items and building sub FP-tree, no pattern search and matching

References

References

- J. Han, M. Kamber, **Data Mining: Concepts and Techniques**, Elsevier Inc. (2006). (Chapter 5)

The end