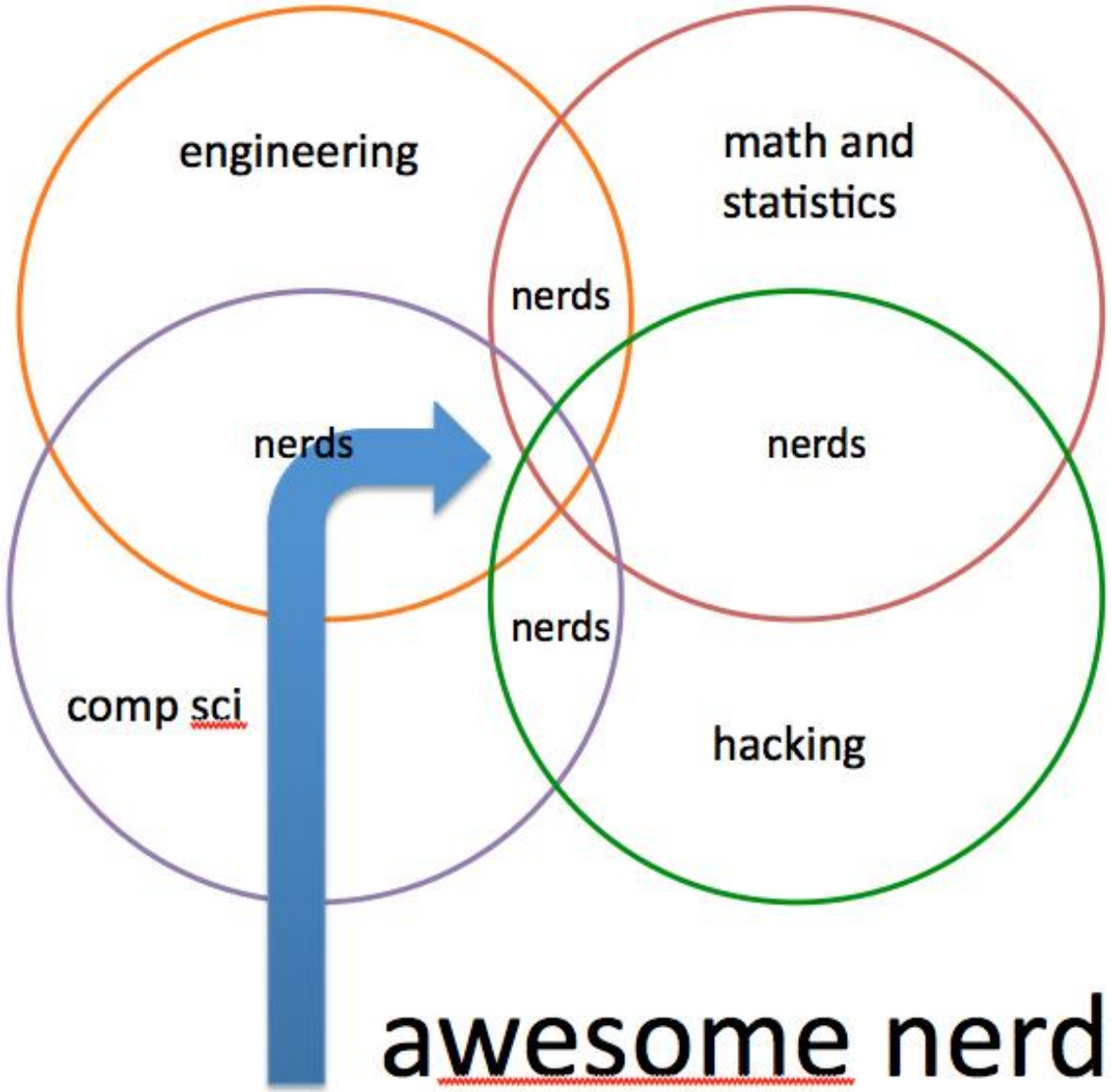
Frequent Pattern Analysis

Data scientists?



awesome nerds

Frequent Pattern

- **Frequent Pattern:** A pattern (a set of items, subsequences, substructures, etc.) that occurs frequently in a data set.
- For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set is a frequent itemset.
- A subsequence, such as buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a shopping history database, is a (frequent) sequential pattern.
- A substructure can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (frequent) structured pattern.

Frequent Itemset Mining Example: Market Basket Analysis

- Suppose, as manager of a retail company, you would like to learn more about the buying habits of your customers. Specifically, you wonder, "*Which groups or sets of items are customers likely to purchase on a given trip to the store?*"
- To answer your question, market basket analysis may be performed on the retail data of customer transactions at your store. You can then use these results to choose marketing strategies and help create a new catalog.

Frequent Pattern Mining Example: Market Basket Analysis

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

Itemsets Discovered:

{Milk,Coke}

{Diaper, Milk}

Rules Discovered:

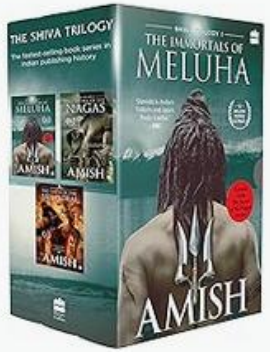
{Milk} --> {Coke}

{Diaper, Milk} --> {Beer}

Frequent Pattern Mining Example: Market Basket Analysis

Frequently bought together

+



+

This item: The Kalki Trilogy (Set of 3 Books) - Avatar of Vishnu; Eye of Brahma; Sword of Shiva

₹729.00

The Shiva Triology Boxset of 3 Books (Perfect Gift for this Festive Season) : The Immortals…

₹777.00

The Ram Chandra Series Boxset: Boxset of 4 Books (Ram - Scion of Ikshvaku, Sita : Warrior of Mithi…

₹966.00

Total price: ₹2,472.00

Add all 3 to Cart

Frequent Pattern Mining Example: Market Basket Analysis

- This process analyzes customer buying habits by finding associations between the different items that customers place in their “shopping baskets”.
- If we think of the universe as the set of items available at the store, then each item has a Boolean variable representing the presence or absence of that item.
- Each basket can then be represented by a Boolean vector of values assigned to these variables.
- The Boolean vectors can be analyzed for buying patterns that reflect items that are frequently associated or purchased together.
- These *patterns* can be represented in the form of *association rules*.

Frequent Pattern Mining Example: Market Basket Analysis

- For example, the information that customers who purchase *milk* also tend to buy *coke* at the same time can be represented using following association rule:

  milk => coke [support = 60%, confidence = 75%]

- **Support and Confidence are two measures of rule interestingness**. They reflect the usefulness and certainity of discovered rules, respectively.
- A support of 60% means that 60% of all the transactions under analysis show that *milk* and *coke* are purchased together.
- A confidence of 75% means that 75% of the customers who purchased a *milk* also bought the *coke*.
- Typically, association rules are considered interesting if they satisfy a *minimum support threshold* ***(min_sup)*** and *a minimum confidence threshold* ***(min_conf)***.

Transaction Data: Supermarket Data

- Concepts:
  - **An Item -** an product in a basket
  - **I -** the set of all items sold in the store
  - **A transaction -** items purchased in a basket; it may have TID (transaction ID)
  - **A transactional dataset:** A set of transactions

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

Definition: Frequent Itemset

- **Itemset**
  - A collection of one or more items
    - Example: {Milk, Bread, Diaper}
  - ***k-itemset***
    - An itemset that contains k items

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

- **Support count or Absolute support**
  - Number of transactions that contain the itemset
  - Example: SupportCount ({*Milk, Bread*}) = 2
- **Support or Relative support**
  - Fraction of transactions that contain an itemset i.e. ***SupportCount / number of Transactions***
- **Frequent Itemset**
  - An itemset whose support is greater than or equal to a ***min_sup*** threshold.

Association Rules

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

Itemsets Discovered:
{Milk,Coke}
{Diaper, Milk}

Rules Discovered:
{Milk} --> {Coke}
{Diaper, Milk} --> {Beer}

Let $I = I_1, I_2, \ldots, I_m$ be an itemset. Let D, the task-relevant data, be a set of database transactions where each transaction T is a nonempty itemset such that $T \subset I$. Each transaction is associated with an identifier, called a TID.

Let A be a set of items. A transaction T is said to contain A if $A \subseteq T$.

- An association rule is an implication of the form $A \implies B$, where $A \subset I, B \subset I, A \neq \phi, B \neq \phi$ and $A \cap B = \phi$.
- The rule $A \implies B$ holds in the transaction set D with **support** s where s is the percentage of transactions in D that contain $A \cup B$.
- The rule $A \implies B$ has **confidence** c in the transaction set D where c is the percentage of transactions in D containing A that also contain B.
- Rules that satisfy both a ***minimum support threshold (min_sup)*** and ***minimum confidence threshold (min_conf)*** are called **strong**.
- By convention, support and confidence values are represented as percentages.

Support and Confidence

- Support can be defined for both an itemset X and an association rule $X \implies Y$ as follows -
  - $Support(X) = \frac{\text{Number of transactions which contains X}}{\text{Total number of transactions}}$
  - $Support(X \implies Y) = \frac{\text{Number of transactions which contains } X \cup Y}{\text{Total number of transactions}}$
    where $X \cup Y$ means transaction should contain each item of itemset X and each item of itemset Y.
- Confidence can be defined for an association $X \implies Y$ as follows -
  - $Confidence(X \implies Y) = \frac{\text{Number of transactions which contains } X \cup Y}{\text{Number of transactions which contains X}} = \frac{support(X \cup Y)}{support(X)} = \frac{support\_count(X \cup Y)}{support\_count(X)}$
- **Goal:** Find all rules that satisfy the user-specified *minimum support threshold (min_sup)* and *minimum confidence threshold (min_conf)*.

FREQUENT ITEMSET: EXAMPLE

- Suppose you are given following dataset, find frequent patterns and association rules.

Tid	Items bought
10	Beer, Nuts, Diaper
20	Beer, Coffee, Diaper
30	Beer, Diaper, Eggs
40	Nuts, Eggs, Milk
50	Nuts, Coffee, Diaper, Eggs, Milk

Let minsup = 50%, minconf = 50%

Frequent Pattterns:

{Beer}:3, {Nuts}:3, {Diaper}:4, {Eggs}:3,

{Beer, Diaper}:3

Association Rules:

- { Beer } → { Diaper } (60%, 100%)
- { Diaper } → { Beer } (60%, 75%)

Association Rule Mining

- In general, association rule mining can be viewed as a two-step process:
  1. **Find all frequent itemsets -** By definition, each of these itemsets will occur at least as frequently as a predetermined minimum support count, *min_sup*
  2. **Generate strong association rules from the frequent itemsets -** By definition, these rules must satisfy minimum support and minimum confidence threshold.
- The overall performance of mining association rules is determined by the first step since the second step is much less costly than the first.

Why use Support and Confidence?

- Support is an important measure because a rule that has very low support may occur simply by chance.
  - A low support rule may be uninteresting from a business perspective because it may not be profitable to promote items that customers seldom buy together.
  - For these reasons, support is often used to eliminate uninteresting rules.
- Confidence measures the reliability of the inference made by a rule.
  - For a given rule X => Y, the higher the confidence, the more likely it is for Y to be present in transactions that contain X.

Frequent Itemset Mining Techniques

EXAMPLE

- Find frequent itemsets and association rules satisfying *min_sup (0.5) and min_conf (0.7)*.

Frequent Itemsets:

1-itemsets: {A} support({A}) = 4/6
{B} support({B}) = 5/6
{C} support({C}) = 3/6

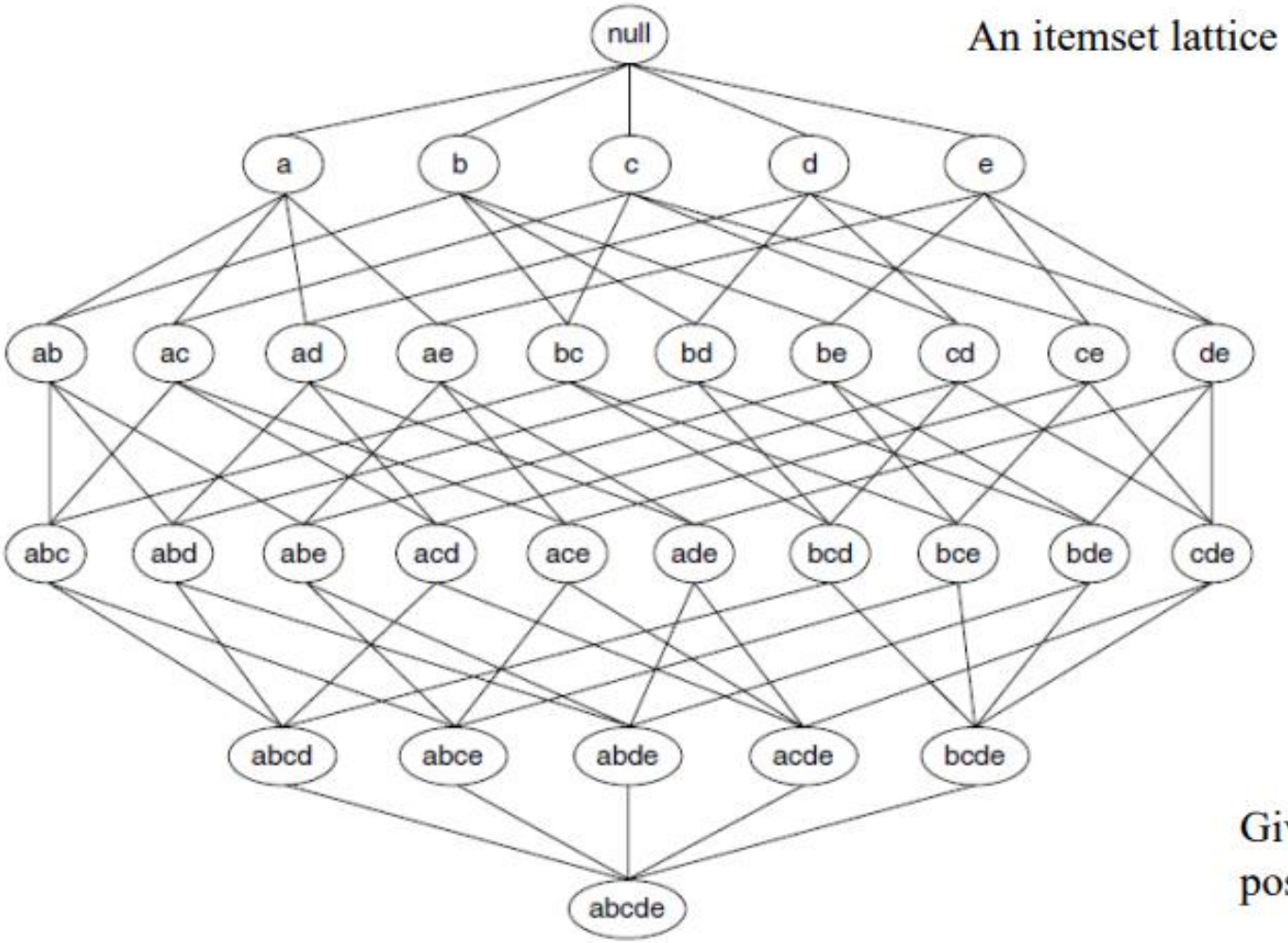
2-itemsets: {A,B} support({A,B}) = 3/6
{B,C} support({B,C}) = 3/6

Association Rules:

A→B conf(A→B) = 3/4
C→B conf(C→B) = 3/3

Transactions
A,B,D
A,B,C,D
A
A,B,C
B,C
B

Frequent Itemset Generation

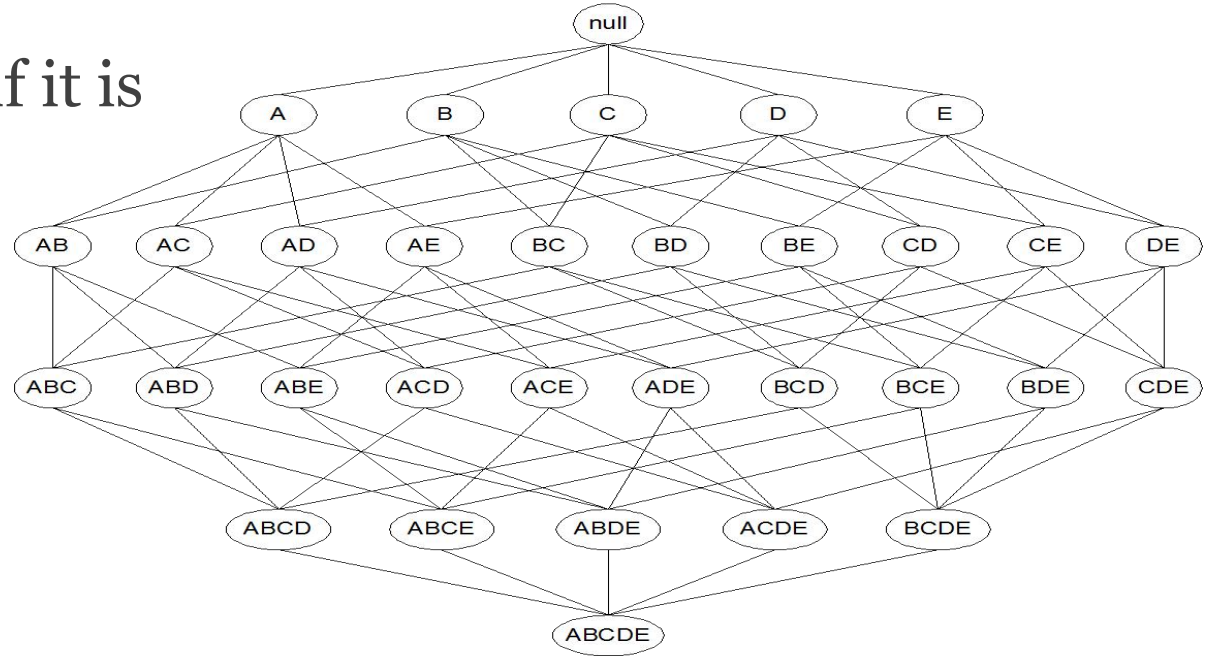


An itemset lattice

Given d items, there are 2^d possible candidate itemsets

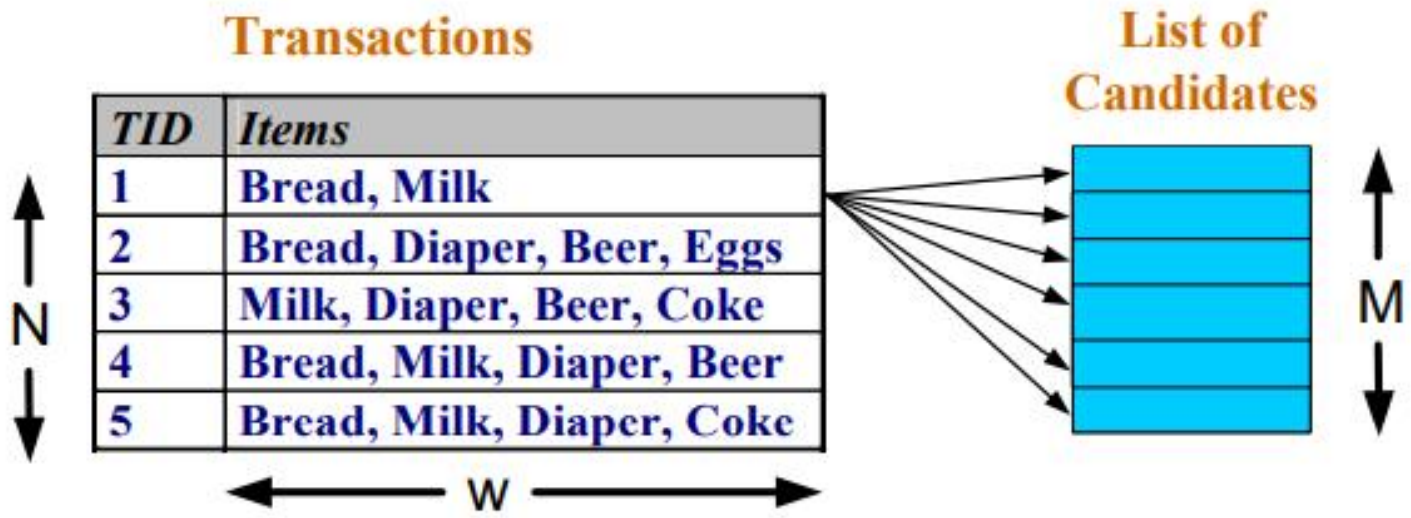
Frequent Itemset Generation: Brute Force Approach

- Try every possible itemset and check if it is frequent.
- How to try the itemsets?
  - Breadth-first search in itemset lattice
  - Depth-first search in itemset lattice
- How to compute the support?
  - For every transaction, check if itemset is included.



Frequent Itemset Generation: Brute Force Approach

- **Brute-force approach:**
  - Each itemset in the lattice is a candidate frequent itemset
  - Count the support of each candidate by scanning the database

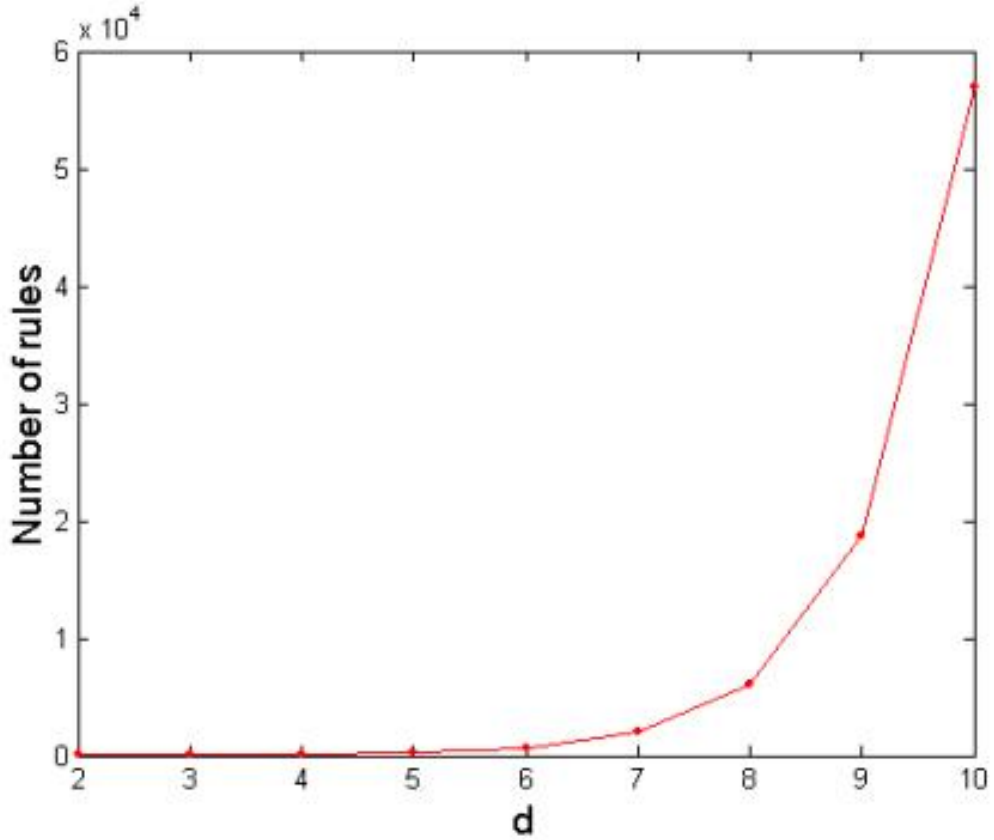


- Match each transaction against every candidate

- Complexity ~ O(NMW)
- Which is bad since $M = 2^d$

Frequent Itemset Generation: Brute Force Approach

- Given d unique items:
  - Total number of itemsets = 2^d
  - Total number of possible association rules:



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

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

If d=6, R = 602 rules

Frequent Itemset Generation Strategies

- Reduce the number of candidates (M)
  - Complete search: $M=2^d$
  - Use pruning techniques to reduce M
  - The **Apriori principle** is an effective way to eliminate some of the candidate itemsets without counting their support values.

- 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 transactions (N)
  - Reduce size of N as the size of itemset increases

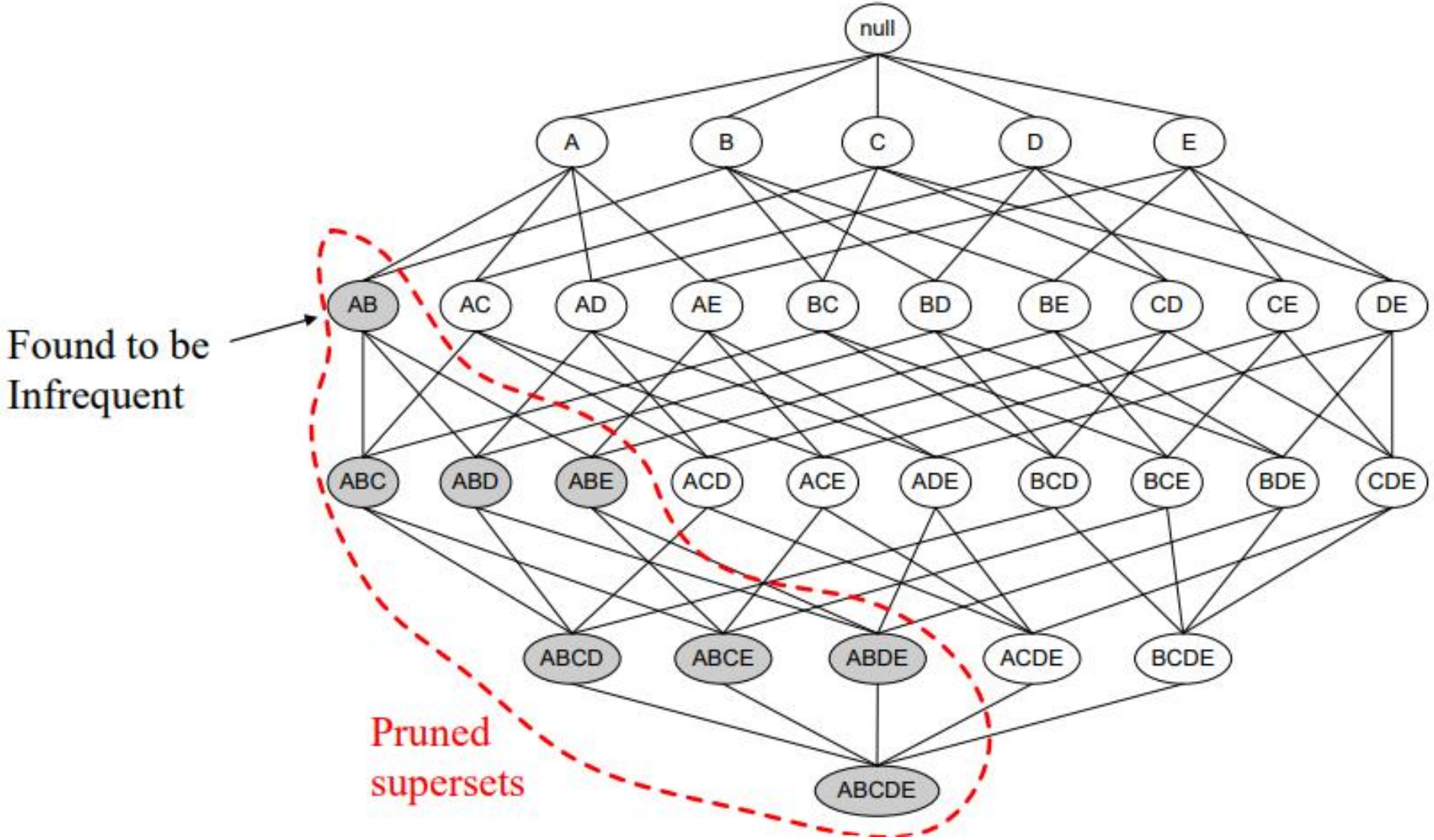
APRIORI ALGORITHM

- **Apriori** algorithm was 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 known as a *level-wise* search, where *k-itemset* are used to explore *(k+1)-itemset*.
- First the set of frequent 1-itemsets is found by scanning the database and counting each item and collecting those items that satisfy minimum support. The resulting set is denoted by L_1.
- Next L1 is used to find L2, the set of frequent 2-itemsets, which is then used to find L3 and so on until no more frequent k-itemsets can be found.
- Finding of each L_k requires one full scan of the database.

APRIORI ALGORITHM: APRIORI PROPERTY

- To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the ***Apriori property*** is used to reduce the search space.
- **Apriori property:** All non-empty subsets of a frequent itemset must also be frequent.
- For example, suppose an itemset *I* does not satisfy min_sup then I is not frequent. Now if an item A is added to I, the resulting itemset (i.e., I ∪ A) cannot occur more frequently than I. Therefore I ∪ A is not frequent either.
- This property belongs to a special category of properties called ***antimonotonicity*** in the sense that if *a set cannot pass a test, all of its supersets will fail the same test as well*. It is called *antimonotonicity* because the property is monotonic in the context of failing a test.

Apriori Algorithm: Apriori Property



APRIORI ALGORITHM STEPS

Given following dataset of transactions, generate all of the frequent itemsets.

Suppose that the minimum support count required is 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

Step 1: Generate all Candidate 1-Itemset C_1

- To generate all candidate 1-itemset, we need to scan entire database and find individual items and their support count.

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

Scan D for count of each candidate →

C_1

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

Step 2: From C_1 create Frequent 1-Itemset L_1

- To create L_1 from C_1, we need to compare support count of each 1-itemset with *minimum support throshold (min_sup)* and select 1-itemsets with support count greater or equal to min_sup

Scan *D* for count of each candidate →

C_1

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

Compare candidate support count with minimum support count →

L_1

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

Step 3: Use L_1 X L_1 to generate C_2

Compare candidate support count with minimum support count

L_1

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

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}

Step 4: From C_2 create Frequent 2-Itemset L_2

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}

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

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

Step 5: Use L_2 X L_2 to generate C_3

Compare candidate support count with minimum support count

L_2

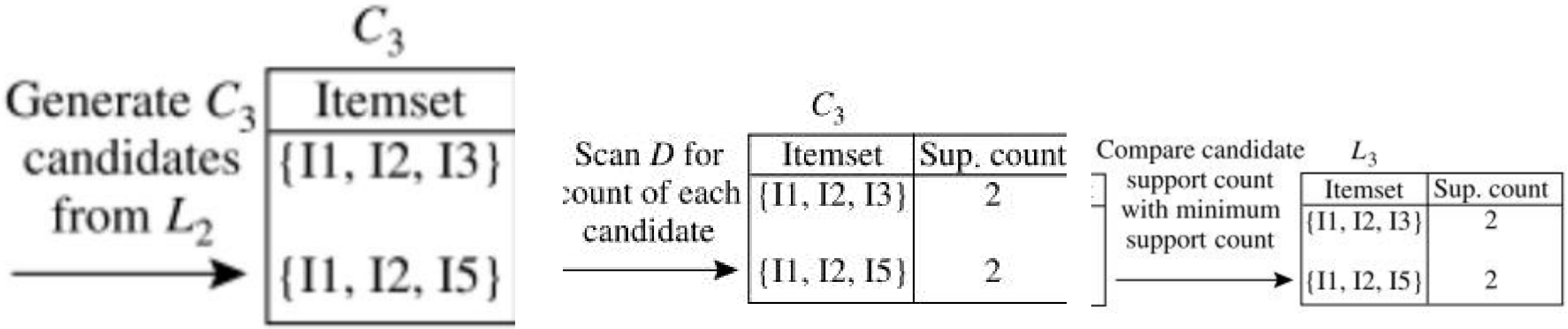
Itemset	Sup. count
{I1, I2}	4
{I1, I3}	4
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2

Generate C_3 candidates from L_2

C_3

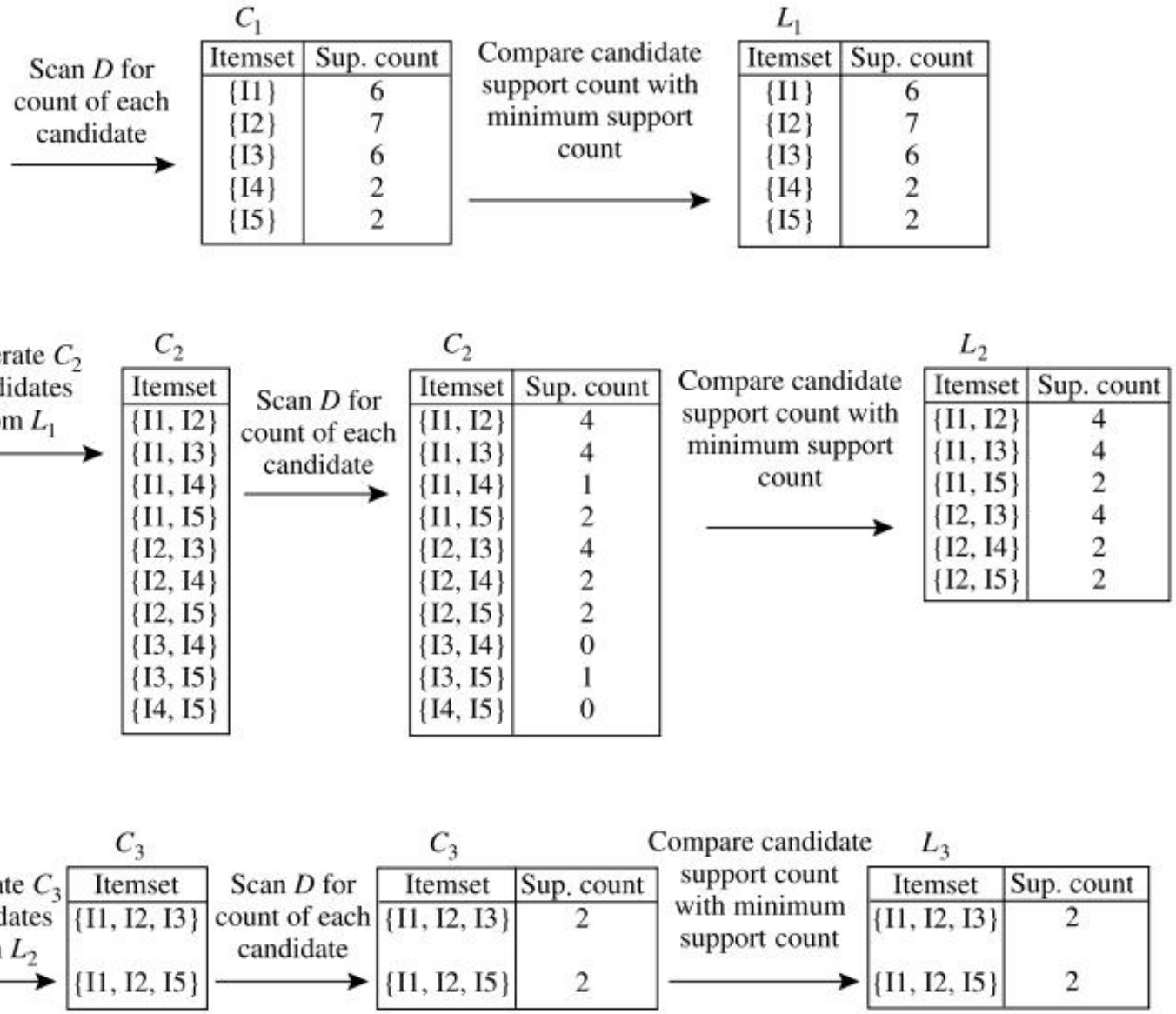
Itemset
{I1, I2, I3}
{I1, I2, I5}

Step 6: Use C_3 to generate Frequent 3-Itemset L_3



APRIORI ALGORITHM STEPS

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



Apriori Algorithm Steps

To generate L_k from L_{k-1} for $k \geq 2$, Apriori algorithm follows a two-step process consisting of **join** and **prune** actions.

1. **The Join Step:**
    a. To find L_k, a set of **candidate** $k-itemsets$ is generated by joining L_{k-1} with itself. This set of candidates is denoted by C_k .
    b. Let l_1 and l_2 be itemsets in L_{k-1}. The notation $l_i[j]$ refers to the j^{th} item in l_i (e.g. $l_1[k-2]$ refers to the second to the last item in l_1). For efficient implementation, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. For the $(k-1)-itemset, l_i$ this means that the items are sorted such that $l_i[1] < l_i[2] < \ldots < l_i[k-1]$.
    c. The join, $L_{k-1} \boxtimes L_{k-1}$ is performed, where members of L_{k-1} are joinable if their first $(k-2)$ items are in common. That is members l_1 and l_2 of L_{k-1} are joined if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \ldots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$ The condition $l_1[k-1] < l_2[k-1]$ is simply to ensure that no duplicates are generated.
    d. The resulting itemset formed by joining l_1 and l_2 is $\{l_1[1], l_1[2], \ldots, l_1[k-2], l_1[k-1], l_2[k-1]\}$.
    e. Similarly all itemsets in L_{k-1} are joined to form C_k.

Apriori Algorithm Steps

2. **The Prune Step:**

a. Candidate set C_k generated by the join step is superset of L_k which contains all of the frequent $k-itemsets$. But it might also contain some itemsets which are not frequent.

b. A database scan is done to determine the count of each candidate in C_k and the candidates which do not support **min_sup** are removed from C_k.

c. Remaining all candidates are frequent, therefore givings us L_k.

d. However C_k can be huge, and so this above process of elimination can involve heavy computation.

e. To reduce the size of C_k, the Apriori property is used which is Any $(k-1)-itemset$ that is not frequent cannot be a subset of a frequent $k-itemset$.

f. Hence if any $(k-1)-subset$ of a candidate $k-itemset$ is not in L_{k-1} then the candidate cannot be frequent either and so can be removed from C_k. This **subset testing** can be done quickly by maintaining a hash tree of all frequent itemsets.

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

- D, a database of transactions;
- min_sup, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

```
(1)   L1 = find_frequent_1-itemsets(D);
(2)   for (k = 2; Lk−1 ≠ φ; k++) {
(3)       Ck = apriori_gen(Lk−1);
(4)       for each transaction t ∈ D { // scan D for counts
(5)           Ct = subset(Ck, t); // get the subsets of t that are candidates
(6)           for each candidate c ∈ Ct
(7)               c.count++;
(8)       }
(9)       Lk = {c ∈ Ck|c.count ≥ min_sup}
(10)  }
(11)  return L = ∪k Lk;

procedure apriori_gen(Lk−1:frequent (k − 1)-itemsets)
(1)   for each itemset l1 ∈ Lk−1
(2)       for each itemset l2 ∈ Lk−1
(3)           if (l1[1] = l2[1]) ∧ (l1[2] = l2[2])
                  ∧... ∧ (l1[k − 2] = l2[k − 2]) ∧ (l1[k − 1] < l2[k − 1]) then {
(4)               c = l1 ⋈ l2; // join step: generate candidates
(5)               if has_infrequent_subset(c, Lk−1) then
(6)                   delete c; // prune step: remove unfruitful candidate
(7)               else add c to Ck;
(8)           }
(9)   return Ck;

procedure has_infrequent_subset(c: candidate k-itemset;
          Lk−1: frequent (k − 1)-itemsets); // use prior knowledge
(1)   for each (k − 1)-subset s of c
(2)       if s ∉ Lk−1 then
(3)           return TRUE;
(4)   return FALSE;
```

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

- D, a database of transactions;
- *min_sup*, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

(1) L_1 = find_frequent_1-itemsets(D);
(2) **for** ($k = 2$; $L_{k-1} \neq \phi$; k++) {
(3) C_k = apriori_gen(L_{k-1});
(4) **for each** transaction $t \in D$ { // scan D for counts
(5) C_t = subset(C_k, t); // get the subsets of t that are candidates
(6) **for each** candidate $c \in C_t$
(7) c.count++;
(8) }
(9) $L_k = \{c \in C_k | c.count \geq min\_sup\}$
(10) }
(11) **return** $L = \cup_k L_k$;

```
procedure apriori_gen(L_{k-1}:frequent (k − 1)-itemsets)
(1)     for each itemset l_1 ∈ L_{k-1}
(2)         for each itemset l_2 ∈ L_{k-1}
(3)             if (l_1[1] = l_2[1]) ∧ (l_1[2] = l_2[2])
                    ∧... ∧ (l_1[k − 2] = l_2[k − 2]) ∧ (l_1[k − 1] < l_2[k − 1]) then {
(4)                 c = l_1 ⋈ l_2; // join step: generate candidates
(5)                 if has_infrequent_subset(c, L_{k-1}) then
(6)                     delete c; // prune step: remove unfruitful candidate
(7)                 else add c to C_k;
(8)             }
(9)     return C_k;

procedure has_infrequent_subset(c: candidate k-itemset;
          L_{k-1}: frequent (k − 1)-itemsets); // use prior knowledge
(1)     for each (k − 1)-subset s of c
(2)         if s ∉ L_{k-1} then
(3)             return TRUE;
(4)     return FALSE;
```

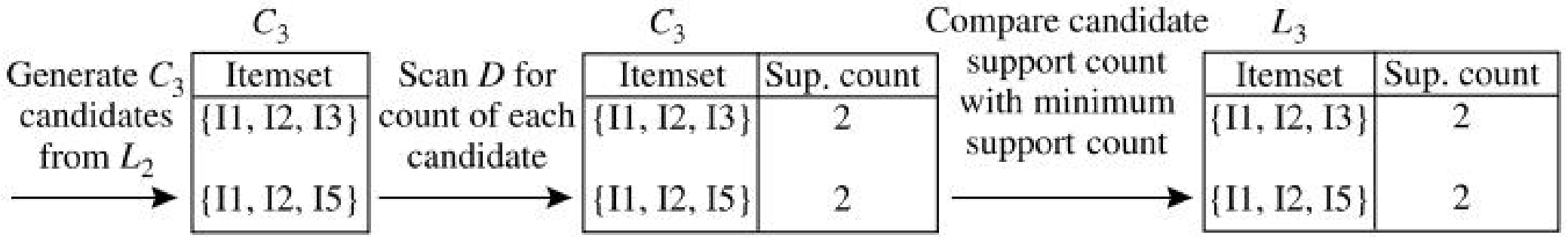
Generating Association rules from Frequent Itemsets

- 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).

Association rules can be generated as follows -

1. For each frequent itemset l, generate all non-empty subsets of l.
2. For every non-empty subset s of l, generate the rule "$s \implies (l - s)$" if $\frac{support\_count(l)}{support\_count(s)} \geq min\_conf$ where min_conf is the minimum confidence threshold. Because we know that, $confidence(A \implies B) = \frac{support\_count(A \cup B)}{support\_count(A)}$.

GENERATING ASSOCIATION RULES



- The data contain frequent itemset X = {I1, I2, I5}. What are the association rules that can be generated from X?
  - The non-empty subsets of X are {I1}, {I2}, {I5}, {I1, I2}, {I1, I5}, {I2, I5}
  - Resulting association rules are -

$\{I1, I2\} \Rightarrow I5$, $confidence = 2/4 = 50\%$
$\{I1, I5\} \Rightarrow I2$, $confidence = 2/2 = 100\%$
$\{I2, I5\} \Rightarrow I1$, $confidence = 2/2 = 100\%$
$I1 \Rightarrow \{I2, I5\}$, $confidence = 2/6 = 33\%$
$I2 \Rightarrow \{I1, I5\}$, $confidence = 2/7 = 29\%$
$I5 \Rightarrow \{I1, I2\}$, $confidence = 2/2 = 100\%$

If the minimum confidence threshold is, say, 70%, then only the second, third, and last rules are output, because these are the only ones generated that are strong.

EXERCISE

- Find all association rules for given dataset of transactions. Assume 0.3 for the minimum support value.

Trans ID	Items Purchased
101	Milk, bread, eggs
102	Milk, Juice
103	Juice, butter
104	Milk, bread, eggs
105	Coffee, eggs
106	Coffee
107	Coffee, Juice
108	Milk, bread, eggs, cookies
109	Cookies, butter
110	Milk, bread

Improving the Efficiency of Apriori: Hash-based Technique

- A hash-based technique can be used to reduce the size of the candidate *k-itemset*, C_k for k > 1.
- For example, when scanning each transaction (e.g., let t = {i1, i2, i4}) in the database to generate the frequent 1-itemsets, L1, we can generate all the 2-itemsets for each transaction (e.g., three 2-itemsets {i1, i2}, {i1, i4}, and {i2, i4} for transaction t), hash them into the different buckets of a hash table structure, and increase the corresponding bucket counts.
- A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set. Such a hash-based technique may substantially reduce the number of candidate k-itemsets examined (especially when k = 2).

HASH-BASED TECHNIQUE

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

Create hash table H_2 using hash function $h(x, y) = ((order\ of\ x) \times 10 + (order\ of\ y))\ mod\ 7$

H_2

bucket address	0	1	2	3	4	5	6
bucket count	2	2	4	2	2	4	4
bucket contents	{I1, I4} {I3, I5}	{I1, I5} {I1, I5}	{I2, I3} {I2, I3} {I2, I3} {I2, I3}	{I2, I4} {I2, I4}	{I2, I5} {I2, I5}	{I1, I2} {I1, I2} {I1, I2} {I1, I2}	{I1, I3} {I1, I3} {I1, I3} {I1, I3}

Hash table, H2, for candidate 2-itemsets. This hash table was generated by scanning the given transactions while determining L1. If the minimum support count is, say, 3, then the itemsets in buckets 0, 1, 3, and 4 cannot be frequent and so they should not be included in C2.

Improving the Efficiency of Apriori: Transaction Reduction

- A transaction that does not contain any frequent *k-itemsets* cannot contain any frequent *(k + 1)-itemsets*.
- Therefore such a transaction can be marked or removed from further consideration because subsequent database scans for *j-itemsets*, where $j>k$, will not need to consider such a transaction.

Improving the Efficiency of Apriori: Sampling

- The basic idea of the sampling approach is to pick a random sample S of the given data D and then search for the frequent itemsets in S instead of D.
- In this way, we trade off some degree of accuracy against efficiency. Because we are searching for frequent itemsets in S rather than in D, it is possible that we will miss some of the global frequent itemsets.
- To reduce this possibility, we use a *lower support threshold than the minimum support* to find the frequent itemsets local to *S (denoted* L_S*)*. The rest of the database is then used to compute the actual frequencies of each itemset in L_S.
- The sampling approach is especially beneficial when efficiency is of utmost importance such as in computationally intensive applications that must be run frequently.

Problems with Apriori Algorithm

- As we have seen, In many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However, it can suffer from two nontrivial costs.
  - *It may still need to generate a huge number of candidate sets*. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
  - It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets.
- "Can we design a method that mines the complete set of frequent itemsets without such a costly candidate generation process?"

Frequent Pattern Growth (FP-Growth) Algorithm

- FP-growth algorithm takes a different approach to discovering frequent itemsets.
  - FP-growth does not subscribe to the generate-and-test paradigm of Apriori algorithm.
- FP-growth algorithm encodes the data set using a compact data structure called an FP-tree and extracts frequent itemsets directly from this structure.
  - Uses a compressed representation of the database using an FP-tree.
  - Once an FP-tree has been constructed, it uses a recursive divide-and-conquer approach to mine the frequent itemsets.

FP-Tree Construction

- An FP-tree is a compressed representation of the input data.
- It is constructed by reading the data set one transaction at a time and mapping each transaction onto a path in the FP-tree.
  - Different transactions can have several items in common, their paths may overlap.
  - The more the paths overlap with one another, the more compression we can achieve using the FP-tree structure.

FP-Tree Construction

- Each node in the tree contains the label of an item along with a counter that shows the number of transactions mapped onto the given path.
  - Initially, the FP-tree contains only the root node represented by the null symbol.
  - Every transaction maps onto one of the paths in the FP-tree.
- The size of an FP-tree is typically smaller than the size of the uncompressed data because many transactions in market basket data often share a few items in common.
  - best-case scenario, all transactions have same set of items
    - FP-tree contains only a single branch
  - Worst-case scenario happens when every transaction has a unique set of items
    - FP-tree is effectively the same as the size of original data
- Physical storage requirement for FP-tree is higher because it requires additional space to store pointers between nodes and counters for each item.

FP-Growth Algorithm

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

FP-Growth Algorithm

Step 1: Calculate the support count of each 1- itemset

$\{I_1\} \rightarrow 6$

$\{I_2\} \rightarrow 7$

$\{I_3\} \rightarrow 6$

$\{I_4\} \rightarrow 2$

$\{I_5\} \rightarrow 2$

Step 2: Order items in each transaction based on decreasing value of support count $I_2 > I_1 > I_3 > I_4 > I_5$

$T_{100} \rightarrow I_2, I_1, I_5$

$T_{200} \rightarrow I_2, I_4$

$T_{300} \rightarrow I_2, I_3$

$T_{400} \rightarrow I_2, I_1, I_4$

$T_{500} \rightarrow I_1, I_3$

$T_{600} \rightarrow I_2, I_3$

$T_{700} \rightarrow I_1, I_3$

$T_{800} \rightarrow I_2, I_1, I_3, I_5$

$T_{900} \rightarrow I_2, I_1, I_3$

FP-Growth Algorithm

Step 3: Construct FP-Tree for each transaction

$T_{100} \rightarrow I_2, I_1, I_5$

$T_{200} \rightarrow I_2, I_4$

$T_{300} \rightarrow I_2, I_3$

$T_{400} \rightarrow I_2, I_1, I_4$

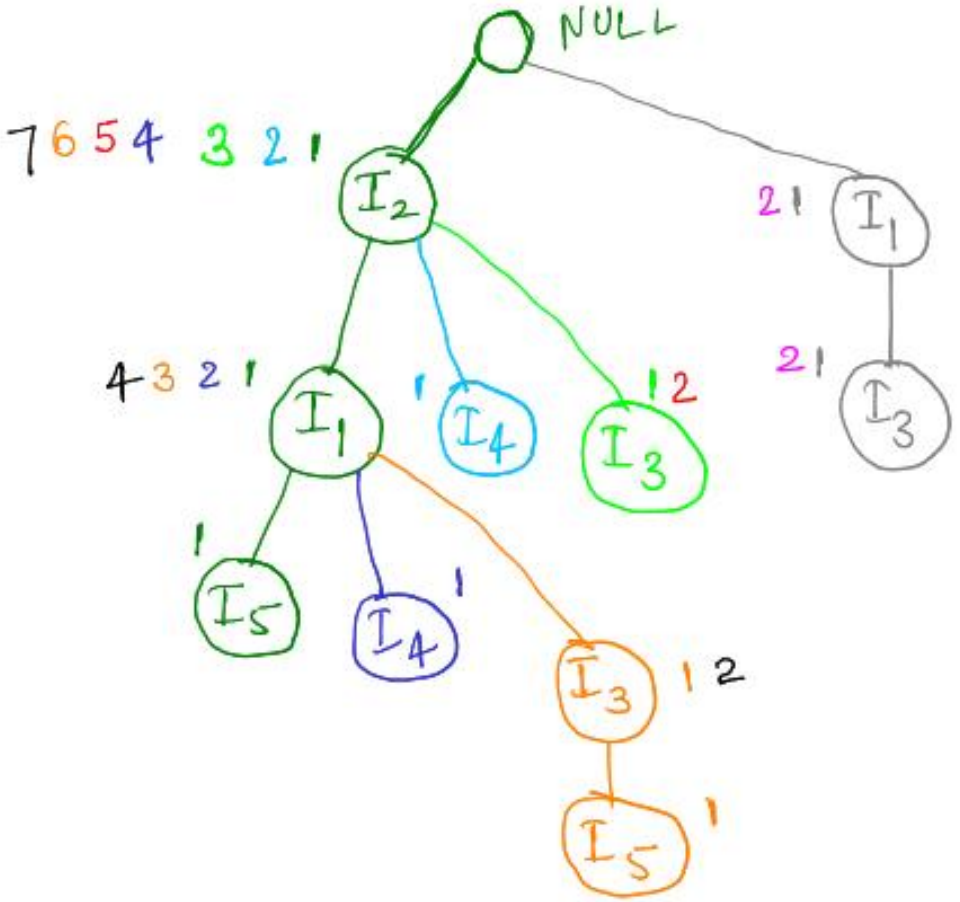
$T_{500} \rightarrow I_1, I_3$

$T_{600} \rightarrow I_2, I_3$

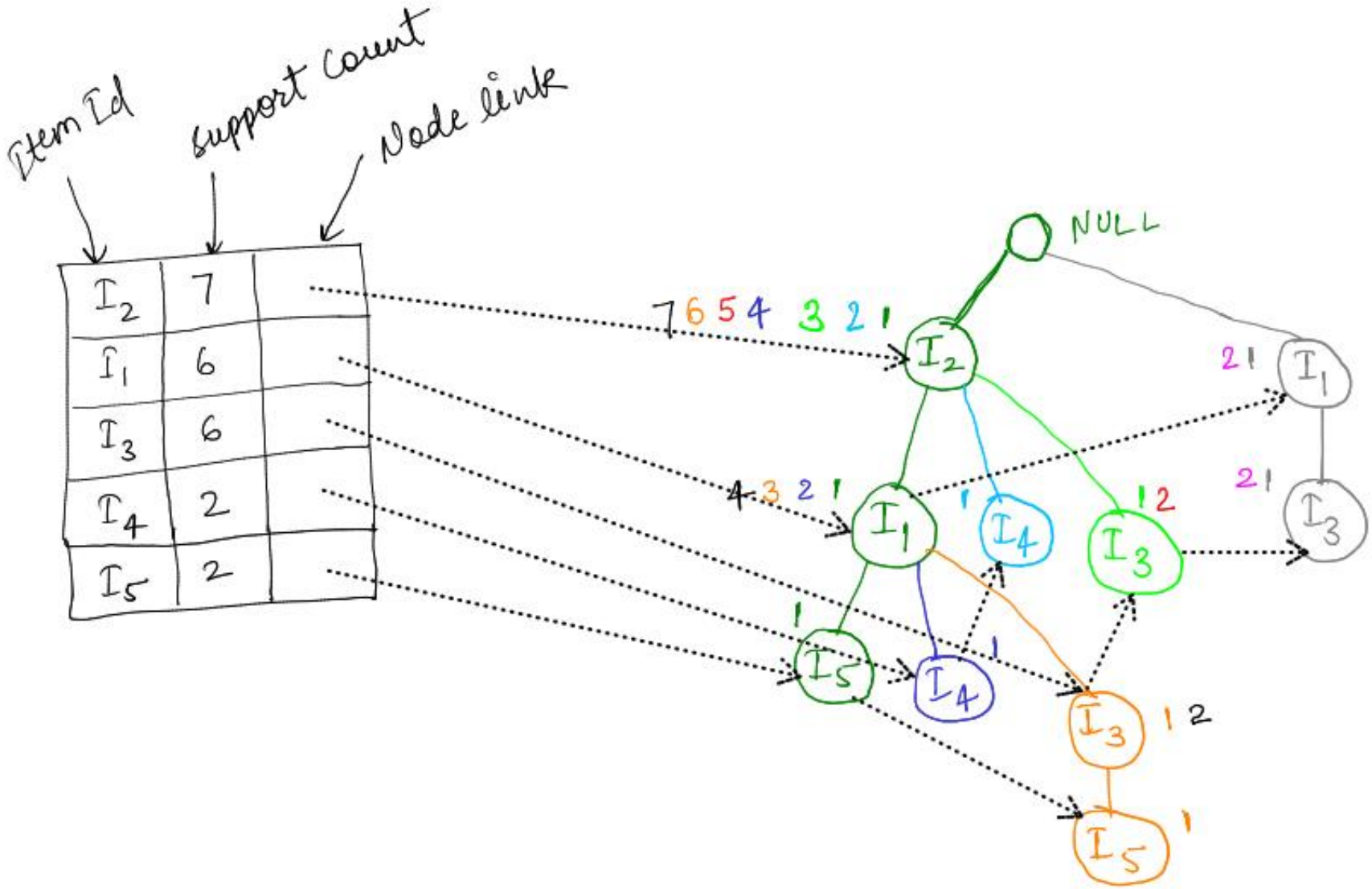
$T_{700} \rightarrow I_1, I_3$

$T_{800} \rightarrow I_2, I_1, I_3, I_5$

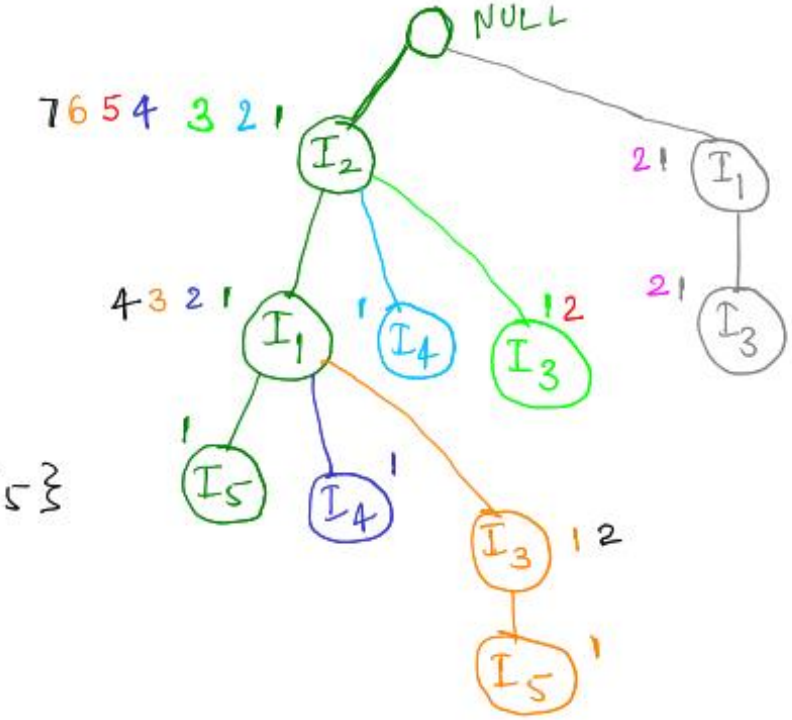
$T_{900} \rightarrow I_2, I_1, I_3$



FP-Growth Algorithm



Step 4: For each I-itemset find all the paths leading to that in FP-Tree and Generate conditional FP-tree

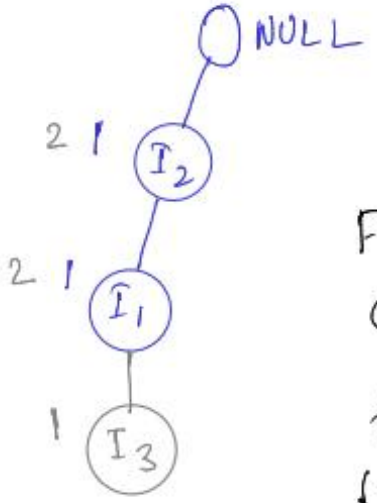


For $\{I_5\}$: Two paths

$\{I_2, I_1\}: 1$

$\{I_2, I_1, I_3\}: 1$

Conditional FP-Tree for $\{I_5\}$



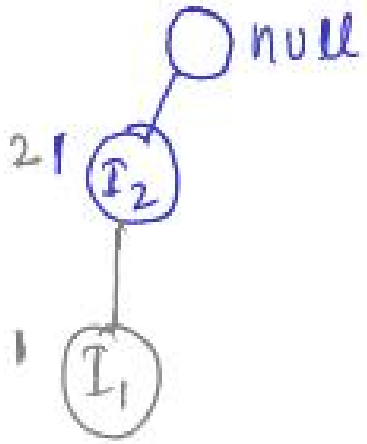
From conditional FP-Tree for $\{I_5\}$, we can say that only $\{I_2\}$ and $\{I_1\}$ are frequent items which can pair-up with $\{I_5\}$ so frequent patterns which can include $\{I_5\}$ are —

$\{I_1, I_5\}: 2$, $\{I_2, I_5\}: 2$, $\{I_2, I_1, I_5\}: 2$

For $\{I_4\}$: Two Paths

$\{I_2\} : 1$, $\{I_2, I_1\} : 1$

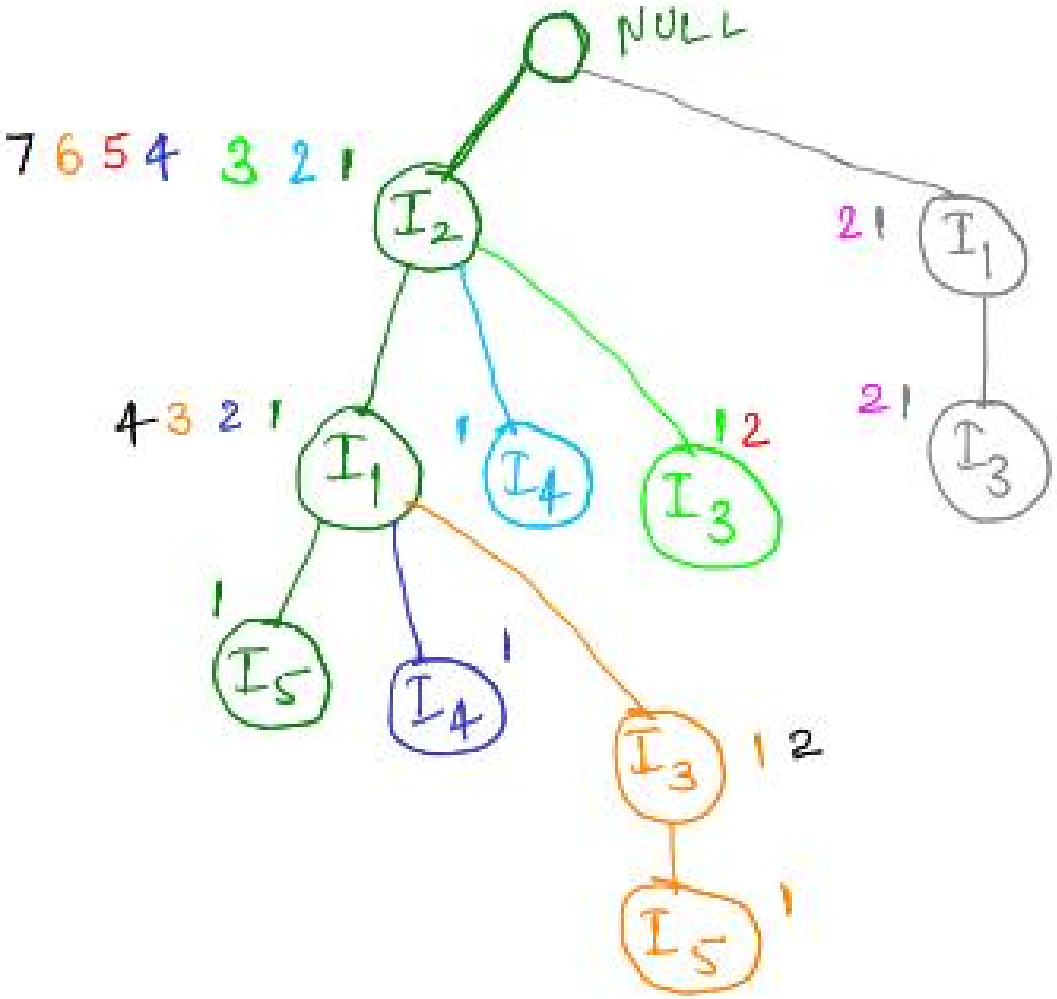
Conditional FP-Tree



only $\{I_2\}$ can create frequent itemset with $\{I_4\}$.

Frequent itemset which include $\{I_4\}$ are -

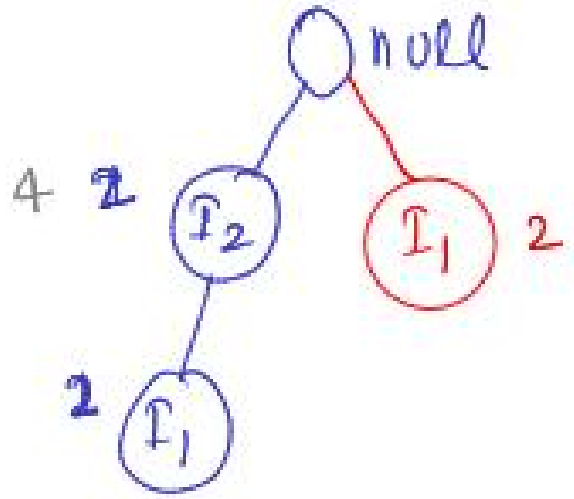
$\{I_2, I_4\} : 2$

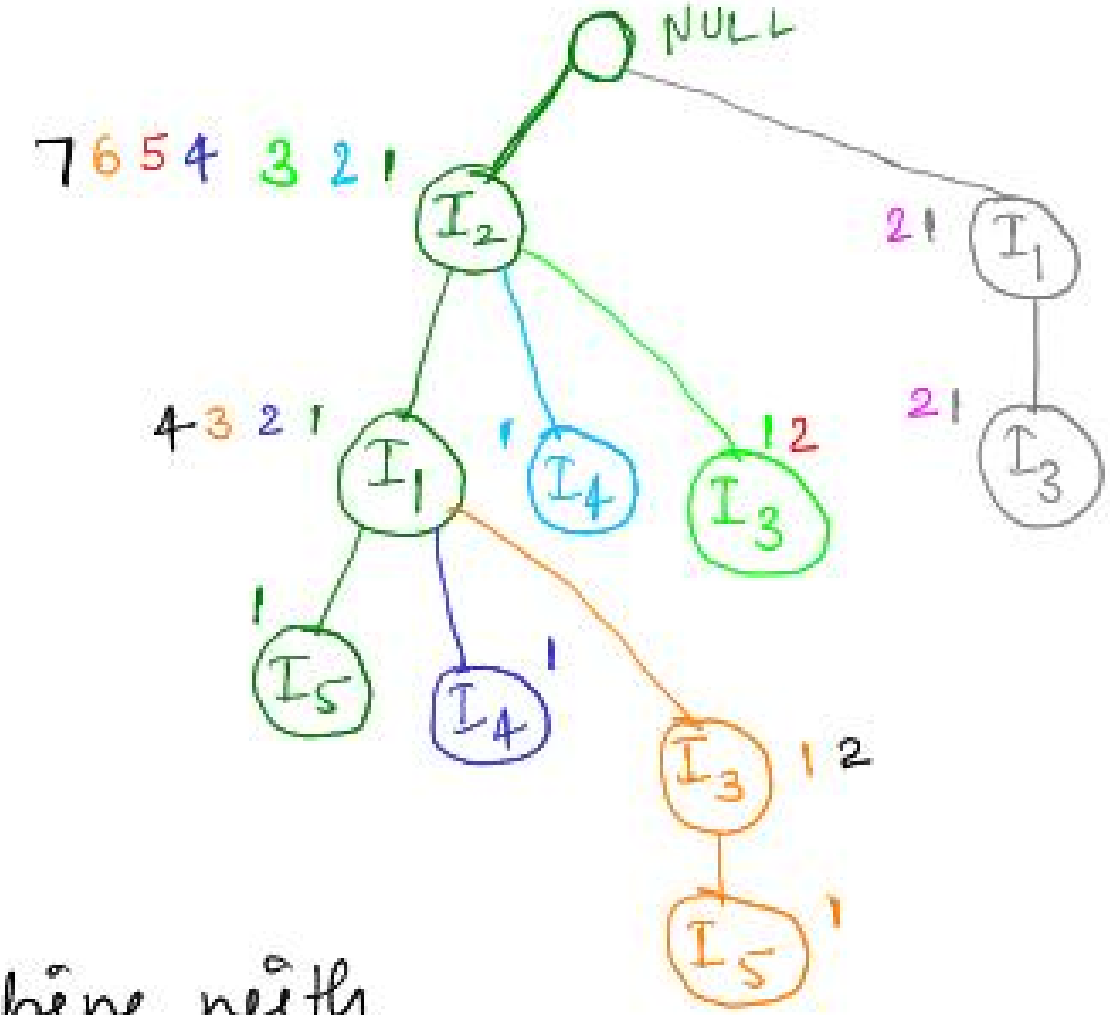


For {I3} : Three paths

{I2, I1}: 2, {I2}: 2, {I1}: 2

So Conditional FP-Tree



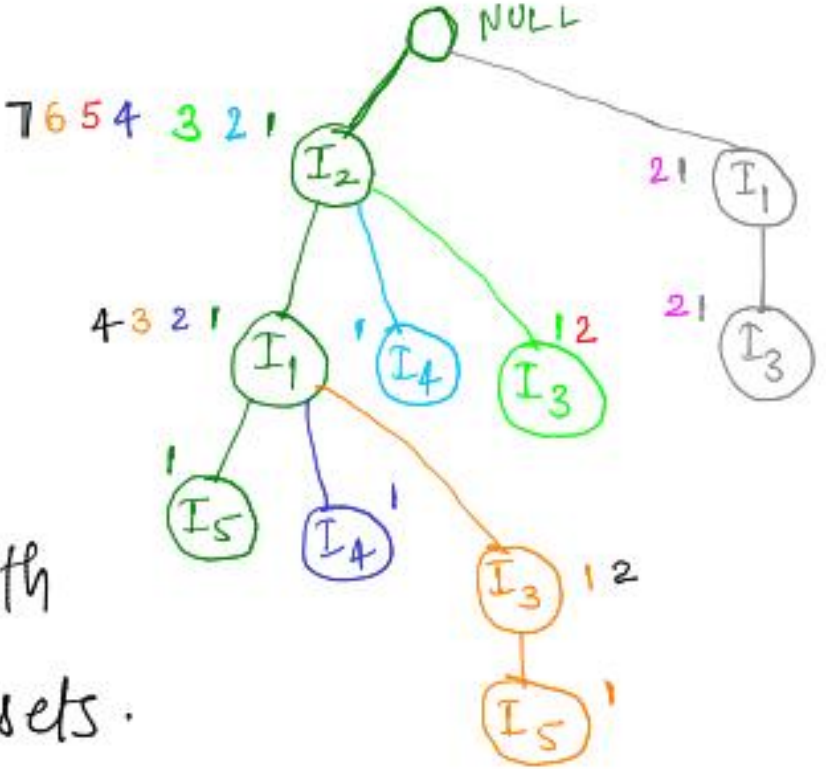


So frequent patterns which can combine with {I3} are

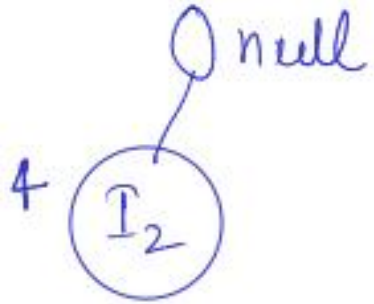
{I2, I1, I3}: 2 , {I2 I3}: 4 , {I1, I3}: 4

For $\{I_1\}$: Two paths

$\{I_2\}$: 4 second path only contains NULL node. We can discard it.



Conditional FP-tree



So only $\{I_2\}$ can pair up with $\{I_1\}$ to make frequent itemsets.

Frequent itemsets are –

$\{I_2, I_1\}$: 4

FP-Growth Algorithm

Table 4.2 Mining the FP-tree by creating conditional (sub-)pattern bases.

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}

Algorithm: **FP_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- D, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.
Method:

1. The FP-tree is constructed in the following steps:
   a. Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the *list* of frequent items.
   b. Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in D do the following. Select and sort the frequent items in *Trans* according to the order of L. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, α)
(1)   if Tree contains a single path P then
(2)     for each combination (denoted as β) of the nodes in the path P
(3)       generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)   else for each a_i in the header of Tree {
(5)     generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)     construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)     if Tree_β ≠ ∅ then
(8)       call FP_growth(Tree_β, β); }
```

Algorithm: **FP_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- D, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.
Method:

1. The FP-tree is constructed in the following steps:
   a. Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the *list* of frequent items.
   b. Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in D do the following. Select and sort the frequent items in *Trans* according to the order of L. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that *N.item-name* = *p.item-name*, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, α)
(1)   if Tree contains a single path P then
(2)     for each combination (denoted as β) of the nodes in the path P
(3)       generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)   else for each a_i in the header of Tree {
(5)     generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)     construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)     if Tree_β ≠ ∅ then
(8)       call FP_growth(Tree_β, β); }
```

Transactions
A,B,D
A,B,C,D
A
A,B,C
B,C
B

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

itemset	TID_set
I1	{T100, T400, T500, T700, T800, T900}
I2	{T100, T200, T300, T400, T600, T800, T900}
I3	{T300, T500, T600, T700, T800, T900}
I4	{T200, T400}
I5	{T100, T800}

MINING FREQUENT ITEMSETS USING THE VERTICAL DATA FORMAT

- Both the Apriori and FP-growth methods mine frequent patterns from a set of transactions in ***TID-itemset format (i.e., {TID : itemset})***, where TID is a transaction ID and itemset is the set of items bought in transaction TID. This is known as the *horizontal data format*.
- Alternatively, data can be presented in ***item-TID_set format (i.e., {item : TID_set})***, where item is an item name, and TID_set is the set of transaction identifiers containing the item. This is known as the *vertical data format*.

Table 4.1 A transactional data set.

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

Table 4.3 The vertical data format of the transaction data set *D* of Table 4.1.

itemset	TID_set
I1	{T100, T400, T500, T700, T800, T900}
I2	{T100, T200, T300, T400, T600, T800, T900}
I3	{T300, T500, T600, T700, T800, T900}
I4	{T200, T400}
I5	{T100, T800}

MINING FREQUENT ITEMSETS USING THE VERTICAL DATA FORMAT

itemset	TID_set
I1	{T100, T400, T500, T700, T800, T900}
I2	{T100, T200, T300, T400, T600, T800, T900}
I3	{T300, T500, T600, T700, T800, T900}
I4	{T200, T400}
I5	{T100, T800}

itemset	TID_set
{I1, I2}	{T100, T400, T800, T900}
{I1, I3}	{T500, T700, T800, T900}
{I1, I4}	{T400}
{I1, I5}	{T100, T800}
{I2, I3}	{T300, T600, T800, T900}
{I2, I4}	{T200, T400}
{I2, I5}	{T100, T800}
{I3, I5}	{T800}

itemset	TID_set
{I1, I2, I3}	{T800, T900}
{I1, I2, I5}	{T100, T800}

Mining Frequent Itemsets using the Vertical Data Format

- First, we transform the horizontally formatted data into the vertical format by scanning the data set once. The support count of an itemset is simply the length of the TID_set of the itemset.
- Starting with k = 1, the frequent *k-itemsets* can be used to construct the candidate ***(k + 1)-itemsets*** based on the Apriori property. The computation is done by intersection of the TID_sets of the frequent k-itemsets to compute the TID_sets of the corresponding ***(k + 1)-itemsets***. This process repeats, with **k** incremented by 1 each time, until no frequent itemsets or candidate itemsets can be found.
- Besides taking advantage of the Apriori property in the generation of candidate (k + 1)-itemset from frequent k-itemsets, another merit of this method is that there is no need to scan the database to find the support of (k + 1)-itemsets (for $k \geq 1$). This is because the TID_set of each k-itemset carries the complete information required for counting such support.
- The TID_sets can be quite long, taking substantial memory space as well as computation time for intersecting the long sets.

Mining Frequent Itemsets using the Vertical Data Format

- To further reduce the cost of registering long TID_sets, as well as the subsequent costs of intersections, we can use a technique called *diffset*, which keeps track of only the differences of the TID_sets of a **(k + 1)-itemset** and a corresponding **k-itemset**.
- For instance, we have *{I1} = {T100, T400, T500, T700, T800, T900} and {I1, I2} = {T100, T400, T800, T900}*. The diffset between the two is ***diffset({I1, I2}, {I1}) = {T500, T700}***. Thus rather than recording the four TIDs that make up the intersection of {I1} and {I2}, we can instead use diffset to record just two TIDs, indicating the difference between {I1} and {I1, I2}.

Compact Representation of Frequent Itemsets

- A major challenge in mining frequent itemsets from a large data set is the fact that such mining often generates a huge number of itemsets satisfying the *minimum support (min_sup) threshold*, especially when *min_sup* is set low.
- This is because if an itemset is frequent, each of its subsets is frequent, each of its subsets is frequent as well. A long itemset will contain a combinatorial number of shorter frequent subitemsets.
- It is useful to identify a small representative set of itemsets from which all other frequent itemsets can be derived.
  - Maximal Frequent Itemsets
  - Closed Frequent Itemsets

Compact Representation of Frequent Itemsets

For example a frequent itemset of length 100, such as $a_1, a_2, \ldots, a_{100}$ will contain -

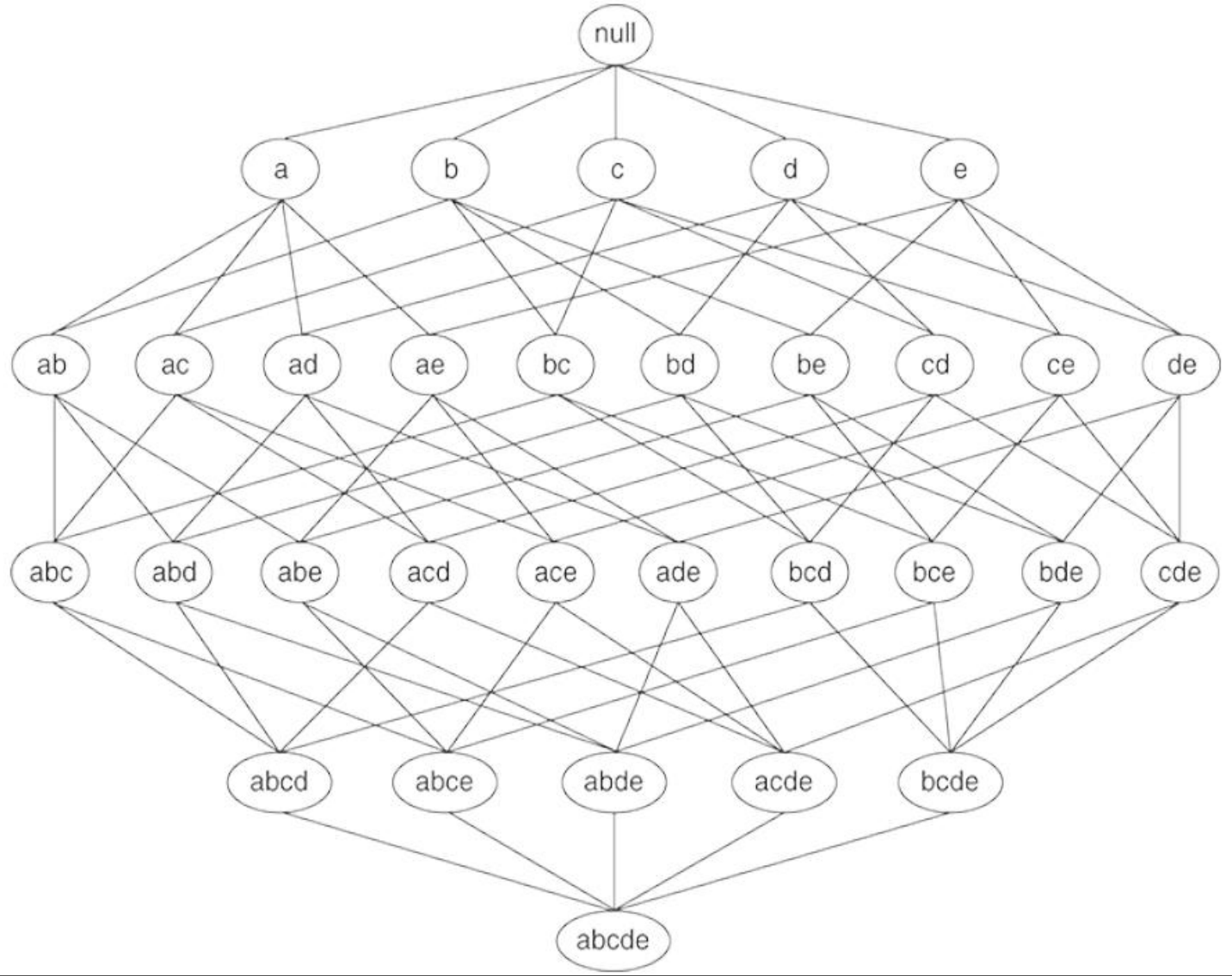
- $\binom{100}{1}$ = 100 frequent 1-itemsets: $\{a_1\}, \{a_2\}, \ldots, \{a_{100}\}$
- $\binom{100}{2}$ = 4950 frequent 2-itemsets:
  $\{a_1, a_2\}, \{a_1, a_3\}, \{a_1, a_4\} \ldots, \{a_2, a_3\}, \{a_2, a_4\}, \ldots, \{a_{99}, a_{100}\}$

The total number of frequent itemsets that it contains is thus -

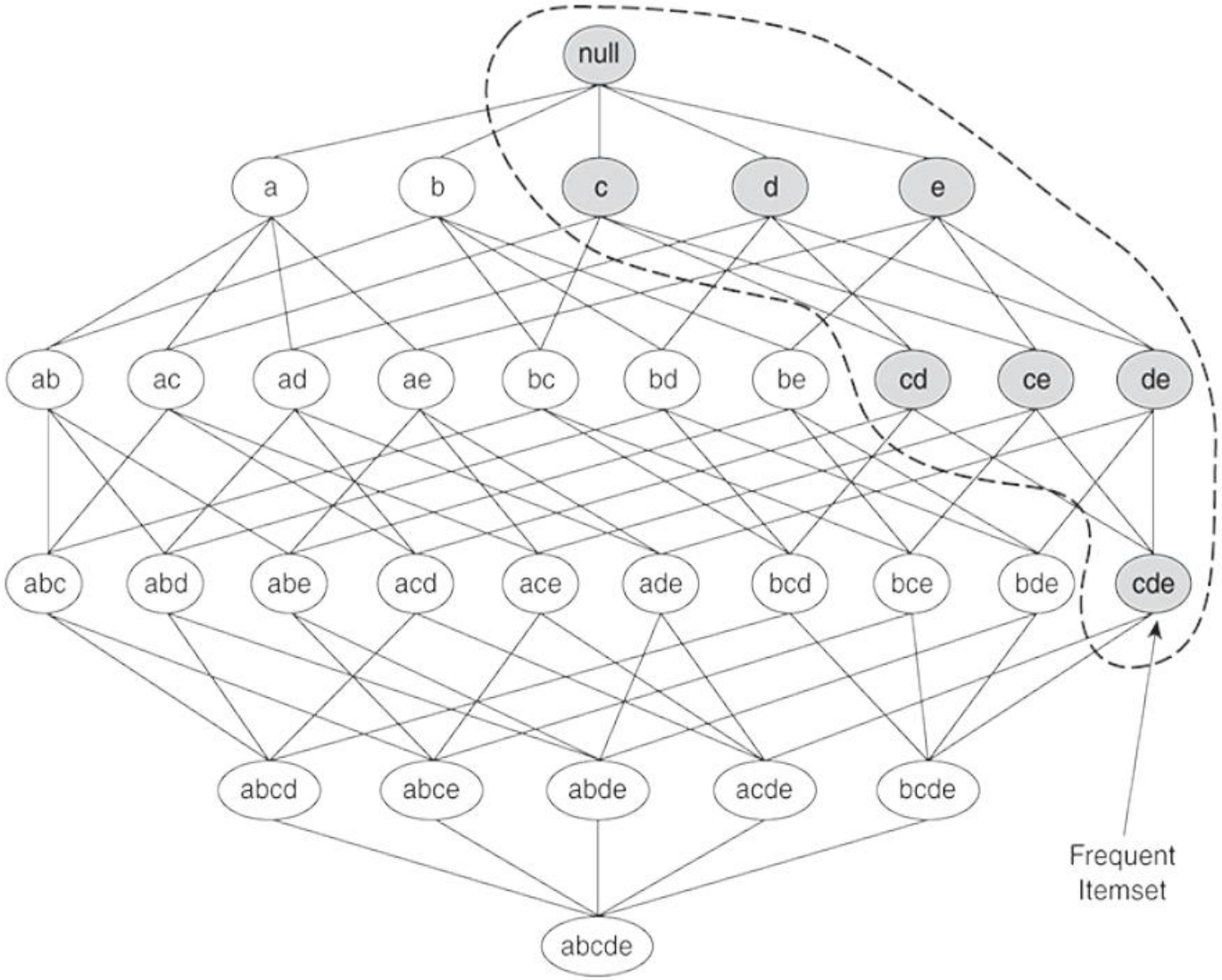
$$\binom{100}{1} + \binom{100}{2} + \ldots + \binom{100}{100} = 2^{100} - 1 \approx 1.27 \times 10^{30}$$

- This is too huge a number of itemsets for any computer to compute or store. To overcome this difficulty, we introduce the concepts of *closed frequent itemset* and *maximal frequent itemset*.

Compact Representation of Frequent Itemsets



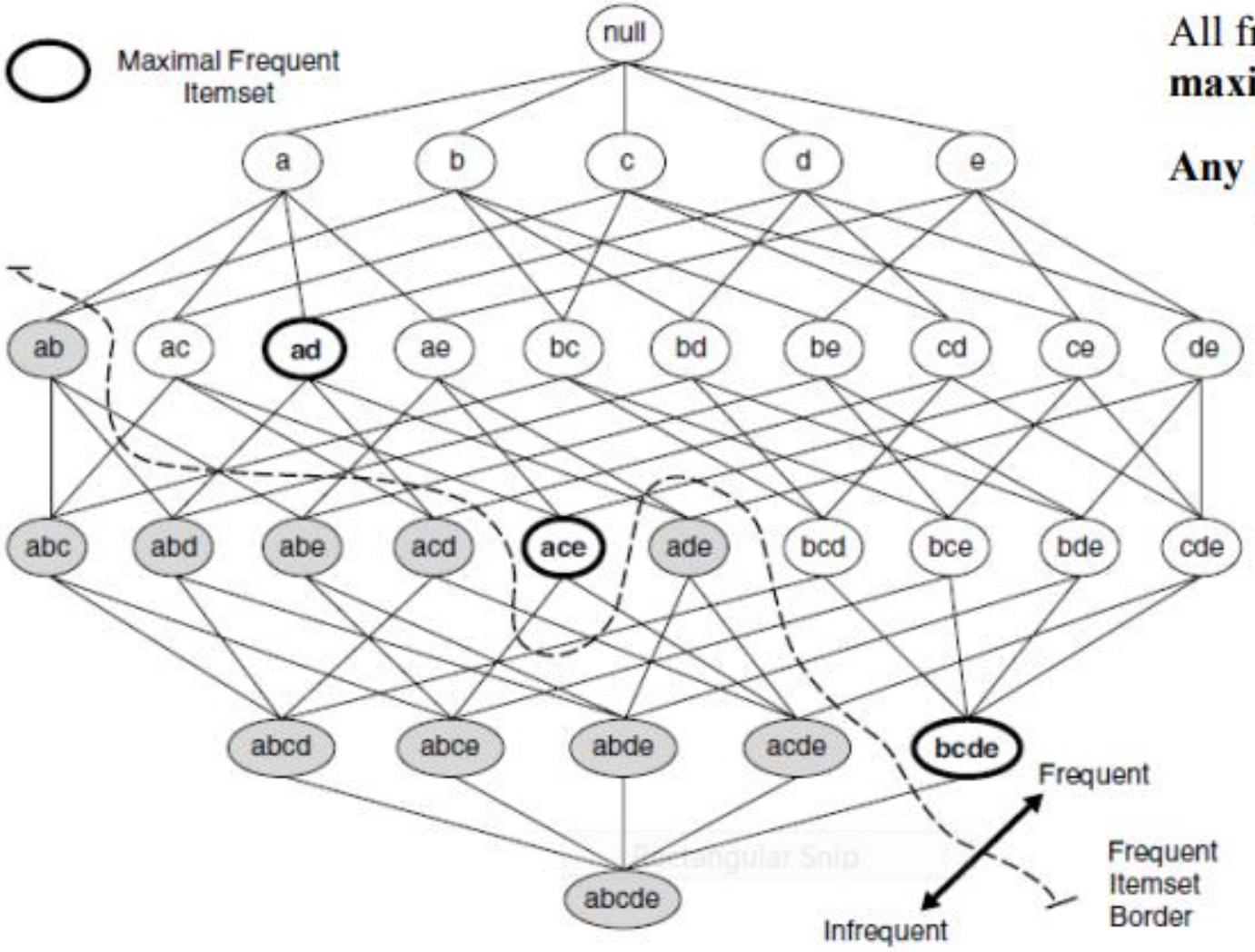
Compact Representation of Frequent Itemsets



Maximal Frequent Itemsets

- A **maximal frequent itemset** is defined as a frequent itemset for which none of its immediate supersets are frequent.
- Maximal frequent itemsets effectively provide a compact representation of frequent itemsets.
- Maximal frequent itemsets form the smallest set of itemsets from which all frequent itemsets can be derived.

Maximal Frequent Itemset



All frequent itemsets can be derived from **maximal frequent itemsets ad, ace, bcde.**

Any frequent itemset $\subseteq$
a maximal frequent itemset

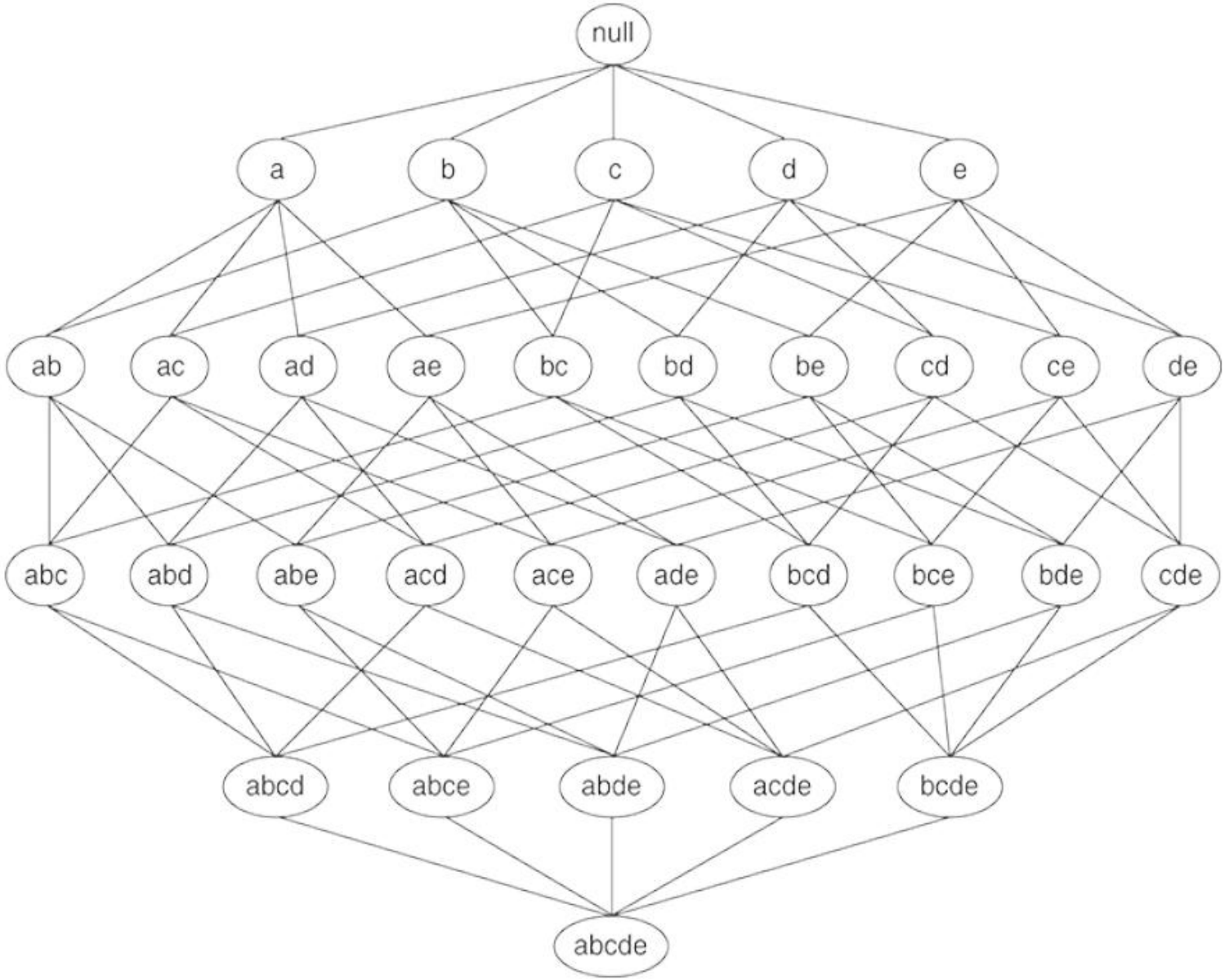
Closed Frequent Itemsets

- An itemset X is **closed** if none of its immediate supersets has exactly the same support count as X.
- Put another way, X is not closed if at least one of its immediate supersets has the same support count as X.
- Closed itemsets provide a minimal representation of itemsets without losing their support information.
- An itemset is a **closed frequent itemset** if it is closed and its support is greater than or equal to min_sup.

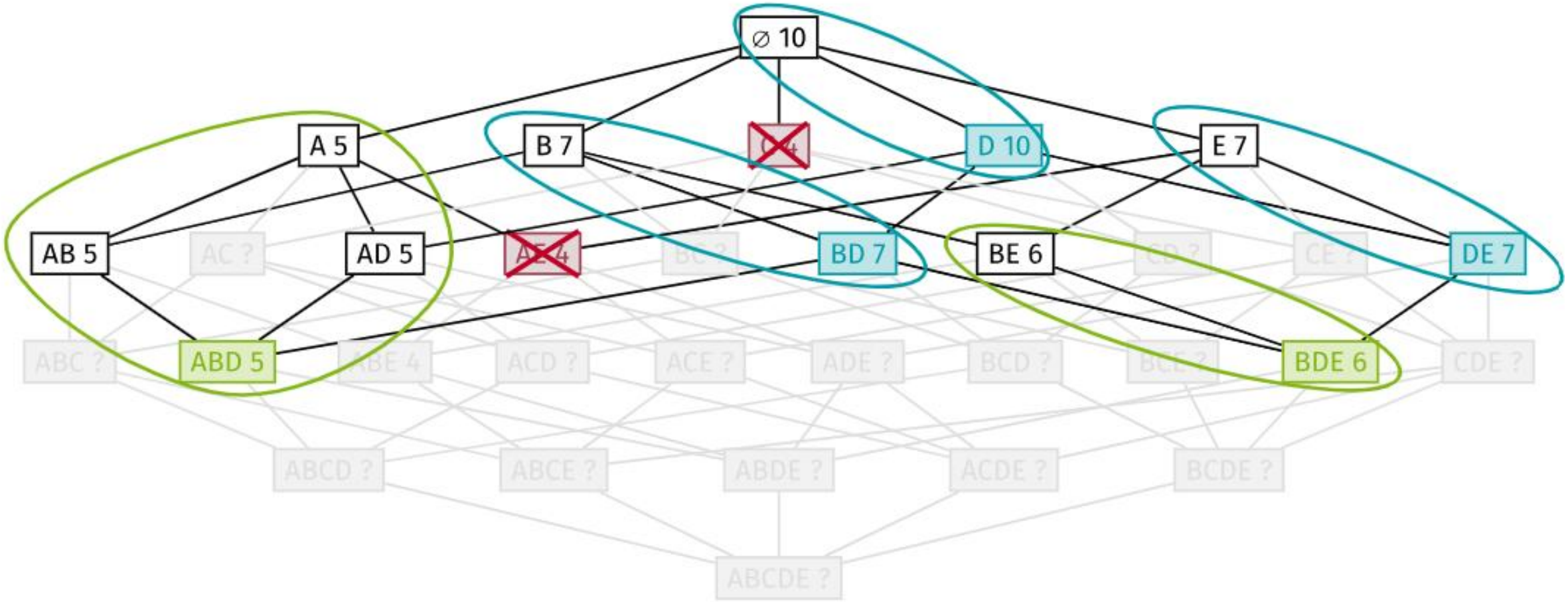
Closed Frequent Itemsets

TID	Items
1	abc
2	abcd
3	bce
4	acde
5	de

minsup = 40%



Closed Itemset Example

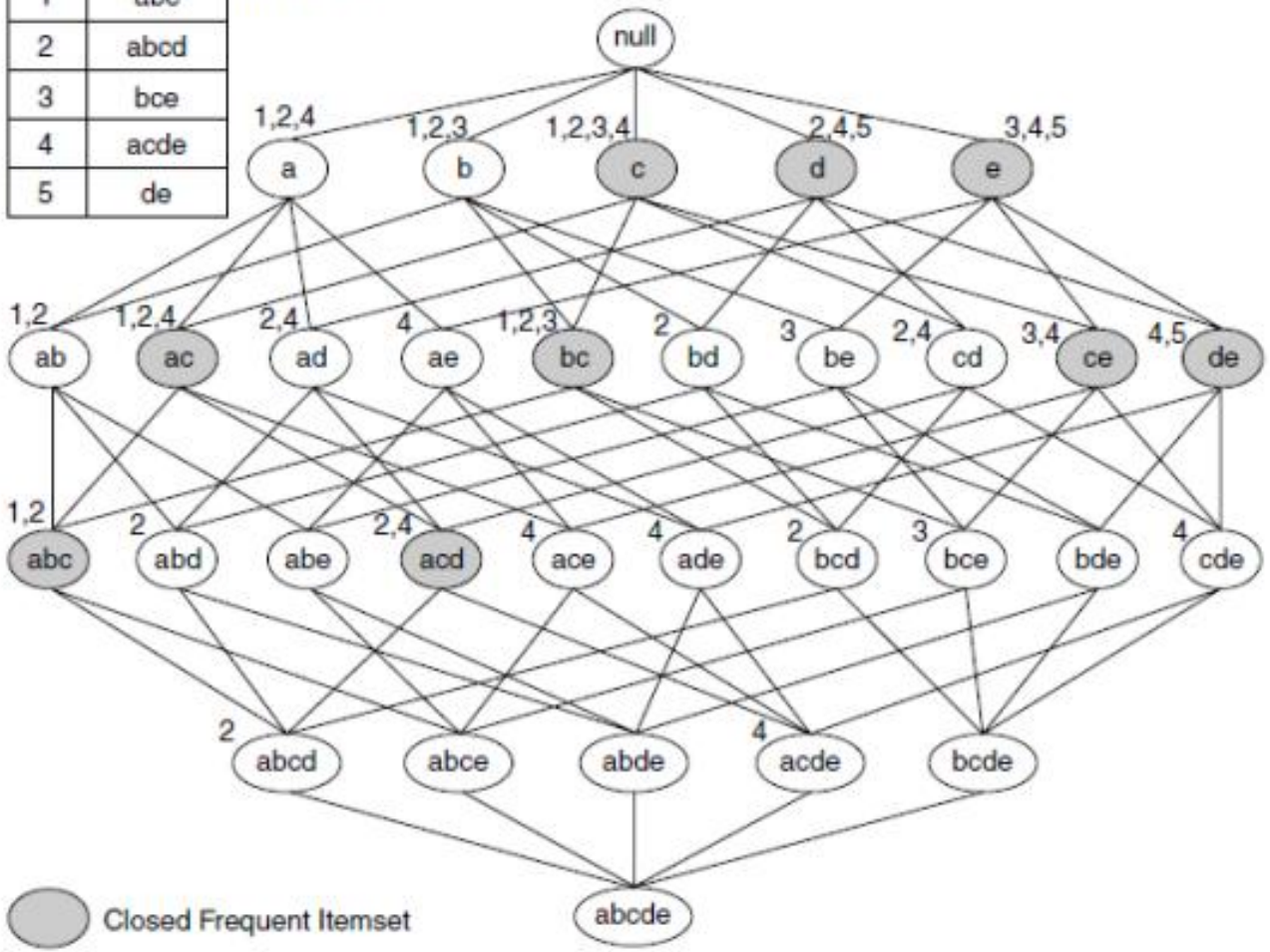


The green and blue itemsets are closed, circles indicate groups of the same support.

Closed Frequent Itemsets

TID	Items
1	abc
2	abcd
3	bce
4	acde
5	de

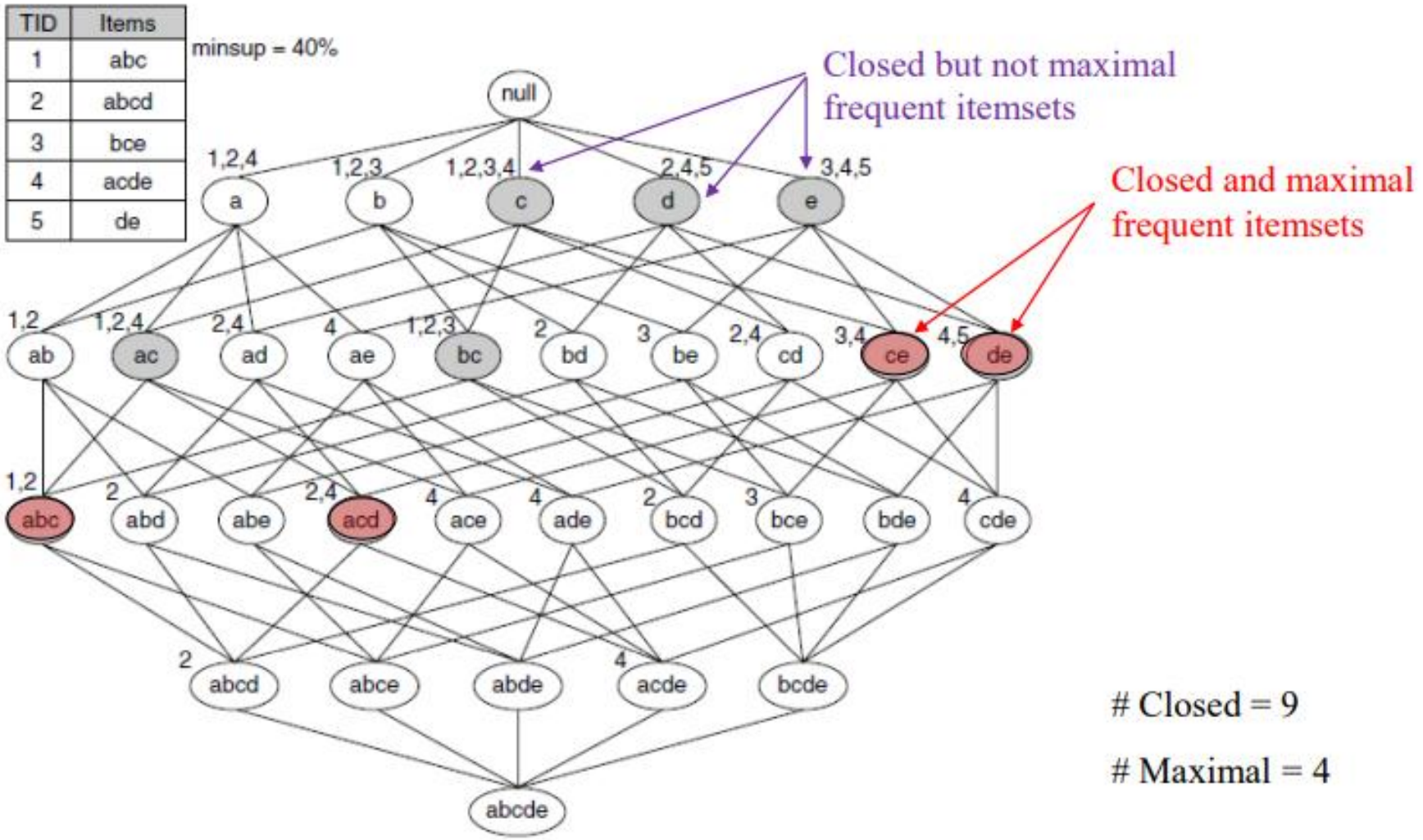
minsup = 40%



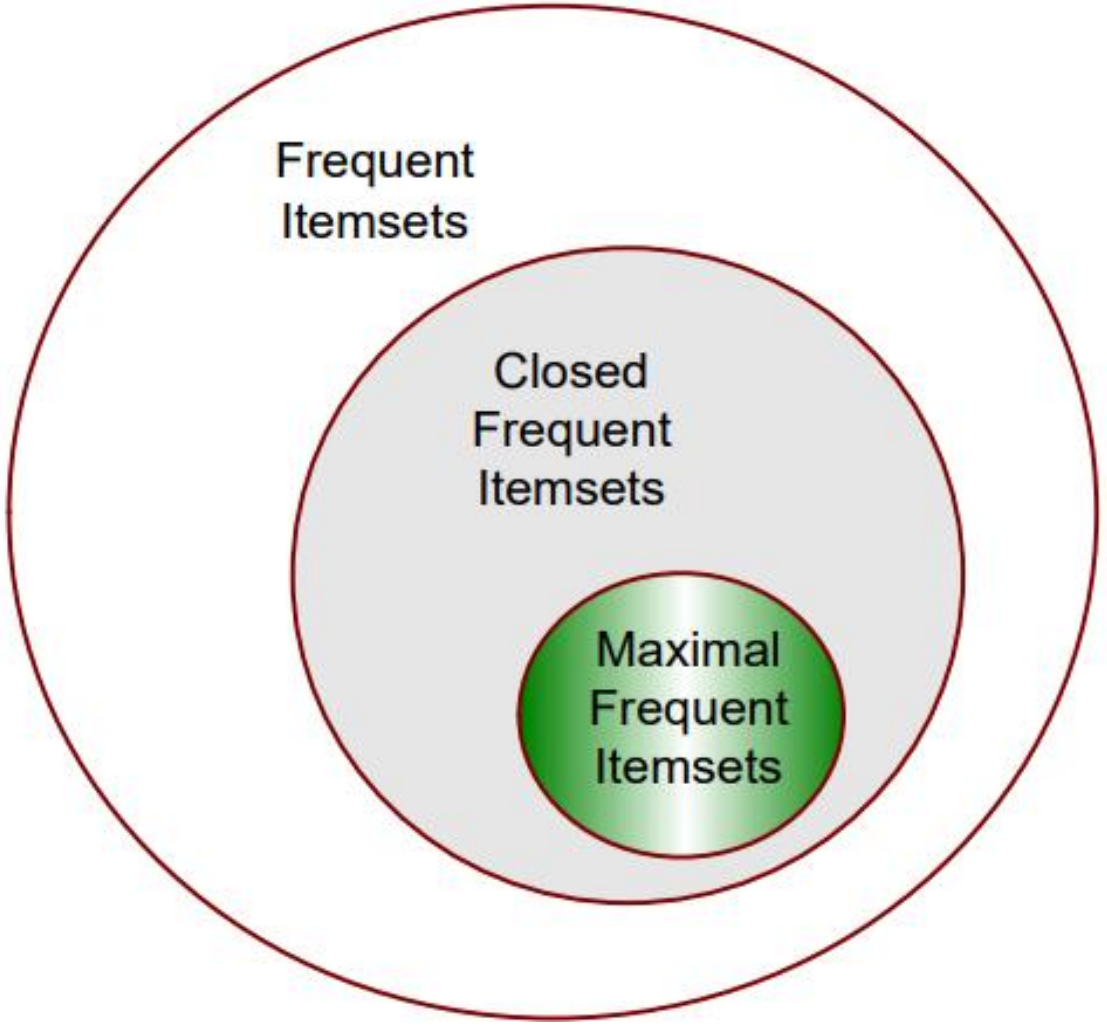
All subsets of a closed frequent itemset are frequent and their supports is greater than or equal to the support of that closed frequent itemset.

For example, all subsets of a closed frequent itemset **abc** are frequent and their supports ≥ support of **abc**.

Maximal vs Closed Itemsets



Maximal vs Closed Itemsets



Closed and Maximal Frequent Itemsets

Suppose that a transaction database has only two transactions: $\{< a_1, a_2, \ldots, a_{100} >; < a_1, a_2, \ldots, a_{50} >\}$. Let the minimum support count threshold be *min_sup = 1*.

closed frequent itemsets and their support counts are -

$$C = \{\{a_1, a_2, \ldots, a_{100}\} : 1; \{a_1, a_2, \ldots, a_{50}\} : 2\}$$

Maximal frequent itemsets and their support counts are -

$$M = \{\{a_1, a_2, \ldots, a_{100}\} : 1\}$$

The set of closed frequent itemsets contains complete information regarding the frequent itemsets.

Closed and Maximal Frequent Itemsets

For example, from C, we can derive -

- $\{a_2, a_{45}\} : 2$ since $\{a_2, a_{45}\}$ is a subitemset of the itemset $\{a_1, a_2, \ldots, a_{50}\} : 2$
- $\{a_8, a_{55}\} : 1$ since $\{a_8, a_{55}\}$ is not a subitemset of the previous itemset but is subitemset of the itemset $\{a_1, a_2, \ldots, a_{100}\} : 1$

However from the maximal frequent itemset, we can only say that both itemsets $\{a_2, a_{45}\}$ and $\{a_8, a_{55}\}$ are frequent but we can not say anything about their actual support counts.

Thus it's more desirable to mine closed frequent itemsets.

Mining Closed and Max Patterns

- How can we mine closed frequent itemsets?
- A naïve approach would be to first mine the complete set of frequent itemsets and then remove every frequent itemset that is a proper subset of, and carries the same support as, an existing frequent itemset. However, this is quite costly.
- A recommended methodology is to prune the search space as soon as we can identify the case of closed itemsets during mining.
- **Itemset merging.** If every transaction containing a frequent itemset X also contains an itemset Y but not any proper superset of Y, then $X \cup Y$ forms a frequent closed itemset and there is no need to search for any itemset containing X but no Y.

Mining Closed and Max Patterns

Table 4.2 Mining the FP-tree by creating conditional (sub-)pattern bases.

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}

For example, the projected conditional database for prefix itemset {I5:2} is {{I2, I1}, {I2, I1, I3}}, from which we can see that each of its transactions contains itemset {I2, I1} but no proper superset of {I2, I1}. Itemset {I2, I1} can be merged with {I5} to form the closed itemset {I5, I2, I1: 2}, and we do not need to mine for closed itemsets that contain I5 but not {I2, I1}.

Evaluation of Association Patterns

- Association rule algorithms tend to produce too many rules
  - many of them are ***uninteresting*** or ***redundant***
  - $\{A,B\} \rightarrow \{D\}$ is ***Redundant*** if $\{A,B,C\} \rightarrow \{D\}$ and $\{A,B\} \rightarrow \{D\}$ have same support & confidence
  - An association rule $X \longrightarrow Y$ is **redundant** if there exists another rule $X' \rightarrow Y'$, where X is a subset of X' and Y is a subset of Y', such that the support and confidence for both rules are identical.

- ***Interestingness measure*** can be used to prune/rank the derived patterns

- In the original formulation of association rules, support & confidence are the only measures used

Computing Interestingness Measures

- Given a rule $X \rightarrow Y$, information needed to compute rule interestingness can be obtained from a contingency table

Contingency table for $X \rightarrow Y$

	Y	$\overline{Y}$	
X	f_{11}	f_{10}	f_{1+}
$\overline{X}$	f_{01}	f_{00}	f_{0+}
	f_{+1}	f_{+0}	$\|T\|$

f_{11}: support of X and Y
f_{10}: support of X and $\overline{Y}$
f_{01}: support of $\overline{X}$ and Y
f_{00}: support of $\overline{X}$ and $\overline{Y}$

Drawback of Confidence

	Coffee	$\overline{\text{Coffee}}$	
Tea	15	5	20
$\overline{\text{Tea}}$	75	5	80
	90	10	100

Association Rule: $Tea \implies Coffee$

$$Confidence(Tea \implies Coffee) = \frac{Support(\{Tea, Coffee\})}{Support(\{Tea\})} = \frac{15}{20} = 0.75$$

- Although confidence is high, rule is misleading.

Measure for Association Rules

- So, what kind of rules do we really want?
  - Confidence($X \rightarrow Y$) should be sufficiently high
    - To ensure that people who buy X will more likely buy Y than not buy Y
  - Confidence($X \rightarrow Y$) > support(Y)
    - Otherwise, rule will be misleading because having item X actually reduces the chance of having item Y in the same transaction
    - Is there any measure that capture this constraint?
      - Answer: Yes. There are many of them.

STRONG RULES ARE NOT NECESSARILY INTERESTING

- How can we tell which strong association rules are really interesting? Strong rules are association rules which satisfies min_sup and min_conf criteria.
- A misleading “strong” association rule. Suppose we are interested in analyzing transactions with respect to the purchase of computer games and videos. Let game refer to the transactions containing computer games, and video refer to those containing videos. Of the 10,000 transactions analyzed, the data show that 6000 of the customer transactions included computer games, whereas 7500 included videos, and 4000 included both computer games and videos. Suppose that a data mining program for discovering association rules is run on the data, using a minimum support of, say, 30% and a minimum confidence of 60%. The following association rule is discovered:

$$buys(X, \text{“}computer\ games\text{”}) \Rightarrow buys(X, \text{“}videos\text{”})$$
$$[support = 40\%, confidence = 66\%].$$

Correlation Analysis

- As we have seen so far, the support and confidence measures are insufficient at filtering out uninteresting association rules.
- To tackle this weakness, a *correlation measure* can be augmented to the support–confidence framework for association rules. **Lift** or **Interest** is such a correlation measure.

$$\text{Lift}(A,B) = \text{conf}(A \rightarrow B) / \text{support}(B)$$
$$= \text{support}(A \cup B) / \text{support}(A)\ \text{support}(B)$$
$$\text{Interest}(A,B) = \text{support}(A \cup B) / \text{support}(A)\ \text{support}(B)$$

$$\text{Interest}(A,B) \begin{cases} = 1 & \text{if } A \text{ and } B \text{ are independent} \\ > 1 & \text{if } A \text{ and } B \text{ are positively correlated} \\ < 1 & \text{if } A \text{ and } B \text{ are negatively correlated} \end{cases}$$

EXAMPLE: LIFT / INTEREST

	Coffee	$\overline{\text{Coffee}}$	
Tea	15	5	20
$\overline{\text{Tea}}$	75	5	80
	90	10	100

$$Confidence(Tea \implies Coffee) = \frac{Support(\{Tea,Coffee\})}{Support(\{Tea\})} = \frac{15}{20} = 0.75$$

- Although confidence is high, rule is misleading.

$$Lift(Tea \implies Coffee) = Confidence(Tea \implies Coffee)/Support(Coffee) = \frac{0.75}{0.9} = 0.833$$

From this we can say that Tea and Coffee are negatively correlated.