Data-Mining-Module-4-Important-Topics-PYQs

1. Describe any three methods to improve the efficiency of the Apriori Algorithm

Apriori is an algorithm used to find **frequent itemsets** (sets of items that often appear together) and create **association rules** (like "if bread is bought, milk is also bought").

It works in steps:

1. Find frequent single items (like {milk}, {bread}).
2. Use those to find frequent 2-item combinations (like {milk, bread}).
3. Then use those to find frequent 3-item combinations, and so on.

At each level, it scans the entire dataset again — which can be very slow if the dataset is large.

Three Ways to Make Apriori Faster:

1. Hash-Based Technique

What it means:
Instead of checking every possible itemset, we use a **hash table** (a fast way of storing and looking up data) to reduce how many candidate itemsets we need to consider.

How it helps:
When generating 2-item or 3-item sets, we can **hash** them into buckets. If a bucket has low count (less than minimum support), we skip all itemsets in that bucket — saving time.

> 🧠 **New term: Hash table** – A data structure that stores data using a key-value format and allows quick searching.

2. Transaction Reduction

What it means:
We reduce the number of transactions (shopping baskets) we look at in each round.

How it helps:
If a transaction doesn't contain any of the current frequent itemsets, it **won't help in future rounds**, so we can ignore it in the next scans.

> 🧠 **Example:**
> If a shopping basket doesn't contain any frequent 2-itemsets, it definitely won't help find any frequent 3-itemsets — so skip it!

3. Partitioning

What it means:
We split the data into smaller parts (partitions), find frequent itemsets in each partition, and then combine results.

How it helps:
Instead of scanning the whole dataset multiple times, we do **just two scans**:

1. First, find frequent itemsets in each partition.
2. Second, scan again to verify which of those are frequent in the full dataset.

> 🧠 **Why it works:**
> If an itemset is frequent in the full data, it **must be frequent in at least one partition**.

Method	What it Does	Why it Helps
Hash-Based	Uses hashing to filter out unlikely itemsets	Reduces candidate itemsets
Transaction Reduction	Skips irrelevant transactions in later scans	Less data to scan each time
Partitioning	Splits data and finds frequent sets locally first	Only two scans of full data

2. Define support, confidence and frequent itemset in association rule mining context.

1. Support

What it means:
Support tells us **how often an item or itemset appears** in the total dataset.

> 💡 In simple words:
> If 2 out of 100 customers bought both a **computer and antivirus software**, the **support** for `{computer, antivirus}` is **2%**.

Why it's important:
Support helps identify how **useful** or **popular** a rule is in the real dataset.

2. Confidence

What it means:
Confidence shows **how often the rule has been true** — among the transactions that contain the "if" part, how many also contain the "then" part.

> 💡 Example:
> If 60 out of 100 people who bought a **computer** also bought **antivirus software**, then:
> **Confidence of the rule `computer ⇒ antivirus` = 60%**

Why it's important:
Confidence tells us **how strong or reliable** the rule is.

3. Frequent Itemset

What it means:
A **frequent itemset** is a **group of items** that often appear together in many transactions.

> 💡 Example:
> In a store, if many people buy both **bread and butter**, then `{bread, butter}` is a **frequent itemset**.

What does k-itemset mean?
If an itemset has **k items**, it’s called a **k-itemset**.

- `{bread}` → 1-itemset
- `{bread, butter}` → 2-itemset
- `{bread, butter, jam}` → 3-itemset

Term	Meaning	Example
Support	% of all transactions that include the itemset	2% of all transactions include `{computer, antivirus}`
Confidence	% of transactions with item A that also include item B	60% of those who bought a computer also bought antivirus
Frequent Itemset	Group of items that appear together often	`{bread, butter}` is a frequent itemset

3. Write about the bi-directional searching technique for pruning in pincer search algorithm

Bi-directional searching means that the algorithm searches from **both ends** — from small to large itemsets (bottom-up) **and** from large to small itemsets (top-down) — at the same time.

This is the core idea of the **Pincer Search algorithm** and helps in **pruning** (removing) unpromising itemsets early.

Let's Break it Down:

In **Apriori**, we only do a **bottom-up** search:

- Start with 1-itemsets → 2-itemsets → 3-itemsets...
- Continue until no more frequent itemsets are found.
- But we may waste time checking many unnecessary combinations.

In Pincer Search:

We do **both**:

- **Bottom-Up Search:**
  Like Apriori — build larger frequent itemsets step-by-step.
- **Top-Down Search:**
  At the same time, we try to **guess the largest possible frequent itemsets** (called **Maximal Frequent Itemsets** or MFIs), and use them to cut down the number of itemsets we need to check.

How Bi-directional Pruning Works

Bi-directional pruning means:

From the Bottom (Bottom-Up):

- We check small itemsets and grow them (like {A} → {A,B} → {A,B,C}).
- If a set is **not frequent**, we **stop growing it**.
  ➤ Example: If {A,B} is infrequent, we don't check {A,B,C}.

From the Top (Top-Down):

We want to **guess big itemsets** (like `{A, C, D, E}`) that we **think might be frequent**, and then **check their smaller parts (subsets)** to confirm.

If any subset is **not frequent**, then we can **stop checking** that big itemset and its related ones.

Let's Take a Real Example:

Imagine we have these transactions (shopping baskets):

Transaction ID	Items
T1	A, C, D, E
T2	A, C
T3	A, D, E
T4	C, D
T5	A, C, D, E

We **guess** (from the data) that the set `{A, C, D, E}` might be a **maximal frequent itemset** (a large group of items that appear together often).

So we put `{A, C, D, E}` into our **MFI list** (Maximal Frequent Itemset list).

Now comes the key part (bi-directional pruning):

We check smaller parts of it — like `{A, C}`, `{C, D}`, `{A, D, E}`.
Suppose we find that:

- `{A, C}` appears **only once** in the dataset (so it's **not frequent**).

Then we can say:

> "If `{A, C}` itself is not frequent, then the bigger set `{A, C, D, E}` **definitely won't be frequent either**."

So we can **prune** (remove) `{A, C, D, E}` and **save time** — no need to scan the whole database for it.

Why is this powerful?

Because:

- **Top-down pruning** helps **avoid checking large itemsets** that will eventually fail.
- **Bottom-up pruning** helps **avoid growing itemsets** from infrequent smaller sets.
- When both work together, we **save time** and **reduce database scans**.

Direction	What It Does	How It Helps
Bottom-Up	Grow itemsets step-by-step	Like Apriori — builds frequency
Top-Down	Track largest frequent sets	Prunes large sets early
Combined	Bi-directional pruning	Fewer checks, faster mining

Pincer Search Algorithm:

The Pincer Search algorithm is an efficient approach for mining frequent itemsets in large datasets. It combines both **bottom-up** (support-based pruning) and **top-down** (maximal frequent itemset search) approaches.

Steps of the Pincer Search Algorithm:

Step 1: Generate candidate 1-itemsets and compute their support

- Begin by generating all **candidate 1-itemsets** (individual items) from the dataset.
- **Compute their support**, which means how frequently each item appears in the transactions.

Step 2: Identify frequent 1-itemsets and extend them to larger itemsets

- From the candidate 1-itemsets, identify the **frequent 1-itemsets** (those that meet the minimum support threshold).

- Use the frequent 1-itemsets to generate **candidate 2-itemsets** (pairs of items) and extend this process to generate larger itemsets.

> **Note:** This is similar to the **Apriori** algorithm.

Step 3: Simultaneously perform a top-down search

- **Maintain a list of Maximal Frequent Itemsets (MFIs)** — these are itemsets that are frequent and cannot be extended further.
- For each iteration, perform a **top-down search** to find potential large frequent itemsets.
- The idea is to try to **guess large itemsets** that might be frequent and check their **smaller subsets**.

Step 4: Prune infrequent itemsets early using MFI knowledge

- Use the knowledge of **Maximal Frequent Itemsets (MFIs)** to **prune infrequent itemsets early**.
- If any **subset of an itemset** is infrequent (and we know that larger sets containing that subset won't be frequent), **eliminate** that itemset and its supersets from further consideration.

> **Example:** If `{A, C}` is not frequent, **stop checking** larger sets like `{A, C, D, E}`.

Step 5: Refine candidate itemsets until no new frequent itemsets are found

- Continue refining your **candidate itemsets** by extending them to larger ones and checking their support.
- **Stop when no new frequent itemsets** can be found after pruning the non-promising itemsets.

---

4. Discuss the significance of association rule mining in market basket analysis.

What is Association Rule Mining?

Association rule mining is a method used to discover interesting relationships or patterns in large sets of data. It helps in identifying how items are related to each other in transactions, often used in areas like retail, to understand customer buying behaviors.

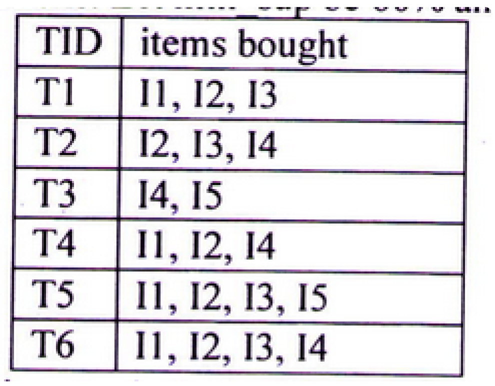
Significance of Association Rule Mining in Market Basket Analysis:

In market basket analysis, **association rule mining** plays a key role by helping retailers understand which products are commonly bought together. This insight can help in creating strategies to boost sales, optimize store layouts, and personalize marketing efforts.

1. **What is Market Basket Analysis?**
    - It's a technique used by retailers to analyze customer buying habits.
    - Retailers observe which items are often bought together in a single shopping trip or transaction (like items in a **shopping basket**).
    - For example, if a customer buys **milk**, they may often buy **bread** as well. Market basket analysis finds such patterns.
2. **How is Association Rule Mining Useful?**
    - It helps retailers discover these **associations** between products that customers tend to buy together.
    - For example, an association rule might look like this:
        - **Milk => Bread** (meaning if a customer buys milk, they are likely to buy bread as well).
    - Retailers can use these insights to make decisions like:
        - **Product Placement**: Placing related items (like bread and milk) closer together to make it easier for customers to find them.
        - **Promotions**: Offering discounts on items frequently bought together (like milk and bread) to encourage customers to buy both.

---

5. A database has six transactions. Let min_sup be 60% and min_conf be 80%



TID	items bought
T1	I1, I2, I3
T2	I2, I3, I4
T3	I4, I5
T4	I1, I2, I4
T5	I1, I2, I3, I5
T6	I1, I2, I3, I4

Find frequent itemsets using Apriori algorithm and generate strong association rules from a three-item dataset.

To solve this problem using the **Apriori Algorithm**, we will follow these steps

Step 1: Find Frequent 1-itemsets

We first count the occurrence of each item in the dataset to see which items meet the **minimum support** threshold (60%).

Given the dataset:

TID	Items Bought
T1	I1, I2, I3
T2	I2, I3, I4
T3	I4, I5
T4	I1, I2, I4
T5	I1, I2, I3, I5
T6	I1, I2, I3, I4

Support Count Calculation:

- **I1** appears in T1, T4, T5, and T6 (4 times)
- **I2** appears in T1, T2, T4, T5, and T6 (5 times)
- **I3** appears in T1, T2, T5, and T6 (4 times)
- **I4** appears in T2, T4, T6 (3 times)
- **I5** appears in T3, T5 (2 times)

Total transactions = 6
Minimum Support = 60% of 6 = 3.6, which rounds up to 4 transactions.

Frequent 1-itemsets:

- **I1**: Appears in 4 transactions → **Frequent**
- **I2**: Appears in 5 transactions → **Frequent**
- **I3**: Appears in 4 transactions → **Frequent**
- **I4**: Appears in 3 transactions → **Not Frequent**

- **I5**: Appears in 2 transactions → **Not Frequent**

Thus, the frequent 1-itemsets are **{I1}, {I2}, {I3}**.

Step 2: Generate 2-itemsets from Frequent 1-itemsets

Next, we generate candidate 2-itemsets by pairing frequent 1-itemsets and check if their support meets the minimum threshold.

Candidate 2-itemsets:

- {I1, I2}: Appears in T1, T4, T5, T6 (4 times)
- {I1, I3}: Appears in T1, T5, T6 (3 times)
- {I2, I3}: Appears in T1, T2, T5, T6 (4 times)

Frequent 2-itemsets:

- **{I1, I2}**: Appears in 4 transactions → **Frequent**
- **{I1, I3}**: Appears in 3 transactions → **Not Frequent**
- **{I2, I3}**: Appears in 4 transactions → **Frequent**

So, the frequent 2-itemsets are **{I1, I2}, {I2, I3}**.

Step 3: Generate 3-itemsets from Frequent 2-itemsets

Now, we generate candidate 3-itemsets by pairing frequent 2-itemsets and check if their support meets the minimum threshold.

Candidate 3-itemsets:

- {I1, I2, I3}: Appears in T1, T5, T6 (3 times)

Frequent 3-itemsets:

- **{I1, I2, I3}**: Appears in 3 transactions → **Not Frequent**

Thus, the frequent 3-itemset is None

Step 4: Generate Strong Association Rules from Frequent Itemsets

At this point, we have identified the **frequent itemsets** from Step 1, Step 2, and Step 3. Now we will generate **strong association rules** from those frequent itemsets.

We need to generate rules only from itemsets that have **at least 2 items**. For each frequent itemset, we create rules and calculate their **confidence** to see if they meet the minimum confidence threshold.

What is Confidence?

Confidence measures how likely it is that if an item is bought, another item will be bought as well.

For example, the rule **I1 ⇒ I2** means "if I1 is bought, then I2 is likely to be bought."
The formula for confidence is:

$$\text{Confidence}(A \Rightarrow B) = \frac{\text{Support}(A, B)}{\text{Support}(A)}$$

Where:

- **Support(A, B)** is the number of transactions where both items (A and B) appear together.
- **Support(A)** is the number of transactions where item A appears.

If the **confidence** is greater than or equal to the **minimum confidence** threshold (in this case 80%), we will consider the rule **strong**.

Generating Rules from Frequent 2-itemsets:

We have the following frequent 2-itemsets:

- **{I1, I2}**
- **{I2, I3}**

We will generate the two possible rules for each of these itemsets:

For {I1, I2} (Frequent 2-itemset):

We can create two rules:

1. **I1 ⇒ I2**
    - **Support(I1, I2) = 4** (Both I1 and I2 appear together in 4 transactions)
    - **Support(I1) = 4** (I1 appears in 4 transactions)

- **Confidence(I1 ⇒ I2) = 4 ÷ 4 = 1.0 (100%)**
  Since **Confidence = 100%**, which is greater than the minimum confidence (80%), this rule is **strong**.

2. **I2 ⇒ I1**
   - **Support(I1, I2) = 4** (Both I1 and I2 appear together in 4 transactions)
   - **Support(I2) = 5** (I2 appears in 5 transactions)
   - **Confidence(I2 ⇒ I1) = 4 ÷ 5 = 0.8 (80%)**
     Since **Confidence = 80%**, which is equal to the minimum confidence (80%), this rule is also **strong**.

For {I2, I3} (Frequent 2-itemset):

We can create two rules:

1. **I2 ⇒ I3**
   - **Support(I2, I3) = 4** (Both I2 and I3 appear together in 4 transactions)
   - **Support(I2) = 5** (I2 appears in 5 transactions)
   - **Confidence(I2 ⇒ I3) = 4 ÷ 5 = 0.8 (80%)**
     Since **Confidence = 80%**, which meets the minimum confidence (80%), this rule is **strong**.
2. **I3 ⇒ I2**

   - **Support(I2, I3) = 4** (Both I2 and I3 appear together in 4 transactions)
   - **Support(I3) = 4** (I3 appears in 4 transactions)
   - **Confidence(I3 ⇒ I2) = 4 ÷ 4 = 1.0 (100%)**

   Since **Confidence = 100%**, which is greater than the minimum confidence (80%), this rule is **strong**.

Step 5: Final Strong Association Rules

From the above calculations, we have the following **strong association rules** (rules that meet the minimum confidence of 80%):

1. **I1 ⇒ I2** (Confidence = 100%)
2. **I2 ⇒ I1** (Confidence = 80%)
3. **I2 ⇒ I3** (Confidence = 80%)
4. **I3 ⇒ I2** (Confidence = 100%)

Summary of Results:

- **Frequent 1-itemsets:** {I1}, {I2}, {I3}
- **Frequent 2-itemsets:** {I1, I2}, {I2, I3}
- **Frequent 3-itemsets:** None
- **Strong Association Rules:**
  1. **I1 ⇒ I2** (Confidence = 100%)
  2. **I2 ⇒ I1** (Confidence = 80%)
  3. **I2 ⇒ I3** (Confidence = 80%)
  4. **I3 ⇒ I2** (Confidence = 100%)

This concludes the process of finding frequent itemsets and generating strong association rules using the **Apriori algorithm**.

6. Describe the working of dynamic itemset counting technique with suitable example. Specify when to move an itemset from dashed structures to solid structures.

DIC is a **faster and smarter version of the Apriori algorithm**. It helps find frequent itemsets (popular combinations of items) from large transaction databases like supermarket purchases.

Instead of scanning the database over and over again like Apriori, **DIC does things on the go — while it's already scanning**. It starts checking new combinations even **before the current step is finished**.

Example to Understand

Transaction	Items Bought
T1	A, B, C
T2	A, C
T3	B, C, D
T4	A, B, D
T5	A, B, C, D

Let’s say **min support** = 3 (so an item or itemset must appear in **at least 3 transactions** to be "frequent").

Working of DIC (Step-by-Step)

Step 1: Partition the Database

Instead of looking at the whole database at once, DIC **splits it into chunks** called **partitions**. This lets it process data bit by bit.
Let’s divide the data into 3 partitions:

- **Partition 1** → T1, T2
- **Partition 2** → T3, T4
- **Partition 3** → T5

Step 2: Start with Frequent 1-Itemsets

- First, we look at the **first partition** (T1, T2):
    - T1: A, B, C
    - T2: A, C
- Now count how many times each item appears
    - A → 2 times
    - B → 1 time
    - C → 2 times
- So now we guess that **A and C might be frequent**, and **B is maybe frequent**, but we need more data to be sure.
- While going through more partitions, we keep **dynamically adding new itemsets** to check if they become frequent.

Step 3: Dynamically Count Bigger Itemsets

As we scan Partition 2 (T3, T4)

- T3: B, C, D
- T4: A, B, D

We update the counts:

- A → now 4 times

- B → now 4 times
- C → now 3 times
- D → now 2 times
  Now we look at **pairs (2-itemsets)** like:
- {A, B} → Appears in T1, T4, T5 → 3 times → ✅ frequent
- {B, C} → Appears in T1, T3, T5 → 3 times → ✅ frequent
- {A, D} → Only appears in T4, T5 → 2 times → ❌ not frequent → **Prune it** (i.e., remove it)

So even before finishing all partitions, we already know some combinations are frequent or not. That's why DIC is faster.

Step 4: Generate Rules (Same as Apriori)

From frequent itemsets like:

- {A, B}
- {B, C}
- {B, D}

We can now generate **association rules** like

- If A is bought ⇒ B is likely bought too
- If C is bought ⇒ B is likely bought too
  We only keep the rules that have **high confidence** (for example, ≥ 80%).

When to move from dashed to solid structures

- This is how DIC keeps track of itemsets:
  - **Dashed Circle→ New itemset. Still being counted. We don't yet know if it's frequent
  - **Solid Circle** → We're confident now. This itemset has been seen enough, so it's **confirmed frequent**.
  - **Cross** → **Itemset has low support** → Discard it (Pruned)**
- When do we move from Dashed to Solid?
  - When we **finish scanning all partitions**, **and** the itemset's **count is ≥ min support**, we **convert the dashed circle to solid**.

7. Explain the partitioning algorithm for finding large itemset and explain how it removes the disadvantage of Apriori algorithm.

What is the Partitioning Algorithm?

It’s a smarter way to find **frequent itemsets** (popular combinations of items) from large datasets like supermarket purchases — similar to Apriori, but **much faster and more efficient**.

Imagine you work in a supermarket chain with **8 stores**. Instead of analyzing all sales from all stores at once (which is slow), you first **analyze each store separately**, then **combine the results** to find popular products sold across all stores.

Some Terms First

- **Itemset**: A group of items (e.g., {A, B} means A and B are bought together).
- **Frequent Itemset**: An itemset that appears **often enough** (based on a threshold).
- **Support**: Number of times an itemset appears in transactions.
- **min_sup (minimum support)**: The minimum number of times an itemset must appear to be considered frequent.

Steps of the Partitioning Algorithm (with simple explanation)

Step 1: Partition the Database

Partition 1 → {T1, T2, T3, T4}
Partition 2 → {T5, T6, T7, T8}

Step 2: Find Local Frequent Itemsets in Each Partition

Partition 1 Transactions:

TID	Items Bought
T1	A, B, C
T2	A, C
T3	B, C, D
T4	A, B, D

Let's check the counts of combinations that appear **at least twice**:

- `{A}` → T1, T2, T4 ✅ (3 times)
- `{B}` → T1, T3, T4 ✅ (3 times)
- `{C}` → T1, T2, T3 ✅ (3 times)
- `{D}` → T3, T4 ✅ (2 times)

Frequent 1-itemsets in Partition 1: `{A}` , `{B}` , `{C}` , `{D}`

Now for **frequent 2-itemsets** (at least 2 times):

- `{A, B}` → T1, T4 ✅ (2 times)
- `{A, C}` → T1, T2 ✅ (2 times)
- `{B, C}` → T1, T3 ✅ (2 times)
- `{B, D}` → T3, T4 ✅ (2 times)

Frequent 2-itemsets in Partition 1: `{A, B}` , `{A, C}` , `{B, C}` , `{B, D}`

What about 3-itemsets?

- `{A, B, C}` → Only in T1 ❌ (1 time)
- `{A, B, D}` → Only in T4 ❌
- `{B, C, D}` → Only in T3 ❌

No Frequent 3-itemsets in Partition 1

Partition 2 Transactions:

TID	Items Bought
T5	A, B, C, D
T6	B, C
T7	A, D
T8	A, B, C

Same process:

- `{A}` → T5, T7, T8 ✅ (3 times)
- `{B}` → T5, T6, T8 ✅ (3 times)

- `{C}` → T5, T6, T8 ✅ (3 times)
- `{D}` → T5, T7 ✅ (2 times)

Frequent 1-itemsets in Partition 2: `{A}`, `{B}`, `{C}`, `{D}`
Now frequent 2-itemsets:

- `{A, B}` → T5, T8 ✅ (2 times)
- `{A, C}` → T5, T8 ✅ (2 times)
- `{B, C}` → T5, T6, T8 ✅ (3 times)
- `{A, D}` → T5, T7 ✅ (2 times)
- `{B, D}` → Only in T5 ❌ (1 time)

Frequent 2-itemsets in Partition 2: `{A, B}`, `{A, C}`, `{B, C}`, `{A, D}`

3-itemsets?

- `{A, B, C}` → T5, T8 ✅ (2 times)
- `{A, B, D}` → Only T5 ❌
- `{B, C, D}` → Only T5 ❌

Frequent 3-itemset in Partition 2: `{A, B, C}`

Step 3: Merge Local Frequent Itemsets

We combine the frequent itemsets from **both partitions**:

- From Partition 1: `{A, B}`, `{A, C}`, `{B, C}`, `{B, D}`
- From Partition 2: `{A, B}`, `{A, C}`, `{B, C}`, `{A, D}`, `{A, B, C}`

Combined Candidate Itemsets:

- `{A, B}`, `{A, C}`, `{B, C}`, `{B, D}`, `{A, D}`, `{A, B, C}`

Step 4: Global Scan to Find Final Frequent Itemsets

Itemset	Appears In	Count
`{A, B}`	T1, T4, T5, T8	4 ✅
`{A, C}`	T1, T2, T5, T8	4 ✅
`{B, C}`	T1, T3, T5, T6, T8	5 ✅

Itemset	Appears In	Count
`{B, D}`	T3, T4	2 ✅
`{A, D}`	T4, T5, T7	3 ✅
`{A, B, C}`	T1, T5, T8	3 ✅

Why is Partitioning Algorithm Better Than Apriori?

Problem in Apriori	How Partitioning Fixes It
Apriori scans the database **many times** for each itemset size	Partitioning does **only one final scan**
Apriori generates **lots of candidate itemsets**	Partitioning works only with **local frequent itemsets**, reducing the number
Apriori uses **more memory and time**	Partitioning works in **smaller chunks** (less memory, faster)

8. What is FP growth algorithm? What are the advantages?

The **Frequent Pattern Growth (FP-Growth)** algorithm is used to find **frequent itemsets** (combinations of items that often appear together in transactions), especially in **large datasets**.

It is **faster and more efficient than Apriori** because it:

- **Avoids generating too many combinations (candidates)**.
- **Avoids scanning the database again and again**.

Key Concepts

- **Itemset**: A set of items bought together (e.g., {A, B, C}).
- **Support**: How many transactions contain that itemset.
- **Minimum Support (min_sup)**: A user-defined threshold — only itemsets that appear *at least* this many times are considered *frequent*.
- **FP-Tree (Frequent Pattern Tree)**: A compact data structure that stores the transactions using shared paths to save memory.

Steps of FP-Growth Algorithm

Step 1: Scan the database to find frequent items

- Count how many times each item appears in all transactions.
- **Remove items that don't meet min_sup.**

Example: From these transactions:

```
T1: A, B, C
T2: A, C
T3: B, C, D
T4: A, B, D
T5: A, B, C, D
```

```
A – 4 times
B – 4 times
C – 4 times
D – 3 times
```

If `min_sup = 2`, all items are frequent.

Step 2: Build the FP-Tree

- **Sort items in each transaction** by **descending frequency**.
- Insert each transaction into the tree. If a path is already in the tree, **reuse the path and increase counts**.
  This helps **save space** and **combine repeated parts** of the data.

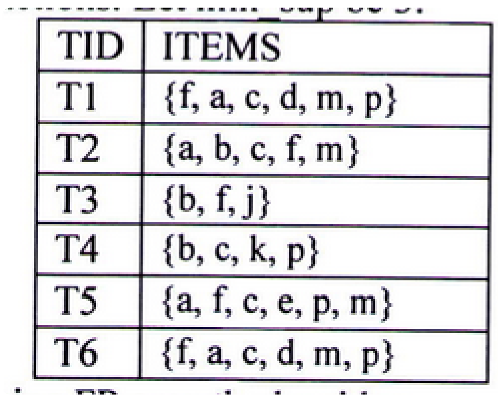
Step 3: Extract frequent itemsets using conditional pattern bases

- For each item, build its **conditional FP-tree**:
  - A smaller tree showing only transactions where the item appears.
  - Helps find combinations involving that item.
- Repeat this step **recursively** to find all frequent itemsets.

Advantages of FP-Growth

Feature	Benefit
No candidate generation	Saves time and memory
Fewer database scans	Just 2 passes (once for frequency, once for tree)
Space-efficient	Uses compact FP-tree with shared paths
Fast on large datasets	Works much better than Apriori on large data

9. A database has six transactions. Let min_sup be 3. Find frequent itemsets using FP growth algorithm.



TID	ITEMS
T1	{f, a, c, d, m, p}
T2	{a, b, c, f, m}
T3	{b, f, j}
T4	{b, c, k, p}
T5	{a, f, c, e, p, m}
T6	{f, a, c, d, m, p}

Step 1: Transaction Table

TID	Items
T1	{f, a, c, d, m, p}
T2	{a, b, c, f, m}
T3	{b, f, j}
T4	{b, c, k, p}
T5	{a, f, c, e, p, m}
T6	{f, a, c, d, m, p}

Step 2: Count Item Frequencies

We count how many times each item appears:

Item	Frequency
f	5
a	4
c	5
d	2
m	4
p	4
b	3
j	1
k	1
e	1

Now, eliminate items that don't meet **min_sup = 3**:
✅ Frequent items: `f, a, c, m, p, b`
❌ Remove: `d, j, k, e`

Step 3: Reorder Items in Each Transaction (by frequency)

We now reorder the items in each transaction by **descending frequency**:
Let's use this frequency order: `f > c > a > m > p > b`

TID	Cleaned & Reordered Items
T1	f, c, a, m, p
T2	f, c, a, m, b
T3	f, b
T4	c, p, b
T5	f, c, a, m, p
T6	f, c, a, m, p

Step 4: Build the FP-Tree

We now construct the FP-tree by inserting the reordered transactions:

Start with a NULL root.

1. **T1: f → c → a → m → p**
2. **T2: f → c → a → m → b**
3. **T3: f → b**
4. **T4: c → p → b**
5. **T5: f → c → a → m → p**
6. **T6: f → c → a → m → p**

```
Root
 └── f:5
      ├── c:5
           ├── a:5
                ├── m:5
                     ├── p:3
                     └── b:1
      └── b:1
 └── c:1
      ├── p:1
           └── b:1
```

Now we **mine the tree from bottom up** using **conditional FP-trees**.

Start with b:

- Paths with `b`:
  - f, c, a, m → b (from T2)
  - f → b (from T3)
  - c → p → b (from T4)
- Create conditional pattern base for b:
  - {f, c, a, m}: 1
  - {f}: 1
  - {c, p}: 1

Now recursively find frequent itemsets in this conditional base.
Frequent itemsets with b:

- `{b}` : 3
- `{b, f}` : 2
- `{b, c}` : 2
- `{b, c, f}` : 1 (not enough support)
- `{b, p}` : 1 (not enough support)
- `{b, a}` : 1 (not enough support)

Next: p

Paths with p:

- f, c, a, m → p (T1, T5, T6)
- c → p (T4)

Pattern base:

- {f, c, a, m}: 3
- {c}: 1

Itemsets with p:

- `{p}` : 4
- `{p, f}` : 3
- `{p, c}` : 4
- `{p, a}` : 3
- `{p, m}` : 3
- `{p, c, a}` : 3
- `{p, f, c}` : 3
- `{p, f, a}` : 3
- `{p, f, m}` : 3
- `{p, f, c, a}` : 3

⚡ Repeat for m, a, c, f...

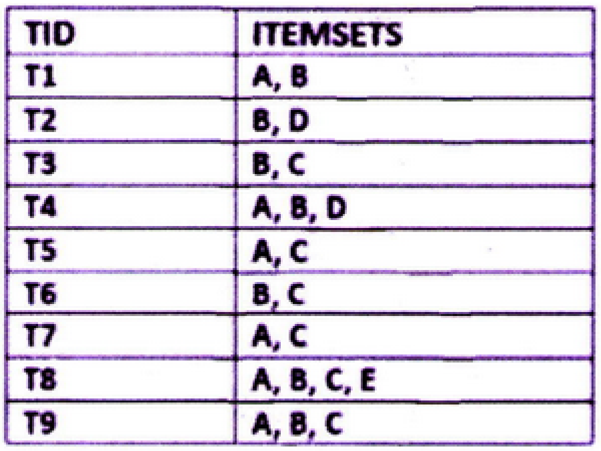
(We'll get many more combinations.)

Final Frequent Itemsets (min_sup = 3)

Here are some of the frequent itemsets:

- `{f}` (5)
- `{c}` (5)
- `{a}` (4)
- `{m}` (4)
- `{p}` (4)
- `{b}` (3)
- `{f, c}` (4)
- `{f, a}` (4)
- `{f, m}` (4)
- `{f, p}` (3)
- `{c, a}` (4)
- `{c, p}` (4)
- `{a, m}` (4)
- `{a, p}` (3)
- `{m, p}` (3)
- `{f, c, a}` (4)
- `{f, c, m}` (4)
- `{f, a, m}` (4)
- `{c, a, m}` (4)
- `{f, c, a, m}` (4)
- `{p, f, c, a}` (3)
- `{b, f}` (2) ❌
- `{b, c}` (2) ❌

10. State the Apriori principle in candidate generation. Find out the frequent item sets with minimum support of 2 using Apriori for the following data.



TID	ITEMSETS
T1	A, B
T2	B, D
T3	B, C
T4	A, B, D
T5	A, C
T6	B, C
T7	A, C
T8	A, B, C, E
T9	A, B, C

The **Apriori Principle** says:

> **"If an itemset is frequent, then all of its subsets must also be frequent."**

In simple words:

- If `{A, B, C}` is a frequent itemset, then `{A}`, `{B}`, `{C}`, `{A, B}`, `{A, C}`, and `{B, C}` must also be frequent.
- So, **if any subset is not frequent**, we can **skip checking its supersets**. This saves time by **reducing the number of candidate itemsets**.

Now let's apply Apriori Algorithm step by step to the data with min support = 2

TID	Items
T1	A, B
T2	B, D
T3	B, C
T4	A, B, D
T5	A, C
T6	B, C
T7	A, C
T8	A, B, C, E
T9	A, B, C

Step 1: Find frequent 1-itemsets (L1)

Count support for each item:

```
A → 6  ✅
B → 7  ✅
C → 6  ✅
D → 2  ✅
E → 1  ❌ (Remove)
```

L1 = {A}, {B}, {C}, {D}

Step 2: Generate candidate 2-itemsets (C2) using L1

```
{A, B}, {A, C}, {A, D}, {B, C}, {B, D}, {C, D}
```

Count their support

```
{A, B} → 4  ✅
{A, C} → 4  ✅
{A, D} → 1  ❌
{B, C} → 4  ✅
{B, D} → 2  ✅
{C, D} → 0  ❌
```

L2 = {A, B}, {A, C}, {B, C}, {B, D}

Step 3: Generate candidate 3-itemsets (C3) from L2

Only combine itemsets that share two items:

```
{A, B, C}
```

Count support

```
{A, B, C} → 2  ✅
```

L3 = {A, B, C}

Final Frequent Itemsets:

- **L1**: {A}, {B}, {C}, {D}
- **L2**: {A, B}, {A, C}, {B, C}, {B, D}
- **L3**: {A, B, C}