**🔹 Understanding the PCY Algorithm**

**1. What is the purpose of the PCY Algorithm, and how does it improve over the Apriori algorithm?**

**PCY (Park-Chen-Yu) Algorithm** is an optimized version of **Apriori** that reduces the number of candidate pairs by using **hashing and bitmaps**.  
🔹 **Improvement Over Apriori:**  
✅ **Less memory usage**: Instead of storing all candidate pairs, it uses a **hash table** (buckets).  
✅ **Faster pruning**: The **bitmap** efficiently eliminates infrequent pairs early.  
✅ **Reduces unnecessary computations** in the second pass.

**2. Why do we use hashing in PCY, and how does it help reduce memory usage?**

✅ Hashing allows us to **map item pairs into fixed-size buckets** instead of storing every pair explicitly.  
✅ Instead of storing all possible pairs, we **only store hash counts** in a bucket table.  
✅ If a bucket count is below the threshold, **all pairs in that bucket are discarded** in the second pass.

**3. What are the two main passes in the PCY algorithm, and what happens in each pass?**

* **Pass 1:**
  + Count individual item frequencies.
  + Use **hashing to count pairs in buckets**.
* **Pass 2:**
  + Identify frequent items.
  + Use the **bitmap to filter candidate pairs**.
  + Count frequent pairs.

**🔹 Code-Specific Questions**

**4. How does the hash\_pair function work, and why is hashing necessary in the second pass?**

def hash\_pair(self, pair):

return hash(pair) % self.num\_buckets

✅ **Explanation:**

* It applies Python’s **hash function** to a pair.
* The result is **modulo num\_buckets**, which ensures all pairs are placed into one of **100 buckets**.
* If a **bucket has a low count**, we **ignore all pairs in that bucket**, improving performance.

**5. Why did you use defaultdict(int) for storing item counts, pair counts, and bucket counts?**

✅ **defaultdict(int) initializes missing keys with 0 automatically**, so there's no need to check if a key exists before incrementing.

Example:

item\_counts = defaultdict(int)

item\_counts["apple"] += 1 # No KeyError!

🚀 **Benefit:** Cleaner code and avoids manual if key in dict: checks.

**6. Why did you sort the frequent items, frequent pairs, and association rules before displaying them?**

✅ Sorting helps **show the most important patterns first**.  
✅ **Sorting by frequency** helps businesses prioritize the most relevant insights.

Example:

sorted(frequent\_items.items(), key=lambda x: x[1], reverse=True)

🚀 **Benefit:** High-frequency patterns are easier to analyze.

**🔹 Spark & Data Handling**

**7. Why did you use Apache Spark to load the dataset instead of Pandas?**

✅ Spark handles **big data** better than Pandas, which loads everything into memory.  
✅ **Parallel processing** makes operations **faster**.  
✅ The dataset is **grouped efficiently** before conversion into Python lists.

**8. What does the groupBy("Member\_number", "Date").agg(collect\_set("itemDescription")) statement do?**

✅ Groups transactions **by customer (Member\_number) and date (Date)**.  
✅ **Collects all purchased items** for that transaction as a **set**.  
✅ Helps create a **list of item baskets**, which is needed for PCY.

Example:

| **Member\_number** | **Date** | **itemDescription** |
| --- | --- | --- |
| 1111 | 2023-01-01 | Milk |
| 1111 | 2023-01-01 | Bread |

✅ After groupBy → {1111, "2023-01-01"} → {"Milk", "Bread"}

**🔹 Association Rule Mining**

**9. How do you calculate the confidence of an association rule?**

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

✅ **Meaning:**

* If **70% of people who buy Milk also buy Bread**, confidence = **0.7**.
* High confidence **suggests strong relationships**.

**10. Why do we generate two confidence values for each item pair (Item1 → Item2 and Item2 → Item1)?**

✅ The relationship is **not always symmetrical**.

* **Example:** People who buy **Peanut Butter → Jelly** (High confidence ✅)
* But **Jelly → Peanut Butter** might have **low confidence** ❌.

**11. Why did you sort the association rules by confidence before displaying them?**

✅ **High-confidence rules** are more useful for decision-making.  
✅ Helps prioritize **strongest associations** for recommendations.

Example:

sorted(association\_rules, key=lambda x: x[2], reverse=True)[:30]

🚀 **Benefit:** Shows the most **impactful** rules first.

**🔹 Performance & Optimization**

**12. How does using hash buckets in PCY help improve performance compared to counting all pairs directly?**

✅ Instead of checking **all item pairs** (which is slow), we **hash** them into **fixed-size buckets**.  
✅ This reduces **memory usage** and speeds up **pruning** of infrequent pairs.

**13. What would happen if the number of hash buckets (num\_buckets) was too small?**

✅ **More collisions**, meaning many different pairs **end up in the same bucket**.  
✅ This can **keep infrequent pairs in the candidate list**, reducing efficiency.  
✅ Solution: **Increase num\_buckets**.

**🔹 Error Handling & Edge Cases**

**14. What would happen if a transaction (basket) contains only one item?**

✅ **Pairs cannot be formed**, so that basket is **ignored** in the second pass.  
✅ Example: {Milk} → No pairs possible.

**15. What modifications would you make if the dataset contained missing or corrupted values?**

✅ Use **data cleaning** steps in Spark:

df = df.dropna() # Remove missing values

🚀 **Benefit:** Ensures **high-quality** data.

**🔹 Practical Applications**

**16. How can the PCY algorithm be applied in real-world industries like retail or e-commerce?**

✅ **Retail:**

* Identifying **products frequently bought together**.
* **Optimizing store layout** (place related items nearby). ✅ **E-commerce:**
* **Recommendation systems**: "Customers who bought **X** also bought **Y**." ✅ **Fraud detection:**
* Detecting **suspicious transaction patterns**.

**🔹 Bonus: Advanced Enhancements**

**17. How would your implementation change if transactions contained more than just item descriptions (e.g., item categories or prices)?**

✅ Store additional information like **category** or **price ranges**.  
✅ Modify association rule mining to **group similar products**.

Example:

df.groupBy("Category").agg(collect\_set("itemDescription"))

🚀 **Benefit:** More meaningful insights **beyond simple pairs**.

**18. What challenges might arise when scaling this algorithm to distributed systems like Hadoop or Apache Flink?**

✅ **Data partitioning**: Ensuring **even distribution** across nodes.  
✅ **High memory usage**: Optimizing **bucket count** to reduce load.  
✅ **Parallelization**: Ensuring efficient **data shuffling** for pair counting.