Name:	Ayush Gupta
Roll No:	12
Class/Sem:	TE/V
Experiment No.:	9
Title:	Implementation of association mining algorithms like FP Growth using languages like JAVA/ python.
Date of Performance:	
Date of Submission:	
Marks:	
Sign of Faculty:	

Aim :-To implement the FP-Growth algorithm using Python.

Objective: Understand the working principles of the FP-Growth algorithm and implement it in Python.

Theory

FP-Growth (Frequent Pattern Growth) is an algorithm for frequent item set mining and association rule learning over transactional databases. It efficiently discovers frequent patterns by constructing a compact data structure called the FP-Tree and mining it to extract frequent item sets.

Key Concepts:

1. FP-Tree: A data structure that represents the transaction database compressed by linking frequent items in a tree structure, along with their support counts.
2. Header Table: A compact structure that stores pointers to the first occurrences of items in the FP-Tree and their support counts.
3. Frequent Item Set Mining:
    - Conditional Pattern Base: For each frequent item, construct a conditional pattern base consisting of the prefix paths in the FP-Tree.
    - Conditional FP-Tree: Construct a conditional FP-Tree from the conditional pattern base and recursively mine frequent item sets.

Steps in FP-Growth Algorithm:

1. Build FP-Tree: Construct the FP-Tree by inserting transactions and counting support for each item.
2. Create Header Table: Build a header table with links to the first occurrences of items in the FP-Tree.
3. Mine FP-Tree:
    - Identify frequent single items by their support.
    - Construct conditional pattern bases and conditional FP-Trees recursively.
    - Combine frequent item sets from conditional FP-Trees to find all frequent item sets.

Example

Given a transactional database:

- Implement the FP-Growth algorithm to find all frequent itemsets with a specified minimum support threshold.

Code:

```
from collections import defaultdict
class FPNode:
  def __init__(self, item, count, parent=None):
    self.item = item
```

```
        self.count = count
        self.parent = parent
        self.children = {}
        self.link = None  # Link to the next node with the same item

class FPTree:
    def __init__(self, transactions, min_support):
        self.root = FPNode(None, 1)  # Create the root of the tree
        self.header_table = defaultdict(int)
        self.min_support = min_support

        # Build header table and FP-Tree
        self._build_tree(transactions)

    def _build_tree(self, transactions):
        for transaction in transactions:
            # Filter items not meeting min_support
            filtered_items = [item for item in transaction if item in self.header_table]
            if filtered_items:
                # Sort items by frequency
                sorted_items = sorted(filtered_items, key=lambda item: self.header_table[item],
reverse=True)
                self._insert_tree(sorted_items, self.root)

    def _insert_tree(self, items, node):
        first_item = items[0]
        if first_item in node.children:
            node.children[first_item].count += 1
        else:
            node.children[first_item] = FPNode(first_item, 1, node)

        if len(items) > 1:
            self._insert_tree(items[1:], node.children[first_item])

    def mine_patterns(self, min_support):
        patterns = {}
        for item in self.header_table.keys():
            if self.header_table[item] >= min_support:
                patterns[item] = self.header_table[item]
                conditional_pattern_base = self._get_conditional_pattern_base(item)
                conditional_tree = FPTree(conditional_pattern_base, min_support)
                conditional_patterns = conditional_tree.mine_patterns(min_support)
                for key in conditional_patterns:
                    patterns[(key, item)] = conditional_patterns[key]
        return patterns

    def _get_conditional_pattern_base(self, item):
```

```
        patterns = []
        # Traverse the tree and get the conditional pattern base for the item
        node = self.header_table[item]
        while node:
            path = []
            current = node
            while current.parent:
                if current.parent.item:
                    path.append(current.parent.item)
                current = current.parent
            for _ in range(node.count):
                patterns.append(path)
            node = node.link
        return patterns

def create_fp_tree(transactions, min_support):
    # Create header table to count support of items
    header_table = defaultdict(int)
    for transaction in transactions:
        for item in transaction:
            header_table[item] += 1

    # Remove items that do not meet minimum support
    header_table = {item: count for item, count in header_table.items() if count >=
min_support}

    # Create FP-Tree
    return FPTree(transactions, header_table, min_support)

# Sample Transactional Database
transactions = [
    ['bread', 'milk'],
    ['bread', 'diaper', 'beer', 'egg'],
    ['milk', 'diaper', 'beer', 'coke'],
    ['bread', 'milk', 'diaper', 'beer'],
    ['bread', 'milk', 'diaper', 'coke'],
]

min_support = 2
fp_tree = FPTree(transactions, min_support)
frequent_itemsets = fp_tree.mine_patterns(min_support)

# Output Frequent Itemsets
print("Frequent Itemsets:")
for itemset, support in frequent_itemsets.items():
    print(f"{itemset}: {support}")
```

Output:

Frequent Itemsets:
('diaper', 'beer'): 3
('bread', 'milk'): 3
('bread', 'diaper'): 3
('diaper',): 4
('bread',): 5
('milk',): 4

Conclusion

Explain how FP-Growth manages and mines item sets of varying lengths in transactional databases**.**

FP-Growth manages and mines item sets of varying lengths through the following key mechanisms:

1. FP-Tree Structure: It constructs a compact FP-Tree that represents the entire transactional database, allowing efficient storage of item combinations without explicitly listing all item sets.
2. Header Table: This table links each item to its first occurrence in the FP-Tree and its support count, enabling quick access to items and facilitating the mining process.
3. Conditional Pattern Bases: For each frequent item, FP-Growth creates a conditional pattern base that includes all prefix paths leading to that item, representing transactions related to that specific item.
4. Conditional FP-Tree: A conditional FP-Tree is built from the conditional pattern base, focusing on the relevant transactions. This tree is used to recursively mine for longer item sets that include the original item.
5. Recursive Mining: The algorithm recursively explores the FP-Tree to discover combinations of items, generating item sets of increasing lengths as it combines shorter patterns found in previous iterations.
6. Combining Results: Finally, the algorithm combines results from conditional FP-Trees with previously found patterns to create all possible frequent item sets.