**ASSIGNMENT 1**

This is a detailed and interesting assignment. Here's a suggested approach to help you get started with the Rule Engine:

**1. Data Structure for AST**

You can represent the AST as a tree of Node objects with a structure like this:

**Python:**

class Node:

def \_\_init\_\_(self, node\_type, value=None, left=None, right=None):

self.type = node\_type # "operator" or "operand"

self.value = value # e.g., 'age > 30'

self.left = left # left child (Node)

self.right = right # right child (Node)

**Example**: For a simple condition age > 30 AND department = 'Sales', the tree might look like:

**Shell :**

AND

/ \

age>30 department='Sales'

**2. Data Storage**

You can choose a database like **PostgreSQL** or **MongoDB**. Here's an example schema:

**PostgreSQL Schema**:

* **Rules Table**:

CREATE TABLE rules (

id SERIAL PRIMARY KEY,

rule\_string TEXT,

ast\_json JSONB

);

You can store the AST as JSON, allowing flexibility to modify it.

**MongoDB Schema**:

{

"rule\_id": ObjectId("..."),

"rule\_string": "(age > 30 AND ...)",

"ast": {

"type": "operator",

"value": "AND",

"left": { "type": "operand", "value": "age > 30" },

"right": { "type": "operand", "value": "department = 'Sales'" }

}

}

**3. API Design**

1. **create\_rule(rule\_string)**:
   * Parse the rule\_string and construct the AST. You can use Python's ast module or a custom parser to tokenize and build the tree.

Example (simple parsing logic):

**Python :**

def create\_rule(rule\_string):

# Parse rule\_string, build Node-based AST

# Tokenize by AND/OR, then build tree

pass

1. **combine\_rules(rules)**:
   * Combine multiple rules efficiently by merging ASTs. One heuristic might be to group conditions based on common operators.
2. **evaluate\_rule(json\_data)**:
   * Traverse the AST and evaluate conditions using the json\_data.

Example:

**Python:**

def evaluate\_rule(ast, data):

if ast.type == 'operator':

left\_result = evaluate\_rule(ast.left, data)

right\_result = evaluate\_rule(ast.right, data)

if ast.value == 'AND':

return left\_result and right\_result

elif ast.value == 'OR':

return left\_result or right\_result

elif ast.type == 'operand':

# Evaluate conditions, e.g., 'age > 30'

return eval\_operand(ast.value, data)

**4. Test Cases**

* Create individual rules and verify AST representation.
* Test combinations and evaluate results against sample data.

**Bonus Points**

* **Error Handling**: Add checks for invalid rules or malformed JSON data.
* **Validations**: Ensure attributes like age, department exist in a predefined catalog.
* **Rule Modification**: Provide functionality to modify existing rules by changing operators or values.

**ASSIGNMENT 2**

For the second application, here's a breakdown to help you get started with the **Real-Time Data Processing System for Weather Monitoring**:

**1. Data Retrieval from OpenWeatherMap API**

* **API Key**: First, sign up on [OpenWeatherMap](https://openweathermap.org/) and get an API key. You’ll need this to fetch real-time weather data.
* **API Call**: Write a function to call the API at configurable intervals (e.g., every 5 minutes):

**Python :**

import requests

import time

def get\_weather\_data(city, api\_key):

url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api\_key}"

response = requests.get(url)

return response.json()

# Call at intervals

while True:

for city in ['Delhi', 'Mumbai', 'Chennai', 'Bangalore', 'Kolkata', 'Hyderabad']:

weather\_data = get\_weather\_data(city, YOUR\_API\_KEY)

# Process and store data

time.sleep(300) # Sleep for 5 minutes

**2. Processing the Data**

* **Convert temperature from Kelvin to Celsius**:

**Python :**

def kelvin\_to\_celsius(kelvin):

return kelvin - 273.15

* **Extract weather parameters**: From the API response, retrieve necessary fields like temp, feels\_like, main, etc.

**3. Daily Rollups and Aggregates**

* For each city, store the weather updates in a database (e.g., PostgreSQL or MongoDB) with timestamps.
* At the end of each day, calculate aggregates such as:
  + **Average temperature**: (sum of temperatures) / number of updates
  + **Maximum/Minimum temperature**: Track the highest and lowest temperatures during the day.
  + **Dominant weather condition**: The condition that appears most frequently (you can use a counter for occurrences).

**Example Data Schema**:

**SQL :**

CREATE TABLE weather\_summary (

id SERIAL PRIMARY KEY,

city VARCHAR(50),

date DATE,

avg\_temp FLOAT,

max\_temp FLOAT,

min\_temp FLOAT,

dominant\_condition VARCHAR(50)

);

**4. Alerting System**

* Allow users to configure thresholds (e.g., temp > 35°C).
* Track the last few weather updates and trigger alerts if thresholds are breached.

**Python:**

def check\_alert(temp, threshold):

return temp > threshold

* For alerts, you can either print them to the console or implement an email notification system using libraries like smtplib.

**5. Visualizations**

* Use a library like **Matplotlib** or **Plotly** to create graphs for daily summaries and trends over time.
* You could also use **Dash** or **Flask** to build a simple web UI to display these visualizations.

**6. Test Cases**

* **System Setup**: Ensure proper connection to the API with the correct API key.
* **Data Retrieval**: Test retrieval at configurable intervals and ensure the correct parsing of data.
* **Temperature Conversion**: Validate conversions from Kelvin to Celsius (or Fahrenheit).
* **Daily Summaries**: Simulate weather data over several days and ensure the correct calculation of aggregates.
* **Threshold Alerts**: Simulate different weather conditions to ensure alerts trigger only when thresholds are crossed.

**Bonus**

* **Extended Parameters**: You can add extra weather parameters like humidity and wind speed from the API response and incorporate them into the rollups and aggregates.
* **Weather Forecasts**: The OpenWeatherMap API also provides forecast data. You can add functionality to fetch and summarize predicted conditions.

Would you like help with any specific part of this application? We can also work on how to integrate everything and structure the codebase for clarity and maintainability.

**GITHUB LINK : https://github.com/psraja555/ZEOTAP**