STEVENS
INSTITUTE OF TECHNOLOGY
1870

Final Project Presentation

CS 562 - DBMS II

Team Name: Startreks

Team Members: Soham Talekar and Chinmay Dharap

Date: 05-07-2024

Overview

In-memory query processing (without using Postgresql engine).

- A conceptually simple query processor
- Optimized algorithmic steps
- In memory steps
- Quick execution
- Decoupling of group by and select

Database Management System

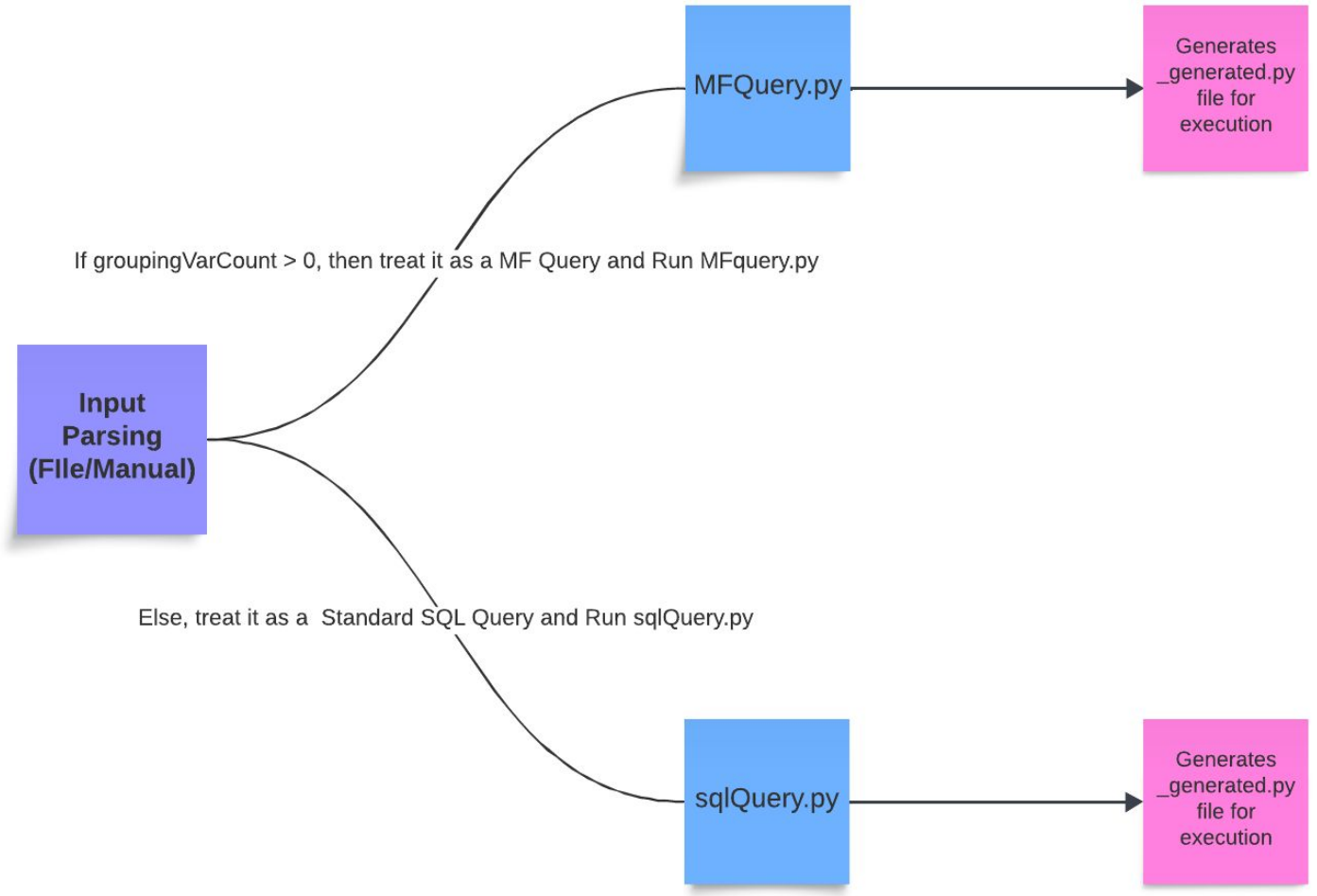
Technology Stack

- Utilizing **Postgresql**
- **Python** as our base language for query processor
- **psycopg2** python driver to make a stable connection with the database and load in all the data into memory
- Library **PrettyTable** to tabulate the result

Project Workflow

- You can input MF or SQL Queries, 2 options:
  - An input File
  - Textual Input
- The processor parses through the file.
- Determination of query type based on the number of grouping variables.
- Selection of Suitable Algorithm:
  - SQL Algorithm
  - MF Query Algorithm
- Generation of a python File based on the input
- Execution of the python file

Workflow Diagram



Approach for SQL Queries

- Our code first fetches all the data from postgresql db using basic “select *” query and stores it in memory for processing. Once the query results are stored we are filtering the results for potential WHERE conditions if there are any in the input using a separate parsing function “**apply_conditions()**”
- This functions identifies all the WHERE conditions and splits the conditions based on **and** & **or** operator. It is one of the Heuristic Optimization method taught by Prof. Kim that helps us to filter out most of the unnecessary data the start
- Once the data has been filtered we implement the main logic for processing SQL Queries which is as follows:
  - For each row in the data, initialize a Key-Value pair where the Key will be made of grouping attribute of that current row in the table and Value (dictionary) will store the columns data for that given row
  - **if** key not in MF_Struct, then create a new entry. Then loop through the fVects & initialize the values of each aggregate function to calculate
    - Initialize COUNT to 1, SUM to current row’s quant value , MIN & MAX to current quant values and AVERAGE to a dict with 3 component (SUM, COUNT, AVG). Each is calculated and stored when row gets updated. Finally inserts this new row in MF_Struct
  - **else**, (means the row already exists in MF_Struct) update the existing entry and update the comma separated fVector attributes
  - Finally once we have the output in MF_Struct,
    - **if** there are any Having conditions, check them based on eval string ('>', '<', '==', '<=', '>=', 'and', 'or', 'not', '*', '/', '+', '-'). Then looping through the earlier **updated MF_Struct** again to fulfill the having condition and store it in a new dictionary **row_info.** Lastly adding each row from row_info to output.
    - **else**, (means no having condition) all MF_Struct rows will be output

Approach for MF Queries

1. A simple approach as guided by professor Kim and the TAs:

2. Initialization of the loop, based on the number of grouping variables
    a. i.e 1 variable the two scans
    b. 2 variables then three scans and so on
3. On the 0th iteration initialization of the MF structure or Htable
    a. Basically a dictionary storing the grouping attributes as keys
    b. And initializing each of the required select attribute as the value of this key
4. On later iterations:
    a. Loop through F-vector (aggregates)
    b. If the grouping variable matches corresponding iteration, compute all the values from the predicate (such that)
    c. Store ‘such that’ values at the corresponding keys in the dictionary
5. Due to the O(1) time complexity of dictionaries, (internal hash tables) , the time complexity for finding keys and updating corresponding values is very quick
6. Finally after computing the aggregates, run a scan for each of the keys of the MF structure
    a. Having clause if it exists is computed
    b. The Final output is generated too
7. Display the results

Input Query Structure

SELECT ATTRIBUTE(S):
cust, 1_avg_quant, 2_avg_quant, 3_avg_quant, 3_sum_quant
NUMBER OF GROUPING VARIABLES(n):
3
GROUPING ATTRIBUTES(V):
cust
F-VECT([F]):
1_sum_quant, 1_avg_quant, 2_avg_quant, 3_sum_quant, 3_avg_quant, 3_sum_quant
SELECT CONDITION-VECT([C]):
1.cust=cust and 1.state='NY' and 1.year=2016
2.cust=cust and 2.state='CT' and 2.year=2016
3.cust=cust and 3.state='NJ' and 3.year=2016
HAVING_CONDITION(G):
1_avg_quant > 2_avg_quant and 1_avg_quant >= 3_avg_quant

Our engine can parse both types of inputs: **file as CLI and manual input**

Limitations of engine

A few limitations we ran into:

- No support for EMF syntax
- A faster execution, provided a single scanned process was feasible
- Prone to error if improper input formatting
- no robust error handling
- no error checking for table presence

Recap & Future Scope

- Can process SQL queries
- Can process MF Queries
- Capable of taking a properly formatted text file as an input and a manual input
- Uses cursor scan to fetch the whole database and stores in memory for further operations
- Can’t process EMF queries

STEVENS
INSTITUTE OF TECHNOLOGY
1870

THANK YOU

Stevens Institute of Technology
1 Castle Point Terrace, Hoboken, NJ 07030