



# CS562 – The Project

Due Date: 12/15/2023 (Fri.)

**Objectives:** To become familiar with the concept of *Complex Ad-Hoc OLAP Query Processing* and understand how a simple extension of SQL can help simplify the expression of complex ad-hoc OLAP queries.

**Introduction:** Your mission in this project is to build a query processing engine for Ad-Hoc OLAP queries. The query construct is based on an extended SQL syntax known as MF and EMF queries (i.e., Multi-Feature and Extended Multi-Feature queries).

**PROBLEM:** Ad-hoc OLAP queries (also known as multi-dimensional queries) expressed in standard SQL, even the simplest types, often lead to complex relational algebraic expressions with multiple joins, group-bys, and sub-queries. When faced with the challenges of processing such queries, traditional query optimizers do not consider the “big picture”. Rather, they try to optimize a series of joins and group-bys, leading to poor performance.

**A SOLUTION:** Provide a syntactic framework to allow succinct expression of ad-hoc OLAP queries by extending the *group-by* statement and adding the new clause, *such that*, and in turn, provide a simple, efficient and scalable algorithm to process the queries.

Please refer to the following two research articles for further details on the new syntax and the corresponding processing algorithm:

- “*Querying Multiple Features of Groups in Relational Databases*”, D. Chatziantoniou and K. Ross
- “*Evaluation of Ad Hoc OLAP: In-Place Computation*”, D. Chatziantoniou

The MF and EMF queries take advantage of the grouping variables and such that clause to avoid the need for using multiple sub-queries and joins, and thereby making the expression of the queries more succinct.

**Grouping Variables** represent subsets of tuples within (or outside) each group (esp., in MF/EMF queries) and the **Such That** clause provides predicates to define the ranges of the grouping variables (i.e., they act as the ‘where’ clauses for the grouping variables). A new relational operator  $\Phi$  is introduced to capture the essence of the extended SQL (of the grouping variables and the corresponding *such that* clause). Additionally, there is an evaluation algorithm for the new relational operator that is simple and scalable.

The new relational operator  $\Phi$  has the following arguments:

- S = List of projected attributes for the query output
- n = Number of grouping variables
- V = List of grouping attributes
- [F] = {F0, F1, ..., Fn}, list of sets of aggregate functions. Fi represents a list of aggregate functions for each grouping variable.
- [ $\sigma$ ] = { $\sigma_0$ ,  $\sigma_1$ , ...,  $\sigma_n$ }, list of predicates to define the ranges for the grouping variables.
- G = Predicate for the *having* clause

The evaluation algorithm is as follows, and it utilizes a compact data structure known as '**mf-structure**' – it holds the data corresponding roughly to the output of the MF/EMF query.  $H$  represents the mf-structure in the following algorithm. For further details on the 'mf-structure', please refer to the second article, "*Evaluation of Ad Hoc OLAP: In-Place Computation*" (Note that the following algorithm is for the EMF queries, but it also works for the MF queries. The algorithm can be tailored for the MF queries by modifying the third line to "for the entries of  $H$  with matching grouping attributes").

```
for scan sc=0 to n {
  for each tuple t on scan {
    for all entries of H,
      check if the defining condition of grouping var
      Xsc is satisfied. If yes, update Xsc's aggregates of the entry
      appropriately.
      X0 denotes the group (the defining condition of X0 is X0.S = S,
      where S denotes the grouping attributes.)
  }
}
```

## Project Description:

The following is an outline of the key functionalities of the query processing engine (QPE you will be implementing for the project:

- The MF and EMF queries for the project will be based on the following *schema*
  - sales (cust, prod, day, month, year, state, quant, date)  
The table stores the information about the purchases of a product by a customer on a date and state for a sale amount.

If time permits, we may try using multiple input tables – an idea that is well within the realm of MF/EMF queries.

- The *input* for the query processing engine is the list of arguments for the new operator  $\Phi$  (in place of the actual query represented in SQL) – i.e., you can assume the query has already been transformed into a corresponding relational algebraic expression. For example, for the following query,

```
select cust, sum(x.quant), sum(y.quant), sum(z.quant)
  from sales
 group by cust: x, y, z
such that x.state = 'NY'
       and y.state = 'NJ'
       and z.state = 'CT'
having sum(x.quant) > 2 * sum(y.quant) or avg(x.quant) >
avg(z.quant);
```

you can expect an input such as

```
SELECT ATTRIBUTE(S):
cust, 1_sum_quant, 2_sum_quant, 3_sum_quant
NUMBER OF GROUPING VARIABLES(n):
3
GROUPING ATTRIBUTES(V):
cust
F-VECT([F]):
1_sum_quant, 1_avg_quant, 2_sum_quant, 3_sum_quant, 3_avg_quant
SELECT CONDITION-VECT([σ]):
1.state='NY'
2.state='NJ'
```

```
3.state='CT'
HAVING CONDITION(G):
1_sum_quant > 2 * 2_sum_quant or 1_avg_quant > 3_avg_quant
```

- Given the input as described above, the query processing engine **generates a program** (written in Python, C, C++, Java, etc.) which implements the evaluation algorithm mentioned earlier. The generated program (for a given MF/EMF query) goes against the `sales` table stored in the PostgreSQL database and generates the output corresponding to the query represented by the input. The generated program can be separately compiled and executed to produce the final output for the input query.

### Project Plan:

The following is an outline of the project schedule to help you manage your own progress.

| Date       | Your Query Processing Engine should be able to...   |
|------------|---|
| 11/10/2023 | Read the input and generate a program to create the 'mf-structure' (in memory) – i.e., the generated program contains a list of type/class definitions and variables for the 'mf-structure'.  |
| 11/17/2023 | <p>Separately generate a program to fulfill a simple query. Use the following as a sample query:</p> <pre>select cust, prod, avg(quant), max(quant)   from sales  where year=2009 group by cust, prod</pre> <p>Make sure the generated program computes the aggregate functions (e.g., avg, max) internally without relying on the PostgreSQL DBMS engine (i.e., the only allowed DB operation is a cursor/scan-based processing of the underlying table).</p> <p>The main purpose of this exercise is to help you “visualize” the implementation of the evaluation algorithm in the generated program.</p> |
| 12/01/2023 | Generate a program to fulfill a simple MF/EMF query that requires a single scan (i.e., an MF/EMF query with one grouping variable).   |
| 12/15/2023 | Generate a program to fulfill MF/EMF queries requiring multiple scans.  |

### Grading:

This is a team project – a team consisting of up to 2 members – and the grading of the project will be largely based on the following guideline (similar to other programming assignments you have worked on in the past).

- (80 pts.) Logic/Correctness
- (20 pts.) Programming Style (e.g., comments, indentation, use of functions, etc.). You must include a program header, function header, etc. to clearly state what your program and functions are designed to do. Also, for inline comments, please state clearly the purpose of those statements – for you as the programmer and to help others better understand your programming logic.



A program with compilation errors will earn no more than 50 points.

I will schedule a series of sessions for Friday, December 15, 2023 (day after the last day of the semester), and your team will be presenting a demo of your projects for 15-20 min. During the demo, you will be asked to input different queries, modify the queries, and in certain cases, modify the code for some simple changes to the algorithms.

Please prepare a list of queries to present at the demo prior to the presentation, so we can move quickly through your presentation portion of the demo.

***Running the  
Query  
Processing  
Engine***

- Your query processing engine must be able to read the input from both a file and interactively from the user.
- The generated program can be manually compiled and executed to generate the final output or the engine itself can directly compile and execute the generated program and output the result of the input query.

***Submission:***

Please submit all of your source code (files) with your names and CWIDs (Campus Wide IDs). Include all other files/libraries required to compile and run your code.



## Project Grade Sheet for \_\_\_\_\_

| Major Area         | Item   | Max        | Deduct | Score | %           | Total |
|--------------------|--|------------|--------|-------|-------------|-------|
| <i>Compilation</i> | If fails, subtract ...                                 | <b>50</b>  |        |       |             |       |
| <i>Logic</i>       |  |            |        |       |             |       |
|                    | <b>Total</b>   | <b>100</b> |        |       | <b>80%</b>  |       |
| <i>Style</i>       | Header Comment<br>(Overall comments about the project) | 10         |        |       |             |       |
|                    | Function Comments                                      | 20         |        |       |             |       |
|                    | Line Comment   | 20         |        |       |             |       |
|                    | Meaningful Names<br>(for functions, variables, etc.)   | 10         |        |       |             |       |
|                    | Strings – Left Justified                               | 15         |        |       |             |       |
|                    | Numbers – Right Justified                              | 15         |        |       |             |       |
|                    | Modular design (use of classes, methods/functions)     | 10         |        |       |             |       |
|                    |  |            |        |       |             |       |
|                    |  |            |        |       |             |       |
|                    | <b>Total</b>   | <b>100</b> |        |       | <b>20%</b>  |       |
| <i>Total</i>       |  | <b>100</b> |        |       | <b>100%</b> |       |