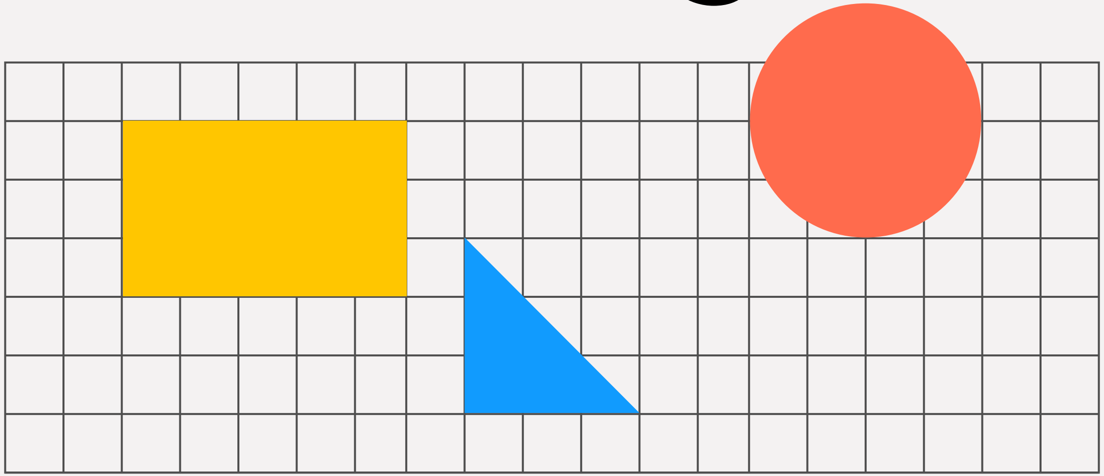
Extended SQL Processing Tool



Schema Squad:
Ayushi Gupta, Lilli Nappi, Justine Wang

Standard SQL Setbacks

- Grouping and aggregate queries can be difficult to express in standard SQL
  - Can cause conceptual and implementation problems to database system
  - Complex SQL expressions with multiple joins, group by clauses, and subqueries can be difficult to comprehend

- Why is this important?
  - Aggregation is intensively used in a wide range of applications
  - Crucial for query languages to be able to succinctly and *efficiently* express these queries

Solution - MF Queries

Standard SQL

```
SELECT x.product, sum(x.quantity),
sum(y.quantity), sum(z.quantity)
FROM Sales x, Sales y, Sales z
WHERE x.product=y.product AND
x.month=1 AND x.year=1997 AND
y.product=z.product AND y.month=2 AND
y.year=1997 AND z.month=3 AND
z.year=1997
GROUP BY x.product
```

→

Multiple-Feature Query

```
SELECT product,
sum(X.quantity),
sum(Y.quantity),
sum(Z.quantity)
FROM Sales
WHERE year=''1997''
GROUP BY product : X, Y, Z
SUCH THAT X.month = 1,
Y.month = 2,
Z.month = 3
```

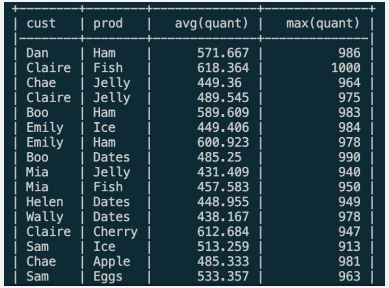
The multiple-feature query extends the GROUP BY clause to include a SUCH THAT clause which is used to define the grouping variables.

Project Overview

Input

Python Generator

Output



cust	prod	avg(quant)	max(quant)
Dan	Ham	571.667	986
Claire	Fish	618.364	1000
Chae	Jelly	449.36	964
Claire	Jelly	489.545	975
Boo	Ham	589.609	983
Emily	Ice	449.406	984
Emily	Ham	600.923	978
Boo	Dates	485.25	990
Mia	Jelly	431.409	940
Mia	Fish	457.583	950
Helen	Dates	448.955	949
Wally	Dates	438.167	978
Claire	Cherry	612.684	947
Sam	Ice	513.259	913
Chae	Apple	485.333	981
Sam	Eggs	533.357	963

Phi Operators

```
s = prod, x_sum_quant, y_sum_quant,
z_sum_quant
n = 3
v = prod
F = [x_sum_quant, y_sum_quant,
z_sum_quant]
sigma = [X.month=1, Y.month=2,
Z.month=3]
G = null
```

OR

MF Query

```
SELECT prod,
sum(X.quant),
sum(Y.quant),
sum(Z.quant)
FROM sales
WHERE year=2017
GROUP BY prod : X, Y, Z
SUCH THAT X.month = 1,
Y.month = 2, Z.month = 3
```

→

process_info()
read_file()
schema_info()
H_table()

→

mf_struct

H table

SQL Limitations

Upon execution of the below SQL query upon the Sales table, the system is unable to process the query structure. It conducts a 3-way self-join and computes the aggregate functions over the entire table. This method of evaluation can be very expensive and can produce inaccurate results, as shown below.

```
SELECT x.prod, sum(x.quant),
sum(y.quant), sum(z.quant)
FROM Sales x, Sales y, Sales z
WHERe x.prod=y.prod AND x.month=1
AND
  x.year=2017 AND y.prod=z.prod AND
  y.month=2 AND y.year=2017 AND
  z.month=3 AND z.year=2017
GROUP BY x.prod
```

	prod character varying (20)	sum bigint	sum bigint	sum bigint
1	Apple	2099526	2261644	2119390
2	Butter	2055420	2459985	2545240
3	Cherry	2498560	2106560	2622080
4	Dates	662700	819450	570900
5	Eggs	3625622	1875236	4060121
6	Fish	2148800	3378880	3266560
7	Grapes	1295232	1355264	1308864
8	Ham	3938060	3444120	3187386
9	Ice	1535688	1868832	1177092
10	Jelly	1953567	1303731	1430247

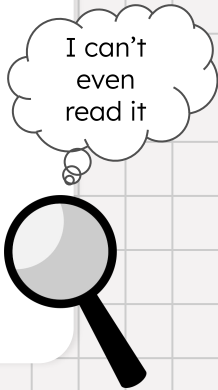
To rub it in some more…

This is the equivalent standard SQL query that generates a table with the correct results.

	prod character varying (20)	jan_sum numeric	feb_sum numeric	mar_sum numeric
1	Apple	7341	7343	11645
2	Butter	7212	9647	7880
3	Cherry	9760	6583	8194
4	Dates	4418	5463	5709
5	Eggs	12677	3706	13579
6	Fish	5372	10559	10208
7	Grapes	6746	5294	6817
8	Ham	8561	9567	7699
9	Ice	9141	8652	4671
10	Jelly	8457	9117	5239

```
SELECT x.prod, SUM(x.sum_quant) AS jan_sum,
              SUM(y.sum_quant) AS feb_sum,
              SUM(z.sum_quant) AS mar_sum
FROM
    (SELECT prod, SUM(quant) AS sum_quant
     FROM sales
     WHERE month = 1 AND year = 2017
     GROUP BY prod) x
JOIN
    (SELECT prod, SUM(quant) AS sum_quant
     FROM sales
     WHERE month = 2 AND year = 2017
     GROUP BY prod) y
ON x.prod = y.prod
JOIN
    (SELECT prod, SUM(quant) AS sum_quant
     FROM sales
     WHERE month = 3 AND year = 2017
     GROUP BY prod) z
ON x.prod = z.prod
GROUP BY x.prod;
```

(generated by ChatGPT)



Code Implementation

However, implementing an equivalent MF Query, using our generator, results in the correct implementation of the aggregate functions.

```
SELECT prod, sum(X.quant),
sum(Y.quant), sum(Z.quant)
FROM sales
WHERE year=2017
GROUP BY prod : X, Y, Z
SUCH THAT X.month = 1, Y.month = 2,
Z.month = 3
```

prod	sum(x.quant)	sum(y.quant)	sum(z.quant)
Apple	7341	7343	11645
Ham	8561	9567	7699
Fish	5372	10559	10208
Cherry	9760	6583	8194
Grapes	6746	5294	6817
Ice	9141	8652	4671
Jelly	8457	9117	5239
Butter	7212	9647	7880
Dates	4418	5463	5709
Eggs	12677	3706	13579

Code Breakdown

Input Parsing: read_file()

Reads an MF query file and each line is checked for specific keywords, and it extracts its components
Inputs: filename
Outputs select, From, where, group_by, such_that, having:

```python
def read_file(filename):
    """
    Reads an Extended SQL query and extracts its operands
    """
    with open(filename, 'r') as file:
        lines = file.readlines()
        for line in lines:
            line = line.lower().strip() # make everything lowercase

        # initalize vars
        select = ""
        From = ""
        where = ""
        group_by = ""
        such_that = ""
        having = ""

        # Extract info
        if len(lines) > 0:
            select = lines[0][7:].strip().lower()
        if len(lines) > 1:
            From = lines[1][5:].strip().lower()
        if len(lines) > 2:
            if "where" in lines[2].lower():
                where = lines[2][6:].strip().lower()
```

Schema Information: schema_info()

connects to PostgreSQL database and queries the INFORMATION_SCHEMA.COLUMNS table, retrieving column names and their data types and max lengths for the sales table

```python
def schema_info():
    """
    This retrieves schema infomation from the database using 'information_schema.columns'
    """
    # hard coded query to send to database
    query = f"""
    SELECT COLUMN_NAME, DATA_TYPE, CHARACTER_MAXIMUM_LENGTH
    FROM INFORMATION_SCHEMA.COLUMNS
    WHERE TABLE_SCHEMA = 'public' and TABLE_NAME = 'sales'
    """
    load_dotenv()

    user = os.getenv('USER')
    password = os.getenv('PASSWORD')
    dbname = os.getenv('DBNAME')

    conn = psycopg2.connect("dbname="+dbname+" user="+user+" password="+password,
                            cursor_factory=psycopg2.extras.DictCursor)
    cur = conn.cursor()
    cur.execute(query)
    data = cur.fetchall()
    conn.close()
    return data # return schema data
```

Code Breakdown

Query Components: process_info()

processes SELECT, GROUP BY, SUCH THAT, and aggregates
populates the mf_struct object
Outputs: group by vars and F vect: list of dictionaries for each aggregate

```python
def process_info(select, group_by, such_that, having, mf_struct, schemaData):
    """
    This function processes an MF query and extracts its information,
    populating mf_struct with 6 operands of Phi
    """
    V = [] # list of grouping attributes
    F_VECT = [] # list of aggregate functions

    # potential aggregates
    aggregates = ["sum", "count", "avg", "min", "max"]

    # potential group by attributes
    groupAttrs = ["cust", "prod", "day", "month", "year", "state", "quant", "date"]

    # add projected vals to struct.s
    mf_struct.s = select

    # add group_by vars to mf_struct.v
    group_by_vars = []
    if len(such_that) == 0:
        group_by = group_by.split(",") # normal SQL query (no such that conditions, group by seperated by comma)
    else:
```

SUCH THAT Conditions: process_conditions()

Takes in mf_struct and group by variables to separate them based on the such that clause.
Returns such that conditions

```python
def process_conditions(mf_struct, group_by_vars):
    """
    Processes the such that conditions in mf_struct, seperating them
    by group var, column name, and its condition
    """
    # Loop through each predicate in sigma (such_that conditions)
    operators = ['==', '!=', '>', '<']
    conditions = {gv: [] for gv in group_by_vars}
    for predicate in mf_struct.sigma:
        for op in operators:
            if op in predicate:
                # Split predicate to get the gv value, column name and condition value
                gv_col , cond = predicate.strip().split(op)
                gv, col = gv_col.strip().split('.')
                gv = gv.strip()
                conditions[gv]
```

Code Breakdown

H-table Construction: H_table

Loops through database rows, groups rows based on attributes in GROUP BY, checks if SUCH THAT conditions are satisfied using eval_conditions, and calculates aggregate values for SUM, AVG, MAX, etc

```
for row in cur:
    # only include unique rows based on group vars
    if (len(mf_struct.v) != 0):
        row_combo = tuple(row[col_name.index(gv)] for gv in mf_struct.v) # find index of group var based 
    else: # if no group by then use entire row as tuple
        row_combo = row

    # if not in unique then add it (group by function)
    if row_combo not in unique:
        unique.append(row_combo)
    # if there are aggregates, keep track of their values
    if len(F_VECT) != 0:
        for f in F_VECT:
            # if tuple of group by vals not already in agg_values, add new row
            if row_combo not in agg_values[f['agg']]:
                agg_values[f['agg']][row_combo] = []

            # filter based on such that conditions
            if (len(such_that) != 0):
                conditions = process_conditions(mf_struct, group_by_vars)
                if eval_conditions(row, conditions, f) == False:
                    continue # if it does not meet such that conditions, skip that row for the aggregate

            # get rid of period and only have column name to find index
            if '.' in f['arg']:
                name = f['arg'].split('.')[1]
                agg_values[f['agg']][row_combo].append(row[col_name.index(name)])
```

HAVING clause

Filters the H-table based on the HAVING clause and normalizes column names for compatibility

```
def preprocess_having_clause(having, H):
    """
    Translates HAVING clause into a pandas-compatible query string.
    """
    having = having.replace("=", "==").replace("<>", "!=")
    for col in H.columns:
        if col in having:
            having = having.replace(col, f"`{col}`")
    return having
```

Technical Stack & Limitations

Python for the main program logic, PostgreSQL for database and querying.
Python Libraries:

- *psycopg2* for interacting with PostgreSQL
- *pandas* for data computation and dataframe
- *tabulate* for formatting output tables
- *dotenv*: for loading database credentials

Tools:

- Query input is read from .txt files

Limitations:

- Our program performs multiple full scans of the database table to compute aggregates, applies SUCH THAT conditions, and builds the H-table, resulting in performance inefficiencies for large datasets, since minimal scanning techniques (e.g., combining scans or reducing redundant operations) are not implemented
- Additional limitations include hard coded columns (cust, prod, etc.) making our schema dependency static, possible performance overhead for large datasets due to pandas, operators sometimes ambiguous (i.e. use of minus in splitting/dates), and lack of query optimization as our aggregates are calculated
- In the future, these limitations, implementing minimal scanning, making the columns dynamics, and accounting for scalability can be addressed and improved upon.