College of Computing and Data Science

SC3020 Database System Principles
Project 2 Report (Group 2)

Name	Email	Matric Number	Contribution
Tan Kok Ann Jeff	JTAN564@e.ntu.edu.sg	U2121621D	Code Implementation and Report
Jarel Tan Zhi Wen	JARE0004@e.ntu.edu.sg	U2121582H	Code Implementation and Report
Lui Shi Ying	LU0018NG@e.ntu.edu.sg	U2121282F	Code Implementation and Report
Solomon Duke Tneo Yruan Rui	SOLO0003@e.ntu.edu.sg	U2122205J	Code Implementation and Report
Tan Jing Han	JTAN576@e.ntu.edu.sg	U2121577J	Code Implementation and Report

Introduction

The goal of this project is to create a pipe syntax, an alternative representation of the Structured Query Language (SQL), based on a given SQL query. For context, SQL pipe syntax is a more flexible SQL version as it can be written in any order, unlike traditional SQL, where the query must follow a specific order (Laithy, 2024). With this advantage, it not only enhances readability but also provides clearer insight into the sequence in which query operations are executed.

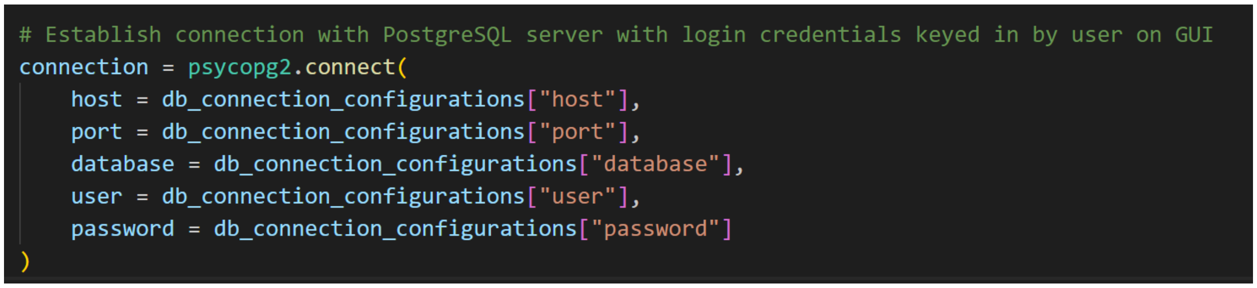
Section 1: Dataset

For the dataset, we used the recommended TPC-H data for testing. The TPC-H Tools v3.0.1.zip file was downloaded from the tpc.org website. After extracting and converting the nine .tbl files to .csv files, we imported them into a PostgreSQL database named "TPC-H". To prepare the database, we first created nine tables using the recommended "SQL commands for creating TPC-H data tables", executed through the "run SQL query" tool. Subsequently, the CSV files were manually imported into each table with the pipe character "|" as the delimiter. The tables were added in the following order: region.csv, nation.csv, supplier.csv, customer.csv, part.csv, partsupp.csv, customer.csv, orders.csv, lineitem.csv.

Section 2: Preprocessing

The code begins with the db_connect_and_obtain_qep function, which uses the .connect() function, designed to establish a connection session between the Python application and a PostgreSQL database using the psycopg2 library. It takes in the five necessary connection parameters:

i. Host - host name (eg: localhost)
ii. Port - port number (eg: 5432)
iii. Database - name of database (eg: TPC-H)
iv. User - username
v. Password - user's password



```python
# Establish connection with PostgreSQL server with login credentials keyed in by user on GUI
connection = psycopg2.connect(
    host = db_connection_configurations["host"],
    port = db_connection_configurations["port"],
    database = db_connection_configurations["database"],
    user = db_connection_configurations["user"],
    password = db_connection_configurations["password"]
)
```

The five parameters above will be taken from the login credentials keyed in by the user on our Graphical User Interface (GUI).

After a connection is securely established, a cursor object is created to facilitate the execution of SQL queries with the database. The cursor then uses the .execute function, takes the SQL query entered by the user in the GUI, and executes it using the PostgreSQL EXPLAIN

(FORMAT JSON, ANALYZE) command. This command instructs PostgreSQL to run the query, collect execution statistics, and return the query execution plan (QEP) in a structured JSON format.

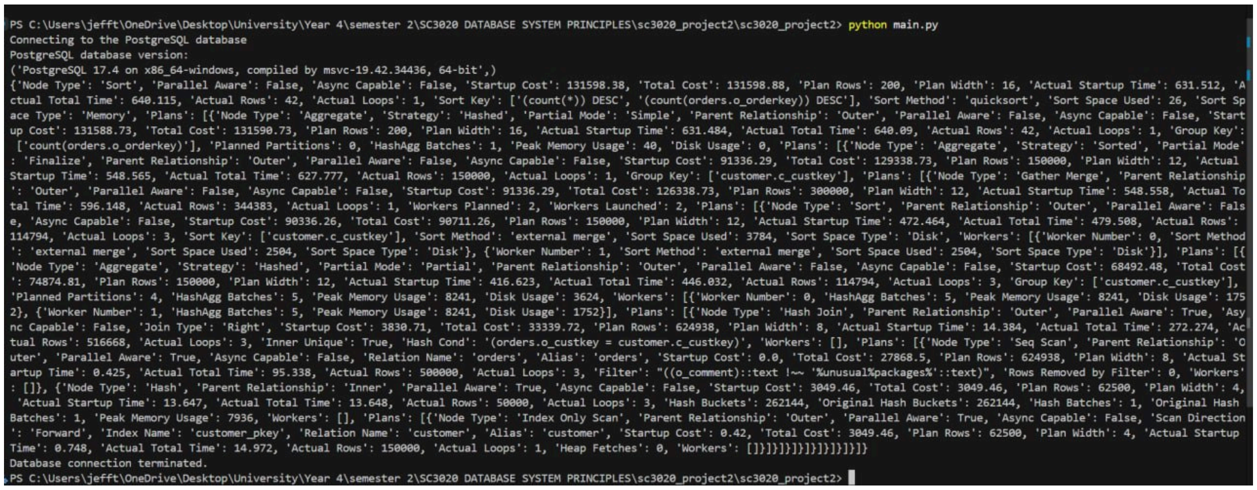
```python
input_sql_query = db_connection_configurations['query']

# Create cursor object to facilitate execution of SQL queries via connection
cursor = connection.cursor()

# Execute EXPLAIN ANALYSE on SQL query to get JSON output showing QEP and other statistics
cursor.execute(f"EXPLAIN (FORMAT JSON, ANALYZE) {input_sql_query}")
result = cursor.fetchone()[0]
```

Finally, the function closes both the cursor and database connection and returns the root node of the QEP JSON.

```python
cursor.close()
connection.close()

# Return QEP
return result[0]['Plan']
```

Section 3: Pipe Syntax

Pipe syntax was introduced by Google for BigQuery as an alternative syntax for SQL queries, which is easier to read, write, and understand. The purpose of the pipsyntax.py script is to generate the pipe syntax of the given SQL query, leveraging the QEP JSON object generated by PostgreSQL, which is shown in the figure below.

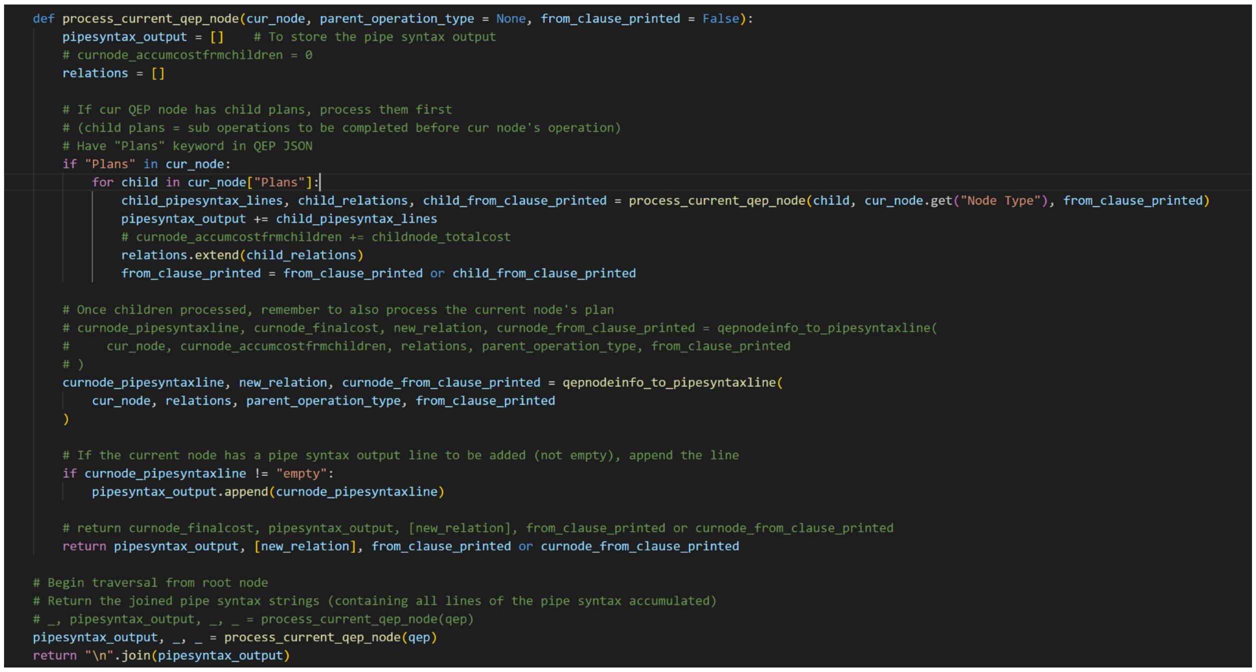


```
PS C:\Users\jefft\OneDrive\Desktop\University\Year 4\semester 2\SC3020 DATABASE SYSTEM PRINCIPLES\sc3020_project2\sc3020_project2> python main.py
Connecting to the PostgreSQL database
PostgreSQL database version:
('PostgreSQL 17.4 on x86_64-windows, compiled by msvc-19.42.34436, 64-bit',)
{'Node Type': 'Sort', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 131598.38, 'Total Cost': 131598.88, 'Plan Rows': 200, 'Plan Width': 16, 'Actual Startup Time': 631.512, 'A
ctual Total Time': 640.115, 'Actual Rows': 42, 'Actual Loops': 1, 'Sort Key': ['(count(*)) DESC', '(count(orders.o_orderkey)) DESC'], 'Sort Method': 'quicksort', 'Sort Space Used': 26, 'Sort Sp
ace Type': 'Memory', 'Plans': [{'Node Type': 'Aggregate', 'Strategy': 'Hashed', 'Partial Mode': 'Simple', 'Parent Relationship': 'Outer', 'Parallel Aware': False, 'Async Capable': False, 'Start
up Cost': 131588.73, 'Total Cost': 131590.73, 'Plan Rows': 200, 'Plan Width': 16, 'Actual Startup Time': 631.484, 'Actual Total Time': 640.09, 'Actual Rows': 42, 'Actual Loops': 1, 'Group Key':
 ['count(orders.o_orderkey)'], 'Planned Partitions': 0, 'HashAgg Batches': 1, 'Peak Memory Usage': 40, 'Disk Usage': 0, 'Plans': [{'Node Type': 'Aggregate', 'Strategy': 'Sorted', 'Partial Mode'
: 'Finalize', 'Parent Relationship': 'Outer', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 91336.29, 'Total Cost': 129338.73, 'Plan Rows': 150000, 'Plan Width': 12, 'Actual
Startup Time': 548.565, 'Actual Total Time': 627.777, 'Actual Rows': 150000, 'Actual Loops': 1, 'Group Key': ['customer.c_custkey'], 'Plans': [{'Node Type': 'Gather Merge', 'Parent Relationship
': 'Outer', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 91336.29, 'Total Cost': 126338.73, 'Plan Rows': 300000, 'Plan Width': 12, 'Actual Startup Time': 548.558, 'Actual To
tal Time': 596.148, 'Actual Rows': 344383, 'Actual Loops': 1, 'Workers Planned': 2, 'Workers Launched': 2, 'Plans': [{'Node Type': 'Sort', 'Parent Relationship': 'Outer', 'Parallel Aware': Fals
e, 'Async Capable': False, 'Startup Cost': 90336.26, 'Total Cost': 90711.26, 'Plan Rows': 150000, 'Plan Width': 12, 'Actual Startup Time': 472.464, 'Actual Total Time': 479.508, 'Actual Rows':
114794, 'Actual Loops': 3, 'Sort Key': ['customer.c_custkey'], 'Sort Method': 'external merge', 'Sort Space Used': 3784, 'Sort Space Type': 'Disk', 'Workers': [{'Worker Number': 0, 'Sort Method
': 'external merge', 'Sort Space Used': 2504, 'Sort Space Type': 'Disk'}, {'Worker Number': 1, 'Sort Method': 'external merge', 'Sort Space Used': 2504, 'Sort Space Type': 'Disk'}], 'Plans': [{
'Node Type': 'Aggregate', 'Strategy': 'Hashed', 'Partial Mode': 'Partial', 'Parent Relationship': 'Outer', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 68492.48, 'Total Cost
': 74874.81, 'Plan Rows': 150000, 'Plan Width': 12, 'Actual Startup Time': 416.623, 'Actual Total Time': 446.032, 'Actual Rows': 114794, 'Actual Loops': 3, 'Group Key': ['customer.c_custkey'],
'Planned Partitions': 4, 'HashAgg Batches': 5, 'Peak Memory Usage': 8241, 'Disk Usage': 3624, 'Workers': [{'Worker Number': 0, 'HashAgg Batches': 5, 'Peak Memory Usage': 8241, 'Disk Usage': 175
2}, {'Worker Number': 1, 'HashAgg Batches': 5, 'Peak Memory Usage': 8241, 'Disk Usage': 1752}], 'Plans': [{'Node Type': 'Hash Join', 'Parent Relationship': 'Outer', 'Parallel Aware': True, 'Asy
nc Capable': False, 'Join Type': 'Right', 'Startup Cost': 3830.71, 'Total Cost': 33339.72, 'Plan Rows': 624938, 'Plan Width': 8, 'Actual Startup Time': 14.384, 'Actual Total Time': 272.274, 'Ac
tual Rows': 516668, 'Actual Loops': 3, 'Inner Unique': True, 'Hash Cond': '(orders.o_custkey = customer.c_custkey)', 'Workers': [], 'Plans': [{'Node Type': 'Seq Scan', 'Parent Relationship': 'O
uter', 'Parallel Aware': True, 'Async Capable': False, 'Relation Name': 'orders', 'Alias': 'orders', 'Startup Cost': 0.0, 'Total Cost': 27868.5, 'Plan Rows': 624938, 'Plan Width': 8, 'Actual St
artup Time': 0.425, 'Actual Total Time': 95.338, 'Actual Rows': 500000, 'Actual Loops': 3, 'Filter': "((o_comment)::text !~~ '%unusual%packages%'::text)", 'Rows Removed by Filter': 0, 'Workers'
: []}, {'Node Type': 'Hash', 'Parent Relationship': 'Inner', 'Parallel Aware': True, 'Async Capable': False, 'Startup Cost': 3049.46, 'Total Cost': 3049.46, 'Plan Rows': 62500, 'Plan Width': 4,
 'Actual Startup Time': 13.647, 'Actual Total Time': 13.648, 'Actual Rows': 50000, 'Actual Loops': 3, 'Hash Buckets': 262144, 'Original Hash Buckets': 262144, 'Hash Batches': 1, 'Original Hash
Batches': 1, 'Peak Memory Usage': 7936, 'Workers': [], 'Plans': [{'Node Type': 'Index Only Scan', 'Parent Relationship': 'Outer', 'Parallel Aware': True, 'Async Capable': False, 'Scan Direction
': 'Forward', 'Index Name': 'customer_pkey', 'Relation Name': 'customer', 'Alias': 'customer', 'Startup Cost': 0.42, 'Total Cost': 3049.46, 'Plan Rows': 62500, 'Plan Width': 4, 'Actual Startup
Time': 0.748, 'Actual Total Time': 14.972, 'Actual Rows': 150000, 'Actual Loops': 1, 'Heap Fetches': 0, 'Workers': []}]}]}]}]}]}]}]}]}
Database connection terminated.
PS C:\Users\jefft\OneDrive\Desktop\University\Year 4\semester 2\SC3020 DATABASE SYSTEM PRINCIPLES\sc3020_project2\sc3020_project2>
```

Fig 1: QEP JSON Object

The QEP JSON object that is generated from PostgreSQL's EXPLAIN function is highly complex, featuring nested nodes that represent different plans executed by PostgreSQL during the execution of the SQL query. Thus, the pipesyntax.py script aims to take the low-level and complex QEP output and dynamically generate a readable, step-by-step representation of the operations of the query plan and data flow in the form of pipe syntax. To help us understand the various nodes and operations, we referred to the following website: https://pganalyze.com/.

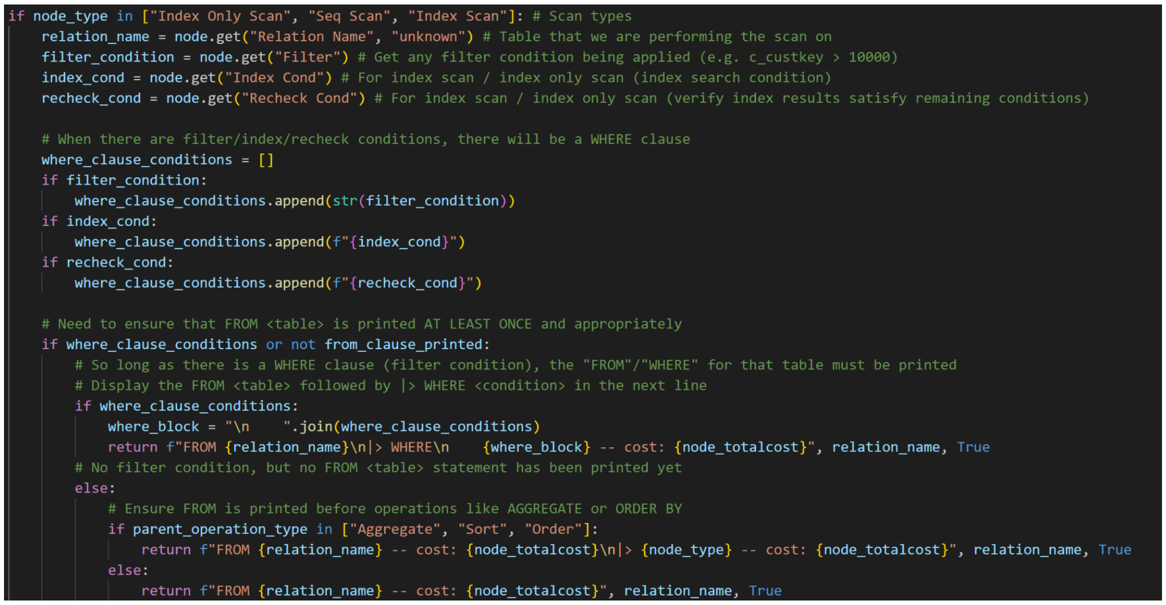
The main function of pipesyntax.py is generate_pipe_syntax(qep), which takes the QEP JSON as the input and traverses the QEP tree structure recursively with the help of the process_current_qep_node() helper function. For each node that the function parses, it extracts data from the node including fields such as the cost of the operation and operation type, appending it recursively to the parent node as it processes children nodes in a bottom-up execution. Finally, it returns the final pipe syntax as a joined string.



```
def process_current_qep_node(cur_node, parent_operation_type = None, from_clause_printed = False):
    pipesyntax_output = []    # To store the pipe syntax output
    # curnode_accumcostfrmchildren = 0
    relations = []

    # If cur QEP node has child plans, process them first
    # (child plans = sub operations to be completed before cur node's operation)
    # Have "Plans" keyword in QEP JSON
    if "Plans" in cur_node:
        for child in cur_node["Plans"]:
            child_pipesyntax_lines, child_relations, child_from_clause_printed = process_current_qep_node(child, cur_node.get("Node Type"), from_clause_printed)
            pipesyntax_output += child_pipesyntax_lines
            # curnode_accumcostfrmchildren += childnode_totalcost
            relations.extend(child_relations)
            from_clause_printed = from_clause_printed or child_from_clause_printed

    # Once children processed, remember to also process the current node's plan
    # curnode_pipesyntaxline, curnode_finalcost, new_relation, curnode_from_clause_printed = qepnodeinfo_to_pipesyntaxline(
    #     cur_node, curnode_accumcostfrmchildren, relations, parent_operation_type, from_clause_printed
    # )
    curnode_pipesyntaxline, new_relation, curnode_from_clause_printed = qepnodeinfo_to_pipesyntaxline(
        cur_node, relations, parent_operation_type, from_clause_printed
    )

    # If the current node has a pipe syntax output line to be added (not empty), append the line
    if curnode_pipesyntaxline != "empty":
        pipesyntax_output.append(curnode_pipesyntaxline)

    # return curnode_finalcost, pipesyntax_output, [new_relation], from_clause_printed or curnode_from_clause_printed
    return pipesyntax_output, [new_relation], from_clause_printed or curnode_from_clause_printed

# Begin traversal from root node
# Return the joined pipe syntax strings (containing all lines of the pipe syntax accumulated)
# _, pipesyntax_output, _, _ = process_current_qep_node(qep)
pipesyntax_output, _, _ = process_current_qep_node(qep)
return "\n".join(pipesyntax_output)
```

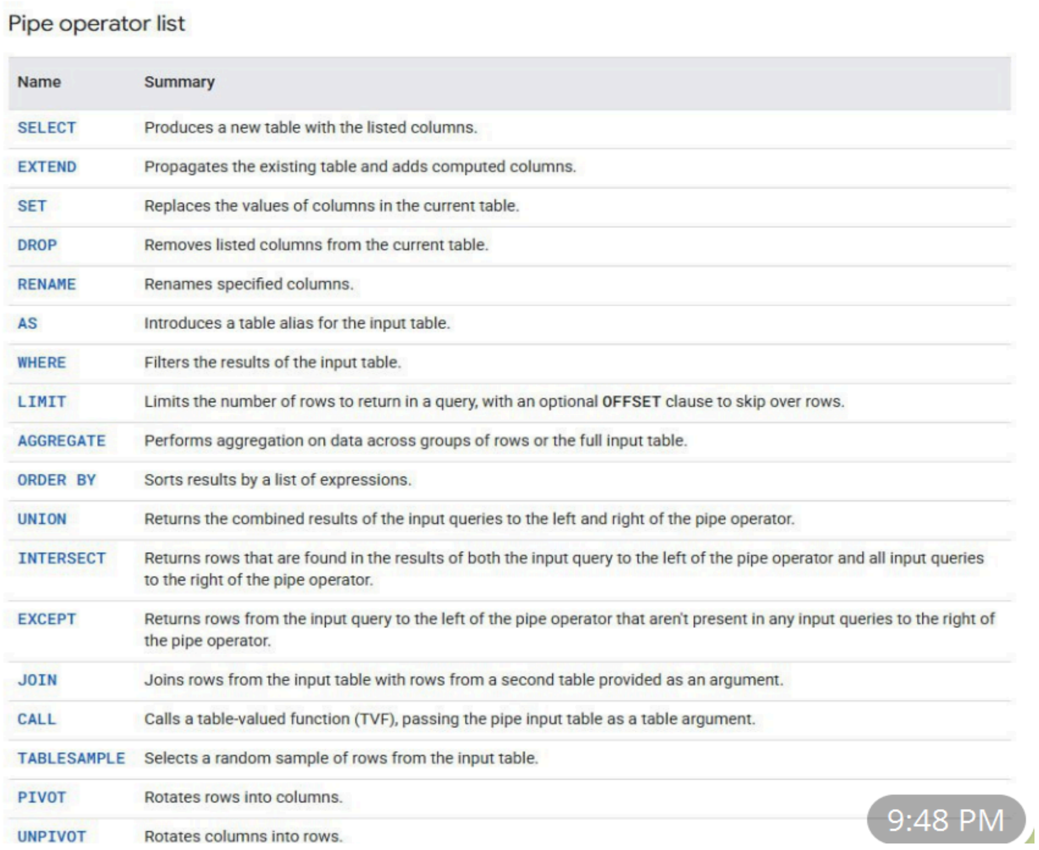
During this generation of the pipe syntax, the qepnodeinfo_to_pipesyntaxline() function helps to translate the node types into pipe syntax operations. This is because there are many different QEP JSON operation node types that map to different pipe syntax operators that need to be processed differently.

For a start, the pipe syntax usually starts with the standard FROM clause. Hence we start by searching for a node-type that is either "Seq Scan", "Index Scan", or "Index Only Scan". These scans are the most common base table access operations in a QEP. Once a scan node is detected, the table name, filter or index conditions are extracted and appended below the FROM clause using a WHERE block. We also print the FROM <table> clause if there are no conditions, if a FROM clause has not been printed or the parent operation is AGGREGATE, SORT or ORDER, which would be invalid without a FROM clause.



```python
if node_type in ["Index Only Scan", "Seq Scan", "Index Scan"]: # Scan types
    relation_name = node.get("Relation Name", "unknown") # Table that we are performing the scan on
    filter_condition = node.get("Filter") # Get any filter condition being applied (e.g. c_custkey > 10000)
    index_cond = node.get("Index Cond") # For index scan / index only scan (index search condition)
    recheck_cond = node.get("Recheck Cond") # For index scan / index only scan (verify index results satisfy remaining conditions)

    # When there are filter/index/recheck conditions, there will be a WHERE clause
    where_clause_conditions = []
    if filter_condition:
        where_clause_conditions.append(str(filter_condition))
    if index_cond:
        where_clause_conditions.append(f"{index_cond}")
    if recheck_cond:
        where_clause_conditions.append(f"{recheck_cond}")

    # Need to ensure that FROM <table> is printed AT LEAST ONCE and appropriately
    if where_clause_conditions or not from_clause_printed:
        # So long as there is a WHERE clause (filter condition), the "FROM"/"WHERE" for that table must be printed
        # Display the FROM <table> followed by |> WHERE <condition> in the next line
        if where_clause_conditions:
            where_block = "\n    ".join(where_clause_conditions)
            return f"FROM {relation_name}\n|> WHERE\n    {where_block} -- cost: {node_totalcost}", relation_name, True
        # No filter condition, but no FROM <table> statement has been printed yet
        else:
            # Ensure FROM is printed before operations like AGGREGATE or ORDER BY
            if parent_operation_type in ["Aggregate", "Sort", "Order"]:
                return f"FROM {relation_name} -- cost: {node_totalcost}\n|> {node_type} -- cost: {node_totalcost}", relation_name, True
            else:
                return f"FROM {relation_name} -- cost: {node_totalcost}", relation_name, True
```

Following the FROM clause in the pipe syntax, various pipe operators are used to represent subsequent operations in the query. These operators are chained together using the pipe symbol (|>) to describe the data logical flow. The complete list of available pipe operators is provided below:



Pipe operator list

Name	Summary
SELECT	Produces a new table with the listed columns.
EXTEND	Propagates the existing table and adds computed columns.
SET	Replaces the values of columns in the current table.
DROP	Removes listed columns from the current table.
RENAME	Renames specified columns.
AS	Introduces a table alias for the input table.
WHERE	Filters the results of the input table.
LIMIT	Limits the number of rows to return in a query, with an optional **OFFSET** clause to skip over rows.
AGGREGATE	Performs aggregation on data across groups of rows or the full input table.
ORDER BY	Sorts results by a list of expressions.
UNION	Returns the combined results of the input queries to the left and right of the pipe operator.
INTERSECT	Returns rows that are found in the results of both the input query to the left of the pipe operator and all input queries to the right of the pipe operator.
EXCEPT	Returns rows from the input query to the left of the pipe operator that aren't present in any input queries to the right of the pipe operator.
JOIN	Joins rows from the input table with rows from a second table provided as an argument.
CALL	Calls a table-valued function (TVF), passing the pipe input table as a table argument.
TABLESAMPLE	Selects a random sample of rows from the input table.
PIVOT	Rotates rows into columns.
UNPIVOT	Rotates columns into rows.

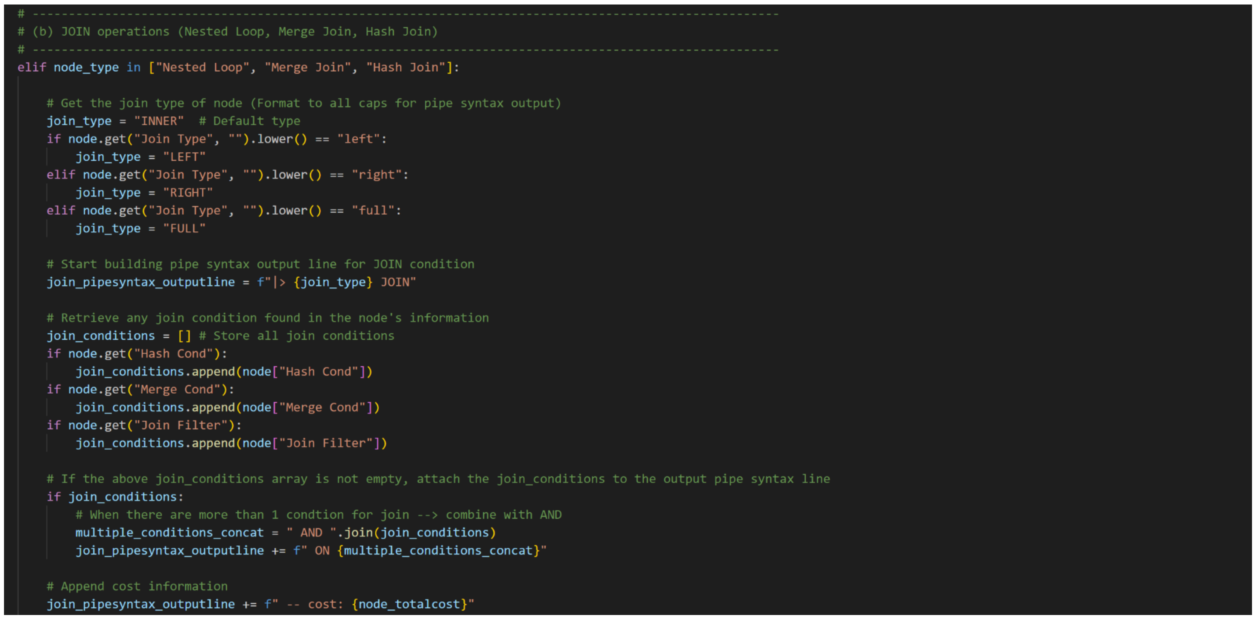
9:48 PM

Fig 2: Pipe Syntax Operators (*Work With Pipe Query Syntax*, n.d.)

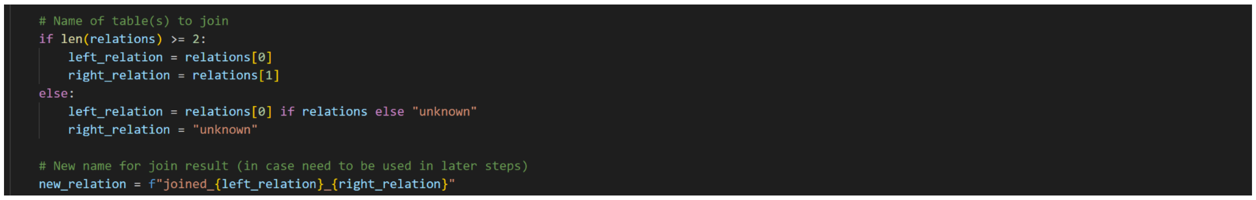
Join Operations

The JOIN operation joins rows from the input table with rows from a second table provided as an argument. The function searches for nodes that are of type “Nested Loop”, “Merge Join” or “Hash Join” to identify JOIN operations in the QEP. Once a join node is detected, the function first defaults it to an INNER JOIN type, but adjusts it to LEFT, RIGHT or FULL depending on the value specified under the “Join Type” field in the node.

The function also looks for any join conditions that may be attached to the node specifically under the “Hash Cond”, “Merge Cond” or “Join Filter” and aggregates them into a single ON clause. If multiple conditions exist, they are joined using the logical AND operator to form a combined condition string. The function also takes in the cost information and attempts to name the left and right tables involved in the join.



```python
# ----------------------------------------------------------------------------------------------------
# (b) JOIN operations (Nested Loop, Merge Join, Hash Join)
# ----------------------------------------------------------------------------------------------------
elif node_type in ["Nested Loop", "Merge Join", "Hash Join"]:

    # Get the join type of node (Format to all caps for pipe syntax output)
    join_type = "INNER"  # Default type
    if node.get("Join Type", "").lower() == "left":
        join_type = "LEFT"
    elif node.get("Join Type", "").lower() == "right":
        join_type = "RIGHT"
    elif node.get("Join Type", "").lower() == "full":
        join_type = "FULL"

    # Start building pipe syntax output line for JOIN condition
    join_pipesyntax_outputline = f"|> {join_type} JOIN"

    # Retrieve any join condition found in the node's information
    join_conditions = [] # Store all join conditions
    if node.get("Hash Cond"):
        join_conditions.append(node["Hash Cond"])
    if node.get("Merge Cond"):
        join_conditions.append(node["Merge Cond"])
    if node.get("Join Filter"):
        join_conditions.append(node["Join Filter"])

    # If the above join_conditions array is not empty, attach the join_conditions to the output pipe syntax line
    if join_conditions:
        # When there are more than 1 condtion for join --> combine with AND
        multiple_conditions_concat = " AND ".join(join_conditions)
        join_pipesyntax_outputline += f" ON {multiple_conditions_concat}"

    # Append cost information
    join_pipesyntax_outputline += f" -- cost: {node_totalcost}"
```

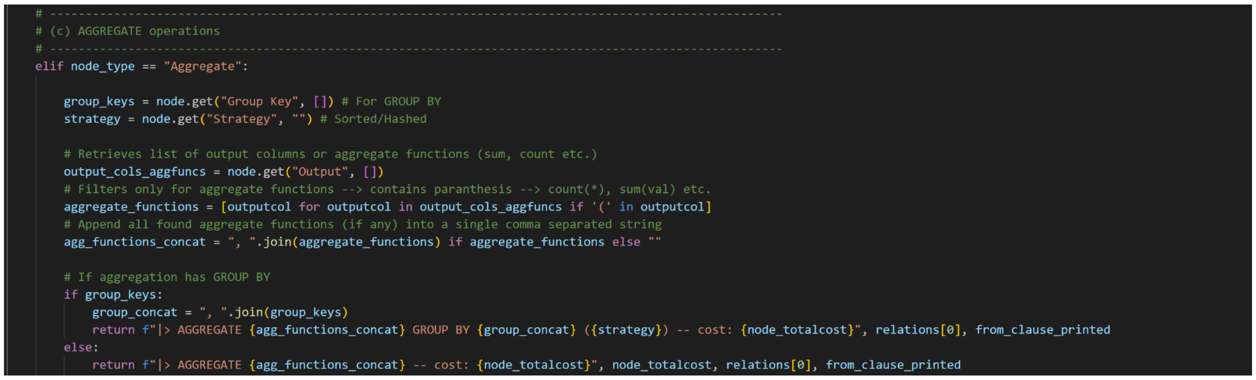


```python
    # Name of table(s) to join
    if len(relations) >= 2:
        left_relation = relations[0]
        right_relation = relations[1]
    else:
        left_relation = relations[0] if relations else "unknown"
        right_relation = "unknown"

    # New name for join result (in case need to be used in later steps)
    new_relation = f"joined_{left_relation}_{right_relation}"
```

Aggregate Operations

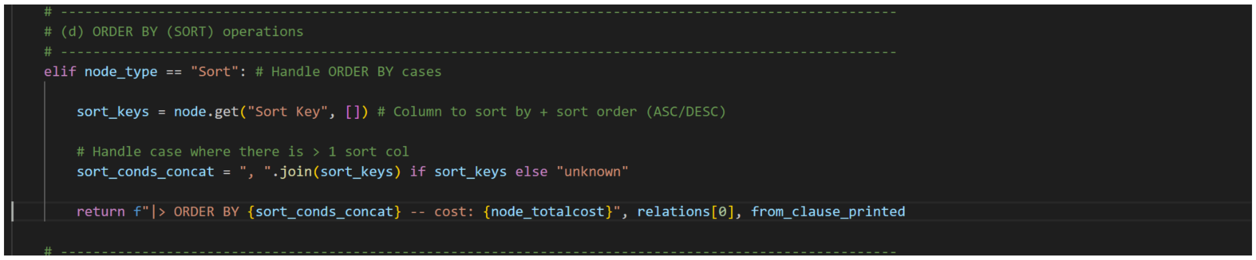
The AGGREGATE operation handles the grouping and summarisation of data. The function identifies aggregation by searching for a node of type “aggregate”. Once found, it extracts the Group Key which corresponds to the GROUP BY clause in SQL. The function also extracts the Strategy which corresponds to the method used for aggregation - either sorted or hashed.

Next, the function also looks for the Output field of the node to extract the list of output columns. It does so to identify the actual aggregate functions, distinguished by the parentheses such as count (*) or sum(val). The functions are then concatenated into a single string separated by comma. If the node has a Group Key, the function formats the line as |> AGGREGATE <functions> GROUP BY <keys> (<strategy>) together with its cost.



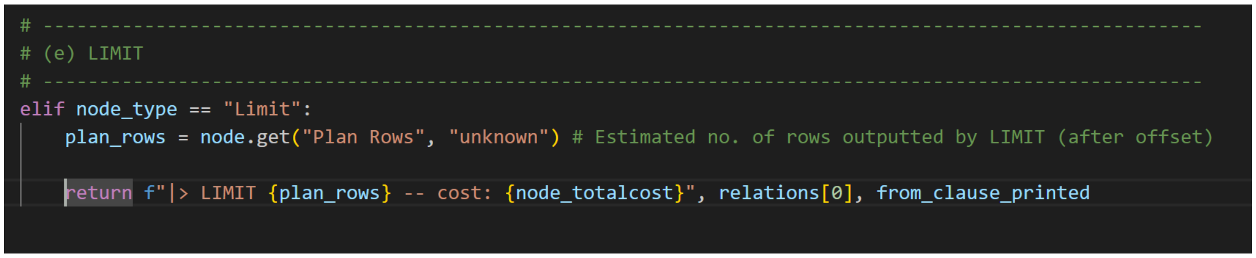
```python
# ----------------------------------------------------------------------------------------------------
# (c) AGGREGATE operations
# ----------------------------------------------------------------------------------------------------
elif node_type == "Aggregate":

    group_keys = node.get("Group Key", []) # For GROUP BY
    strategy = node.get("Strategy", "") # Sorted/Hashed

    # Retrieves list of output columns or aggregate functions (sum, count etc.)
    output_cols_aggfuncs = node.get("Output", [])
    # Filters only for aggregate functions --> contains paranthesis --> count(*), sum(val) etc.
    aggregate_functions = [outputcol for outputcol in output_cols_aggfuncs if '(' in outputcol]
    # Append all found aggregate functions (if any) into a single comma separated string
    agg_functions_concat = ", ".join(aggregate_functions) if aggregate_functions else ""

    # If aggregation has GROUP BY
    if group_keys:
        group_concat = ", ".join(group_keys)
        return f"|> AGGREGATE {agg_functions_concat} GROUP BY {group_concat} ({strategy}) -- cost: {node_totalcost}", relations[0], from_clause_printed
    else:
        return f"|> AGGREGATE {agg_functions_concat} -- cost: {node_totalcost}", node_totalcost, relations[0], from_clause_printed
```

Order By Operations
The ORDER BY operation handles the sorting of data based on one or more columns as specified in the QEP. It is indicated by nodes of node type “Sort”. After retrieving the Sort Key, which may also include one or more columns along with their sorting order either in Ascending or Descending order. These sort conditions are then concatenated into a comma-separated string to represent the SORT clause in the pipe syntax format. The final output line is then constructed as |> ORDER BY <columns>.



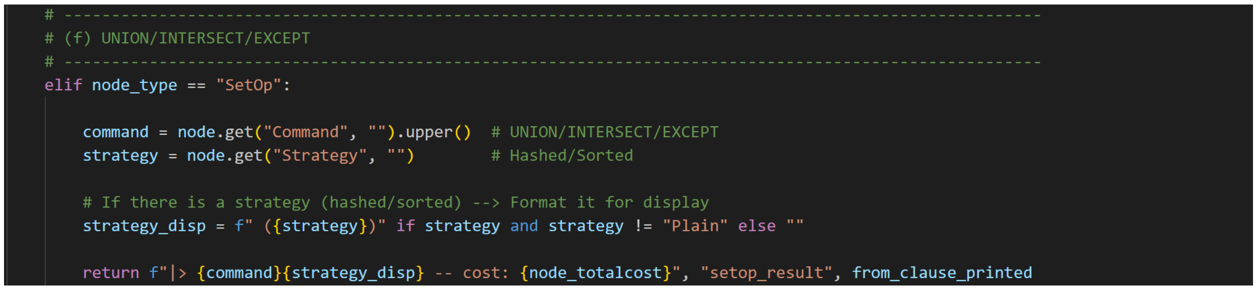
```python
# ----------------------------------------------------------------------------------------------------
# (d) ORDER BY (SORT) operations
# ----------------------------------------------------------------------------------------------------
elif node_type == "Sort": # Handle ORDER BY cases

    sort_keys = node.get("Sort Key", []) # Column to sort by + sort order (ASC/DESC)

    # Handle case where there is > 1 sort col
    sort_conds_concat = ", ".join(sort_keys) if sort_keys else "unknown"

    return f"|> ORDER BY {sort_conds_concat} -- cost: {node_totalcost}", relations[0], from_clause_printed

# ----------------------------------------------------------------------------------------------------
```

Limit Operations
The LIMIT operation restricts the number of rows returned from the result set. When the function encounters a node of type “Limit”, the function extracts the “Plan Rows” values which is the estimated number of rows after the limit is applied. The function then constructs the output line as |> LIMIT <rows>.



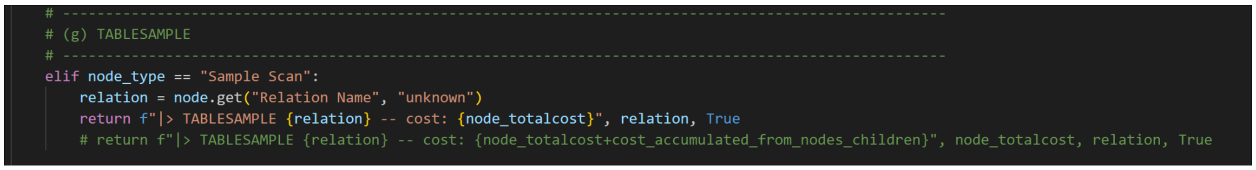
```python
# ----------------------------------------------------------------------------------------------------
# (e) LIMIT
# ----------------------------------------------------------------------------------------------------
elif node_type == "Limit":
    plan_rows = node.get("Plan Rows", "unknown") # Estimated no. of rows outputted by LIMIT (after offset)

    return f"|> LIMIT {plan_rows} -- cost: {node_totalcost}", relations[0], from_clause_printed
```

Union/Intersect/Except Operations
UNION, INTERSECT and EXCEPT are set operations that can be captured using nodes of type “SetOp”. The function retrieves the type of set operation from the node’s Command field and checks for the strategy, to check if the operation is carried out using hashed or sorted method. The final pipe syntax is generated in the form |> <COMMAND> (<Strategy>).



```
# ------------------------------------------------------------------------------------------------------
# (f) UNION/INTERSECT/EXCEPT
# ------------------------------------------------------------------------------------------------------
elif node_type == "SetOp":

    command = node.get("Command", "").upper()  # UNION/INTERSECT/EXCEPT
    strategy = node.get("Strategy", "")        # Hashed/Sorted

    # If there is a strategy (hashed/sorted) --> Format it for display
    strategy_disp = f" ({strategy})" if strategy and strategy != "Plain" else ""

    return f"|> {command}{strategy_disp} -- cost: {node_totalcost}", "setop_result", from_clause_printed
```

Table Sample Operations
The TABLESAMPLE operation is used to return a random sample of rows from a table, essential for approximating query processing. The function searches for a “Sample Scan” node and retrieves its relation name. It then outputs the pipe syntax as |> TABLESAMPLE <table>.

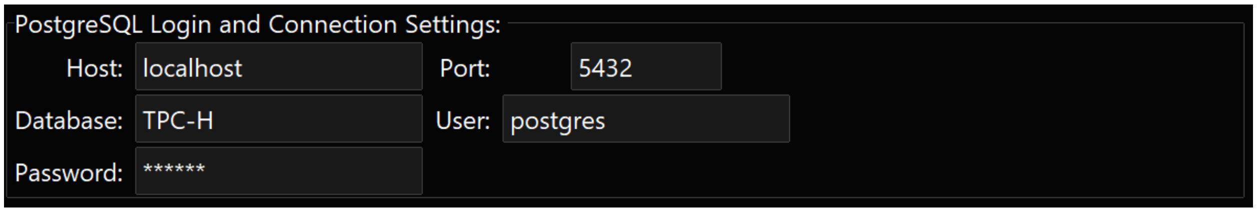


```
# ----------------------------------------------------------------------------------------------------
# (g) TABLESAMPLE
# ----------------------------------------------------------------------------------------------------
elif node_type == "Sample Scan":
    relation = node.get("Relation Name", "unknown")
    return f"|> TABLESAMPLE {relation} -- cost: {node_totalcost}", relation, True
    # return f"|> TABLESAMPLE {relation} -- cost: {node_totalcost+cost_accumulated_from_nodes_children}", node_totalcost, relation, True
```

Section 4: Interface
For the user interface, we use a ttkbootstrap library, which allows developers to easily create user-friendly GUI applications with predefined styles, themes, and even custom widgets (Ttkbootstrap - Ttkbootstrap, n.d.). In our GUI setup, we use the “cyborg” theme and title the GUI as “SC3020 Group 2 SQL to Pipe-Syntax Converter”. The GUI is separated into 3 parts:

i. PostgreSQL Login and Connection Settings
ii. Input SQL Query to convert
iii. Output - Pipe Syntax Query, QEP Text and QEP Visual Graph

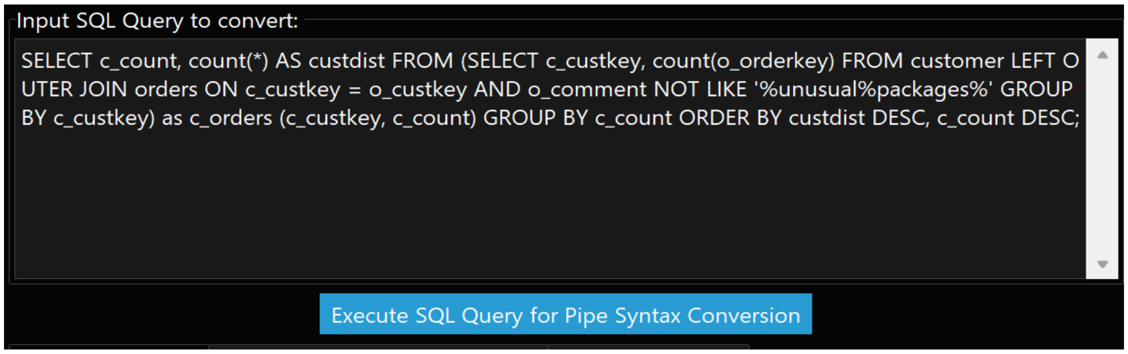
The first part, the Database Connection Settings panel, takes in the host name, port number, database name, username and password, essential for the connection to any PostgreSQL schema as discussed in section 2.



PostgreSQL Login and Connection Settings:

Host:	localhost	Port:	5432
Database:	TPC-H	User:	postgres
Password:	******		

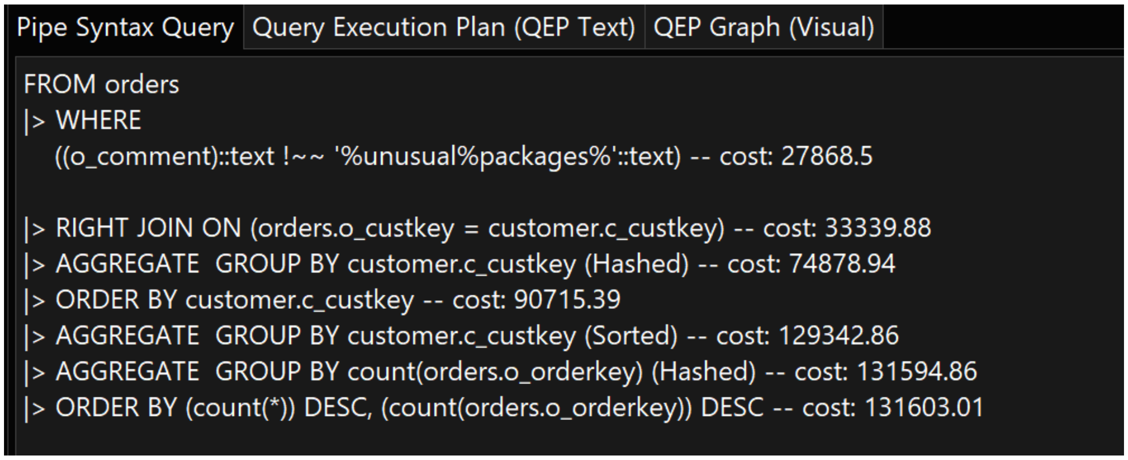
Below the first panel is the SQL Query panel for users to input their SQL query, this will be the SQL query input that we will put into our programme to generate the pipe syntax. Upon entering their SQL query, the user will have to select the “Execute SQL Query for Pipe Syntax Conversion” button to initiate the execute_run_sql_generate_pipesyntax function.


Input SQL Query to convert:
Execute SQL Query for Pipe Syntax Conversion

```
SELECT c_count, count(*) AS custdist FROM (SELECT c_custkey, count(o_orderkey) FROM customer LEFT O
UTER JOIN orders ON c_custkey = o_custkey AND o_comment NOT LIKE '%unusual%packages%' GROUP
BY c_custkey) as c_orders (c_custkey, c_count) GROUP BY c_count ORDER BY custdist DESC, c_count DESC;
```

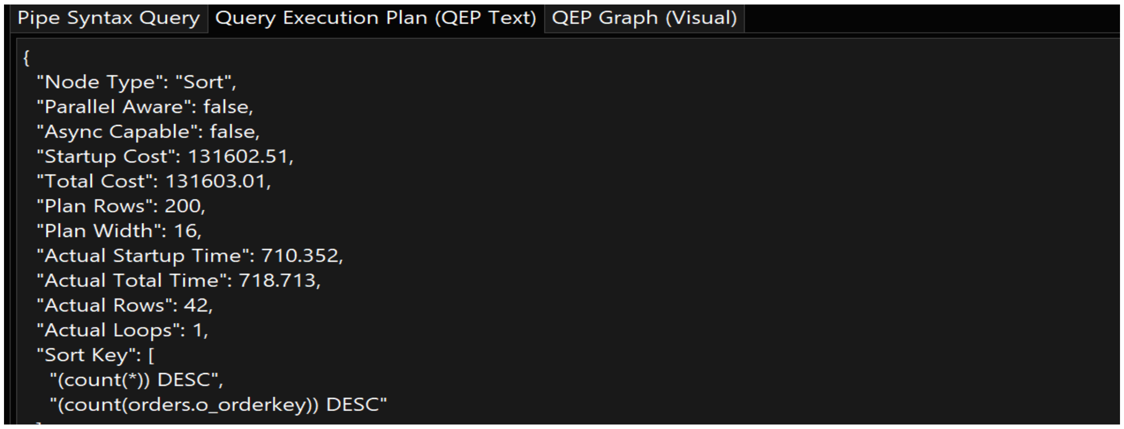
Finally the last panel consists of 3 tabs - Pipe Syntax Query, QEP Text and QEP Graph tab. These three tabs show the 3 different outputs of our programme.

The Pipe Syntax Query tab displays the pipe syntax query of the given SQL Query, this is our results panel which displays the final pipe syntax output generated.



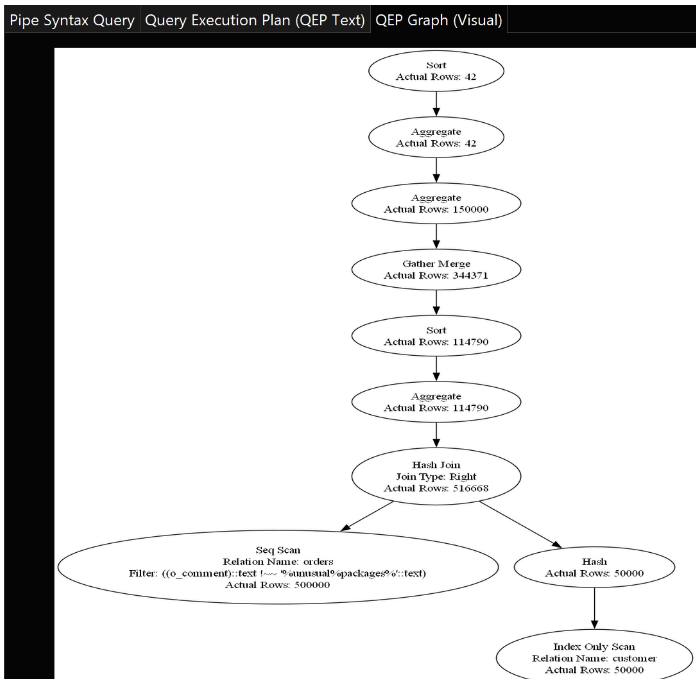
```
FROM orders
|> WHERE
   ((o_comment)::text !~~ '%unusual%packages%'::text) -- cost: 27868.5

|> RIGHT JOIN ON (orders.o_custkey = customer.c_custkey) -- cost: 33339.88
|> AGGREGATE  GROUP BY customer.c_custkey (Hashed) -- cost: 74878.94
|> ORDER BY customer.c_custkey -- cost: 90715.39
|> AGGREGATE  GROUP BY customer.c_custkey (Sorted) -- cost: 129342.86
|> AGGREGATE  GROUP BY count(orders.o_orderkey) (Hashed) -- cost: 131594.86
|> ORDER BY (count(*)) DESC, (count(orders.o_orderkey)) DESC -- cost: 131603.01
```

The Query Execution Plan (QEP Text) tab displays the plain QEP JSON object produced from the PostgreSQL EXPLAIN (FORMAT JSON, ANALYZE) command.



```
{
  "Node Type": "Sort",
  "Parallel Aware": false,
  "Async Capable": false,
  "Startup Cost": 131602.51,
  "Total Cost": 131603.01,
  "Plan Rows": 200,
  "Plan Width": 16,
  "Actual Startup Time": 710.352,
  "Actual Total Time": 718.713,
  "Actual Rows": 42,
  "Actual Loops": 1,
  "Sort Key": [
    "(count(*)) DESC",
    "(count(orders.o_orderkey)) DESC"
```

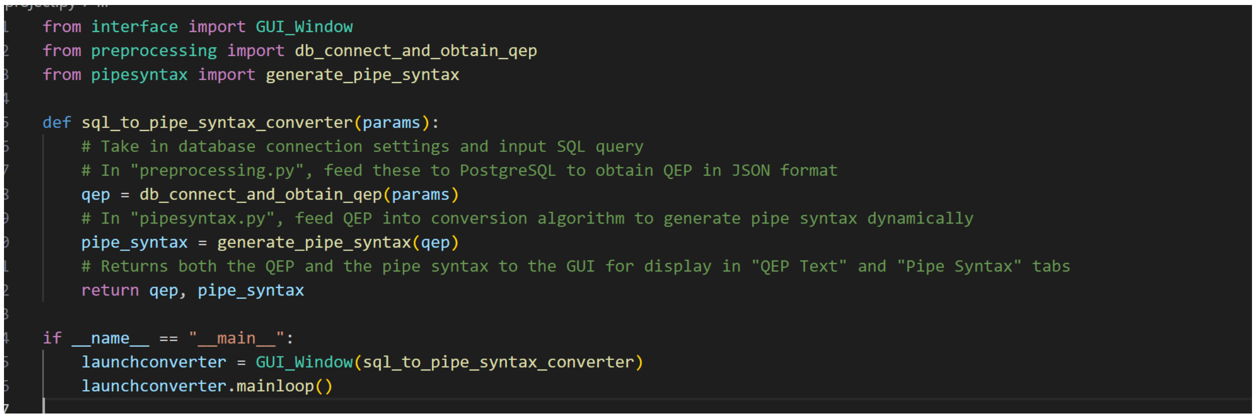
The last tab shows the QEP Graph that we have plotted based on the structure of the QEP JSON object, this helps us to visualise the QEP and its sequential order of operations.



Section 5: Project

The sql_to_pipe_syntax_converter function serves as the main integration point for converting an SQL query into its equivalent pipe syntax. It takes in the parameters needed for database connection as mentioned in Section 2 followed by taking the SQL query input and feeding it to PostgreSQL to obtain the QEP in JSON format. From the JSON file, it generates the pipe syntax as explained in Section 3. The QEP text and Pipe Syntax are then returned in the GUI as mentioned in Section 4.

The __main__ block initializes the Graphical Interface by instantiating the GUI Window with the converter function and launching the GUI main event using the mainloop(), enabling users to visually interpret SQL execution flows in an intuitive pipe syntax based format.



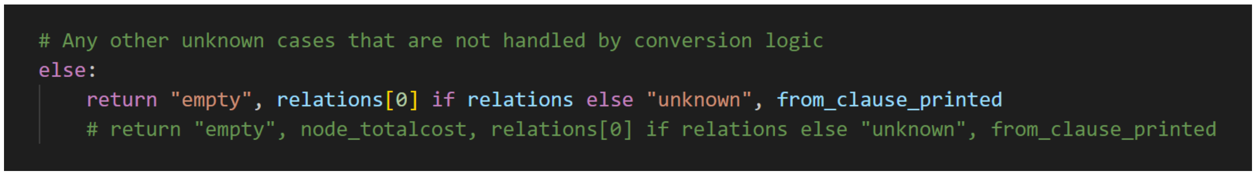
```python
from interface import GUI_Window
from preprocessing import db_connect_and_obtain_qep
from pipesyntax import generate_pipe_syntax

def sql_to_pipe_syntax_converter(params):
    # Take in database connection settings and input SQL query
    # In "preprocessing.py", feed these to PostgreSQL to obtain QEP in JSON format
    qep = db_connect_and_obtain_qep(params)
    # In "pipesyntax.py", feed QEP into conversion algorithm to generate pipe syntax dynamically
    pipe_syntax = generate_pipe_syntax(qep)
    # Returns both the QEP and the pipe syntax to the GUI for display in "QEP Text" and "Pipe Syntax" tabs
    return qep, pipe_syntax

if __name__ == "__main__":
    launchconverter = GUI_Window(sql_to_pipe_syntax_converter)
    launchconverter.mainloop()
```

Section 6: Limitations

Based on the list of pipe operators mentioned in figure 1, the program only works for the following operators:

i. WHERE
ii. LIMIT
iii. AGGREGATE
iv. ORDER BY
v. UNION
vi. INTERSECT
vii. EXCEPT
viii. JOIN
ix. TABLESAMPLE

Any other operators will not work as we only choose the more commonly used operators. At the same time, for table scanning, our program only works for index only scan, sequential scan or index scan. It is also important to note that only nested loop, merge join and hash join are the only three recognizable join operations our program recognises. Should there be a node that does not match any of the known types mentioned above, the function defaults to a generic output by returning “empty” as the pipe syntax along with the first available relation name or “unknown” if no relations are present.



```python
# Any other unknown cases that are not handled by conversion logic
else:
    return "empty", relations[0] if relations else "unknown", from_clause_printed
    # return "empty", node_totalcost, relations[0] if relations else "unknown", from_clause_printed
```

Section 7: Reference

Laithy, M. E. (2024, October 25). How Pipe Syntax Fix SQL Design Problems. DEV Community. https://dev.to/mohamed_el_laithy/how-pipe-syntax-fix-significant-design-problems-in-sql-47g8

ttkbootstrap - ttkbootstrap. (n.d.). https://ttkbootstrap.readthedocs.io/en/latest/

Work with pipe query syntax. (n.d.). Google Cloud.

https://cloud.google.com/bigquery/docs/pipe-syntax-guide

Section 8: Requirements and Instructions

1. Download Graphviz
    a. Go to graphviz.org/download/
    b. For windows, choose graphviz-12.2.1 (64 bit) EXE installer [sha256]
    c. In the Graphviz setup, select “next”, “I agree” and “Add Graphviz to the system PATH for **all users**” .
    d. Choose the install location and start menu folder.
2. Unzip the file
3. Ensure that the computer consists of the requirements as stated in requirements.txt OR run pip install -r requirements.txt
4. Run project.py file