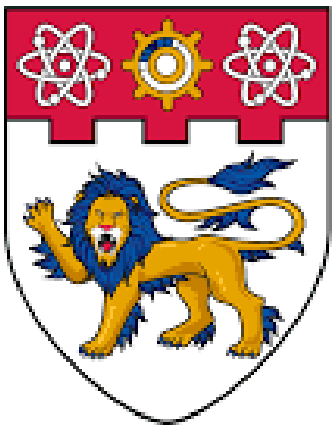


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC3020 DATABASE SYSTEM PRINCIPLES

PROJECT 2

AGARWAL LAKSHYA	U2123904L
ARORA ANUSHREE	U2123922C
BALAJI VARSHA	U2123383L
JINDAL JAI	U2123034K

LLM Declaration:
Chat-GPT was used to assist with understanding certain library specific functionalities

Table of Contents

1. Introduction

The project was made to be an academic tool for understanding query execution plans generated by postgreSQL as part of requirements for the NTU, SCSE module SC3020 Database System Principles.

1.1 Project Methodology

The project uses a backend and frontend developed in Flask and React respectively. We have deviated from the original requirement of using a python visualisation tool such as Tkinter or Streamlit as we felt that it was restrictive and a hindrance to scalability.

We have made use of Bootstrap and some react libraries for visualisation functionalities and aesthetics such as Echarts for rendering a tree structure.

As the project was made as an academic project and is not to be deployed at scale in the near future, we have been non-restrictive in security such as creating an authDetails.json file containing user details for easy access for the backend and have resorted to hardcoding the links and ports for applications.

1.2 TPC-H Dataset

The TPC-H dataset added to PostgreSQL was given to us to be used for developing and testing the project.

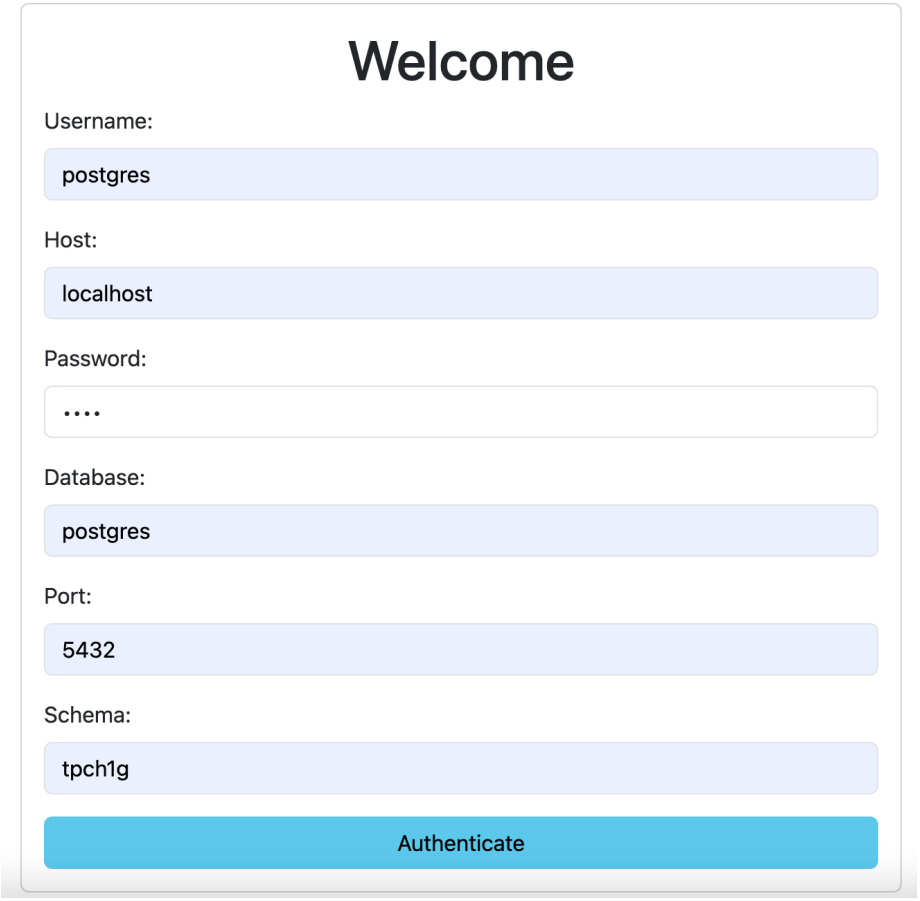
Below is a summary of the relations found in the generated PostgreSQL database:

TABLE	INDEXES	DESCRIPTION	# OF TUPLES
CUSTOMER	C_custkey, c_nationkey	Contains the list of all customers and their details	150,000
LINEITEM	(l_orderkey, l_partkey, l_suppkey, l_linenumber), l_orderkey	Contains the list of all transport lines and their details	6,001,215
NATION	N_nationkey, n_regionkey	Contains the list of available nations	25
ORDERS	O_orderkey, o_custkey	Contains the list of all orders and their details	1,500,000
PART	p_partkey	Contains the list of all	200,000

		parts and their details	
PARTSUPP	(ps_partkey, ps_suppkey), ps_partkey	Contains the list of all the suppliers of different parts	800,000
REGION	r_regionkey	Contains the list of all the region	5
SUPPLIER	S_suppkey, s_nationkey	Contain the list of all the suppliers and their details	10,000
			Total: 8661245

1.3 Setup and Authentication

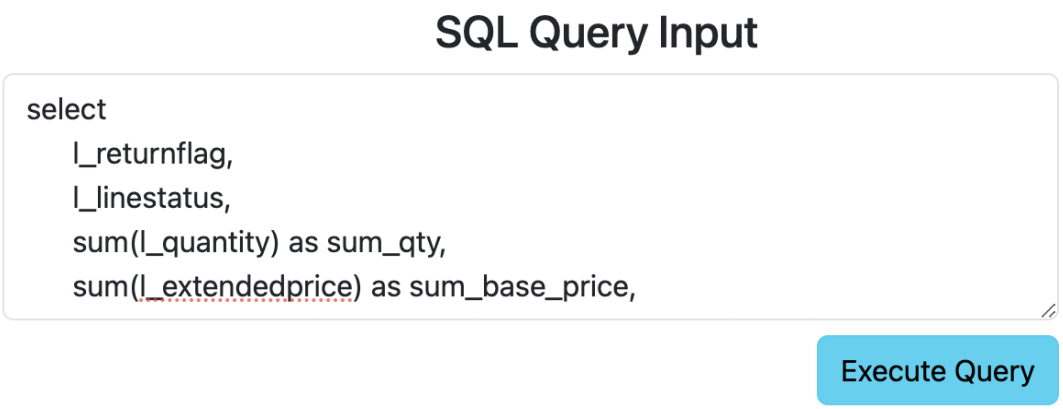
The following is the authentication page where you can enter your database authentication details and where it is hosted. All fields have been set to required.



The input also takes a schema input which was provided to us as TPCH1G in the project description. We provide this beforehand so the front end can get some values during authentication which can speed up its working later.

2. Interface Components

2.1 SQL Query Input



SQL Query**:** The primary input for the system is an SQL query provided by the user. This query serves as the foundation for the subsequent analysis of both disk block access and the Query Execution Plan (QEP) done by the PostgreSQL server communicated using a flask python.

We append the explain and analyse commands with the SQL query in the query tool which returns us a json for further visualisation.

Upon receiving the analysis JSON, the frontend restructures the data to make it ingestible by the Echarts library. Additionally the JSON also contains the information about the blocks accessed which are visualised as described in the following sections.

2.2 Query Execution Plan Tree

Overview
As mentioned earlier, this component of the interface leverages an external Javascript library, Apache Echarts. It is a powerful and interactive charting library with its own predefined charts. For our purpose we are making use of the Series-Tree built-in chart type. The tree is displayed in a scrollable component to accommodate for trees with larger depths due to complex queries. The leaf nodes are coloured to be a darker blue, for readability. These are the nodes where the bulk of the disk I/Os are happening. The internal nodes are light blue and are using the data loaded from the disk into the memory to do some operation such as merge, hash, aggregate etc. Each node is labelled with its respective Node Type returned by the Explain JSON.

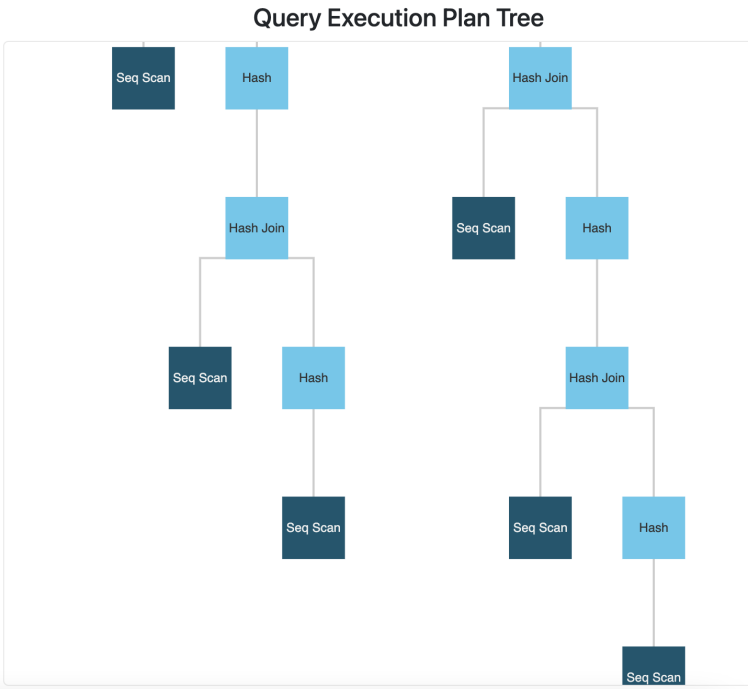


Image: Sample Tree with leaf nodes in dark blue and internal nodes in light blue

Hovering over a node

When hovering over a particular node, some basic information is displayed depending on the node type:

1. If the node accesses the disk(essentially a leaf node), it displays the table name and the number of blocks accessed.

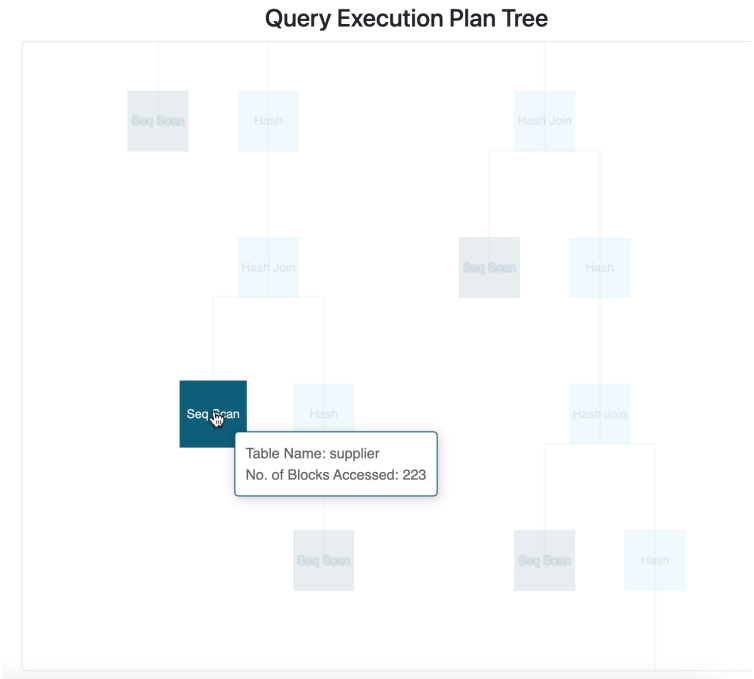


Image: Hovering over a leaf node

2. All other nodes(the internal nodes) display the startup cost and the total cost.

Image: Hovering over an internal node

Clicking a Node

Clicking on a node triggers a pop up with more detailed node-specific information, offering a comprehensive view of the selected query operation. The pop-up utilises a tab-based navigation layout to structure the information in a user-friendly manner. All nodes, regardless of node type, display a *General Statistics* tab. Nodes that access the disk(e.g. scans) or utilise the blocks for operations(e.g. joins and aggregates) display an additional *Blocks Accessed* tab to display the individual blocks from a particular table or multiple tables with the number of tuples utilised from the block. This feature allows the users to visually see the flow of data up the tree. Operations such as gather, gather merge, sort, memoize, etc do not display this tab as they utilise the same blocks from their child nodes.

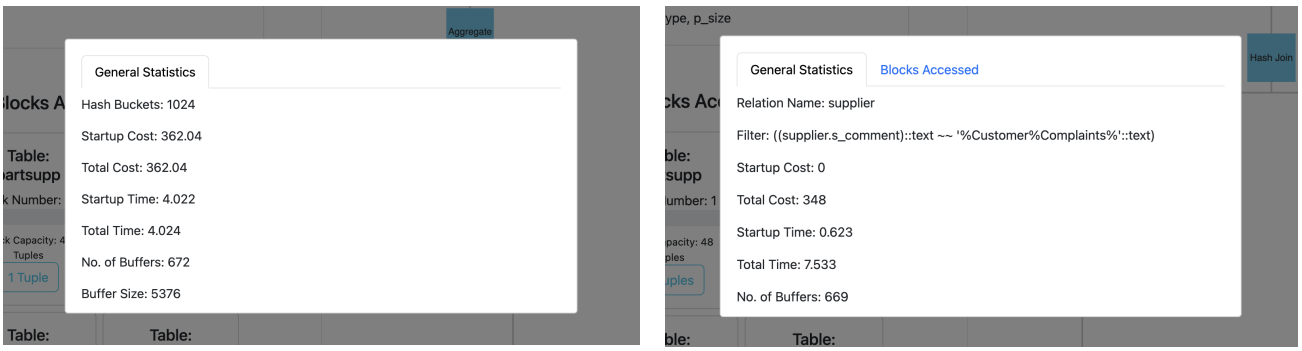


Image: Clicking on a node with that doesn't access blocks vs. clicking on a node that does

General Statistics Tab

All nodes display the general information Startup Cost, Total Cost, Startup Time, Total Time, No. of Buffers, Buffer Size and Actual Rows.

Additionally, depending on the type of node more statistics are displayed:

1. Scans - Display the Relation Name and the scan Filter if present. Index scans also display the Index Name and the Index Condition.

Image: Seq scan and Index Scan

2. Joins - Hash joins and merge joins display the join type along with hash and merge conditions.

Image: Hash join and Merge Join

3. Sort - Display the sort key and the sort method

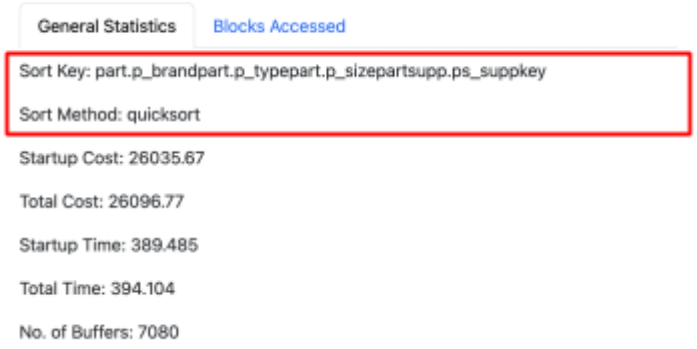
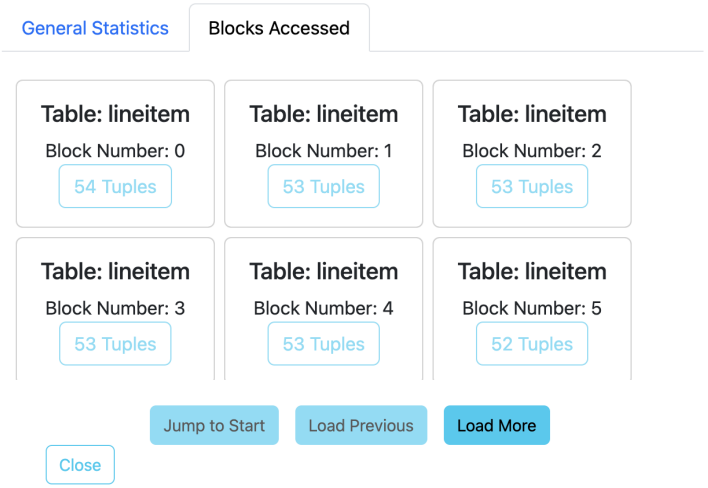


Image: Sort operation

4. Hash - Displays the number of hash buckets

Image: Hash operation

Blocks Accessed Tab



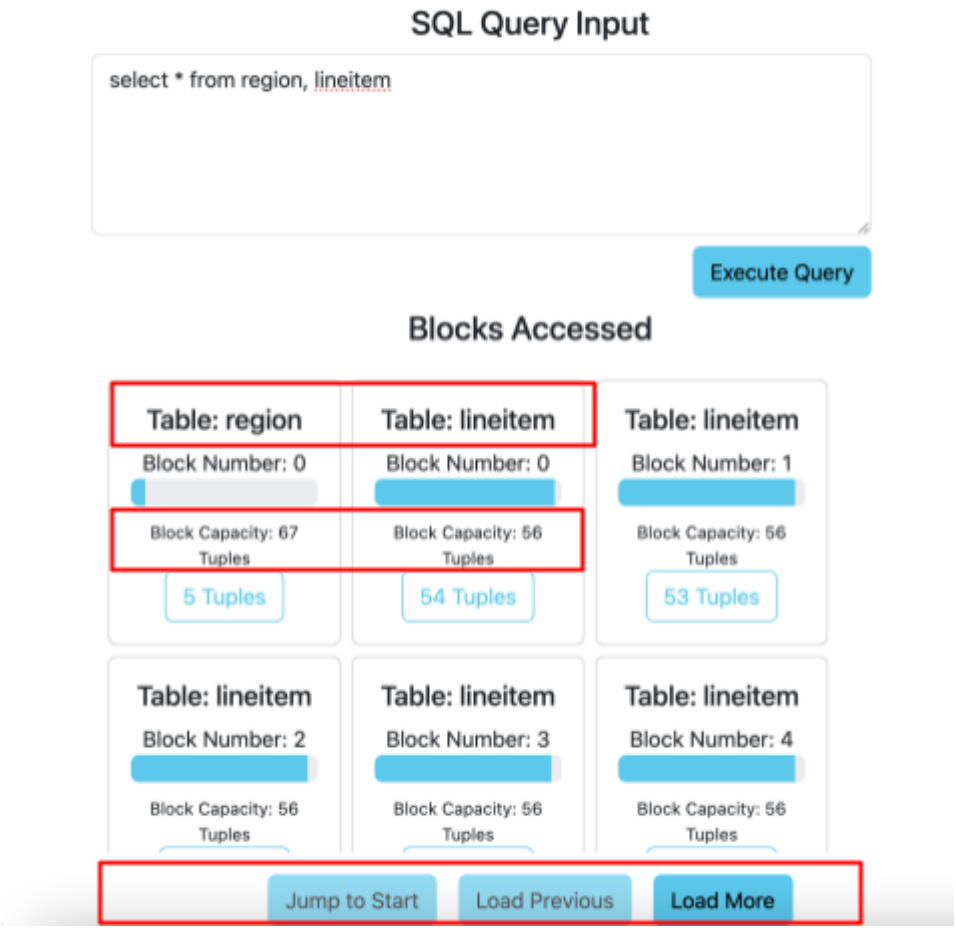
2.3 Blocks Accessed Visualisation

Overview

The block visualisation component on the bottom left of the interface displays the blocks that correspond to the final output of the query, i.e. the blocks that reach the root node of the Query Execution Plan. Each block displays the table from which it stores the tuples and the block number. We also use a percentage bar indicator to see how much data was actually accessed from the blocks. Below the bar we also display the maximum number of tuples the block can hold for records of that table. The algorithm implemented to calculate this is explained in the later sections of the report. Finally the block also displays the tuple count as a clickable button that can be expanded to see the tuples inside the block. Thus, the user can effectively visualise both the number of blocks accessed and how much data from each block has been read with glances.

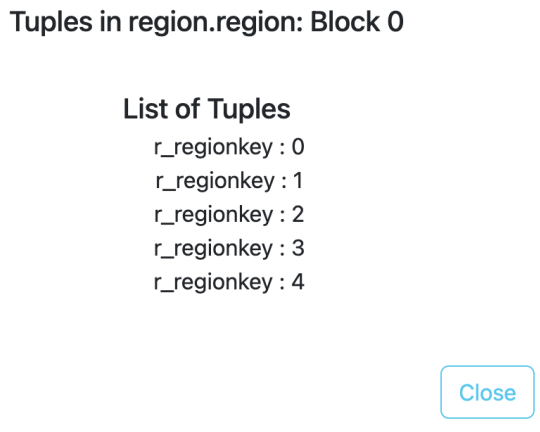
Incremental Rendering

Since some of the query blocks may go into thousands and even tens of thousands due to the size of the tables, they are displayed as a scrollable array of 100 block sets for better readability as well as smooth rendering. Clicking on the load more, renders the next 100 blocks. The user can also view the previous 100 and jump all the way back to the start.



Clicking a Block

Clicking on the tuple count button triggers a modal window that displays the tuple information of that particular block by showing the list of primary keys. This allows the user to interactively explore the content of individual blocks.



3. Backend Implementation

The algorithm for building the tree and extracting the blocks accessed by queries is implemented in Python, utilizing the Flask framework. The backend communicates with PostgreSQL to execute the SQL query and retrieve the QEP.

The Flask framework is a lightweight and versatile web framework for Python. Flask facilitates the communication between the frontend and the PostgreSQL database, handling SQL query execution and QEP retrieval. It exposes two endpoints:

/authenticate: Handles the authentication of the database connection. The credentials provided (host, port, username, password, and database name) are used to establish a connection to the PostgreSQL database.

/explain: Accepts an SQL query, executes it using the PostgreSQL EXPLAIN feature, and returns the Query Execution Plan (QEP) as a JSON response.

Each backend endpoint is implemented as a REST service as shown below:

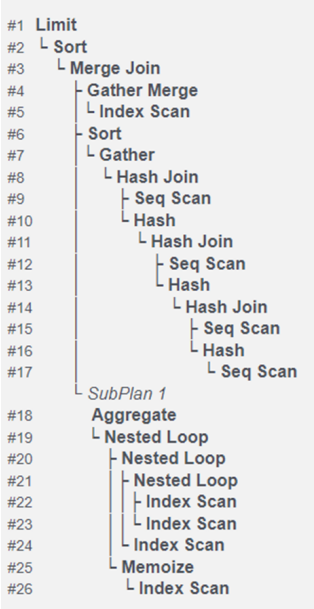
```python
class ValidateDBConnection(Resource):

    # corresponds to the GET request.
    # this function is called whenever there
    # is a GET request for this resource
    lakshya-aga +1
    def get(self):

        response = jsonify({'message': 'Use Post to send SQL'})
        response.status_code = 400
        return response

    # Corresponds to POST request
    Varshbalaji +1
    def post(self):
        requestJSON = request.get_json()  # status code
        dbHostIP = requestJSON.get("dbHostIP")
        dbPort = requestJSON.get("dbPort")
        dbName = requestJSON.get("dbName")
        dbUser = requestJSON.get("dbUser")
        dbPassword = requestJSON.get("dbPassword")
```

The following diagram represents the Plan Tree returned by our backend Explain service for an SQL with a 4-way join having sub-selects:



The nodes in the plan give information about the tables being accessed (relations), data, predicate, grouping and other constructs that can be used to rebuild the SQL at every node level.

Upon analysing how our explain plan for an array of complex queries was constructed, we decided to develop an algorithm that allows us to visualise what blocks are accessed/utilised at each node of the QEP Tree and how they progress through each level. From our analysis, we noticed that Sub-selects get represented either under SubPlans or InitPlan (a special case of subplan that only needs to be run once), each of which we have accounted for in our algorithm for fetching blocks at a node level.

3.1 Dynamic SubQuerying Algorithm

3.1.1 Blocks Accessed (Floating up the Tree):

Note: In our tree, Leaf Nodes display the blocks accessed from the disk while Intermediate nodes show which blocks are utilised at that particular node after applying various predicates. In our front-end, we refer to both as blocks accessed (display convenience).

As we wanted our application to be able to display the blocks utilized at each node (operation) of the tree and how they progress up to the root, we built a dynamic subquerying algorithm for building custom SQLs to fetch the intermediate Blocks Accessed at each node. The data blocks accessed/remaining after filters at each level, are fetched using the Plan Information provided by Postgres Explain. **CTID** is used to extract block and tupleID details. These custom-built SQLs fetch the CTIDs accessed at each node and we enhance our Explain JSON Response returned by Postgres to capture additional custom attributes: Blocks

Accessed at each Node, Tuple Count within that block.The block and tupleID details can then be interactively displayed on the front-end.

The following SQL is used to explain how the backend analyzes the plan and prepares the block visualization data using custom SQLs:

```
select
        s_acctbal,
        s_name,
        n_name,
        p_partkey,
        p_mfgr,
        s_address,
        s_phone,
        s_comment
from
        part,
        supplier,
        partsupp,
        nation,
        region
where
        p_partkey = ps_partkey
        and s_suppkey = ps_suppkey
        and p_size = 15
        and p_type like '%BRASS'
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'EUROPE'
```

One of the ‘Leaf’ Nodes of the Explain Plan of the above query is as follows: It does an “Index Scan” to access “Region” table with specified filters on the SQL:


```
"Plans": [
    {
        "Node Type": "Index Scan",
        "Parent Relationship": "Outer",
        "Parallel Aware": false,
        "Async Capable": false,
        "Scan Direction": "Forward",
        "Index Name": "region_pkey",
        "Relation Name": "region",
        "Schema": "tpch1g",
        "Alias": "region_1",
        "Startup Cost": 0.14,
        "Total Cost": 0.45,
        "Plan Rows": 1,
        "Plan Width": 4,
        "Actual Startup Time": 0.008,
        "Actual Total Time": 0.008,
        "Actual Rows": 0,
        "Actual Loops": 5,
        "Output": [
            "region_1.r_regionkey"
        ],
        "Index Cond": "(region_1.r_regionkey = nation_1.n_regionkey)",
        "Rows Removed by Index Recheck": 0,
        "Filter": "(region_1.r_name = 'EUROPE'::bpchar)",
        "Rows Removed by Filter": 1,
        "Shared Hit Blocks": 8,
        "Shared Read Blocks": 2,
        "Shared Dirtied Blocks": 0,
        "Shared Written Blocks": 0,
        "Local Hit Blocks": 0,
        "Local Read Blocks": 0,
        "Local Dirtied Blocks": 0,
        "Local Written Blocks": 0,
        "Temp Read Blocks": 0,
        "Temp Written Blocks": 0,
```


This information is processed for that particular node to build an SQL that returns its block & tuple information respectively. For the blocks accessed at a particular node, the SQL built at

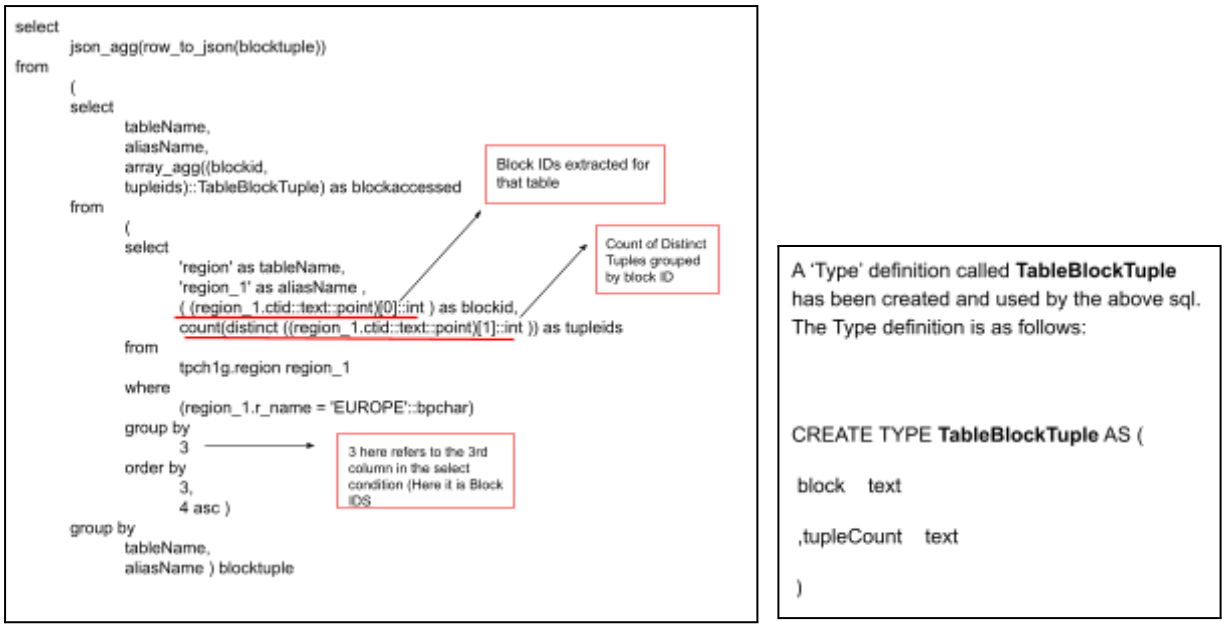
that node's level comprises the blocks accessed by all its children after applying the predicates/conditions at each lower level.

For each node, what is built is :

1. BlocksSQL
2. TuplesSQL

BlocksSQL: The custom SQL built on the backend to fetch CTIDs of intermediate data blocks accessed/utilized at each node using the predicate, and filtering information from the Explain Plan.

A sample custom BlocksSQL built on the backend is shown below :



In this custom SQL, the CTID data is split into blocks and tuples and constructed into an array using the TYPE definition. The query is dynamically built and executed to generate a JSON data element that is then appended to the Explain plan of that particular node as follows:


```
"blocksAccessed": [
    [
        {
            "tablename": "region",
            "aliasname": "region_1",
            "blockaccessed": [
                {
                    "block": "0",
                    "tuplecount": "1"
                }
            ]
        }
    ]
]
```

TuplesSQL: Custom SQL built to fetch query-specific, qualified tuple information of the corresponding block accessed (block specific tuple data)

Format of the SQL:
"tupleSQL": " Select json_agg(row_to_json(blocktuple)) from (select distinct ((?.ctid::text::point)[1]::int) as tupleid, ?.* from tpch1g.region region_1 where (region_1.r_name = 'EUROPE'::bpchar) and ((?.ctid::text::point)[0]::int) = ? order by 2 asc) blocktuple",

This generated Tuple SQL has place holders marked with ? that are replaced dynamically by an individual block number received from the front end. The functionality to execute the TupleSQL is achieved by a special backend REST service to extract tuple data within an individual block.

getBlockTuples: This service is invoked to get the tuple information of individual blocks when they are clicked on by the user on the front-end. Every node, during analysis of Explain, returns the "tupleSQL" JSON attribute that the algorithm built using CTID. This SQL is passed to the server with table related information and the selected block number so that individual data elements of the qualified rows in the block can be returned.

A sample TupleSQL is shown below:

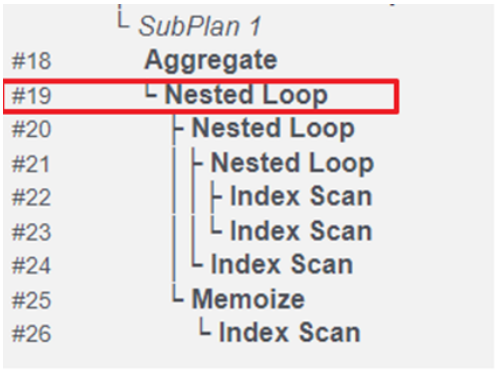
```
select
        json_agg(row_to_json(blocktuple))
from
        (
        select
                distinct (( region_1.ctid::text::point)[1]::int )
as tupleid,
                region_1.*
        from
                tpch1g.region region_1
        where
                (region_1.r_name = 'EUROPE'::bpchar)
                and ( (region_1.ctid::text::point)[0]::int ) =
0
        order by
                2 asc ) blocktuple
```

The response upon execution of the above custom built Tuple SQL query is as follows:

```
[
    {
        "tupleid": 4,
        "r_regionkey": 3,
        "r_name": "EUROPE                    ",
        "r_comment": "ly final courts cajole furiously final excuse"
    }
]
```


As the nodes are processed **'Bottom-up'** using **DFS** tree navigation algorithm, the SQL generated at higher level nodes will have joins that access multiple tables. To consolidate all the blocks accessed across tables, our algorithm captures the CTID related information at each of the lower levels and combines them using SQL UNIONs

As an example, let us consider Node 19 from the aforementioned 4-way join QEP, representing the branch of a SubPlan (sub select) as shown below:



This query has a 4-way join consolidating the 4 Index-Scans and intermediate Nested Loops. Using our algorithm, the Blocks/Tuples SQL generated at a JOIN node will be a UNION of the queries built for all the 4 tables. This results in us obtaining the table-wise blocks, tuples accessed after applying the consolidated predicates/filters at that node level. The consolidated filters will be a combination of all predicates/conditions present at the child nodes as well as our current Join node. The following SQL is dynamically generated on the backend:

```
select json_agg(row_to_json(blocktuple))
from (
    select tableName,   aliasName,  array_agg((blockid,tupleids)::TableBlockTuple) as blockaccessed
    from (select 'partsupp' as tableName, 'partsupp_1' as aliasName , ( (partsupp_1.ctid::text::point)[0]::int ) as blockid,
                count(distinct ((partsupp_1.ctid::text::point)[1]::int )) as tupleids
        from    tpch1g.partsupp partsupp_1, tpch1g.supplier supplier_1, tpch1g.nation nation_1, tpch1g.region region_1
        where   (supplier_1.s_suppkey = partsupp_1.ps_suppkey)
            and (nation_1.n_nationkey = supplier_1.s_nationkey)
            and (region_1.r_regionkey = nation_1.n_regionkey)
            and (region_1.r_name = 'EUROPE'::bpchar)
        group by 3  order by 3, 4 asc )  group by tableName, aliasName
    union
    select tableName,   aliasName,  array_agg((blockid,tupleids)::TableBlockTuple) as blockaccessed
    from (select 'supplier' as tableName, 'supplier_1' as aliasName , ( (partsupp_1.ctid::text::point)[0]::int ) as blockid,
                count(distinct ((partsupp_1.ctid::text::point)[1]::int )) as tupleids
        from    tpch1g.partsupp partsupp_1, tpch1g.supplier supplier_1, tpch1g.nation nation_1, tpch1g.region region_1
        where   (supplier_1.s_suppkey = partsupp_1.ps_suppkey)
            and (nation_1.n_nationkey = supplier_1.s_nationkey)
            and (region_1.r_regionkey = nation_1.n_regionkey)
            and (region_1.r_name = 'EUROPE'::bpchar)
        group by 3  order by 3, 4 asc )  group by tableName, aliasName
    union
    select tableName,   aliasName,  array_agg((blockid,tupleids)::TableBlockTuple) as blockaccessed
    from (select 'nation' as tableName, 'nation_1' as aliasName , ( (partsupp_1.ctid::text::point)[0]::int ) as blockid,
                count(distinct ((partsupp_1.ctid::text::point)[1]::int )) as tupleids
        from    tpch1g.partsupp partsupp_1, tpch1g.supplier supplier_1, tpch1g.nation nation_1, tpch1g.region region_1
        where   (supplier_1.s_suppkey = partsupp_1.ps_suppkey)
            and (nation_1.n_nationkey = supplier_1.s_nationkey)
            and (region_1.r_regionkey = nation_1.n_regionkey)
            and (region_1.r_name = 'EUROPE'::bpchar)
        group by 3  order by 3, 4 asc )  group by tableName, aliasName
    union
    select tableName,   aliasName,  array_agg((blockid,tupleids)::TableBlockTuple) as blockaccessed
    from (select 'region' as tableName, 'region_1' as aliasName , ( (partsupp_1.ctid::text::point)[0]::int ) as blockid,
                count(distinct ((partsupp_1.ctid::text::point)[1]::int )) as tupleids
        from    tpch1g.partsupp partsupp_1, tpch1g.supplier supplier_1, tpch1g.nation nation_1, tpch1g.region region_1
        where   (supplier_1.s_suppkey = partsupp_1.ps_suppkey)
            and (nation_1.n_nationkey = supplier_1.s_nationkey)
            and (region_1.r_regionkey = nation_1.n_regionkey)
            and (region_1.r_name = 'EUROPE'::bpchar)
        group by 3  order by 3, 4 asc )
    group by tableName, aliasName
) blocktuple
```

The JSON returned by above SQL will be as follows:

3.1.2 Blocks Accessed (Overall Query):

In addition to showing how blocks accessed floats up every node in the tree, we also wanted to display the overall blocks accessed by the query.. The overall blocks accessed are the blocks accessed at our root node after our algorithm finishes its traversal. We assigned nodeIDs (added to the explain response) to each node based on our traversal order to make it easy for our frontend to extract the block accessed at the root level (nodeId = 1 for root). On our frontend, we wanted to display the fill percentage of each block to indicate just how full

the blocks were and which tuples were in them. To obtain the fill percentage, we first needed to calculate the block's capacity for a particular table (maximum number of tuples a table's block could hold).

getMaxTuples:
To visually render % usage of blocks in each table, this service is invoked. The service receives tableSchema data viz. 'tpch1g' and automatically calculates the maximum number of rows that can fit into a block for the specific table. It uses the Postgres default block size (8192 bytes) and other postgres functions like pg_column_size to extract the size of data in all rows (excluding index) and the size of each row. TuplesperBlock for a table is calculated using the formula

$$TuplesperBlock = \frac{Blocksize}{(BytesOfAllRowsInTheTable\ /\ Number\ Of\ Rows\ In\ The\ Table)}$$

Denominator: Bytes per Row

3.2 Algorithm Code Analysis (Accounting for Complex Queries)

The SQL passed in the POST request body is used to build postgreSQL Explain command as follows:

```
explainQuerySQL = "explain (analyze, verbose, BUFFERS, FORMAT json) " + querySQL
```

The Explain Query is executed and the Plan Json is extracted and enriched.

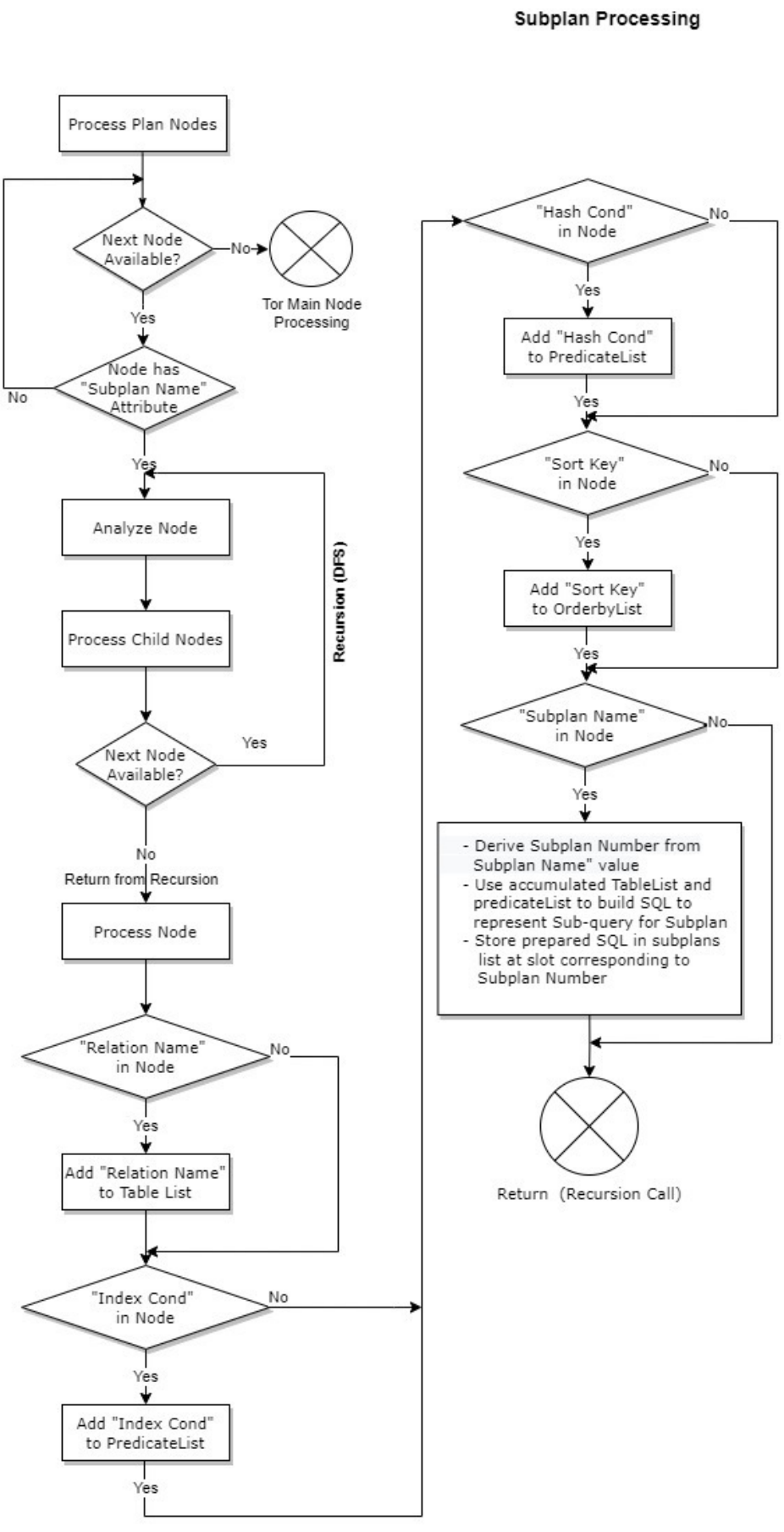
Important Considerations:

1. The following Nodes in the Explain JSON are ignored for specific actions that extract Blocks /Tuples that qualify for that node…

```
ignoreNodes = ["Gather Merge", "Gather", "Hash", "Memoize"]
```

   The reason for ignoring these nodes is that they perform internal actions on the same blocks and hence access the same blocks as their corresponding children. These ignore nodes don't involve any new database actions that would modify the blocks accessed at that level.

2. **Sub Select Processing/ Sub Plan Processing**


Subplan Processing
Process Plan Nodes
Next Node Available?
No
Tor Main Node Processing
Yes
Node has "Subplan Name" Attribute
No
Yes
Analyze Node
Process Child Nodes
Next Node Available?
Yes
Recursion (DFS)
No
Return from Recursion
Process Node
"Relation Name" in Node
No
Yes
Add "Relation Name" to Table List
"Index Cond" in Node
No
Yes
Add "Index Cond" to PredicateList
Yes
"Hash Cond" in Node
No
Yes
Add "Hash Cond" to PredicateList
Yes
"Sort Key" in Node
No
Yes
Add "Sort Key" to OrderbyList
Yes
"Subplan Name" in Node
No
Yes
- Derive Subplan Number from Subplan Name" value
- Use accumulated TableList and predicateList to build SQL to represent Sub-query for Subplan
- Store prepared SQL in subplans list at slot corresponding to Subplan Number
Return (Recursion Call)

During our analysis of the Explain plans for various complex queries, we noticed that the attribute ‘Sub Plan Name’ was included in plans where the query being executed had a sub select (select within a select). To allow our dynamic algorithm for account for such queries, we performed Sub Plan processing to build custom Sub Plan queries (representing the sub select) on the backend:

Nodes are traversed via Recursion based DFS algorithm to first process nodes with “Subplan Name”. Every node is checked for possession of the Subplan attribute and processed from that node level (to the bottom) independently to build subqueries corresponding to the sub plans as shown below.

```
# iterate through nodes tuple for SubPlan Processing
for node in nodes:
    # print("Processing ", node["nodeId"])
    mainPlanProcessing=False
    if ("Subplan Name" in node):
        # print("Processing ", node["nodeId"])
        tableList = ();predicateList = ();orderByList = ();groupByList = ()
        analyze_node(node)
```

tableList, predicateList, orderByList, groupByList are lists maintained while initiating the processing of each node and traversing / processing child nodes from that level. At the end of processing for that node, these lists will have the consolidated tables accessed, predicates (gathered from both the node and its children) and other details required to rebuild queries and extract the CTIDs at that node level.

A flag **mainPlanProcessing** controls if the traversal is for the Subplan or Main Plan node processing. We analyse the Subplan as its own tree (subplan tree), recursively processing predicates/ tables for each of the children in the Subplan using the function **analyze_node(node)** as shown below:

```
def analyze_node(node):
    global tableList, predicateList, orderByList, groupByList, subplans, ignoreNodes,  mainPlanProcessing, limitCount, cacheSQLs

    if 'Plans' in node.keys():
        for childnode in node['Plans']:
            #print("Drilling to ", childnode["nodeId"])
            if not ("Subplan Name" in childnode and mainPlanProcessing):       # Subplan query is already built.. so no need to traverse subplan tree
                analyze_node(childnode)
```

Subplan nodes are typically involved in the ‘Filter’ or ‘Join Filter’ of the main Plan nodes. They can be of the name ‘Subplan #’ or ‘Initplan’

"Subplan Name": "SubPlan 1"

"Subplan Name": "InitPlan 1 (returns $1)" (Init Plan is usually returned for Group by- Having queries. Parent nodes will refer to such a subplan using a $, which is substituted with the sub select query)

The Parent nodes to these subplan nodes will refer to the specific subplan name using either a $ (for Init Subplans) or SubPlan # (normal Sub plan type) in their filters as follows:

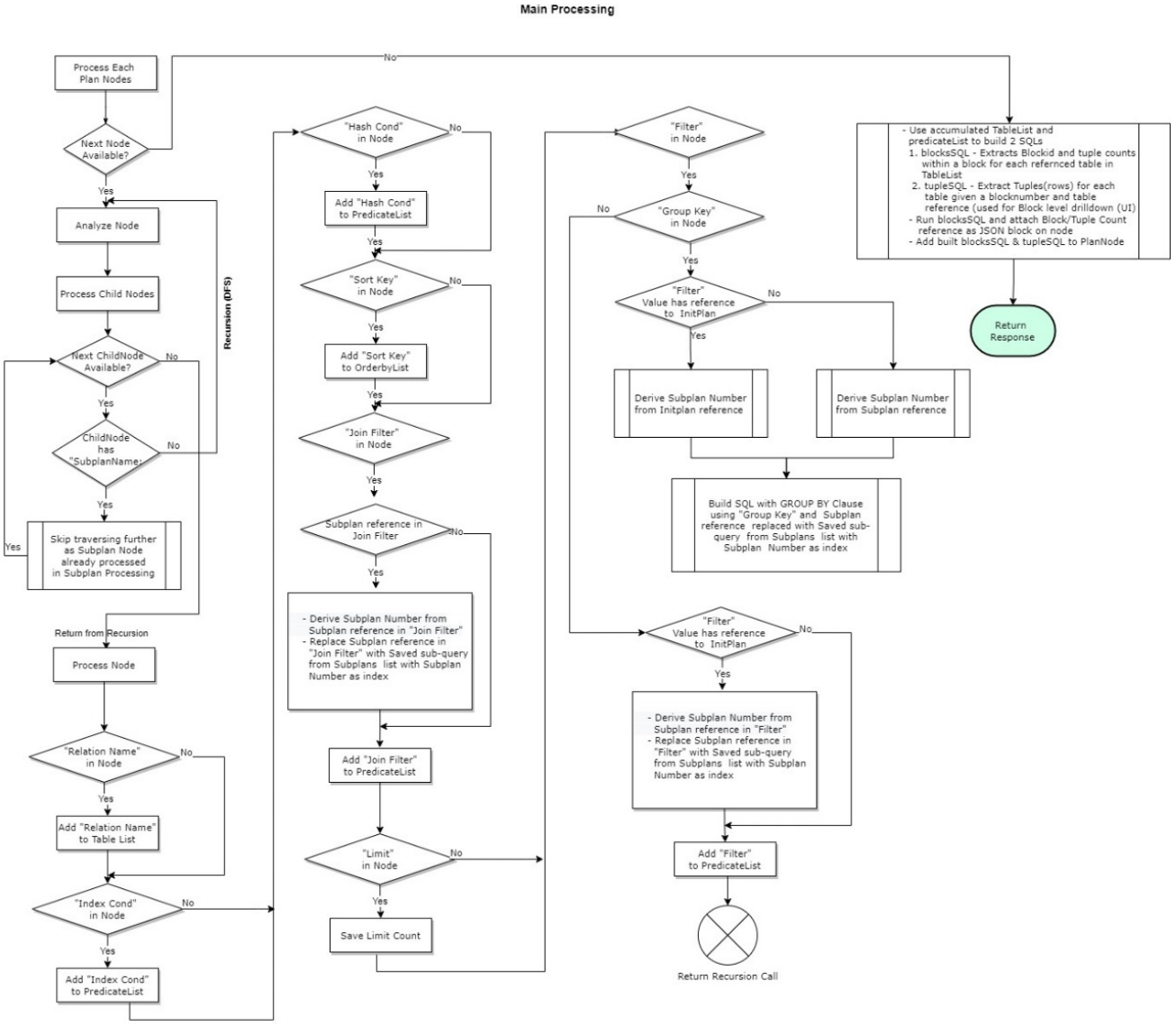
"Join Filter": "(partsupp.ps_supplycost = (**SubPlan 1**))"

"Filter": "(sum((partsupp.ps_supplycost * (partsupp.ps_availqty)::numeric)) > **$1**)"

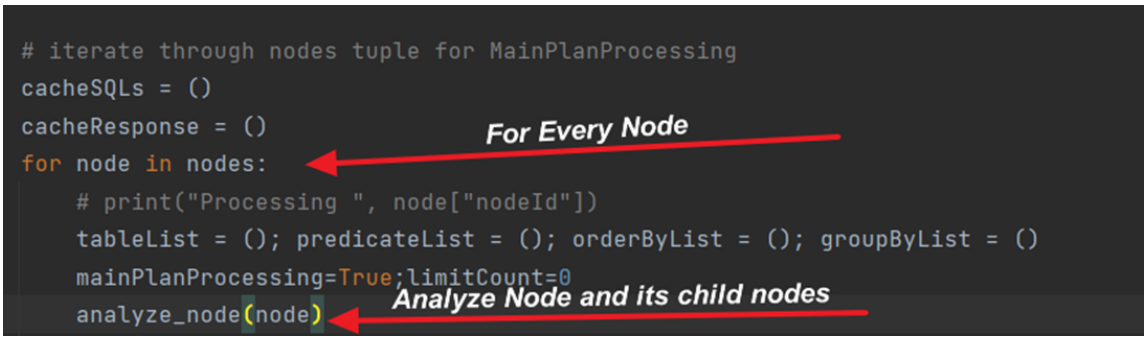
Subplan Processing requires consolidating all tables, predicate details required to rebuild the subquery that can be substituted in the parent node at the slots (instead of (SubPlan 1)) as highlighted above.

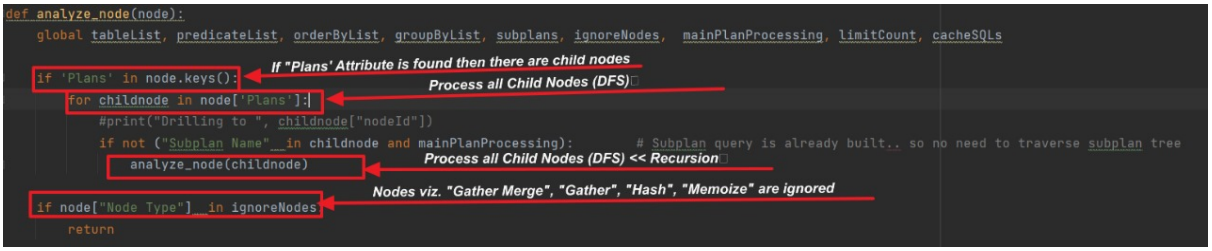
The following code fragment highlights the subplan related processing:

3. **Main Plan Processing:**



After processing subPlans, processing of each node in the main plan is performed. Every node is processed individually using DFS traversal to gather tables, predicate filters and other details required to build the SQL at that node level. This SQL is used to extract CTID information of a main plan node (intermediate blocks Accessed).





While traversing if we hit the child node that has ‘Subplan Name’ and the current processing is mainPlanProcessing (Not subPlanProcessing) then we skip the node along with its child nodes as we have already processed subplans and prepared a “subplans” tuple with subqueries required for the parent node at a higher level.

When the **recursion** returns at **every node** level, we gather the tableList, predicateList and other lists needed to build the SQL to **extract blocks** and tuples at that **intermediate node** level. The block **visualization** is then rendered on the frontend.

The processing of these lists are done as follows within **analyze_node(node)**:

- “Relation Name” attributes if exist are added to tableList (gathering all tables needed to build the SQL at that node level)

```
if ("Relation Name" in node):
    table = node["Schema"]+"."+node["Relation Name"]+ " " + node["Alias"]
    if table not in tableList:
        tableList += (table,)
```

- “Hash Cond” & “Merge Cond” conditions are added to predicateList as predicates required for building the SQL at that node level to filter out the CTIDs being accessed as we float up the tree.

```
if ("Hash Cond" in node):
    predicateVar = node["Hash Cond"]
    if predicateVar not in predicateList:
        predicateList += (predicateVar,)

if ("Merge Cond" in node):
    predicateVar = node["Merge Cond"]
    if predicateVar not in predicateList:
        predicateList += (predicateVar,)
```

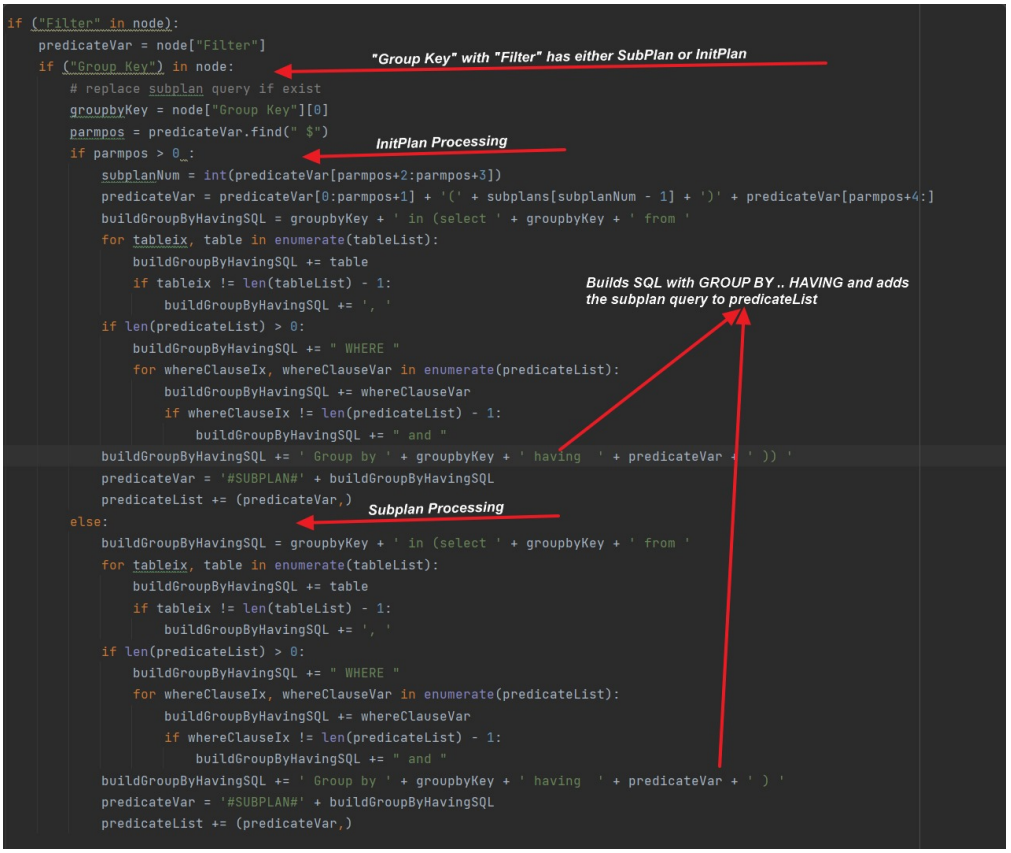
- We observed that the **"Join Filter" attribute** is generated in the main Plan (for a main plan node) to refer to Subplans. The Subplan number referred to in this node is extracted and the subplan query based on the slot number (each slot in the tuple list that collects subplans refers to a particular subplan number) is added to the predicate list for that main plan node. As these queries are already validated while doing subplan processing, a prefix #SUBPLAN# is added to subplan predicate before adding it to the list (stripped off while building SQL). Using this marker no further validations are performed.

```
if ("Join Filter" in node):
    predicateVar = node["Join Filter"]
    subplanPos = predicateVar.find("SubPlan ")
    if subplanPos >= 0 :
        #subplanName = predicateVar[subplanPos: subplanPos + 10]
        subplanNum  = int(predicateVar[subplanPos + 8: subplanPos + 9])
        predicateVar = '#SUBPLAN#' + predicateVar[0: subplanPos ] + subplans[subplanNum-1] + predicateVar[subplanPos + 9: ]
        predicateList += (predicateVar,);
```

- Filter" conditions are processed as follows

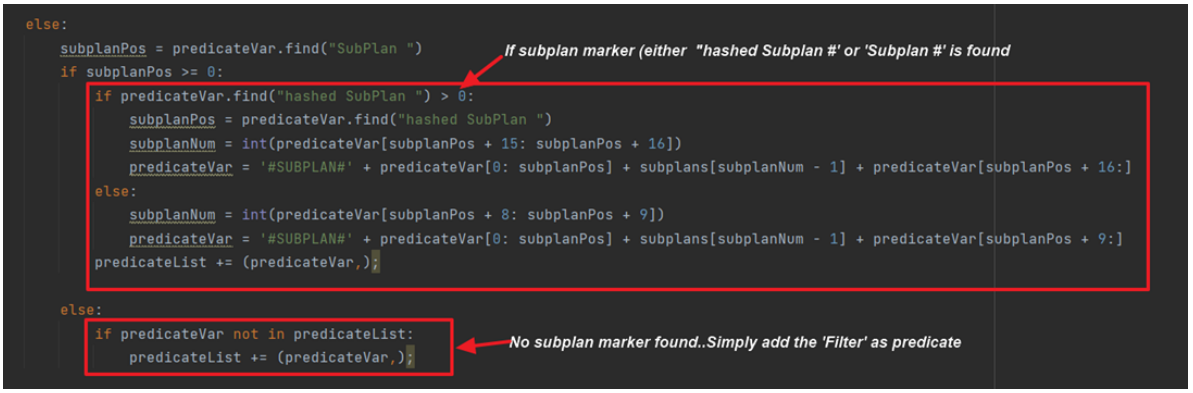
ACCOUNTING FOR GROUP BY:

If "Group Key" exists along with "Filter" then the sql to be constructed will have the " GROUP BY …. HAVING …' clause. Group Key Filters always have a subplan which can be either of type Subplan or type InitPlan ($ marker). The subplan query built in either case is added to the predicate list as accounted below:



The constructed subquery (filter) is added to predicate list

If "Group Key" doesn't exist, the filter is first checked for whether it includes a SubPlan based on whether it contains the marker. If the Subplan marker is found then the subplan number is extracted to perform a look up in the subplans list at the slot corresponding to the subplan number. The marker in the filter is then replaced with the corresponding subquery at that position in the list as shown below.



ACCOUNTING FOR LIMIT:

If SQL analyzed has a LIMIT then the plan will have a "Node Type" captured with Limit. This node is processed to capture the limit on the number of rows that are

returned. The limit count is then added to the TupleSQLs to mimic the limit on number of tuples returned

```
if (node["Node Type"] == "Limit"):
    limitCount = node["Actual Rows"]
```

Construction Of The Node Wise Queries To Extract CTIDs

When the DFS based node processing is done on a particular node then we have all the details in various lists: tableList, predicateList etc to finally build the SQLs to extract the CTIDs at that node level.

Data attributes “blocksSQL” & “tupleSQL” are built and added to the explain JSON at each node level. In our algorithm for generating these SQLs, we first add the initial standard part of the Select SQL to capture JSON format of Blocks and Tuples and then add the tables to the ‘FROM’ part of the SQL using tableList.

```
if node["Node Type"] not in ignoreNodes and not "Subplan Name" in node :
    blocksSQL = " Select json_agg(row_to_json(blocktuple)) from ( "
    tupleSQL = " Select json_agg(row_to_json(blocktuple)) from ( "
    for tabix, table in enumerate(tableList) :
        tableName = tableNames[tabix]
        tableAlias = tableAliases[tabix]
        blocksSQL += "select  tableName, aliasName, array_agg((blockid,tupleids)::TableBlockTuple) as blockaccessed from  ( "
        blocksSQL += " select '" + tableName + "' as tableName, '" + tableAlias \
                  + "' as aliasName , ( (" + tableAlias + ".ctid::text::point)[0]::int ) as blockid, count(distinct ((" \
                  + tableAlias + ".ctid::text::point)[1]::int )) as tupleids"
        blocksSQL += " from "

        if tabix == 0 :
            tupleSQL += " select   distinct (( ?" + ".ctid::text::point)[1]::int ) as tupleid, "+ "?.*"
            tupleSQL += " from "
        for tableix, allTable in enumerate(tableList):
            blocksSQL += allTable
            if tabix == 0:
                tupleSQL += allTable
            if tableix != len(tableList) - 1:
                blocksSQL +=  ', '
                if tabix == 0:
                    tupleSQL += ', '
```

This is followed by the addition of predicates along with standard SQL code to complete the query for that particular node. The above block SQL construction is done for each table in the list and are combined with a ‘Union’. ‘Limit Count’ if captured on the plan will be added with ‘ LIMIT ‘ and this ensures that when blocks and tuples are returned for visualization the corresponding limits are enforced.

The “blocksSQL” that extracts filtered blocks, tuple counts within the blocks for each table is executed at each node level and the sql response (JSON) is added as the “blocksAccessed” attribute on the node within the Explain Plan.

3.3. Performance Improvement

Caching:

While implementing our algorithm, we observed that building a query at each level increased the time required to return our response to the front-end. In order to optimise our time required to render our visualisations on the front-end, we employed the technique of Caching to cache the responses to blockSQLs that are repeatedly executed as we progress through the tree. When our algorithm identified 'Aggregation' or 'Sort' nodes, the SQL constructed to extract block/tuple info is the same as its child node that would have been a 'Nested Loop' or 'Hash Join'. To avoid overheads with running the same queries to capture block/tuples info, a caching mechanism was set up that caches in a list the "blocksSQL" as well as "blocksSQLResponse". The cache list is looked up for matching SQLs. If a match is found, the cached response is added for the node instead of repeating the sql execution to capture blocksSQLResponse

4. Limitations

Frontend

1. **Delayed Rendering of Block Accessed within the tree -** Due to certain limitations of the Echarts library, when the user wants to view more blocks in the Blocks Accessed tab within the node of a tree, there is a rendering delay after clicking on the Load More button. To view more blocks, the user must exit the pop up and reopen it. This can be accommodated for in the future by exploring newly updated libraries to visualise the tree.
2. **Difficulty displaying the entire tuple information within a Block** - Due to each table containing a different number of attributes, even though the backend was returning the tuple with data for all of its attributes, we were only able to display the primary key. This is because our modal component was implemented keeping in mind reusability across tables and datasets. In the future, we can try to implement a dynamic table construction system to display the entire tuples.

Backend

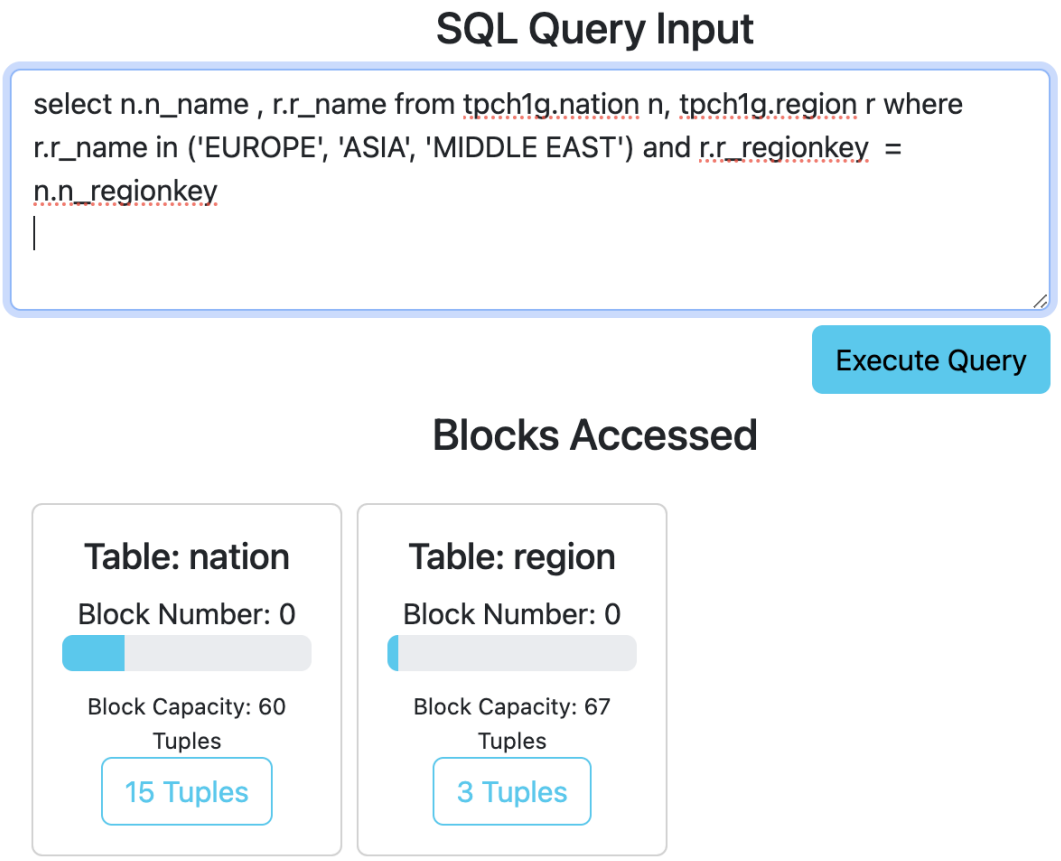
1. **Performance Overhead -** Although the backend employs caching to avoid repeated execution of subqueries, the complexity of the queries as well as the size of the tables can influence the processing and execution time. This results in a few queries taking over 40 seconds to completely render on the interface. This bottleneck is also due the time Postgres takes to generate the Explain plan. Indexes can be manually added on our database side to speed up the execution.

5. Sample Test Cases Considered

1. **Two- Way Join**

```
select n.n_name , r.r_name from
tpch1g.nation n, tpch1g.region r
where r.r_name in ('EUROPE',
'ASIA', 'MIDDLE EAST') and
r.r_regionkey  = n.n_regionkey
```

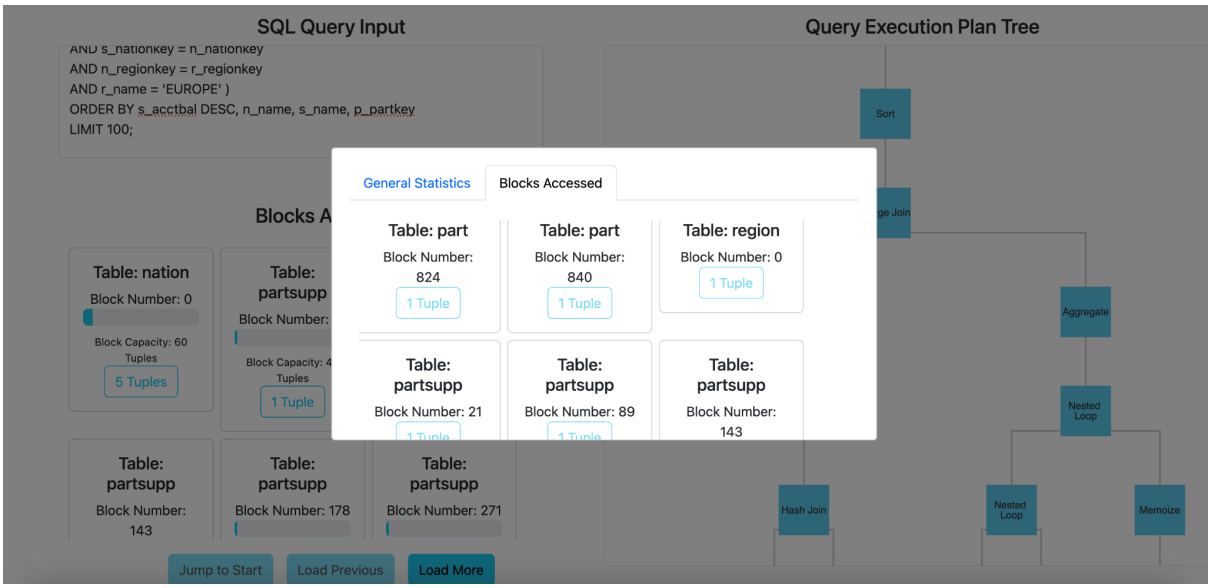
This test case demonstrates how our application handles a two way join and returns the block accessed/ utilised at each tree node table wise



2. Four-Way Join

```
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address,
s_phone, s_comment
FROM part, supplier, partsupp, nation, region
WHERE
p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND p_size = 15
AND p_type LIKE '%BRASS'
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE'
AND ps_supplycost = (
SELECT
MIN(ps_supplycost)
FROM partsupp, supplier, nation,region
WHERE
p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE' )
ORDER BY s_acctbal DESC, n_name, s_name, p_partkey
LIMIT 100;
```

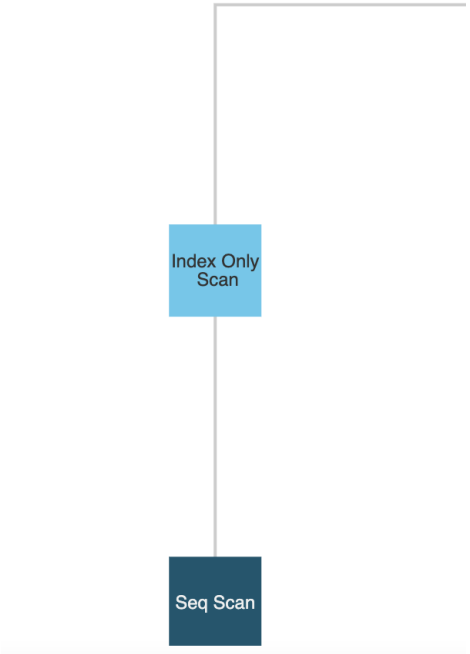
This test case demonstrates how our applications handles a complex 4 way join query with a **LIMIT**, generating **table-wise blocks accessed at the respective nodes**



3. **NOT IN**

```
SELECT p_brand, p_type, p_size, COUNT(DISTINCT
ps_suppkey) AS supplier_cnt FROM partsupp, part
WHERE p_partkey = ps_partkey AND p_brand <>
'Brand#45' AND p_type NOT LIKE 'MEDIUM POLISHED%'
AND p_size IN (49, 14, 23, 45, 19, 3, 36, 9) AND
ps_suppkey NOT IN ( SELECT s_suppkey FROM supplier
WHERE s_comment LIKE '%Customer%Complaints%' )
GROUP BY p_brand, p_type, p_size ORDER BY
supplier_cnt DESC, p_brand, p_type, p_size
```

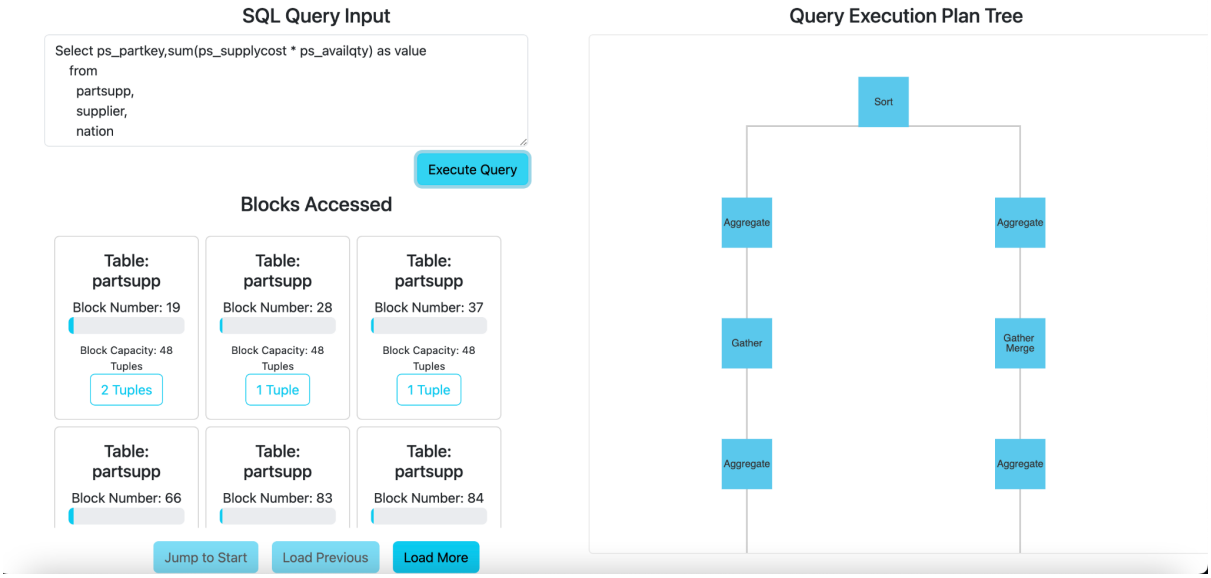
This test case is used to show how the tree is built for queries that use NOT IN. We can see from the tree that the sequential scan at the bottom is involved in a sub plan for only supplier (sub select) and the index only scan retrieves only blocks accessed for partsupp after applying the suplan as a predicate



4. Sub Select (Using Init Subplan)

```
Select ps_partkey,sum(ps_supplycost * ps_availqty) as value
  from
   partsupp,
   supplier,
   nation
  where
   ps_suppkey = s_suppkey
   and s_nationkey = n_nationkey
   and n_name = 'GERMANY'
   and ps_supplycost > 20
   and s_acctbal > 10
  group by
   ps_partkey having
    sum(ps_supplycost * ps_availqty) > (
     select
      sum(ps_supplycost * ps_availqty) * 0.0001000000
     from
      partsupp,
      supplier,
      nation
     where
      ps_suppkey = s_suppkey
      and s_nationkey = n_nationkey
      and n_name = 'GERMANY'
    )
  order by
   value desc
```

This test case shows how our application handles sub selects and uses blocks accessed/ utilzed for subplans



5. **Select on a large table**

```
select * from lineitem
```

Our application allows us to view all the blocks accessed from a large table like lineitem with improved performance. All the Blocks can be viewed by clicking the “Load More” button.