NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Project 2 SC3020

AY2023-2024

Group ID: 10

Group members:

Name	Matric No.
Ng Ho Chi	U2122010C
Koh Ming En	U2121271F
Koh Jin Kiong Brian	U2022245E
Kng Yew Chian	U2021777G

1. Project Summary

Our project aims to facilitate visual exploration of disk blocks accessed by a given input SQL query as well as various features of the corresponding query execution plan (QEP). We first utilize the `EXPLAIN` keyword provided by PostgreSQL to generate the QEP, together with the (`ANALYSE, BUFFERS, COST`) parameters to output the metadata of the QEP. We visualize the QEP using a Reingold Tilford tree structure and display the plan and associated cost at each node.

To provide alternative query plans, we allow the user to toggle certain scans on and off using the `SET` keyword provided by PostgreSQL. By toggling off the access method, PostgreSQL will prioritize other access methods unless absolutely necessary, which in general will generate a new QEP.

Lastly, to visualize the disk blocks accessed by the SQL query, we utilize the `ctid` column provided by PostgreSQL to determine the number of tuples accessed per data block. We visualize this distribution by plotting it in a histogram.

2. Installation Guide

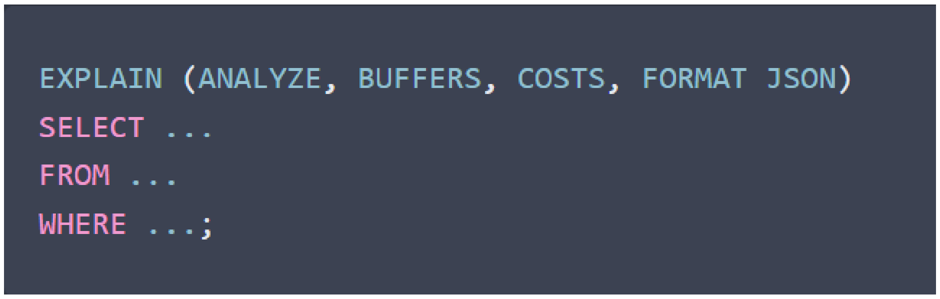
Do follow the readme provided in the folder to:

1. Install PostgreSQL
2. Import the TPC-H dataset
3. Install all python dependencies
4. Update environment variables
5. Run the application

3. Software Details

Visualizing the QEP

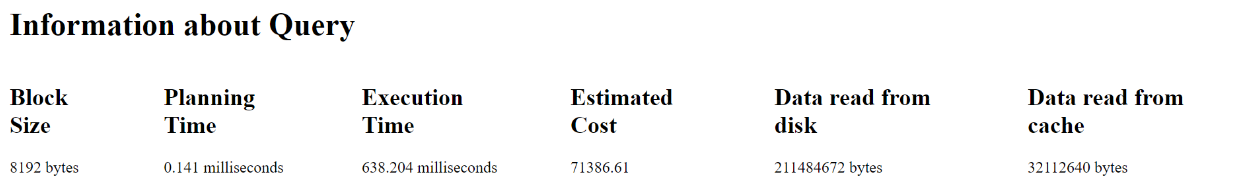
We utilize the `EXPLAIN (ANALYSE, BUFFERS, COST FORMAT JSON)` keywords provided by PostgreSQL to retrieve the QEP in JSON format, which is easier for us to parse. The structure of the query is as follows, where the above statement is appended before the provided query:



```sql
EXPLAIN (ANALYZE, BUFFERS, COSTS, FORMAT JSON)
SELECT ...
FROM ...
WHERE ...;
```

Figure 1: SQL Query to retrieve QEP

We first retrieve the metadata from the QEP, which includes data such as the planning time, execution time and buffers used. We display the metadata to the web application after the user submits their SQL query.

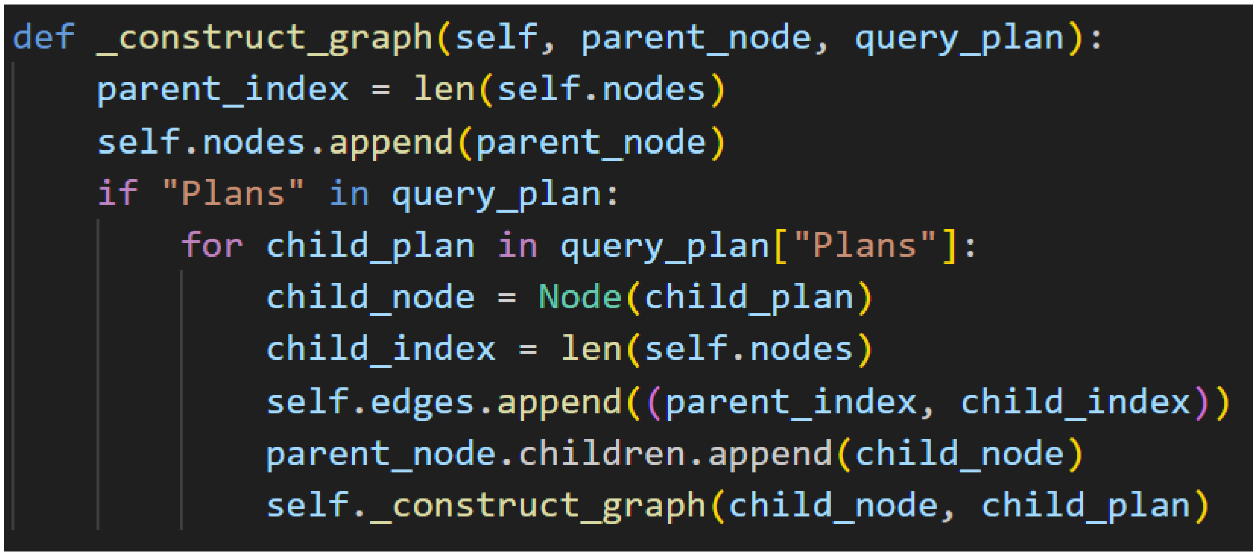


Information about Query

Block Size	Planning Time	Execution Time	Estimated Cost	Data read from disk	Data read from cache
8192 bytes	0.141 milliseconds	638.204 milliseconds	71386.61	211484672 bytes	32112640 bytes

Figure 2: Metadata about query

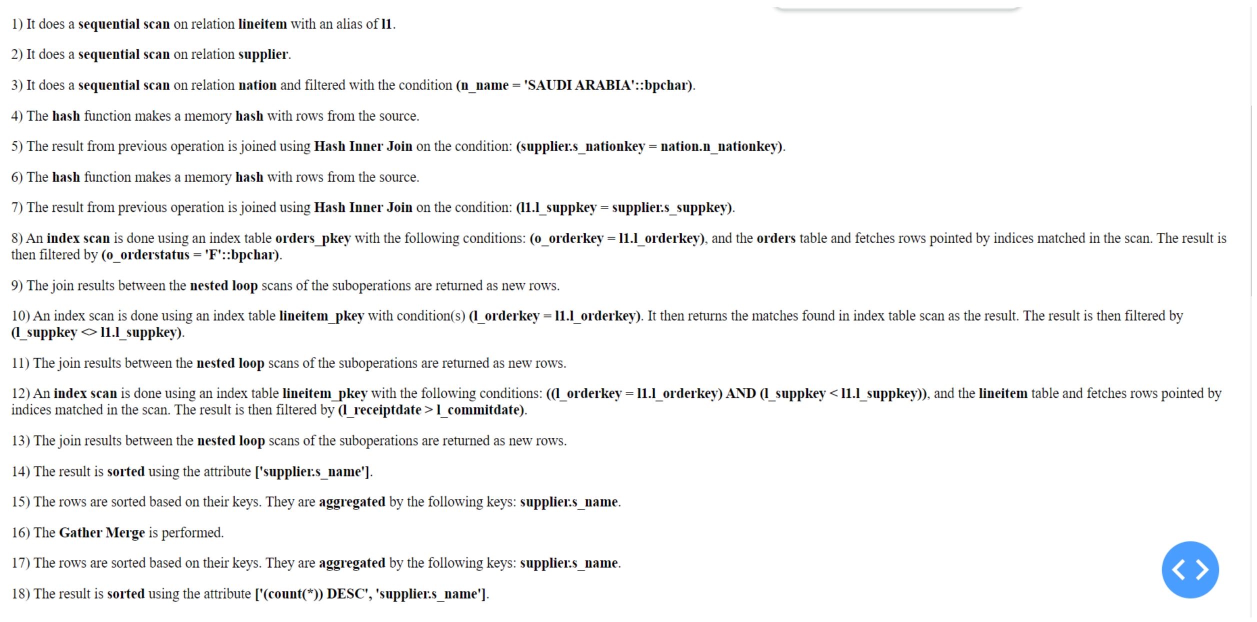
Next, we generate the tree structure of the QEP using a **depth-first traversal** algorithm, where we explore the child nodes starting from the root recursively until we have reached a leaf node, followed by backtracking to visit the other nodes. In figure 3, we provide the code for the algorithm.



```python
def _construct_graph(self, parent_node, query_plan):
    parent_index = len(self.nodes)
    self.nodes.append(parent_node)
    if "Plans" in query_plan:
        for child_plan in query_plan["Plans"]:
            child_node = Node(child_plan)
            child_index = len(self.nodes)
            self.edges.append((parent_index, child_index))
            parent_node.children.append(child_node)
            self._construct_graph(child_node, child_plan)
```

Figure 3: Depth-first traversal of QEP

After constructing the graph, we utilize it to translate the QEP into a human readable format. As PostgreSQL provides the plan name, estimated cost for each plan and the format of the plan, we convert this information into a sentence describing the plan in english. Figure 4 demonstrates a sample translation of the QEP for a query.



1) It does a **sequential scan** on relation **lineitem** with an alias of **l1**.

2) It does a **sequential scan** on relation **supplier**.

3) It does a **sequential scan** on relation **nation** and filtered with the condition **(n_name = 'SAUDI ARABIA'::bpchar)**.

4) The **hash** function makes a memory **hash** with rows from the source.

5) The result from previous operation is joined using **Hash Inner Join** on the condition: **(supplier.s_nationkey = nation.n_nationkey)**.

6) The **hash** function makes a memory **hash** with rows from the source.

7) The result from previous operation is joined using **Hash Inner Join** on the condition: **(l1.l_suppkey = supplier.s_suppkey)**.

8) An **index scan** is done using an index table **orders_pkey** with the following conditions: **(o_orderkey = l1.l_orderkey)**, and the **orders** table and fetches rows pointed by indices matched in the scan. The result is then filtered by **(o_orderstatus = 'F'::bpchar)**.

9) The join results between the **nested loop** scans of the suboperations are returned as new rows.

10) An index scan is done using an index table **lineitem_pkey** with condition(s) **(l_orderkey = l1.l_orderkey)**. It then returns the matches found in index table scan as the result. The result is then filtered by **(l_suppkey <> l1.l_suppkey)**.

11) The join results between the **nested loop** scans of the suboperations are returned as new rows.

12) An **index scan** is done using an index table **lineitem_pkey** with the following conditions: **((l_orderkey = l1.l_orderkey) AND (l_suppkey < l1.l_suppkey))**, and the **lineitem** table and fetches rows pointed by indices matched in the scan. The result is then filtered by **(l_receiptdate > l_commitdate)**.

13) The join results between the **nested loop** scans of the suboperations are returned as new rows.

14) The result is **sorted** using the attribute **['supplier.s_name']**.

15) The rows are sorted based on their keys. They are **aggregated** by the following keys: **supplier.s_name**.

16) The **Gather Merge** is performed.

17) The rows are sorted based on their keys. They are **aggregated** by the following keys: **supplier.s_name**.

18) The result is **sorted** using the attribute **['(count(*)) DESC', 'supplier.s_name']**.

Figure 4: An example QEP translated to human-readable format

Additionally, we also use the python library `igraph` to plot the graph on the web application. We use the Reingold Tilford layout to display the nodes in a tree-like structure. The graph is fully interactive, meaning that the user can focus on certain sections of the QEP, hover over the node to view the full plan in human-readable format and download the plot. An example of the tree structure is showcased in figure 5.

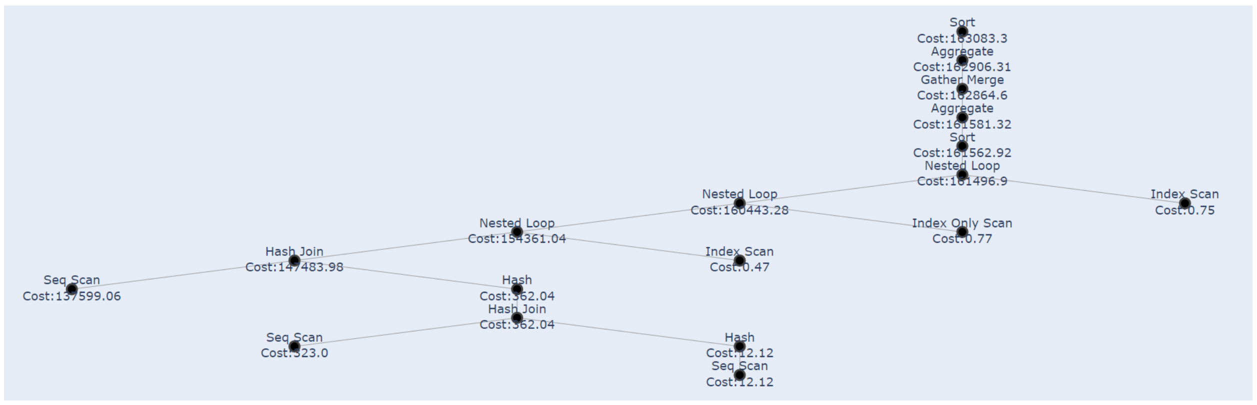
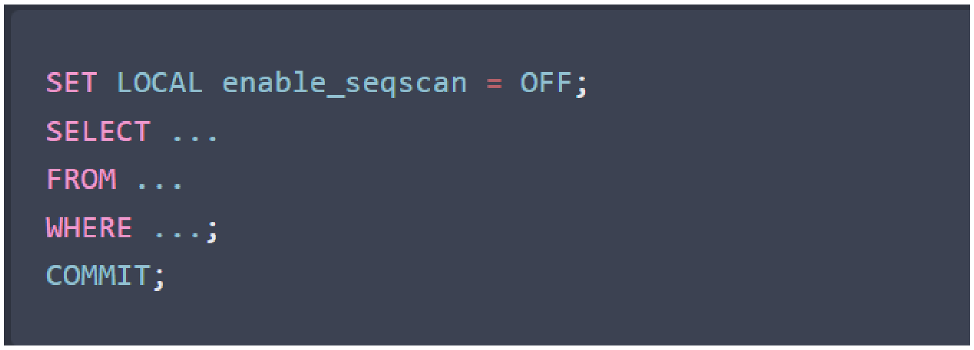


Figure 5: An example graph displayed in the web application

Alternative Query Plans

While PostgreSQL does not display the alternative query plans it considers by default, we can manipulate it by disabling some access methods to force PostgreSQL to prioritize other scans first. We achieve this by using the `SET LOCAL` keyword to update environment variables such as `enable_seqscan, enable_indexscan, enable_bitmapscan`. If the scan is disabled, PostgreSQL will give it a very high cost, such that it will only consider it if no other scans are feasible. Figure 6 provides a sample query with sequential scans disabled.

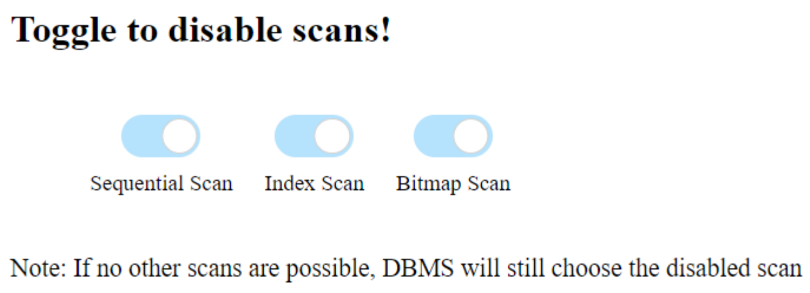


```
SET LOCAL enable_seqscan = OFF;
SELECT ...
FROM ...
WHERE ...;
COMMIT;
```

Figure 6: A sample query with sequential scans disabled

The COMMIT statement is necessary after the query to reset the environment variables back to default values (all scans are enabled).

We provide three toggleable buttons in the web application for the user to disable the scans before analyzing the query.

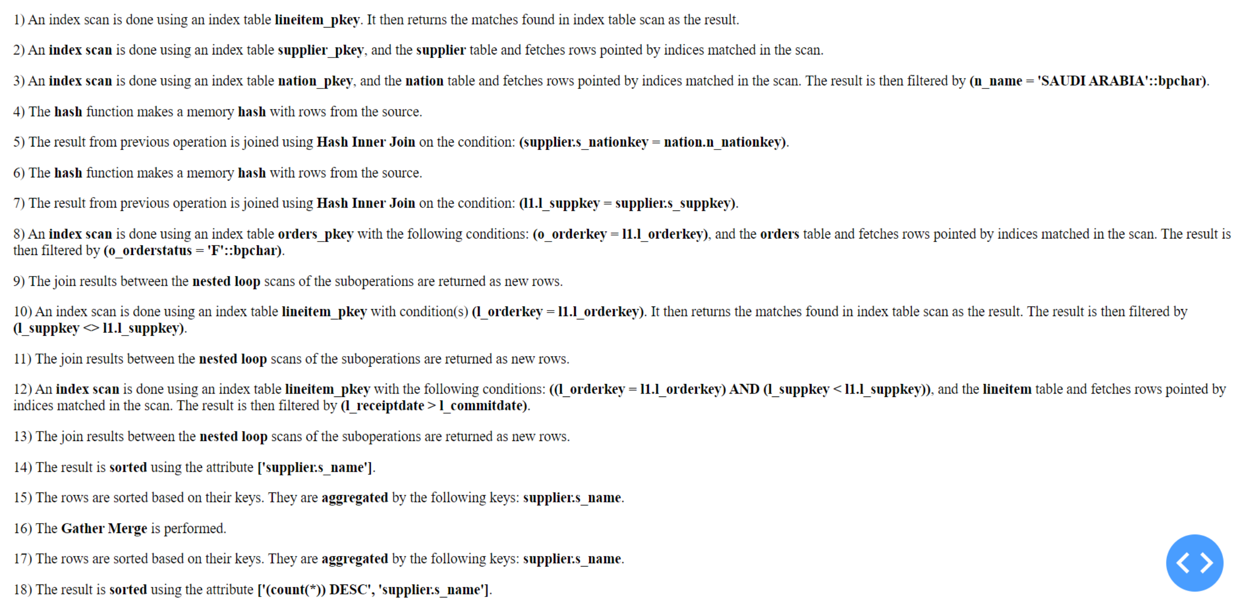


Toggle to disable scans!

Note: If no other scans are possible, DBMS will still choose the disabled scan

Figure 7: Toggleable buttons to disable scan

We demonstrate this feature by disabling sequential scans for the same query as the QEP from figure 5. As seen in figure 8, PostgreSQL instead chooses to perform index scan instead of sequential scans.



1) An index scan is done using an index table **lineitem_pkey**. It then returns the matches found in index table scan as the result.

2) An **index scan** is done using an index table **supplier_pkey**, and the **supplier** table and fetches rows pointed by indices matched in the scan.

3) An **index scan** is done using an index table **nation_pkey**, and the **nation** table and fetches rows pointed by indices matched in the scan. The result is then filtered by **(n_name = 'SAUDI ARABIA'::bpchar)**.

4) The **hash** function makes a memory **hash** with rows from the source.

5) The result from previous operation is joined using **Hash Inner Join** on the condition: **(supplier.s_nationkey = nation.n_nationkey)**.

6) The **hash** function makes a memory **hash** with rows from the source.

7) The result from previous operation is joined using **Hash Inner Join** on the condition: **(l1.l_suppkey = supplier.s_suppkey)**.

8) An **index scan** is done using an index table **orders_pkey** with the following conditions: **(o_orderkey = l1.l_orderkey)**, and the **orders** table and fetches rows pointed by indices matched in the scan. The result is then filtered by **(o_orderstatus = 'F'::bpchar)**.

9) The join results between the **nested loop** scans of the suboperations are returned as new rows.

10) An index scan is done using an index table **lineitem_pkey** with condition(s) **(l_orderkey = l1.l_orderkey)**. It then returns the matches found in index table scan as the result. The result is then filtered by **(l_suppkey <> l1.l_suppkey)**.

11) The join results between the **nested loop** scans of the suboperations are returned as new rows.

12) An **index scan** is done using an index table **lineitem_pkey** with the following conditions: **((l_orderkey = l1.l_orderkey) AND (l_suppkey < l1.l_suppkey))**, and the **lineitem** table and fetches rows pointed by indices matched in the scan. The result is then filtered by **(l_receiptdate > l_commitdate)**.

13) The join results between the **nested loop** scans of the suboperations are returned as new rows.

14) The result is **sorted** using the attribute **['supplier.s_name']**.

15) The rows are sorted based on their keys. They are **aggregated** by the following keys: **supplier.s_name**.

16) The **Gather Merge** is performed.

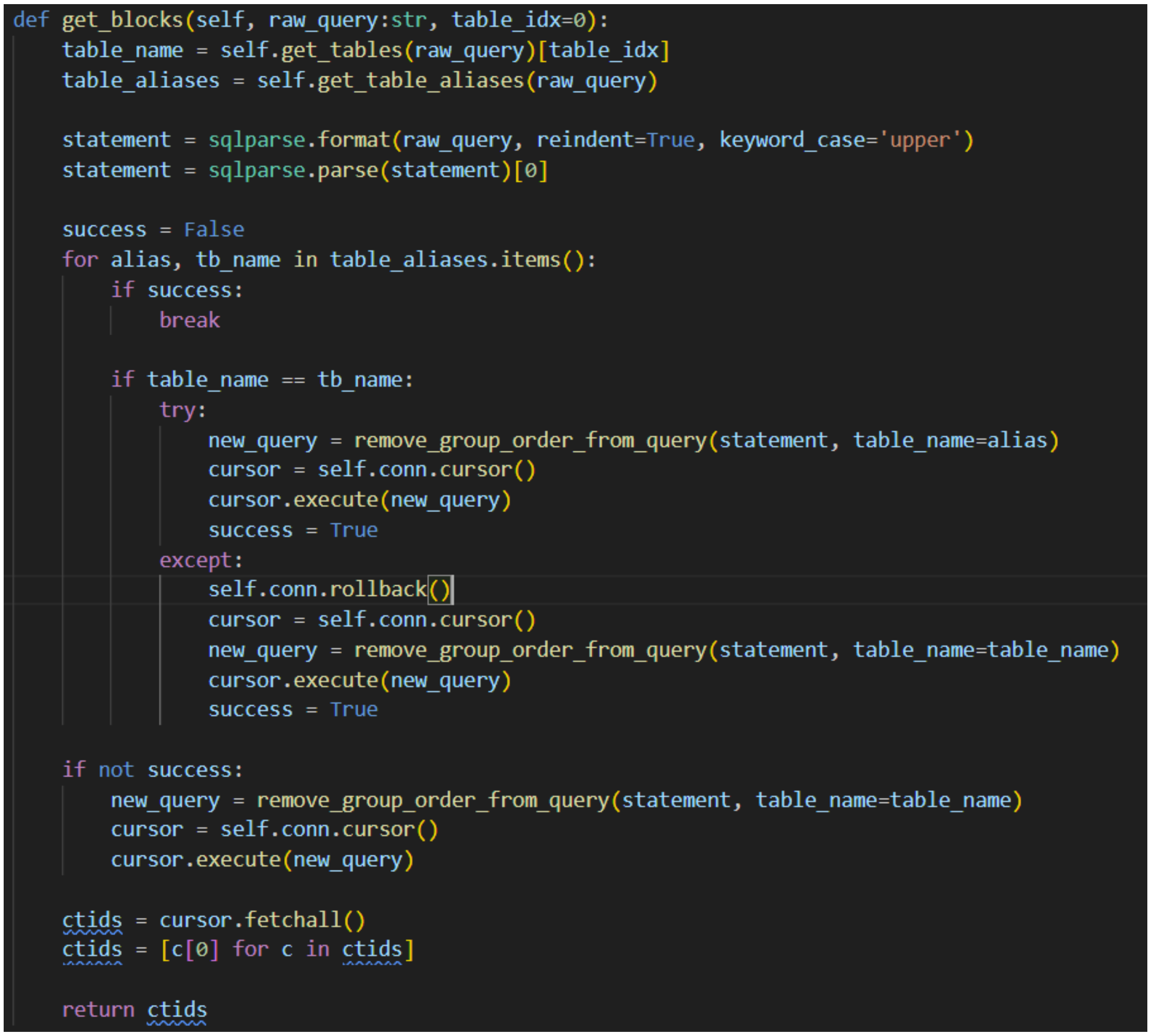
17) The rows are sorted based on their keys. They are **aggregated** by the following keys: **supplier.s_name**.

18) The result is **sorted** using the attribute **['(count(*)) DESC', 'supplier.s_name']**.

Figure 8: PostgreSQL choosing to perform index scan instead of sequential scans

Visualizing data blocks

To visualize how much data is retrieved from each block, we parse the given query using the python libraries sql_metadata and sqlparse and instead select only the `ctid` columns, which provide the block number that each row belongs to. Figure 9 demonstrates our algorithm for parsing the query.



```python
def get_blocks(self, raw_query:str, table_idx=0):
    table_name = self.get_tables(raw_query)[table_idx]
    table_aliases = self.get_table_aliases(raw_query)

    statement = sqlparse.format(raw_query, reindent=True, keyword_case='upper')
    statement = sqlparse.parse(statement)[0]

    success = False
    for alias, tb_name in table_aliases.items():
        if success:
            break

        if table_name == tb_name:
            try:
                new_query = remove_group_order_from_query(statement, table_name=alias)
                cursor = self.conn.cursor()
                cursor.execute(new_query)
                success = True
            except:
                self.conn.rollback()
                cursor = self.conn.cursor()
                new_query = remove_group_order_from_query(statement, table_name=table_name)
                cursor.execute(new_query)
                success = True

    if not success:
        new_query = remove_group_order_from_query(statement, table_name=table_name)
        cursor = self.conn.cursor()
        cursor.execute(new_query)

    ctids = cursor.fetchall()
    ctids = [c[0] for c in ctids]

    return ctids
```

Figure 9: Algorithm to parse SQL query

We then aggregate the block numbers and plot it on a histogram using the `plotly` library. Figure 10 demonstrates our histogram for a sample query.

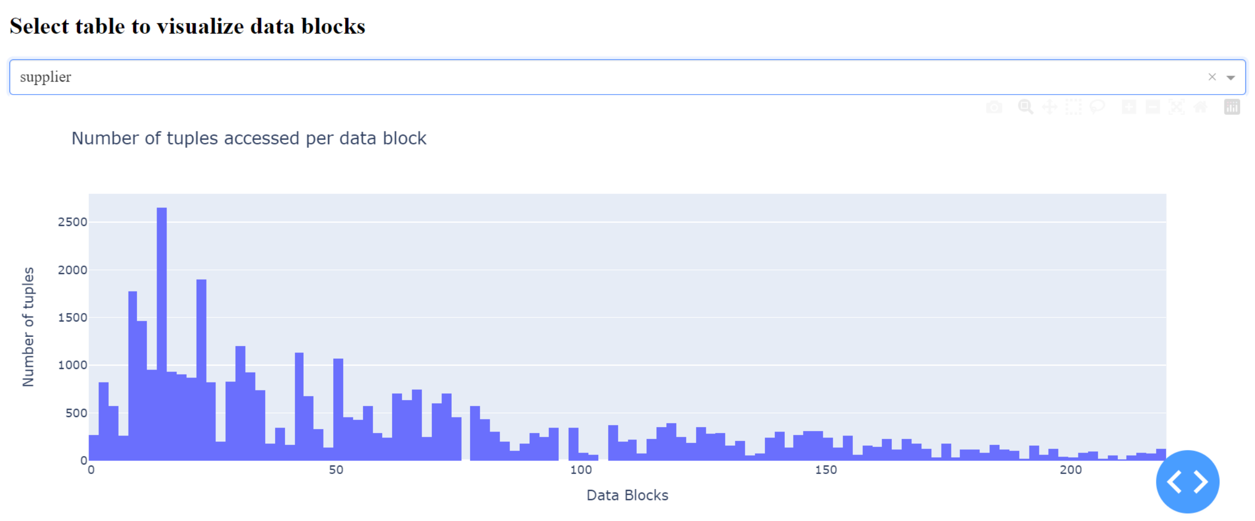

Select table to visualize data blocks
supplier
Number of tuples accessed per data block
Number of tuples
Data Blocks
0
500
1000
1500
2000
2500
0
50
100
150
200

Figure 10: An example histogram on the web application

As ctid is unique to each table, we provide a dropdown table for the user to select which table to visualize the data blocks for. Figure 11 again demonstrates the histogram for the same query, but with a different table.

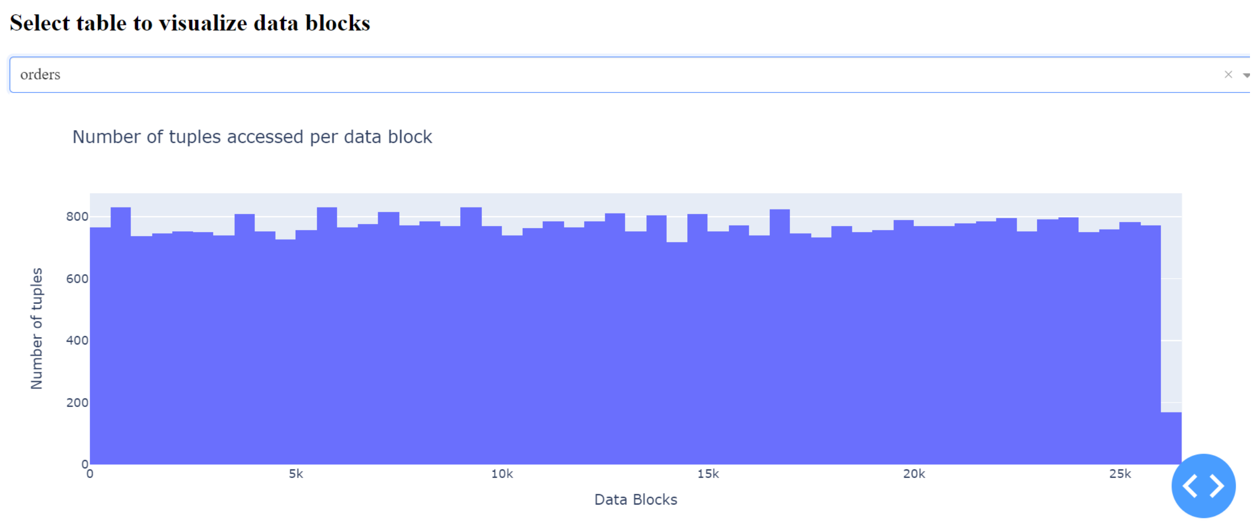

Select table to visualize data blocks
orders
Number of tuples accessed per data block
Number of tuples
Data Blocks
0
200
400
600
800
0
5k
10k
15k
20k
25k

Figure 11: Another example histogram on the same query but on a different table