NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

# CZ4031 Database System Principles

Project 2

**Submitted by: Group 12**

| | |
|---|---|
| Grace Ting Yan Teng | (U1522386K) |
| Jeremy Thia Ming Xuan | (U1620928G) |
| Lee Pei Shan | (U1520986E) |
| Marathe Ajinkya Avinash | (U1522716K) |

School of Computer Science and Engineering

2018

# Table of Contents

## Introduction

The aim of this project is to build a tool that enables interactive visual exploration of Query Execution Plans (QEP) for SQL queries. We will design and implement an algorithm that takes as input a SQL query, and its execution plan, and correlate it using colour codes and visual exploration. There will be a Graphical User Interface (GUI) for the entire application.

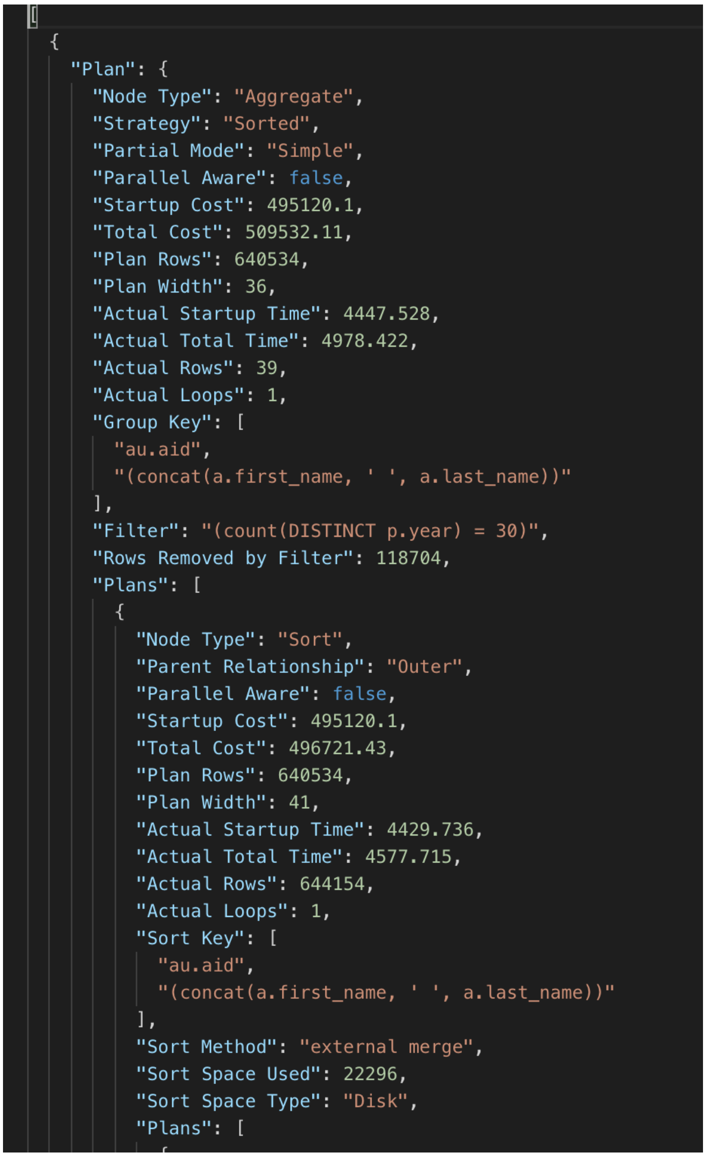
### Tools

The Python language will be used for this application:

- ***psycopg2*** library will be used to connect the application to the PostgreSQL database
- ***json*** library will be used to process the json file which PostgreSQL will output

# Algorithm

To generate the QEP for the specified query, we will make use of the ‘EXPLAIN’ function in PostgreSQL. PostgreSQL devises a query plan for each query it receives. The ‘EXPLAIN’ function will return the query plan which the planner creates for a query. The arguments for this function allows viewing of detailed information about the query plan as well as choosing the output format. Our choice of output is the json format. We added the following code ‘EXPLAIN (ANALYZE, FORMAT JSON)’ to the start of our query before sending it to the PostgreSQL database for processing. This will return a detailed analysis of the query plan including execution time.



```
[
  {
    "Plan": {
      "Node Type": "Aggregate",
      "Strategy": "Sorted",
      "Partial Mode": "Simple",
      "Parallel Aware": false,
      "Startup Cost": 495120.1,
      "Total Cost": 509532.11,
      "Plan Rows": 640534,
      "Plan Width": 36,
      "Actual Startup Time": 4447.528,
      "Actual Total Time": 4978.422,
      "Actual Rows": 39,
      "Actual Loops": 1,
      "Group Key": [
        "au.aid",
        "(concat(a.first_name, ' ', a.last_name))"
      ],
      "Filter": "(count(DISTINCT p.year) = 30)",
      "Rows Removed by Filter": 118704,
      "Plans": [
        {
          "Node Type": "Sort",
          "Parent Relationship": "Outer",
          "Parallel Aware": false,
          "Startup Cost": 495120.1,
          "Total Cost": 496721.43,
          "Plan Rows": 640534,
          "Plan Width": 41,
          "Actual Startup Time": 4429.736,
          "Actual Total Time": 4577.715,
          "Actual Rows": 644154,
          "Actual Loops": 1,
          "Sort Key": [
            "au.aid",
            "(concat(a.first_name, ' ', a.last_name))"
          ],
          "Sort Method": "external merge",
          "Sort Space Used": 22296,
          "Sort Space Type": "Disk",
          "Plans": [
            {
```

Figure 1: Snippet of Output by PostgreSQL

From the json output, we parse it into its individual json objects and arrays. The different arrays show the individual nodes which relate to a specific action that the query plan is carrying out. Next, we use the individual json objects and arrays to form a tree-like structure to show how the plan works, indicated with arrows. If a node has multiple arrows pointing to it, it means that there are two inputs to this action.

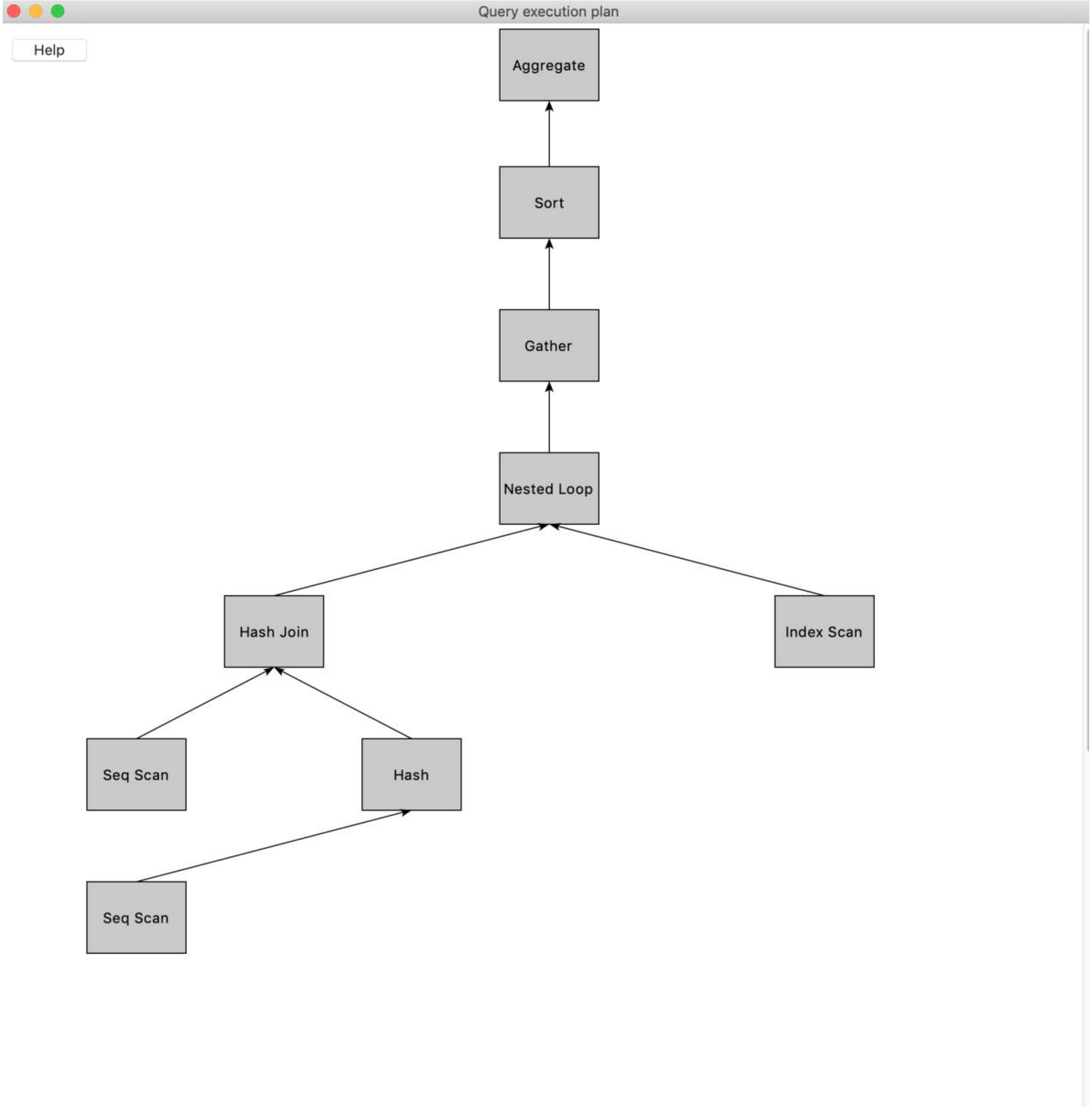


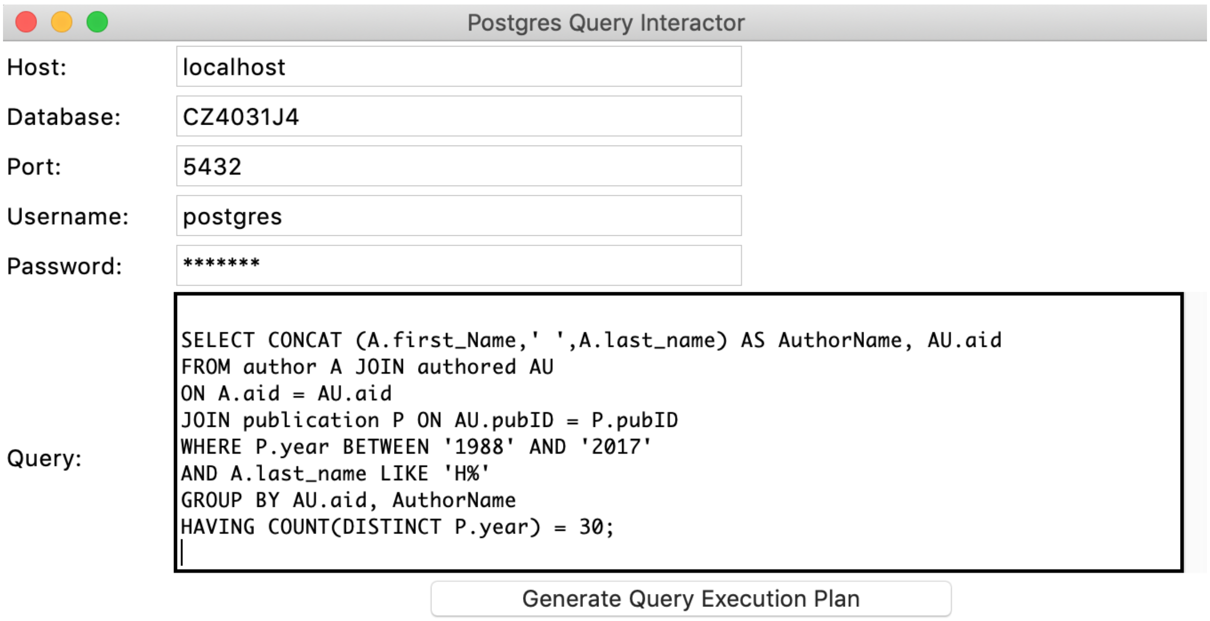
Figure 2: Sample of tree-structured Query Plan

# Graphical User Interface

There are two interfaces used in this application namely the Query Interface and Graphical View Interface.

## Query Interface

The first interface is for the user to input details for their database as well as input their queries in. The figure below shows the first interface of our application.



Host: localhost

Database: CZ4031J4

Port: 5432

Username: postgres

Password: *******

Query:

```
SELECT CONCAT (A.first_Name,' ',A.last_name) AS AuthorName, AU.aid
FROM author A JOIN authored AU
ON A.aid = AU.aid
JOIN publication P ON AU.pubID = P.pubID
WHERE P.year BETWEEN '1988' AND '2017'
AND A.last_name LIKE 'H%'
GROUP BY AU.aid, AuthorName
HAVING COUNT(DISTINCT P.year) = 30;
```

Figure 3: First Interface- Query Interface

The information for the database can be preconfigured in the 'config.json' file located in the folder. Alternatively, if the user wishes to change the database on-the-fly, they can choose to change it in the fields above. The password column is hidden to protect the privacy of users. The query box can be used to enter the query which they wish to view the query plan for. Once all fields have been filled, the user can go ahead and click on the '*Generate Query Execution Plan*' button which will bring us to the interactive view of the plan, this is our second interface.

## Graphical View Interface

Our second interface is for users to interact and view more information about the query plan. The interface will consist of the graphical view of the query plan, interaction through hovering and a help page.

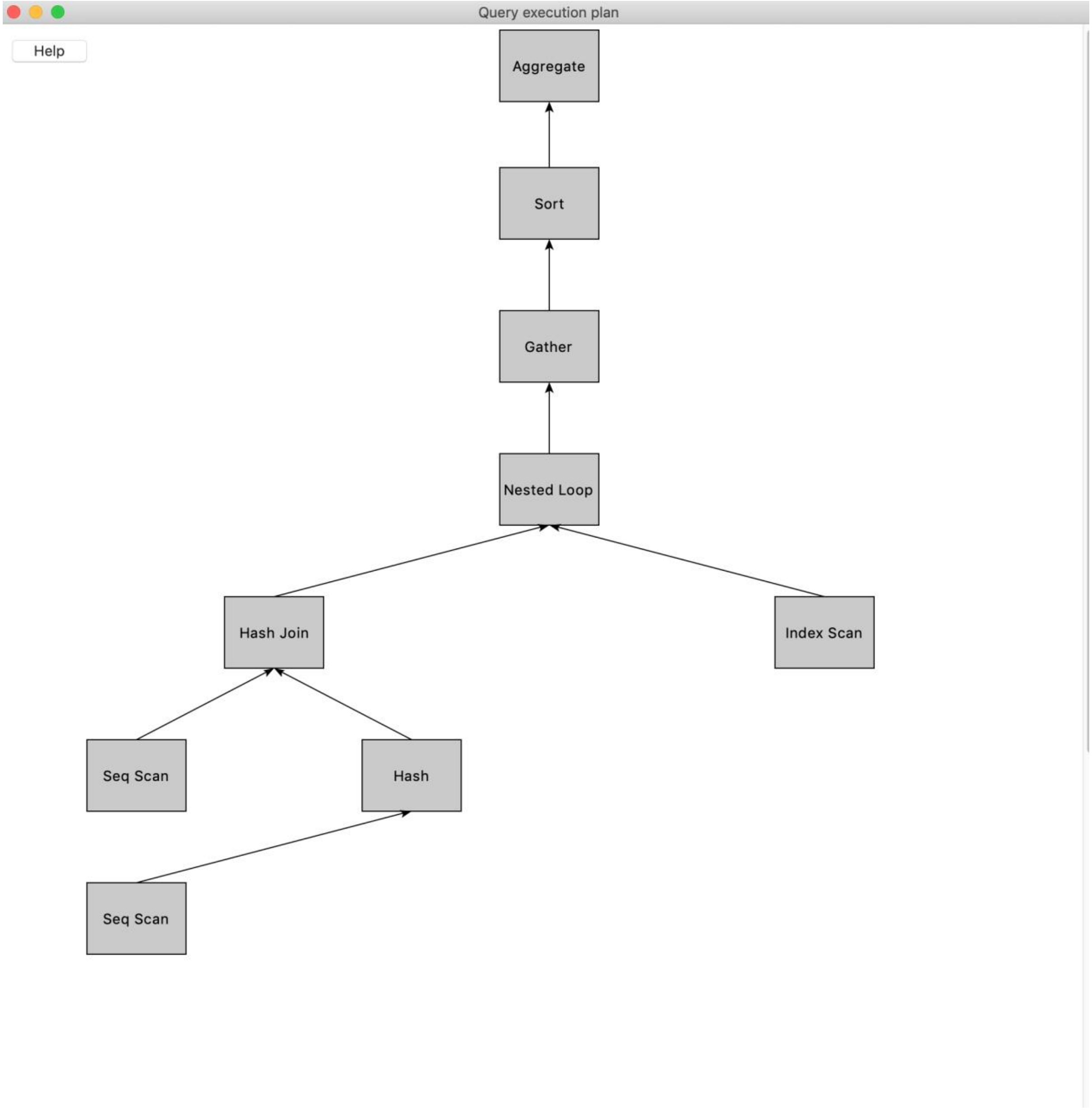
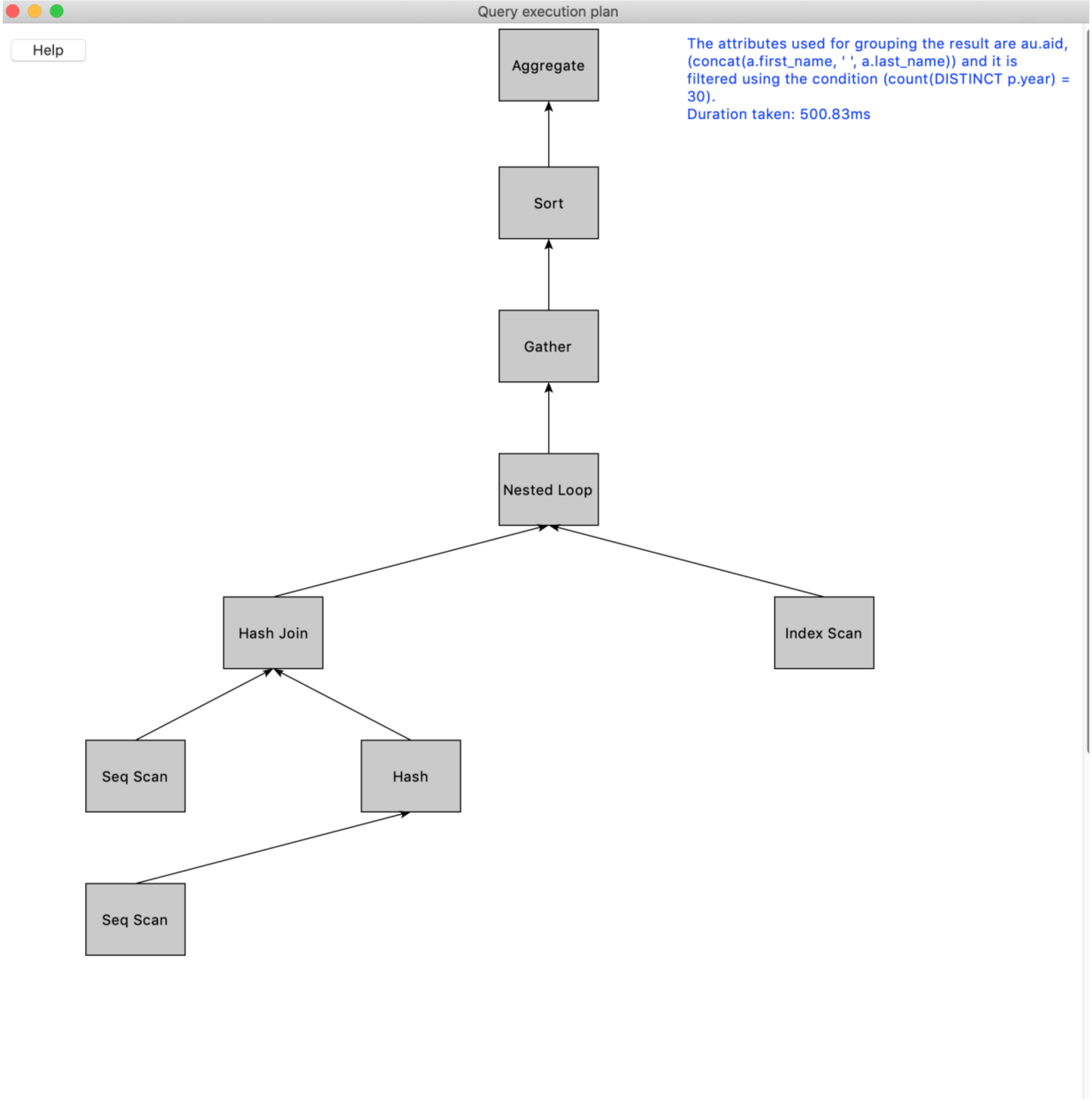


Figure 4: Second Interface- Graphical Plan Interface

The different rectangles indicate an action that is taken by the database to produce the query result. This diagram is read bottom-up, as the arrows direction indicates. The first action carried out by the database is from the bottom (the leaf node).

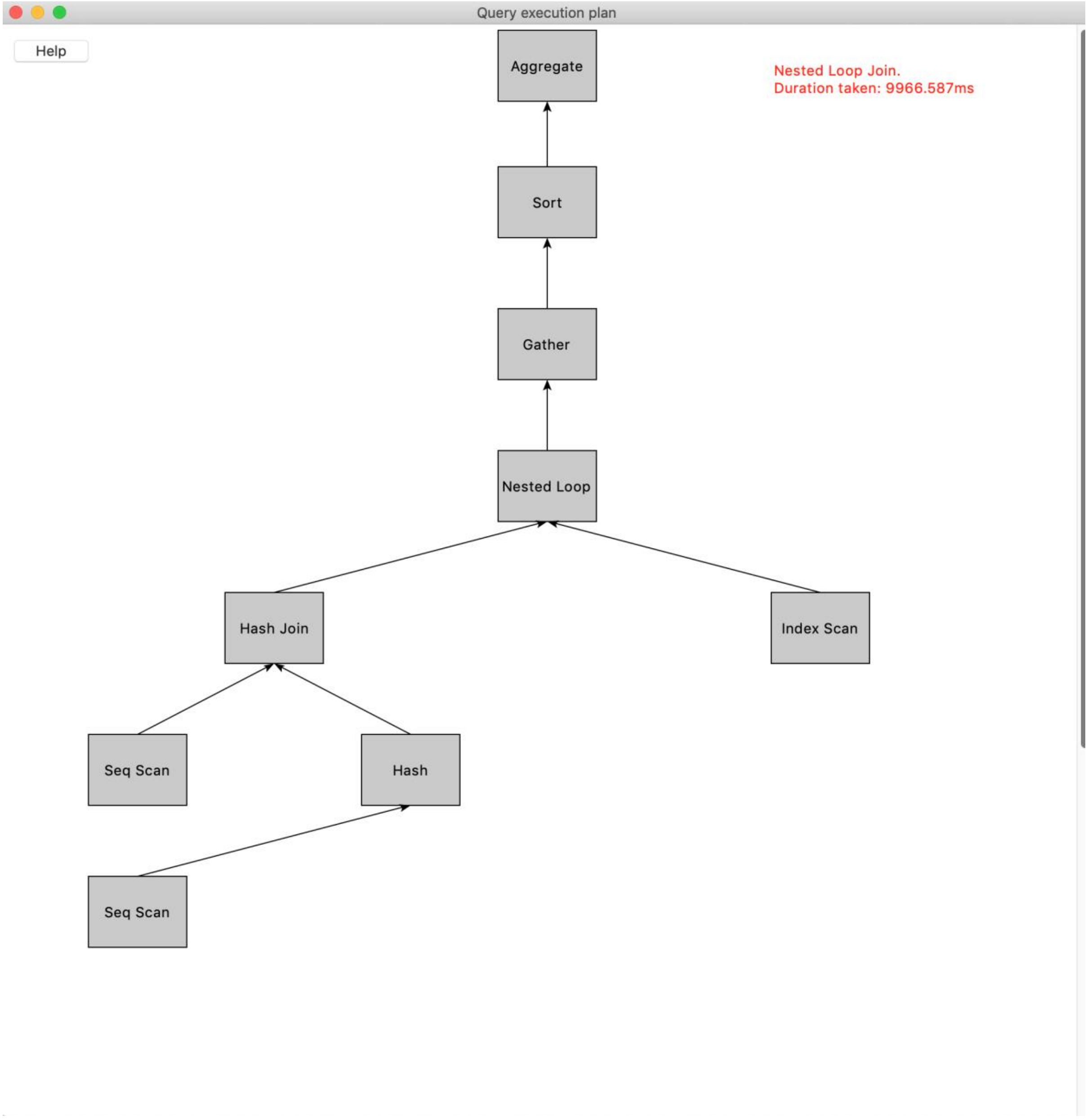
By hovering over a node, the graph will display information about its action at the top right-hand corner of the interface. The example below displays the information when the mouse was hovered across the ‘Aggregate’ node at the top of the interface. It shows information depending on the node type such search conditions and which table it searches on. The duration taken is also displayed in the same area.



The attributes used for grouping the result are au.aid, (concat(a.first_name, ' ', a.last_name)) and it is filtered using the condition (count(DISTINCT p.year) = 30).
Duration taken: 500.83ms

Figure 5: Screengrab when mouse hovers over *Aggregate* node

One feature of our graph is the ability to filter out which action takes the longest duration. The node which takes the longest duration will have its text information highlighted in red text. The example below displays the 'Nested Loop' as the node which has the longest duration.

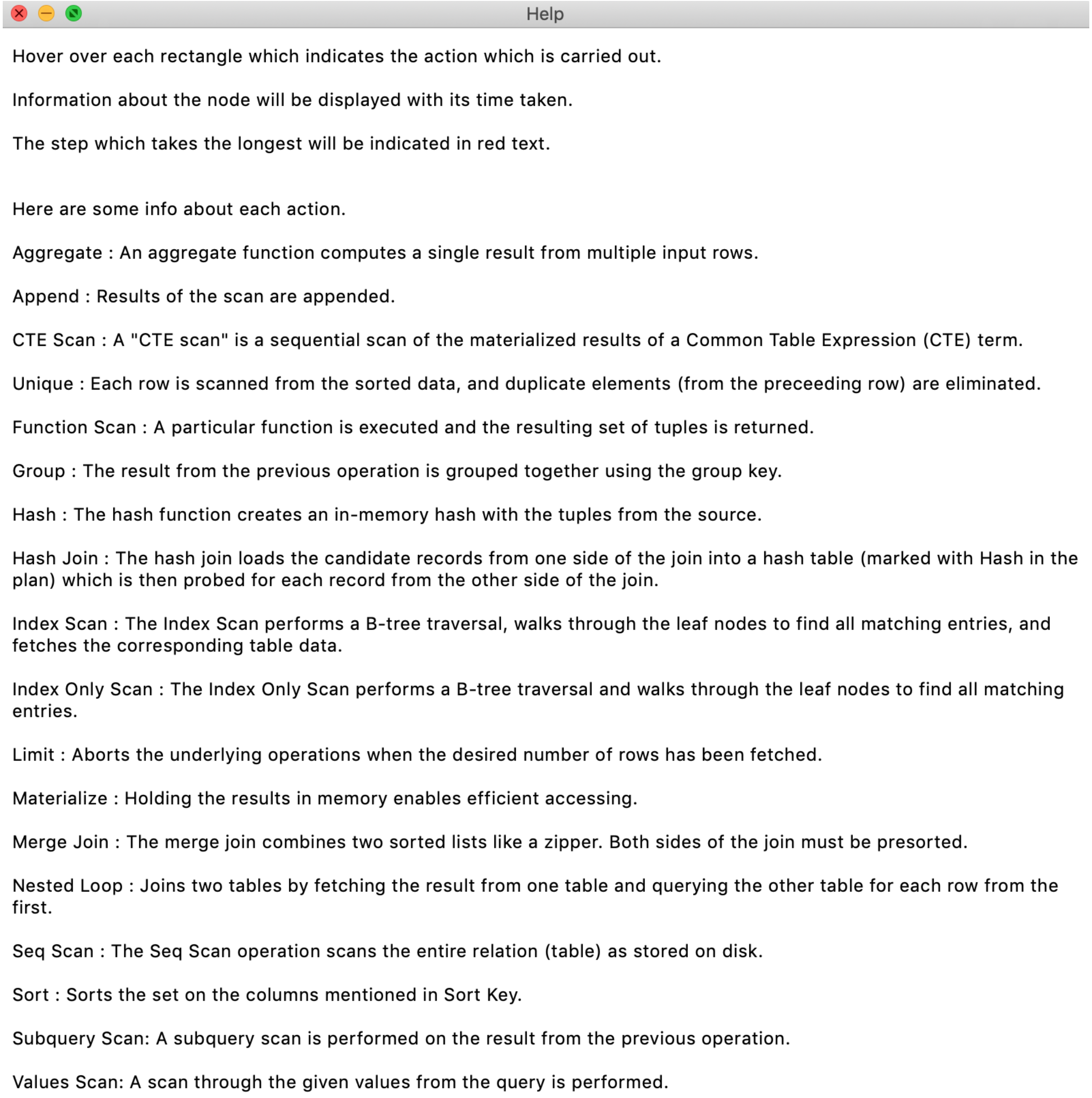


Nested Loop Join.
Duration taken: 9966.587ms

Figure 6: Screengrab showing details of step with Longest Duration

## Help Page

There is a help button located at the top left-hand corner of the application which shows information about how to use the graphical interface and provide a brief description about the various nodes displayed.



Hover over each rectangle which indicates the action which is carried out.

Information about the node will be displayed with its time taken.

The step which takes the longest will be indicated in red text.

Here are some info about each action.

Aggregate : An aggregate function computes a single result from multiple input rows.

Append : Results of the scan are appended.

CTE Scan : A "CTE scan" is a sequential scan of the materialized results of a Common Table Expression (CTE) term.

Unique : Each row is scanned from the sorted data, and duplicate elements (from the preceeding row) are eliminated.

Function Scan : A particular function is executed and the resulting set of tuples is returned.

Group : The result from the previous operation is grouped together using the group key.

Hash : The hash function creates an in-memory hash with the tuples from the source.

Hash Join : The hash join loads the candidate records from one side of the join into a hash table (marked with Hash in the plan) which is then probed for each record from the other side of the join.

Index Scan : The Index Scan performs a B-tree traversal, walks through the leaf nodes to find all matching entries, and fetches the corresponding table data.

Index Only Scan : The Index Only Scan performs a B-tree traversal and walks through the leaf nodes to find all matching entries.

Limit : Aborts the underlying operations when the desired number of rows has been fetched.

Materialize : Holding the results in memory enables efficient accessing.

Merge Join : The merge join combines two sorted lists like a zipper. Both sides of the join must be presorted.

Nested Loop : Joins two tables by fetching the result from one table and querying the other table for each row from the first.

Seq Scan : The Seq Scan operation scans the entire relation (table) as stored on disk.

Sort : Sorts the set on the columns mentioned in Sort Key.

Subquery Scan: A subquery scan is performed on the result from the previous operation.

Values Scan: A scan through the given values from the query is performed.

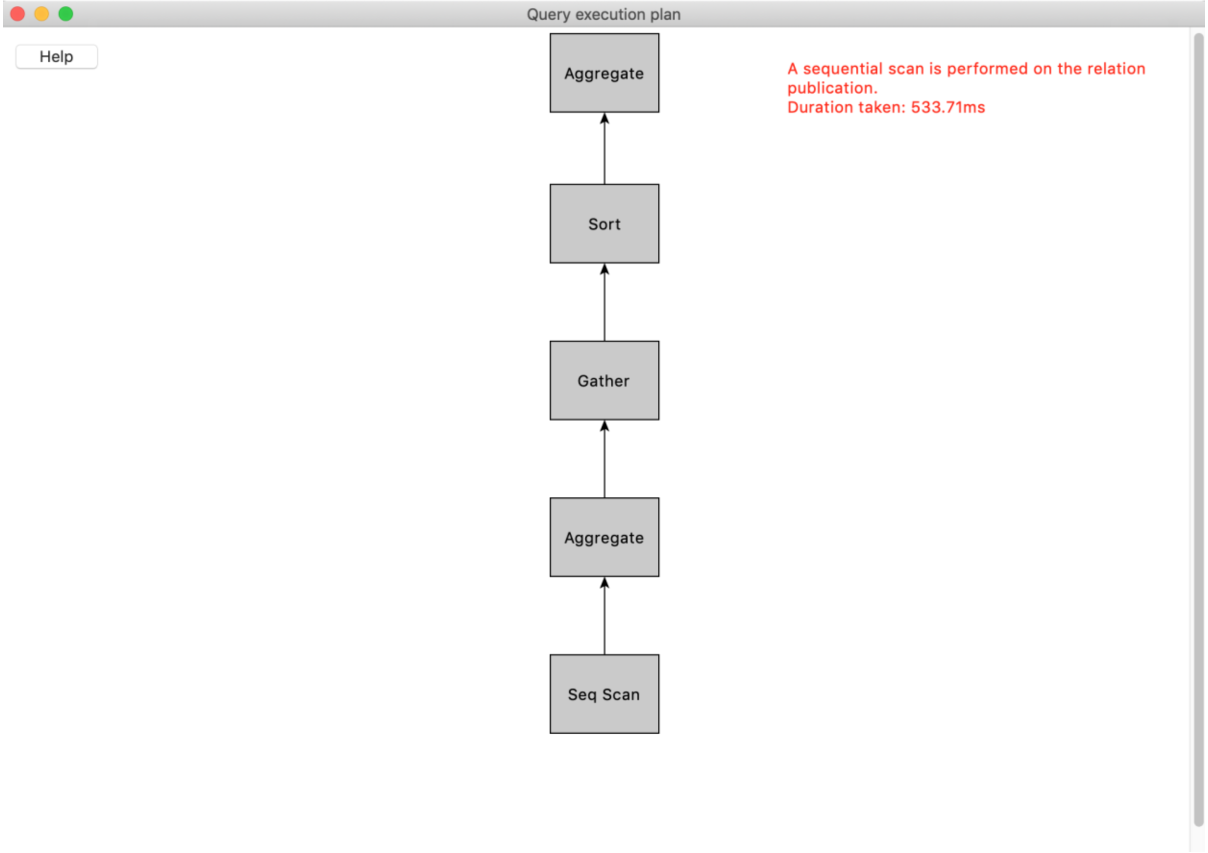
Figure 7: Help Page

# Testing

To test our application and ensure it works with a range of queries, we tried 4 queries from our first assignment which is based on the dblp database.

## Query 1

```
SELECT pub_type, count(pub_Type) AS num
FROM publication
GROUP BY pub_type
```

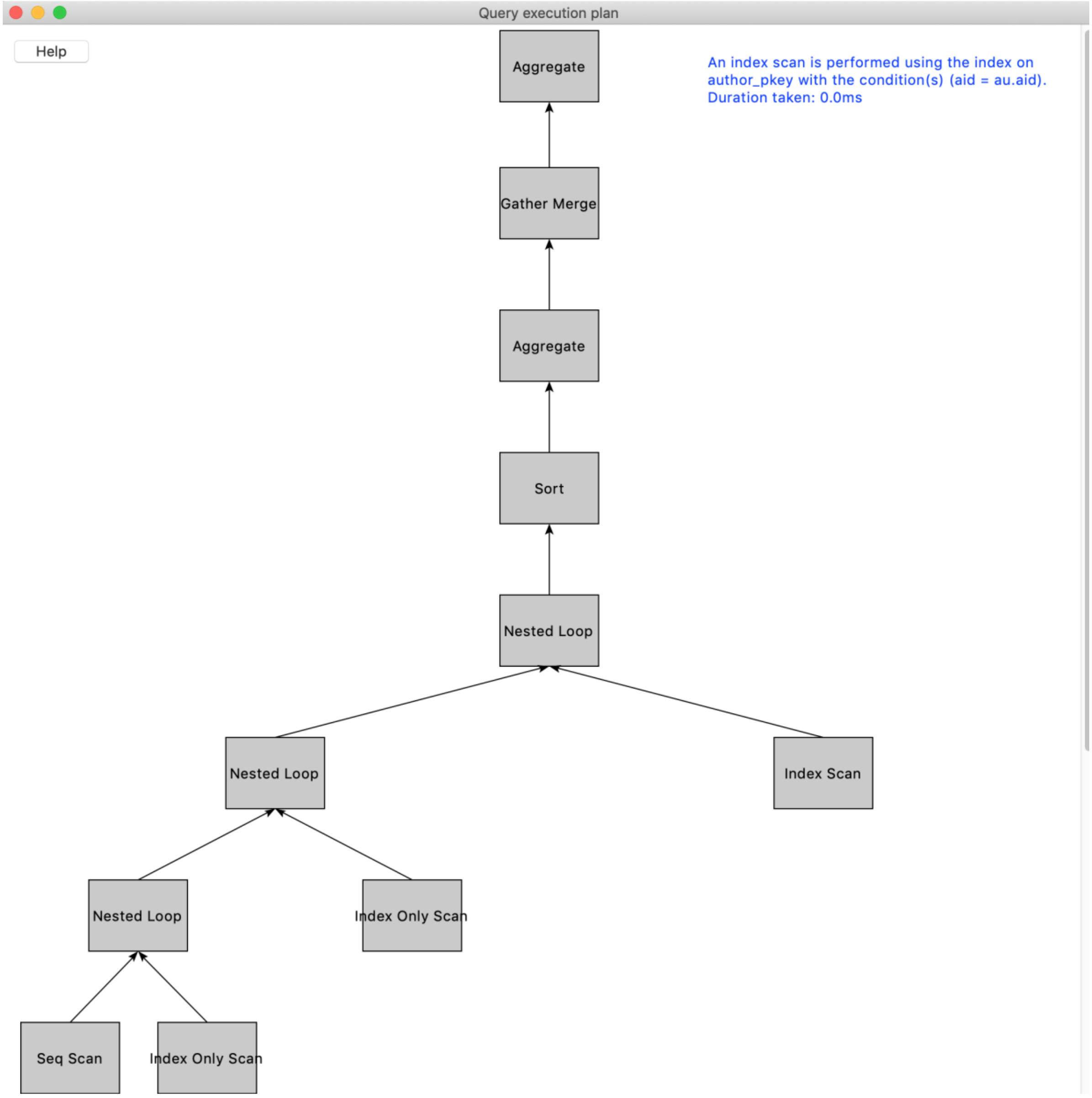


A sequential scan is performed on the relation publication.
Duration taken: 533.71ms

Figure 8: QEP of Query 1

## Query 2

```
SELECT A.*, COUNT(P.pubID) AS count
FROM author A
INNER JOIN authored AU
ON A.aid = AU.aid
INNER JOIN publication P
ON AU.pubID = P.pubID
INNER JOIN author B
ON B.aid= AU.aid
WHERE P.YEAR = '2015' AND P.pubkey LIKE '%conf/amcc%'
GROUP BY A.aid
HAVING COUNT(P.pubID) >=2
```

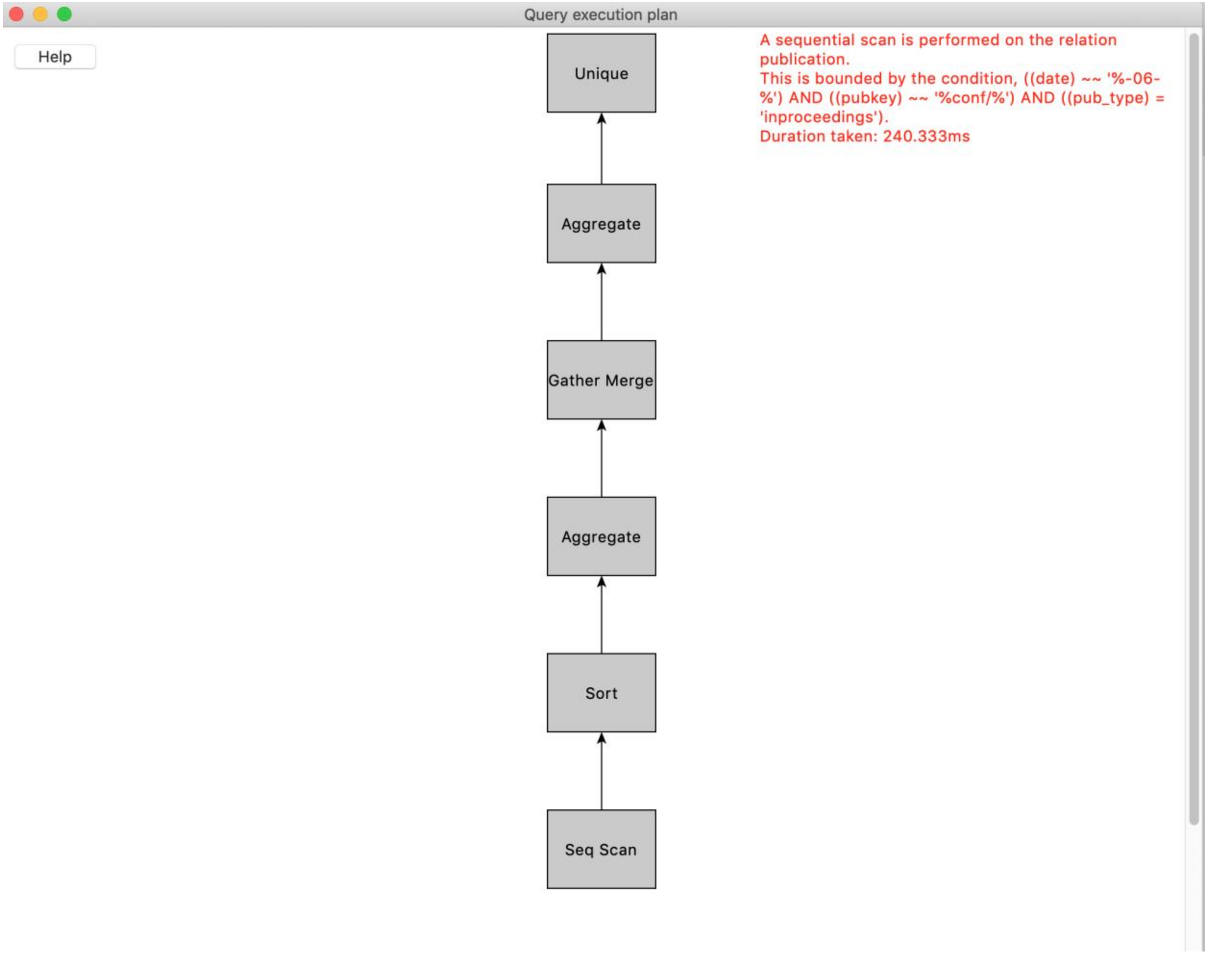


An index scan is performed using the index on author_pkey with the condition(s) (aid = au.aid). Duration taken: 0.0ms

Figure 9: QEP of Query 2

## Query 3

```
SELECT DISTINCT conf_name
FROM publication
WHERE date LIKE '%-06-%'
AND pub_type = 'inproceedings'
AND pubkey LIKE '%conf/%'
GROUP BY conf_name, year
HAVING COUNT(*) > 100;
```

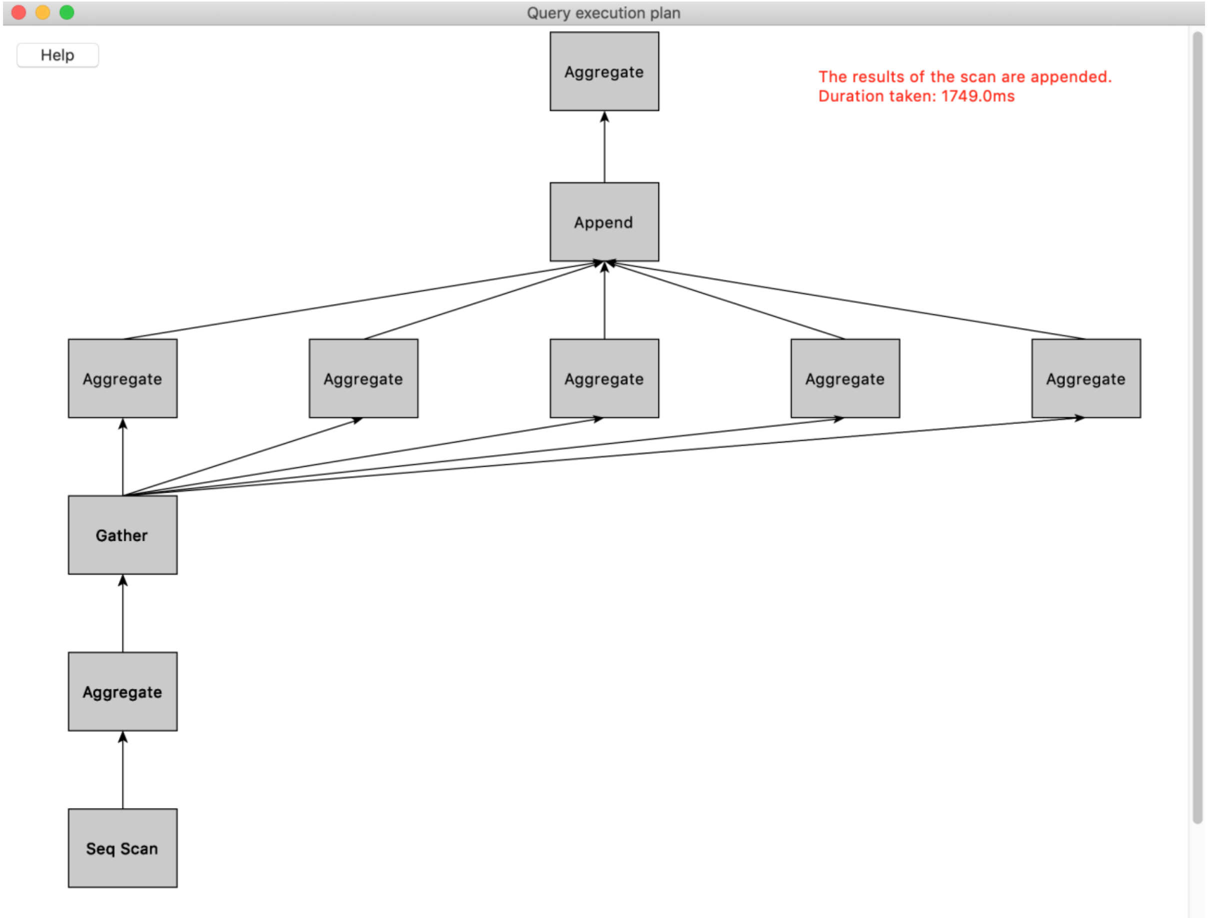


A sequential scan is performed on the relation publication.
This is bounded by the condition, ((date) ~~ '%-06-%') AND ((pubkey) ~~ '%conf/%') AND ((pub_type) = 'inproceedings').
Duration taken: 240.333ms

Figure 10: QEP of Query 3

## Query 4

```
(SELECT '1970-1979' AS decade, count(*) AS num FROM "1970_1979")
UNION
(SELECT '1980-1989' AS decade, count(*) AS num FROM "1980_1989")
UNION
(SELECT '1990-1999' AS decade, count(*) AS num FROM "1990_1999")
 UNION
(SELECT '2000-2009' AS decade, count(*) AS num FROM "2000_2009")
UNION
(SELECT '2010-2019' AS decade, count(*) AS num FROM "2010_2019")
```



The results of the scan are appended.
Duration taken: 1749.0ms

Figure 11: QEP of Query 4