Atomic Spectroscopy

Article

Analysis of Tools for Measuring PostgreSQL Query Cost

Raghavi Shirur[1], Vedang Joshi[2]
[1,2] Department of Computer Engineering, Institute of Engineering & Technology, Devi Ahilya Vishvavidyalaya, Indore, Madhya Pradesh, India. 452017

ABSTRACT: Amidst the realm of database management, the precise prediction of query execution time holds paramount importance in the pursuit of performance optimization. This intricate endeavor hinges upon intricate cost models yet is full of challenges owing to the intricacies of selectivity estimation and the propensity for cost-modeling errors. In the domain of open-source databases, PostgreSQL emerges as a standout contender, distinguished by its sophisticated cost models. Delving into the crux of this matter, the present paper undertakes a comprehensive analysis utilizing a spectrum of tools such as EXPLAIN, EXPLAIN ANALYZE, PgAdmin, Dalibo, and Depesz. Through these lenses, the challenges aforementioned are meticulously dissected, revealing the nuanced landscape of query execution. Notably, PostgreSQL's trajectory in addressing these challenges is meticulously showcased, illuminating its evolutionary journey. The insights thus garnered extend their value to the intricacies of query optimization, where the judicious selection of query cost tools plays a pivotal role. By bridging theory and practice, this exposition contributes to refining database systems, equipping practitioners and researchers with invaluable knowledge for enhancing efficiency and proficiency.

Keywords: PostgreSQL, Database query, EXPLAIN, EXPLAIN ANALYSE, PgAdmin, Dalibo, Depesz.

Ethics Declaration
The authors declare no competing interests that could be perceived as influencing the research, analysis, or interpretation of the findings presented in this paper. This work was conducted with utmost integrity and objectivity, and there are no affiliations, or financial arrangements that might have biased the outcomes or the reporting of the results. The authors have complied with the guidelines provided in "Competing Interests and Funding" to ensure transparency and to provide readers with a clear understanding of potential influences on the research presented herein.

Data Availability Statement
All data generated or analyzed during this study are included in this published article.

INTRODUCTION

The intricate operational dynamics underlying the PostgreSQL query execution mechanism [13] exhibit complexity; nonetheless, an exhaustive comprehension thereof assumes pivotal significance in harnessing the complete potential of the database. Every executed query entails a meticulously orchestrated determination of a detail-oriented charted plan, denoted as the Query Plan, which in turn empowers PostgreSQL to adeptly identify the sought-after values. This iterative process of enhancement is denoted as the Query Cost optimization, thereby ensconcing the assurance of paramount efficiency and optimal performance.

The basic concepts for PostgreSQL query cost measurement [1] are explained in the following section.

Plan Structure: A query plan is structured as a tree composed of nodes. Each node possesses a specific type and can have multiple child nodes depending on its type. Although each node type may exhibit different behaviors, the fundamental mechanism remains consistent: a parent node retrieves data from its child nodes row by row. The child nodes can either generate data directly, such as by reading from tables, or retrieve data from their own child nodes.

The need for Query Cost [13] is to optimize our database's performance in terms of CPU consumption and disk I/O, it is essential to consider the metric of query cost. The query cost provides valuable information on the efficiency of our database operations. This cost is primarily influenced by the query planning process, which involves determining the most effective arrangement of plan nodes to execute the query. By carefully managing the query planning stage, we can achieve better performance [1] and maximize the capabilities of our database.
Query Planning: Planning [2] determines the best arrangement of plan nodes for executing our query and depends on several factors.

- The intended meaning of our query: The query plan [3] must accurately retrieve the specific rows requested by our query. However, it's important to note that if multiple valid results fulfill the query's semantics, we may receive different rows each time. This is why we cannot rely on a simple "SELECT * FROM <table_name> LIMIT 1" without specifying an "ORDER BY" clause.
- The availability of indexes: Indexes can significantly enhance the efficiency of data retrieval in Postgres. To optimize performance, it is crucial to understand query execution and maintain appropriate indexes for the schema and query workload.
- The current configuration settings: The planner cost constants and resource consumption settings can be dynamically configured on a per-database, per-user, per-session, or even per-query basis. Understanding and adjusting these settings appropriately can impact query execution and overall performance.
- The data statistics gathered by Postgres: Postgres collects statistics about our data through manual execution of the ANALYZE command or automatic vacuum processes. These statistics play a vital role in query optimization by providing insights into the data distribution and allowing the planner to make informed decisions during query execution.

To develop a plan, Postgres inspects our query and evaluates based on the alternatives given above. It estimates [14] a cost for each leaf node and propagates those costs up the tree to calculate costs for internal nodes and eventually the root. It considers several plans and picks the cheapest based on cost settings, available indexes, and heuristics based on collected database statistics. We'll look at some tools that will enable us to have a clear idea of measuring query performance.

ANALYSIS OF TOOLS

The present paper throws light on the comprehensive evaluation of five distinct tools:

- EXPLAIN
- EXPLAIN ANALYSE
- PgAdmin
- Dalibo
- Depesz

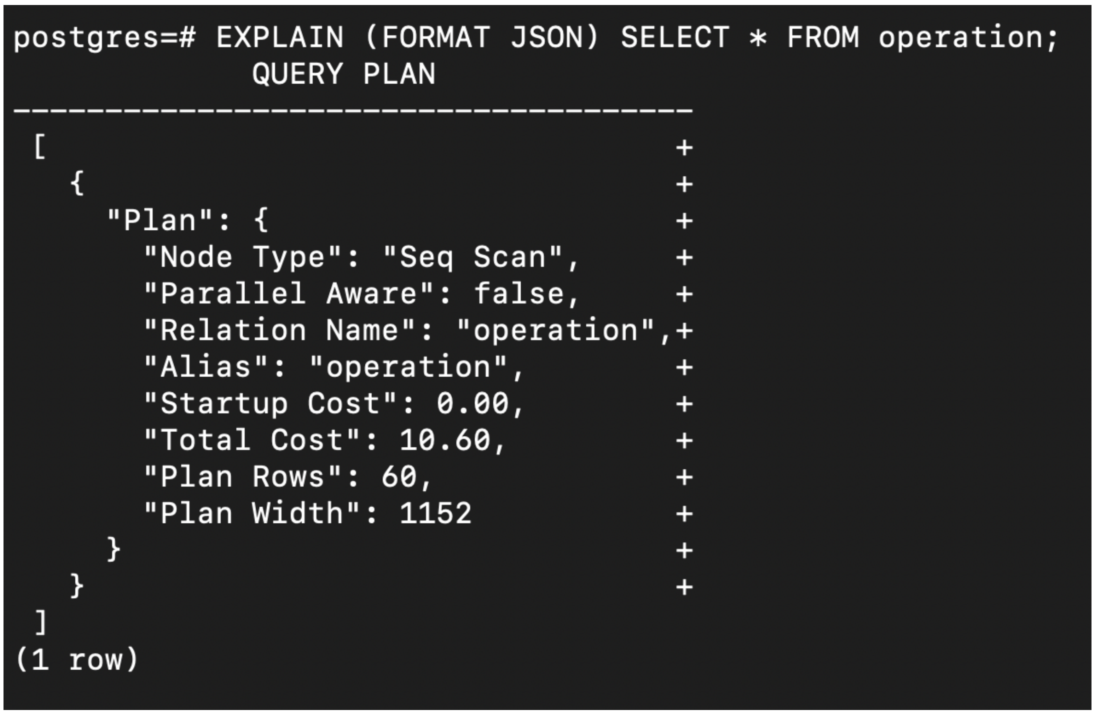
Every tool undergoes a thorough and methodical exposition, scrutiny, and evaluation, meticulously adhering to a systematic, step-by-step methodology, complemented by illustrative instances. The foremost intent of this undertaking is to impart a profound comprehension of the potentialities and constraints inherent in each tool. This equips readers with the requisite knowledge to prudently deliberate when opting for the most pertinent tool tailored to their distinct requisites. Via this exhaustive examination, our objective is to elucidate the distinctive attributes and operational capacities inherent in each tool, thereby endowing users with the acumen to execute judicious and effective selections aligned with their undertakings or obligations.

EXPLAIN

PostgreSQL provides a comprehensive mechanism to examine query planning and associated costs. The EXPLAIN [4] command offers valuable benefits by presenting query costs and insights in a metric format. By expressing your desired outcome to the database server, it takes on the responsibility of determining the most efficient approach to deliver the requested results. In PostgreSQL, the planner utilizes the query structure, data properties, and various other factors to generate a well-optimized plan for query execution.

This is demonstrated with a couple of examples. To use the EXPLAIN command, you tack on EXPLAIN before the statement you want to run. This will return the estimated plan and cost, in plain text by default. We'll run the following queries and demystify all the sub-components of the returned output.

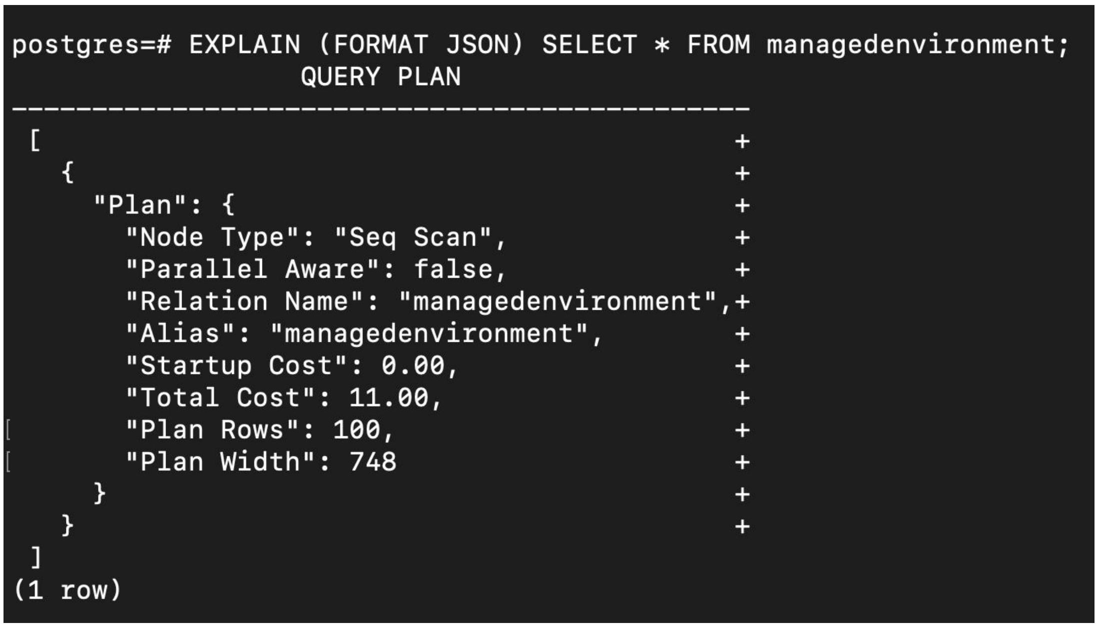
```
EXPLAIN (FORMAT JSON) SELECT * FROM operation;
```



```
postgres=# EXPLAIN (FORMAT JSON) SELECT * FROM operation;
             QUERY PLAN
------------------------------------
 [                                 +
   {                               +
     "Plan": {                     +
       "Node Type": "Seq Scan",    +
       "Parallel Aware": false,    +
       "Relation Name": "operation",+
       "Alias": "operation",       +
       "Startup Cost": 0.00,       +
       "Total Cost": 10.60,        +
       "Plan Rows": 60,            +
       "Plan Width": 1152          +
     }                             +
   }                               +
 ]
(1 row)
```

Fig. 2.1.1 Postrgresql query output for the above command for operation table

```
EXPLAIN (FORMAT JSON) SELECT * FROM managedenvironment;
```



```
postgres=# EXPLAIN (FORMAT JSON) SELECT * FROM managedenvironment;
                  QUERY PLAN
---------------------------------------------
 [                                          +
   {                                        +
     "Plan": {                              +
       "Node Type": "Seq Scan",             +
       "Parallel Aware": false,             +
       "Relation Name": "managedenvironment",+
       "Alias": "managedenvironment",       +
       "Startup Cost": 0.00,                +
       "Total Cost": 11.00,                 +
       "Plan Rows": 100,                    +
       "Plan Width": 748                    +
     }                                      +
   }                                        +
 ]
(1 row)
```

Fig. 2.1.2 Postrgresql query output for the above command in managedenvironment table

As previously mentioned, the query plan [5] structure consists of a tree composed of plan nodes. At the lowest level of the tree are scan nodes responsible for retrieving raw rows from a table. Different types of scans exist depending on the table access methods, including sequential scans, index scans, and bitmap index scans. Additionally, non-table row sources like VALUES clauses and set-returning functions in the FROM clause have their own scan node types. If the query involves operations such as joining, aggregation, sorting, or others on the raw rows, additional nodes above the scan nodes are added to perform these operations. Similar to the scan nodes, there can be multiple ways to execute these operations, resulting in different node types. The EXPLAIN output provides a line for each node in the plan tree, displaying the node's basic type and the planner's cost estimates for executing that particular node. Additional lines, indented from the node's summary line, may appear to provide additional properties of the node. The first line, representing the summary of the topmost node, displays the estimated total execution cost for the entire plan. This number serves as the planner's target to minimize when generating the optimal plan.

Due to the absence of a WHERE clause in this query, it is necessary to scan all rows of the table. As a result, the planner has opted for a straightforward sequential scan plan. The figures presented in parentheses provide the following information from left to right:

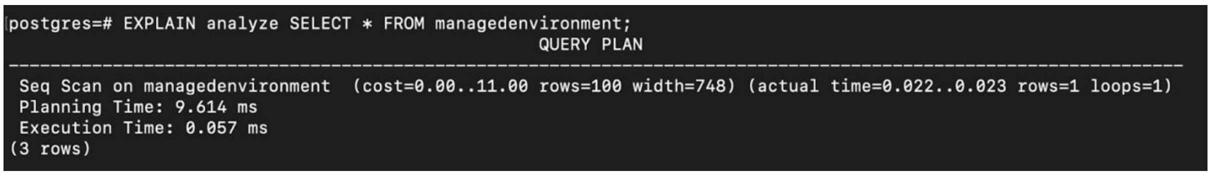
- Estimated start-up cost: This refers to the time required before the output phase can commence. For instance, it could involve the time needed for sorting in a sort node.
- Estimated total cost: This value is provided based on the assumption that the plan node will be executed in its entirety, meaning that all available rows will be retrieved. However, it is important to note that a node's parent node may choose to stop before reading all available rows, as exemplified in the case of a LIMIT clause.
- Estimated number of rows output by this plan node: Again, this estimation assumes that the node will be fully executed, resulting in the indicated number of output rows.
- The estimated average width of rows output by this plan node, measured in bytes: This metric represents the anticipated average size of the rows produced by the plan node.

The costs assigned to query nodes are measured in arbitrary units defined by the planner's cost parameters. Typically, these costs are based on disk page fetches, with seq_page_cost serving as the baseline unit (set to 1.0) and other parameters adjusted relative to it. When examining query costs, the startup cost indicates the expense incurred to initialize a node, while the total cost reflects the overall expense associated with that node. The specific cost values are influenced by various factors such as seq_page_cost, cpu_tuple_cost, and other relevant settings. It is worth noting that higher cost values indicate greater expense or complexity in executing the node. The examples provided in this context assume default cost parameter values, but these can be customized to reflect specific system configurations and requirements.

Understanding the cost of an upper-level node is crucial, as it encompasses the costs of all its child nodes. However, it's important to note that the cost calculation only considers factors that are relevant to the planner. Notably, it does not account for the time taken to transmit result rows to the client, which can impact the actual elapsed time. This omission is deliberate since the planner cannot alter this aspect by modifying the plan. It's worth emphasizing that every correct plan will yield the same set of rows as output.

The rows value requires some clarification as it does not represent the number of rows processed or scanned by the plan node. Instead, it indicates the number of rows emitted by the node. This value is often lower than the number scanned due to filtering based on WHERE-clause conditions applied at the node. Ideally, the estimate for the top-level rows should approximate the actual number of rows returned, updated, or deleted by the query.
Conclusion: The EXPLAIN command provides a concise and efficient means to obtain query cost information. It serves as an effective tool when seeking cost details without excessive visuals or unnecessary intricacies. While the command provides estimated costs based on statistics, it's important to acknowledge that the actual costs may differ in practice.

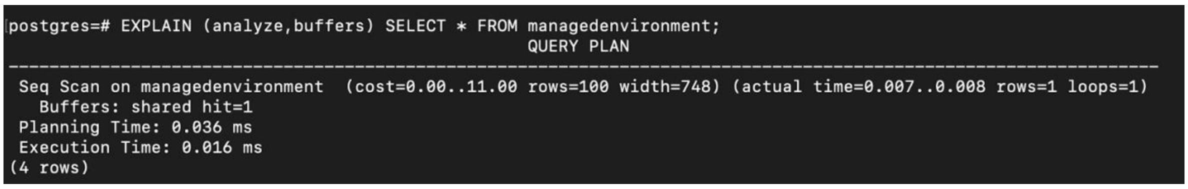
EXPLAIN ANALYZE
The EXPLAIN ANALYZE command offers an expanded version of the EXPLAIN command. It not only provides the estimated plan and statistics but also executes the query and presents the actual runtime statistics. By contrast, when using EXPLAIN alone, PostgreSQL does not execute the query but instead generates an estimated execution plan based on available statistics. Consequently, there can be significant variations between the estimated plan and the actual plan. Thankfully, PostgreSQL allows us to execute the query by utilizing EXPLAIN ANALYZE instead of EXPLAIN. The usage of EXPLAIN ANALYZE is similar to EXPLAIN. However, it's important to exercise caution when dealing with update or delete statements. In such cases, it might be advisable to avoid using ANALYZE or consider encapsulating the entire statement within a transaction that can be rolled back if needed. Below are some examples to understand its difference from EXPLAIN.



```
postgres=# EXPLAIN analyze SELECT * FROM managedenvironment;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Seq Scan on managedenvironment  (cost=0.00..11.00 rows=100 width=748) (actual time=0.022..0.023 rows=1 loops=1)
 Planning Time: 9.614 ms
 Execution Time: 0.057 ms
(3 rows)
```

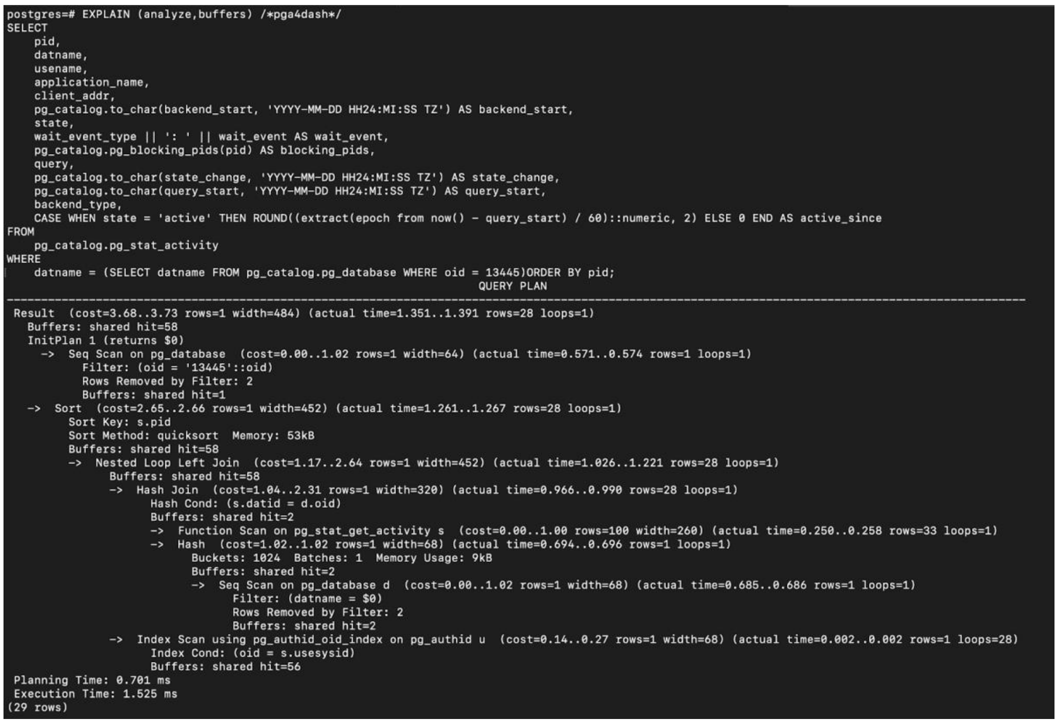
Fig. 2.2.1 EXPLAIN ANALYZE command output on managedenvironment table

We can add buffers to EXPLAIN ANALYZE, it will also give us information about the number of rows removed by a filter, the number of buffers used, and more:



```
postgres=# EXPLAIN (analyze,buffers) SELECT * FROM managedenvironment;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Seq Scan on managedenvironment  (cost=0.00..11.00 rows=100 width=748) (actual time=0.007..0.008 rows=1 loops=1)
   Buffers: shared hit=1
 Planning Time: 0.036 ms
 Execution Time: 0.016 ms
(4 rows)
```

Fig. 2.2.2 EXPLAIN ANALYZE command output with buffers option on managedenvironment table



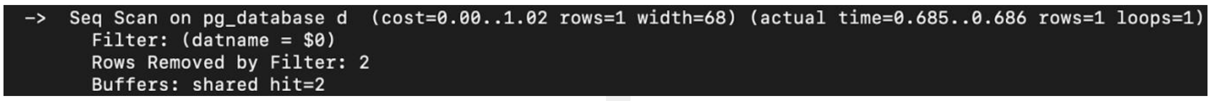
```
postgres=# EXPLAIN (analyze,buffers) /*pga4dash*/
SELECT
    pid,
    datname,
    usename,
    application_name,
    client_addr,
    pg_catalog.to_char(backend_start, 'YYYY-MM-DD HH24:MI:SS TZ') AS backend_start,
    state,
    wait_event_type || ': ' || wait_event AS wait_event,
    pg_catalog.pg_blocking_pids(pid) AS blocking_pids,
    query,
    pg_catalog.to_char(state_change, 'YYYY-MM-DD HH24:MI:SS TZ') AS state_change,
    pg_catalog.to_char(query_start, 'YYYY-MM-DD HH24:MI:SS TZ') AS query_start,
    backend_type,
    CASE WHEN state = 'active' THEN ROUND((extract(epoch from now() - query_start) / 60)::numeric, 2) ELSE 0 END AS active_since
FROM
    pg_catalog.pg_stat_activity
WHERE
    datname = (SELECT datname FROM pg_catalog.pg_database WHERE oid = 13445)ORDER BY pid;
                                                                  QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 Result  (cost=3.68..3.73 rows=1 width=484) (actual time=1.351..1.391 rows=28 loops=1)
   Buffers: shared hit=58
   InitPlan 1 (returns $0)
     ->  Seq Scan on pg_database  (cost=0.00..1.02 rows=1 width=64) (actual time=0.571..0.574 rows=1 loops=1)
           Filter: (oid = '13445'::oid)
           Rows Removed by Filter: 2
           Buffers: shared hit=1
   ->  Sort  (cost=2.65..2.66 rows=1 width=452) (actual time=1.261..1.267 rows=28 loops=1)
         Sort Key: s.pid
         Sort Method: quicksort  Memory: 53kB
         Buffers: shared hit=58
         ->  Nested Loop Left Join  (cost=1.17..2.64 rows=1 width=452) (actual time=1.026..1.221 rows=28 loops=1)
               Buffers: shared hit=58
               ->  Hash Join  (cost=1.04..2.31 rows=1 width=320) (actual time=0.966..0.990 rows=28 loops=1)
                     Hash Cond: (s.datid = d.oid)
                     Buffers: shared hit=2
                     ->  Function Scan on pg_stat_get_activity s  (cost=0.00..1.00 rows=100 width=260) (actual time=0.250..0.258 rows=33 loops=1)
                     ->  Hash  (cost=1.02..1.02 rows=1 width=68) (actual time=0.694..0.696 rows=1 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 9kB
                           Buffers: shared hit=2
                           ->  Seq Scan on pg_database d  (cost=0.00..1.02 rows=1 width=68) (actual time=0.685..0.686 rows=1 loops=1)
                                 Filter: (datname = $0)
                                 Rows Removed by Filter: 2
                                 Buffers: shared hit=2
               ->  Index Scan using pg_authid_oid_index on pg_authid u  (cost=0.14..0.27 rows=1 width=68) (actual time=0.002..0.002 rows=1 loops=28)
                     Index Cond: (oid = s.usesysid)
                     Buffers: shared hit=56
 Planning Time: 0.701 ms
 Execution Time: 1.525 ms
(29 rows)
```

Fig 2.2.3 An intricate and complex query output

Every query plan consists of nodes. Nodes can be nested, and are executed from the inside out. This means that the innermost node is executed before the outer node. This can be best thought of as nested function calls, returning their results as they unwind.

- The structure of the execution plan [5] is represented by indented statements denoted by an →, indicating child nodes of the outer/parent node. This hierarchical tree structure demonstrates the presence of multiple steps or levels in the execution plan.
- When utilizing the ANALYZE option, we gain insights into the time taken to generate the query plan and execute the query itself (excluding planning time).
- Estimated statistics are derived from the query plan and are presented within the first set of parentheses on each node line or summary. The topmost or root node provides overall totals, including those from its child nodes. Each child node or step has its own statistics, which may also include child nodes. The cost estimate does not have specific units and should not be interpreted in terms of time or any particular resources. Additionally, the plan provides an estimated count of rows that will be returned or match the given condition.

By utilizing ANALYZE, we obtain a second set of parentheses that provide actual run statistics. These statistics include the actual time spent (in milliseconds) within a specific node and the count of rows returned. Furthermore, loops can indicate if a particular node was executed multiple times.



```
 ->  Seq Scan on pg_database d  (cost=0.00..1.02 rows=1 width=68) (actual time=0.685..0.686 rows=1 loops=1)
       Filter: (datname = $0)
       Rows Removed by Filter: 2
       Buffers: shared hit=2
```

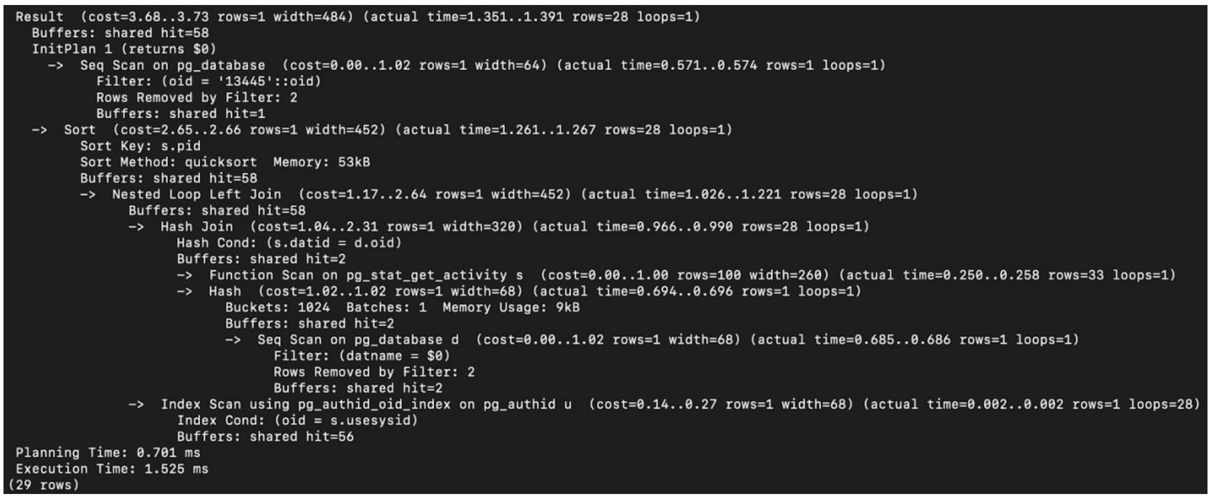
Fig 2.2.4 Nodes are indicated using a -> followed by the type of node taken

Here the first node executed is Seq scan on pg_database d. The Filter: is an additional filter applied to the results of the node.

The format of the costs field is as follows:

```
STARTUP COST..TOTAL COST
```

Each node in a plan has a set of associated statistics [Fig 2.2.5], such as the cost, the number of rows produced, the number of loops performed, and more. For example:

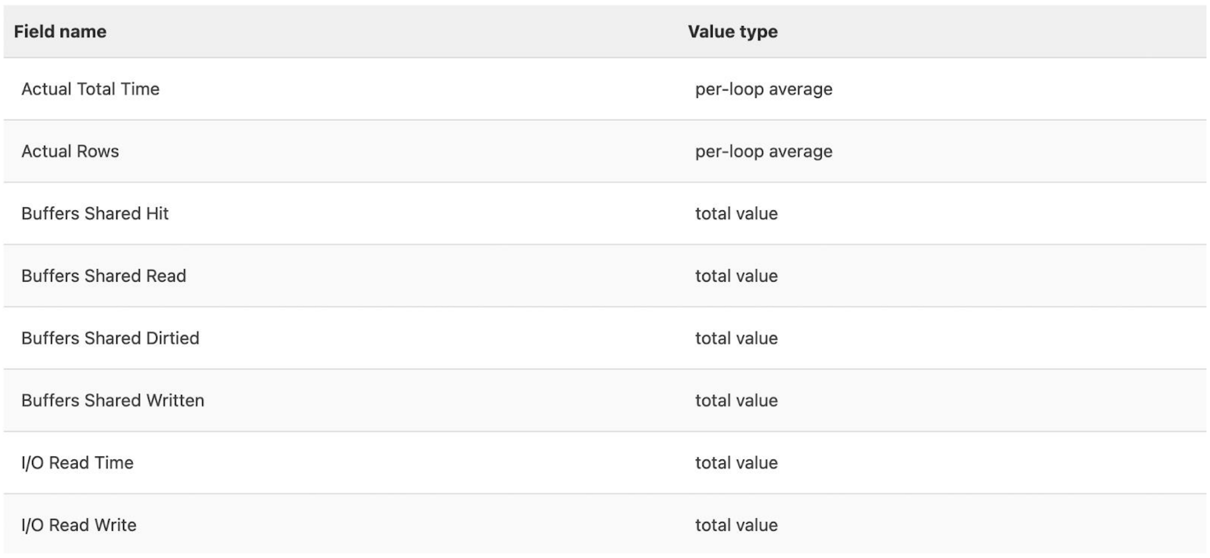


```
 Result  (cost=3.68..3.73 rows=1 width=484) (actual time=1.351..1.391 rows=28 loops=1)
   Buffers: shared hit=58
   InitPlan 1 (returns $0)
     ->  Seq Scan on pg_database  (cost=0.00..1.02 rows=1 width=64) (actual time=0.571..0.574 rows=1 loops=1)
           Filter: (oid = '13445'::oid)
           Rows Removed by Filter: 2
           Buffers: shared hit=1
   ->  Sort  (cost=2.65..2.66 rows=1 width=452) (actual time=1.261..1.267 rows=28 loops=1)
         Sort Key: s.pid
         Sort Method: quicksort  Memory: 53kB
         Buffers: shared hit=58
         ->  Nested Loop Left Join  (cost=1.17..2.64 rows=1 width=452) (actual time=1.026..1.221 rows=28 loops=1)
               Buffers: shared hit=58
               ->  Hash Join  (cost=1.04..2.31 rows=1 width=320) (actual time=0.966..0.990 rows=28 loops=1)
                     Hash Cond: (s.datid = d.oid)
                     Buffers: shared hit=2
                     ->  Function Scan on pg_stat_get_activity s  (cost=0.00..1.00 rows=100 width=260) (actual time=0.250..0.258 rows=33 loops=1)
                     ->  Hash  (cost=1.02..1.02 rows=1 width=68) (actual time=0.694..0.696 rows=1 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 9kB
                           Buffers: shared hit=2
                           ->  Seq Scan on pg_database d  (cost=0.00..1.02 rows=1 width=68) (actual time=0.685..0.686 rows=1 loops=1)
                                 Filter: (datname = $0)
                                 Rows Removed by Filter: 2
                                 Buffers: shared hit=2
               ->  Index Scan using pg_authid_oid_index on pg_authid u  (cost=0.14..0.27 rows=1 width=68) (actual time=0.002..0.002 rows=1 loops=28)
                     Index Cond: (oid = s.usesysid)
                     Buffers: shared hit=56
 Planning Time: 0.701 ms
 Execution Time: 1.525 ms
(29 rows)
```

Fig 2.2.5 Query execution to demonstrate associated statistics

When using EXPLAIN ANALYZE, these statistics will also include the actual time (in milliseconds) spent, and other runtime statistics (for example, the actual number of produced rows). Using EXPLAIN (ANALYZE, BUFFERS) will also give us information about the number of rows removed by a filter, the number of buffers used, and more.

It should be noted that some statistics are per-loop averages, while others are total values:



Field name	Value type
Actual Total Time	per-loop average
Actual Rows	per-loop average
Buffers Shared Hit	total value
Buffers Shared Read	total value
Buffers Shared Dirtied	total value
Buffers Shared Written	total value
I/O Read Time	total value
I/O Read Write	total value

Fig 2.2.6 Field Names depicting different Value Types

Conclusion: Explain Analyze provides detailed insights based on actual query execution, unlike the Explain command which relies on statistics. However, caution should be exercised when using this command as it executes the query itself. Explain Analyze also provides information about the involved nodes in query execution and allows for comparison between actual and estimated values, which can help identify unexpected behaviors.

PgAdmin

PgAdmin [8] holds a prominent position as a widely embraced graphical user interface (GUI) tool dedicated to the proficient administration of PostgreSQL databases. This tool stands distinguished for its comprehensive array of functionalities, effectively catering to all essential PostgreSQL operations. The tool's interactive dashboard serves as a centralized hub of information and functional tools, facilitating users in the seamless and efficient management of their databases. Within this holistic dashboard, a wealth of details and options are made available, enabling users to delve into the intricacies of their databases. Each component within PgAdmin contributes to a cohesive ecosystem of database oversight, where users can glean insights into the purpose of every facet. The tool's utility extends beyond its encompassing dashboard. PgAdmin empowers users by enabling a spectrum of tasks, ranging from fundamental database management to the nuanced execution of queries and the fine-tuning of server configurations. The software serves as a pivotal bridge between users and PostgreSQL databases, where actions are transformed into intuitive operations through a user-friendly interface.

In essence, PgAdmin stands as an indispensable tool in the PostgreSQL realm, harmonizing advanced functionality with a user-centric design philosophy. This graphical interface eliminates barriers, democratizing database management for both novice and experienced users. With its capability to accommodate a breadth of tasks, PgAdmin emerges as a valuable asset for those seeking streamlined and effective interaction with their PostgreSQL databases.

Graphs

- Server sessions or Database sessions: The *Server sessions* or *Database sessions* graph displays the connections to the server or database. It shows the total, active, and idle sessions for the server or database.

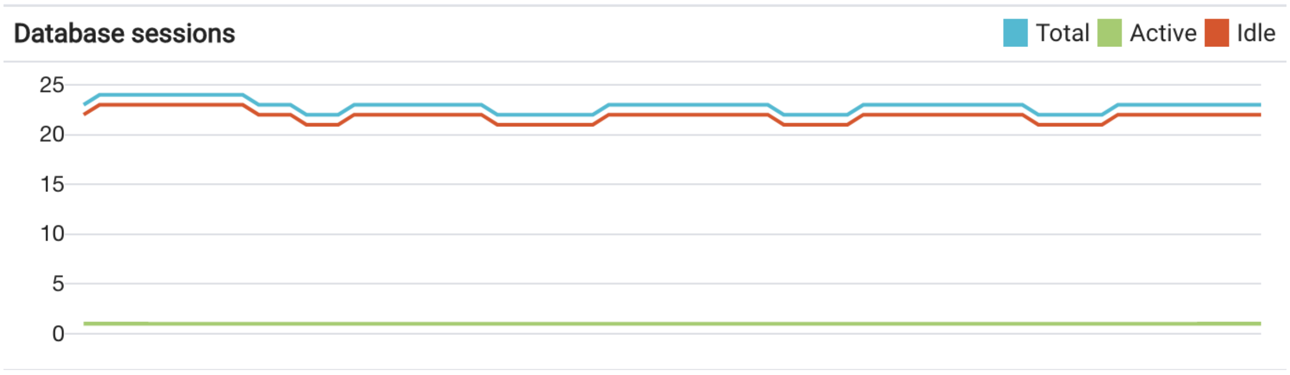


Fig. 2.3.1.1 Graphical representation of Database sessions

- Transactions per second: The *Transactions per second* graph shows the *commits*, rollbacks, and *total transactions* per second that are taking place on the server or database.

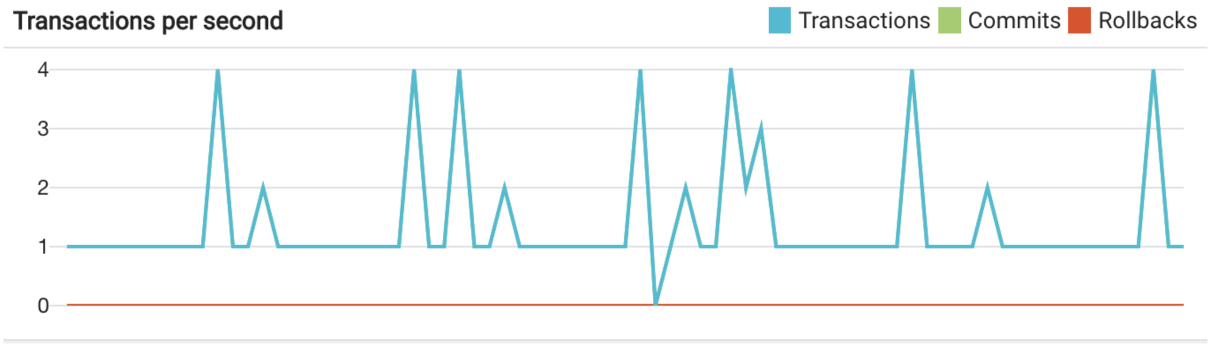


Fig. 2.3.1.2 Graphical representation of Transactions per second

- Tuples in: The *Tuples in* the graph display the number of tuples inserted, updated, and deleted on the server or database.

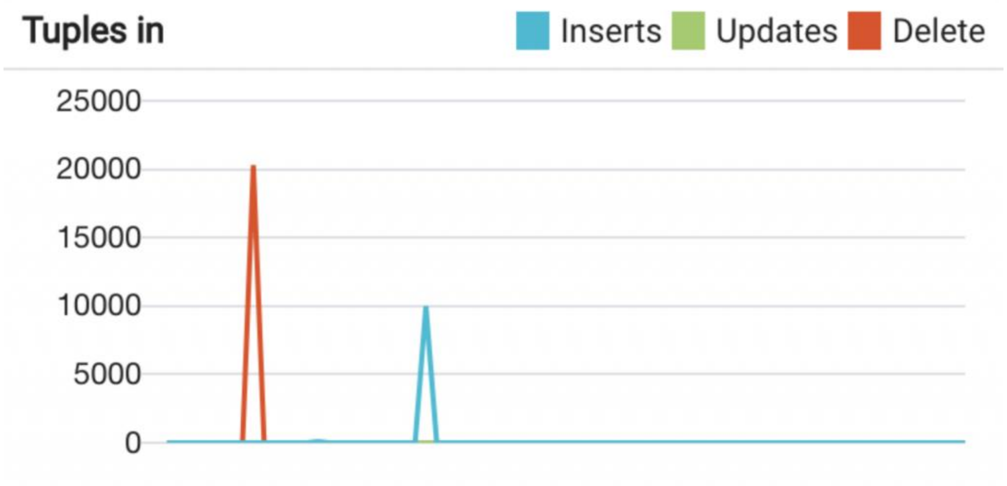


Fig. 2.3.1.3 Graphical representation of Tuples in

- Tuples out: The *Tuples out graph* displays the number of tuples fetched and returned from the server or database.

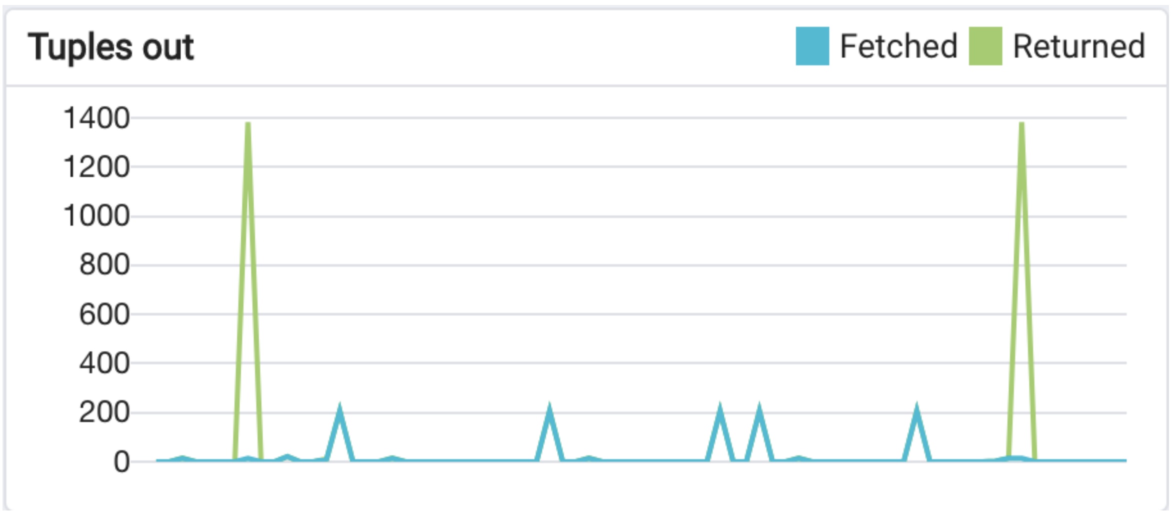


Fig. 2.3.1.4 Graphical representation of Tuples out

- Block I/O: The *Block I/O* graph displays the number of blocks read from the filesystem or fetched from the buffer cache (but not the operating system's file system cache) for the server or database.

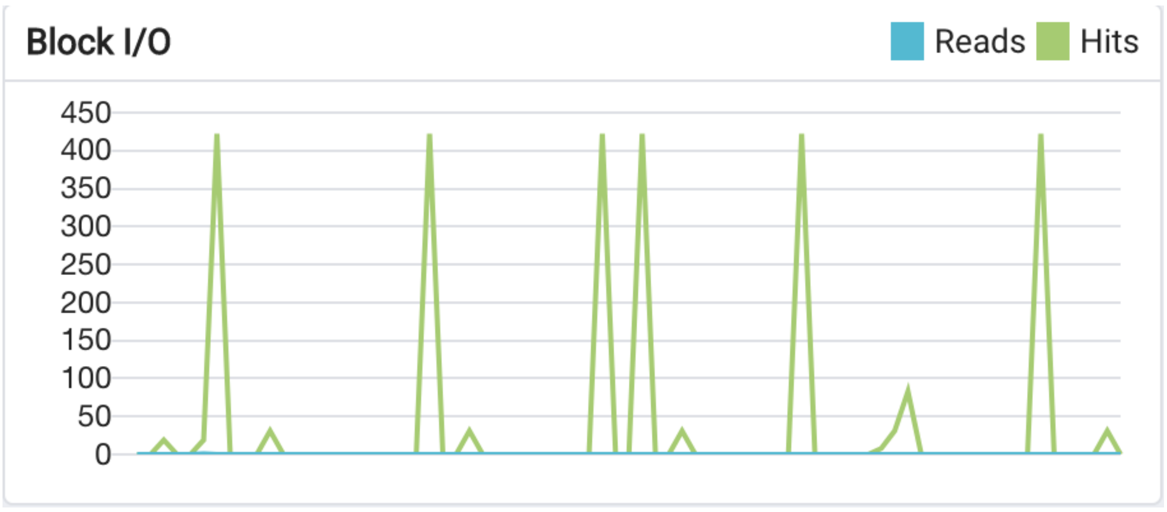
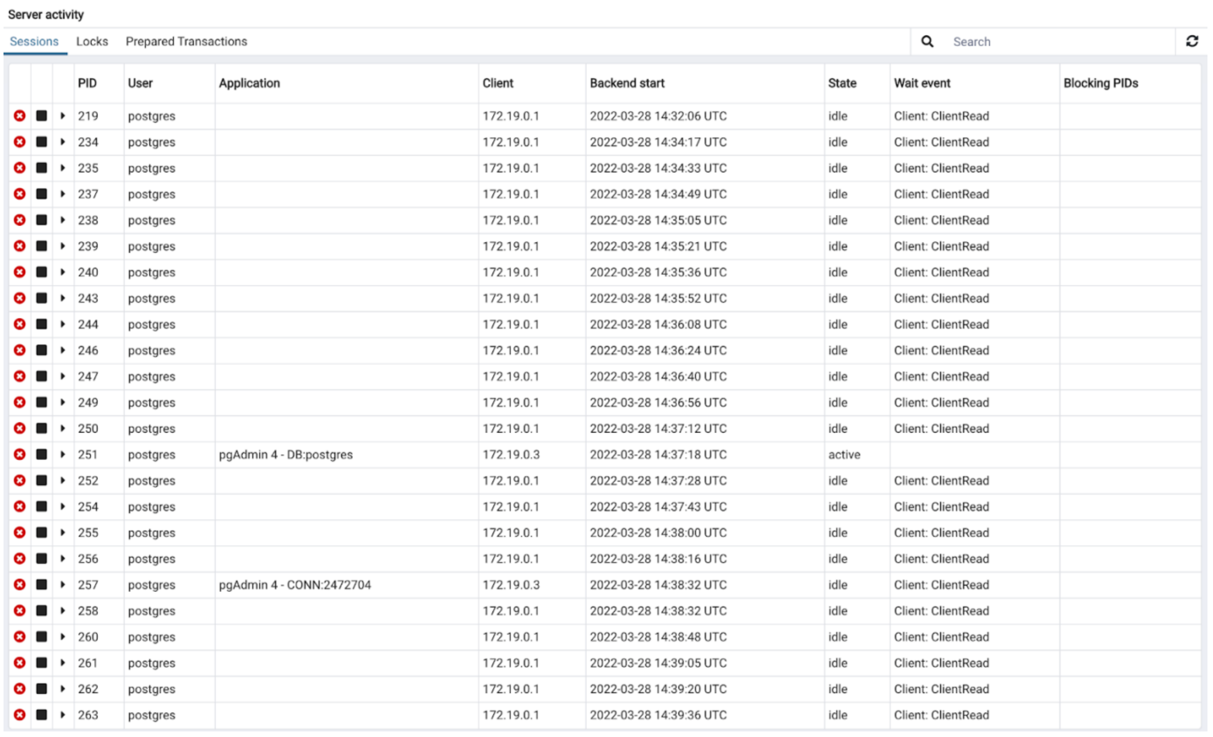


Fig. 2.3.1.5 Graphical representation of Block I/O

Server Activity

The Server activity panel displays information [9] about sessions, locks, prepared transactions, and server configuration (if a server is selected in the browser tree). Use the *Refresh* button to update the information in the table, and use the *Search box* to filter the results.

- Sessions: It shows all the active sessions for the selected Server or Database. Users can set the warning and alert threshold value to highlight the long-running queries on the dashboard from Preferences.


Server activity

Sessions Locks Prepared Transactions

			PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
⊗	■	▸	219	postgres		172.19.0.1	2022-03-28 14:32:06 UTC	idle	Client: ClientRead	
⊗	■	▸	234	postgres		172.19.0.1	2022-03-28 14:34:17 UTC	idle	Client: ClientRead	
⊗	■	▸	235	postgres		172.19.0.1	2022-03-28 14:34:33 UTC	idle	Client: ClientRead	
⊗	■	▸	237	postgres		172.19.0.1	2022-03-28 14:34:49 UTC	idle	Client: ClientRead	
⊗	■	▸	238	postgres		172.19.0.1	2022-03-28 14:35:05 UTC	idle	Client: ClientRead	
⊗	■	▸	239	postgres		172.19.0.1	2022-03-28 14:35:21 UTC	idle	Client: ClientRead	
⊗	■	▸	240	postgres		172.19.0.1	2022-03-28 14:35:36 UTC	idle	Client: ClientRead	
⊗	■	▸	243	postgres		172.19.0.1	2022-03-28 14:35:52 UTC	idle	Client: ClientRead	
⊗	■	▸	244	postgres		172.19.0.1	2022-03-28 14:36:08 UTC	idle	Client: ClientRead	
⊗	■	▸	246	postgres		172.19.0.1	2022-03-28 14:36:24 UTC	idle	Client: ClientRead	
⊗	■	▸	247	postgres		172.19.0.1	2022-03-28 14:36:40 UTC	idle	Client: ClientRead	
⊗	■	▸	249	postgres		172.19.0.1	2022-03-28 14:36:56 UTC	idle	Client: ClientRead	
⊗	■	▸	250	postgres		172.19.0.1	2022-03-28 14:37:12 UTC	idle	Client: ClientRead	
⊗	■	▸	251	postgres	pgAdmin 4 - DB:postgres	172.19.0.3	2022-03-28 14:37:18 UTC	active		
⊗	■	▸	252	postgres		172.19.0.1	2022-03-28 14:37:28 UTC	idle	Client: ClientRead	
⊗	■	▸	254	postgres		172.19.0.1	2022-03-28 14:37:43 UTC	idle	Client: ClientRead	
⊗	■	▸	255	postgres		172.19.0.1	2022-03-28 14:38:00 UTC	idle	Client: ClientRead	
⊗	■	▸	256	postgres		172.19.0.1	2022-03-28 14:38:16 UTC	idle	Client: ClientRead	
⊗	■	▸	257	postgres	pgAdmin 4 - CONN:2472704	172.19.0.3	2022-03-28 14:38:32 UTC	idle	Client: ClientRead	
⊗	■	▸	258	postgres		172.19.0.1	2022-03-28 14:38:32 UTC	idle	Client: ClientRead	
⊗	■	▸	260	postgres		172.19.0.1	2022-03-28 14:38:48 UTC	idle	Client: ClientRead	
⊗	■	▸	261	postgres		172.19.0.1	2022-03-28 14:39:05 UTC	idle	Client: ClientRead	
⊗	■	▸	262	postgres		172.19.0.1	2022-03-28 14:39:20 UTC	idle	Client: ClientRead	
⊗	■	▸	263	postgres		172.19.0.1	2022-03-28 14:39:36 UTC	idle	Client: ClientRead	

Fig. 2.3.2.1 Server Activity Dashboard

To stop a session and remove it from the table, you can utilize the Terminate icon located in the first column. The server will prompt you for confirmation before terminating the session. In the second column, you will find the Cancel icon which allows you to terminate an active query without closing the session. The server will ask for confirmation before canceling the query. When a query is canceled, the State column in the table will change from Active to Idle. The session will remain in the table until it is terminated. For more detailed information about a selected session, you can click on the Details icon in the third column. This will open the Details panel, displaying additional session information.

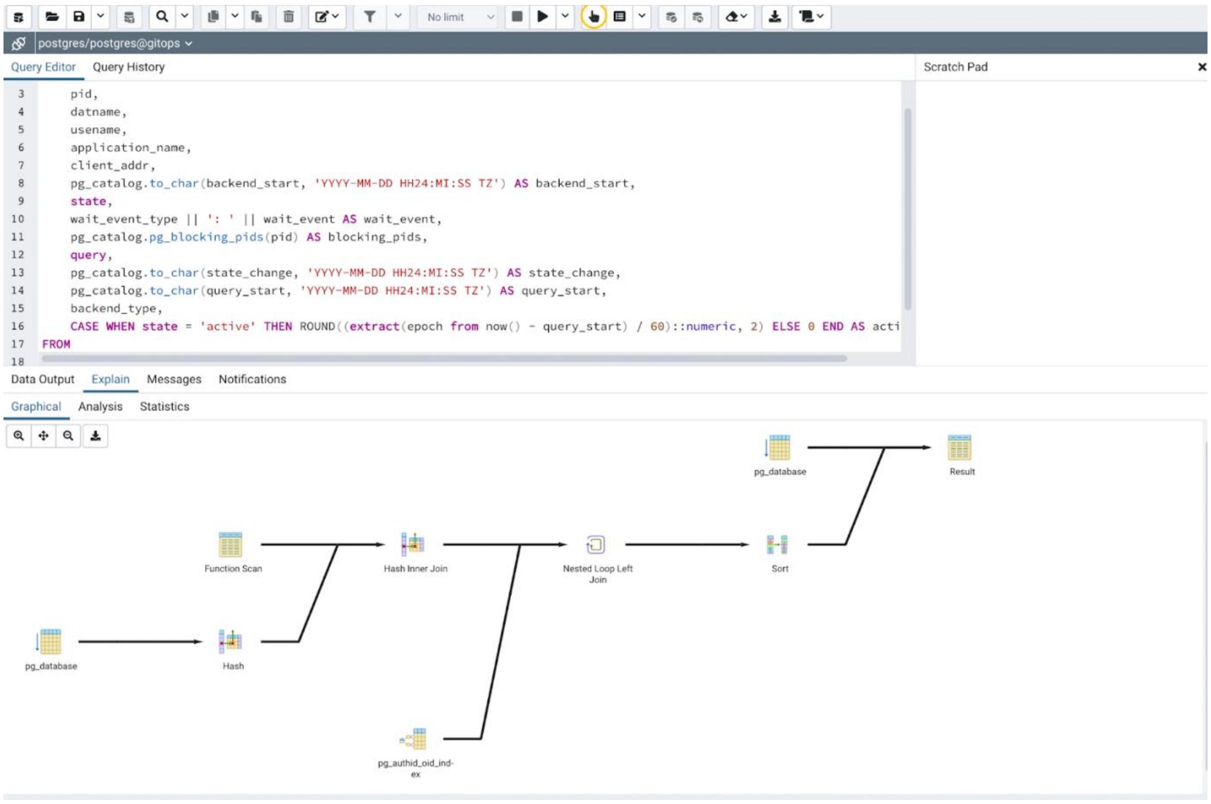
Explain Analyze Visualizer

PgAdmin [10] offers an array of tools, including a prominent one within the Query Tool. Notably, the Explain and Explain Analyze features play a vital role in this toolkit. These features process input queries from the Query Editor and present outputs across three tabs, each with specific insights. The *Explain* and *Explain Analyze* functionalities are pivotal in revealing the execution plan and performance details of a query. By utilizing these tools, users gain a comprehensive understanding of how PostgreSQL interprets, optimizes, and executes queries. The tab-based output organization enables users to systematically explore execution strategies, access methods, and associated costs. This breakdown aids in comprehending resource usage and performance bottlenecks.

In essence, the integration of Explain and Explain Analyze in PgAdmin [10]'s Query Tool empowers users with profound insights into query behavior. The structured output presentation reinforces the tool's role in demystifying the intricate aspects of query execution in PostgreSQL.

Using the following query in the query space followed by opting Explain Analyze option

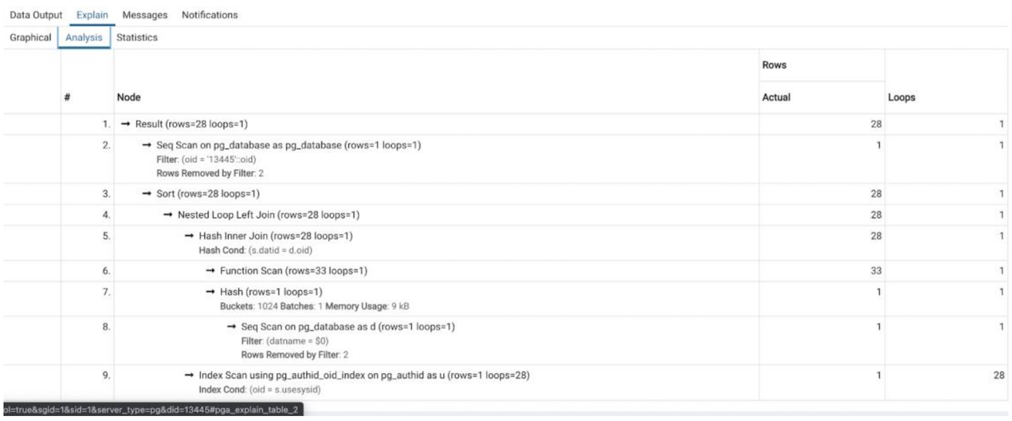
```
/*pga4dash*/
SELECT
   pid,
   datname,
   usename,
   application_name,
   client_addr,
   pg_catalog.to_char(backend_start, 'YYYY-MM-DD HH24:MI:SS TZ') AS backend_start,
   state,
   wait_event_type || ': ' || wait_event AS wait_event,
   pg_catalog.pg_blocking_pids(pid) AS blocking_pids,
   query,
   pg_catalog.to_char(state_change, 'YYYY-MM-DD HH24:MI:SS TZ') AS state_change,
   pg_catalog.to_char(query_start, 'YYYY-MM-DD HH24:MI:SS TZ') AS query_start,
   backend_type,
   CASE WHEN state = 'active' THEN ROUND((extract(epoch from now() - query_start) / 60)::numeric, 2)
ELSE 0 END AS active_since
FROM
   pg_catalog.pg_stat_activity
WHERE
   datname = (SELECT datname FROM pg_catalog.pg_database WHERE oid = 13445)ORDER BY pid
```



```
    pid,
    datname,
    usename,
    application_name,
    client_addr,
    pg_catalog.to_char(backend_start, 'YYYY-MM-DD HH24:MI:SS TZ') AS backend_start,
    state,
    wait_event_type || ': ' || wait_event AS wait_event,
    pg_catalog.pg_blocking_pids(pid) AS blocking_pids,
    query,
    pg_catalog.to_char(state_change, 'YYYY-MM-DD HH24:MI:SS TZ') AS state_change,
    pg_catalog.to_char(query_start, 'YYYY-MM-DD HH24:MI:SS TZ') AS query_start,
    backend_type,
    CASE WHEN state = 'active' THEN ROUND((extract(epoch from now() - query_start) / 60)::numeric, 2) ELSE 0 END AS acti
FROM
```

Fig 2.3.3.1 Graphical tab gives a visual representation of how the query is fetching the desired tuples

The *Analysis tab* in the tool displays the plan details in a tabular format. Each row corresponds to an Explain Plan Node and contains various information such as node details, exclusive and inclusive timing, differences between actual and planned rows, actual rows, planned rows, and loops [discussed in section 2.2]. By setting Thresholds, we can determine when our query is not performing optimally and not retrieving values in the most efficient manner.

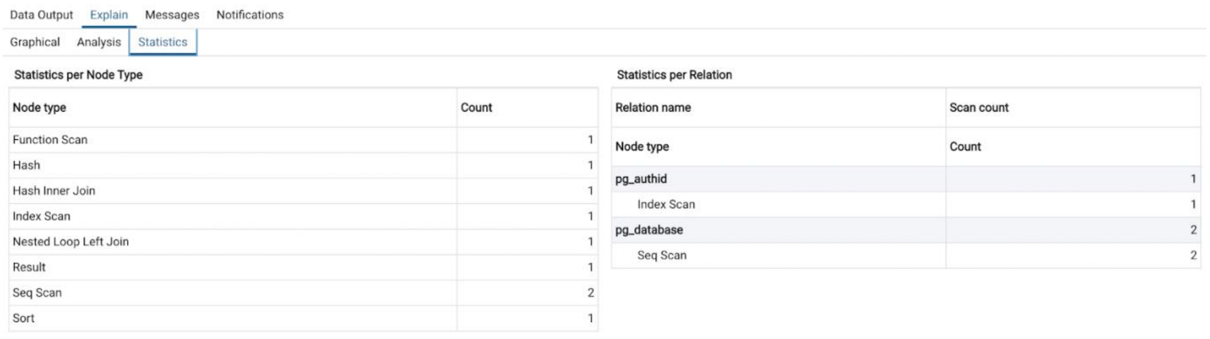


		Rows	
#	Node	Actual	Loops
1.	→ Result (rows=28 loops=1)	28	1
2.	→ Seq Scan on pg_database as pg_database (rows=1 loops=1) Filter: (oid = '13445'::oid) Rows Removed by Filter: 2	1	1
3.	→ Sort (rows=28 loops=1)	28	1
4.	→ Nested Loop Left Join (rows=28 loops=1)	28	1
5.	→ Hash Inner Join (rows=28 loops=1) Hash Cond: (s.datid = d.oid)	28	1
6.	→ Function Scan (rows=33 loops=1)	33	1
7.	→ Hash (rows=1 loops=1) Buckets: 1024 Batches: 1 Memory Usage: 9 kB	1	1
8.	→ Seq Scan on pg_database as d (rows=1 loops=1) Filter: (datname = $0) Rows Removed by Filter: 2	1	1
9.	→ Index Scan using pg_authid_oid_index on pg_authid as u (rows=1 loops=28) Index Cond: (oid = s.usesysid)	1	28

```
ol=true&sgid=1&sid=1&server_type=pg&did=13445#pga_explain_table_2
```

Fig 2.3.3.2 Analysis tab metadata

The *Statistics tab* shows two tables: Statistics per Node Type and Statistics per Table/Relation. Both these tables contain insights regarding the query that we execute [discussed in section 2.2].



Statistics per Node Type

Node type	Count
Function Scan	1
Hash	1
Hash Inner Join	1
Index Scan	1
Nested Loop Left Join	1
Result	1
Seq Scan	2
Sort	1

Statistics per Relation

Relation name	Scan count
Node type	Count
pg_authid	1
Index Scan	1
pg_database	2
Seq Scan	2

Fig 2.3.3.2 Statistics tab metadata

Conclusion: PgAdmin serves as a popular and user-friendly dashboard for accessing database information. It effectively presents data in a clear and comprehensible manner. The Query Tool facilitates visual interpretation of query costs, while configurations can be adjusted as needed. The Analysis tab allows for highlighting triggers and thresholds, the Graphical tab provides a query history view, and the dashboard Graphs offer valuable insights into the database. Overall, PgAdmin is a valuable tool for database management.

DALIBO

The *Explain* command serves as a means to extract a query's execution plan, offering raw data. This unprocessed information can be further utilized by inputting it, along with the query, into *explain.dalibo.com* [11]. Notably, this tool takes on the task of meticulously dissecting the data, presenting it in a visually comprehensible manner. The tool's capacity lies in delivering lucid and detailed insights, effectively unraveling the complexities of the query's execution intricacies. Looking at the example used above we get :
[These can be directly accessed <u>here</u>]

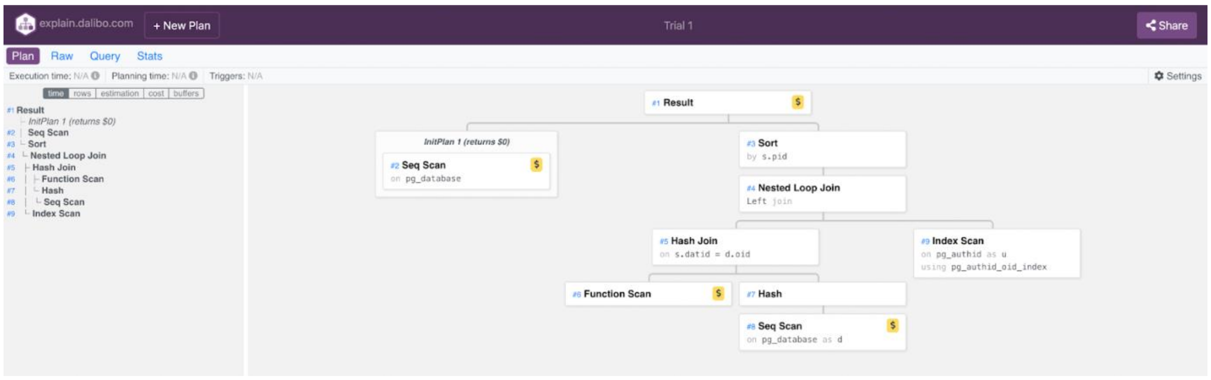


Fig 2.4.1 Overview of the structure of query

DALIBO's query plan provides a comprehensive visualization of PostgreSQL's query execution strategies. It transforms raw Explain command output into a visually intuitive representation. This tool facilitates an insightful understanding of query optimization, revealing access methods, join strategies, and cost evaluations. DALIBO's query plan aids database professionals in deciphering and enhancing performance by presenting intricate execution details in a user-friendly format.

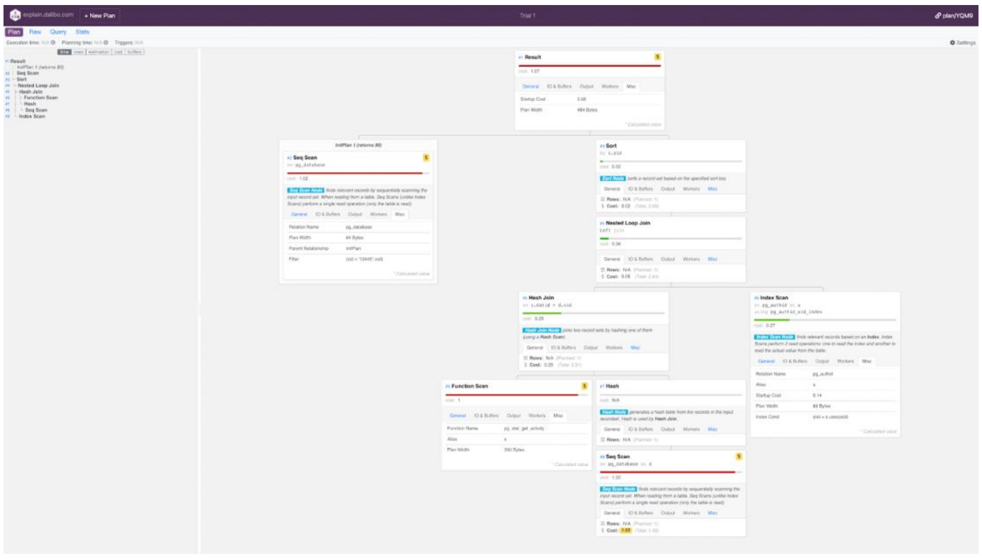
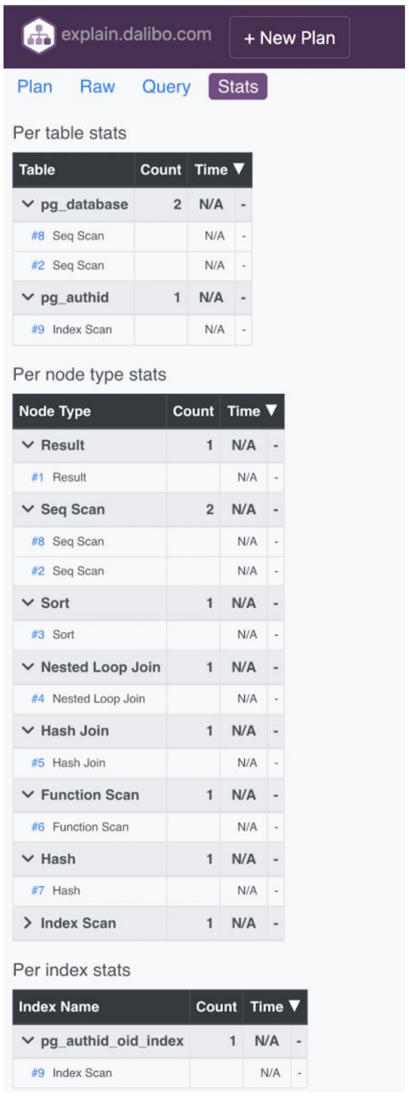


Fig 2.4.2 Each card is clicked for detailed figures and structure



Per table stats

Table	Count	Time ▼	
pg_database	2	N/A	-
#8 Seq Scan		N/A	-
#2 Seq Scan		N/A	-
pg_authid	1	N/A	-
#9 Index Scan		N/A	-

Per node type stats

Node Type	Count	Time ▼	
Result	1	N/A	-
#1 Result		N/A	-
Seq Scan	2	N/A	-
#8 Seq Scan		N/A	-
#2 Seq Scan		N/A	-
Sort	1	N/A	-
#3 Sort		N/A	-
Nested Loop Join	1	N/A	-
#4 Nested Loop Join		N/A	-
Hash Join	1	N/A	-
#5 Hash Join		N/A	-
Function Scan	1	N/A	-
#6 Function Scan		N/A	-
Hash	1	N/A	-
#7 Hash		N/A	-
Index Scan	1	N/A	-

Per index stats

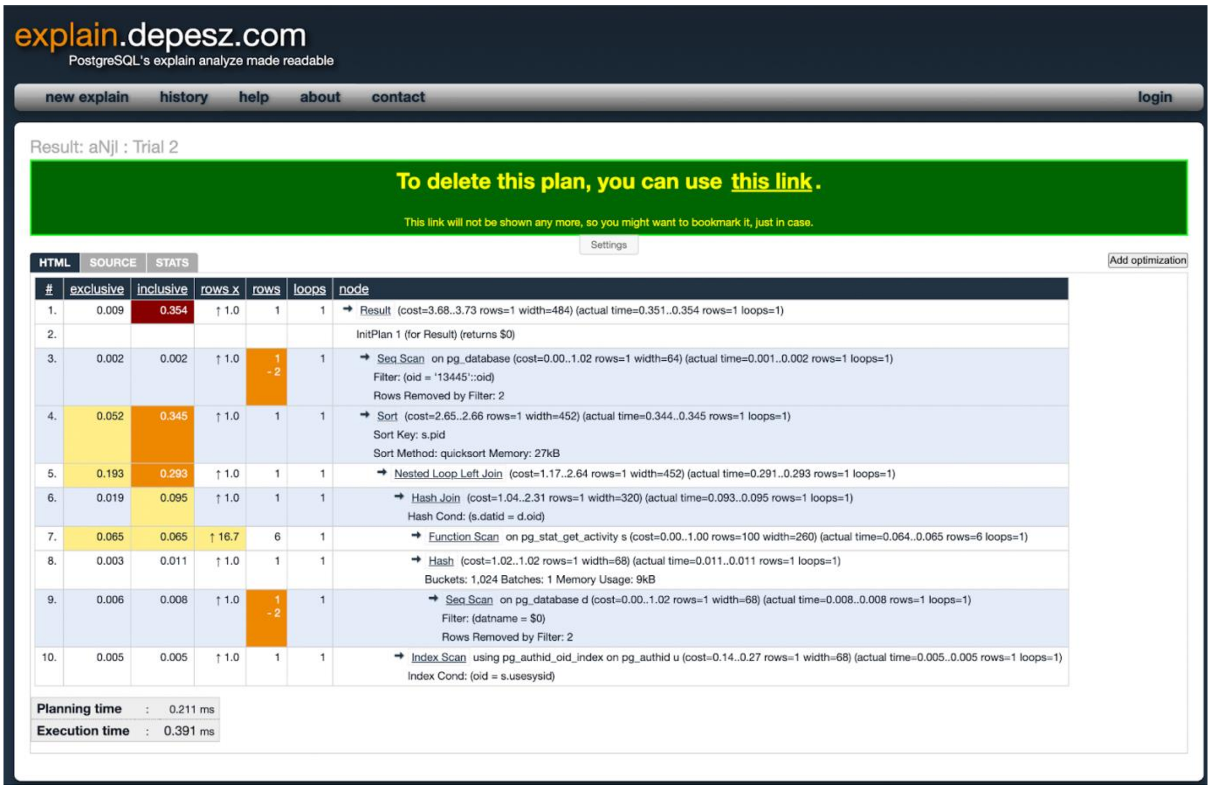
Index Name	Count	Time ▼	
pg_authid_oid_index	1	N/A	-
#9 Index Scan		N/A	-

Fig 2.4.3 The Stats tab shows information in a tabular form

Conclusion: This web-based tool offers a valuable means of assessing query costs in a machine-independent manner, as both the query plan and the query itself are manually input. Noteworthy for its thoroughness, the tool meticulously furnishes comprehensive insights at each stage, table, and node of the query execution process. The graphical chart representation serves as a portal to delve deeper into each card, revealing nuanced options such as General, I/O Buffers, Output, Workers, and Miscellaneous. This capability empowers users to engage in granular analysis, enabling a more profound understanding of query performance and facilitating well-informed optimization strategies.

DEPESZ
This tool is a visualization tailored for the EXPLAIN ANALYZE command which is fed to - explain.depesz.com [12]. Here too the query is put manually, and the query plan which is then represented with colour gradations. It operates by accepting the query plan as manual input, subsequently translating it into a graphical representation enriched with color gradations. A distinguishing feature is the option for users to log in, enabling the logging of instances with enhanced clarity. By considering the example at hand, the intricacies of query execution become more transparent, as the visualization effectively conveys the nuances of PostgreSQL's query optimization process. This tool aligns with the aim of simplifying the comprehension of complex query plans, thus facilitating informed decision-making in database optimization endeavors. The instances would be registered clearly because a user has the option to log in. Looking at the example used here we get :

[These can be directly accessed here]



Result: aNjl : Trial 2

To delete this plan, you can use this link.

This link will not be shown any more, so you might want to bookmark it, just in case.

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.009	0.354	↑ 1.0	1	1	→ Result (cost=3.68..3.73 rows=1 width=484) (actual time=0.351..0.354 rows=1 loops=1)
2.						InitPlan 1 (for Result) (returns $0)
3.	0.002	0.002	↑ 1.0	1 - 2	1	→ Seq Scan on pg_database (cost=0.00..1.02 rows=1 width=64) (actual time=0.001..0.002 rows=1 loops=1) Filter: (oid = '13445'::oid) Rows Removed by Filter: 2
4.	0.052	0.345	↑ 1.0	1	1	→ Sort (cost=2.65..2.66 rows=1 width=452) (actual time=0.344..0.345 rows=1 loops=1) Sort Key: s.pid Sort Method: quicksort Memory: 27kB
5.	0.193	0.293	↑ 1.0	1	1	→ Nested Loop Left Join (cost=1.17..2.64 rows=1 width=452) (actual time=0.291..0.293 rows=1 loops=1)
6.	0.019	0.095	↑ 1.0	1	1	→ Hash Join (cost=1.04..2.31 rows=1 width=320) (actual time=0.093..0.095 rows=1 loops=1) Hash Cond: (s.datid = d.oid)
7.	0.065	0.065	↑ 16.7	6	1	→ Function Scan on pg_stat_get_activity s (cost=0.00..1.00 rows=100 width=260) (actual time=0.064..0.065 rows=6 loops=1)
8.	0.003	0.011	↑ 1.0	1	1	→ Hash (cost=1.02..1.02 rows=1 width=68) (actual time=0.011..0.011 rows=1 loops=1) Buckets: 1,024 Batches: 1 Memory Usage: 9kB
9.	0.006	0.008	↑ 1.0	1 - 2	1	→ Seq Scan on pg_database d (cost=0.00..1.02 rows=1 width=68) (actual time=0.008..0.008 rows=1 loops=1) Filter: (datname = $0) Rows Removed by Filter: 2
10.	0.005	0.005	↑ 1.0	1	1	→ Index Scan using pg_authid_oid_index on pg_authid u (cost=0.14..0.27 rows=1 width=68) (actual time=0.005..0.005 rows=1 loops=1) Index Cond: (oid = s.usesysid)

Planning time	:	0.211 ms
Execution time	:	0.391 ms

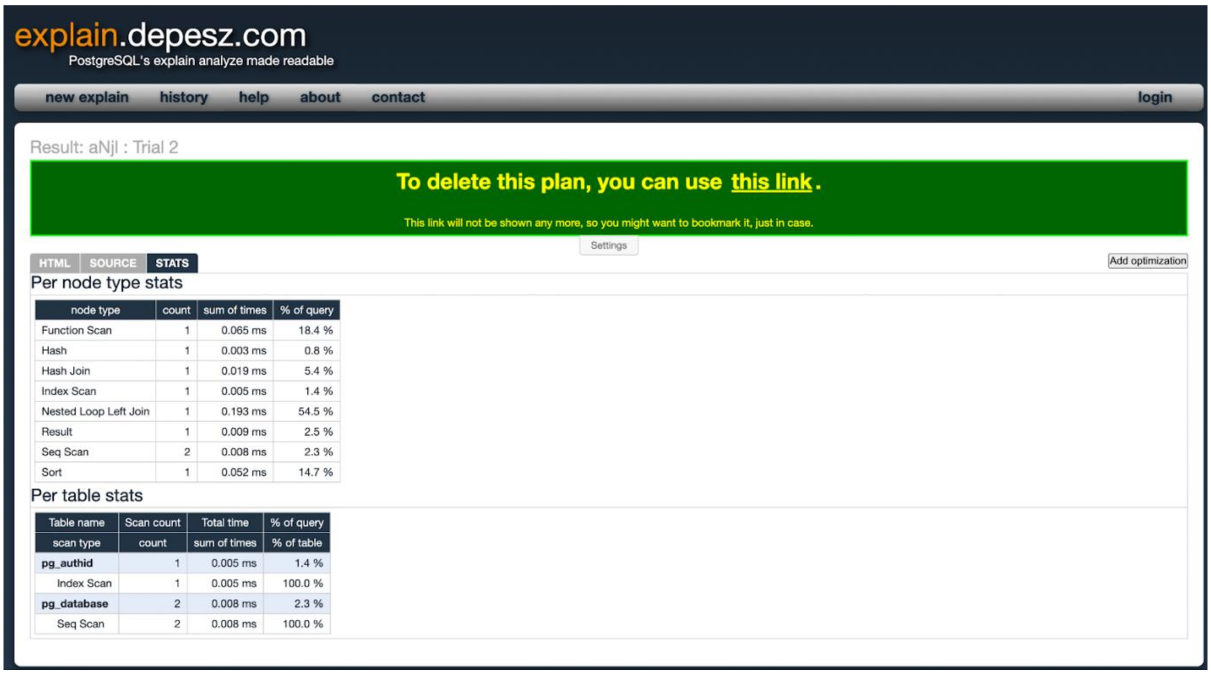
Fig 2.5.1 depesz dashboard that provides visualization of query cost

The exclusive and inclusive columns are something that we can gain insights from. These two columns predominantly mean:

- The *exclusive* column is the total amount of time PostgreSQL spent evaluating this node, without time spent in its subnodes.
- The inclusive column is just like the *exclusive*, but it doesn't exclude the time of subnodes. Hence the top node's time would be equivalent to the total time.

An example can be found here: *exclusive* and *inclusive*.

These columns can also be configured for PgAdmin. Furthermore, the Stats tab shows:



Result: aNjl : Trial 2

To delete this plan, you can use this link.

This link will not be shown any more, so you might want to bookmark it, just in case.

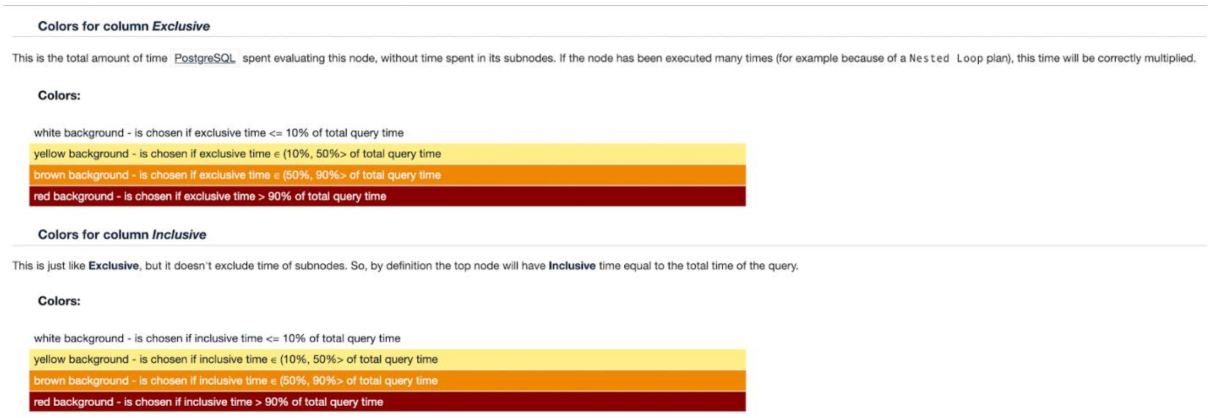
Per node type stats

node type	count	sum of times	% of query
Function Scan	1	0.065 ms	18.4 %
Hash	1	0.003 ms	0.8 %
Hash Join	1	0.019 ms	5.4 %
Index Scan	1	0.005 ms	1.4 %
Nested Loop Left Join	1	0.193 ms	54.5 %
Result	1	0.009 ms	2.5 %
Seq Scan	2	0.008 ms	2.3 %
Sort	1	0.052 ms	14.7 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
pg_authid	1	0.005 ms	1.4 %
Index Scan	1	0.005 ms	100.0 %
pg_database	2	0.008 ms	2.3 %
Seq Scan	2	0.008 ms	100.0 %

Fig 2.4.3 The Stats tab shows information per node and per table



Colors for column *Exclusive*

This is the total amount of time PostgreSQL spent evaluating this node, without time spent in its subnodes. If the node has been executed many times (for example because of a `Nested Loop` plan), this time will be correctly multiplied.

Colors:

white background - is chosen if exclusive time <= 10% of total query time
yellow background - is chosen if exclusive time ∊ (10%, 50%> of total query time
brown background - is chosen if exclusive time ∊ (50%, 90%> of total query time
red background - is chosen if exclusive time > 90% of total query time

Colors for column *Inclusive*

This is just like **Exclusive**, but it doesn't exclude time of subnodes. So, by definition the top node will have **Inclusive** time equal to the total time of the query.

Colors:

white background - is chosen if inclusive time <= 10% of total query time
yellow background - is chosen if inclusive time ∊ (10%, 50%> of total query time
brown background - is chosen if inclusive time ∊ (50%, 90%> of total query time
red background - is chosen if inclusive time > 90% of total query time

Fig 2.4.4 Colour chart inference table

Conclusion: This tool merges the features of Dalibo and PgAdmin to create a comprehensive dashboard. It offers enhanced visibility by highlighting each node in the exclusive and inclusive columns, allowing for easy identification of query cost anomalies. Additional optimizations can be implemented, and it provides the ability to compare with previous query plans. Overall, this tool provides a robust platform for analyzing and optimizing query performance.

CONCLUSION

This paper has undertaken a comprehensive analysis of tools for measuring PostgreSQL query cost, shedding light on the intricacies of query execution and optimization. Through the utilization of tools such as EXPLAIN, EXPLAIN ANALYSE, PgAdmin, Dalibo, and Depesz, the challenges and nuances associated with query performance have been dissected and explored. The trajectory of PostgreSQL in addressing these challenges has been meticulously showcased, illuminating its journey of evolution. The insights derived from this analysis hold immense value for practitioners and researchers alike in the realm of database systems.

The study commenced by emphasizing the paramount importance of precise query execution time prediction in database management. This task hinges upon intricate cost models, facing challenges related to selectivity estimation and cost-modeling errors. The study then delved into the foundational concepts underlying PostgreSQL query cost measurement, highlighting factors like plan structure, query planning, and the role of indexes and data statistics in influencing query optimization.

Subsequently, the paper explored a range of tools for query cost analysis, starting with the EXPLAIN command. It elucidated how EXPLAIN offers insight into query planning and associated costs, aiding in the determination of the most efficient approach for delivering query results. The paper discussed the format of costs, the role of cost constants, and their influence on query execution decisions.

The EXPLAIN ANALYZE tool was introduced, offering an expanded version of EXPLAIN that executes the query and provides actual runtime statistics. The paper detailed how EXPLAIN ANALYZE aids in understanding query performance through actual execution insights, including time spent, rows returned, and loops performed.

The analysis extended to GUI tools like PgAdmin, highlighting its user-centric design and versatility in database management. The various graphical representations, graphs, and panels within PgAdmin were explored to illustrate its efficacy in presenting information about server sessions, transactions, tuples, and I/O operations. The Explain Analyze Visualizer within PgAdmin was discussed as a vital tool for query plan visualization and analysis.

The study then shifted its focus to external tools such as Dalibo and Depesz. Dalibo was showcased as a platform for transforming raw EXPLAIN command output into visually intuitive representations, aiding in deciphering query execution strategies and optimization. Depesz was introduced as a comprehensive tool that combines features of both Dalibo and PgAdmin, providing a robust platform for analyzing and optimizing query performance through visualizations and color-coded representations.

In conclusion, this analysis aims to provide a comprehensive overview of tools for measuring PostgreSQL query cost. The paper's systematic approach offers valuable insights into the intricate world of query execution and optimization, equipping practitioners and researchers with essential knowledge for enhancing database system efficiency and proficiency. The adoption of various tools in this study has bridged the gap between theory and practice, contributing to the refinement of database systems and fostering an environment of informed decision-making in query optimization endeavors.

REFERENCES

1. Nilesh Jayanandana (2021), PostgreSQL Monitoring: The Best Tools and Key Metrics to Help Improve Database Performance, *sematext*, https://sematext.com/blog/postgresql-monitoring/
2. PostgreSQL v15 (2023), Chapter 15. Parallel Query, Part II. The SQL Language, *PostgreSQL Official Documentation*, https://www.postgresql.org/docs/current/parallel-query.html
3. Lawrence Jones (2022), Debugging the Postgres query planner, *GoCardless*, https://gocardless.com/blog/debugging-the-postgres-query-planner/
4. Using EXPLAIN, *PostgreSQL Wiki*, https://wiki.postgresql.org/wiki/Using_EXPLAIN
5. Minh Nguyen (2021), How to read PostgreSQL query plan, *Medium, Geek Culture* https://medium.com/geekculture/how-to-read-postgresql-query-plan-df4b158781a1
6. PostgreSQL v15 (2023), F.32. pg_stat_statements, Appendix F. Additional Supplied Modules, *PostgreSQL Official Documentation*, https://www.postgresql.org/docs/current/pgstatstatements.html
7. Hans-Jürgen Schönig (2015), PG_STAT_STATEMENTS: THE WAY I LIKE IT, *cybertec*, https://www.cybertec-postgresql.com/en/pg_stat_statements-the-way-i-like-it/
8. Developer Tools, pgAdmin 4, *pgAdmin Official Documentation*, https://www.pgadmin.org/docs/pgadmin4/latest/developer_tools.html
9. PostgreSQL 15, 14.2. Statistics Used by the Planner, Chapter 14. Performance Tips, *PostgreSQL Official Documentation*, https://www.postgresql.org/docs/current/planner-stats.html
10. Configuring pgadmin and Postgres, Tutorials, *Database Labs*, https://www.databaselabs.io/help/tutorials/configuring-pgadmin-postgres
11. Dalibo Official website, *explain.dalibo.com* PostgreSQL execution plan visualizer, https://explain.dalibo.com/
12. Despesz Official website, *explain.depesz.com*, PostgreSQL explain analyse made readable, https://explain.depesz.com/
13. S. Currim, R. T. Snodgrass, Y-K. Suh, R. Zhang, M. W. Johnson, and C. Yi. "*Dbms metrology: Measuring query time*". SIGMOD, 2013.
14. W. Wu, S. Zhu Y. Chi, J. Tatemura, H. Hacig¨um¨u¸s, and J.F. Naughton. "*Predicting query execution time: Are optimizer cost models really unusable?*" ICDE, 2013.