

Mathematisch-Naturwissenschaftliche Fakultät  
Wilhelm-Schickard-Institut für Informatik  
Database Systems Research Group

---

Bachelorthesis Computer Science

**Execution Visualized: Shareable Plan Visualization for DuckDB based  
on PEV2**

---

Matthis Noël

31.05.2025

Examiner

Prof. Torsten Grust

Supervisor

Denis Hirn

**Matthis Noël:**

*Execution Visualized: Shareable Plan Visualization for DuckDB based on PEV2*

Bachelorthesis Computer Science

Eberhard Karls Universität Tübingen

From 01.02.2025 to 31.05.2025

# Selbständigkeitserklärung

---

Hiermit versichere ich, dass ich die vorliegende Bachelorthesis selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ich habe KI für Korrekturen der Rechtschreibung und Grammatik genutzt, ohne dass es dabei zu inhaltlich relevanter Textgeneration oder Übersetzungen kam. Das heißt, ich habe von mir verfasste Texte in derselben Sprache korrigieren lassen. Es handelt sich um rein sprachliche Korrekturen, so dass die von mir ursprünglich intendierte Bedeutung nicht wesentlich verändert oder erweitert wurde. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

Tübingen, 30.05.2025

---

Ort, Datum



---

Matthis Noël



# Abstract

---

The DuckDB Explain Visualizer is a web-based tool for visualizing and sharing execution plans generated by the DuckDB database management system. Inspired by the PostgreSQL Execution Plan Visualizer 2 (PEV2), the tool has been adapted to support the JSON structure and execution characteristics of DuckDB. The visualizer offers a comprehensive interface that includes a tree-based plan view, a grid layout, statistical summaries and direct plan sharing functionality. Users can upload JSON plans via `EXPLAIN` or `PRAGMA` statements and interactively explore detailed operator-level metrics, such as execution time, row counts and result sizes. The tool enhances readability and supports performance debugging, query optimization, and collaborative analysis. Challenges in implementation included adapting the parser to the DuckDB-specific structure, modifying the visualization components and integrating runtime statistics. The result is an intuitive and shareable visual representation of query plans that improves transparency and accessibility for DuckDB developers, researchers, and educators.



# Contents

---

Abstract	v
1 Introduction	1
1.1 Relation Database Management Systems and SQL	1
1.1.1 Data Types and Constraints	1
1.2 Execution Plan Visualization	2
1.3 Motivation	2
2 Implementation	5
2.1 The PostgreSQL Execution Plan Visualizer 2	5
2.2 From PEV2 to DuckDB Explain Visualizer	8
2.2.1 PostgreSQL vs. DuckDB JSON Plan Structure	8
2.2.2 Plan View: Node Construction	13
2.2.3 Plan View: Diagram	21
2.2.4 Grid View	23
2.2.5 Stats View	24
2.2.6 Plan Sharing Service	25
3 Results	27
3.1 How to use the DuckDB Explain Visualizer	27
3.2 Features	28
3.3 Future Work	28
Bibliography	33



# Introduction

---

The goal of this thesis is to redesign and implement a version of the PostgreSQL Explain Visualizer 2 (PEV2) library capable of interpreting and visualizing DuckDB execution plans. This involves adapting the current architecture of PEV2, which was originally designed for PostgreSQL, to support the structure, semantics, and specific features of execution plans generated by DuckDB.

## 1.1 Relation Database Management Systems and SQL

Relational Database Management Systems (RDBMSs) and SQL are fundamental tools for managing structured data in modern computing. A database is simply a collection of data, while an RDBMS is the software that allows you to store, manage, and interact with that data. SQL is used to efficiently access, manipulate, and query this information.

Popular RDBMSs include PostgreSQL, MySQL, SQLite, and the newer DuckDB. All of these systems follow the relational data model, which organizes data into tables—called relations—consisting of rows (tuples) and columns (attributes). Despite this common foundation, each system differs in how it handles data types, SQL dialects, and performance optimizations. **PostgreSQL** is praised for its standard compliance and extensibility. **MySQL** is valued for its ubiquity and speed. **SQLite** is preferred for its lightweight, serverless design. **DuckDB** is designed for embedded use and optimized for analytical workloads [1, 2].

Most RDBMSs implement their own dialect of SQL, often extending the SQL standard, which is maintained by the American National Standards Institute (ANSI), the International Organization for Standardization (ISO), and the International Electrotechnical Commission (IEC) to provide advanced functionality. However, full compliance with the standard is rare due to its complexity (for example, the SQL:2011 version includes 179 core features). In comparison, PostgreSQL implements 160 of these core features [2].

### 1.1.1 Data Types and Constraints

Each RDBMS defines its own set of data types, which determines what kind of data can be stored in each column. Common types include integers, strings, dates, and booleans. For example, MySQL provides a `tinyint` type that can be signed (from -128 to 127) or unsigned (0 to 255). These differences mean that data types are not always directly interchangeable between systems. [2]

## 1.2 Execution Plan Visualization

For developers who work with relational databases, understanding how queries are executed by the database engine is critical to ensure both correctness and performance. An execution plan provides detailed information on the steps the database takes to retrieve and process data such as table scans, index usage, join strategies and filter operations.

However, these execution plans are often provided in raw or semistructured formats that can be difficult to interpret, especially for those without extensive experience with database internals. To address this challenge, execution plan visualization presents these plans in a more accessible and readable format—in example, as a textual graph within the terminal—making it easier to understand the logical structure and flow of query execution. This form of visualization offers several important benefits to SQL developers:

- It helps **identify performance problems** by clearly highlighting inefficient operations, such as unnecessary full table scans or costly nested loop joins.
- It supports **debugging efforts** by revealing how the database engine actually interprets and executes a given query, which may differ from the developer’s intentions.
- It facilitates **query optimization** by revealing execution paths and the relative cost of each operation, allowing for more informed tuning decisions.
- It facilitates **learning and communication** by providing a visual aid for understanding query processing and sharing insights with peers.

## 1.3 Motivation

If you spend enough time looking at DuckDB query plans, you will eventually realize that the in-terminal textual graph representation of a query plan (Figure 1.1) has its limitations. It can be difficult to read and understand, especially for complex plans. Worse yet, if the plan is too large, part of it may be cut off, making the entire plan impossible to understand. Additionally, there are no visual cues to help you understand the plan, identify bottlenecks, or find optimization opportunities. [3].

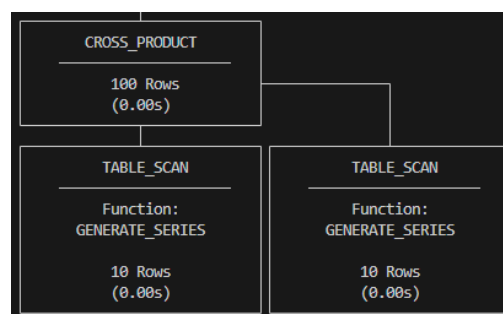


Figure 1.1: DuckDB in-terminal visualization

```

1 EXPLAIN (FORMAT JSON)
2 SELECT x, y
3 FROM generate_series(1, 10) AS x, generate_series(1,10) AS y;

```

Listing 1.1: Example Query

Fortunately, the DuckDB team has introduced an alternative method for representing query plans. As a JSON object, which can be visualized more easily. An example query is shown in Listing 1.1 and the resulting plan in Listing 1.2.

```

1 [
2   {
3     "name": "PROJECTION",
4     "children": [
5       {
6         "name": "CROSS_PRODUCT",
7         "children": [
8           {
9             "name": "GENERATE_SERIES ",
10            "children": [],
11            "extra_info": {
12              "Function": "GENERATE_SERIES",
13              "Estimated Cardinality": "9"
14            }
15          },
16          {
17            "name": "GENERATE_SERIES ",
18            "children": [],
19            "extra_info": {
20              "Function": "GENERATE_SERIES",
21              "Estimated Cardinality": "9"
22            }
23          }
24        ],
25        "extra_info": {}
26      }
27    ],
28    "extra_info": {
29      "Projections": [
30        "struct_pack(generate_series)",
31        "struct_pack(generate_series)"
32      ],
33      "Estimated Cardinality": "81"
34    }
35  }
36 ]

```

Listing 1.2: Example Query Plan

This is similar to the `EXPLAIN (FORMAT JSON)` feature in PostgreSQL. We use the JSON representation to render the query plan in a visually appealing and interactive way that provides additional information, such as the slowest operator or the number of rows processed by each operator [3].

## Implementation

---

### 2.1 The PostgreSQL Execution Plan Visualizer 2

The PostgreSQL Execution Plan Visualizer 2 (PEV2) is an open source interactive web tool for visualizing PostgreSQL execution plans developed by Dalibo [4] based on the Postgres Explain Visualizer (PEV) by Alex Tatiyants [5]. The PEV project was initially written in early 2016 but seems to have been abandoned since then, so Pierre Giraud from the Dalibo company started the PEV2 project. The first release version (v1.1.0) has been published on 13th septembre in 2019. It supports plan sharing, so you can share plans with other developers and view them without generating the plan on your own. All you have to do is share the URL of the opened plan website with them. In addition, the visualizer library on its own—without the sharing service—is designed to work entirely in the browser, requires no server-side components, so all parsing, transformation, and rendering of the plan data happen client-side. Furthermore, PEV2 is designed to be user-friendly: You only have to drag and drop the execution plan as a JSON file in the specified text area, seen in Figure 2.1.

The image shows the PEV2 home view interface. It consists of three main input sections and two buttons at the bottom. The first section is labeled 'Title' and contains a single-line text input field. The second section is labeled 'Plan (text or JSON)' and contains a larger text area with the placeholder text 'Paste execution plan or drop a file'. The third section is labeled 'Query (optional)' and contains another large text area with the placeholder text 'Paste corresponding SQL query or drop a file'. At the bottom left is a purple 'Submit' button, and at the bottom right is a dark grey 'Sample Plans' button with a small downward arrow.

Figure 2.1: PEV2 home view

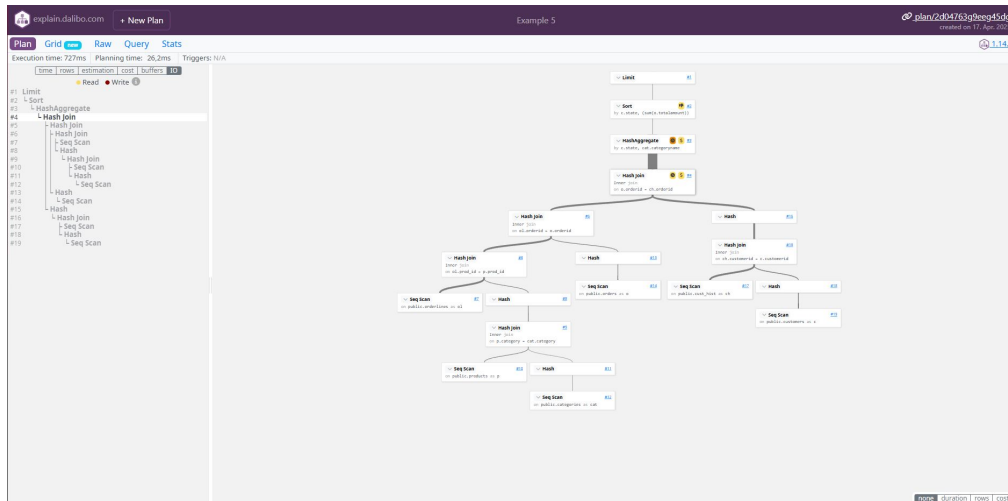


Figure 2.2: PostgreSQL Explain Visualizer 2

After submitting the plan by clicking **SUBMIT**, the plan view opens and shows the whole execution plan as a tree where each node represents an execution node/operator (for example, Hash Join) and after clicking on a node it shows a node description and many technical details about its execution (total execution time, computed rows, join type, etc.). An example of the node visualization is shown in Figure 2.3 and Figure 2.4. At the leftmost section in Figure 2.2 we can see the hierarchical structure of the plan—that section is called *Diagram* in the following mentions—with different statistics options showing the specific statistic value of a single node compared to the maximum value of the statistic value of all nodes. An example is shown in Figure 2.5 for better understanding. In the navigation menu there are some more feature pages like the **Grid**, **Raw**, **Query** and **Stats** tabs.

- **Grid:** Shows the execution plan in hierarchical visualization with the same node details as the plan view except for the general tab information.
- **Raw:** Shows the input JSON file or plain text.
- **Query:** Displays the query text if specified.
- **Stats:** Lists the statistics of all tables, functions, node types and indices.

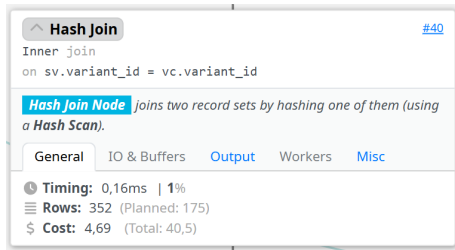


Figure 2.3: PEV2 node general details

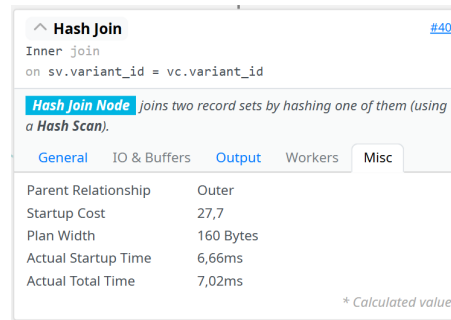


Figure 2.4: PEV2 node misc details

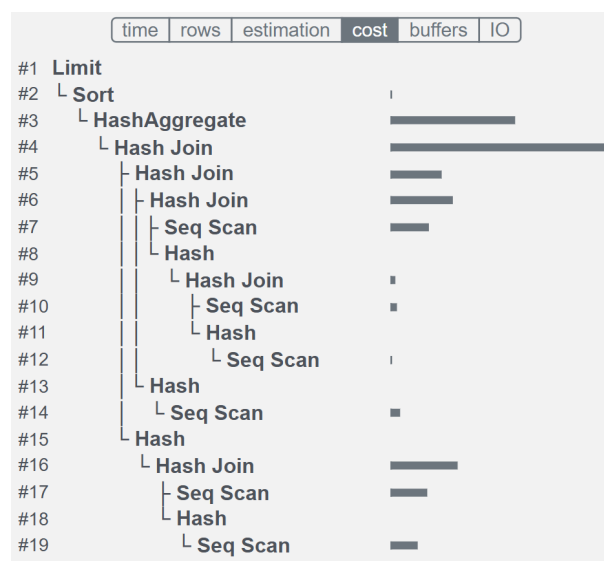


Figure 2.5: PEV2 diagram filtered by cost

TypeScript is the core language used to implement all the logic within PEV2. This includes parsing the execution plan JSON, managing the application state, and dynamically rendering the visual output. The site is structured and styled using HTML and CSS with the help of Bootstrap for responsive design. The user interface is primarily built using the progressive JavaScript framework Vue.js [6]. It allows for reactive data binding and component-based development, which helps organize the application into modular, maintainable pieces. The hierarchical tree of the execution plan—the main part of the website—is rendered graphically with node-based data using the JavaScript library D3.js [7]. It is used to create complex, interactive and scalable data visualizations using SVG, HTML and CSS. It also handles transitions and zoom functionality.

## 2.2 From PEV2 to DuckDB Explain Visualizer

### 2.2.1 PostgreSQL vs. DuckDB JSON Plan Structure

The JSON output of execution plan statements of PostgreSQL and DuckDB has four main differences:

- **Structure:**

**PostgreSQL:** "Plan" is the root node. The "Plan" object inside of it is the child execution node of the root node. The execution nodes have a flat structure. Each specific node has only one child node, so the plan has a more nested structure with more child nodes.

**DuckDB:** The first execution node goes directly into the root JSON object without an extra "Plan" node. The "children" array includes all the child nodes of a specific execution node. Therefore, it is flatter with more child nodes, but it has a more nested structure inside an execution node because of the "extra\_info" object. The nested structure of DuckDB JSON execution plans differs between the following statements: `EXPLAIN (FORMAT JSON)`, `EXPLAIN (ANALYZE, FORMAT JSON)` and the `PRAGMA` variant. This difference is shown in the example below (Listing 2.4, 2.5, 2.7).

- **Cost and Timing information:**

**PostgreSQL:** Has detailed cost estimation, like start-up and total cost. Separates planning and execution time. PostgreSQL does not store the actual operator timings in its plan, so PEV2 had to calculate the actual values.

**DuckDB:** Focus on the actual execution times. Uses simple timing values for each operation. No extra planning time and no cost information.

- **Additional information:**

**PostgreSQL:** Shows "Rows Removed by Filter" and more meta details like buffer hits, Write-Ahead Logging (WAL) records and Just-in-time (JIT) compiling.

**DuckDB:** It has a minimalist approach with only core information, for example, it shows only the filter condition without the number of rows removed.

- **Naming:**

**PostgreSQL:** Uses "Node Type" for execution nodes (execution operators).

**DuckDB:** Uses "name" for execution nodes (execution operators) with Statement `EXPLAIN (FORMAT JSON)`. With Statement `EXPLAIN (ANALYZE, FORMAT JSON)` and via the `PRAGMA` method the execution nodes has the "operator\_type" key that is equivalent to "name" and additionally the "operator\_name" key. The difference is displayed in Listing 2.4, 2.5 and 2.7.

To illustrate the aforementioned differences between PostgreSQL and DuckDB execution plans, consider the following simple SQL code:

```
1 CREATE TABLE t(  
2   x int,  
3   y int  
4 );  
  
5 INSERT INTO t(x,y) VALUES  
6 (1, 2),  
7 (4, 5),  
8 (8, 1),  
9 (42, 4711);  
  
10 EXPLAIN (FORMAT JSON)  
11 SELECT *  
12 FROM t  
13 WHERE x = 42;
```

Listing 2.1: Example query

Executing that SQL code with PostgreSQL (version 15) computes the following execution plan:

```
1 [  
2   {  
3     "Plan": {  
4       "Node Type": "Seq Scan",  
5       "Parallel Aware": false,  
6       "Async Capable": false,  
7       "Relation Name": "t",  
8       "Alias": "t",  
9       "Startup Cost": 0.00,  
10      "Total Cost": 38.25,  
11      "Plan Rows": 11,  
12      "Plan Width": 8,  
13      "Filter": "(x = 42)"  
14    }  
15  }  
16 ]
```

Listing 2.2: PostgreSQL query plan via EXPLAIN (FORMAT JSON)

With the option **ANALYZE**, the PostgreSQL execution plan has additional properties shown in Listing 2.3. For this particular query that includes actual timings, rows, loops, triggers and planning time.

```

1  [
2  {
3    "Plan": {
4      "Node Type": "Seq Scan",
5      "Parallel Aware": false,
6      "Async Capable": false,
7      "Relation Name": "t",
8      "Alias": "t",
9      "Startup Cost": 0.00,
10     "Total Cost": 38.25,
11     "Plan Rows": 11,
12     "Plan Width": 8,
13     "Actual Startup Time": 0.014,
14     "Actual Total Time": 0.014,
15     "Actual Rows": 1,
16     "Actual Loops": 1,
17     "Filter": "(x = 42)",
18     "Rows Removed by Filter": 3
19   },
20   "Planning Time": 0.106,
21   "Triggers": [
22   ],
23   "Execution Time": 0.026
24 }
25 ]

```

**Listing 2.3:** PostgreSQL query plan via EXPLAIN (ANALYZE, FORMAT JSON)

Executing the SQL Code in 2.1 with DuckDB (version 1.2.1) the execution plan looks like this:

```

1  [
2  {
3    "name": "SEQ_SCAN ",
4    "children": [],
5    "extra_info": {
6      "Table": "t",
7      "Type": "Sequential Scan",
8      "Projections": [
9        "x",
10       "y"
11     ],
12     "Filters": "x=42",
13     "Estimated Cardinality": "1"
14   }
15 }
16 ]

```

**Listing 2.4:** DuckDB query plan via EXPLAIN (FORMAT JSON)

We directly see the more nested structure inside an execution node. In this particular example, we only have one execution operator which represents a sequential scan. If we add the **ANALYZE** option to the **EXPLAIN** statement, we get the following plan:

```

1  {
2    "query_name": "",
3    "blocked_thread_time": 0.0,
4    "cpu_time": 0.0,
5    "extra_info": {},
6    "cumulative_cardinality": 0,
7    "cumulative_rows_scanned": 0,
8    "result_set_size": 0,
9    "latency": 0.0,
10   "rows_returned": 0,
11   "children": [
12     {
13       "cpu_time": 0.0,
14       "extra_info": {},
15       "cumulative_cardinality": 0,
16       "operator_type": "EXPLAIN_ANALYZE",
17       "operator_name": "EXPLAIN_ANALYZE",
18       "operator_cardinality": 0,
19       "cumulative_rows_scanned": 0,
20       "operator_rows_scanned": 0,
21       "operator_timing": 1e-7,
22       "result_set_size": 0,
23       "children": [
24         {
25           "cpu_time": 0.0,
26           "extra_info": {
27             "Table": "t",
28             "Type": "Sequential Scan",
29             "Projections": [
30               "x",
31               "y"
32             ],
33             "Filters": "x=42",
34             "Estimated Cardinality": "1"
35           },
36           "cumulative_cardinality": 0,
37           "operator_type": "TABLE_SCAN",
38           "operator_name": "SEQ_SCAN ",
39           "operator_cardinality": 1,
40           "cumulative_rows_scanned": 0,
41           "operator_rows_scanned": 4,
42           "operator_timing": 0.0000233,
43           "result_set_size": 8,
44           "children": []
45         }
46       ]
47     }
48   ]
49 }

```

Listing 2.5: DuckDB query plan via EXPLAIN (ANALYZE, FORMAT JSON)

We see that the **ANALYZE** option of DuckDB (Listing 2.5) adds two additional nodes. The actual execution plan is no longer the root node; instead, it is wrapped inside an operator named **EXPLAIN\_ANALYZE**, like the executed statement. This statement is inside another operator with no usable properties. I assume this is due to DuckDB's ongoing development and will be resolved in the future. However, the actual execution plan shows that there are a few more properties with the **ANALYZE** option, including operator result size (**result\_set\_size**), computed operator rows (**operator\_cardinality**), **operator\_timing** and **cpu\_time**. Additionally, there is another way to obtain an execution plan in DuckDB via the profiling pragma [8]. The updated example query for the profiling pragma method is shown in Listing 2.6 and the resulting execution plan is shown in Listing 2.7.

```

1 PRAGMA enable_profiling = 'json';
2 SELECT *
3 FROM t
4 WHERE x = 42;

```

Listing 2.6: Example query via profiling pragma

```

1 {
2   "query_name": "SELECT * FROM t WHERE x = 42;",
3   "blocked_thread_time": 0.0,
4   "cpu_time": 0.0000117,
5   "extra_info": {},
6   "cumulative_cardinality": 1,
7   "cumulative_rows_scanned": 4,
8   "result_set_size": 8,
9   "latency": 0.0004763,
10  "rows_returned": 1,
11  "children": [
12    {
13      "cpu_time": 0.0000117,
14      "extra_info": {
15        "Table": "t",
16        "Type": "Sequential Scan",
17        "Projections": [
18          "x",
19          "y"
20        ],
21        "Filters": "x=42",
22        "Estimated Cardinality": "1"
23      },
24      "cumulative_cardinality": 1,
25      "operator_type": "TABLE_SCAN",
26      "operator_name": "SEQ_SCAN ",
27      "operator_cardinality": 1,
28      "cumulative_rows_scanned": 4,
29      "operator_rows_scanned": 4,
30      "operator_timing": 0.0000117,
31      "result_set_size": 8,
32      "children": []

```

```

33     }
34   ]
35 }

```

**Listing 2.7:** DuckDB query plan via PRAGMA method

As we can see here, the unusable root node of the analyzed version of the `EXPLAIN` statement has been removed, leaving only usable information with the profiling pragma method. The root node now has a `query_name` field that is not empty and consists of the executed query. Additionally, all properties of the root node that were falsely set to 0 now have their real values, such as `latency`, `rows_returned`, `cpu_time`, etc. However, not only were the root node property values (overall plan statistics) falsely set to 0; the cumulative properties in the actual execution plan now have correct values as well. I assume that the profiling pragma method is the best way to get a detailed DuckDB execution plan. The `EXPLAIN (ANALYZE, FORMAT JSON)` statement does not provide correct information and has to be improved in the future.

### 2.2.2 Plan View: Node Construction

All key values in the JSON file need to be processed for node construction in the visualization, so we need to extract their values from the file to display them as metadata on the various plan nodes. Since this is already implemented for PostgreSQL in the forked PEV2 repository in `enums.ts`, I had to replace the PostgreSQL execution plan keys with the DuckDB execution plan keys shown in Table 2.1. The table lists all the JSON object keys required to read the JSON execution plan of DuckDB. Additionally, PEV2 JSON object keys are listed that correspond to the respective DuckDB keys. A “—” indicates that PEV2 does not have the specific key, as PostgreSQL JSON plans do not include it. The enum values are of the type string.

**Table 2.1:** DuckDB Explain Visualizer enums for JSON key/value processing

DuckDB enum key	DuckDB enum value	PEV2 enum value
QUERY	query_name	—
NODE_TYPE	operator_type	Node Type
NODE_NAME	operator_name	—
NODE_TYPE_EXPLAIN	name	Node Type
ACTUAL_ROWS	operator_cardinality	Actual Rows
ACTUAL_TIME	operator_timing	Actual Total Time
BLOCKED_THREAD_TIME	blocked_thread_time	—
PLANS	children	Plans
CPU_TIME	cpu_time	—
CUMULATIVE_CARDINALITY	cumulative_cardinality	—
CUMULATIVE_ROWS_SCANNED	cumulative_rows_scanned	—
OPERATOR_ROWS_SCANNED	operator_rows_scanned	—
RESULT_SET_SIZE	result_set_size	—
LATENCY	latency	—

DuckDB enum key	DuckDB enum value	PEV2 enum value
ROWS_RETURNED	rows_returned	—
EXTRA_INFO	extra_info	—
RELATION_NAME	Table	Relation Name
PROJECTIONS	Projections	Output
ESTIMATED_ROWS	Estimated Cardinality	Plan Rows
AGGREGATES	Aggregates	—
CTE_NAME	CTE Name	CTE Name
TABLE_INDEX	Table Index	Index Name
GROUPS	Groups	Group Key
JOIN_TYPE	Join Type	Join Type
CONDITIONS	Conditions	Hash Cond / Filter
CTE_INDEX	CTE Index	—
FILTER	Expression	Filter
DELIM_INDEX	Delim Index	—
FUNCTION	Function	Function Name
FUNCTION_NAME	Name	Function Name
NODE_ID	nodeId	nodeId
DEV_PLAN_TAG	plan_	plan_

Additionally, we want to give the various node attributes their expected type. In DuckDB execution plans there are only: **cardinality** (number of rows), **duration** and **size** (operator result size in bytes). This is stored in the following way:

```

1 export const nodePropTypes: { [key: string]: PropType } = {}
2
3 nodePropTypes[NodeProp.ACTUAL_ROWS] = PropType.rows
4 nodePropTypes[NodeProp.CUMULATIVE_CARDINALITY] = PropType.rows
5 nodePropTypes[NodeProp.CUMULATIVE_ROWS_SCANNED] = PropType.rows
6 nodePropTypes[NodeProp.OPERATOR_ROWS_SCANNED] = PropType.rows
7 nodePropTypes[NodeProp.CPU_TIME] = PropType.duration
8 nodePropTypes[NodeProp.BLOCKED_THREAD_TIME] = PropType.duration
9 nodePropTypes[NodeProp.RESULT_SET_SIZE] = PropType.bytes
10 nodePropTypes[NodeProp.ACTUAL_TIME] = PropType.duration
11 nodePropTypes[NodeProp.LATENCY] = PropType.duration
12 nodePropTypes[NodeProp.ROWS_RETURNED] = PropType.rows

```

Next, we need to customize the interfaces that will hold the information from the JSON file. PEV2 has the **interfaces** `IPlan`, `IPlanContent`, `IPlanStats`, and the `Node` **class**. I added a **interface** `ExtraInfo` to hold optional node properties. The customized version looks like this:

- **IPlan**: Represents a single query execution plan along with metadata. PEV2 has an extra `Node[]` to treat all CTE nodes differently from the other nodes. In DuckDB, it is a bit harder to treat CTE nodes differently, because of the different structure and properties. However, this is not really needed, so this field is removed. The new interface looks like this:

```

1 export interface IPlan {
2   id: string
3   name: string
4   content: IPlanContent
5   query: string
6   createdOn: Date
7   planStats: IPlanStats
8   formattedQuery?: string
9 }

```

- **IPlanContent:** Describes the structure and internal metrics of a query plan, so it contains the root node and all child nodes of the JSON file (remember section 2.2.1), the query text and maximum values (`maxRows`, `maxDuration`, etc.) of the plan. JIT compilation, cost values and other PostgreSQL statistics are removed. In the listing below, you can see `"Query Text": string`. This means that this property is optional because not every statement that generates an execution plan stores the query; only the **PRAGMA** method does this.

```

1 export interface IPlanContent {
2   "Query Text"?: string
3   _: Node
4   [NodeProp.PLANS]: Node[]
5   maxRows: number
6   maxRowsScanned: number
7   maxEstimatedRows: number
8   maxResult: number
9   maxDuration: number
10  [k: string]: Node | Node[] | number | string | undefined
11 }

```

- **IPlanStats:** Contains execution statistics and performance metrics, such as latency, execution time, and the maximum values for the entire plan.

```

1 export interface IPlanStats {
2   blockedThreadTime?: number
3   executionTime?: number
4   latency?: number
5   rowsReturned?: number
6   resultSize?: number
7   maxRows: number
8   maxRowsScanned: number
9   maxEstimatedRows: number
10  maxResult: number
11  maxDuration: number
12 }

```

- **ExtraInfo:** Holds optional properties for DuckDB execution plans. It represents all the properties that could be contained by the `extra_info` key in the JSON file, such as projection, join type, filter expression, relation name, function name, indexes, etc.
- **Node:** This class is used to create nodes when parsing the execution plan. These nodes will later represent the nodes in the visualization tree. Again, all PostgreSQL-specific node

properties are removed and DuckDB-specific properties are added.

After explaining the necessary enums and interfaces, I will move on to the actual node construction of the plan view. For this we have the **Vue.js** component `Plan.vue`. First, we need to customize the definition of the root node:

```
1 const rootNode = computed(() => {
2   if (plan.value!.content[NodeProp.CPU_TIME] !== undefined) {
3     // plan is analyzed
4     return plan.value && plan.value.content[NodeProp.PLANS][0]
5   } else {
6     // plan is not analyzed
7     return plan.value && (plan.value.content as unknown as Node)
8   }
9 })
```

Here we have to distinguish between the analyzed and the non-analyzed version of the plan and set the root node according to the statement used. Here I just ask if there is a `cpu_time` in the root JSON object, because only the analyzed plan has this particular key. In the analyzed version, we just want to skip the root object and store only the child of the root object, because as I mentioned earlier, the root object is redundant here.

After that, we need to modify the `onBeforeMount` function to initialize the parsing of the plan. This involves setting up the construction of the tree graph and assigning the correct values to all DuckDB-specific plan statistic values that I defined in the interfaces. The new function is shown in Listing 2.8. The `planJson` is an `IPlanContent` assigned to the parsed JSON plan. To accomplish correct parsing, we call the `fromSource` function (Listing 2.9) of the `plan-service.ts` file which is responsible for sanitizing and preparing the input execution plan data before it is processed and visualized. This function ensures that the data conforms to the expected structure and format, removing any extraneous or malformed elements that might interfere with the visualization process and then parses the plan with the `parseJson` function which uses the `Clarinet` JSON parser and did not require any changes from the `PEV2` repository.

```
1 onBeforeMount(() => {
2   ...
3   let planJson: IPlanContent
4   try {
5     planJson = planService.fromSource(
6       props.planSource
7     ) as unknown as IPlanContent
8     parsed.value = true
9     setActiveTab("plan")
10  } catch (e) {
11    parsed.value = false
12    plan.value = undefined
13    return
14  }
15  queryText.value = (planJson[NodeProp.QUERY] as string) || props.planQuery
16  plan.value = planService.createPlan("", planJson, queryText.value)
17  const content = plan.value.content
```

```

18   planStats.blockedThreadTime =
19     (planJson[NodeProp.BLOCKED_THREAD_TIME] as number) ?? NaN
20   planStats.executionTime = (planJson[NodeProp.CPU_TIME] as number) ?? 0
21   planStats.latency = (planJson[NodeProp.LATENCY] as number) ?? NaN
22   planStats.rowsReturned = (planJson[NodeProp.ROWS_RETURNED] as number) ?? NaN
23   planStats.resultSize = (planJson[NodeProp.RESULT_SET_SIZE] as number) ?? NaN
24   planStats.maxRows = content.maxRows ?? NaN
25   planStats.maxRowsScanned = content.maxRowsScanned ?? NaN
26   planStats.maxResult = content.maxResult ?? NaN
27   planStats.maxEstimatedRows = content.maxEstimatedRows ?? NaN
28   planStats.maxDuration = content.maxDuration ?? NaN
29   plan.value.planStats = planStats
30
31   ...
32 })

```

Listing 2.8: New onBeforeMount function

```

1 public fromSource(source: string) {
2   source = this.cleanupSource(source)
3   return this.parseJson(source)
4 }

```

Listing 2.9: Cleaning source and parsing

The `createPlan` function of `plan-service.ts` defines the `IPlan` interface, which all website views use to display plan properties. Additionally, it calculates the maximum statistic values defined for the plan, such as `maxRows`—what it exactly means is mentioned later—and gives each execution node a unique identifier `id`, to differentiate between the nodes and to add the correct properties to the specific nodes in the tree graph. The exact process is shown in Listing 2.10.

```

1 public createPlan(
2   planName: string,
3   planContent: IPlanContent,
4   planQuery: string
5 ): IPlan {
6   ...
7   if (planContent) {
8     const plan: IPlan = {
9       id: NodeProp.DEV_PLAN_TAG + new Date().getTime().toString(),
10      name: planName || "plan created on " + new Date().toDatestring(),
11      createdOn: new Date(),
12      content: planContent,
13      query: planQuery,
14      planStats: {} as IPlanStats,
15    }
16    this.nodeId = 1
17    if (planContent[NodeProp.CPU_TIME] !== undefined) {
18      // plan is analyzed
19      this.processNode(planContent[NodeProp.PLANS]![0]!, plan)
20    } else {

```

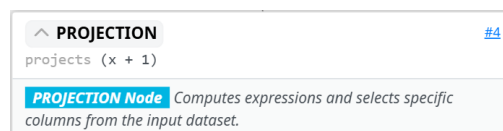
```

21     // plan is not analyzed
22     this.processNode(planContent as unknown as Node, plan)
23   }
24   this.calculateMaximums(plan)
25   return plan
26 } else {
27   throw new Error("Invalid plan")
28 }
29 }
30
31 // recursively walk down the plan
32 public processNode(node: Node, plan: IPlan) {
33   node.nodeId = this.nodeId++
34
35   _each(node[NodeProp.PLANS], (child) => {
36     this.processNode(child, plan)
37   })
38 }

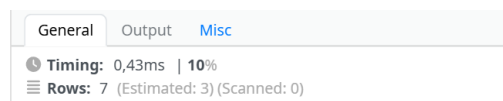
```

Listing 2.10: Creating plan

Now, it already works to parse a DuckDB execution plan and display the plan tree. But now, we need to customize the specific node visualization in the graph. To do this, we need to change the HTML code of the `PlanNode.vue` component. Here we replace all PostgreSQL specific properties with DuckDB specific properties. This component displays the execution **operator type**, the node **id** and the execution node specific descriptions defined in `help-service.ts` (Listing 2.11) as well as some information: the **relation name**, if there is a **projection of columns** or a **filter expression**, other **conditions**, the **join type**, an executed **function** or the **CTE name**. An example is shown in the following illustration:



Once this is set, we need to look at the `PlanNodeDetail.vue` component, which displays all the other node-specific properties of the execution plan, divided into several tabs. In PEV2, these tabs are **General**, **IO & Buffers**, **Output**, **Workers**, and **Misc** (Figure 2.3, 2.4). For DuckDB plans, we only need **General**, **Output** and **Misc** because the other tabs would not contain anything due to non-existent property values in the DuckDB execution plans. So again, I replaced all PostgreSQL specific properties with the DuckDB specific properties. The **General** tab consists only of the time and row values:



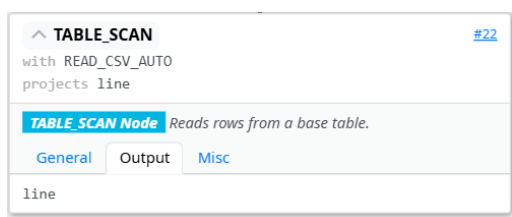
The percentage next to the **Timing** number represents the timing of the specific node in relation to the maximum timing in the entire plan. This is calculated by the function in `node.ts`:

```

1 function calculateDuration() {
2   const executionTime = (plan.value.planStats.executionTime as number) ||
3     (plan.value.content?.[NodeProp.CPU_TIME] as number)
4   const duration = node[NodeProp.ACTUAL_TIME] as number
5   executionTimePercent.value = _.round((duration / executionTime) * 100)
6 }

```

The **Output** tab shows only the projected columns (attributes), if any, or the columns on which an aggregate function has been applied:



The **Misc** tab shows all other property values of the specific execution node:

General	Output	Misc
cpu_time		0,43ms
cumulative_cardinality		7
result_set_size		110 Bytes
Groups		#1,#0,#3,#2

```

1 export const NODE_DESCRIPTIONS: INodeDescription = {
2   NESTED_LOOP_JOIN: `Joins two tables using a nested loop.`,
3   MERGE_JOIN:      `Performs a join by first sorting both tables on the join key
4                     and then merging them efficiently.`,
5   HASH_GROUP_BY:  `Is a group-by and aggregate implementation that uses a hash
6                     table to perform the grouping.`,
7   ...
8 }

```

Listing 2.11: Plan node descriptions

The last thing to do for the tree graph is to visualize the node progress bar of the statistic values for each operator, shown in Figure 2.8. In that figure we can see, that the progress bar is filtered by the amount of calculated rows. In PEV2 we can filter between **none**, **duration**, **rows** and **cost** (Figure 2.6). In DuckDB, we do not have the cost property, but instead, the **result\_size** property. Since the duration filtering is already computed by the `calculateDuration` function shown earlier, we need to do the same for the already existing `calculateRows` function and replace the `calculateCosts` function with a `calculateResult` function. Using the `calculateRows` function as an example, I will demonstrate how the progress bar is calculated. First, the `maxRows` variable is read from `planStats`. This variable contains the highest `operator_cardinality` value

in the entire plan. So, the maximum number of rows calculated by a single operator is stored in `maxRows`. Therefore, it is not the sum of all rows calculated by all operators, but rather the highest value calculated by any one operator in the entire plan. For the `calculateRows` function, the `operator_cardinality` of a specific node is divided by the `maxRows` value and multiplied by 100 to get the percentage. The same applies to the other filter options (duration and result size). The two new functions are shown in Listing 2.12.

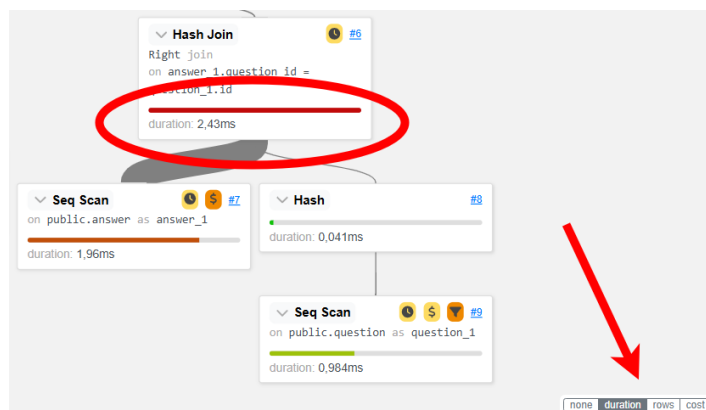


Figure 2.6: PEV2 statistic filtering: duration

```

1 function calculateResult() {
2   const maxResult = plan.value.content.maxResult as number
3   const result = node[NodeProp.RESULT_SET_SIZE] as number
4   resultPercent.value = _.round((result / maxResult) * 100)
5 }
6
7
8
9 function calculateRows() {
10  const maxRows = plan.value.content.maxRows as number
11  const rows = node[NodeProp.ACTUAL_ROWS] as number
12  rowsPercent.value = _.round((rows / maxRows) * 100)
13 }

```

Listing 2.12: New statistic filtering functions

Additionally, in the `NodeBadges.vue` component, I set a node badge for large percentages of an execution nodes result size property to visualize, without filtering, that this particular execution node has a large result size compared to all other execution nodes. The node badges for the duration and rows properties are already implemented. It depends on the percentage if the badge is red, yellow or not displayed. Red means over 90 percent, yellow over 50 percent and for the lower percentages no badge is displayed. The same color mapping applies to the previously mentioned node progress bar. The only difference is that the progress bar is green for percentages under 50 percent (Figure 2.7).

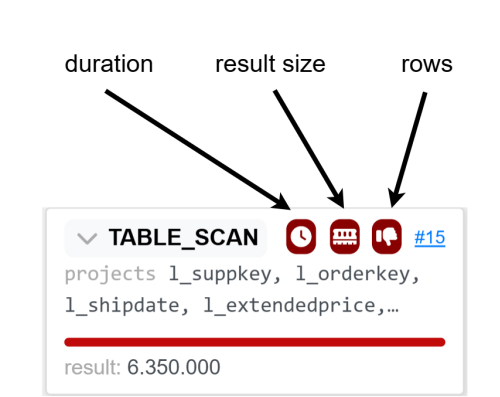


Figure 2.7: DuckDB Explain Visualizer node badges

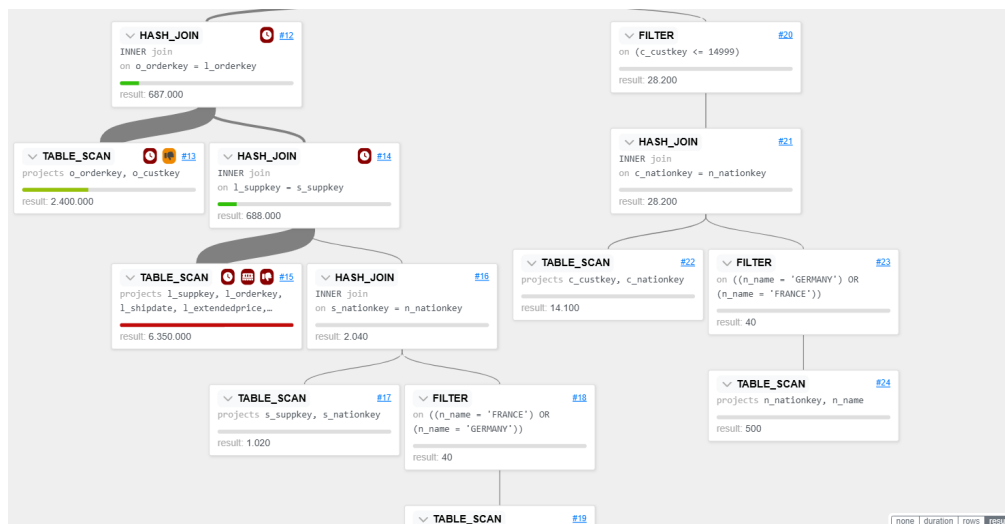


Figure 2.8: DuckDB Explain Visualizer statistic filtering: result size

### 2.2.3 Plan View: Diagram

As mentioned earlier, the diagram is on the left side of the plan view, next to the actual tree graph (PEV2: Figure 2.5). In order to correctly display the hierarchical structure of the plan, I had to adapt the root node as I did with the node construction of the tree graph:

```

1  onBeforeMount(): void => {
2    const savedOptions = localStorage.getItem("diagramViewOptions")
3    if (savedOptions) {
4      _.assignIn(viewOptions, JSON.parse(savedOptions))
5    }
6    if (plan.value.content[NodeProp.CPU_TIME] !== undefined) {
7      // plan is analyzed
8      flatten(plans[0], 0, plan.value.content[NodeProp.PLANS][0], true, [])
9    } else {
10     // plan is not analyzed
11     flatten(plans[0], 0, plan.value.content as unknown as Node, true, [])
12   }
13 }

```

The only difference between the tree graph and this one is that the execution node structure has been flattened, so all properties inside the `extra_info` object are now in the execution node with the other properties specific to that node. This makes accessing the extra info properties easier.

Next, I adapted the possible filtering values. PEV2 allows switching between the statistic values `time`, `rows`, `estimation`, `cost`, `buffers` and `IO`. Since DuckDB execution plans only have `time`, `rows` and `result size` as different types I replaced the PEV2 values with them (Figure 2.9) and used the same computations as in the functions mentioned in the previous section to display the different percentages (`calculateDuration`, `calculateRows` and `calculateResult`), but this time the calculations are placed directly in the HTML template code of the `Diagram.vue` component.

```

1  function getTooltipContent(node: Node): string {
2    let content = ""
3    switch (diagramViewOptions.metric) {
4      case Metric.time:
5        content += timeTooltip.value
6        break
7      case Metric.rows:
8        content += rowsTooltip.values
9        break
10     case Metric.result:
11       content += resultTooltip.value
12       break
13   }
14   if (node[NodeProp.EXTRA_INFO][NodeProp.CTE_NAME]) {
15     content += "<br><em>CTE " + node[NodeProp.EXTRA_INFO][NodeProp.CTE_NAME] + "</em>"
16   }
17   return content
18 }

```

Listing 2.13: Tooltip for statistic values

To display a node's specific value, the visualizer shows a tooltip with the function `getTooltipContent` when the user hovers over the node (Listing 2.13). The specific tooltip variable is a string that displays the value's type and the actual metric. An example is shown in

Figure 2.9. Additionally, the CTE name is displayed within the tooltip when the specific node is a CTE node.

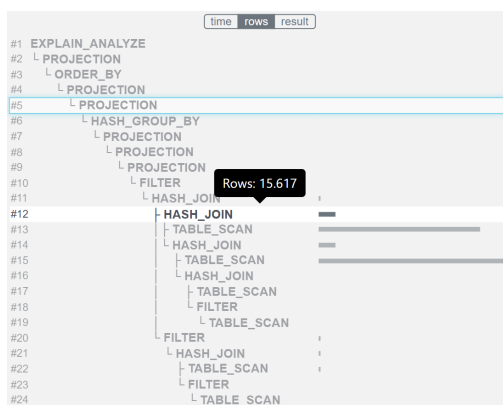


Figure 2.9: DuckDB diagram filtered by rows

## 2.2.4 Grid View

The grid view is an alternative to the plan view, which is less visual and more structured. Like the diagram, it is a hierarchical structure, but it has additional information, such as node details, which are shown when a specific node is clicked. The statistic values are always displayed on the left side of the hierarchy. The **estim** column displays the estimated rows of the current execution node of the JSON plan (key: **Estimated Cardinality**) and the arrow after the specific value shows whether the estimated rows were underestimated, overestimated or correctly estimated. This view is shown in Figure 2.10. By clicking on the node id on the left side—denoted as **#[id-number]**—the visualizer opens the plan view directly and marks the specific node. The same applies to the diagram and stats view.

	time	rows	estim	result	filter
#1	0,42ms	1		8	UNGROUPED_AGGREGATE
L priority_queue					
#2	1,77s	124.300	2.110.000		RECURSIVE_CTE CTE priority_queue using 161
#3	0ms	1		17	PROJECTION
#4	0ms	1		4	DUMMY_SCAN
#5	3,48ms	124.299	7.1698	2.110.000	PROJECTION
PROJECTION Node Computes expressions and selects specific columns from the input dataset.					
Misc Output					
		cpu_time	2,32s		
		cumulative_cardinality	2.908.336		
		cumulative_rows_scanned	3.114.930		
		result_set_size	2 MB		
L best					
#6	0,95ms	0		0	CTE CTE best using 169
#7	0,75ms	455	1	7.740	PROJECTION
#8	01,97ms	455	1	7.280	HASH_GROUP_BY
#9	0,38ms	21.015	0	588.000	PROJECTION
#10	7,23ms	21.015	0	357.000	FILTER
#11	1,13ms	124.300	0	2.110.000	RECURSIVE_CTE_SCAN using 161

Figure 2.10: DuckDB grid view

## 2.2.5 Stats View

The last view is the stats view. It groups the execution nodes in four different tables: **Table**, **Function**, **Node Type** and **CTE**. To display all nodes correctly, we have to define the root node just like in the grid and diagram view with one additional line:

```
1 executionTime.value =
2   plan.value.planStats.executionTime ||
3   (plan.value.content?.[NodeProp.ACTUAL_TIME] as number)
```

This is important because the stats view orders the nodes by execution time and count. An example is shown in Figure 2.11. I will explain the table construction by the table **Function**, shown below. To only extract the function nodes it iterates through all nodes and only store the ones which has the specific function name defined in the JSON plan node. After that, all table rows have to be created and displayed, which is achieved by the `values` variable which is a `StatsTableItemType` array. In that array, all function nodes are pushed into it with all information needed for the `StatsTableItem.vue` component to create the specific table rows with the information: function **name**, **count**, **time** and **timePercent** (Listing 2.15). The other tables are created simultaneously to the function table.

```
1 const perFunction = computed(() => {
2   const functions: { [key: string]: Node[] } = _.groupBy(
3     _.filter(nodes, (n) => n[NodeProp.FUNCTION_NAME] !== undefined),
4     NodeProp.FUNCTION_NAME
5   )
6   const values: StatsTableItemType[] = []
7   _.each(functions, (nodes, functionName) => {
8     values.push({
9       name: functionName,
10      count: nodes.length,
11      time: _.sumBy(nodes, NodeProp.ACTUAL_TIME),
12      timePercent: durationPercent(nodes),
13      nodes,
14    })
15  })
16  return values
17 })
```

Listing 2.14: Table row creation

```
1 function durationPercent(nodes: Node[]) {
2   return _.sumBy(nodes, NodeProp.ACTUAL_TIME) / executionTime.value
3 }
```

Listing 2.15: Calculating the duration percent column

Node Type	Count	Time	Time %
▼ WINDOW	2	23,91ms	57%
#16 WINDOW		18,62ms	45%
#11 WINDOW		5,29ms	13%
> RIGHT_DELIM_JOIN	1	5,08ms	12%
> HASH_GROUP_BY	2	5ms	12%
> HASH_JOIN	1	4,46ms	11%
> UNNEST	1	0,99ms	2%
> INOUT_FUNCTION	1	0,86ms	2%
> PROJECTION	16	0,59ms	1%
> FILTER	2	0,41ms	1%
> INSERT	1	0,24ms	1%
> CROSS_PRODUCT	1	0,18ms	0%
> TABLE_SCAN	1	0,06ms	0%
> UNGROUPED_AGGREGATE	1	0,01ms	0%
> STREAMING_WINDOW	1	0ms	0%
> DUMMY_SCAN	2	0ms	0%
> DELIM_SCAN	1	0ms	0%

Figure 2.11: DuckDB stats view: Node Type table

## 2.2.6 Plan Sharing Service

The plan sharing service of the DuckDB Explain Visualizer is based on the sharing service *explain.dalibo.com* of PEV2. The sharing service is implemented as a Python web application using the Flask framework. It allows users to upload and share execution plans via unique URLs. The Flask app receives execution plans in JSON format through a web form or API, stores them in a database and renders them using the Vue-based PEV2 library. The core functionality includes: plan submission (via POST), plan viewing (via UUID and GET request) and the interactive plan visualization embedded in a web UI.

For DuckDB integration, the original application was forked and modified in several ways:

- **Color Scheme:** The default PEV2 color palette was replaced with DuckDB's signature yellow and black, done by the Database Research Group.
- **Texts and Titles:** All page titles, labels, and explanatory texts were rewritten to reference DuckDB instead of PostgreSQL.
- **Example Plans:** Sample PostgreSQL execution plans were replaced with DuckDB-specific plans.
- **Visualization Component:** The key frontend component—responsible for rendering the plan view—was replaced. In `plan.js`, the original component was replaced with the DuckDB Explain Visualizer component as follows:

```
1 const app = createApp({
2   setup() {
3     const plan = ref(planData);
4     return { plan };
5   },
6   components: {
7     "duckdb-explain-visualizer": Plan,
8   },
9 });
```

- **Data Storage:** Submitted execution plans are stored using the SQLAlchemy Object Relational Mapper, backed by a PostgreSQL database on the server side. If you want to read more about SQLAlchemy, visit [9].

## Results

The first visualization of the following is produced by the in-terminal representation of DuckDB and the second visualization is produced by the DuckDB Explain Visualizer while processing the same query as the in-terminal graph visualization. The processed query is shown in Listing 1.1 in the introduction:



Figure 3.1: In-terminal representation

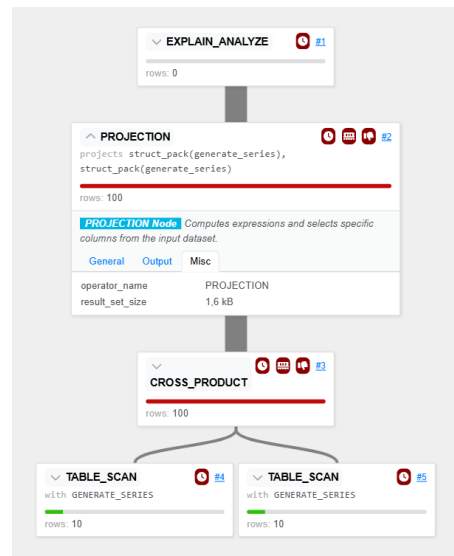


Figure 3.2: DuckDB Explain Visualizer

### 3.1 How to use the DuckDB Explain Visualizer

The DuckDB Explain Visualizer is available here: <https://db.cs.uni-tuebingen.de/explain>.

In order to use the DuckDB Explain Visualizer, you will need a DuckDB query plan in JSON format. You can obtain one by running a query prefixed with **EXPLAIN (FORMAT JSON)**. Then, paste the JSON output into the Visualizer to view the query plan. You can also drag and drop a file into the text area.

```

EXPLAIN (ANALYZE, FORMAT JSON)
[ your-query-here ];

```

Listing 3.1: Get JSON plan via EXPLAIN

You can use the **ANALYZE** option to include runtime statistics in the query plan (Listing 3.1) or you can use the **PRAGMA** method (Listing 3.2). This is useful for long-running queries or queries that you want to analyze later. The output file can then be uploaded to the DuckDB Explain Visualizer. The following commands enable profiling and save the output to a file [3]:

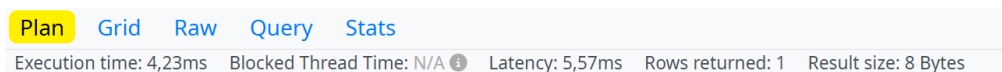
```
PRAGMA enable_profiling = 'json';
PRAGMA profiling_output = '/path/to/file.json'; -- (optional)
[your-query-here];
```

Listing 3.2: Get plan via PRAGMA

## 3.2 Features

One of the DuckDB Explain Visualizer’s main features is its ability to easily share query plans. After uploading a JSON query plan, the tool generates a shareable link that allows others to access and view the plan directly, eliminating the need for re-uploading. This feature is useful for teaching, debugging, and collaboration, where clear communication of query behavior is essential. The tool also displays the raw JSON query plan, providing users with transparency into the plan’s structure and content. It also shows the SQL query that was executed, if it is included in the JSON plan. This enables users to connect the visualized plan directly to the query that produced it.

Another feature is the integration of plan statistics. These statistics are shown within the plan and grid view as an additional row at the top of both views shown in Figure 3.3. This gives users a quick overview of important metrics, such as execution time, row counts, and cost estimates. This enhances their understanding of the performance characteristics of different parts of the query.



Plan	Grid	Raw	Query	Stats
Execution time: 4,23ms	Blocked Thread Time: N/A ⓘ	Latency: 5,57ms	Rows returned: 1	Result size: 8 Bytes

Figure 3.3: General plan statistics

## 3.3 Future Work

There are several potential improvements to the DuckDB Explain Visualizer. One improvement would be to highlight execution nodes in the tree graph that refer to other nodes through indices such as **CTE Index** or **Table Index**. This would help users quickly identify critical performance elements of the plan. Another enhancement would be to provide better support for common table expressions (CTEs), such as the ability to visually represent and isolate their subparts for improved clarity. For example, all CTE computations could be placed as highlighted sub-trees, instantly showing which part of the execution plan is computed by a CTE. Additionally,

automatically recognizing operator types and SQL keywords in the query plan would improve its readability and navigability as well as responsibility to new adaptations in the execution plan structure and naming due to improvements of the DuckDB team. These features would enhance the tool's usability and value in educational and professional settings.



Used AI tools	Version
DeepL write	25.5.1

**Table 1:** Used AI tools



## Bibliography

---

- [1] N/A. *DBMS and SQL Basics Explained*. Accessed: 2025-04-25. 2023. URL: <https://www.freecodecamp.org/news/dbms-and-sql-basics/>.
- [2] DigitalOcean Community. *SQLite vs MySQL vs PostgreSQL: A Comparison of Relational Database Management Systems*. Accessed: 2025-04-25. 2022. URL: <https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems>.
- [3] Denis Hirn. *The DuckDB Explain Visualizer is now available!* <https://db.cs.uni-tuebingen.de/news/duckdb-explain-visualizer/>. Accessed: 2025-04-29. 2025.
- [4] Dalibo. *PEV2 - Query Plan Visualizer*. <https://github.com/dalibo/pev2>. Accessed: 2025-04-25. 2023.
- [5] Alex Tatiyants. *PEV: PostgreSQL Explain Visualizer*. <https://github.com/AlexTatiyants/pev>. Accessed: 2025-05-30. 2013. URL: <https://github.com/AlexTatiyants/pev>.
- [6] Vue.js Team. *Vue.js - The Progressive JavaScript Framework*. Accessed: 2025-04-29. 2024. URL: <https://vuejs.org>.
- [7] Mike Bostock. *What is D3? - D3.js*. Accessed: 2025-04-29. 2024. URL: <https://d3js.org/what-is-d3>.
- [8] DuckDB Developers. *DuckDB Documentation: Profiling Pragmas*. Accessed: 2025-04-30. 2024. URL: <https://duckdb.org/docs/stable/configuration/pragmas.html#profiling>.
- [9] Mike Bayer. *SQLAlchemy: The Database Toolkit for Python*. Version 2.0. SQLAlchemy Project. 2025. URL: <https://www.sqlalchemy.org>.