Jovis: A Visualization Tool for PostgreSQL Query Optimizer

Yoojin Choi
Seoul National University
Seoul, Republic of Korea
cyj@dbs.snu.ac.kr

Juhee Han
Seoul National University
Seoul, Republic of Korea
juheehan@dbs.snu.ac.kr

Kyoseung Koo
Seoul National University
Seoul, Republic of Korea
koo@dbs.snu.ac.kr

Bongki Moon
Seoul National University
Seoul, Republic of Korea
bkmoon@snu.ac.kr

Abstract

Query optimizers are essential components of relational database management systems that directly impact query performance as they transform input queries into efficient execution plans. While users can examine the final plan using the `EXPLAIN` command and existing visualization tools, the internal decision-making process of the optimizer is hidden from users, making it difficult to understand how the plan is constructed. To address this challenge, we present *Jovis*, an interactive visualization tool for exploring the query optimization process in PostgreSQL. *Jovis* provides a comprehensive view of the entire optimization workflow through tailored visualization for each optimization strategy. It further enables users to participate in optimization by providing hints, tuning parameters, and reusing prior optimization results. *Jovis* serves both as an educational tool for learners and a practical resource for database professionals, supporting users in understanding and guiding the optimizer toward better decisions. The source code, data, and/or other artifacts are available at https://github.com/orgs/snu-jovis.

CCS Concepts

• Information systems → Data management systems; **• Human-centered computing → Visualization systems and tools**.

Keywords

Query Optimization, Interactive Visualization, PostgreSQL, Database Management Systems

ACM Reference Format:
Yoojin Choi, Juhee Han, Kyoseung Koo, and Bongki Moon. 2025. Jovis: A Visualization Tool for PostgreSQL Query Optimizer. In *The International Conference on Scalable Scientific Data Management 2025 (SSDBM 2025), June 23–25, 2025, Columbus, OH, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3733723.3733744

1 Introduction

Query optimizers are a core component of relational database management systems (RDBMS) as they play a key role in determining query performance. They transform declarative SQL queries into efficient execution plans by exploring a vast search space and selecting the most optimal plan. At the same time, query optimization is one of the most challenging aspects of DBMS design, primarily due to the NP-hard nature of finding an optimal join order [8] and the complexity of choosing the best physical operators for each logical operation. These challenges have driven decades of research, from the traditional cost-based optimizers such as System R [18] to recent machine learning-based approaches [10, 22].

Understanding the decision-making process of the query optimizer greatly benefits both learners and practitioners. For learners, it provides a grasp of query optimization beyond high-level execution plans, such as how join orders and physical operators are determined. For practitioners, deeper insight into how plans are generated, including considered alternatives, cost estimations, and the rationale behind the final plan, helps explain suboptimal decisions and enables manual tuning through hints or configuration adjustments. Such transparency is also essential for extending or improving the optimizer, as it clarifies its underlying strategies and implementation.

Despite its importance, the query optimization process remains largely opaque to users unless they delve into the source code. While DBMSs like PostgreSQL [21] and MySQL [13] provide `EXPLAIN` and `ANALYZE` commands to display the final execution plan and its estimated cost, along with external tools PEV2 [4] and MySQL Workbench [14] that offer graphical representations of the plan, these outputs only reflect the outcome of the optimization process rather than the reasoning behind it. Several research efforts have aimed to make query optimization more accessible through visualization and interactive features. QO-Insight [1] focuses on visualizing query execution traces of steered query optimizers, which use hints to correct planning mistakes, and MOCHA [19] enables users to analyze the impact of alternative physical operator choices. While these works visualize and help compare the query execution plans, they fall short of providing a comprehensive view of the internal decision-making process of query optimizers.

In this demonstration, we present *Jovis*, an interactive visualization tool designed to reveal the query optimization process of PostgreSQL. Unlike existing tools that focus solely on the final execution plan, *Jovis* provides visibility into *how* the optimizer constructs the plan. It visualizes the explored search space, the evaluated join orders and physical operators, and the cost estimations that drive plan selection. Beyond visualization, *Jovis* introduces

SSDBM 2025, Columbus, OH, USA

ACM ISBN 979-8-4007-1462-7/25/06
https://doi.org/10.1145/3733723.3733744

user-guided optimization, allowing users to steer the optimizer toward better decisions or explore untapped search spaces. By making the optimizer transparent and interactive, *Jovis* not only bridges the gap between high-level execution plans and the underlying optimization logic but also helps users diagnose suboptimal choices and improve query performance.

Key features of *Jovis* include:

- Comprehensive visualizations of PostgreSQL's optimization process, covering both its cost-based optimizer using dynamic programming and genetic algorithm.
- In-depth view of the cost model, including cost estimation formulas that illustrate how costs impact plan selection.
- User-guided optimization, an interactive feature that allows users to guide the optimizer through hints, configuration adjustments, and leveraging prior optimization results.

We chose PostgreSQL as our starting point due to its wide adoption and two complementary optimization approaches. Our tool is designed to be extensible, with future work planned to support additional DBMSs.

2 Related Work

Visualization of Query Optimization. Several systems address specific challenges in visualizing query optimization. Picasso [7] provides graphical profiling and analysis of query optimizer behavior. QE3D [17] introduces a three-dimensional representation of distributed query plans, helping users identify key performance aspects. DBinsight [16] visualizes the query processing pipeline and unifies heterogeneous data structures across RDBMSs through a common interface. MOCHA [19] explores the impact of physical operator selection by allowing users to compare alternative plans and observe corresponding performance changes. QO-Insight [1] offers a visual inspection of steered query optimizers that use hints to address planning mistakes.

A variety of visualization tools for query optimizers have been developed. Postgres Explain Visualizer (PEV) [20] and its successor PEV2 [4] render `EXPLAIN` output into interactive trees, enabling users to analyze the structure and estimated costs of execution plans. Similarly, MySQL Workbench [14] provides graphical plan visualization for MySQL as part of a broader suite of database modeling and administration tools.

PostgreSQL Query Optimizer. In PostgreSQL, the optimizer examines possible query plans by generating data structures called *paths*, which represent partial or complete plans along with their estimated costs. The standard optimizer builds these paths by enumerating possible plans using dynamic programming [11] and evaluates them with a cost model to determine the cheapest path. However, this approach becomes infeasible for queries involving a large number of joins, as exploring the exponentially growing search space requires excessive time and memory. In such cases, PostgreSQL switches to the Genetic Query Optimizer (GEQO), which applies a genetic algorithm to iteratively generate and refine join sequences over successive generations using recombination mechanisms, balancing optimization quality with computational efficiency [6].

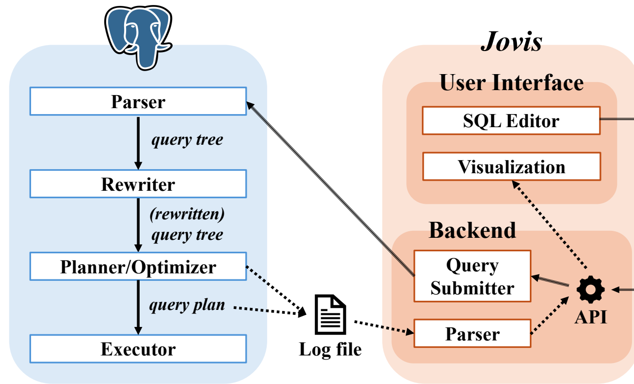


Figure 1: Architecture of *Jovis*

3 *Jovis*

This section provides an overview of *Jovis* architecture, as illustrated in Figure 1. *Jovis* consists of three main components: a patched version of PostgreSQL, an interactive user interface, and a backend. Users submit queries through the frontend's SQL editor, which are transmitted to PostgreSQL via the backend. PostgreSQL processes the queries and generates detailed optimizer logs, which the backend collects and parses into structured JSON format. The processed data is then sent to the frontend for visualization through the interactive interface. We currently provide patches for both PostgreSQL 16 and 17 to add detailed logging of the optimizer's internal decisions, including paths and costs. Users can apply the patch corresponding to their PostgreSQL version.

3.1 Graphical User Interface

Figure 2 represents the graphical user interface (GUI) designed to offer an intuitive and interactive experience for users to explore and understand the query optimization process. Built with React and D3.js [2], the interface leverages data-driven visualizations to present complex optimization details in an accessible manner.

Sidebar A provides a flexible interface for managing query planning visualizations. **Queries** allows users to select, deselect, or delete submitted queries, supporting side-by-side comparison of multiple optimizations. This helps users analyze how different queries or configurations affect the optimizer's decisions. **History** maintains a record of previously executed queries, enabling quick reuse and reducing repetitive input. **Presets** includes predefined queries from widely used benchmarks, TPC-H [15], TPC-DS [12], and Join Order Benchmark (JOB) [9]. It simplifies experimentation and performance evaluation. **SQL editor** B allows users to submit queries, select a database, and execute them for analysis. For both query planning views C and D, users can switch to the `EXPLAIN` tab to view the `EXPLAIN` output as a tree, as presented in Figure 3.

3.1.1 Standard Optimizer. **Query Planning View** C visualizes PostgreSQL's standard optimization process using dynamic programming. The optimizer adopts a bottom-up approach, starting by determining access paths for each base relation. It then incrementally constructs join sequences and selects physical operators until the complete execution plan is achieved.

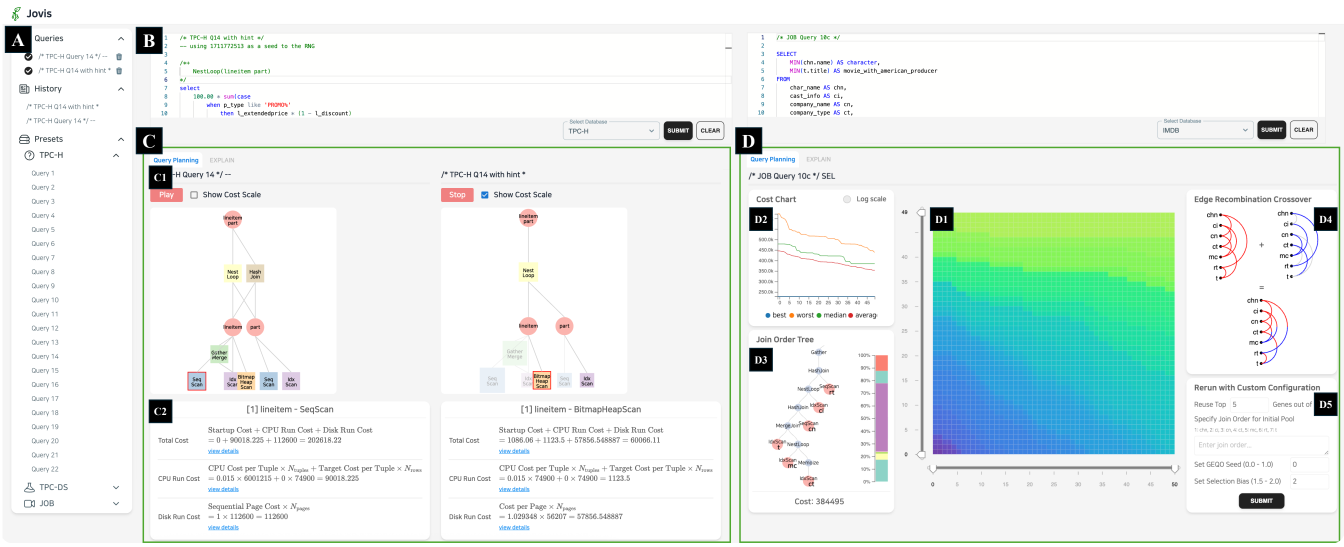


Figure 2: Graphical User Interface of *Jovis*

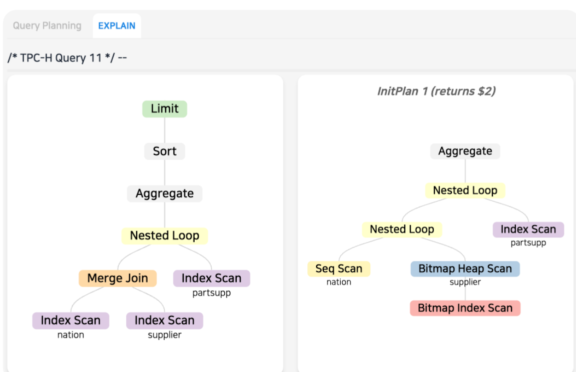


Figure 3: Visualization of `EXPLAIN` for TPC-H Query 11

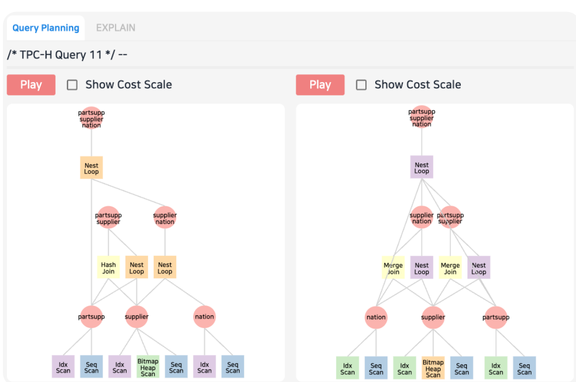


Figure 4: Subquery Support in TPC-H Query 11

To effectively depict the process, *Jovis* employs a directed acyclic graph (DAG), which is well-suited for representing the bottom-up

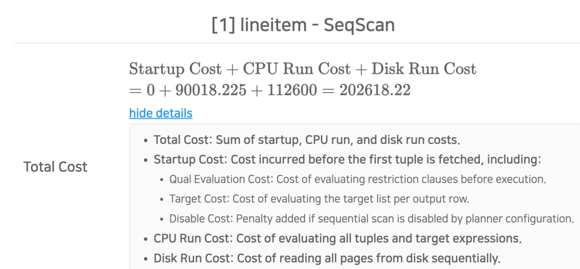
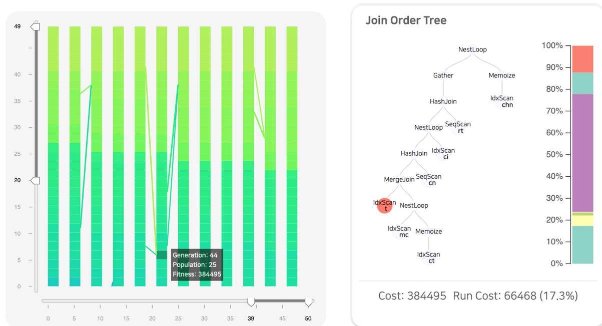


Figure 5: Explanations of Cost Parameters

strategy due to its ability to handle hierarchical relationships and multiple parent nodes. In this structure, circular nodes represent relations or join sequences, while rectangular nodes represent physical operators considered during plan enumeration. Edges indicate the application of an operator to its input relations or join orders. The DAG is displayed in an inverted layout, where operators appear as children of their associated relations. For example, the left DAG in C1 shows multiple scan methods evaluated for each relation, and nested loop join and hash join were explored for joining two relations. As shown in Figure 4 for TPC-H Query 11, *Jovis* also supports subqueries by generating separate DAGs for each one, enabling modular inspection of complex queries.

Users can interact with the DAG through several features. **Play / Stop button** in C1 animates the optimization process, reducing the opacity of non-selected nodes to make the decision path easy to track. **Cost Scale toggle** adjusts the size of operator nodes based on their estimated cost, highlighting why specific operators or join sequences were chosen. The right DAG shows the result after activating both features. Additionally, **clicking on an operator node** reveals detailed cost factors and calculation formulas in C2. By **clicking view details**, users can access explanations for each parameter in the cost formulas, as illustrated in Figure 5.



(a) Range Sliders **(b) Stacked Bar Chart**

Figure 6: Interactive Features for GEQO

Example 1. C2 demonstrates how users can understand the optimizer's decision by examining the cost of a sequential scan and a bitmap heap scan for the `lineitem` relation in TPC-H Query 14. The query needs to access 74,900 tuples out of a total of 6,001,215 (N_{rows}). Although the bitmap heap scan incurs a startup cost for building the list of disk pages to fetch, it significantly reduces the number of pages (N_{pages}) and tuples (N_{tuples}) accessed. This reduction in disk I/O and CPU costs for scanning only the relevant records makes the bitmap heap scan more efficient than a sequential scan, which fetches all pages and processes every record. Due to the query's low selectivity, the bitmap heap scan achieves a much lower cost, leading the optimizer to select it.

By combining these features, *Jovis* helps users visually grasp all paths considered during optimization and the rationale behind the optimizer's decisions, including detailed cost calculations.

3.1.2 Genetic Query Optimizer. **Query Planning View** D visualizes PostgreSQL's GEQO, which applies a genetic algorithm to heuristically explore join orders when exhaustive enumeration becomes impractical (*i.e.*, when the number of relations involved in joins reaches or exceeds the configuration parameter `geqo_threshold`). GEQO iteratively refines a population of candidate join sequences, referred to as *genes*, through selection, crossover, and replacement. Each generation maintains a pool of genes, where lower-cost genes are considered more fit. New genes are generated via crossover, combining segments of two parent genes to produce offspring. The pool is then sorted by estimated cost and the least fit genes are discarded. This process continues until a preset number of join sequences are evaluated.

To visualize this process, *Jovis* employs a grid heatmap, inspired by Cruz [3]. The **heatmap** D1 represents the pool of genes across generations using color gradients to indicate their estimated costs. This effectively demonstrates how GEQO improves fitness over time. In addition, a **line chart** D2 tracks the best, worst, median, and average costs throughout the optimization process, providing an overview of cost trends.

Users can interact with the visualization to explore the optimization process in detail. The **range sliders** in D1 allow users to focus on specific generations or genes. As shown in Figure 6a, when zoomed in on a smaller range, crossover visualizations appear, illustrating which parent genes contribute to offspring through edges. **Clicking on a gene** displays its join sequence as a tree in D3, with operator costs represented as a stacked bar chart. Users can click on each stack to see how much the corresponding operator accounts for the total cost. As presented in Figure 6b, selecting a stack highlights the related operator node in red and displays its cost and percentage at the bottom. When the selected gene is newly generated in the current generation, the **crossover visualization** in D4 is activated. An arc diagram shows how parent genes' join sequences contribute to the offspring's join sequence using color-coded edges. In the **cost chart** D2, users can hover over lines to pinpoint the specific generation, toggle between linear and logarithmic scales, and click on legends to isolate metrics for focused analysis. Moreover, users can rerun the same query with prior optimization results and customized configuration in D5 (See Section 3.3.2 for details).

These dedicated visualizations and interactive features make the complex and opaque GEQO process accessible and understandable. Users can observe the optimizer's evolutionary behavior, understand why certain join sequences are favored, analyze cost distributions, and explore how crossover operations are executed.

3.2 Backend

The *Jovis* backend connects the frontend GUI with PostgreSQL through an Application Protocol Interface service. As shown in Figure 1, the query submitter transmits user queries to PostgreSQL, where they undergo parsing, rewriting, and optimization to determine the optimal execution plan. The executor then runs the execution plan, and relevant log data are returned to the backend. The backend parser processes this data and formats it into JSON, including all necessary fields for visualization in the user interface.

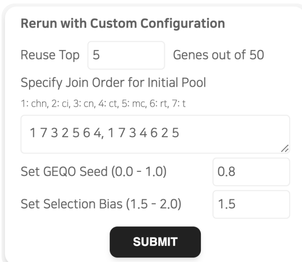
To enhance transparency into the optimization process, *Jovis* introduces a minimal patch to PostgreSQL that enriches its logs with detailed optimizer state data. For the standard optimizer, the patch captures information generated during query plan enumeration, including access paths, path costs, and join sequences. In the case of GEQO, it records each step of join sequence generation and crossover operations. These internal states are written to a log file during query optimization. Upon receiving query results from PostgreSQL, the backend processes the log file. Since the log contains unstructured plain text, the parser is crucial for extracting and converting the data into a structured, readable format. It first identifies the optimizer type and then extracts the corresponding data: path lists, costs, and the cheapest paths for the standard optimizer; generation information, join sequences, and associated costs for GEQO. The parsed data are formatted into JSON, combined with the query result, and passed to the frontend for visualization.

3.3 User-Guided Optimization

Jovis empowers users to actively participate in the query optimization process through user-guided optimization. By influencing the optimizer's decisions or reusing previous optimization results, users can diagnose suboptimal plans, explore alternative plans, and improve query performance in a controlled and intuitive manner.

```
/*+
    SeqScan(sales_returns)
    NestLoop(sales_returns catalog_page date_dim)
*/
```

(a) Hint



(b) Guided GEQO

Figure 7: User-Guided Optimization

3.3.1 Hint. *Jovis* supports the use of hints to guide the optimizer by specifying preferred physical operators or join orders. This capability is useful in scenarios where the optimizer selects suboptimal plans or when manual tuning is required. By providing hints, users can override the optimizer's decisions and analyze the impact of hints. Since PostgreSQL does not natively support hints, *Jovis* integrates the `pg_hint_plan` extension [5] with additional logging capabilities. Hints are specified using a special comment syntax, prefixed with `/*+` and ending with `*/`, as shown in Figure 7a.

Example 2. C in Figure 2 demonstrates using hints to guide the query optimizer. In this example, users apply a hint to TPC-H Query 14 to enforce a nested loop join (right DAG), overriding the optimizer's choice of a hash join (left DAG). This change in the join method also impacts the access paths for base relations. In the hash join plan, the optimizer sequentially scans the `part` relation to build a hash table and then probes it with the `lineitem` relation. In contrast, the nested loop join plan iterates over each record in the `lineitem` relation and looks up matching rows in the `part` relation. The join condition is `l_partkey = p_partkey`, where `p_partkey` is the primary key of the `part`. Since PostgreSQL automatically creates an index on primary keys, the optimizer can perform an index scan on `p_partkey`. As a result, for each row in `lineitem`, matching records can be efficiently retrieved from `part` using the index.

3.3.2 Guided GEQO. *Jovis* enhances GEQO by providing custom initialization of its pool. This feature is especially useful for analytical workloads, where the same queries are often executed repeatedly. By leveraging prior optimization results and tuning configurations, users can improve the optimality of the execution plan and explore the larger search space.

From the first two input fields in Figure 7b, users can specify the number of high-performing genes to include (*e.g.*, the top 5 join sequences with the minimum cost from a prior run) and provide user-defined join sequences. For the second input field, each relation's unique ID and name are given. Users can define join orders by entering space-delimited gene IDs and specify multiple join orders by concatenating them with commas. This guided initialization ensures that the genetic algorithm begins with a combination of proven solutions and user-defined candidates, increasing the likelihood of discovering better execution plans while avoiding the inefficiencies of random initialization.

(a) Default Join Order **(b) User-Guided Join Order**

Figure 8: Guided GEQO in JOB Query 10c

Jovis also provides a user-friendly interface for controlling key GEQO parameters, eliminating the need to manually add SQL commands or edit the PostgreSQL configuration file. From the last two input fields in Figure 7b, users can adjust the `geqo_seed`, which controls the random initialization of the pool to ensure reproducibility, and the `selection_bias`, which determines the selective pressure applied during parent selection in crossover. By tuning these parameters, users can balance exploration (*i.e.*, maintaining diversity among genes) and exploitation (*i.e.*, intensifying the search around high-fitness genes).

Example 3. To show the effectiveness of user-guided optimization in GEQO in improving query performance, we use JOB Query 10c as an example. Figure 8a shows the query plan chosen by GEQO and its associated cost in the default configuration, with `geqo_seed` set to 0 and `selection_bias` set to 2. By rerunning the query using *Jovis* 's user-guided optimization feature as shown in Figure 7b, users set a different `geqo_seed` value, focus on diversity by lowering `selection_bias` to 1.5, and initialize the population with the five lowest-cost genes from the previous run. As a result, GEQO discovers a more efficient execution plan, reducing the cost by 7%, as seen in Figure 8b.

4 Demonstration

In the demonstration, we will showcase the user experience with *Jovis* through two representative scenarios. The demonstration highlights the effectiveness of *Jovis* in both educational and practical contexts, helping users understand and participate in query optimization. In each scenario, users begin by selecting a query from the **Presets** or **History** panel in A or submitting a custom query through B and then choose the target database. Once the query is processed, users can explore tailored optimization visualizations in the **Query Planning** tab and view the final execution plan by switching to the **EXPLAIN** tab. In the **EXPLAIN** tab, users can zoom in and out, reposition the tree, and collapse or expand nodes by clicking on them. This interaction is to simplify complex plans by focusing on specific parts of the execution plan for closer inspection. Users can customize the query view layout and perform side-by-side analysis of multiple query plans using the **Queries** panel in A. By toggling the checkboxes, users can selectively show or hide views for specific queries, and by clicking the trash icon, they can permanently remove a query view.

4.1 Standard Optimizer

In scenarios where dynamic programming is applied for query optimization, *Jovis* constructs a DAG in C1. The DAG presents the scan methods for base relations, join orders, and join strategies considered during query planning. Users can zoom in and out and reposition the DAG to focus on specific portions of the planning. By clicking the **Play** button, users can simulate the step-by-step construction of the final execution plan and observe how the optimizer selects the cheapest paths based on estimated costs. Toggling the **Show Cost Scale** adjusts the size of operator nodes according to their estimated cost, helping users identify cost-intensive operations at a glance. For deeper analysis, users can click on a node to examine detailed cost information in C2, including cost factors, calculation formulas, and the values used in the calculation. Additional explanations for each cost factor can be accessed via the **view details** button. Selected nodes are highlighted with a red outline and can be selected or deselected individually. Users can select multiple nodes to compare cost calculations across operators. For user-guided optimization using hints, users can add directives in the **SQL editor** B. Hints are written at the top of the query, enclosed within `/*+` and `*/`, and consist of physical operators and relations in parentheses, separated by whitespaces. After submitting the query with hints, users can observe how the optimizer incorporates their guidance into the plan.

4.2 Genetic Query Optimizer

In scenarios where GEQO is applied, *Jovis* provides an interactive heatmap in D1 to visualize the evolution of join sequences, represented as genes, over generations. Users can observe how GEQO explores the search space and progressively converges toward plans with better fitness, reflected by lower estimated costs. The **Cost Chart** D2 tracks overall cost trends across generations using various metrics. Users can hover over lines to view precise cost values at specific generations and selectively display metrics by clicking the legend. For deeper exploration, users can hover over a gene within the heatmap to view a tooltip displaying its generation, fitness rank, and estimated cost. Upon selection, *Jovis* presents a detailed view in D3 that includes the join order and physical operators in a tree, along with a stacked bar chart that breaks down the estimated costs by the operator. Users can further focus on specific generations or subsets of genes by adjusting range sliders in D1. When the selected range is sufficiently small, *Jovis* visualizes the parent-child relationships produced by crossover using edges. By selecting an offspring gene, crossover visualization D4 traces the origin of each edge in its join sequence, revealing how sequences are inherited or mutated. If the edge is inherited from both parents, it is displayed as a split color, while mutated edges are shown in gray. For guided GEQO, users can rerun the query by specifying the top N genes to initialize the pool, defining join orders if desired, and adjusting `geqo_seed` and `selection_bias` parameters directly in D5.

Acknowledgments

This work was supported by the National Research Foundation of Korea (grant no. RS-2024-00358054). The authors assume all responsibility for the content of the paper.

References

[1] Christoph Anneser, Mario Petruccelli, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, Ryan Marcus, and Alfons Kemper. 2023. QO-Insight: Inspecting Steered Query Optimizers. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3922–3925.
[2] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (2011), 2301–2309.
[3] António Malta Lopes da Cruz. 2014. *Visualization for Genetic Algorithms.* Master's thesis. University of Coimbra.
[4] Dalibo. 2024. PEV2: Postgres Explain Visualizer 2. https://github.com/dalibo/pev2.
[5] NTT OSS Center DBMS Development and Support Team. 2024. pg_hint_plan: Extension adding support for optimizer hints in PostgreSQL. https://github.com/ossc-db/pg_hint_plan.
[6] The PostgreSQL Global Development Group. 2024. PostgreSQL: Documentation: 16: Chapter 62. Genetic Query Optimizer. https://www.postgresql.org/docs/16/geqo.html
[7] Jayant R. Haritsa. 2010. The Picasso database query optimizer visualizer. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 1517–1520.
[8] Toshihide Ibaraki and Tiko Kameda. 1984. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems (TODS)* 9, 3 (1984), 482–502.
[9] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
[10] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data.* Virtual Event, China, 1275–1288.
[11] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the 2008 International Conference on Management of Data.* Vancouver, Canada, 539–552.
[12] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases.* Seoul, Korea, 1049–1058.
[13] Oracle. 2025. *MySQL.* Retrieved March 10, 2025 from https://www.mysql.com/
[14] Oracle. 2025. MySQL Workbench. https://www.mysql.com/products/workbench/ Version 8.0.41.
[15] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM SIGMOD Record* 29, 4 (2000), 64–71.
[16] Ying Rong, Hui Li, Kankan Zhao, Xiyue Gao, and Jiangtao Cui. 2022. DBinsight: A Tool for Interactively Understanding the Query Processing Pipeline in RDBMSs. In *Proceedings of the 31st International Conference on Information & Knowledge Management.* Atlanta, GA, USA, 4960–4964.
[17] Daniel Scheibli, Christian Dinse, and Alexander Boehm. 2015. QE3D: Interactive Visualization and Exploration of Complex, Distributed Query Plans. In *Proceedings of the 2015 International Conference on Management of Data.* Melbourne, Victoria, Australia, 877–881.
[18] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 International Conference on Management of Data.* Boston, Massachusetts, USA, 23–34.
[19] Jess Tan, Desmond Yeo, Rachael Neoh, Huey-Eng Chua, and Sourav S Bhowmick. 2022. MOCHA: a tool for visualizing impact of operator choices in query execution plans for database education. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3602–3605.
[20] Alex Tatiyants. 2016. pev: Postgres Explain Visualizer. https://github.com/AlexTatiyants/pev.
[21] The PostgreSQL Global Development Group. 2025. *PostgreSQL.* Retrieved March 10, 2025 from https://www.postgresql.org/
[22] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data.* Philadelphia, PA, USA, 931–944.