Proceedings of the
13th International Conference on Applied Informatics
Eger, Hungary, February 19–21, 2026
DOI: 10.17048/icai.2026.34
pp. 34–47

Practical Prediction of Query Execution Time in PostgreSQL Using Lightweight Machine Learning Models

Tamás Balla[ab]

[a]Department of Computational Science, Eszterházy Károly Catholic University, Hungary
balla.tamas@uni-eszterhazy.hu

[b]Doctoral School of Informatics, Eötvös Loránd University, Hungary

Abstract. Accurate prediction of query execution time is important for workload scheduling, admission control, and service-level management in modern database systems. In PostgreSQL, the optimizer selects execution plans using internal cost estimates that reflect relative expected work rather than actual wall-clock runtime. This paper investigates whether these optimizer cost estimates, combined with a small set of plan-structure features extracted from machine-readable EXPLAIN output, can be calibrated into practical runtime predictions for analytical SQL queries. The study uses the TPC-H benchmark at scale factor 1 in a controlled single-user PostgreSQL environment. Features are extracted from EXPLAIN (FORMAT JSON), while runtime labels are obtained from EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON). A linear baseline is compared with a Gradient Boosting Decision Tree (GBDT) regressor. The results show that optimizer cost and measured runtime are strongly correlated (Pearson's $r = 0.8625$), but a direct linear model fails to capture their relationship, yielding very poor predictive performance (MAE = 97,616.04 ms, $R^2 = -8,869.94$). In contrast, the GBDT model achieves substantially better accuracy on the random evaluation split (MAE = 74.07 ms, $R^2 = 0.9924$), indicating that the cost-to-runtime relationship is strongly nonlinear. These findings support the view that PostgreSQL's optimizer cost contains useful predictive signal, but practical runtime prediction requires nonlinear calibration. The study is intentionally modest in scope and should be interpreted as a proof of concept.

Keywords: query performance prediction, cost model, machine learning

2020 *Mathematics Subject Classification:* Primary 68P15, 68T05

1. Introduction

SQL is a declarative language: users specify what result they want, while the database management system decides how to produce it. In relational database systems, this decision is typically made by a cost-based optimizer, which evaluates alternative execution strategies and selects the plan with the lowest estimated cost. Since the classical System R work, cost-based optimization has remained a central principle of relational query processing, and PostgreSQL follows the same general architecture of parsing, rewriting, planning, and execution.

A fundamental difficulty, however, is that optimizer cost is not the same as elapsed execution time. In PostgreSQL, planner costs are internal relative values derived from estimated I/O and CPU work, and they are intended for comparing alternative plans rather than predicting wall-clock latency directly. By contrast, actual execution time is an observed runtime quantity that depends not only on plan structure and estimated cardinalities, but also on hardware, cache state, statistics quality, and measurement conditions. As a result, even when optimizer cost contains useful signal, it cannot be treated as a direct time estimate.

This distinction matters in practice. Accurate runtime prediction is relevant to workload scheduling, admission control, service-level management, and general query-performance diagnostics. In such scenarios, it is often not enough to know which plan PostgreSQL prefers internally; practitioners also need a realistic estimate of how long a query is likely to run. This motivates the central question of the present study: can PostgreSQL's internal optimizer cost be calibrated into a practically useful predictor of execution time?

The problem is closely related to the broader literature on query performance prediction and learned database components. Earlier studies have shown that machine learning models can predict query latency and resource consumption from optimizer- and plan-derived features, while more recent work has explored deep plan representations, learned cost models, and learned query optimization. However, many of these approaches either require richer plan encodings, more invasive integration, or substantially larger training settings. In contrast, the present work intentionally adopts a lightweight and practical perspective.

Rather than replacing PostgreSQL's optimizer or modifying the database engine, this paper studies runtime prediction as a cost-to-runtime calibration problem. The approach uses only standard PostgreSQL interfaces: pre-execution features are extracted from `EXPLAIN (FORMAT JSON)`, while runtime labels are obtained from `EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)`. The empirical setting is deliberately modest: the TPC-H analytical benchmark at scale factor 1, executed in a controlled single-user environment. A simple linear baseline is compared against a Gradient Boosting Decision Tree (GBDT) regressor trained on optimizer cost and a small set of interpretable plan-structure features.

The contribution of the paper is therefore practical rather than architectural. First, it frames PostgreSQL runtime prediction as a nonlinear calibration problem rather than as a learned-optimizer problem. Second, it demonstrates a compact

and reproducible workflow for extracting predictive features from PostgreSQL's machine-readable plan output without modifying the engine. Third, it provides empirical evidence that optimizer cost contains strong predictive signal, but that effective runtime estimation requires a nonlinear model. The study is intentionally limited in scope and is best interpreted as a proof of concept; nevertheless, it shows that useful runtime prediction can be achieved with standard PostgreSQL tooling and lightweight machine learning.

2. Related Work

The present study is situated at the intersection of cost-based query optimization, query performance prediction, and learned database components.

The problem builds on classical cost-based optimization in relational database systems. Since the System R optimizer, query execution has been formulated as a search over alternative access paths and join strategies, where the optimizer selects the plan with the lowest estimated cost [15]. Modern systems, including PostgreSQL, continue to follow this principle, relying on statistics-driven cardinality estimates and analytical cost models to compare candidate plans [10, 12, 13]. At the same time, prior work has shown that optimizer quality is limited by estimation errors, so estimated cost and actual runtime may diverge substantially [4].

A closely related line of work is query performance prediction (QPP), where runtime or resource estimation is treated as a supervised learning problem. Ganapathi et al. demonstrated that machine learning can predict multiple execution-related metrics from query and plan characteristics [2]. Akdere et al. and Li et al. showed that optimizer-derived features can support accurate prediction of SQL query performance and resource consumption, even when direct reliance on analytical optimizer estimates is insufficient [1, 5]. Wu et al. further argued that optimizer cost is not unusable for runtime prediction, but rather requires a more appropriate modeling strategy than naive direct scaling [17, 18].

More recent work has extended this direction toward learned query optimization and learned cost estimation. Plan-structured neural models have been proposed for latency prediction [8], while systems such as Neo and Bao explore learned components for plan selection and optimizer guidance [6, 7]. On the cost-estimation side, learned models have also been proposed as alternatives or complements to traditional analytical cost formulas [3, 16, 20]. Industrial-scale evidence likewise confirms the practical relevance of runtime prediction for scheduling and resource-management tasks [19]. In this design space, the present work adopts a deliberately lightweight position: rather than replacing PostgreSQL's optimizer, it investigates whether PostgreSQL's native cost estimate and a small set of interpretable plan-structure features can be calibrated into useful runtime predictions by a nonlinear regressor.

3. Methodology

The study formulates PostgreSQL query runtime estimation as a supervised regression task. For each query instance q, a feature vector $x(q)$ is extracted from the pre-execution plan returned by `EXPLAIN (FORMAT JSON)`, while the target value $y(q)$ is the measured execution time in milliseconds obtained from the top-level `Execution Time` field of `EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)`. The learning objective is to estimate a function $f\colon x(q) \mapsto \hat{y}(q)$, where $\hat{y}(q)$ denotes the predicted execution time.

The feature design is intentionally lightweight and interpretable. The final feature set consists of PostgreSQL's root-level estimated cost together with a small number of aggregated structural descriptors derived from the query plan: `total_cost`, `plan_rows`, `node_count`, `join_count`, and `scan_count`. The structural descriptors were computed by recursively traversing the JSON plan tree returned by `EXPLAIN (FORMAT JSON)`. The variable `node_count` denotes the total number of plan nodes in the tree, including the root node and all nested child plans. The variable `join_count` counts plan nodes whose type corresponds to a join operator, including `Hash Join`, `Merge Join`, and `Nested Loop`. The variable `scan_count` counts scan-related operators, including sequential scans, index scans, index-only scans, and bitmap scan variants. The variables `total_cost` and `plan_rows` were taken from the root-level plan node and therefore represent PostgreSQL's estimated total plan cost and estimated output cardinality for the complete query plan, respectively. All features are available before query execution and do not rely on post-execution statistics. They were selected because they are easy to extract from machine-readable plan output and meaningful from the perspective of relational query processing. In particular, `total_cost` represents PostgreSQL's internal estimate of expected work, while the structural counts provide additional information about plan complexity.

The target variable reflects instrumented server-side execution time rather than end-to-end application latency. This distinction matters because `EXPLAIN ANALYZE` introduces measurement overhead, and the reported execution time does not include all possible client-side or network-related delays. Nevertheless, it provides a consistent and practically meaningful label for controlled runtime prediction experiments. To avoid information leakage, only pre-execution plan attributes are used as predictors; post-execution values such as `Actual Rows` or runtime buffer statistics are not included in the feature matrix.

Two predictive models were considered. The baseline is a simple linear regression using `total_cost` as the only predictor, intended to test whether PostgreSQL's internal cost estimate is already approximately proportional to measured runtime. The nonlinear model is a Gradient Boosting Decision Tree (GBDT) regressor trained on the full feature set. GBDT was chosen because it can capture nonlinear relationships and feature interactions while remaining well suited to small tabular datasets.

Model performance was evaluated using Mean Absolute Error (MAE), Root

Mean Squared Error (RMSE), and the coefficient of determination (R^2). In addition, Pearson correlation between `total_cost` and measured execution time was computed in order to assess whether optimizer cost contains predictive signal independently of any particular regression model. Beyond scalar metrics, the analysis also used prediction-versus-actual scatter plots, residual diagnostics, and an ablation experiment in which `total_cost` was removed from the feature set. The purpose of the ablation was to determine whether the nonlinear predictor primarily acts as a calibration layer over PostgreSQL's internal cost estimate, or whether structural plan features alone are sufficient for accurate runtime prediction.

4. Experimental Setup

4.1. Workload

The experiments were conducted using the TPC-H decision-support benchmark at scale factor 1. TPC-H was selected because it provides a standardized analytical workload consisting of 22 business-oriented SQL query templates with substantial variation in joins, aggregations, filtering conditions, and intermediate result sizes. This makes it an appropriate benchmark for studying the relationship between optimizer-derived plan characteristics and observed execution time.

The study used the 22 standard TPC-H query templates as the workload basis. Since the goal of the paper is runtime prediction rather than benchmark certification, TPC-H was used here solely as a controlled source of analytical SQL queries. No official audited TPC-H performance metric, such as QphH@Size, is claimed.

4.2. Database System and Execution Environment

All experiments were performed using PostgreSQL 16.11 in a controlled single-user environment. The measurements were carried out on a Lenovo ThinkPad L15 laptop upgraded to 32 GB RAM, with PostgreSQL running on Ubuntu 24.04 under Windows Subsystem for Linux (WSL).

The single-user setting was chosen deliberately in order to reduce interference from concurrent sessions and to isolate the relationship between query-plan properties and runtime as clearly as possible. The reported runtimes should therefore be interpreted as measurements obtained under controlled analytical conditions rather than as estimates for highly concurrent production workloads.

4.3. Plan and Runtime Collection

For each query, two types of PostgreSQL output were collected.

First, `EXPLAIN (FORMAT JSON)` was used to obtain the pre-execution query plan in machine-readable form. This output served as the source of predictive features, including the optimizer's estimated total cost and aggregated structural descriptors of the plan.

Second, `EXPLAIN ANALYZE` with `BUFFERS` and JSON output was used to collect measured execution statistics after query execution. From this output, the top-level `Execution Time` field was extracted as the regression target.

This measurement protocol creates a clean temporal separation between features and labels: the predictive model is trained only on information available before execution, while actual runtime values are obtained from separate analyzed executions.

4.4. Measurement Protocol

The workflow followed a per-query measurement strategy. For each TPC-H query template, a JSON plan representation was first collected for feature extraction. Query execution was then measured using one warm-up run and three retained measured runs per query in order to reduce the influence of first-run effects such as cold-cache behavior.

The SQL query instances were generated automatically by the benchmark tooling. The final regression dataset therefore contained 66 measured query executions, corresponding to 22 query templates with three retained measured runs each. Each row of the final dataset represents one retained measured execution paired with its pre-execution plan-derived features.

4.5. Train-Test Split and Feature Set

The predictive models were evaluated using a random train-test split generated with Scikit-learn's `train_test_split` function. The split used `test_size=0.3` and `random_state=42`. Accordingly, 70% of the data were used for training and 30% for testing, with a fixed random seed to ensure reproducibility.

The final feature set used in the experiments consisted of the following variables: `total_cost`, `plan_rows`, `node_count`, `join_count` and `scan_count`.

These features were extracted from PostgreSQL's machine-readable plan output and were all available before query execution.

4.6. Experimental Scope

The experimental design intentionally favors methodological clarity over scale. The benchmark size, workload scope, and controlled execution setting make the study suitable as a proof-of-concept evaluation of cost-to-runtime calibration, but they also limit the strength of generalization claims.

In particular, the study does not attempt to model:

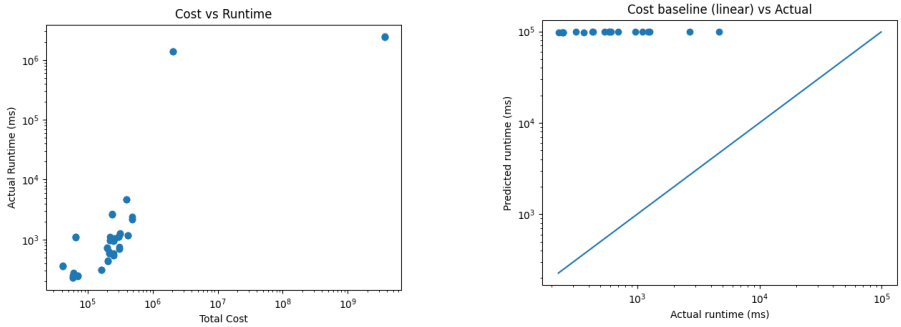
- concurrent multi-user workloads,
- end-to-end client-observed latency,
- workload drift across different database states, or

- optimizer behavior under substantially different hardware or PostgreSQL configurations.

Instead, the focus is on a narrower and more defensible question: whether PostgreSQL's optimizer cost and a small set of plan-structure features can be transformed into useful runtime predictions in a controlled analytical environment.

5. Results

The empirical results support four main observations. First, PostgreSQL's internal optimizer cost contains substantial predictive signal with respect to measured execution time. The Pearson correlation coefficient between `total_cost` and observed runtime was $r = 0.8625$, indicating a strong positive relationship between optimizer cost and query latency. However, this association alone is not sufficient to justify a direct linear calibration. As shown in Figure 1, optimizer cost and runtime are related, but the relationship is not captured adequately by a simple linear model.



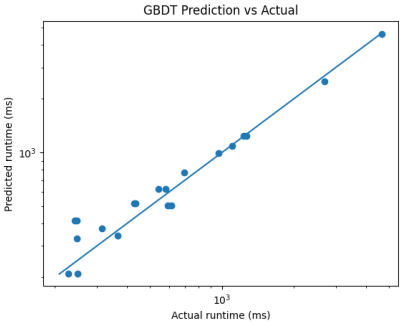
(a) Optimizer cost vs. actual runtime.

(b) Linear baseline prediction vs. actual runtime.

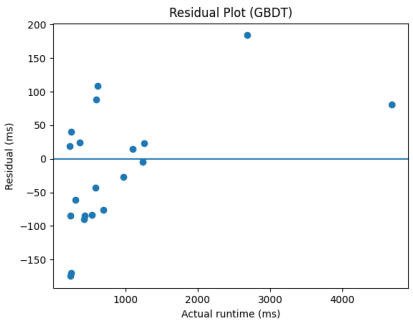
Figure 1. Relationship between PostgreSQL optimizer cost and measured execution time. Although optimizer cost is strongly associated with runtime, a direct linear baseline fails to provide an adequate calibration.

Second, the linear baseline fails to model the relationship between optimizer cost and execution time in a useful way. Using `total_cost` as the only predictor, the linear model achieved a mean absolute error of 97,616.04 ms and a coefficient of determination of $R^2 = -8869.94$. The strongly negative R^2 indicates that the linear model performs substantially worse than a trivial mean predictor. This behavior is also visible in Figure 1(b), where the predictions collapse into a narrow range and deviate strongly from the identity line. In other words, optimizer cost is informative, but it is not linearly calibrated to observed execution time.

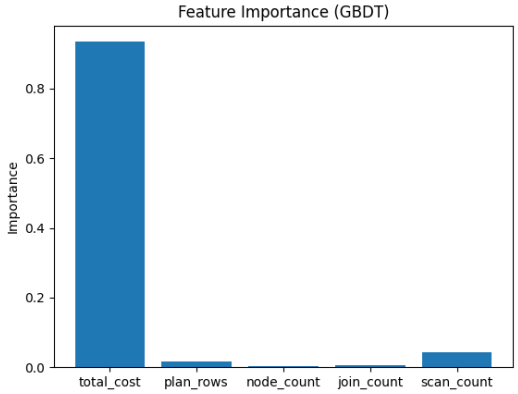
Third, the nonlinear Gradient Boosting Decision Tree (GBDT) model substantially improves predictive performance. The GBDT regressor achieved a mean



(a) GBDT prediction vs. actual runtime.



(b) Residual plot of the GBDT model.



(c) Feature importance of the GBDT model.

Figure 2. Evaluation of the nonlinear GBDT model. The model closely follows the identity line, residuals remain centered around zero, and feature importance indicates that `total_cost` is the dominant predictive signal.

absolute error of 74.07 ms and $R^2 = 0.9924$, indicating a very close fit between predicted and measured runtime on the evaluation subset. Although this result indicates strong predictive performance on the random test split, it should be interpreted cautiously. The dataset contains repeated measured executions of the same 22 TPC-H query templates, and the random split may place structurally identical or highly similar plans in both the training and test subsets. Consequently, the reported R^2 value should be understood primarily as evidence of interpolation performance under the chosen controlled workload and measurement protocol, rather than as evidence of generalization to unseen query templates or production workloads. More demanding evaluation protocols, such as grouped splits by query template or parameter instantiation, are left for future work. As shown in Figure 2(a), most predictions lie close to the identity line across the observed runtime range. This result strongly suggests that the relationship between PostgreSQL optimizer

cost and actual execution time is highly nonlinear, and that this nonlinearity can be captured effectively by a lightweight tree-based regressor.

Fourth, residual analysis supports the stability of the nonlinear model. Figure 2(b) shows residuals centered around zero with no clear systematic trend across the runtime range. This indicates that the GBDT model does not consistently overestimate or underestimate execution time. Although a small number of larger residuals remain for slower queries, the overall residual distribution does not indicate a strong structural bias.

Finally, the feature-importance analysis highlights the dominant role of PostgreSQL's internal cost estimate. As shown in Figure 2(c), `total_cost` contributes most of the predictive power, while the remaining structural plan features provide smaller but non-negligible refinements. This interpretation is further supported by the ablation analysis. When `total_cost` was removed from the feature set, predictive performance deteriorated substantially, with MAE increasing to 1,339.60 ms and R^2 decreasing to -1.8071. This result indicates that the structural plan descriptors alone were insufficient for accurate runtime prediction in the present setting. Instead, the learned model is best understood primarily as a nonlinear calibration layer over PostgreSQL's internal optimizer signal, while the additional structural features provide complementary refinements.

Taken together, these findings support a clear interpretation. PostgreSQL's optimizer cost is not a direct runtime estimate, and a linear cost-to-time mapping is ineffective. At the same time, the cost signal is highly informative, and a lightweight nonlinear model can transform it into a practically useful predictor of query execution time in a controlled analytical environment.

Table 1. Predictive performance of the evaluated models.

Model	MAE (ms)	R^2
Linear baseline (`total_cost` only)	97,616.04	$-8,869.94$
GBDT (full feature set)	74.07	0.9924
GBDT without `total_cost`	1,339.60	-1.8071

6. Discussion

The results suggest that PostgreSQL query runtime prediction can be interpreted as a calibration problem rather than as a replacement problem. PostgreSQL's optimizer already encodes substantial information about expected query execution effort in its internal cost estimate, but that estimate is not intended to represent wall-clock latency directly [11, 14]. The empirical findings of the present study show that the optimizer's cost signal is informative, but that its relationship to actual runtime is strongly nonlinear. This interpretation is consistent with earlier work showing that optimizer-derived signals can support runtime prediction, while direct use of analytical cost estimates is often insufficient [1, 18].

This interpretation is supported by the contrast between the linear baseline and the GBDT model. The strong Pearson correlation indicates that optimizer cost and runtime are meaningfully related, yet the linear baseline performs extremely poorly. This means that the problem is not the absence of signal, but the inadequacy of a simple linear mapping. In other words, PostgreSQL's optimizer cost should not be interpreted as a direct runtime estimate, but it can serve as a useful latent signal that requires calibration.

From this perspective, the GBDT model does not replace the optimizer. Instead, it acts as a lightweight nonlinear correction layer built on top of PostgreSQL's existing cost model. This distinction is important. The goal of the study is not to redesign plan search, cardinality estimation, or operator costing inside the database engine. Rather, it is to demonstrate that the information already exposed by PostgreSQL through standard `EXPLAIN` interfaces can be reused to obtain practically useful runtime predictions with minimal engineering overhead. In this sense, the present work is closer to query performance prediction than to full learned-optimizer systems such as Neo or Bao [6, 7].

The feature-importance analysis reinforces this interpretation. The dominant role of `total_cost` suggests that the optimizer's own estimate remains the primary predictive signal, while the additional structural plan features provide complementary information that helps the model correct systematic distortions. This is also consistent with more recent arguments in the learned cost model literature that useful information already exists inside traditional optimization pipelines and should often be refined rather than discarded [3, 20]. The ablation result supports the same conclusion: once `total_cost` is removed, predictive quality degrades substantially. The structural descriptors alone are therefore insufficient, but they improve performance when used in combination with PostgreSQL's internal estimate.

These findings position the proposed approach closer to the query performance prediction literature than to full learned-optimizer systems. The method is deliberately lightweight, engine-external, and based on interpretable tabular features rather than full plan-tree neural encodings [1, 2, 8]. This makes it attractive for applied settings where integration effort and reproducibility matter as much as predictive accuracy. In particular, the workflow relies only on machine-readable `EXPLAIN` output and does not require changes to PostgreSQL itself.

At the same time, the results should not be overgeneralized. The present study uses a small analytical workload in a controlled single-user environment, and the resulting model should be understood as workload- and environment-specific. The reported predictive success therefore demonstrates feasibility rather than universality. A model trained under one PostgreSQL version, hardware setting, and benchmark configuration should not automatically be assumed to transfer unchanged to other environments.

Nevertheless, the study has a clear practical implication. It shows that PostgreSQL's optimizer cost, although not a runtime estimate in itself, can be transformed into an accurate predictor of execution time in the evaluated controlled setting when combined with a lightweight nonlinear regression model. This opens

a realistic path toward external runtime-estimation tools that can support scheduling, admission control, and performance analysis without requiring invasive modifications to the database engine. Similar application motivations have also been emphasized in both academic and industrial runtime prediction work [17, 19].

7. Limitations

The present study has several limitations that should be considered when interpreting the results.

First, the dataset is relatively small. Although the experiments cover all 22 standard TPC-H query templates at scale factor 1, the final regression dataset contains only 66 retained measured executions. This limits the statistical diversity of the workload and constrains the strength of generalization claims. In particular, repeated executions of the same benchmark templates provide less structural variety than a larger corpus of heterogeneous analytical queries. The study should therefore be interpreted as a proof-of-concept evaluation rather than as evidence of a universally robust runtime estimator.

Second, the experimental workload is restricted to a controlled single-user environment. This choice was deliberate, since it reduces interference and makes the relation between plan-derived features and observed runtime easier to analyze. However, it also means that the reported model does not capture effects caused by concurrency, workload interference, queueing delays, or resource contention. These factors are highly relevant in operational database settings and have been emphasized in prior work on runtime prediction for dynamic and concurrent workloads [17, 19]. As a result, the predictive performance reported here should not be assumed to transfer directly to multi-user production environments.

Third, the target variable is instrumented server-side execution time reported by `EXPLAIN ANALYZE`, not end-to-end application latency. PostgreSQL documentation explicitly notes that `EXPLAIN ANALYZE` adds profiling overhead to query execution and that the reported execution time does not include all client-side and network-related costs [9, 14]. Consequently, the learned model estimates PostgreSQL-reported execution time under controlled measurement conditions, which is a meaningful but narrower target than user-perceived response time.

Fourth, the evaluation uses a random train-test split with a fixed seed. While this supports reproducibility, it also introduces a possible optimism bias because retained executions of the same query template may appear in both training and test partitions. Under such conditions, the model may benefit from seeing structurally similar plans during training. A more demanding evaluation protocol could group samples by query template or by parameter instantiation in order to assess generalization more strictly. This issue is particularly relevant in light of the small dataset size.

Fifth, the feature set is intentionally lightweight and restricted to a small number of aggregated plan-level descriptors. This is an advantage from the perspective of interpretability and simplicity, but it also limits representational richness. More

expressive approaches, including operator-level models and plan-structured neural representations, may capture aspects of runtime behavior that are not visible in a compact tabular feature set [1, 8]. The present study does not attempt such comparisons and therefore cannot claim superiority over richer model families.

Finally, the learned calibration should be regarded as environment-specific. PostgreSQL planner costs depend on configuration parameters, database statistics, and the execution environment, and these factors can vary across installations [11, 12]. The mapping learned in this study is therefore tied to the particular PostgreSQL version, hardware setting, and benchmark configuration used during measurement. Applying the same model in a different environment would likely require retraining or recalibration.

These limitations do not invalidate the main result of the paper, but they define its scope clearly. The contribution is not a universal latency predictor for PostgreSQL, but rather a compact demonstration that optimizer cost can be transformed into an accurate runtime predictor through lightweight nonlinear calibration under controlled analytical conditions.

8. Conclusion

This paper investigated whether PostgreSQL's internal optimizer cost can be calibrated into a practical predictor of query execution time using a lightweight machine learning approach. The evaluation was conducted on the TPC-H benchmark at scale factor 1 in a controlled single-user PostgreSQL environment, using features extracted from pre-execution query plans.

The results support a clear but deliberately limited conclusion. PostgreSQL's optimizer cost contains substantial predictive signal, as shown by the strong positive correlation between `total_cost` and measured runtime. However, this signal is not linearly calibrated to wall-clock execution time: the linear baseline performed extremely poorly, whereas the Gradient Boosting Decision Tree model achieved high predictive accuracy on the random test split. Because the dataset contains repeated executions of the same TPC-H query templates, this result should be interpreted as proof-of-concept evidence under controlled conditions rather than as strong evidence of generalization to unseen query templates or production workloads. Residual analysis further indicated that the nonlinear model did not exhibit clear systematic bias, and feature-importance analysis showed that `total_cost` remained the dominant predictive variable, complemented by a small set of plan-structure descriptors.

Taken together, these findings suggest that runtime prediction in PostgreSQL can be understood as a nonlinear cost-calibration problem. The proposed approach does not attempt to replace PostgreSQL's optimizer or modify the database engine. Instead, it demonstrates that useful runtime prediction can be achieved externally, using standard PostgreSQL interfaces and an interpretable lightweight model.

The contribution of the paper is therefore practical and methodological. It shows that even a small, controlled experimental setup can provide evidence that

optimizer-derived signals are valuable for runtime prediction, provided that they are modeled appropriately. At the same time, the study remains deliberately modest in scope. The reported results should be interpreted as a proof of concept tied to a specific benchmark, PostgreSQL configuration, and execution environment rather than as a universal latency estimator.

Several directions for future work follow naturally from this study. A larger dataset with more diverse query parameterizations and additional workloads would allow stronger generalization claims. More demanding evaluation protocols, such as grouped splits by query template or parameter instantiation, would provide a stricter assessment of robustness. It would also be valuable to extend the experiments to concurrent multi-user settings, where queueing and resource contention influence runtime behavior more strongly. Finally, future work could compare the present lightweight tabular approach with richer operator-level or plan-structured models in order to better understand the trade-off between simplicity, interpretability, and predictive performance.

The study demonstrates that PostgreSQL's internal cost signal, although not designed as a runtime estimate, can be transformed into an accurate execution-time predictor through lightweight nonlinear calibration under controlled analytical conditions.

References

[1] M. Akdere, U. Cetintemel, M. Riondato, E. Upfal, S. B. Zdonik: *Learning-based Query Performance Modeling and Prediction*, in: 2012 IEEE 28th International Conference on Data Engineering, 2012, pp. 390–401, DOI: 10.1109/ICDE.2012.64.

[2] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, D. A. Patterson: *Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning*, in: Proceedings of the 2009 IEEE International Conference on Data Engineering, 2009, pp. 592–603, DOI: 10.1109/ICDE.2009.130.

[3] R. Heinrich, X. Li, M. Luthra, Z. Kaoudi: *Learned Cost Models for Query Optimization: From Batch to Streaming Systems*, Proceedings of the VLDB Endowment 18.12 (2025), pp. 5482–5487, DOI: 10.14778/3750601.3750699.

[4] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, T. Neumann: *How Good Are Query Optimizers, Really?*, Proceedings of the VLDB Endowment 9.3 (2015), pp. 204–215, DOI: 10.14778/2850583.2850594.

[5] J. Li, A. C. Konig, V. Narasayya, S. Chaudhuri: *Robust Estimation of Resource Consumption for SQL Queries Using Statistical Techniques*, Proceedings of the VLDB Endowment 5.11 (2012), pp. 1555–1566, DOI: 10.14778/2350229.2350269.

[6] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska: *Bao: Making Learned Query Optimization Practical*, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 1275–1288, DOI: 10.1145/3448016.3452838.

[7] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, N. Tatbul: *Neo: A Learned Query Optimizer*, Proceedings of the VLDB Endowment 12.11 (2019), pp. 1705–1718, DOI: 10.14778/3342263.3342644.

[8] R. Marcus, O. Papaemmanouil: *Plan-Structured Deep Neural Network Models for Query Performance Prediction*, Proceedings of the VLDB Endowment 12.11 (2019), pp. 1733–1746, DOI: 10.14778/3342263.3342646.

[9] PostgreSQL Global Development Group: *EXPLAIN*, Accessed: 2026-04-19, PostgreSQL Global Development Group, 2026, URL: https://www.postgresql.org/docs/current/sql-explain.html.

[10] PostgreSQL Global Development Group: *Planner/Optimizer*, Accessed: 2026-04-19, PostgreSQL Global Development Group, 2026, URL: https://www.postgresql.org/docs/current/planner-optimizer.html.

[11] PostgreSQL Global Development Group: *Query Planning*, Accessed: 2026-04-19, PostgreSQL Global Development Group, 2026, URL: https://www.postgresql.org/docs/current/runtime-config-query.html.

[12] PostgreSQL Global Development Group: *Statistics Used by the Planner*, Accessed: 2026-04-19, PostgreSQL Global Development Group, 2026, URL: https://www.postgresql.org/docs/current/planner-stats.html.

[13] PostgreSQL Global Development Group: *The Path of a Query*, Accessed: 2026-04-19, PostgreSQL Global Development Group, 2026, URL: https://www.postgresql.org/docs/current/query-path.html.

[14] PostgreSQL Global Development Group: *Using EXPLAIN*, Accessed: 2026-04-19, PostgreSQL Global Development Group, 2026, URL: https://www.postgresql.org/docs/current/using-explain.html.

[15] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price: *Access Path Selection in a Relational Database Management System*, in: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, 1979, pp. 23–34, DOI: 10.1145/582095.582099.

[16] J. Sun, G. Li: *An End-to-End Learning-based Cost Estimator*, Proceedings of the VLDB Endowment 13.3 (2019), pp. 307–319, DOI: 10.14778/3368289.3368296.

[17] W. Wu, Y. Chi, H. Hacigümüş, J. F. Naughton: *Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads*, Proceedings of the VLDB Endowment 6.10 (2013), pp. 925–936, DOI: 10.14778/2536206.2536219.

[18] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüş, J. F. Naughton: *Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?*, in: 2013 IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 1081–1092, DOI: 10.1109/ICDE.2013.6544899.

[19] Z. Wu, R. Marcus, Z. Liu, P. Negi, V. Nathan, P. Pfeil, G. Saxena, M. Rahman, B. Narayanaswamy, T. Kraska: *Stage: Query Execution Time Prediction in Amazon Redshift*, in: Companion of the 2024 International Conference on Management of Data, 2024, pp. 280–294, DOI: 10.1145/3626246.3653391.

[20] J. Yang, S. Wu, D. Zhang, J. Dai, F. Li, G. Chen: *Rethinking Learned Cost Models: Why Start from Scratch?*, Proceedings of the ACM on Management of Data 1.4 (2023), pp. 1–27, DOI: 10.1145/3626769.