Can Learned Query Optimizers Adapt Across DBMS Architectures? A Case Study with Balsa and DuckDB

Wangshu Hong
University of Pennsylvania
Philadelphia, PA
wangshuhong2020@gmail.com

Ryan Marcus
University of Pennsylvania
Philadelphia, PA
rcmarcus@seas.upenn.edu

ABSTRACT

Learned query optimizers such as Balsa have demonstrated strong performance on PostgreSQL-style database systems, but their portability to modern analytical engines remains unclear. This work adapts Balsa to DuckDB, a vectorized analytical database that lacks both plan hinting and statement-level timeout support. To enable reinforcement-learning-based optimization in this setting, we introduce lightweight portability mechanisms based on SQL-level plan forcing and timeout management for safe exploration. We further incorporate censored-observation training to improve stability during reinforcement-learning-based exploration. Experiments on the Join Order Benchmark (JOB) [4] and JOB-Complex [8] benchmarks show that Balsa can learn competitive query plans on DuckDB without modifying the underlying optimizer or execution engine. These results suggest that learned query optimizers can adapt to substantially different DBMS architectures using lightweight integration mechanisms.

VLDB Workshop Reference Format:
Wangshu Hong and Ryan Marcus. Can Learned Query Optimizers Adapt Across DBMS Architectures? A Case Study with Balsa and DuckDB. VLDB 2026 Workshop: Applied AI for Database Systems and Applications (AIDB 2026).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/winstonhong15/balsa.

1 INTRODUCTION

Learned query optimizers have recently demonstrated strong performance improvements over traditional cost-based optimizers [13], particularly on PostgreSQL-style database systems. Prior systems such as Neo [6], Bao [5], Lero [12], and Balsa [9] use machine learning to improve components of query optimization or replace traditional optimizers entirely. Among these systems, Balsa is notable for learning query optimization policies directly through reinforcement learning without relying on expert demonstrations. Evaluations on PostgreSQL show that Balsa can outperform traditional optimizers on analytical workloads after only a few hours of training.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

Despite these promising results, it remains unclear whether learned query optimizers can be effectively adapted beyond traditional row-oriented database architectures (most real-world deployments, such as [1, 11, 14], have not used "full" RL-powered optimizers). Most prior work evaluates learned optimizers on PostgreSQL or PostgreSQL-like systems that expose optimizer hinting interfaces and tuple-at-a-time execution models. Modern analytical databases such as DuckDB [7] differ substantially from these environments, employing vectorized execution, columnar storage, and embedded execution architectures. These architectural differences may significantly alter the runtime characteristics and optimization opportunities observed by learned optimizers.

This work presents an adaptation of Balsa [9] to DuckDB. Integrating Balsa into DuckDB introduces several systems challenges because DuckDB exposes neither optimizer hinting nor statement-level timeout support. To address these limitations, we implement lightweight SQL-level plan forcing and timeout management for DuckDB. Additionally, we replace the ad-hoc timeout strategy used in Balsa (which improved upon the ad-hoc strategy in Neo [6]) with censored observation, improving learning stability.

Using the Join Order Benchmark (JOB) [4] and JOB-Complex [8] benchmark workloads, we evaluate Balsa on both DuckDB and PostgreSQL backends. Our results show that Balsa can learn competitive query plans on DuckDB without modifying the underlying optimizer or execution engine. These findings suggest that learned query optimizers can be adapted across substantially different DBMS architectures using lightweight integration mechanisms rather than invasive engine modifications.

This work makes the following contributions:

- We demonstrate that Balsa can be adapted to DuckDB through a lightweight SQL-level plan-forcing mechanism based on AST-guided query reconstruction and deterministic join-order preservation.
- We implement timeout management together with censored-observation training to support stable reinforcement-learning-based exploration.
- We evaluate the resulting system on JOB and JOB-Complex across DuckDB and PostgreSQL backends, providing an empirical study of learned query optimization on both vectorized and row-oriented database systems.

2 BACKGROUND

In this section, we describe the basics of Balsa and the specifics of the DuckDB query optimizer.

2.1 Balsa

Balsa [9] is a learned end-to-end query optimizer that uses reinforcement learning to optimize join ordering and plan selection. Unlike prior systems that rely on expert-generated plans, Balsa bootstraps from a lightweight simulator and subsequently refines its value network using real query execution feedback. During planning, Balsa performs beam search guided by a learned value function implemented using a tree convolutional neural network (TCNN). Prior evaluations demonstrate that Balsa can outperform traditional query optimizers on PostgreSQL for analytical workloads such as TPC-H and JOB. Existing implementations of Balsa rely heavily on PostgreSQL-specific functionality, particularly optimizer hinting through the pg_hint_plan extension [2] and native statement timeout support. These assumptions complicate portability to systems that expose fewer optimizer-control interfaces.

2.2 DuckDB

DuckDB [7] is an embedded analytical database designed for OLAP workloads. Unlike PostgreSQL's row-oriented tuple-at-a-time execution model, DuckDB employs vectorized execution and columnar storage. Queries are executed in batches, enabling efficient CPU utilization and SIMD acceleration. DuckDB presents an especially challenging target for learned query optimization because it exposes neither optimizer hinting interfaces nor APIs for direct plan injection. Furthermore, DuckDB does not provide statement-level timeout controls, which are commonly used by reinforcement-learning-based optimizers to bound exploration cost during training.

These architectural differences make DuckDB a useful testbed for studying whether learned query optimizers can be effectively adapted beyond PostgreSQL-style systems.

3 ADAPTING BALSA TO DUCKDB

Adapting learned query optimizers to new DBMS architectures requires addressing differences in optimizer-control interfaces and execution infrastructure. Existing learned optimizers [1, 5, 9] commonly rely on optimizer hinting or direct plan injection APIs to enforce plans selected during reinforcement-learning-based exploration. DuckDB exposes neither capability, requiring alternative mechanisms for plan enforcement and safe exploration. To address these challenges, we introduce a lightweight SQL-level plan-forcing mechanism together with timeout management for handling high-latency and censored runtime observations.

3.1 SQL-Level Plan Forcing

Balsa relies on controlling query execution plans during reinforcement-learning-based exploration. Existing learned query optimizers commonly achieve this using optimizer hinting interfaces or direct plan injection APIs [1, 5, 9]. In PostgreSQL, for example, Balsa relies on the pg_hint_plan extension [2] to enforce join ordering and operator selection. However, many modern analytical database systems expose limited optimizer-control interfaces, making direct integration difficult. DuckDB provides neither optimizer hinting nor APIs for direct logical or physical plan injection.

To address this limitation, we implement a lightweight SQL-level plan-forcing mechanism based on deterministic join-order preservation. DuckDB preserves user-specified join order when join reordering is disabled via optimizer configuration. We exploit this behavior by reconstructing SQL queries whose join structure matches the join tree selected by Balsa.

The SQL reconstruction process operates on Balsa's internal abstract syntax tree (AST) representation. Given a candidate join tree, a recursive tree-to-SQL procedure emits nested join expressions in the desired left-deep order while preserving predicates and projections from the original query. When join reordering is disabled, DuckDB preserves the reconstructed join structure selected by Balsa for the workloads considered in this study.

Because the mechanism operates entirely through SQL rewriting, the approach may also be applicable to other DBMSes lacking explicit optimizer hinting interfaces. The current implementation primarily controls join ordering while leaving lower-level operator implementation decisions to DuckDB's native optimizer and execution engine. Consequently, the system should be interpreted as enforcing join structure rather than fully specifying physical plans.

3.2 Timeout Management

Reinforcement-learning-based exploration can occasionally generate highly inefficient join plans whose execution latency substantially exceeds the baseline optimizer. Without additional safeguards, these pathological plans can significantly increase training cost and destabilize value-function updates. To bound exploration overhead, learned optimizers have used timeouts. Prior work handled timeouts by (incorrectly) labeling timed-out queries as having a latency equal to the timeout value [5, 6, 9]. Thus, if a query's true latency was 5 minutes, but the query timed out after 1 minute, the query would be labeled as having a latency of 1 minute, thus penalizing predictive models that would correctly estimate the query's latency.

We propose an approach that does not penalize predictive models for estimating that a timed-out query's latency is higher than the timeout value. For each query, candidate plans are executed under an adaptive timeout threshold derived from the baseline optimizer runtime. Plans exceeding the timeout threshold are terminated early and treated as *censored runtime observations* because their true execution latency is unknown beyond the timeout limit. To incorporate these incomplete observations into value-function training, we adopt a loss formulation inspired by prior work on learning under censored response data [3] and recent applications of censored observations in learned query optimization [10].

For timeout samples, predictions below the timeout threshold continue to incur squared-error penalties because they underestimate the observed runtime bound. However, predictions exceeding the timeout threshold receive zero loss because they remain consistent with the censored observation. Figure 1 illustrates the censored-observation loss behavior used during training.

This mechanism reduces the influence of extreme latency outliers during reinforcement learning while preserving informative supervision for underestimated runtime predictions, improving overall training stability during exploration.

4 EVALUATION

We evaluate the adapted Balsa system on DuckDB (v1.2.2) and compare against PostgreSQL (v12.5) using Balsa's original PostgreSQL-based integration. PostgreSQL experiments primarily evaluate the

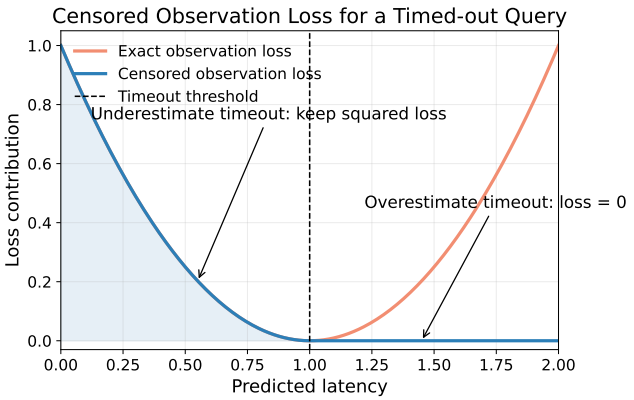

Figure 1: Censored-observation loss for timeout samples. Underestimates of the timeout threshold are penalized, while overestimates incur zero loss.

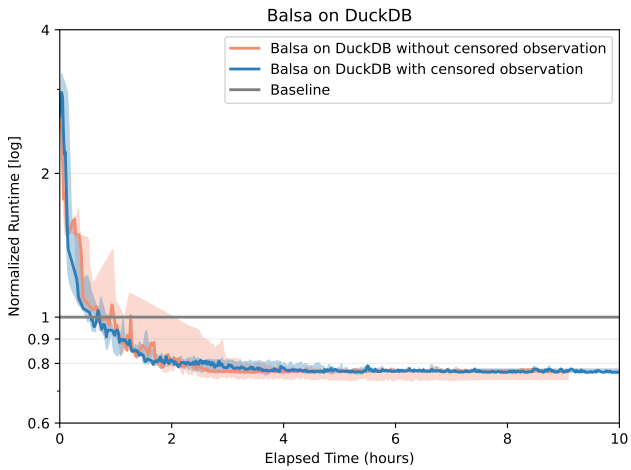

Figure 2: Training convergence behavior of Balsa on DuckDB for the JOB workload. Normalized runtime of training queries (log scale) versus elapsed training time.

impact of censored-observation training across systems with different execution architectures. Experiments are conducted on a machine equipped with an Intel Core Ultra 7 155H CPU and 64GB RAM running Arch Linux.

Evaluation uses the Join Order Benchmark (JOB) [4] and JOB-Complex [8]. JOB contains analytical join queries over the IMDb dataset, while JOB-Complex introduces more challenging join predicates and irregular join structures. Following prior Balsa evaluations, workloads are partitioned into training and testing sets using randomized query splits.

We compare standard Balsa training against a censored-observation variant on both DuckDB and PostgreSQL backends. Results report normalized runtime relative to each system's native optimizer baseline. Reported speedups are computed using the median runtime across five independent evaluation runs.

4.1 Performance on DuckDB

Table 1: Median Wall-clock Time to Reach Runtime Thresholds on JOB Across Five Independent Runs.

Time to:	Standard	With Censored Observation
2× Baseline	**4.02**	6.48
1.1× Baseline	36.65	**23.70**
Baseline	41.53	**34.41**
0.9× Baseline	**61.07**	63.80
0.8× Baseline	104.72	**94.70**

Figure 2 illustrates the training convergence behavior of Balsa on DuckDB. Across both standard and censored-observation training, Balsa successfully learns query plans that outperform DuckDB's native optimizer within the first hour of reinforcement-learning-based exploration.

During early exploration, both variants occasionally evaluate pathological join orders whose runtimes substantially exceed the baseline optimizer. The censored-observation variant reduces the impact of these extreme outliers by avoiding excessive penalties for timeout-censored executions, leading to more stable value-function updates and lower runtime variance during training.

Table 2: Median Wall-clock Time to Reach Runtime Thresholds on JOB-Complex Across Five Independent Runs.

Time to:	Standard	With Censored Observation
2× Baseline	37.53	**28.97**
1.1× Baseline	132.61	**89.10**
Baseline	139.39	**106.01**
0.9× Baseline	177.65	**154.92**
0.8× Baseline	**198.92**	219.94

Table 1 and Table 2 summarize the wall-clock time required to reach different normalized runtime thresholds on JOB and JOB-Complex. The results show that censored-observation training improves convergence behavior across several practical operating regions, particularly near baseline and moderately sub-baseline runtimes. On JOB, the censored-observation variant reaches the baseline and 1.1× baseline thresholds more quickly than standard training, while also reaching the strongest observed threshold (0.8× baseline) faster overall. On JOB-Complex, censored observation consistently improves convergence speed through the 0.9× baseline region, although standard training converges slightly faster at the most aggressive threshold.

These results suggest that censored-observation training primarily improves exploration stability rather than uniformly outperforming standard reinforcement learning across all convergence targets.

Figure 3 summarizes final speedup results on JOB and JOB-Complex. On JOB, Balsa achieves up to 1.32× training-query speedup and 1.29× test-query speedup relative to DuckDB's native optimizer. On JOB-Complex, gains are smaller but remain competitive given DuckDB's already highly optimized analytical execution engine,

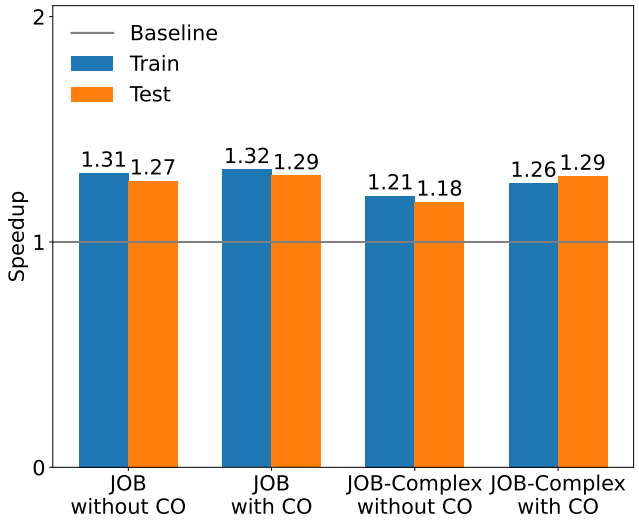

Figure 3: Balsa speedup relative to DuckDB's native optimizer on JOB and JOB-Complex.

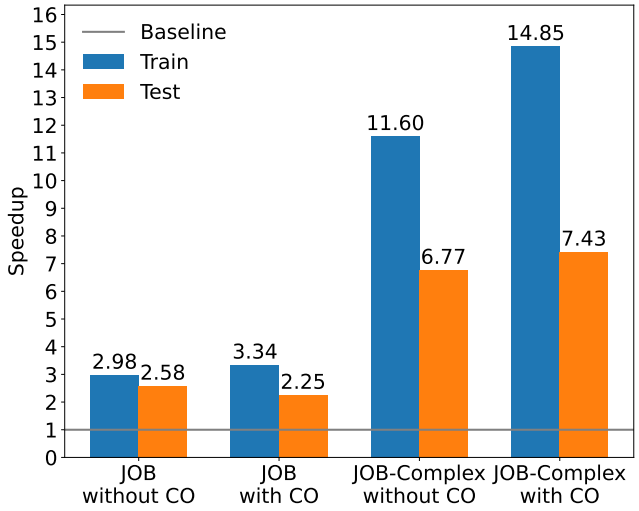

Figure 4: Balsa speedup relative to PostgreSQL's native optimizer on JOB and JOB-Complex.

reaching up to 1.26× training-query speedup and 1.29× test-query speedup under censored-observation training.

Although the absolute improvements on DuckDB are modest compared to prior PostgreSQL-based learned optimizers, the results demonstrate that reinforcement-learning-based query optimization remains effective under vectorized execution using only lightweight SQL-level plan forcing. More broadly, the findings suggest that learned optimizers can continue to identify beneficial join-ordering decisions even in systems with highly optimized native analytical execution engines.

4.2 Cross-System Adaptation

To study portability across execution architectures, we compare Balsa's behavior on DuckDB and PostgreSQL backends. Across both systems, Balsa can be adapted using a common reinforcement-learning framework despite substantial differences in optimizer interfaces and execution models.

Figure 4 summarizes final speedup results on PostgreSQL for both JOB and JOB-Complex workloads. Compared to DuckDB, Balsa achieves substantially larger runtime improvements on PostgreSQL across both workloads. Under censored-observation training, Balsa reaches up to 3.34× training-query speedup on JOB and up to 14.85× training-query speedup on JOB-Complex. Test-query improvements are smaller but remain substantial, reaching up to 2.25× on JOB and 7.43× on JOB-Complex.

The contrast between DuckDB and PostgreSQL highlights how DBMS architecture influences the optimization opportunity available to learned query optimizers. DuckDB's vectorized execution engine and highly optimized analytical query processing pipeline appear to reduce the runtime gap between alternative join orders, limiting the performance gains achievable through learned optimization.

In contrast, PostgreSQL exhibits substantially greater sensitivity to join-order selection, enabling larger gains from reinforcement-learning-based optimization. This behavior is consistent with prior analyses of JOB-Complex [8], which report substantially larger optimization gaps than the original JOB benchmark due to its more challenging join structures and predicates. Overall, these results suggest that reinforcement-learning-based query optimization can be effectively adapted across substantially different DBMS architectures using lightweight SQL-level integration mechanisms.

Censored-observation training generally improves exploration stability and reduces the impact of pathological plans during reinforcement learning. The effect is most pronounced on DuckDB workloads, where censored observation consistently improves convergence behavior across multiple runtime thresholds. On PostgreSQL, the impact on final runtime improvement is less consistent, although censored observation still reduces early-stage runtime variance during exploration.

5 CONCLUSION

This work adapts Balsa to DuckDB, a vectorized analytical database lacking optimizer hinting and statement-level timeout support. Using lightweight SQL-level plan forcing together with timeout management and censored-observation training, Balsa learns competitive query plans on DuckDB without modifying the underlying optimizer or execution engine.

Experiments on JOB and JOB-Complex show that reinforcement-learning-based query optimization remains effective across substantially different DBMS architectures, although the magnitude of achievable improvement depends strongly on the underlying execution engine and optimizer design. Future work includes extending the approach to additional optimizer decisions and larger analytical workloads.

REFERENCES

[1] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3515–3527. https://doi.org/10.14778/3611540.3611544

[2] NTT OSS Center DBMS Development and Support Team. 2020. pg_hint_plan. https://github.com/ossc-db/pg_hint_plan.

[3] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2013. Bayesian Optimization With Censored Response Data. arXiv:1310.1947 [cs.AI] https://arxiv.org/abs/1310.1947

[4] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[5] 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 (SIGMOD '21)*. China. https://doi.org/10.1145/3448016.3452838 Award: 'best paper award'.

[6] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[7] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[8] Johannes Wehrstein, Timo Eckmann, Roman Heinrich, and Carsten Binnig. 2025. JOB-COMPLEX: A Challenging Benchmark for Traditional & Learned Query Optimization. In *VLDB 2025 Workshop: Applied AI for Database Systems and Applications (AIDB 2025)*. VLDB Endowment.

[9] 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 (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 931–944. https://doi.org/10.1145/3514221.3517885

[10] Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2024. Low Rank Approximation for Learned Query Optimization. In *Proceedings of the Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Santiago, AA, Chile) *(aiDM '24)*. Association for Computing Machinery, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.1145/3663742.3663974

[11] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, Philadelphia PA USA, 2299–2311. https://doi.org/10.1145/3514221.3526052

[12] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1466–1479. https://doi.org/10.14778/3583140.3583160

[13] Rong Zhu, Lianggui Weng, Bolin Ding, and Jingren Zhou. 2024. Learned Query Optimizer: What is New and What is Next. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 561–569. https://doi.org/10.1145/3626246.3654692

[14] Rong Zhu, Lianggui Weng, Wenqing Wei, Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian, Bolong Zheng, and Jingren Zhou. 2024. PilotScope: Steering Databases with Machine Learning Drivers. *PVLDB* 17, 5 (2024), 980–993. https://doi.org/10.14778/3641204.3641209