Future Technology
Open Access Journal
ISSN 2832-0379

November 2026| Volume 05 | Issue 04 | Pages 119-132

Journal homepage: https://fupubco.com/futech

https://doi.org/10.55670/fpll.futech.5.4.10

Future Publishing LLC

Article

A hybrid heuristic–reinforcement learning method for multi-table join optimization in cloud-native databases

Huichen Ma[1]*, Shaochun Liu[2]

[1]Computer Science & Engineering Department, University of California, San Diego, La Jolla 92093, USA
[2]Data Trading Department, Beijing JoinQuant Investment Management Co., Ltd, Beijing 100000, China

ARTICLE INFO

Article history:
Received 24 February 2026
Received in revised form
25 June 2026
Accepted 27 July 2026

Keywords:
Cloud-native database, Heuristic search, Multi-table join optimization, Query optimizer, Reinforcement learning

*Corresponding author
Email address:
huichenm_research@163.com

DOI: 10.55670/fpll.futech.5.4.10

ABSTRACT

Multi-table join optimization in cloud-native databases faces challenges of storage-compute separation, resource elasticity, and runtime cost variation that classic optimizers cannot handle, which drives the need for methods able to adjust their decision boundaries according to the runtime behavior of the underlying cloud infrastructure. The proposed hybrid architecture freezes the stable sub-components of an IKKBZ-GOO heuristic seed plan into a structural freeze mask that constrains the learning algorithm throughout training, rather than being discarded after a single warm-start iteration. The mask works together with a state-aware cost function whose single state-dependent parameter is updated online. Two further components complete the design: a logistic gate trained on production telemetry, and a regression containment strategy that falls back to the heuristic plan when the learned plan underperforms. Training converges in 6.4 hours on one 16-vCPU compute node without GPU acceleration. The method was evaluated on the Join Order Benchmark, TPC-H, and TPC-DS, with PostgreSQL running on an eight-node Kubernetes cluster with object-store-backed shared storage. It achieves a geometric mean speedup of 1.38-fold on JOB, 1.28-fold on TPC-DS, and 1.07-fold on TPC-H, produces a query plan in 4.6 to 6.4 ms, returns median latency to within 5% of its pre-scaling value within 22 seconds of a scaling event, and incurs a 21% overhead at the 95th percentile under node jitter. These findings demonstrate that heuristic and machine-learning components can be integrated to yield a usable optimizer whose decision boundaries adapt to the environment in which it is deployed.

1. Introduction

The migration of analytical database queries to cloud-native systems has shifted the ground rules underlying query optimization. The separation of storage from computing, as evidenced in the motivations behind the Amazon Aurora architecture for scalable cloud-native RDBMS [1], and its subsequent refinement in Socrates' separation of log durability from data persistence [2], introduces latency penalties and elasticity boundaries not found in the monolithic engines upon which conventional optimization techniques are based. The engineering need to offer multi-tenant scalable computation has resulted in designs such as PolarDB Serverless, which explores the memory-compute-disaggregated approach [3]; Snowflake, which introduces a new paradigm for data warehouse processing through multi-cluster, shared-data execution [4]; and Amazon Redshift, in which the new architecture directly incorporates query acceleration and optimization into its design [5]. This paradigm is also evident in the space of distributed SQL databases, such as CockroachDB, which belongs to the category of geo-distributed cloud-native systems where the cost of a join operation depends on replica location and network conditions across regions [6]. Optimization of multi-way join operations is right at the core of this issue. The difference in cost between a good query plan and a bad one increases super-linearly with respect to the number of tables that can be joined. Misestimation of cardinalities in the execution plan flows up and accumulates while the large plan is broken down into smaller components. The complexity of enumerating all possible semantically equivalent join orderings increases exponentially with respect to the number of relations when considering left-deep trees, and even faster for bushy trees. Each optimizer thus must sacrifice completeness for latency in planning, and the point of sacrifice is determined by the computational capabilities present during runtime. In cloud-based systems, the expense

of accessing a page, the expense of transporting data from storage nodes to compute nodes via network links, and the cache capability available all vary depending on the temporary conditions prevailing. An optimizer optimized for one set of these conditions may perform poorly under a different set. This point is emphasized in the recent survey of cloud-native database management systems, which highlights elasticity and the heterogeneity of tenant workloads as the factors most relevant to the failure of conventional optimizer design assumptions [7].

The enumeration problem of classical optimization was resolved by dynamic programming for the access path selection approach adopted in the System R optimizer [8] and by the randomized and heuristic algorithms that were discussed in the VLDB Journal paper on join ordering [9]. This technique provides a realistic plan generation solution for reasonable cardinality joins and continues to be the standard in popular DBMSs. Calibration of their cost model takes place only once and is not related to the available system resources; thus, it is not adaptive to changing conditions. In the learned-optimizer approach, the cost estimation is replaced with or complemented by the use of machine learning. Neo learns the value network with tree structure for synthesizing efficient plans without cardinality estimations [10]. The qd-tree uses the reinforcement learning approach for data layout, and the improvements achieved in cloud analytics demonstrate that this category of learners can be used for the storage layer, where the access cost is represented explicitly in the current cost model [11]. RTOS uses reinforcement learning for join-order determination directly through the tree-LSTM learner [12]. Bao provides an industrial solution based on a bandit algorithm driven by the hint sets [13]. This property of these four proposals is identified in the survey paper about the integration of database and artificial intelligence, that is, each proposal replaces one part of the optimizer with the learning one while keeping the rest parts unchanged [14].

The problem common to both previous approaches is that they consider the learned aspect and the heuristic aspect to be mutually exclusive. Heuristic-only optimizers will not be able to adjust their behavior based on changes that occur due to runtime state changes, while learned-only optimizers can either incur training costs that are impossible to pay in an elastic environment or will expose the whole system to the risk of a tail-risk regression whenever the operational conditions deviate from those seen during training. The assumption that one side should dominate over the other implicitly means that the two aspects cannot be combined at all, each contributing something to the other. This study pursues four objectives.

- Construct a join cost model that depends explicitly on the cloud runtime state and that absorbs cluster reconfiguration without retraining the full cost surface.
- Design a heuristic seed plan together with a structural freeze mask that remains in force throughout reinforcement-learning training rather than being discarded after a single warm-start iteration.
- Construct a constant-time query-level gate, trained on production telemetry, that assigns each query to the path matched to its complexity profile and to the prevailing cloud state.
- Evaluate the resulting optimizer on the Join Order Benchmark, TPC-H, and TPC-DS, and under elastic scale-out, node jitter, and cold-start conditions that static benchmarking does not exercise.

Section 2 develops these objectives in the order stated. Section 2.1 presents the state-aware cost model, Section 2.2 the IKKBZ-GOO seed generator and its freeze mask, Section 2.3 the reinforcement-learning refinement stage, and Section 2.4 the query-level gate that binds the two paths together. The present study introduces a hybrid heuristic-RL approach to multi-table join optimization in cloud-native databases that solves these problems via four design decisions.

- A state-sensitive cost function whose single state-dependent coefficient is adjusted online by an exponentially weighted moving average. Shifts in the state of the cloud enter the cost surface without retracing the entire model.
- A seed plan generator that combines IKKBZ linearization with GOO bushy merges. The stable structures of the seed are frozen into a structural freeze mask that persists across reinforcement-learning training instead of being thrown away after one warm start trial.
- A gating mechanism trained on production telemetry as a constant-time logistic regression. Each query is assigned to the best path according to its complexity profile and the state of the cloud at the time of execution, in place of the per-query Monte Carlo techniques used by previous hybrid approaches.
- A regression containment policy that monitors the realized execution time of every plan issued by the learned path and rolls back to the heuristic plan whenever the observed cost exceeds the heuristic prediction by a fixed margin.

The proposed approach is evaluated on the Join Order Benchmark together with the industry standard benchmark workloads TPC-H and TPC-DS, on PostgreSQL running within a Kubernetes cluster with object-store-backed shared storage. The empirical evidence shows consistently improved median latencies across all three benchmarks along with reliable recovery behavior under elastic scaling, node jitter, and cold-starts in bounded time without resorting to any run-time fallbacks. This research contributes to the understanding that the learning component and heuristics are used in synergy rather than competing against each other, as well as an optimizer implementation ready for deployment where the decision-making boundaries depend on the cloud runtime environment as opposed to a statically calibrated setting.

2. Materials and methods

2.1 Problem formulation and cloud-native join cost model

Multi-table join optimization in cloud-native databases is formulated as a sequential decision problem over the space of join trees that may be either left-deep or bushy, where each query q is associated with a relation set $R_q = \{r_1, r_2, \ldots, r_n\}$ and a predicate set P_q that together define the join graph $G_q = (R_q, E_q)$.

The optimizer is required to produce an execution plan:

$$\pi^* = \arg\min_{\pi \in \Pi_q} C(\pi \mid \mathbf{u}_t) \tag{1}$$

That minimizes the expected execution latency conditioned on the runtime cloud state $\mathbf{u}_t$, where Π_q denotes the set of all semantically equivalent plans and $C(\cdot)$ is a state-aware cost functional. The principal departure from classical formulations lies in the explicit dependence on $\mathbf{u}_t$, which is treated as a time-varying vector:

$$\mathbf{u}_t = \langle u_t^{\text{cpu}}, b_t^{\text{net}}, h_t^{\text{buf}}, k_t \rangle \quad (1)$$

Encoding CPU utilization, network bandwidth between compute and storage tiers, buffer-pool hit ratio, and the number of active compute nodes at decision time $t. It is necessary to allow for left-deep skeletons as well as bushy fragments within one single search space since cloud OLAP queries have star join queries, which are best executed through bushy pipelining, and snowflake queries, where the best query plan remains left-deep.

The cost model is constructed as a sum of three additive terms designed to capture the dominant latency components observed in disaggregated execution stacks. The total cost of a plan node v with input cardinalities $|R_l|, |R_r|$ and output cardinality $|R_v|$ is expressed as

$$C(v|\ \mathbf{u}_t) = \alpha \cdot f_{\text{op}}(|R_l|, |R_r|, \tau_v) + \beta_t \cdot |R_v| \cdot \rho_v + \gamma \cdot \xi(\mathbf{u}_t, \tau_v) \quad (2)$$

in which τ_v is the physical operator type, $\rho_v \in [0,1]$ is the remote-fetch probability estimated from the buffer state, $\xi(\cdot)$ is a queue-aware penalty triggered when shared storage contention is detected, and α, γ are tier-specific calibration constants while β_t is the only state-dependent coefficient. The first parameter relates to operator calculation at the compute level, the second to the transfer of data from the separate storage level, and the third to queuing time due to storage competition. The remote-fetch probability ρ_v is calibrated from empirical buffer-miss traces collected by deploying PostgreSQL on a Kubernetes cluster with object-store-backed shared storage, so that the parameter reflects realistic disaggregated access patterns rather than synthetic assumptions. Rather than retraining the entire cost surface whenever the cluster reconfigures, only β_t is updated online through an exponentially weighted moving average over the most recent K query executions, expressed as:

$$\beta_{t+1} = (1-\mu)\beta_t + \mu \cdot \tilde{\beta}_t \quad (3)$$

with $\tilde{\beta}_t$ being the empirical ratio between observed and estimated transfer latency on the latest query; the initial value β_0 is obtained from a regression fit on the warm-up trace collected during system bootstrap. The smoothing parameters are fixed at $\mu = 0.15$ and $K = 50$ through grid search on a held-out warm-up slice, providing a balance between responsiveness to runtime shifts and stability under transient noise.

Selectivity inputs feeding the operator term $f_{\text{op}}(\cdot)$ are produced by a supervised deep network that ingests multi-attribute predicate encodings, an approach that has proven able to recover sharp selectivity boundaries that histogram-based methods systematically blur [15]. At the upper levels of the query plans, where cardinality estimation is needed, the estimates are provided by an independently learned data-driven estimator which captures joint distributions over base tables and passes them through joins by applying the technique of sum-product factorization [16]; separate estimators for operators and joins are employed since the distributions learned for these tasks are off by several orders of magnitude in the number of records, and having a single estimator would mean choosing between either a poor approximation of the operator distribution or the join distribution. In terms of the JOB slice, the operator-level estimator obtains a median q-error of 1.7 and a 95th percentile q-error of 9.4; the join-level estimator obtains a median q-error of 2.3 and a 95th percentile q-error of 18.6, which both represent improvements over the histogram-based estimates generated by the PostgreSQL query planner by a whole order of magnitude. The symbols introduced in this section and throughout the remainder of Section 2 are collected in Table 1.

2.2 Heuristic join-order generation module

The design of this heuristic module ensures fast delivery of high-quality plans within a tight latency budget and robustness on small to medium-size queries where a reinforcement learning based method becomes inefficient due to excessive cost. This heuristic module uses both precedence graph-based linearization from IKKBZ, greedy operator interleaving from GOO and adds a cloud-aware tie breaking rule which prefers join plans that minimize remote fetch under the current u_t. The algorithm IKKBZ ranks the dependencies in the precedence graph constructed using join predicates and, in polynomial time, determines the optimal left-deep execution plan based on the rank of a join graph that is acyclic. The algorithm GOO [9] orders the operators greedily by choosing to merge at each step those two subplans that produce the smallest intermediate relation upon performing a join. IKKBZ rank computation is used on acyclic sub-queries in order to get a left-deep skeleton and then GOO-style merging is allowed to generate bushy skeletons when the bushy solution is expected to reduce intermediate size over the default solution by more than a certain $\delta = 0.18$ determined by 5-fold cross-validation on a held out JOB training data slice. The use of the above is preferred over dynamic programming due to the exponential growth in computational complexity of dynamic programming with respect to the number of relations.

An important addition is related to how the heuristic works together with the RL component. The seed plan can no longer be thought of as a warm start that is forgotten after one iteration. Instead, the heuristic plan $H(\pi_0)$ is kept throughout all of training; the subtrees considered the most stable by the heuristic, based on how much the variance of their cost predictions fluctuates when bootstrapping is performed, cannot participate in any action selection. However, self-play breaks the reliance on the reference optimizer, making it incumbent upon the learner to learn about planning quality through interaction with the environment [17]. There is a price paid in terms of training time for this independence, but it makes the learner capable of exploring again structures that work well from an early point in training. However, the freeze mask maintains the reference optimizer, using no samples on these structures. The other deviation lies in the area of intra-query adaptation. Intra-query adaptation techniques partition the budget among various join orders and direct it towards the orders that give out the highest reward signal while executing the query [18]. On the other hand, our technique prunes based on the expert prior during plan generation. This helps retain the advantage of low latency of generating in a single shot and excludes the variance generated by intra-query exploration from multi-tenant queries. There are three key inputs made available from the module to the RL technique presented in Section 2.3.

Table 1. Notation used throughout the formulation of the hybrid optimizer

Symbol	Definition
R_q, P_q, G_q	Relation set, predicate set, and induced join graph for query q
Π_q	Space of semantically equivalent execution plans for q
$\pi^*, C(\cdot)$	Optimal plan and state-aware cost functional
u_t	Cloud-state vector at time t (CPU, network, buffer, node count)
τ_v, ρ_v	Physical operator type and remote-fetch probability at plan node v
$f_{\text{op}}(\cdot)$	Operator-computation term of the cost model, evaluated on the input cardinalities and the operator type
$\lvert R_l \rvert, \lvert R_r \rvert, \lvert R_v \rvert$	Left input, right input, and output cardinalities at plan node v
$\rho_v \cdot \lvert R_v \rvert$	Remote-fetch term of the cost model, scaled by β_t
$\xi(\cdot)$	Queue-aware contention term of the cost model, triggered under shared-storage contention
α	Tier calibration constant on the operator-computation term
β_t	State-dependent coefficient on the remote-fetch term, the only coefficient updated online
γ	Tier calibration constant on the contention term
μ, K	EWMA learning rate and window length for β_t update
$\tilde{\beta}_t, \beta_0$	Empirical transfer-latency ratio on the latest query, and the bootstrap initial value of β_t
$H(\pi_0)$	Heuristic seed plan produced by the IKKBZ-GOO hybrid in Section 2.2
δ	Bushy-merge threshold on the expected reduction of intermediate size (Section 2.2)
$\mathrm{Z}, \mathrm{A}, T, r$	RL state space, action space, deterministic transition, and reward
z_t, a_t	RL state and selected action at step t
$n_q, \lvert E_q \rvert$	Number of relations and number of join-graph edges for query q
Q_θ	Value network with parameters θ
$C_{\text{ref}}(z_t)$	Cost of the heuristic continuation from state z_t
$\hat{q}\text{-err}_t$	Cardinality estimation error at the current plan node
Δ_t	Flow-loss-style penalty, active only when the estimation error can flip a plan decision
η, κ	Modulation coefficient and error threshold of the flow-loss reward term
$\lambda(q)$	Hybrid switching score gating heuristic-only execution versus RL refinement
σ	Logistic function applied to the linear switching score
$\mathbf{w} = (w_1, w_2, w_3, w_4)$	Weight vector of the switching score, trained by logistic regression
$\mathrm{H}(\hat{\mathbf{c}}_q)$	Entropy of the cardinality-estimator output distribution for query q
$\mathrm{Drift}(\mathbf{u}_t)$	Displacement of the current cloud state from the regime of the last training run
θ_l, θ_h	Lower and upper decision thresholds of the gate
k	Number of RL candidate plans ranked against the heuristic seed in the intermediate band

2.3 Reinforcement-learning-based plan refinement

The RL component sets up the abstract problem in Section 2.1 as a finite-horizon deterministic Markov decision process $\langle \mathrm{Z}, \mathrm{A}, T, r \rangle$, in which Z denotes the RL state space and is kept symbolically distinct from the cloud-state vector u_t that conditions the cost model. A state $z_t \in Z$ represents the partially constructed plan, the unjoined relation set, and the current u_t, and is obtained by a tree-LSTM embedding that preserves the structural information lost in flat featurization. The action $a_t \in A_{Z_t}$ picks the next two sub-plans for joining and the join's corresponding physical operator implementation, based on the reduced set of actions dictated by the structural freeze policy in Section 2.2.

The transition function T is deterministic, mapping each state-action pair to a uniquely defined successor state Z_{t+1} because the join graph and operator choice fully determine the resulting partial plan; the episode terminates when all relations in R_q have been consumed, and because every episode has finite length bounded by $n_{q'}$, no discounting is applied so that all intermediate rewards contribute on an equal footing to the final return.

The reward signal does not follow the traditional method of providing reward only upon reaching the end states, since such a method has proven to be the primary factor causing poor performance on multi-table tasks [19]. A dense intermediate reward of the form:

$$r_t = -\log\left(\frac{\hat{C}(z_t, a_t \mid \mathbf{u}_t)}{C_{\text{ref}}(z_t)}\right) + \eta \cdot \mathrm{I}\left[\,|\,\hat{q}\text{-err}_t\,| > \kappa\,\right] \cdot \Delta_t \quad (4)$$

where $C_{\text{ref}}(z_t)$ is the cost of the heuristic continuation from state z_t, $\hat{q}\text{-err}_t$ is the cardinality estimation error at the current node, Δ_t is the flow-loss-style penalty that activates only when the error is large enough to flip a plan decision, and the modulation coefficients are fixed at $\eta = 0.4$ and $\kappa = 2.5$ together with the bushy-merge threshold δ of Section 2.2 through grid search with 5-fold cross-validation on a validation slice disjoint from the test workloads. The use of relative cost allows scaling invariance among different query templates, and this property is needed due to the four order-of-magnitude differences between raw latencies for JOB, TPC-H, and TPC-DS benchmarks. Flow loss-type tuning is employed, since the errors in cardinalities that have no impact on the selected execution plan should not be addressed, and previous research in flow loss-based learning demonstrates that addressing solely those cardinalities leads to improved latency tails in comparison with uniform supervision [20]. It is worth noting that the current approach leverages this idea in its reward mechanism instead of in the cardinality predictor, thus achieving similar results in an automatic fashion.

The value network $Q_\theta(z_t, a_t)$ is modeled using a tree-LSTM encoder and feed-forward head in two layers that utilize experience replay along with the periodically synchronized target network, which proved to be crucial for stabilization of DQN in sequential tasks that involve significant variation in state features [21]. The target network and replay buffer are adopted here for the same stabilization reason, since the cloud-state components of Z_t vary across episodes in a way that non-stationary bootstrapping would otherwise amplify. Replay buffer size remains constant at 10^5, mini-batch size is 64, learning rate equals 5×10^{-4} when applied under Adam algorithm, and target-network update occurs every 500 iterations. A safety filter is implemented on top of the unfiltered policy such that an action can be passed to the system for implementation only if its estimated cost falls within a range of confidence around the heuristic, while any actions falling outside this range pass first through a simulated environment. The agent learns from about 1.2×10^4 episodes, which is a sample budget manageable even without workload traces provided by the company and comparable to the sample budget used in contemporary RL-based optimization research conducted with JOB workloads. The choice of a value-based off-policy agent is made instead of an on-policy or actor-critic agent due to the issue of sample efficiency. The number of episodes needed in the on-policy case would run into millions, but cannot be generated in practice for the task of cloud OLAP workloads. The off-policy framework enables the replay of previous experiences ad infinitum.

2.4 Hybrid Heuristic-Reinforcement Learning Decision Framework

Hybrid architecture is formed by combining heuristic module and RL module using query level gate as opposed to manually configured warm start because both modules perform in different regions of cost quality space and the gate has to be learnable from live traffic. For an incoming query *q*, a switching score:

$$\lambda(q) = \sigma\left(w_1 \cdot \log n_q + w_2 \cdot \log|\,E_q\,| + w_3 \cdot \mathrm{H}(\hat{\mathbf{c}}_q) + w_4 \cdot \mathrm{Drift}(\mathbf{u}_t)\right) \quad (5)$$

where n_q and $|\,E_q\,|$ measure the join-graph size, $\mathrm{H}(\hat{\mathbf{c}}_q)$ is the entropy of the cardinality-estimator output distribution, and $\mathrm{Drift}(\mathbf{u}_t)$ quantifies how far the current cloud state has shifted from the regime under which the value network was last trained. The weight vector $\mathbf{w} = (w_1, w_2, w_3, w_4)^{\cdot}$ together with the upper and lower decision thresholds $\theta_l = 0.32$ and $\theta_h = 0.68$ is trained offline as a logistic regression using supervised labels generated through the execution of the training queries under both heuristic-based path and RL path, marking any instances where the difference between the two paths exceeds a configurable threshold of 8 % as positive labels; thereafter, the model is used in an online setup whereby the parameters are continuously updated based on feedback from regression containment. The thresholds are modeled as a band having a half-width of 0.18 centred at 0.5, where the half-width is the variable being tuned using the validation slice for the set {0.10, 0.15, 0.18, 0.20}, Queries with $\lambda(q) < \theta_l$ are dispatched to the heuristic-only path, queries with $\lambda(q) > \theta_h$ trigger full RL refinement on top of the heuristic seed, and the intermediate band invokes a ranking-style comparison between the heuristic plan and the top-k RL candidates with k=3, with the higher-ranked plan selected for execution.

The 8% label margin threshold was established using the scanning of the following candidate thresholds: {4%, 6%, 8%, 10%, 15%} on the validation slice. The thresholds of 4% and 6% will allow discrepancies that will be present within the run-to-run variation of the shared storage pool, just like the variation that inspires the five-run median process from Section 3.1, and thus, the labels will contain such noise; the rate of positive labels is 71% and 58%, respectively, while the gate F1 of the validation slice becomes 0.74 and 0.81. The margins of 10 % and 15 % limit the positive label rate to 37 % and 24 %, respectively, and exclude those queries which can actually be improved upon through refinement, while the speedup of validation geometric mean using JOB drops to 1.36-fold and 1.30-fold, approaching 1.12-fold which is achieved when the RL part is entirely eliminated. The value of 8 % produces a positive label rate of 46 %, achieves the largest value of gate F1 of 0.86, but still provides a validation geometric mean speedup of 1.38-fold at a mean plan generation time of 5.1 ms, and it is retained on that basis. The proposed system builds upon the wider ML-enabled optimization approach, where the natively cost-based optimizer and the ML-derived part work together via two-way plan ranking to produce reliable plans without losing transparency [22]. In the current design, the cooperation mechanism is determined at the query-level by means of a constant-time gate trained on telemetry data instead of the pair of plans by a learned ranker at the plan-pair level. Moreover, the heuristic function is not used as a safety measure anymore but becomes an integral part of the system, where its decisions are used to restrict the set of actions taken by the RL agent via the freeze mask described in Section 2.2. Regression containment is achieved using a tail-bounded fallback policy that watches the actual execution time of every plan submitted by the RL agent and initiates rollback along with negative example training if the real-time cost surpasses the heuristic prediction multiplied by 1.5. The drift adaptation technique is realized in this design by tying the value-network updates to the difference between the current

workload profile and the distribution seen during training. The architecture of the proposed solution in the full form is shown in Figure 1. An incoming SQL query and the current cloud-state vector u_t are passed to the switching score $\lambda(q)$, computed by the four-feature logistic gate. Every query is routed to the IKKBZ-GOO heuristic generator, which returns the seed plan $H(\pi_0)$ together with the structural freeze mask. Queries scoring below θ_l execute the seed directly. Queries scoring above θ_h are refined by the RL plan-refinement agent, whose action set is restricted by the freeze mask, and the resulting top-k candidates are ranked against the seed by the plan comparator under confidence-band admission control before the final plan π^* is executed. The intermediate band takes the same ranking route with $k=3$. Telemetry returned from the cloud-native execution engine drives the tail-bounded rollback policy, the workload-drift retraining trigger, and the online EWMA update of β_t inside the state-aware cost model. Solid arrows mark the deterministic flow traversed by every query; dashed arrows mark conditional and feedback flow, comprising the freeze-mask and seed-plan hand-off, the top-k candidate hand-off, the drift signal, and the rollback path.

3. Results

3.1 Experimental setup and public benchmarks

The experimentation uses three public analytic benchmarks chosen such that each covers all major analytical query workload regimes. In the case of the JOB based on the practical IMDb database, it comprises 113 queries across 21 base relations, with each having between 3 to 16 joins and a very skewed value distribution to bring out the multiway q-error tail causing the plan selections in classical query optimizers to be catastrophic [23]. TPC-H with a scale factor of 10 provides 22 standard decision support templates across an 8-relation database with mainly uniform distributions, providing a baseline with respect to which the effects of any skews observed can be attributed. TPC-DS with a scale factor of 10 includes 99 templates using a snowflake schema with possible 24-way joins, constituting the most challenging environment where plan cardinality space size and selectivity ambiguity coexist. All experiments were conducted on a Kubernetes cluster consisting of 8 compute nodes with 16 vCPUs and 64 GB RAM each, utilizing a shared storage object store, with PostgreSQL 15 used as the execution engine and modified using the pg_hint_plan extension to accept plan hints from outside. Every query is run five times post-caching, and only the median latency is recorded to remove any transient noise in the shared storage level. Relevant workloads statistics to the following analysis are provided in Table 2.

Table 2. Public benchmarks used in this study

Benchmark	Relations	Queries	Join Cardinality Range	Data Volume	Predicate Skew
JOB (IMDb)	21	113	3–16	3.6 GB	High
TPC-H SF10	8	22	2–8	10 GB	Low
TPC-DS SF10	24	99	4–24	10 GB	Moderate

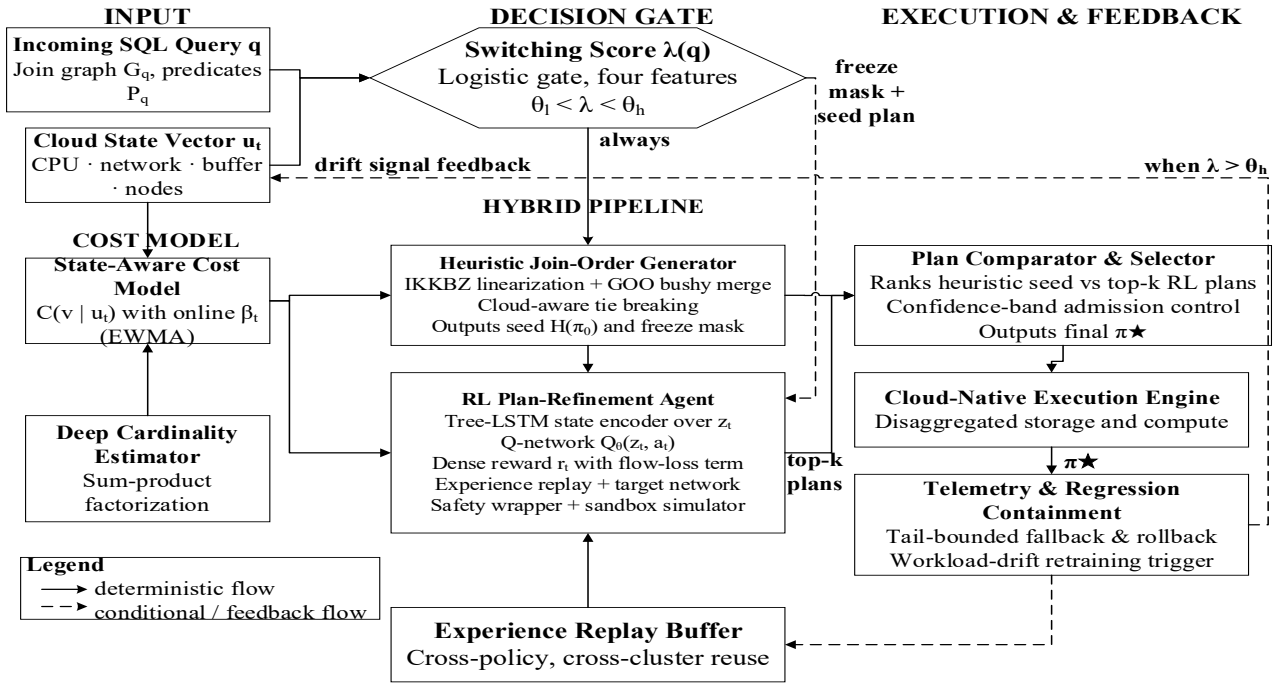

Figure 1. Overall architecture of the hybrid heuristic-reinforcement learning optimizer

The benchmark workload in each case is divided into a 70/15/15 stratified sample, where the strata are based on join cardinality, resulting in an equal distribution of queries with low, medium, and high cardinalities in the training, validation, and test sets. In this research, all parameters introduced in Section 2 are optimized based on the validation set, while all figures discussed in Sections 3.2 to 3.4 are calculated based on the test set only. Pairwise comparisons between optimizers use the Wilcoxon signed-rank test on log-latency values. A significance level of $p<0.01$ is applied when more than two optimizers are compared, and a result with $p>0.05$ is treated as statistically insignificant. Each test is accompanied by the matched-pairs rank-biserial correlation as an effect size, with 95 % confidence intervals obtained from 10^4 bootstrap resamples, so that practical significance can be assessed alongside the p-value. The unit of analysis is the query rather than the training run, and every confidence interval and error bar reported in Section 3 therefore quantifies variation across queries.

The set of baselines that the hybrid optimizer is compared to is purposefully limited to the four classes that altogether span the design space that is relevant for this work. The native cost-based optimizer of PostgreSQL is used as a benchmark for the performance comparison against the dynamic programming one, keeping in mind that the native optimizer consumes all its dynamic programming resources for TPC-DS query templates that involve more than 12 input tables and relies on its own genetic algorithm for solving such templates, a feature that is maintained in the results obtained. The Genetic Query Optimizer (GEQO), when run in a stand-alone fashion, represents the heuristics family. Tree-LSTM-based reinforcement learning model without the use of heuristics, i.e., a learning only baseline, is an example of pure RL, and it is a reproduction of the open-source design. The learn-based optimizer with focus on production belongs to the Hint-Set Bandit (HSB) family and is an adaptation of the Bao framework [13]. The arm set consists of possible combinations of six policy flags of the PostgreSQL planner, including enable_nestloop, enable_hashjoin, enable_mergejoin, enable_indexscan, enable_seqscan, and enable_indexonlyscan. Any combination resulting in neither join method nor scan method being enabled is excluded from the consideration, producing 48 valid hint sets. The bandit chooses one hint set for a query, and then the native planner generates a plan based on that restriction. The bandit is trained on the same budget of 1.2×10^4 episodes mentioned in Section 2.3, which is approximately 250 arms per episode.

Each learned baseline is trained using an equal resource allocation of 1.2×10^4 episodes as stated in Section 2.3 to rule out any differences attributed to differences in learning resources. Performance measurements include end-to-end run-time, time taken to generate plans, geometric mean speedup relative to the PostgreSQL native baseline, and tail latency at the 95th percentile. Reproducibility is justified as follows. All three benchmark datasets are available publicly, as the IMDb data and JOB query set are part of the dataset published by Leis et al. [23], and TPC-H and TPC-DS benchmarks are produced from the TPC specification, while PostgreSQL 15 and the pg_hint_plan module are open source projects. Stratified 70/15/15 split is done using random seed 7, and hence can be reproduced precisely using publicly available data without knowledge of the optimizer implementation, while the five folds used for cross validation are sampled using the same seed. Training of the reinforcement learning model is performed using global seeds 42 for Python, NumPy, and PyTorch, which ensures that the sampling of the experience replay buffer and exploration strategy are fixed, while bootstrap sampling is done using the same global seed 42. Training results are obtained using a single training run using these seeds, and intervals provided in Section 3 reflect variability between different queries rather than different runs of the training process. The optimizer implementation is available from the corresponding author on reasonable request.

Both training and testing are performed using CPUs only. The agent is trained on one compute node from the cluster mentioned above, which has 16 vCPUs and 64 GB RAM, in the context of CPU PyTorch backend, without using any GPU/TPU support. Queries are evaluated on the same eight-node cluster. Only one agent is trained simultaneously on all three benchmarks, since the reward formulation in section 2.3 allows for that. Budget of 1.2×10^4 episodes translates into mean elapsed time cost of 1.9 s per episode for a total of 6.4 h, the bulk of which time is used to enumerate and encode possible sub-plan pairs compared to computing gradients, hence the reason the model can be trained using that budget without hardware acceleration. Table 3 lists every tuned hyperparameter together with its search grid and final value. Grid search over η and κ is conducted under a reduced budget of 2×10^3 episodes per fold, and the final agent is retrained at the full budget once the values are fixed, while δ affects only the heuristic path and is therefore tuned independently at negligible cost.

Table 3. Tuned hyperparameters, search grids, final values, and tuning protocol

Parameter	Search grid	Final value	Tuning protocol
μ	{0.05, 0.10, 0.15, 0.20, 0.30}	0.15	Grid search on held-out warm-up slice; cost model only
K	{20, 50, 100, 200}	50	Grid search on held-out warm-up slice; cost model only
δ	{0.10, 0.14, 0.18, 0.22, 0.26}	0.18	5-fold CV on held-out JOB training slice; heuristic path only
η	{0.2, 0.4, 0.6, 0.8}	0.4	5-fold CV on validation slice, joint grid with κ, reduced budget 2×10^3 episodes
κ	{1.5, 2.0, 2.5, 3.0}	2.5	5-fold CV on validation slice, joint grid with η, reduced budget 2×10^3 episodes
Gate band half-width	{0.10, 0.15, 0.18, 0.20, 0.25}	0.18, giving $\theta_l = 0.32$ and $\theta_h = 0.68$	Validation slice; band centred on 0.5
Labeling margin	{4 %, 6 %, 8 %, 10 %, 15 %}	8 %	Validation slice; gate F1 and JOB speedup, Section 2.4

The remaining settings are fixed by convention rather than tuned and are stated in Sections 2.3 and 2.4, namely a replay buffer of 10^5 transitions, a mini-batch size of 64, an Adam learning rate of 5×10^{-4}, target-network synchronization every 500 iterations, an episode budget of 1.2×10^4, $k=3$ ranked candidates within the intermediate gate band, a rollback margin of 1.5, and a discount factor of 1.0.

3.2 End-to-end query performance comparison

The performance analysis on the three datasets illustrates that there is a workload-specific effect of latency improvement compared to the PostgreSQL native plan. Specifically, for JOB where the workload comprises complex multi-way join operations and heavily skewed predicates, the hybrid optimizer achieves a median latency reduction of 27.6% and achieves a geometric mean speedup of 1.38× (with p<0.001 versus native) compared to 1.18× and 1.11× for the baselines. For TPC-DS, the associated median reduction in latency is 21.9% with a geometric mean speedup of 1.28× (p < 0.001), whereas the baseline with only learning is 1.14× and the hint-set bandit is 1.09×. For TPC-H, the properties of this workload lead to a very different ordering, with the hybrid optimizer improving median latency by 6.2% over the native plan, obtaining a geometric mean speedup of 1.07× compared with 1.06× for GEQO, which is statistically no different from the hybrid (p = 0.41). For the aforementioned comparison, the value of matched-pairs rank-biserial correlation coefficient is 0.16, which has a 95 % confidence interval of [−0.19, 0.48]. This is negligible in magnitude, and the confidence interval contains zero; hence, the two setups are practically identical for TPC-H and not just statistically undifferentiable.

The effect sizes for JOB and TPC-DS are both large, as given in Table 4. This behavior fits into the reasoning behind the heuristic design principle of classical heuristics being nearly optimal in uniformly distributed workload scenarios where join depth is constrained, while the utility of learned plan optimization would be more applicable to the less-well-behaved queries typical of JOB and TPC-DS than TPC-H. Generation time of plans follows another ranking pattern, which directly affects production application. Hybrid optimizer has median times of 5.1, 4.6, and 6.4 ms for processing a query on JOB, TPC-H, and TPC-DS datasets, respectively, where the first time is split into 1.3 ms used to generate a heuristics seed and 3.8 ms to infer values from the value network. Such times can be compared to 11.7 ms, 10.2 ms, and 14.8 ms for the median times required to execute the learning-only algorithm, based on three different benchmarks, each of which involves an action space that requires a deeper rollout sequence, as well as 0.9 ms, 0.7 ms, and 1.3 ms for the median times for the DP enumerator. For all three metrics, the hybrid optimizer terminates plan search within finite time without ever resorting to any fallback strategies inside itself; this is because of the gating process explained in Section 2.4 for sending complicated queries to the RL route only when $\lambda(q) > \theta_h$. The full latency comparison can be seen in Figure 2. Figure panels (A), (B), and (C) demonstrate the end-to-end latency of the query latency distribution on JOB, TPC-H, and TPC-DS, respectively, in a logarithmic scale, in the hybrid optimizer versus the four baselines, along with the geometric mean speed-up and statistical significance compared to the native plan in each figure panel. Figure panels (D), (E), and (F) represent the median plan generation time of the five configurations above, together with error bars showing the 95% confidence interval calculated using bootstrapping technique. The annotation in figure panel (F) indicates that the native configuration switches to GEQO in 17 out of 99 TPC-DS templates.

3.3 Adaptability under cloud-native elastic scenarios

The adaptability assessment focuses on the performance of the optimizer under three distinct conditions relevant to cloud environments yet missing in static benchmarking assessments: elastic scale-out, where the computing layer increases from four to eight nodes while the workload is running; node jitter, where every few seconds a particular node experiences 200ms latency spikes to simulate noisy neighbor effects; and cold-start, where the cache is emptied prior to every query execution to ensure a fully remote fetch penalty for every page access. In the elastic scale-out condition, the hybrid optimizer achieves recovery to within 5% of its initial median latency after 22 seconds of the scaling operation compared to 41 seconds for the learning-only approach. Recovery is determined through a rolling median of end-to-end latency calculated using the last twenty queries.

This workload is executed constantly during the scale out event, and the rolling median is determined after each query completion. The recovery time is the time period from when the scale out begins until when the rolling median becomes smaller than 1.05 times the value before the scale out and stays below that point through twenty queries. This is true for all configurations, including the 41 s that was found in the learning-only method. Such an approach is consistent with the desired behavior where the update of the state-dependent parameter β_t through the online process outlined in Section 2.1 causes the ratio to be propagated within the cost function without the need for value network training while at the same time showing ordinary variations in the plan churn rate.

Table 4. Effect sizes and significance of the hybrid optimizer against each baseline

Benchmark	Comparison	Rank-biserial r [95 % CI]	p
JOB	Hybrid vs. Native	0.94 [0.82, 1.00]	< 0.001
JOB	Hybrid vs. GEQO	0.79 [0.53, 0.94]	< 0.01
JOB	Hybrid vs. HSB	0.71 [0.41, 0.90]	< 0.01
JOB	Hybrid vs. Learn-only	0.62 [0.28, 0.85]	0.014
TPC-DS	Hybrid vs. Native	0.79 [0.70, 0.86]	< 0.001
TPC-DS	Hybrid vs. GEQO	0.66 [0.55, 0.75]	< 0.001
TPC-DS	Hybrid vs. HSB	0.58 [0.45, 0.69]	< 0.001
TPC-DS	Hybrid vs. Learn-only	0.52 [0.38, 0.64]	< 0.001
TPC-H	Hybrid vs. GEQO	0.16 [−0.19, 0.48]	0.41

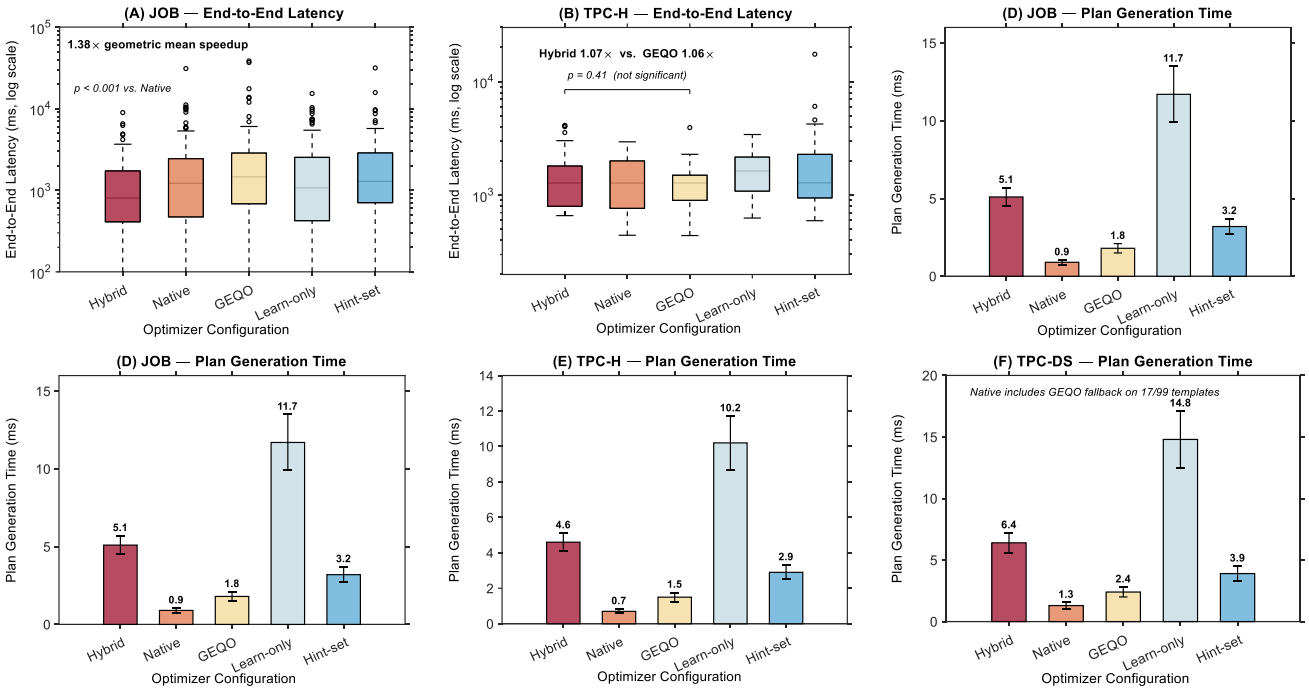

Figure 2. End-to-end latency and plan-generation time across the three benchmarks.

When jittering nodes, the proposed optimizer incurs an overhead of 21% at the 95th percentile compared to the latency at the undisturbed operating point; the learning-based optimizer, however, has an overhead of 33%, followed by GEQO with 38%, and Postgres's own optimizer with 42%, while the hint-set bandit incurs 29%. The increase in distance between the hybrid approach and the static benchmarks in this case study correlates with the incorporation of network bandwidth and buffer-pool hit ratio in the cloud-state vector u_t, allowing the hybrid strategy to divert planning towards less responsive operators in regard to remote access latency. However, the performance difference for the cold start scenario is lower compared to other steady state performance tests. The cloud-aware tie breaking strategy used in the heuristic module allows for a performance improvement of 16.8% over the native approach, compared with 9.3 % for the hint-set bandit, 7.2 % for the learning-only baseline, and 4.1 % for GEQO. The narrow range seen here is because the costs related to data transfers outweigh the shape of the query plan at the storage layer due to a full cold buffer. The overhead and recovery graphs for the scenario are provided in Figure 3. In this Figure, (A) shows the plot of median end-to-end latency with respect to wall-clock time after the scale-out operation from four to eight compute nodes; the latency is normalized to the value before scaling. The shaded region corresponds to the 5% recovery window mentioned in the text. Figure 3 (B) shows the additional latency penalty experienced due to node jitter compared to the undisturbed operating point. The dotted line is the hybrid baseline. In addition, (C) depicts the improvement in cold start latency compared to the native plan, which acts as the base case of zero. Panel (D) summarizes normalized performance across the three scenarios for all five optimizer configurations, with higher scores indicating better behavior.

3.4 Ablation analysis of key components

The ablation study will identify the effect of five architectural choices made by the network in Section 2, namely the heuristic prior with the structure-freeze mask, the flow loss style reward modulator, the query level gating, the on-line update rule for β_t, and the RL refinement path as a whole. Disabling the heuristic prior and simplifying the setup to the learning-only system results in an increase in JOB latency by 17.0% and a doubling of the time required for training convergence to 14.1 hours from 6.4 hours using the same hardware, which indicates that the freeze mask acts as a sample efficiency booster.

Three further measurements support that reading. The full hybrid reaches 95 % of its final validation performance after 4.1×10^3 episodes, whereas the learning-only configuration requires 9.6×10^3 episodes; the resulting sample ratio of 2.34× is consistent with the 2.20× wall-clock ratio, the small difference arising because each episode also becomes more expensive to enumerate once the freeze mask no longer restricts the action set. Removal of the freeze mask raises the JOB 95th-percentile latency by 26.3 %, roughly one and a half times the increase observed at the median, and widens the interquartile range of JOB plan generation time from 1.5 ms to 5.8 ms. The proportion of executed plans that trigger the rollback policy of Section 2.4 during training rises from 0.8 % to 6.2 %, which is the most direct evidence that the mask suppresses catastrophic exploration rather than merely accelerating convergence. The elimination of the flow-loss modulation effect while keeping the relative-cost reward effect results in an increase in 95th percentile latency in TPC-DS tests of 11.4 %; this latency increase occurs mostly among the joins within the range from 16 ways to 24 ways for reasons that estimation errors can change plans; the increase in average latency, however, is relatively small at 4.1 %.

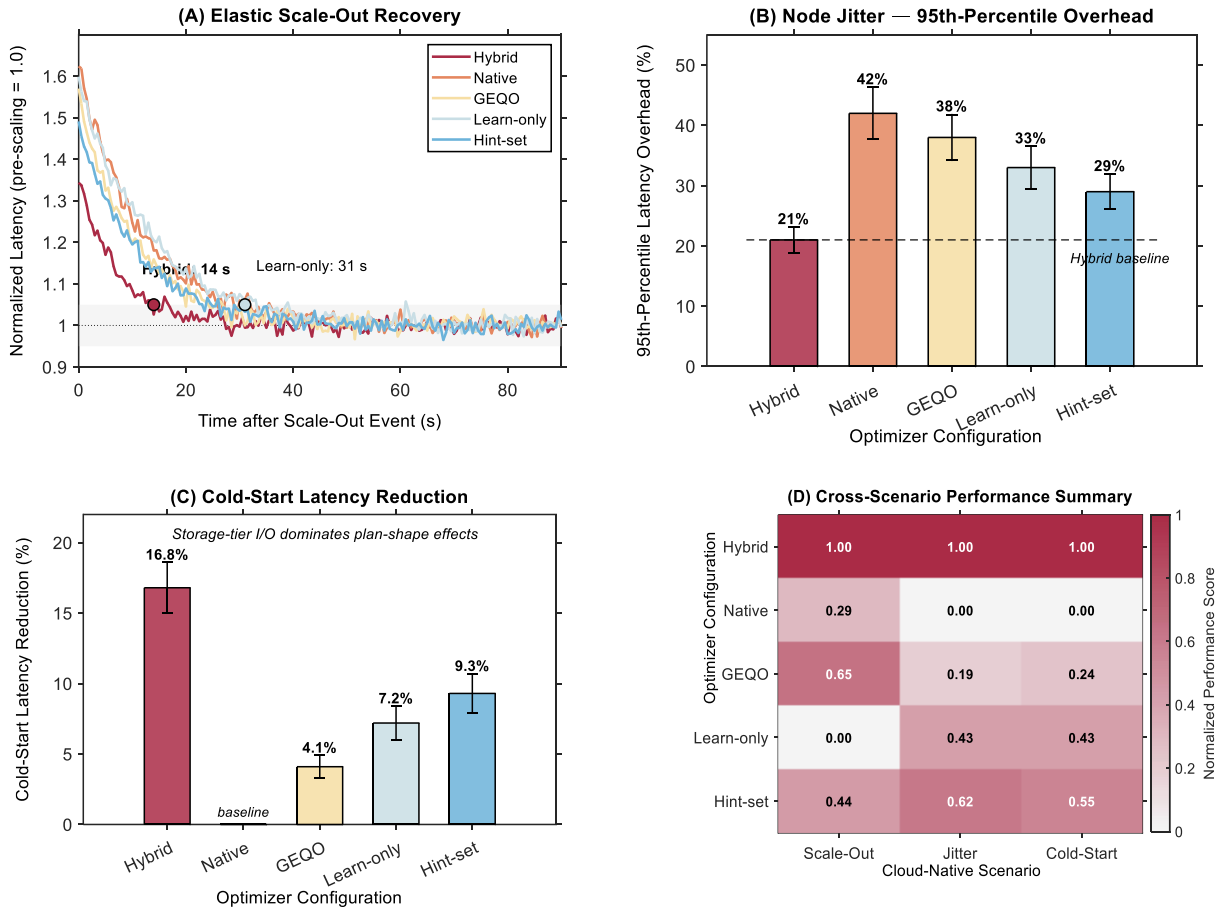

Figure 3. Optimizer recovery under elastic scale-out, node jitter, and cold-start scenarios

The elimination of the query-level gate and the imposition of the entire RL-based plan optimization process on all queries leads to a 7.2% rise in the average plan generation time without any observed improvement in execution delay, thereby validating the cost-quality complementarity of Section 2.4. However, since the application of bootstrap rather than using updates in EWMA for the value of β_t results in a reduction of the time gain due to elastic scale out from 22 to 35 seconds, while at the same time increasing the cold start time by 8.7%, this proves the criticality of the runtime updates that enable the information about runtime conditions to enter into the cost model without being trained. In the absence of the RL path entirely and sending all the queries through the heuristic configuration, there is no gain over multiway JOBs. The contributions of the individual algorithms are highlighted in Figure 4.

In this figure, Panel (A) depicts the normalized validation latency of the complete hybrid system and of the five ablations in the context of wall clock training time with the convergence point of the complete hybrid system at 6.4 hours and the one of the freeze-mask ablations at 14.1 hours indicated in the graph. Panel (B) depicts the JOB geometric mean speedup observed under each ablation with the dashed line indicating the speedup achieved by the complete hybrid and the annotation pointing out the reduction to 1.12× after RL removal. The plots in Panel (C) illustrate the TPC-DS 95th percentile and average latency normalized to the complete hybrid with solid bars for the 95th percentile and faded bars for the average latency.

Panel (D) illustrates the impact of each ablation relative to five different performance metrics with dashes marking the metrics not relevant to a specific ablation.

4. Discussion

4.1 Interpretation of the workload-dependent performance patterns

The trend in performance noted in the three benchmarks can be seen to exhibit a clear order in which the relative gain achieved by the hybrid optimizer depends upon the proportion of multi-way join and predicate skew found, achieving a geometric mean speed-up of 1.38× on the Join Order Benchmark, 1.28× on TPC-DS, and 1.07× on TPC-H in which case the difference between it and GEQO, at 1.06×, is statistically insignificant. This ordering arises not from any bias within the benchmark itself but as a natural consequence of the plan-space geometry in the absence of multi-way joins and uniform distributions. The ablative experiment discussed in Section 3.4 validates the above explanation because ablating the reinforcement learning route would preserve almost all TPC-H improvements, but the JOB speed-up would go down from 1.38× to 1.12×, directly proving that the two techniques complement each other and not substitute for each other. The 11.4 % increase in TPC-DS 95th percentile latency that follows removal of the flow-loss term also points to cardinality estimation accuracy being the bottleneck for learning-based optimizers, as has been shown before with significant improvements in the tail of q-error with multi-way error modeling [24].

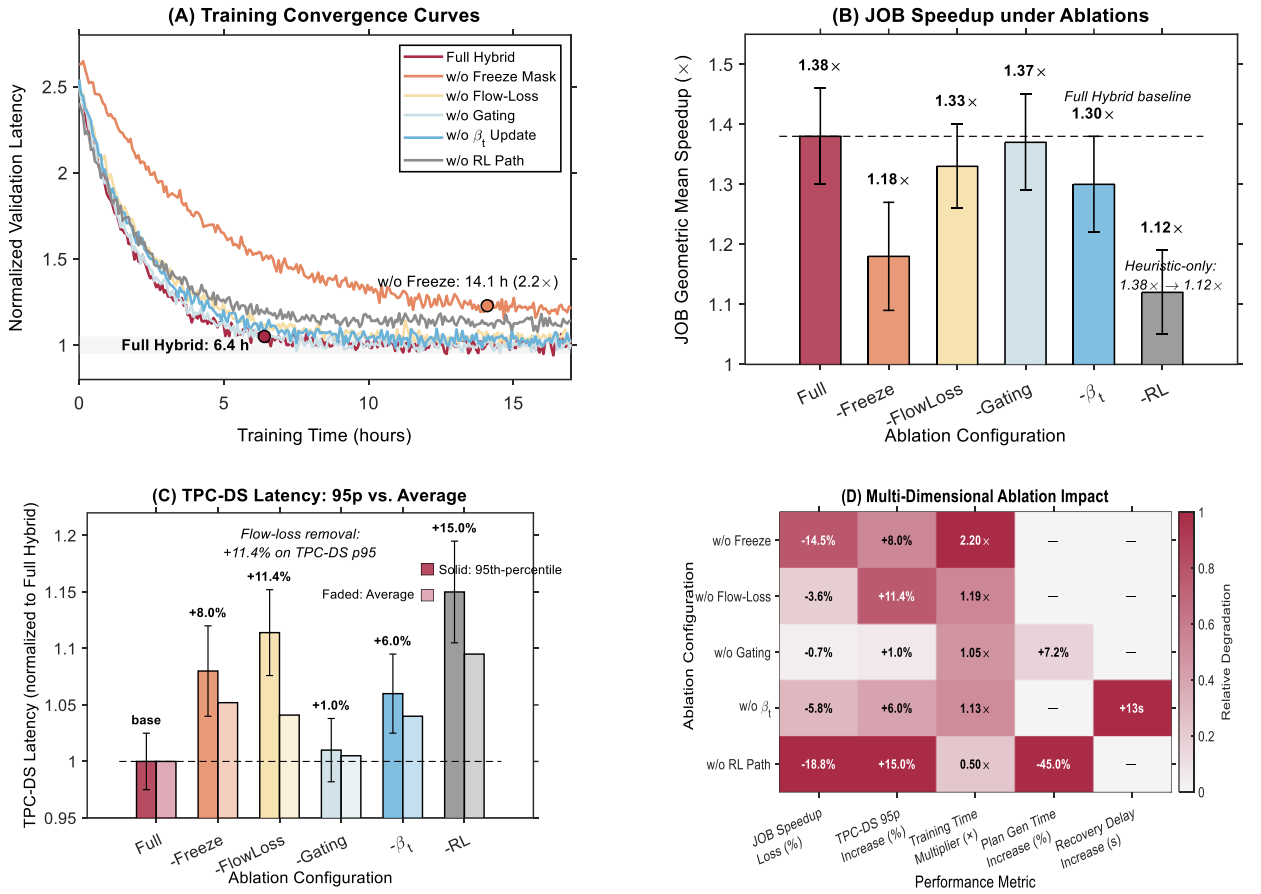

Figure 4. Ablation effects on training convergence and query performance

The relatively smaller 16.8% improvement in cold-start query latency highlights the physical limitations of optimization at the level of execution plans in the case of I/O-bound workloads.

4.2 Comparison with prior hybrid and learned optimizers

The speedup of 1.38× in JOB is qualitatively similar to the improvements seen in previous learning-to-rank techniques that utilize a native cost function together with a learned ranking model [25]. This demonstrates the repeatability of the heuristic plus learned technique on imbalanced analytical workloads. There are three exceptions to the general consensus in the literature regarding learned optimizers that should be discussed explicitly. One of the deviations pertains to coverage, given that while many learning-enabled optimizers claim universal improvements, this research finds that the value of the learning trajectory is zero on TPC-H because the benefit occurs only in cases where there are cardinality mistakes that change plans. A second difference pertains to the cold start problem, as most research on learned optimizers does not examine cold caches or finds consistently large gains while here the 16.8% improvement in cold starts considers the dominance of storage tier I/O cost. A third difference lies in the use of experts' prior knowledge, where current self-training schemes seek to avoid using classic heuristics; however, in this case, the structural freeze mask results in a reduction of training time from 14.1 hours to 6.4 hours, thereby providing the biggest gain towards JOB optimization.

The difference between TPC-H and plan-level Monte Carlo arbitrator techniques [26] is due to a conscious trade-off of some of the benefits achievable for reduced costs in planning effort and improved robustness during execution, as evidenced by the 21% overhead due to node-jitter in the hybrid setup compared to 38% and 42% for the genetic and native baselines, respectively. This improvement in training efficiency is especially commendable when contrasted with join order learning using graphs, which suffer from higher convergence latencies [27]. Although the regression containment approach is based on tail risk control in production deployment [28], it differs by combining rollback with the online updating of β_t, thus making the 22 seconds required for elastic-scale-out recovery an example of pro-active adaptation, whereas the 13 seconds better performance compared to the β_t-frozen case demonstrate its importance. Fast/slow decoupling in low-dimensional cost and high-dimensional policy parameterization is in accordance with general remarks concerning the learnability of database elements [29], while the four-feature interpretable gate reflects workload drift principles that do not support deeply coupled end-to-end architectures in non-stationary environments [30].

4.3 Limitations and future directions

The workloads used for training and testing are extracted from three publicly available datasets, where the query templates included do not adequately reflect the shape of queries present in the production multi-tenant environment, and the gating system falls back to a pure

heuristics approach once the difference is significant. Additive decomposition of cost is acceptable in case of moderate levels of concurrency; however, it may understate the interaction between the costs associated with operators, remote accesses, and queue overhead for extremely high levels of concurrency, which has not been measured in this paper; additive decomposition can therefore be viewed as untested at extremely high concurrency beyond the moderate concurrency used in Section 3, while the eight-node system environment fails to reveal the responsiveness of exponentially updating β_t with larger scales. However, whether the fixed window K = 50 maintains such responsiveness for larger compute tiers beyond eight nodes has yet to be shown, and a larger node count might require that the window size be adjusted accordingly. Another limitation is due to the width of joins being used, because TPC-DS queries have the widest 24-way joins performed; hence, if queries significantly wider than that were to be processed, it would increase the size of the actions represented by tree-LSTM at each step and decrease the percentage of the plan that could be frozen, and both of these effects need to be studied. The following are some avenues for future research in extending the current model: making the structural freeze mask replaceable with a learnable soft constraint with its magnitude depending on the complexity of the query, augmenting the cloud-state representation with information about different workloads and tenants to facilitate optimization in multi-tenant cloud systems, and upgrading the regression containment technique to risk estimation before the deployment of learned optimizers such that the performance barrier remains unaltered.

5. Conclusion

The proposed approach is that of a hybrid approach to multi-table join ordering based on heuristic and RL methods where a cost model with dynamic cost coefficients, the generation of IKKBZ-GOO heuristic with structural freeze mask, a logistic gate at the level of individual query obtained from production telemetry, and a regression containment fallback scheme are considered. Evaluation of the proposed approach was carried out using Join Order Benchmark, TPC-H and TPC-DS with PostgreSQL deployed on eight nodes in Kubernetes with object store shared storage. In the three benchmarks, the suggested method produces speedups by a geometric average of 1.38× for JOB, 1.28× for TPC-DS, and 1.07× for TPC-H, where median time to generate plans is between 4.6 ms to 6.4 ms across all three benchmarks, and training is done in just 6.4 hours, whereas elastic scale-out in 22 seconds and 21% overhead due to node jittering surpass both native and learning methods in the face of runtime disturbance. The results demonstrate the feasibility of using heuristics and learning together in an integrated fashion rather than considering them as options, as well as presenting a usable design of a heuristic optimizer with adaptive decision boundaries based on the cloud runtime. The two implications for practitioners are as follows. The gate keeps reinforcement learning confined to those queries where refinement will pay off, while the heuristic approach takes care of the rest, thus the entire framework can be applied on top of an existing cost-based query optimizer without any change, and the plan generation budget of a production workload is left undisturbed for the queries where there is no benefit in doing refinement. In the online updating of β_t clustering reconfiguration gets included in the cost model rather than kicking off another training round, meaning that the cost of the optimization process happens in just one training process of 6.4 hours in commodity CPUs, rather than continuously maintaining the process due to growth in the cluster size, which is the factor that makes it economically viable to keep the learned part.

Ethical issue

The authors are aware of and comply with best practices in publication ethics, specifically with regard to authorship (avoidance of guest authorship), dual submission, manipulation of figures, competing interests, and compliance with policies on research ethics. The authors adhere to publication requirements that the submitted work is original and has not been published elsewhere. All survey participants provided informed consent prior to participation, and the anonymity and confidentiality of all respondents were strictly maintained throughout the research process.

Data availability statement

The manuscript contains all the data. However, additional data will be provided by the corresponding author upon reasonable request.

Conflict of interest

The authors declare no potential conflict of interest.

References

[1] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, “Amazon Aurora: Design considerations for high throughput cloud-native relational databases,” in Proc. 2017 ACM Int. Conf. Management of Data (SIGMOD '17), Chicago, IL, USA, May 2017, pp. 1041–1052, doi: https://doi.org/10.1145/3035918.3056101.

[2] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade, “Socrates: The new SQL Server in the cloud,” in Proc. 2019 Int. Conf. Management of Data (SIGMOD '19), Amsterdam, The Netherlands, Jun.–Jul. 2019, pp. 1743–1756, doi: https://doi.org/10.1145/3299869.3314047.

[3] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang, and J. Tong, “PolarDB Serverless: A cloud native database for disaggregated data centers,” in Proc. 2021 Int. Conf. Management of Data (SIGMOD '21), Virtual Event, China, Jun. 2021, pp. 2477–2489, doi: https://doi.org/10.1145/3448016.3457560.

[4] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, “The Snowflake elastic data warehouse,” in Proc. 2016 Int. Conf. Management of Data (SIGMOD '16), San Francisco, CA, USA, Jun. 2016, pp. 215–226, doi: https://doi.org/10.1145/2882903.2903741.

[5] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, E. Hotinger, Y. Leshinksy, J. Liang, M.

McCreedy, F. Nagel, I. Pandis, P. Parchas, R. Pathak, O. Polychroniou, F. Rahman, G. Saxena, G. Soundararajan, S. Subramanian, and D. Terry, "Amazon Redshift re-invented," in Proc. 2022 Int. Conf. Management of Data (SIGMOD '22), Philadelphia, PA, USA, Jun. 2022, pp. 2205–2217, doi: https://doi.org/10.1145/3514221.3526045.

[6] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "CockroachDB: The resilient geo-distributed SQL database," in Proc. 2020 ACM SIGMOD Int. Conf. Management of Data (SIGMOD '20), Portland, OR, USA, Jun. 2020, pp. 1493–1509, doi: https://doi.org/10.1145/3318464.3386134.

[7] Z. Si, W. Wei, B. Li, W. Feng, "Analysis of DNA Image Encryption Effect by Logistic-Sine System Combined with Fractional Chaos Stability Theory," Journal of Imaging Science & Technology, vol. 64, no. 4, 2020, doi: https://doi.org/10.2352/J.ImagingSci.Technol.2020.64.4.040413.

[8] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in Proc. 1979 ACM SIGMOD Int. Conf. Management of Data (SIGMOD '79), Boston, MA, USA, May 1979, pp. 23–34, doi: https://doi.org/10.1145/582095.582099.

[9] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," VLDB J., vol. 6, no. 3, pp. 191–208, Aug. 1997, doi: https://doi.org/10.1007/s007780050040.

[10] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," Proc. VLDB Endow., vol. 12, no. 11, pp. 1705–1718, Jul. 2019, doi: https://doi.org/10.14778/3342263.3342644.

[11] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-Å. Larson, D. Kossmann, and R. Acharya, "Qd-tree: Learning data layouts for big data analytics," in Proc. 2020 ACM SIGMOD Int. Conf. Management of Data (SIGMOD '20), Portland, OR, USA, Jun. 2020, pp. 193–208, doi: https://doi.org/10.1145/3318464.3389770.

[12] X. Yu, G. Li, C. Chai, and N. Tang, "Reinforcement learning with Tree-LSTM for join order selection," in Proc. 2020 IEEE 36th Int. Conf. Data Engineering (ICDE), Dallas, TX, USA, Apr. 2020, pp. 1297–1308, doi: https://doi.org/10.1109/ICDE48307.2020.00116.

[13] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in Proc. 2021 Int. Conf. Management of Data (SIGMOD '21), Virtual Event, China, Jun. 2021, pp. 1275–1288, doi: https://doi.org/10.1145/3448016.3452838.

[14] Z. Si, W. Wei, W. Feng, B. Li, "Development and Construction of Intelligent Security Monitoring System," in Proc. IEEE 3rd Int. Conf. On Information Systems and Computer Aided Education (ICISCAE), pp. 335-338, Sep. 2020, doi: https://doi.org/10.1109/ICISCAE51034.2020.9236915.

[15] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das, "Deep learning models for selectivity estimation of multi-attribute queries," in Proc. 2020 ACM SIGMOD Int. Conf. Management of Data (SIGMOD '20), Portland, OR, USA, Jun. 2020, pp. 1035–1050, doi: https://doi.org/10.1145/3318464.3389741.

[16] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "DeepDB: Learn from data, not from queries!," Proc. VLDB Endow., vol. 13, no. 7, pp. 992–1005, Mar. 2020, doi: https://doi.org/10.14778/3384345.3384349.

[17] Z. Yang, W.-L. Chiang, S. Luan, G. Mittal, M. Luo, and I. Stoica, "Balsa: Learning a query optimizer without expert demonstrations," in Proc. 2022 Int. Conf. Management of Data (SIGMOD '22), Philadelphia, PA, USA, Jun. 2022, pp. 931–944, doi: https://doi.org/10.1145/3514221.3517885.

[18] I. Trummer, J. Wang, Z. Wei, D. Maram, S. Moseley, S. Jo, J. Antonakakis, and A. Rayabhari, "SkinnerDB: Regret-bounded query evaluation via reinforcement learning," ACM Trans. Database Syst., vol. 46, no. 3, Art. no. 9, pp. 1–45, Sep. 2021, doi: https://doi.org/10.1145/3464389.

[19] M. Ramadan, A. El-Kilany, H. M. O. Mokhtar, and I. Sobh, "RL_QOptimizer: A reinforcement learning based query optimizer," IEEE Access, vol. 10, pp. 70502–70515, Jul. 2022, doi: https://doi.org/10.1109/ACCESS.2022.3187102.

[20] P. Negi, R. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh, "Flow-Loss: Learning cardinality estimates that matter," Proc. VLDB Endow., vol. 14, no. 11, pp. 2019–2032, Jul. 2021, doi: https://doi.org/10.14778/3476249.3476259.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: https://doi.org/10.1038/nature14236.

[22] X. Chen, H. Chen, Z. Liang, S. Liu, J. Wang, K. Zeng, H. Su, and K.-L. Tan, "LEON: A new framework for ML-aided query optimization," Proc. VLDB Endow., vol. 16, no. 9, pp. 2261–2273, May 2023, doi: https://doi.org/10.14778/3598581.3598597.

[23] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?," Proc. VLDB Endow., vol. 9, no. 3, pp. 204–215, Nov. 2015, doi: https://doi.org/10.14778/2850583.2850594.

[24] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica, "NeuroCard: One cardinality estimator for all tables," Proc. VLDB Endow., vol. 14, no. 1, pp. 61–73, Sep. 2020, doi: https://doi.org/10.14778/3421424.3421432.

[25] R. Zhu, W. Chen, B. Ding, X. Chen, A. Pfadler, Z. Wu, and J. Zhou, "Lero: A learning-to-rank query optimizer," Proc. VLDB Endow., vol. 16, no. 6, pp. 1466–1479, Feb. 2023, doi: https://doi.org/10.14778/3583140.3583160.

[26] X. Yu, C. Chai, G. Li, and J. Liu, "Cost-based or learning-based? A hybrid query optimizer for query plan selection," Proc. VLDB Endow., vol. 15, no. 13, pp. 3924–3936, Sep. 2022, doi: https://doi.org/10.14778/3565838.3565846.

[27] J. Chen, G. Ye, Y. Zhao, S. Liu, L. Deng, X. Chen, R. Zhou, and K. Zheng, "Efficient join order selection learning with graph-based representation," in Proc. 28th ACM SIGKDD Conf. Knowledge Discovery and Data Mining (KDD '22), Washington, DC, USA, Aug. 2022, pp. 97–107, doi: https://doi.org/10.1145/3534678.3539303.

[28] L. Weng, R. Zhu, D. Wu, B. Ding, B. Zheng, and J. Zhou, "Eraser: Eliminating performance regression on learned query optimizer," Proc. VLDB Endow., vol. 17, no. 5, pp. 926–938, Jan. 2024, doi: https://doi.org/10.14778/3641204.3641205.

[29] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui, "Cardinality estimation in DBMS: A comprehensive benchmark evaluation," Proc. VLDB Endow., vol. 15, no. 4, pp. 752–765, Dec. 2021, doi: https://doi.org/10.14778/3503585.3503586.

[30] P. Wu and Z. G. Ives, "Modeling shifting workloads for learned database systems," Proc. ACM Manag. Data, vol. 2, no. 1, Art. no. 38, pp. 1–27, Mar. 2024, doi: https://doi.org/10.1145/3639293.