PGConf.dev

Yandex Cloud

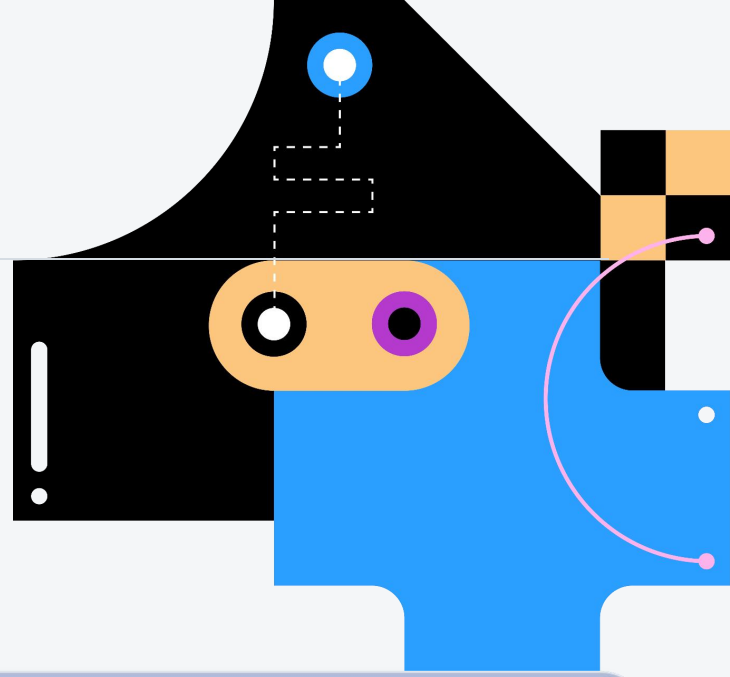
Is there a future for genetic and learned methods in query optimizers?

Alena Rybakina

About me

- Core developer since 2021
  (Postgres Professional 2021-2026 and
  now work in Yandex Cloud mostly with distributed
  systems like Cloudberry)
- Contributing to the PostgreSQL project since 2023
  - OR to ANY transformation
  - Values to ANY transformation
  - Others
- Participated in extension development: AQO

Outline



01 **Foundations**

Why this matters · Selinger DP · GEQO · SA · ACO · SAIO

02 **ML and Adaptive approaches & their critique**

Landscape · MCTS+ML · Practical MCTS · Neo · Bao · critique & benchmarks · Reoptimization

03 **Current state**

What works · What's open · 2026 plot twist · Verdict

PGConf.dev

Yandex Cloud

01

Foundations

Why this matters · Selinger · GEQO · SAIO · MCTS · HybridQO

The main problem: Too many candidates

Why we can't just try them all — and what to do about it

5 tables	10 tables	15 tables	20 tables	50 tables
120	3,628,800	1.3×10^{12}	2.4×10^{18}	3×10^{6} [4]
0.12 ms ✓	3.6 s	~15 days ✗	~77 000 years	?

Numbers are n! (left-deep plans). Time assumes 1 plan/µs cost evaluation — brute force is hopeless beyond ~12 tables.

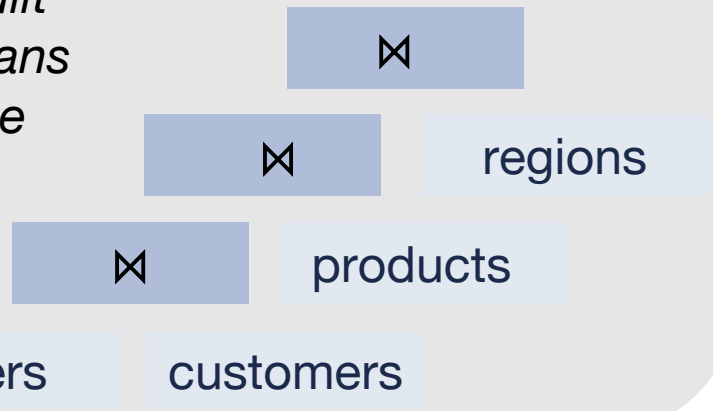
Left-deep (n!) vs Bushy ((2n–2)!/(n–1)!)

5 tables	10 tables	15 tables	20 tables	50 tables
120	**3 628 800**	**1.3×10^{12}**	**2.4×10^{18}**	**3×10^{64}**
bushy: 1 680	bushy: 1.8×10^{10}	bushy: 3.5×10^{18}	bushy: 4.3×10^{27}	bushy: 1.5×10^{91}

Left-deep plan (Postgres-like) - n!

((orders ⋈ customers) ⋈ products) ⋈ regions

Each JOIN is added to the already-built result. The tree leans left, growing one step at a time.



Bushy plan - (2n-2)!/(n-1)!

(orders ⋈ customers) ⋈ (products ⋈ regions)

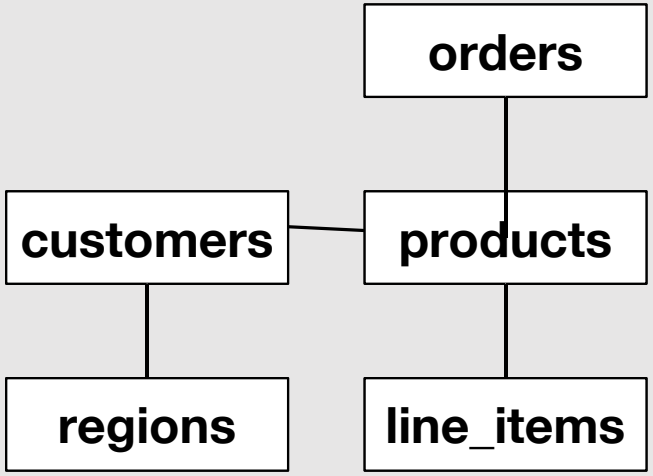
Both subtrees are built in parallel, then joined at the root — much larger search space.

It gets harder if there's a cycle

Key insight: the number of edges doesn't matter - what matters is whether there's a cycle. One "extra" edge that creates a cycle, and the problem turns from $O(n^2)$ into NP-hard.

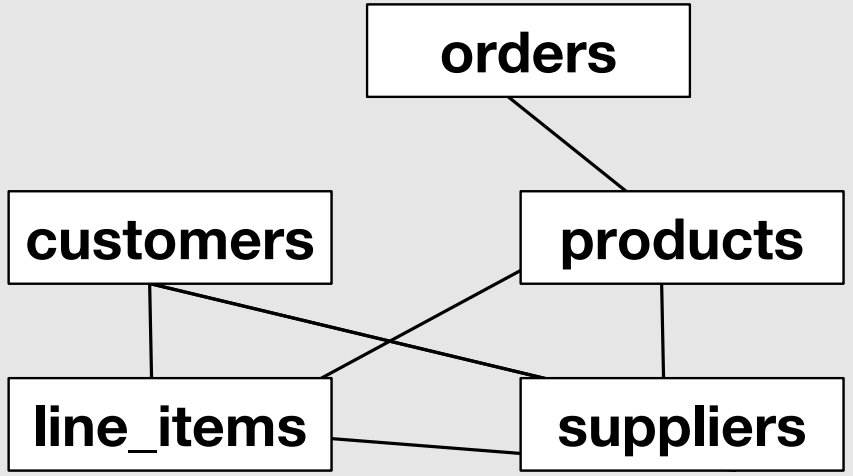
Tree

Each node connects to exactly one "parent". A clear hierarchy.



Graph with a cycle

At least one cycle appears: A → B → C → A. No single hierarchy.



Selinger 1979 — the template every optimizer copied

How it works — dynamic programming over subsets of relations

1 relation:	{A} {B} {C} {D}	*For each table, pick the cheapest access path*
2 relations:	{A,B} {A,C} {B,C} ...	*For each pair, find the cheapest join*
3 relations:	{A,B,C} {A,B,D} ...	*Build from level 2 — try adding each remaining table*
n relations:	{A,B,C,D, ...}	***The full query plan — guaranteed optimal under the cost model***

What it gave us: a cost model (COST = PAGE_FETCHES + W·RSI_CALLS), selectivity factors, and the search strategy above. Still the skeleton of every classical optimizer today.
The limit: 2^n subsets — works for ≤ ~12 joins, then explodes. Beyond that you need a different search strategy.

Source: Selinger, Astrahan, Chamberlin, Lorie, Price — "Access Path Selection in a Relational Database Management System", SIGMOD 1979

How iterative search works — one schema, three variants

The common iterative loop — all three algorithms share this skeleton:

1. Start
pick an initial plan (or a set of plans)

→

2. Modify
generate a candidate

→

3. Accept or reject
compare its cost and decide whether to move to it

→

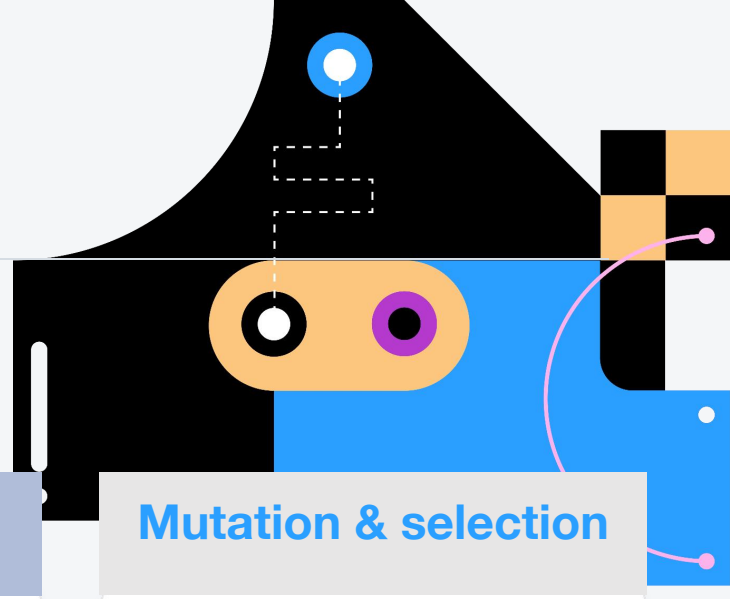
4. Repeat
go back to step 2 until budget runs out; return the best plan seen

Where the algorithms differ — only in steps 2 and 3:

GEQO *genetic algorithm*	**Modify:** keep a whole population of plans; new candidates by recombining two parents (crossover) + small mutations	**Accept:** drop the worst, keep the best (greedy selection). Never goes uphill → can get trapped in a local minimum
SA / SAIO *simulated annealing*	**Modify:** one plan at a time; new candidate by swapping two random subtrees in the current plan	**Accept:** cheaper → always take it; more expensive → take with probability e^(–Δ/T). Can escape local minima
MCTS *tree search*	**Modify:** build a tree of partial plans; expand by simulating random "rollouts" to a full plan	**Accept:** update node statistics; UCT formula balances exploitation (good so far) with exploration (under-tried)

Schema after Selinger 1979 (DP) and Ioannidis & Kang 1990 (cup landscape) · MCTS for joins: AlphaJoin 2020, "Practical MCTS-based Query Optimization" 2026

PostgreSQL GEQO



How GEQO works:

Population

A **chromosome = one join order**, encoded as the sequence of tables. E.g. [3,1,4,2,5] means "join table 3 with table 1, then join the result with 4, ..." — N such chromosomes form the initial population.

Evaluation

For each chromosome compute the plan cost via PostgreSQL's standard cost model. The lower the cost, the better the plan.

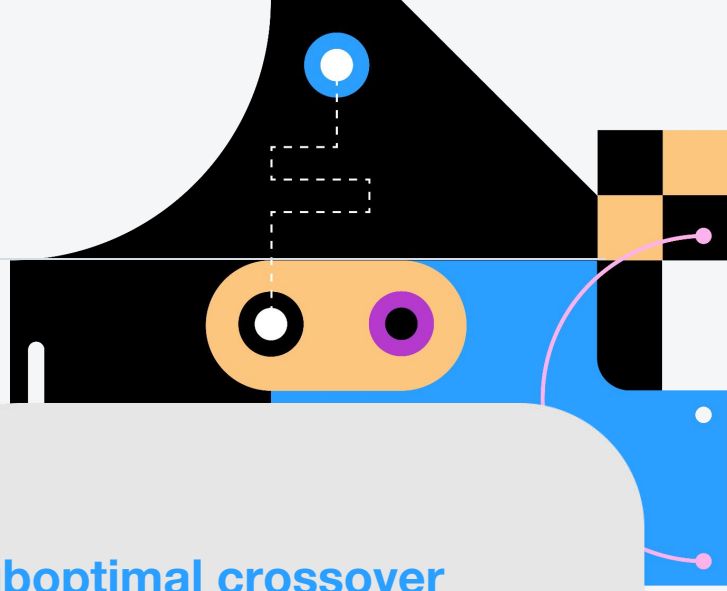
Recombination (crossover)

Take two good plans and "mix" them: take a slice from the first, fill missing tables from the second preserving their relative order. Idea: if A appears before B in both — that's likely meaningful, preserve it.

Mutation & selection

Occasionally swap two random tables. Drop the worst plan, add the best new one. Repeat for several generations — the population evolves toward good plans.

GEQO - the problems



Random population → non-determinism

The population is the set of candidate plans GEQO keeps around. It's built from random permutations, so two identical queries can get different plans. Without fixing geqo_seed the behaviour is unpredictable.

Cost considers only total_cost - ignores startup_cost

For LIMIT 10 it's important to get the first rows quickly, not minimize total cost. GEQO doesn't see this.

Suboptimal crossover

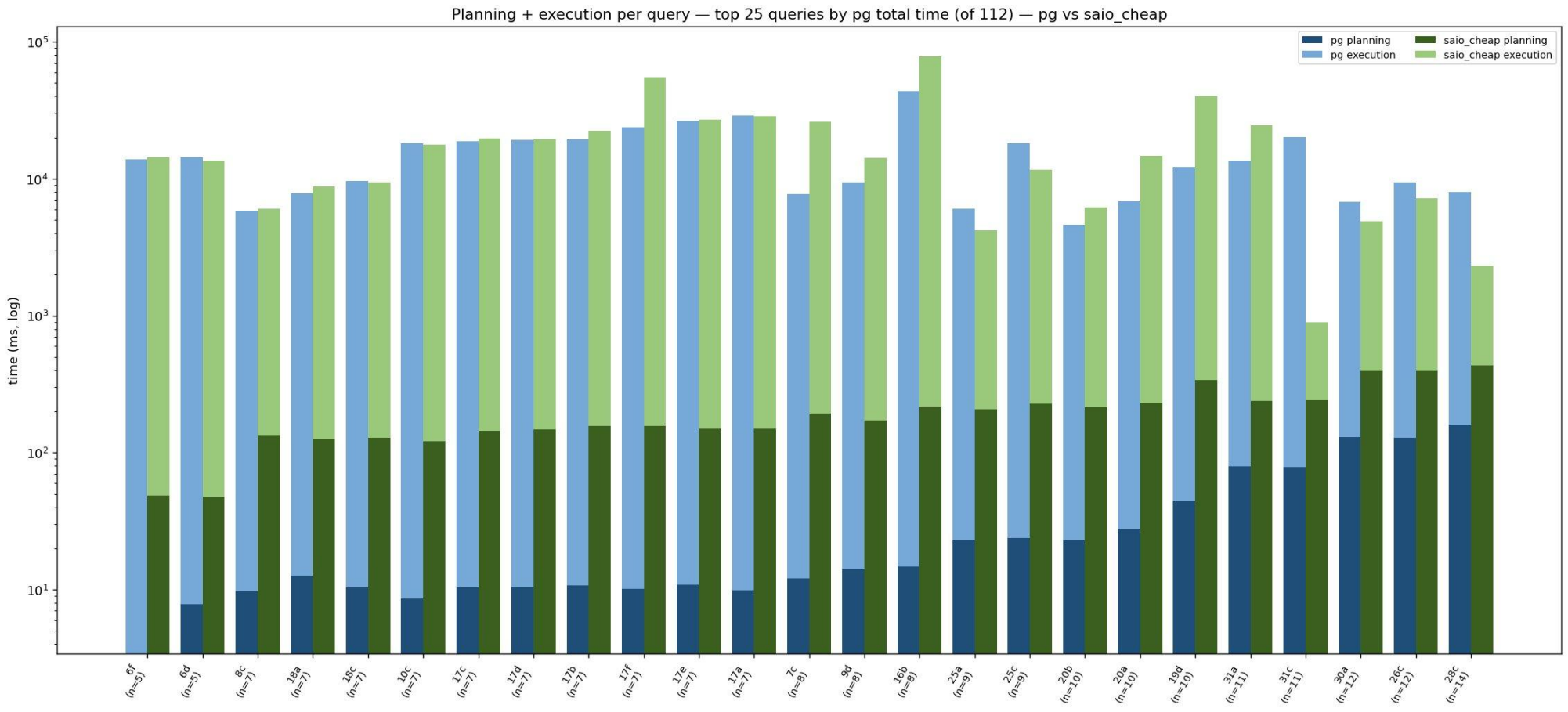
Take two parent plans, cut each in two, glue a piece of the first to a piece of the second to make a child. Idea — adjacent tables that look good together in both parents stay adjacent in the child. But join cost depends on context, so the same two tables side-by-side can cost differently in a new plan.

SAIO: Simulated Annealing (PGCon 2010)

PGCon 2010 (Ottawa) · Jan Urbański · «Replacing GEQO with Simulated Annealing (SAIO)»

Start	Make a move	Accept or reject	Equilibrium	Freeze
Build the initial left-deep plan. Set temperature T = I × number_of_tables.	Pick two random nodes in the plan tree and swap their subtrees. Recompute cost only for the affected part — from the swapped nodes up to their common ancestor.	New plan is cheaper - always take it. New plan is more expensive - take with probability e^((old_cost – new_cost) / T).	After N × number_of_tables steps at this temperature level, lower the temperature: T = T × K (e.g. K = 0.6).	If T < 1 and several consecutive steps are not accepted — stop. Return the best plan found.

SAIO: Results — per-query, top 25 by PG total time

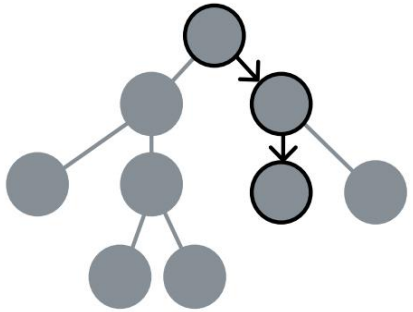


Per-query view confirms the binned picture: dark bars (planning) show SAIO is an order of magnitude above PG on almost every query — e.g. 6f/6d (n=5): ~8 ms PG → ~50 ms SAIO; 17a–f (n=7): ~10 ms → ~150 ms; 28c (n=14): 160 ms → 440 ms. Light bars (execution) sit close together — sometimes SAIO is slightly faster (31c, 28c), often slightly slower or much worse (17f, 16b, 19d → 2–3×). No query in the top-25 shows SAIO recovering its planning overhead through faster execution.
Bottom line: execution distributions overlap, planning distributions don't — SAIO is dominated by PG on this benchmark.

MCTS — four phases, repeated until budget runs out

Repeat 1 → 2 → 3 → 4 until the time/iteration budget is exhausted. The most-visited path from the root is the chosen plan.

1

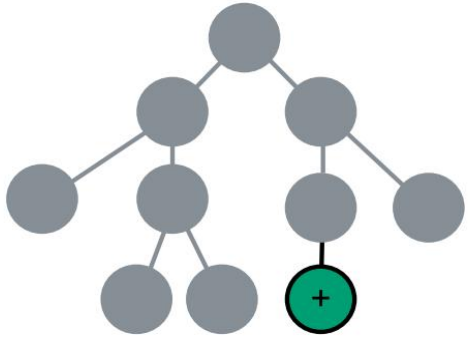


Selection

Walk down the tree from the root. At each node, pick a child using **UCT** (next slide).

Stop when you reach a node that still has unexplored children.

2

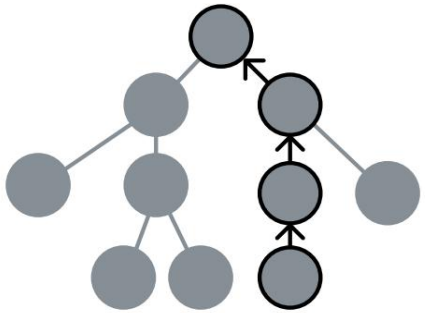


Expansion

Add one **new child** to the tree — one possible next join from this state.

The tree grows by one node per iteration.

4

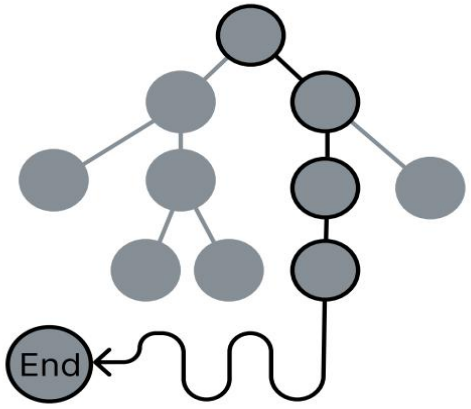


Backpropagation

Push the rollout cost back up the tree. Every visited node updates its visit count and average reward.

These stats are what Selection uses next round.

3



Simulation

From the new node do a **random rollout** — finish the plan by picking random joins. Compute its cost.

The cost is the reward.

Diagram after "Practical MCTS-based Query Optimization", arXiv 2603.16474, 2026 Fig. 1.

UCT — how Selection picks the next move

UCT = **U**pper **C**onfidence Bound applied to **T**rees · Kocsis & Szepesvári 2006 · used only in Selection (step 1)

The score

$$UCT(child) = \bar{X} + c \cdot \sqrt{(\ln N / n)}$$

$\bar{X}$ **EXPLOIT**	+	$c \cdot \sqrt{(\ln N / n)}$ **EXPLORE**
Average reward of this child so far — how good it looked in past rollouts.		Bonus for children visited **few times** (small n) — could still be the best.

Where the symbols come from:

- $\bar{X}$ — average reward of rollouts passing through this child (lower cost = higher reward)
- **N** — total visits at the *parent*
- **n** — visits at *this* child
- **c** — tunable constant (e.g. $\sqrt{2}$). Bigger **c** = more exploration, smaller = more greedy.

Why we need it

In Selection, we walk down the tree. At each node with several children, we have a choice. The dilemma:

- **Pick the child with the best average reward so far** — *but maybe another child is better and you just haven't tried it enough.*
- **Always try the least-explored child** — *but you waste budget on hopeless branches.*

UCT solves it: **pick the child with the highest UCT score**. The score rewards what looks good and gives a small bonus to under-tried options.

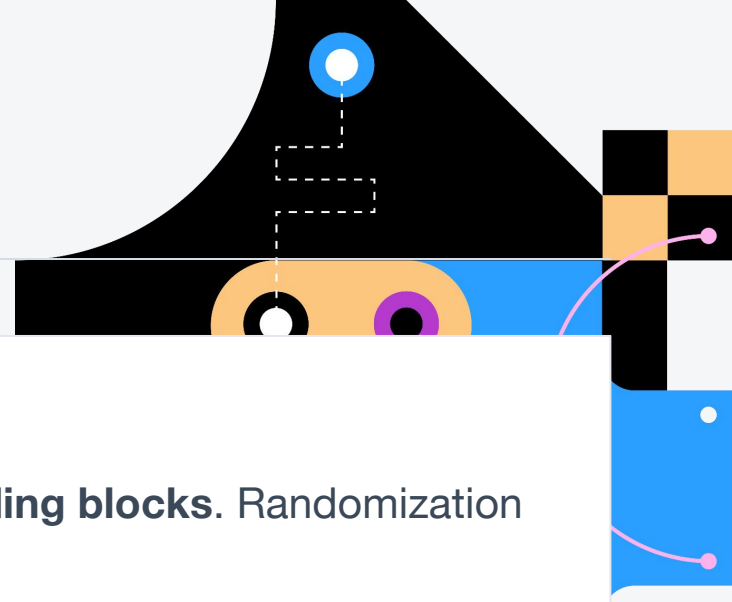
Where it fits in MCTS

1 **Selection** · 2 Expansion · 3 Simulation · 4 Backprop

UCT runs only in step 1. The other three steps don't need it.

Source: Kocsis & Szepesvári, "Bandit Based Monte-Carlo Planning", ECML 2006 · roots: Auer et al. 2002 (UCB1 for multi-armed bandits)

Randomized methods didn't die



The main idea

The ML methods we'll see next don't replace GEQO and SA — they **use them as building blocks**. Randomization didn't disappear — it moved one level up.

Example 1 — GEQO inside MCTS

In "Practical MCTS-based Query Optimization" (2026) the authors use GEQO to produce a starting plan, then let MCTS improve it. **GEQO is the seed**, not the answer.

Example 2 — random rollouts inside AlphaJoin

AlphaJoin (Zhang 2020) keeps the **random-rollout heart of MCTS** and replaces the cost model with a neural net. The randomization stays — ML is just bolted on top of it. This is the doorway into the ML/Adaptive section.

Sources: "Practical MCTS-based Query Optimization: A Reproducibility Study and New MCTS Algorithm for Complex Queries" (arXiv 2603.16474) · AlphaJoin (Zhang VLDB 2020)

HybridQO / HyperQO — step by step on one query

Example: `SELECT * FROM a, b, c, d WHERE a.id=b.id AND b.id=c.id AND c.id=d.id`
(4 tables, 3 join predicates)

1. MCTS explores prefixes

Vanilla MCTS builds a tree of partial join orders: `(a,b)`, `(c,b)`, `(b,d)`... For each prefix it asks the neural net "if we start this way, how fast will the query run?"

2. Pick the best prefix

MCTS converges on the prefix with the lowest predicted time. For our example: `(c, b)`. ML hasn't built a full plan — only said "start with c⋈b, then ask PostgreSQL".

3. Hand off to PostgreSQL

Run query with `/*+ Leading(c b) */`. PG's optimizer (DP or GEQO) completes the plan honoring the prefix — gets `c⋈b⋈d⋈a`.

4. Get a baseline too

In parallel, PostgreSQL builds its own plan without any hint — its native cost-based choice (e.g. `a⋈b⋈c⋈d`). Now we have two candidate plans.

5. Score both plans

A neural net (with Bayesian dropout) predicts for each plan: **(a)** expected execution time, **(b)** how confident it is in that estimate (uncertainty).

6. Confidence gate

If the ML plan has **low uncertainty** → use it (execute `c⋈b⋈d⋈a`). If **high uncertainty** (OOD query?) → fall back to PostgreSQL's plan. *This is the "safety net".*

Claimed gains and where they break

- **Reported: –25% mean latency, –65% tail latency** on JOB vs PostgreSQL. Code at github.com/yxfish13/HyperQO.
- **Does not replicate** under honest train/test splits — the uncertainty model is itself learned, and overfits to training queries.
- Failure mode: on OOD queries the uncertainty model is **over-confidently wrong** — the gate stops firing exactly when it's needed.

Source: Yu, Chai, Li, Liu — "Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection", PVLDB 2022 (arXiv 2204.00224) ·
Critique: "Practical MCTS-based Query Optimization", 2026

Detour: how does the net know it doesn't know?

HyperQO's confidence gate relies on **Bayesian dropout** (Gal & Ghahramani 2016) - a trick to get uncertainty out of a neural net.

The idea

During training, randomly switch off 30% of neurons each batch (standard dropout). At inference, **keep dropout on** and run the same query through the net 50 times. Each run gives a slightly different answer. The **mean** is the prediction, the **spread** is the uncertainty.

Query the net has seen often

50 runs return, in ms:

`500, 498, 502, 499, 501, …`

All clustered around **500 ms**. Spread ≈ ±2 ms.

Low uncertainty - trust the prediction.

Query unlike anything in training

50 runs return, in ms:

`200, 1500, 50, 800, 2000, …`

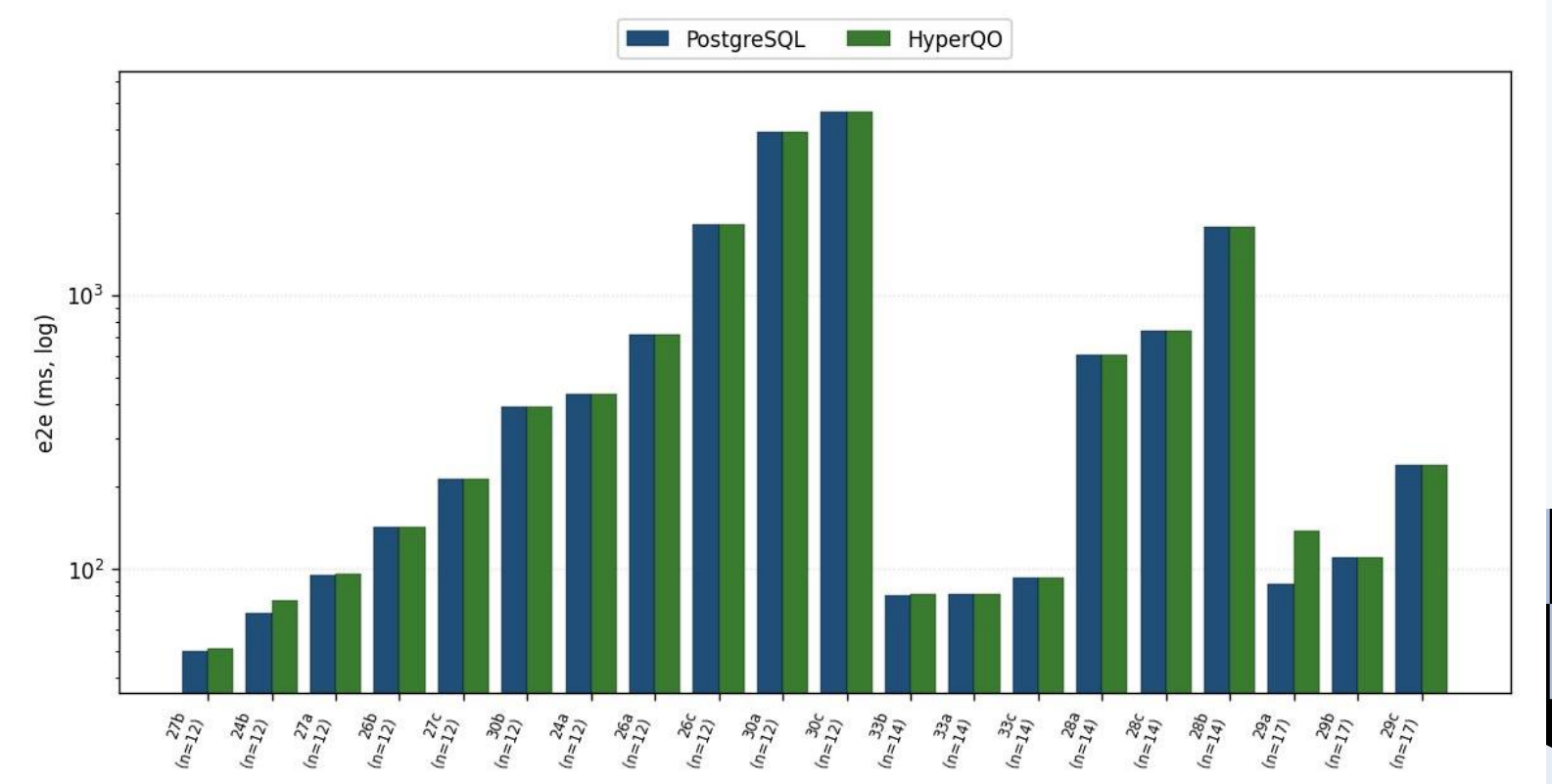
All over the map. **Spread is huge.**

High uncertainty - fall back to PostgreSQL.

Intuition: if knocking out random neurons doesn't change the answer, the net wasn't really using them — it knows the answer. If it does change a lot, the net was guessing.
Caveat: the uncertainty itself is learned, and on truly novel queries it can be over-confidently wrong. That's the HyperQO critique we'll see later.

Source: Gal & Ghahramani, "Dropout as a Bayesian Approximation", ICML 2016 · used in HyperQO (Yu et al. 2022) for the confidence gate

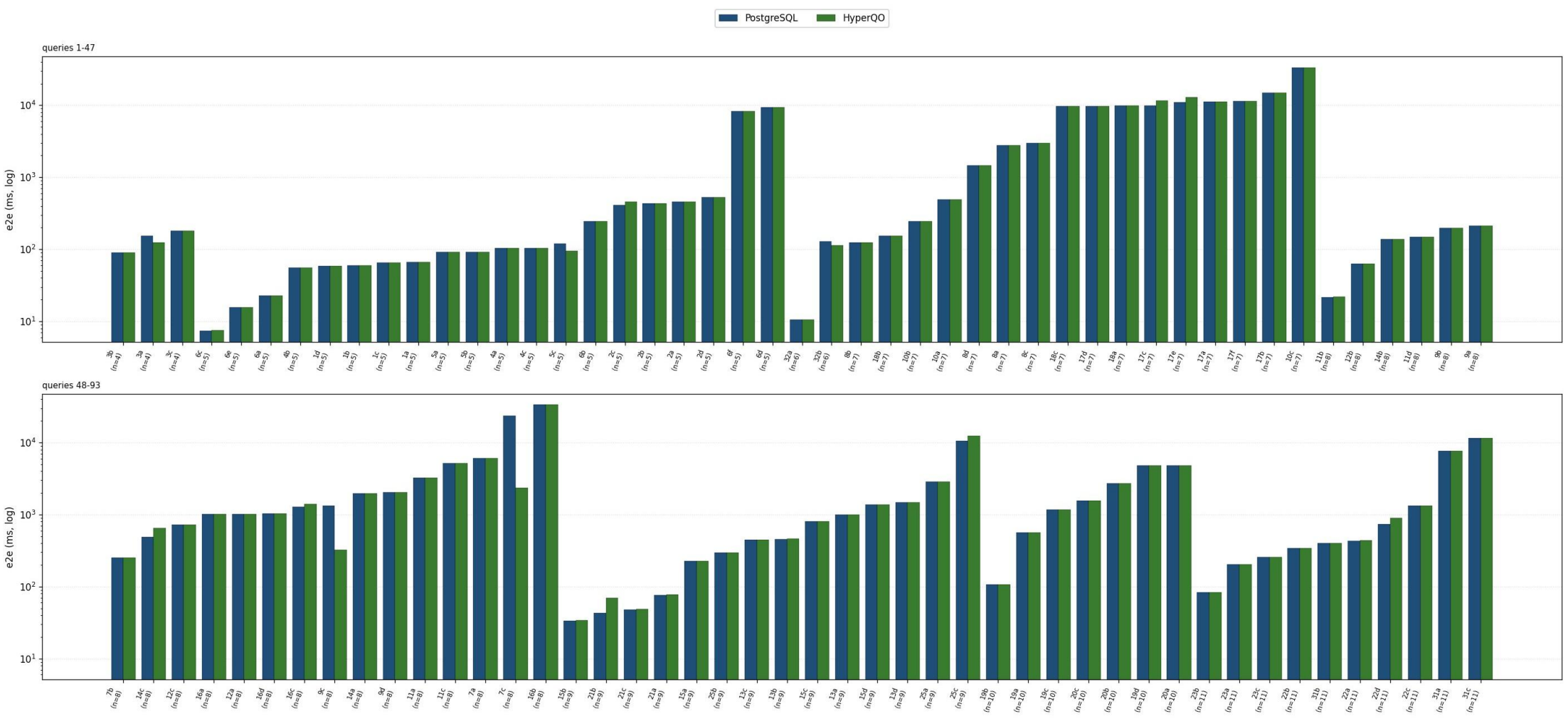
HyperQO: results — large queries (n ≥ 12)



20 queries with n ≥ 12, geomean HyperQO/PG = 1.03× — HyperQO faster on 0, slower on 2. Plans are almost indistinguishable from PG — bars overlap on every query. The confidence gate fires aggressively here: when uncertainty is high, HyperQO falls back to PG's plan, so we mostly observe PG's timings. Only 29a/29b (n=17) show a visible regression (~90 ms → ~130 ms; ~110 ms → similar) where HyperQO chose a different prefix and lost.

Bottom line: the gate works — it stops HyperQO from doing damage on hard queries, but also stops it from helping. Net effect ≈ identical to PG, with extra inference cost.

HyperQO: results — small & medium queries (n = 4–11)



93 queries (n=4–11), geomean HyperQO/PG = 0.97×: HyperQO wins on 5, loses on 8, ties on the rest. Bars are visually identical to PG on most queries — the confidence gate hands control back to PostgreSQL. Where HyperQO wins it's usually a small improvement (9c, 5c); the few losses are also small. Bounded-regret behavior in practice.

Why this matters now

Two forces meet — growing pressure on the optimizer, and a generation of researchers with new tools

The pressure on the optimizer is growing

Data volume tables now hold billions of rows. A small mistake in cardinality estimation can turn a millisecond query into an hours-long one.

Query complexity analytics queries now routinely have 20+ joins. The number of possible join orders explodes — exhaustive search becomes impossible long before n = 30.

Latency expectations interactive analytics expects sub-second response. Planning time (10–100 ms) is no longer negligible — it becomes part of the user-visible latency budget.

Why researchers turn to AI

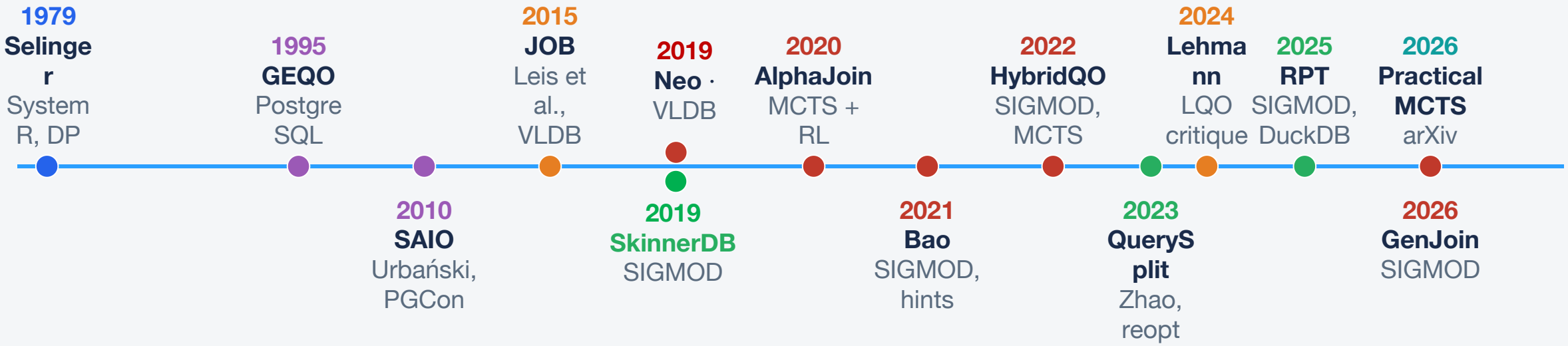
Curiosity New ideas, weekend experiments, the small magic of watching a neural net find a plan no DBA would write. The field rediscovered the joy of trying weird things.

Data keeps changing workloads aren't static — distributions, schemas and access patterns shift daily. A fixed cost model captures yesterday's data, not today's. Adaptivity is no longer a feature — it's the requirement.

The AlphaGo moment (2016) MCTS + neural networks beat human Go champions — a search space with ~10^{170} states. Query plan space is "only" n! — suddenly a tractable target with familiar techniques.

Sources: Deshpande, Ives, Raman 2007 (VLDB tutorial) · Selinger 1979 (System R DP) · Marcus et al. 2019 (Neo motivation) · "Practical MCTS-based Query Optimization", 2026

47 years of join ordering — timeline of works in this talk



- Classical (DP)
- Randomized / Iterative
- Benchmarks / audits
- Learned optimizers
- Reoptimization / Adaptive
- Hybrid (search + PG cost)

PGConf .dev

Yandex Cloud

02

ML and Adaptive approaches

Landscape · MCTS+ML · Practical MCTS · Neo · Bao · critique & benchmarks · Reoptimization

The landscape of learned query optimizers

Twelve+ learned query optimizers — but they fall into **four families** based on **what the model produces**.

① Step-by-step plan generation

RL agent picks one join at a time, builds the plan join-by-join

- **Neo** (2019) — Tree-CNN value net, learns end-to-end from execution latencies
- **AlphaJoin** (2020) — MCTS + neural cost model

Other members of this family (ReJOIN, RTOS, Balsa) are not covered today.

② Hint-set methods

ML picks high-level hints; classical optimizer builds the plan

- **Bao** (2021) — Thompson-sampling bandit picks one of 48 hint sets per query
- **HybridQO** (2022) — MCTS picks a leading-prefix hint, PG completes

Other members of this family (Lero, COOOL) are not covered today.
Trade-off: ML scope is smaller, generalization tends to be better; but limited to what hints can express.

③ Runtime / regret-bounded

Don't predict — try, measure, switch. Learns on this query.

- **SkinnerDB** (Trummer 2019) — UCT per time slice, regret bound

Other members (Liu/Ives/Loo 2014, QuerySplit) — covered in the dedicated section, not detailed today.
We'll come back to this family in a dedicated section.

④ Generative plan-to-plan (new)

Generative model outputs a plan-shaped object directly

- **GenJoin** (Sulimov, Lehmann, Stockinger — SIGMOD 2026)
- A neural net is trained to suggest, for each pair of tables, **which join algorithm to use** (hash, merge, nested loop). Nothing else — PostgreSQL decides the order.
- **First learned method that reliably beats PostgreSQL** on two real-world benchmarks — earlier methods only claimed it.

Taxonomy after: Yan, Uotila, Lu — "Join Order Selection with Deep RL: Fundamentals, Techniques, and Challenges", PVLDB 2023 · GenJoin: Sulimov, Lehmann, Stockinger — SIGMOD 2026

AlphaJoin — MCTS with a learned cost model

In one phrase: "Run MCTS, but instead of executing the plan to score it, ask a neural net how fast the plan looks."

1. Offline - teach a net to rate plans

Collect many (plan, real execution time) pairs.

Sort times into **4 buckets**: fastest - slowest.

Train the **Order Value Network (OVN)** to predict which bucket a plan falls into.

Goal: a cheap function that says "this plan is probably bucket 0 (fast)" or "bucket 3 (slow)".

2. At query time - MCTS explores

Standard 4-phase MCTS — same as on slide 12.

The tree explores partial join orders. Each node is a partial plan.

In **Simulation**, finish the plan by picking random joins for the remaining tables.

Now we have a complete plan — but we never ran it. We need a score.

3. OVN scores - no SQL is run

Feed the complete plan to **OVN**. It returns "bucket 0" (fast) or "bucket 3" (slow).

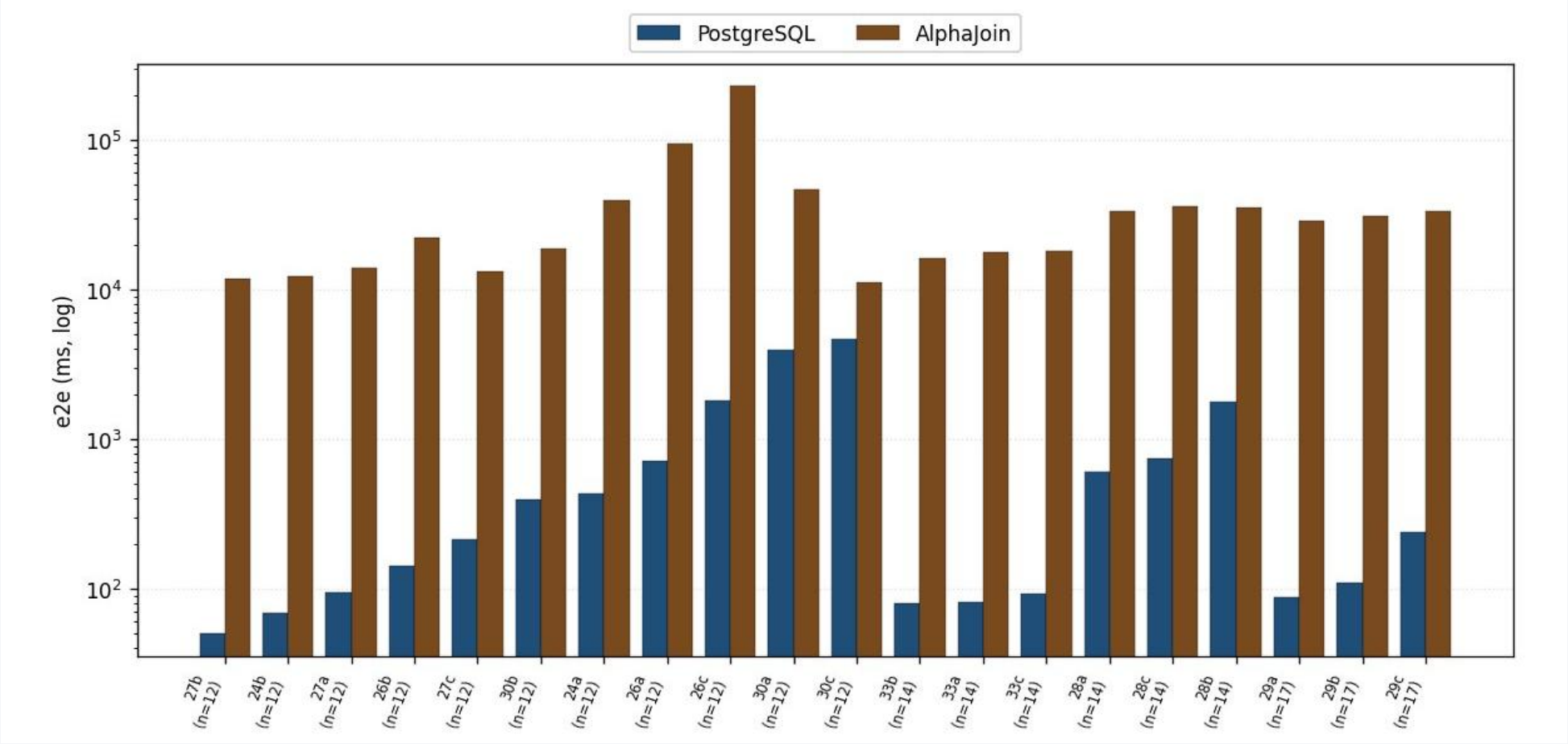
Backprop the score up the tree (good buckets push exploration toward similar plans next time).

After the iteration budget, return the most-visited path. That's the join order to use.

This is the AlphaJoin move: replace the cost model with a learned bucket classifier.

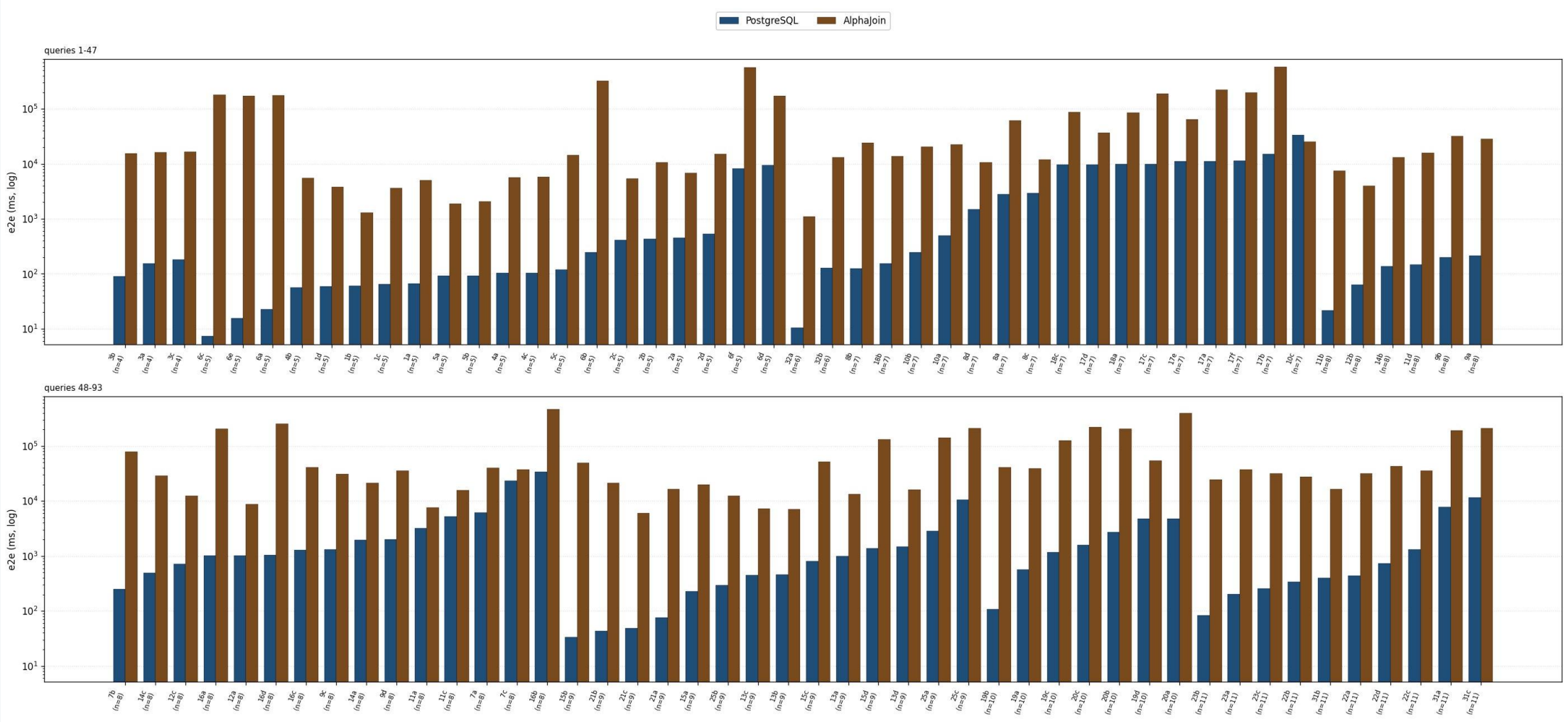
Source: Ji Zhang, "AlphaJoin: Join Order Selection à la AlphaGo", PhD Workshop @ VLDB 2020 · code: github.com/HustAIsGroup/AlphaJoin

AlphaJoin: results — large queries (n ≥ 12)



20 queries with n ≥ 12, geomean AlphaJoin/PG = 87.80× — AlphaJoin slower on all 20, faster on 0. The learned cost model fails exactly where it was supposed to help. Examples: 27b/24b (n=12) jump from ~50–70 ms (PG) to ~12 000 ms (AlphaJoin); 26c (n=12) goes 1.8 s → ~230 s (~130×); 33b/33a/33c, 29a/29b (n=14–17) all sit at ~80–230 ms in PG but at 16–35 s in AlphaJoin (100–400×). **Bottom line:** on large joins AlphaJoin's OVN classifies plans confidently and wrongly — classic OOD breakdown of a learned cost model. PG is uniformly better, often by 2 orders of magnitude.

AlphaJoin: results — small & medium queries (n = 4–11)



93 queries (n=4–11), geomean AlphaJoin/PG = 50.99×: AlphaJoin wins on just 1, loses on 92. The picture is uniform across n — even on the easiest queries (n=4–5) where PG runs in 10–200 ms, AlphaJoin takes 1–200 s. The learned cost model adds nothing on simple plans and breaks badly on hard ones.

Practical MCTS — MCTS without the learned cost model

Stage 1 — find a good opening

For every edge in the join graph (every pair of tables connected by a predicate), run a **short MCTS** (T_pair iterations) with that pair as the starting prefix. Track the best cost found.

Output: the winning prefix **p*** — the pair that led to the cheapest plan in this quick probe.

Stage 2 — refine the full plan

Starting from **p***, run a **longer MCTS** (T_main iterations) to extend the prefix to a complete left-deep plan.

Output: the best left-deep order found. Sent to PostgreSQL via `pg_hint_plan` leading hints.

Cost model

Reward = `-EXPLAIN cost` from PostgreSQL itself.

No neural net, no training. The cost model that ships with PG, used as a black box: rewrite the query with leading hints, run EXPLAIN, read the total cost.

This is the main departure from AlphaJoin / HyperQO.

Extreme UCT

$$UCT_{ext} = \max_t X_{j,t} + 2c \cdot (\ln N / n_j)^{\gamma}$$

Replace the standard UCT mean with the **max reward** seen so far. We want the best plan, not the average plan — there is no adversary, no need to hedge.

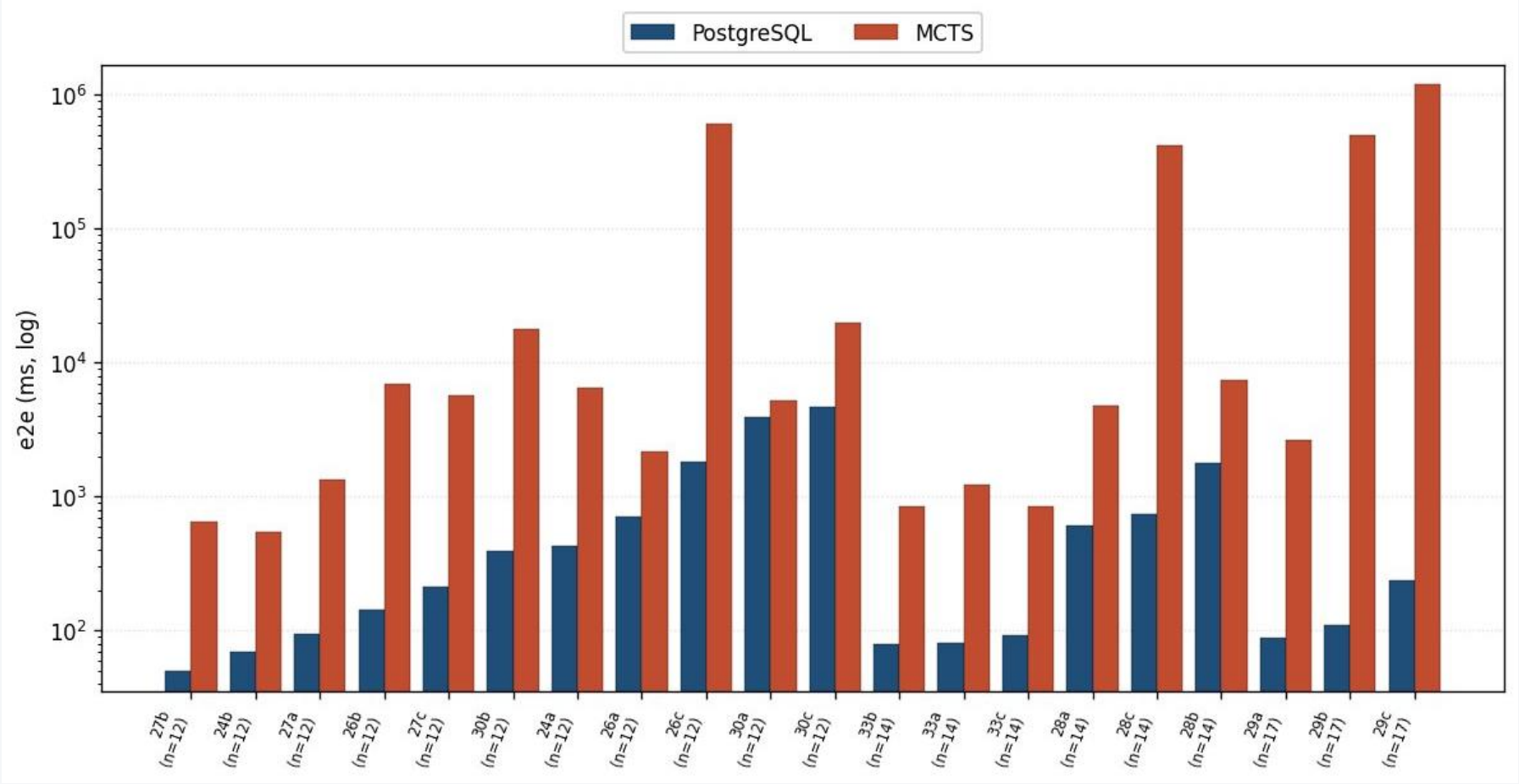
GP operators inside MCTS

During simulation, in addition to random rollouts, apply **mutation** and **crossover** on partial plans — borrowed from genetic programming (MCTS-4-SR framework).

A generalization of PostgreSQL's GEQO, embedded inside MCTS.

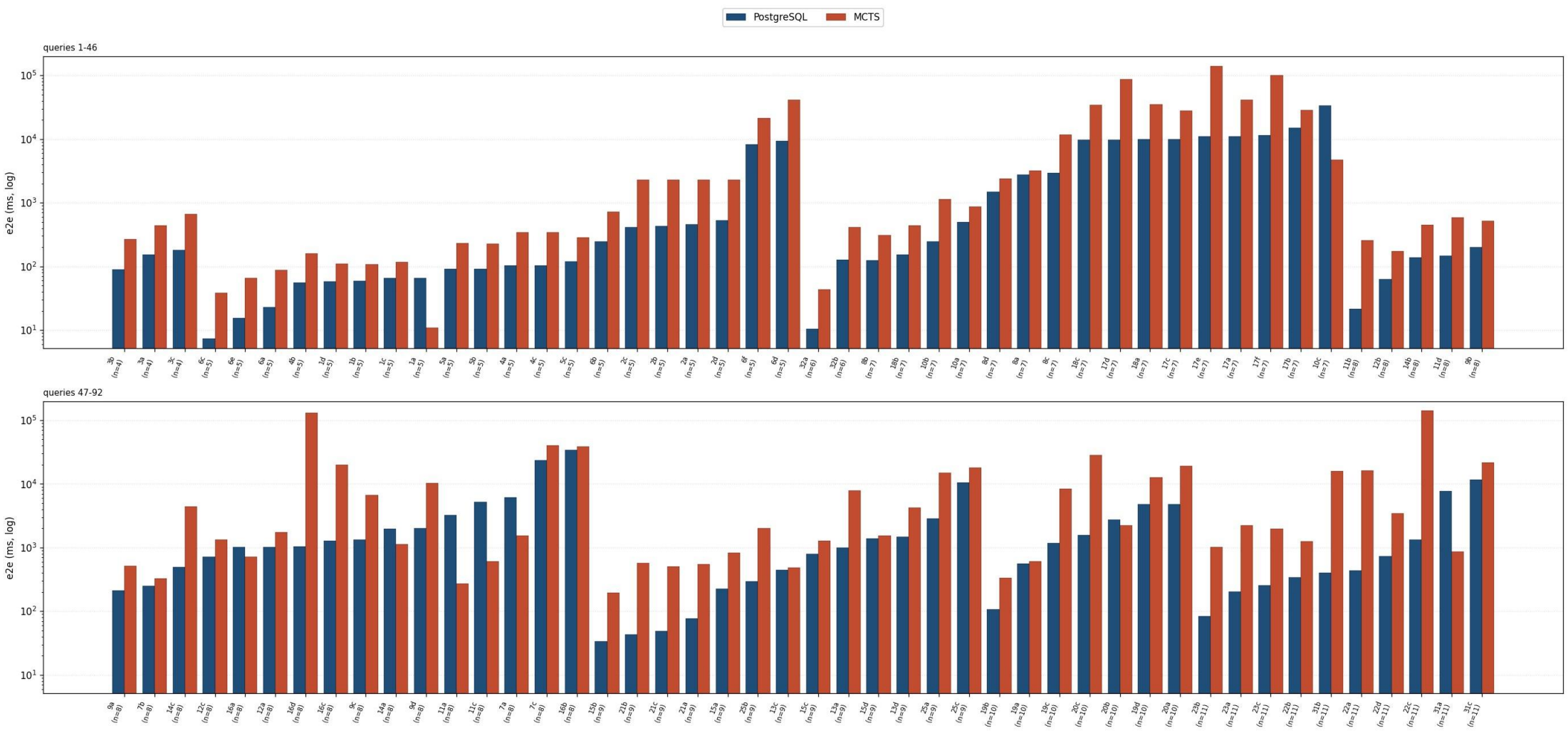
Source: "Practical MCTS-based Query Optimization", arXiv 2603.16474, 2026 — Algorithm 1

Practical MCTS: results — large queries (n ≥ 12)



20 queries with n ≥ 12, geomean MCTS/PG = 28.71× — MCTS slower on all 20, faster on 0. Without learned cost, MCTS still loses badly at scale — the search budget isn't enough to find good plans for n=12–17. Worst cases: 26c (1.8 s → ~580 s, ~320×), 29c (240 ms → ~1.2 M ms, ~5000×), 29b (110 ms → ~500 s). Even "easy" n=14 queries (33a/33b/33c, ~80–90 ms) blow up to 0.8–1.2 s.
Bottom line: MCTS without ML helps in the n=8–11 range (next slide) but doesn't scale to n ≥ 12 with the budgets tested. The search space grows faster than the iteration count.

Practical MCTS: results — small & medium queries (n = 4–11)



92 queries (n=4–11), geomean MCTS/PG = 3.15×: MCTS wins on 9, loses on 83. The picture is much milder than at n ≥ 12 — most regressions are 2–10× rather than 100×+. Wins concentrate where the search budget covers the space (6c, 32a, 22a, 13b). On n=4–7 the planner overhead dominates: PG runs in 10–200 ms, MCTS still does its rollouts.

Neo (VLDB 2019) — the first end-to-end learned optimizer

Marcus et al. — predict the latency of a partial plan with a neural net, build the plan greedily picking the best next step.

How a plan becomes a vector — the featurizer

```
SELECT * FROM A, B, C, D WHERE A.x=C.x AND C.y=B.y AND A.z < 5;
```

(a) Query encoding

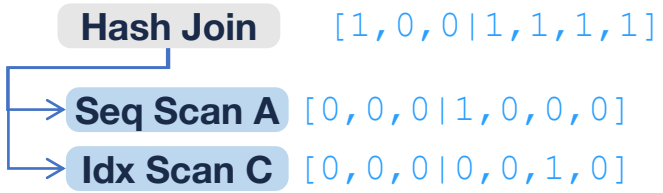
Join graph (adjacency):

	A	B	C	D
A	0	0	1	0
B	0	0	1	0
C	1	1	0	0
D	0	0	0	0

Building the plan greedily:

1. Start: every table is its own subtree.
2. List all legal "next moves": join two subtrees, pick a scan type.
3. Tree-CNN scores each candidate. Keep the lowest.
4. Repeat until one tree remains.

(b) Plan encoding — tree of vectors



Predicate vector:

A.x	A.z	B.y	C.x	C.y	…
0	1	1	0	1	

1 = column appears in a WHERE filter

Reported results

JOB / TPC-H / Stack, vs PostgreSQL baseline

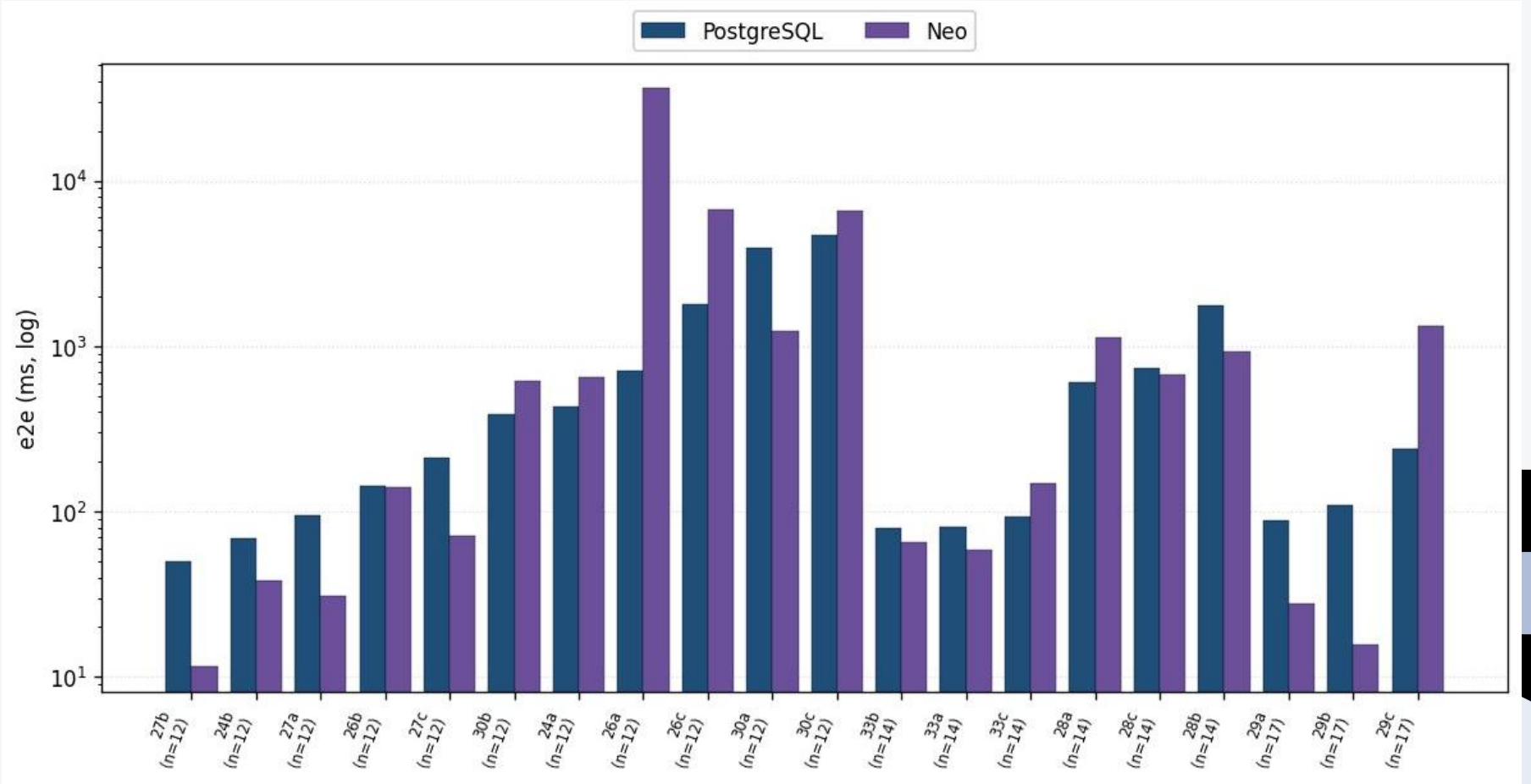
- **Beats PostgreSQL** on all three benchmarks after ~100 training queries.
- **Matches commercial optimizers** (Oracle, MS SQL) on workloads they tested.
- Bootstrapped from PostgreSQL plans — inherits its blind spots, but improves over time.

Caveats:

- Training takes **hours**; schema changes mean full retraining.
- Tail latency can regress badly on OOD queries.

Source: Marcus, Negi, Mao, Zhang, Alizadeh, Kraska, Papaemmanouil, Tatbul — "Neo: A Learned Query Optimizer", PVLDB 2019 (arXiv 1904.03711) · Figs. 3 and 4

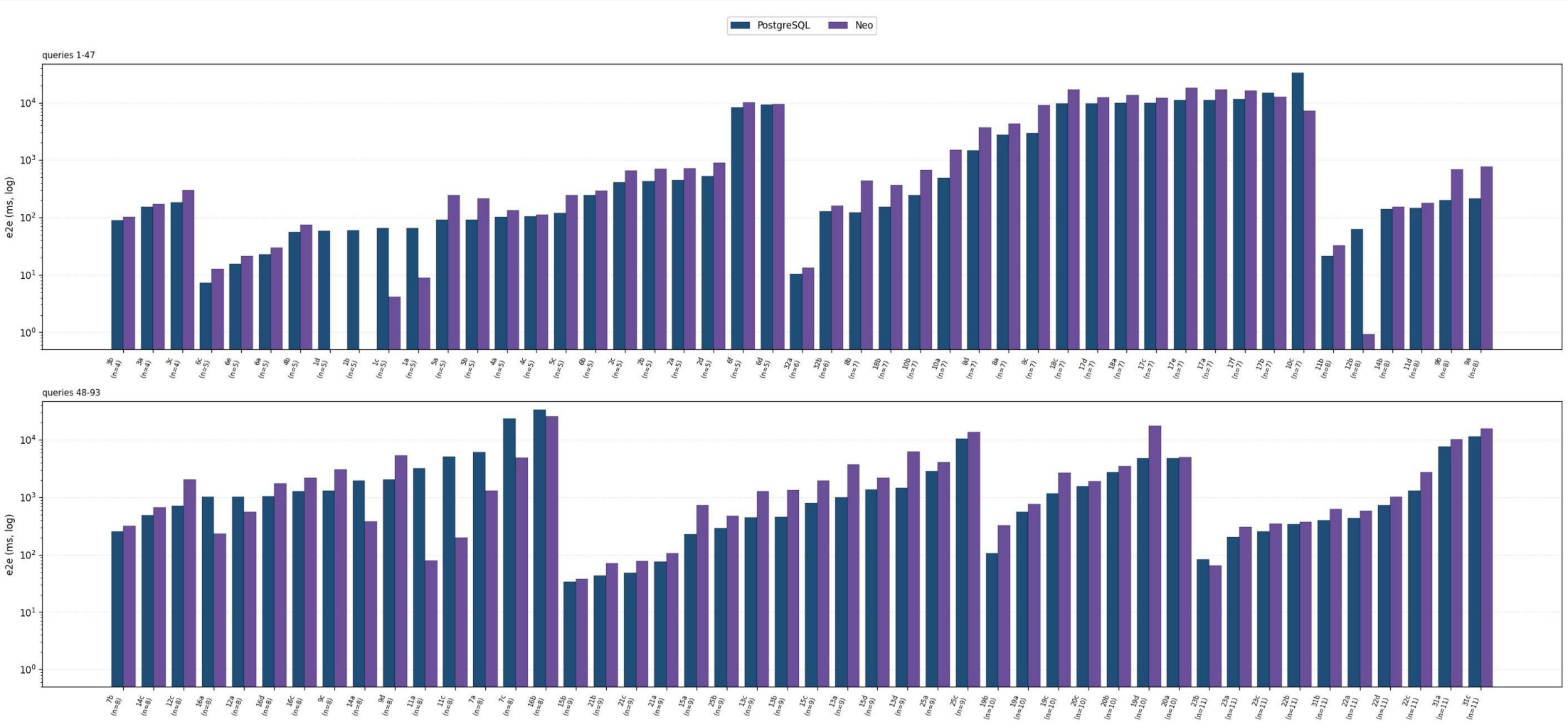
Neo: results — large queries (n ≥ 12)



20 queries with n ≥ 12, geomean Neo/PG = 0.97× — Neo faster on 11, slower on 8. Neo actually wins on the majority here — best learned method so far on large joins. Big wins: 27b (50 → 12 ms, ~4×), 30a (3.9 → 1.2 s, 3.3×), 29b (110 → 16 ms, ~7×), 29a (90 → 28 ms). But the tail is brutal: 26a (700 ms → 37 s, ~50×), 26c (1.8 → 6.8 s), 27a regresses ~3×.

Bottom line: Neo's Tree-CNN value net is competitive on average but has heavy tails. The good queries justify the training cost; the bad ones (50× regression on 26a) are exactly the OOD-failure pattern the LQO critique warns about.

Neo: results — small & medium queries (n = 4–11)



93 queries (n=4–11), geomean Neo/PG = 1.07×: Neo wins on 16, loses on 75. Wins are scattered but real (12b: 62 → 0.9 ms; 11a: 3.2 s → 80 ms, ~40×; 11c: 5.1 s → 200 ms, ~25×). Losses are mostly small (~1.5–3×). On n=4–5 simple queries Neo often adds overhead it can't recover. The big wins on n=8 medium queries are what pulls the geomean close to 1.

Bao (SIGMOD 2021) — steer PostgreSQL, don't replace it

Marcus et al. — same authors as Neo, opposite philosophy. Pick a hint set; PostgreSQL plans. Worst case = PostgreSQL.

What is a hint set?

A combination of PostgreSQL boolean flags. There are ~48 valid combinations:

Hint set #1 (default)
```
hash_join = ON
merge_join = ON
loop_join  = ON
```

Hint set #2
```
hash_join = ON
merge_join = ON
loop_join  = OFF
```

Hint set #3
```
hash_join = ON
merge_join = OFF
loop_join  = OFF
```

…

For each incoming query Bao:

1. Asks PostgreSQL to plan the query under each hint set → gets ~48 candidate plan trees
2. Tree-CNN scores each plan; Thompson sampling picks one — see next slide
3. Executes that plan, records actual latency, retrains

Reported results

IMDb, Stack, Corp workloads vs PostgreSQL & commercial

- Trains in **~1 hour** (an order of magnitude faster than Neo).
- Improves **tail latency** by orders of magnitude on JOB.
- **Bounded regret**: worst case = stock PostgreSQL — never worse.
- Adapts to workload, data, and schema changes without full retraining.

Caveat:

- Inference time + N planner calls per query — overhead can exceed gains on short queries.

Neo vs Bao — same authors, opposite design

ASPECT	NEO	BAO
Role	Replace the optimizer entirely	Pick a hint for PostgreSQL
Action space	$O(k \cdot 2^n)$ — every possible plan tree	O(k) — ~48 hint sets
Training time	Hours · full retrain on schema change	**~1 hour · adapts continuously**

Source: Marcus, Negi, Mao, Tatbul, Alizadeh, Kraska — "Bao: Learning to Steer Query Optimizers", SIGMOD 2021 (arXiv 2004.03814)

Detour: how does Bao decide which hint to try?

Bao picks one hint per query using **Thompson sampling** (Thompson 1933) — a 90-year-old bandit algorithm. Same family of trick as Bayesian dropout.

The idea

For each hint set, Bao has a model of "how fast this hint typically makes plans". Instead of picking the hint with the **best mean**, Bao **draws one random sample** from each hint's model and picks the hint with the best sampled value. When a hint is uncertain (wide distribution), random sampling occasionally picks it — that's exploration.

Early — Bao knows little

Each hint's model is a wide bell curve — Bao has no opinion yet. Sampling once from each:

```
#1 → 320 ms
#2 → 90 ms   ← pick #2
#3 → 500 ms
```

Effectively random → broad exploration

After experience — narrow bell curves

Hint #2 has consistently been fast. Its model is now a narrow curve around 100 ms. Sampling:

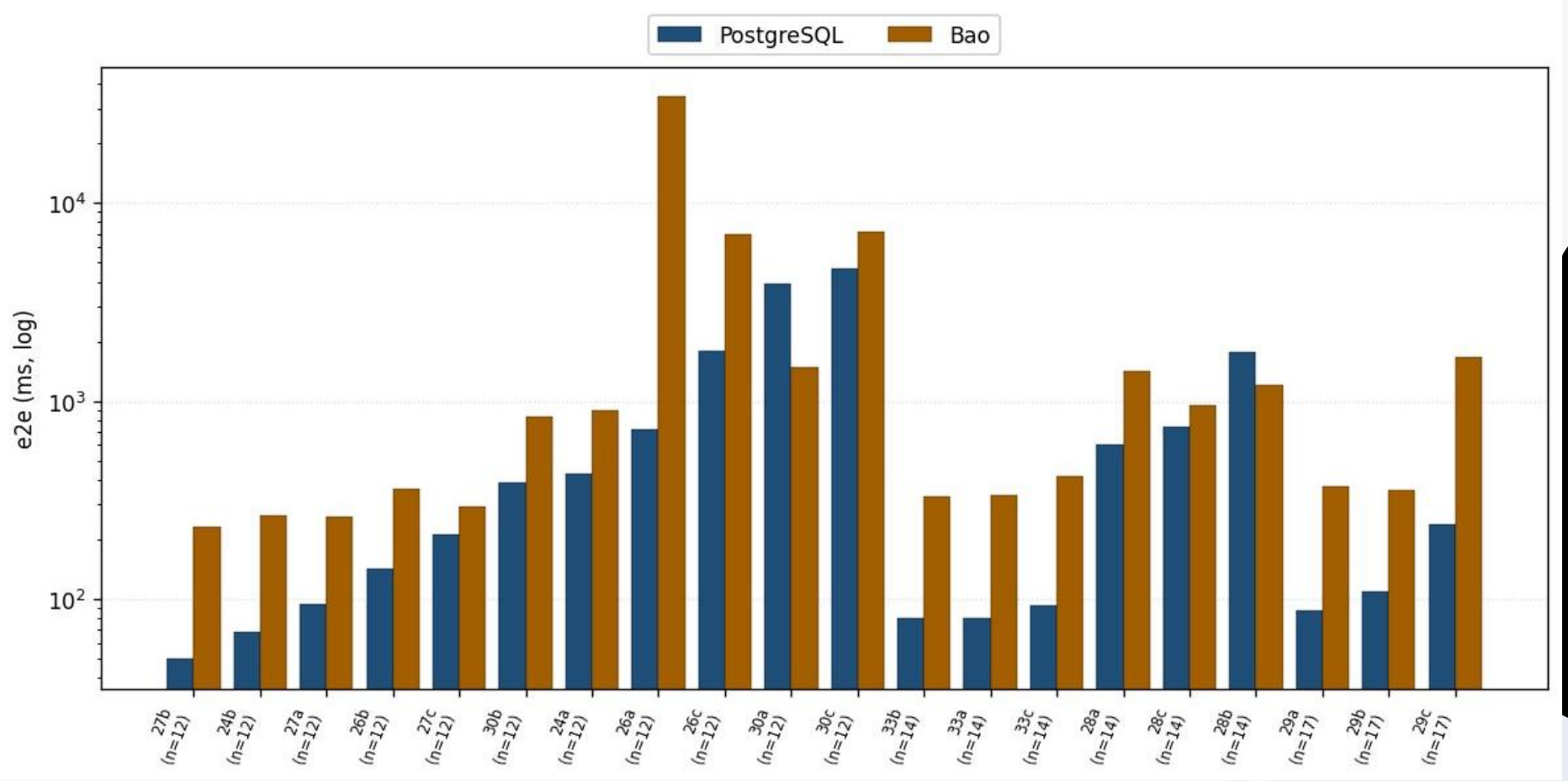
```
#1 → 280 ms
#2 → 102 ms   ← pick #2
#3 → 470 ms
```

Mostly exploitation, occasional explore

Intuition: uncertainty is what drives exploration — no separate exploration parameter to tune.

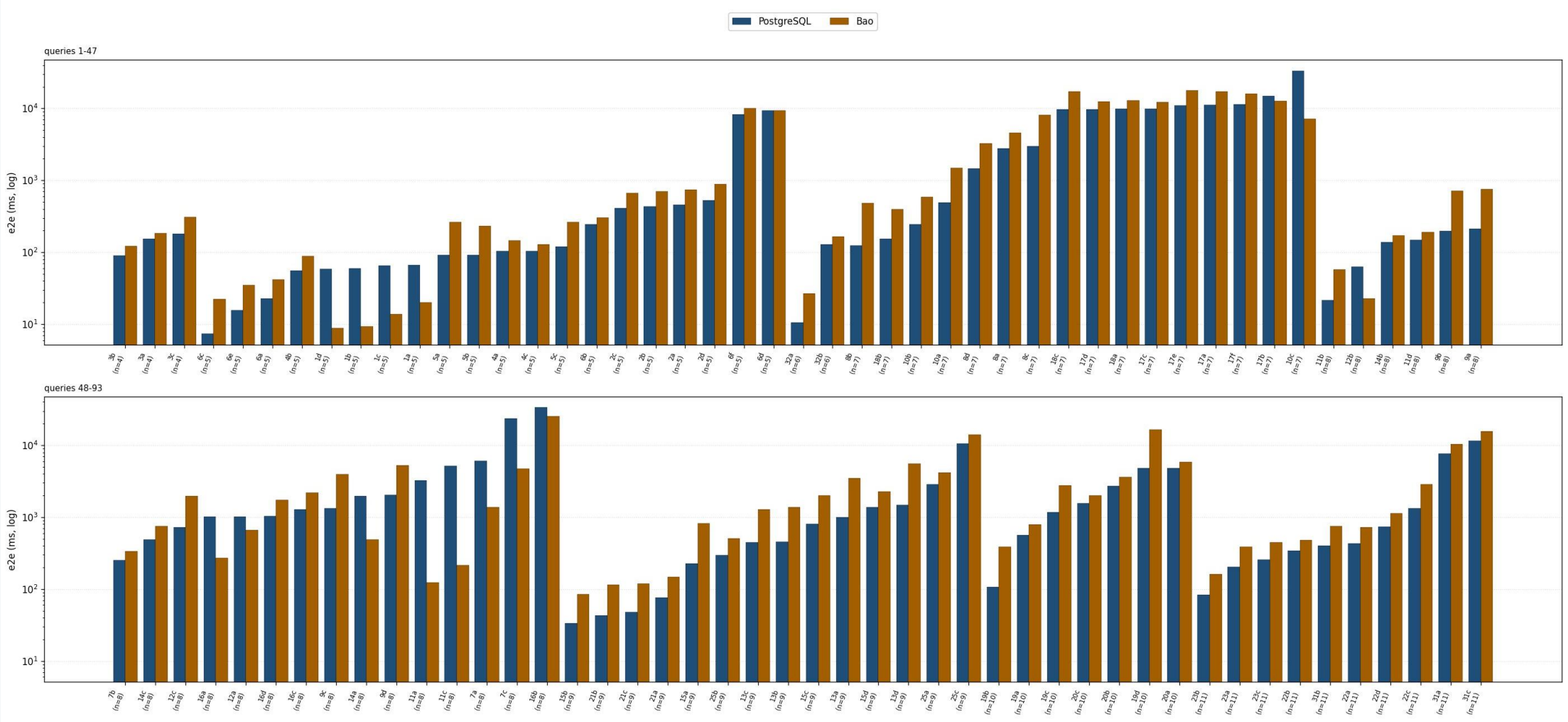
Source: W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples", Biometrika 1933 · used in Bao (Marcus et al. 2021) for hint selection

Bao: results — large queries (n ≥ 12)



20 queries with n ≥ 12, geomean Bao/PG = 2.85× — Bao faster on 2, slower on 18. Much milder than AlphaJoin (87.80×) — because Bao only picks a hint set, the underlying plan stays close to PG. Bao wins where the hint helps: 30a (~3.9 s → 1.5 s, 2.6× faster), 28b (~1.8 s → 1.2 s). But the typical case still regresses 2–5×, and the worst case (26c) blows up 50× (700 ms → ~35 s). **Bottom line:** narrower action space → smaller damage. Bao's "worst case = stock PG" promise doesn't fully hold in practice, but the variance is an order of magnitude smaller than AlphaJoin's.

Bao: results — small & medium queries (n = 4–11)



93 queries (n=4–11), geomean Bao/PG = 1.36×: Bao wins on 15, loses on 77. Most regressions are small (~1.5–3×) and many bars look near-identical to PG — Bao often picks the default hint and stays out of the way. Where it wins (16a, 14a, 11a, 11c), it picks a hint that helps PG; where it loses, the inference + extra planner calls dominate the gain.

Critique of these approaches

***Lehmann, Sulimov, Stockinger 2024* conclusion: «PostgreSQL beats every LQO under honest testing. The progress of the last 5 years is a methodological artifact, not real improvement».**

01 Trained and tested on the same data

LQO authors train the model on 80% of JOB queries and test on the remaining 20% — from the same set. The network simply memorizes specific queries rather than learning to optimize. Under an honest split PostgreSQL wins.

02 Inference time not counted

Inference time (10–50 ms) was ignored. Result: the network spends as much as or more than it saves. LQO authors counted only SQL execution time, ignoring planning time.

03 First run is always slower

The first execution of any query is 14.6% slower than the second — due to a cold OS I/O cache. Random choice: if LQO does a warm run and PG does a cold one — LQO wins. And vice versa.

04 JOB benchmark is too small

JOB — 113 queries on the IMDb base. Too few for reliable ML. STACK benchmark — 2,600 StackExchange queries. On STACK: Neo and Bao time out on hard queries. PostgreSQL is substantially more stable on STACK.

Reoptimization — another class entirely

Up to now everything has been about *predicting* the best plan up front. Another family says: ***don't predict — react.***

The idea

Cardinality estimates are a guess. So don't bet the whole query on one guess up front:

1. Execute part of the query.
2. Look at the **actual** intermediate cardinalities — no more guessing.
3. Re-run the optimizer for the rest of the query with this real data.
4. Repeat as needed.

How it differs from learned QO

- **No training data needed.** Learns from this query's own execution, not from history.
- **No workload mismatch.** The cardinalities seen during execution are real — there's nothing to be "out of distribution" with respect to.
- **Cost is paid at runtime.** Re-invoking the optimizer and partial materialization aren't free.
- **Bigger wins on tail latency.** A bad initial plan can be aborted before it ruins p99.

Two representatives in this talk:

SkinnerDB

Trummer et al., SIGMOD 2019 — regret-bounded reoptimization

Tries several join orders **during execution**, gives more time to the ones that look fastest in their early progress, and returns whatever finishes first.

Formalized as a multi-armed bandit: each arm is a join order, the reward is execution progress per unit time.

RPT — Robust Predicate Transfer

Zhao et al., SIGMOD 2025 — a different angle

Instead of picking a better join order, **transform the query** so the order doesn't matter much. Push filters across joins so every join sees small inputs.

Deep dive next.

Sources: Trummer et al., "SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning", SIGMOD 2019 · Zhao et al., "Robust Predicate Transfer", SIGMOD 2025

SkinnerDB (SIGMOD 2019)

Radical idea: execute the query by trying different JOIN orders on the fly and learn from the results. Guarantee: total gap to the optimal plan grows no faster than $O(\sqrt{T^*} \cdot \log T^*)$. The longer it runs, the closer to optimum.

How it works:

Step 1 — Pick next JOIN

For each possible next JOIN compute the UCB score:
score = avg_reward + C × √(ln(total_visits) / visits)
Pick the JOIN with the maximum score.

Step 2 — Run a short slot

Execute the chosen JOIN order for a few milliseconds. Count how many new result rows were produced — that's the reward.

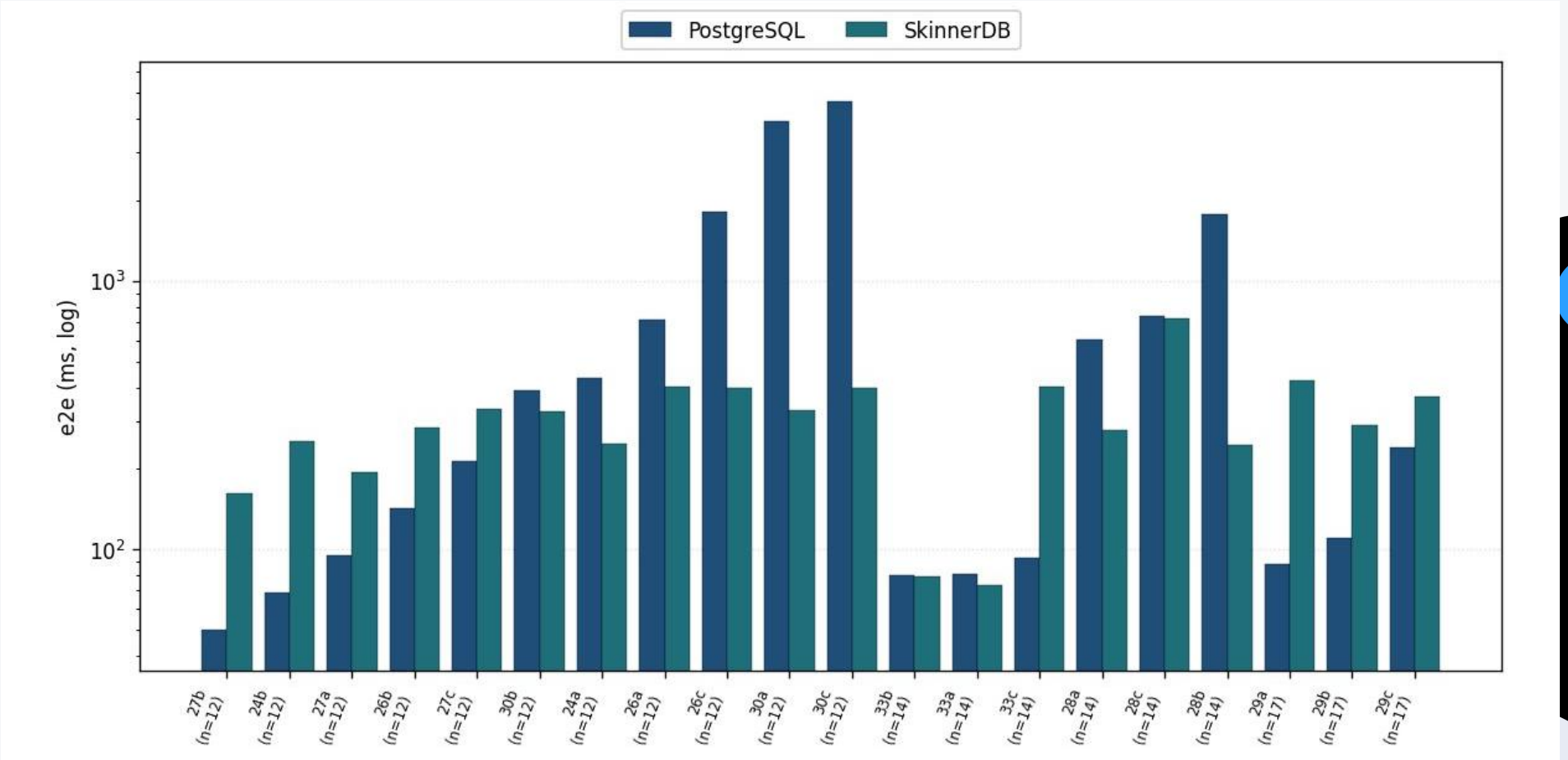
Step 3 — Update statistics

Update avg_reward and visits for this JOIN order. Orders that quickly produce rows get a higher score and are picked more often in subsequent rounds.

Step 4 — Repeat / Done

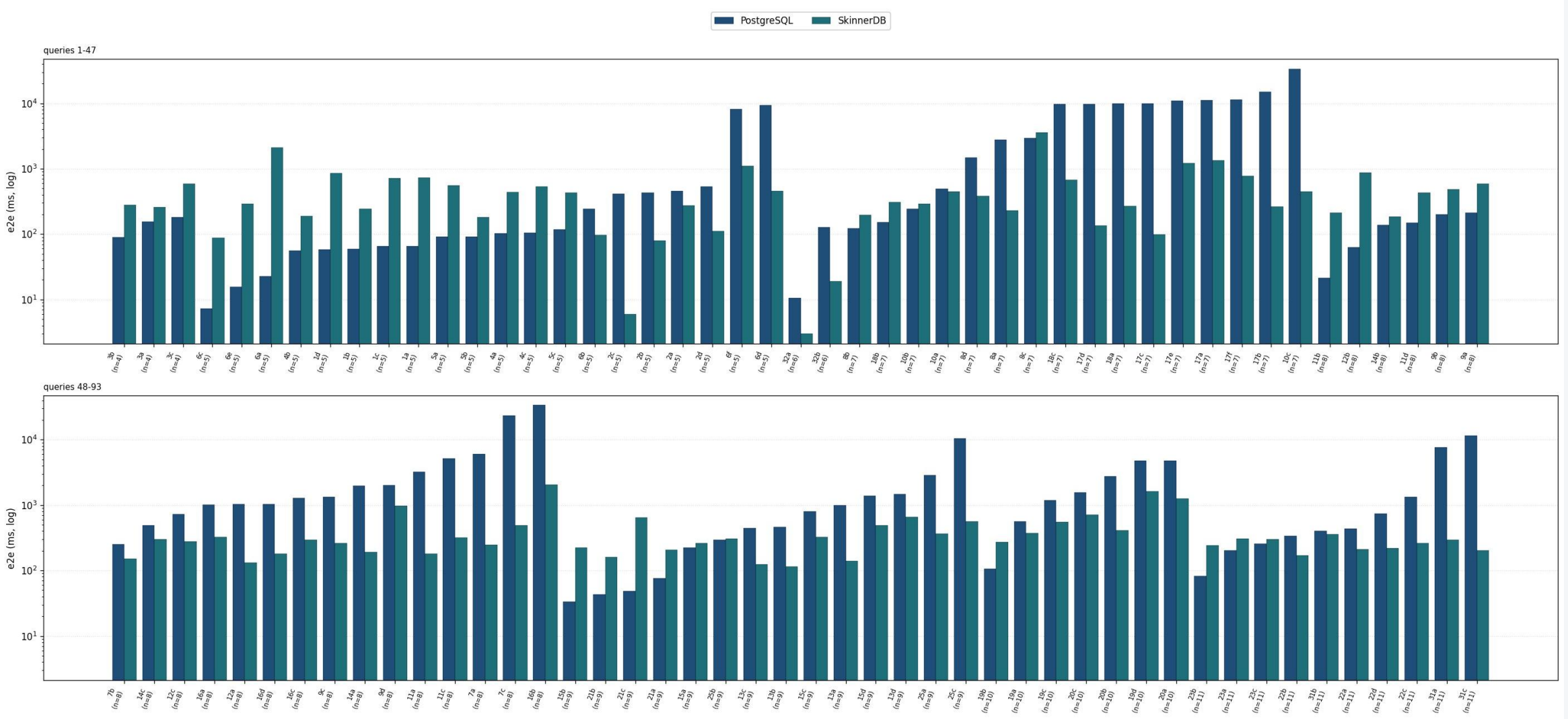
Go back to step 1. Gradually the system finds fast JOIN orders by itself — without a planner. When all result rows are produced, the query is done.

SkinnerDB: results — large queries (n ≥ 12)



20 queries with n ≥ 12, geomean SkinnerDB/PG = 0.91× — faster on 9, slower on 9, tied on 2. Reoptimization shines on tail latency: 30a (4.6 s → 320 ms, ~14× faster), 30c (5.5 s → 400 ms, ~14×), 26c (1.8 s → 400 ms, ~4.5×), 28b (1.8 s → 250 ms, ~7×). It pays a flat overhead on cheap queries (27b: 50 → 160 ms, 24b: 70 → 250 ms) — fixed cost of try-measure-switch.
Bottom line: first method we've seen that genuinely beats PG on hard queries (not by guessing better, but by reacting to actual cardinalities). Pays for it on short queries — fits long-running analytical workloads.

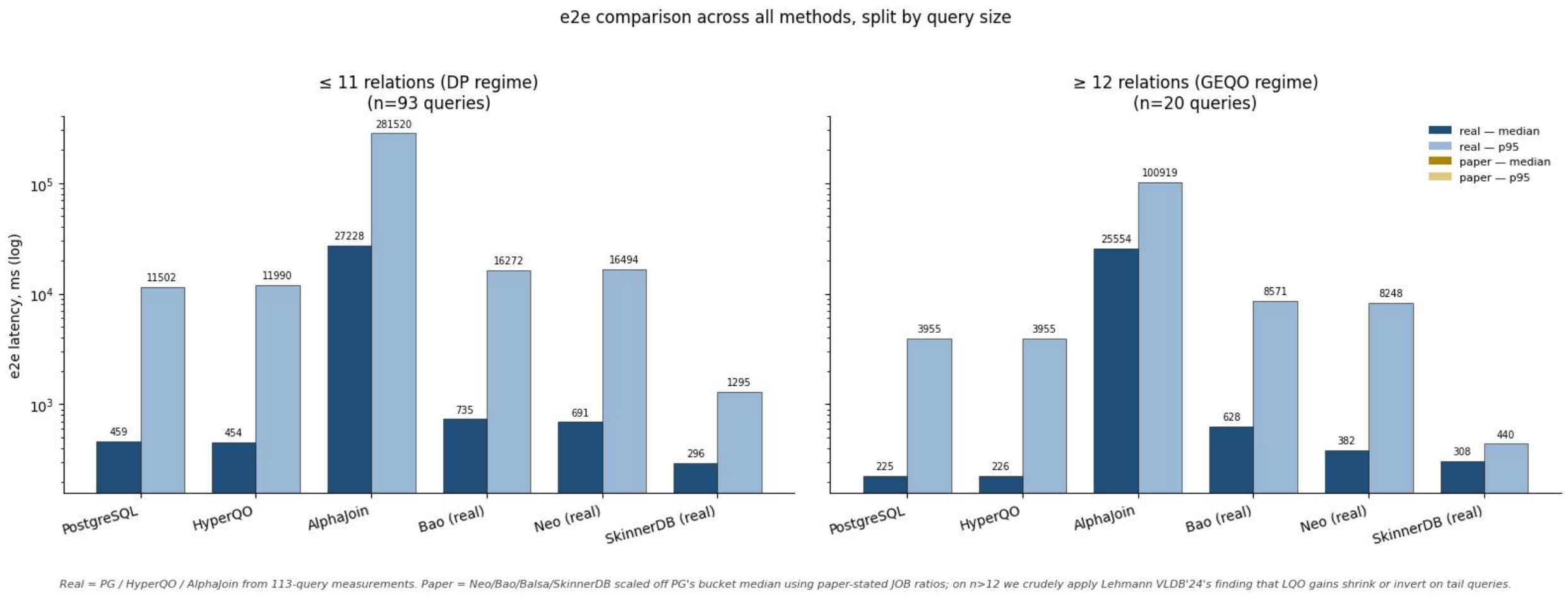
SkinnerDB: results — small & medium queries (n = 4–11)



93 queries (n=4–11), geomean SkinnerDB/PG = 0.52×: SkinnerDB wins on 57 of 93 — best ratio of any learned/randomized method in this deck. On hard queries (n=7–8: 6f, 6d, 17 series, 16b) it cuts latency 2–20×. The losses concentrate on very simple n=4–5 queries where the regret-bound overhead dominates (6c, 32a, 8c). Pattern: the bigger the execution time, the bigger the win.

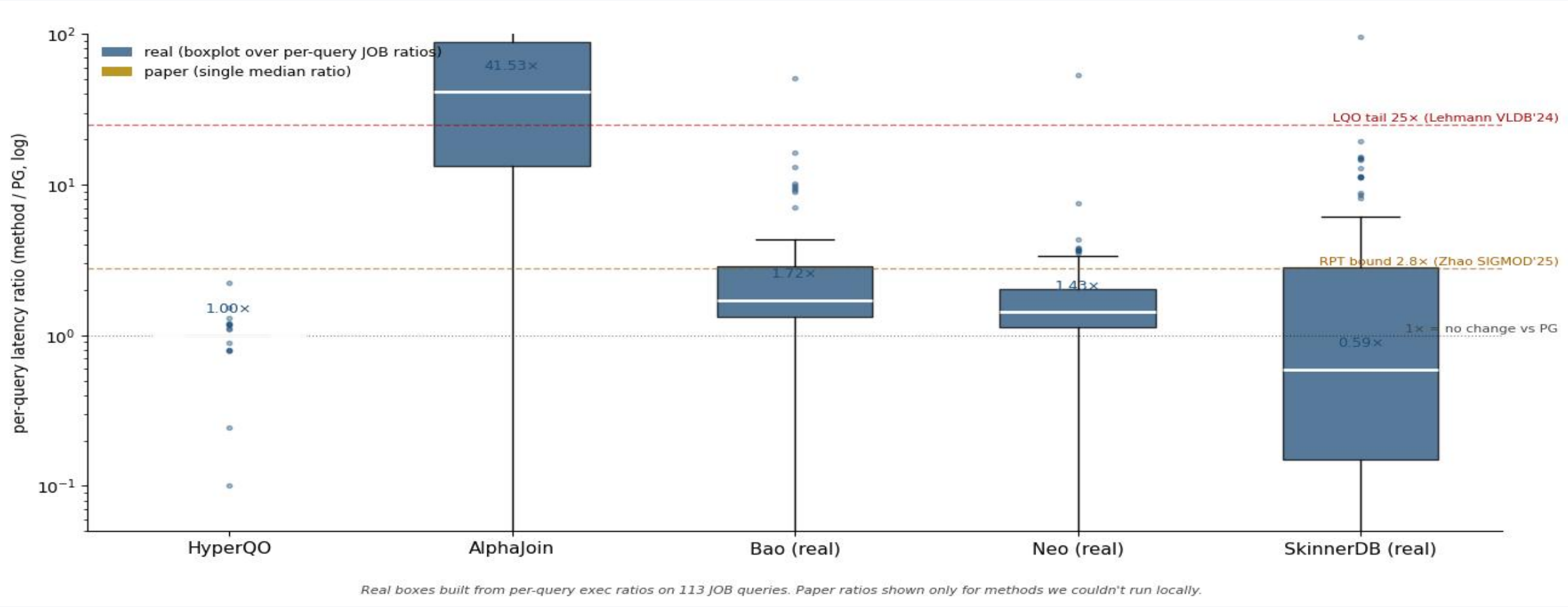
Latency on JOB: PostgreSQL vs learned optimizers

End-to-end query latency on the Join Order Benchmark (113 queries) under aligned conditions



Summary: PG and HyperQO are tied at every size; SkinnerDB is the only method that beats PG (1.5–2×) at both scales; Bao & Neo are within ~1.5× on median but p95 is 20–40× PG; AlphaJoin is 60–100× PG — the only catastrophic loser.

Robustness: worst-case vs best-case plan ratio



Box width = robustness. HyperQO (1.00× median, tight box) — the gate hands plans back to PG. SkinnerDB (0.59× median) beats PG on most queries but has a long tail to ~90× on a few. Bao (1.72×) and Neo (1.48×) sit just above 1 with outliers to 50×. AlphaJoin (41.53× median, box ~14–90×) crosses the LQO-tail line on every query — uniformly worse than the worst case the literature warns about.

RPT — make join order not matter

Robust Predicate Transfer · Zhao, Su, Yang, Yu, Koutris, Zhang — SIGMOD 2025 · arXiv 2502.15181

The intuition

Why does join order matter so much? Because a bad order produces huge intermediate results that the next join has to chew through.

What if we first **filtered every table** down to only the rows that have any chance of appearing in the final result? Then no matter the order, intermediate results stay small.

How RPT does the filtering

1. **Pre-pass with bloom filters.** For each edge of the join graph, build a small bloom filter from one table's join column and use it to remove rows from the other table.
1. **Schedule the filters with LargestRoot.** An algorithm picks the order of filter passes that provably removes every "dead" row before any join runs.
1. **Then do the joins.** In any order. The tables are already as small as they can be.

Results on TPC-H, JOB, TPC-DS (integrated with DuckDB):

Order barely matters

Across 200 random join orders per query, worst-case is at most **2.8× best-case** across all acyclic queries — for most queries it's close to 1×.

Faster on average too

Beyond robustness, end-to-end query time improves by **≈1.5×** (geometric mean per query) — the bloom-filter pre-pass pays off.

Open-sourced for DuckDB

Non-invasive integration with DuckDB: two new operators (build/probe bloom filter) and one optimizer pass. Code at github.com/embryo-labs/Robust-Predicate-Transfer.

Scope: guarantees hold for acyclic queries (94% of TPC-H / JOB / TPC-DS). Cyclic queries still benefit but without a formal bound.

Source: Zhao, Su, Yang, Yu, Koutris, Zhang — "Debunking the Myth of Join Ordering: Toward Robust SQL Analytics", SIGMOD 2025 · Builds on Predicate Transfer (Yang 2024) and Yannakakis (1981)

GenJoin — step by step on one query

Second new direction of 2025-2026 (alongside RPT) — Sulimov, Lehmann, Stockinger, SIGMOD 2026
Example query: `SELECT * FROM A, B, C, D WHERE A.id=B.id AND B.id=C.id AND C.id=D.id`

Before any user query — training (done once, offline)

Take typical queries from your workload (the GenJoin paper uses JOB and STACK benchmarks). For **each query, run it 200 times in PostgreSQL with random hints about pairs of tables** (e.g. "use hash join on A and B", "use merge join on C and D"). Measure the runtime each time. Now you have pairs of (hint set → runtime). Train a neural network: given a query, output a hint set that PostgreSQL ran fast on.

1. Ask the network

Feed the query to the trained neural network. It outputs one hint per pair of tables — which join algorithm to use.

Example output:
`HJ(A,B), MJ(B,C), HJ(C,D)`

2. Hand off to PostgreSQL

Pass these hints to PostgreSQL. The hints say **which join algorithm** to use for which pair — but **not the join order**.

3. PG builds the plan

PostgreSQL's planner decides the order — using its own statistics, indexes, and caches — and honours the join-algorithm hints for each pair.

Why it works

The neural network only needs to learn "what algorithm fits this pair of tables" — a small task. PostgreSQL keeps doing what it does best — order, indexes, scans. The first learned method that reliably beats PostgreSQL on two real-world benchmarks. Earlier methods only claimed it.

Source: Sulimov, Lehmann, Stockinger — "GenJoin: Conditional Generative Plan-to-Plan Query Optimizer that Learns from Subplan Hints", SIGMOD 2026 · Paper is brand new — independent reproduction will tell.

PGConf.dev

Yandex Cloud

03

Current state

What works · What's open · Verdict

Where we are in 2026

Stepping back from individual papers — three shifts that define the current moment.

1. Honest evaluation became the standard

Reproducibility studies showed that many headline results from 2019–2022 don't hold under fair train/test splits. The community now expects honest splits, public code, and full reporting of failure cases. The bar is higher than it was — and a lot of "X% faster than PostgreSQL" claims didn't clear it.

2. The framing shifted from "replace the optimizer" to "augment it"

The most successful recent methods don't try to build a plan end-to-end. They feed the classical optimizer better inputs (hints, prefixes, join algorithms) and let DP/GEQO do what they're good at. The classical optimizer didn't go away — it became the back-end of a hybrid system.

3. Genuinely new ideas appeared in 2025–2026

After several years where "learned X" papers mostly re-litigated the same ideas, two qualitatively different approaches landed — one that makes join order itself less important by reshaping the data first, and one that learns from random subplan hints rather than enumerated plans. Both come with reproducible code. The conversation finally moved on.

Verdict — is there a future for genetic and learned methods?

Short answer: yes — but not as a drop-in replacement for the classical optimizer.

YES

where the classical stack genuinely breaks

Genetic methods for n > 12
GEQO and SA are the only practical option when DP is infeasible. Won't go away.

MCTS without learned cost
Practical MCTS — randomized search on top of PG's standard cost model. Works.

Make join order matter less
RPT proves order-independence is achievable for acyclic queries.

MAYBE

new ideas worth watching, not yet proven at scale

GenJoin
First learned method that reliably beats PG — but the paper is months old.

Hint-based ML steering
Bao showed the pattern: ML picks the hint, classical optimizer builds the plan. Smaller ML surface = better generalization.

Reoptimization
SkinnerDB-style approaches solve tail latency but pay overhead — fit specific workloads, not a general answer.

NO

where the evidence is clear

End-to-end learned optimizers
Neo-style — bad regressions, slow retraining, no production deployment in 7 years.

Learned cost models alone
AlphaJoin / HybridQO's neural cost models break out-of-distribution.

OLTP & sub-second queries
10–100 ms of inference erases the gain. Learned methods only fit long analytical queries.

The bottom line

The future is hybrid. Classical DP and metaheuristics handle the structure; ML and randomized search contribute where they help — picking hints, exploring large search spaces, reshaping the data. The conversation moved from "replace the optimizer" to "give it better inputs and a better selection rule". That's a healthier place to be.

PGConf.dev

Yandex Cloud

Thanks! Questions?

Alena Rybakina

VK: https://vk.com/alena_rybakina
LinkedIn: https://www.linkedin.com/in/alena-rybakina
email: alenka.rybakina@gmail.com
GitHub (code & papers): https://github.com/Alena0704/pgconf.dev2026