When Not to Hint: Adapting an LLM Query Optimizer to a New Database Instance

Advait Badrish
Independent Researcher
advaitbadrish@gmail.com

Hanwen Liu
University of
Southern California
hanwen_liu@usc.edu

Ibrahim Sabek
University of
Southern California
sabek@usc.edu

Abstract

Query optimization is the problem of determining how to efficiently execute a SQL statement, and errors in a database's cost model can leave a query running orders of magnitude slower than necessary. Recent systems fine-tune large language models to correct these mistakes with a *hint* (a directive that constrains the optimizer's choices) and supervise the models with labels obtained from executing candidate plans on a single instance and recording the fastest. Supervision of this kind is specific to the instance that produced it: under the hint labels released with LLMOpt, a fine-tuned hint generator for PostgreSQL, the Join Order Benchmark runs on our own instance at under half the speed of the default optimizer, and models we fine-tune on the labels similarly underperform. Adapting such a model to a new instance does not require relabeling and retraining on all of its data. We relabel 500 of $8{,}592$ training queries with the locally fastest option, including not hinting, and continue training the model on the result. A gate based on the default optimizer's cost estimates then selects a sampled hint. On the benchmark, the resulting pipeline hints on roughly a third of queries, speeding them up by $1.451\times$ and speeding up the whole benchmark by $1.081\times$, above both the default optimizer and the foreign-labeled model it starts from.

1 Introduction

PostgreSQL chooses among admissible execution plans with a cost model that depends on estimates of how many rows each intermediate join will return, and those estimates are often incorrect in ways that compound along a chain of joins [9]. A body of work trains models to correct the resulting mistakes through *hints*, per-query directives, read by PostgreSQL before planning, that can fix some or all of the decisions it would otherwise make on its own. Bao [13] learns online which of a fixed menu of optimizer settings to apply to each incoming query, one of several designs that search a bounded space [4, 19]. More recent systems fine-tune language models to write hint strings directly: LLMOpt(G) [18] reranks its sampled candidates with Bao's cost model; LLM-QO [17] applies preference optimization over plan-quality pairs; and SEFRQO [10, 11] retrieves past query, hint, and latency tuples under a reward computed by executing every sampled candidate. They share a source of supervision. Candidate plans are executed on a database instance, and the latencies that come back decide which hint the model is trained to prefer. They differ in where those latencies are measured. SEFRQO measures on the database it will serve, while LLMOpt distributes labels measured once on the authors' instance. Whether such labels transfer between instances is, to our knowledge, untested. We test LLMOpt because it publishes the candidate plans together with the latencies that ranked them, without which another system's labels cannot be re-executed elsewhere.

Under the hint labels distributed with LLMOpt [18] (*foreign labels*), the Join Order Benchmark runs on our PostgreSQL instance at $0.379\times$ the speed of the unmodified optimizer, and no Llama-3.1-8B

The 10th Machine Learning for Systems Workshop at Neurips 2026, Sydney, Australia

model [7] we fine-tuned on those labels beats leaving the optimizer alone. Each of those labels was the fastest option tried on the instance that produced it, but none of the advantage reproduces on ours. A label records which plan was best under one instance's version, configuration, and hardware, so it is specific to the instance that produced it, not only to the query it was written for.

The label format assigns a hint to every query, including the many where no candidate beats the plan PostgreSQL would have chosen unaided. When we re-execute a sample of training queries on our instance, no hint is the fastest option available for 46% of them. A model trained on these foreign labels hints everywhere. Akioyamen et al. [1] steer PostgreSQL with a classifier choosing between the default plan and one precomputed alternative, so declining remains available to it. A model writing hint strings token by token can decline only if its training labels include declining. Deferring to a default is studied in classification as the reject option, where a model may abstain [3, 5, 12], and is safe in our setting, since the fallback is the database's own optimizer.

Adapting a model to the instance it will run on takes local executions to relabel training data and some retraining, and little of either is needed. First, we re-execute 500 of the $8{,}592$ training queries on the target instance, relabel each with the fastest available candidate including no hint, and continue training the foreign-labeled model on the result (5.5% of its gradient samples). Second, at inference, a cost gate built from PostgreSQL's own estimates screens 8 sampled hints per query without any database executions, in contrast to 48 ranked candidates for Bao, 16 reranked for LLMOpt(G), and the best of 50 online executions for SEFRQO. Neither local retraining nor the cost gate is enough by itself. However, together they outperform the default optimizer, which no tested setup trained on foreign labels does.

2 Method and Measurement

Task and data Each fine-tuned model receives a SQL query with row-count statistics and emits either a `pg_hint_plan` [14] hint string, naming a join order, a join algorithm per join (nested loop, hash, or merge) and a scan method per table (sequential or index), or no hint at all, which leaves planning to PostgreSQL. Our test set is the Join Order Benchmark (JOB) [9], 113 analytical queries over IMDb; training and validation are $8{,}592$ and 100 additional queries over the same schema, distributed with LLMOpt [18] and generated from 33 templates. No test query appears in training, though each shares its join graph with a training query. Labels come from five of Bao's 48 operator settings [13], coarse enable and disable switches over operator classes, which are not per-query hints (Appendix A).

Training Every model here is Llama-3.1-8B [7] fine-tuned to write hint strings, the model size used by LLMOpt and SEFRQO's fine-tuned model [10, 11]. *Foreign labels* are the `argmin` over Bao's five settings as measured on the instance that produced them [18], and we fine-tune on $8{,}592$ of them for $34{,}368$ gradient samples, during which the model never sees an example of abstaining because every query carries a hint. To test whether curriculum ordering [2] matters, we train two such models on the same queries for the same number of steps, the *ordered* one receiving queries grouped by how many tables they join and adding harder groups as training proceeds, and the *shuffled* one drawing from all groups throughout.

Local adaptation continues training either model on 500 training queries drawn uniformly at random and relabeled on our instance, and we repeat it at smaller doses of 124 and 248 relabeled queries (compute-matched) to trace how the effect changes. For each query, we execute the same five frozen plan strings and take the `argmin` over {no hint} $\cup$ the five, allowing abstention as an option in the label set. The two label sets disagree on 67% of these queries. Collecting them costs $13{,}342$ executions, paid once per instance, and the training is 63 optimizer steps.

Cost gate Local relabeling trains the model to decline, but the model makes its decision from only the query text and statistics. A second and independent check is cheap: PostgreSQL already estimates the cost of every plan it considers, and the SQL `EXPLAIN` command reports that estimate for a hinted plan without needing to execute it. Writing $c(p)$ for the estimated cost of plan p and p_0 for the plan produced with no hint, the *cost gate* considers the $k{=}8$ plans sampled at temperature $t{=}1.0$. It keeps those that `pg_hint_plan` accepts in full (69.9% of hinted outputs) and that satisfy $c(p_i) \le T\, c(p_0)$, emits the cheapest of them, and emits no hint when none qualifies. We fix $T{=}2$ throughout, so only plans estimated at most twice the unhinted plan's cost are admitted. The bound

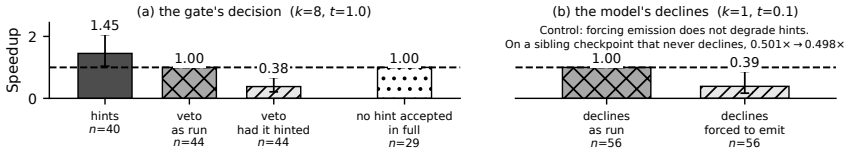


Figure 1: (a) The cost gate over $k{=}8$ samples ($t{=}1.0$). Queries it vetoes are shown as run and as they would have run had it hinted; the 29 with no valid hint are shown separately. (b) The $56/113$ queries the adapted model declines at $k{=}1$ ($t{=}0.1$), as run and with hint emission forced. 95% bootstrap intervals (Appendix A).

is loose because these estimates are unreliable, and its purpose is to reject catastrophic plans priced far above the default rather than to rank the ones that are not. The value was fixed once on a different set of hints and not retuned here (Appendix A).

Measurement All latencies come from a machine running PostgreSQL 12.5 [16], configured as Appendix A reports. We repeat the transfer test on a second machine, differing from ours only in hardware, and on a StackExchange workload run on that machine. Foreign labels fail on both, while labels built on our instance carry over to the second machine (Appendix A). A fixed $60\,\mathrm{s}$ `statement_timeout` sits above the slowest measured baseline ($19.65\,\mathrm{s}$), so no baseline is censored. A hinted execution that reaches the cap is charged there instead of being dropped. Each query and hint pair runs five times after one warmup and we report the median execution latency. Speedup is total baseline latency divided by total latency under the policy being scored, so a value below 1.0 means the hinted workload ran slower. A figure written $x\,[a, b]$ carries a 95% bootstrap confidence interval over queries.

3 Evaluation

The foreign hint labels are worse on our instance than what the default optimizer would have chosen, but the lost time is recoverable in hindsight. An *oracle* chooses whichever option turned out fastest for each query, so it reads runtimes knowable only after the query has run. It bounds any policy over a pool of candidates, and is not a policy itself. An oracle allowed to fall back on PostgreSQL wherever a foreign label would lose lifts those labels from $0.379\times$ to $1.444\times$ while leaving every hint string as released. 71 of the 113 foreign labels slow their own query down, several of them catastrophically.

Oracles over the model's own hints We put our fine-tuned models through the same test. For each we sample $k{=}8$ hints per query and compute two oracles over that one pool, one obliged to emit a hint and one free to emit nothing. The gap between the two therefore measures the abstain option alone. Forced to hint, the best any model manages is $0.864\times$ (on the 93 queries where it emits any hint), and permitted to decline, the *worst* reaches $1.229\times$. This holds on six of six checkpoints, spanning both label sources, both training orders, and all three relabeling doses.

Hints the model declined to write Both oracles pick among hints the model actually wrote, leaving open whether the queries it declines are risky ones. On queries that the model declined, we generate again with the abstain action removed, masking the end-of-sequence token until a hint parsed as complete [6, 15]. The free decoder emits nothing on $56/113$ queries, and the hints the constrained decoder writes in their place run at $0.388\times$ $[0.168, 0.864]$ (Figure 1b). Thus, the model was withholding hints that would have slowed those queries down (Appendix A).

Speedup under the cost gate Applied to the ordered model after local adaptation, the cost gate reaches $1.081\times$ $[1.006, 1.197]$ over all 113 test queries while hinting on $40/113$. On queries it hints, it reaches $1.451\times$ $[1.038, 2.062]$, and on the 44 queries it vetoes, using the cheapest vetoed hint would have run them at $0.381\times$ $[0.210, 0.674]$ (Figure 1a). The same rule over eight samples of each foreign-labeled model's output leaves them at $0.967\times$ and $0.968\times$, below the unmodified optimizer. Scored on the same queries, local adaptation gains $+0.114$ $[+0.013, +0.208]$ under the same gate, so cost screening does not by itself rescue foreign supervision; local adaptation supplies

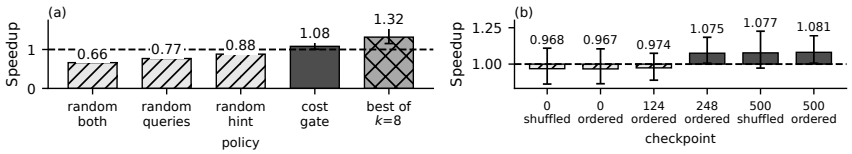


Figure 2: (a) One pool of $k{=}8$ hints from the adapted model, scored under the cost gate, three random policies matched to its hinting rate, and the oracle (best of the eight). (b) The gate on every checkpoint, labeled by relabeled queries and training order. The dashed line is PostgreSQL, which a plain cost minimum over Bao's five frozen plans also returns (Appendix A).

the difference. The shuffled model reaches $1.077\times$ $[0.972, 1.226]$ hinting on $61/113$. The difference between the two training orders without the cost gate, $+0.137$ $[-0.056, +0.339]$, favors the ordered model but its interval includes zero (Appendix A).

The overall speedup is not spread evenly over the benchmark, as queries joining ≥ 12 tables carry almost the entire aggregate improvement (Appendix A). The candidate pool matters as well, since a plain cost minimum over Bao's five frozen plans finishes at $1.002\times$ (Figure 2b; Appendix A). The gate improves every checkpoint but leaves three of six below never hinting: the two foreign-labeled checkpoints and the smallest local dose. The cost gate separates a catastrophic plan from a reasonable one but cannot reliably rank two reasonable ones.

Controls A policy that declines on many queries cannot be fairly compared with one that always hints, since part of its advantage comes from abstaining. We therefore built three random policies that hint on exactly as many queries as the gate does, randomizing which queries to hint, which candidate to use, or both (Figure 2a). All three land below the gated policy, between $0.662\times$ and $0.881\times$, and the bootstrap probability of any of them matching or beating it is at most 0.0083. All 48 single-sample runs (six checkpoints, eight sampling seeds each) are slower than never hinting, so the gain comes from choosing among samples, not from any one of them.

Inference overhead Generating a query's eight samples as one batch takes a mean of $5.96\,\mathrm{s}$ on one GPU (106.5 tokens/s), counting the declines that make up 52% of samples and cost a single token. We report it because a learned optimizer can lose its advantage once model time is charged against the time it saves [8]. Hints are generated before a query first runs and reused on every later execution, amortizing the charge.

4 Conclusion

Supervision built from executed plans is an artifact of the instance that executed them. Labels released with a published system, re-run on a different instance, are slower than the optimizer they were meant to improve; LLMOpt's published labels for Bao's Stack workload [13] reach $0.364\times$ on the second machine (Appendix A). Re-executing a small sample of the training queries costs $13{,}342$ executions collected once and rebuilds the labels around the fastest released candidate, including not hinting. A cost gate then screens the adapted model's 8 samples per query through PostgreSQL's own cost estimates, and together they beat the unmodified optimizer without executing candidates at inference.

Limitations We adapt to one database instance, and the second machine we test on matches its configuration. Appendix A reports this instance's settings and measures the effect of three of them. Systems that execute candidate plans on the target instance, such as Bao, learn from that feedback directly and are the stronger option wherever a workload repeats often enough to pay for those executions. Inference and selection need no execution feedback, but generation cost is repaid only once a query recurs (Appendix A). The abstention result is based on one training seed, and how much the gate recovers depends on both the cost model it screens with and the candidates it screens.

References

[1] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. The unreasonable effectiveness of LLMs for query optimization. In *Machine Learning for Systems Workshop, NeurIPS 2024*, 2024. arXiv:2411.02862.

[2] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009)*, pages 41–48. ACM, 2009. doi: 10.1145/1553374.1553380.

[3] C. K. Chow. On optimum recognition error and reject tradeoff. *IEEE Transactions on Information Theory*, 16(1):41–46, 1970. doi: 10.1109/TIT.1970.1054406.

[4] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. Kepler: Robust learning for faster parametric query optimization. *Proc. ACM Manag. Data*, 1(1):109:1–109:25, 2023. doi: 10.1145/3588963. arXiv:2306.06798.

[5] Yonatan Geifman and Ran El-Yaniv. Selective classification for deep neural networks. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, pages 4878–4887, 2017. arXiv:1705.08500.

[6] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2023)*, pages 10932–10952, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.674. arXiv:2305.13971.

[7] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. The Llama 3 herd of models. arXiv preprint arXiv:2407.21783, 2024.

[8] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. Is your learned query optimizer behaving as you expect? A machine learning perspective. *Proc. VLDB Endow.*, 17(7):1565–1577, 2024. doi: 10.14778/3654621.3654625. arXiv:2309.01551.

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

[10] Hanwen Liu, Qihan Zhang, Ryan Marcus, and Ibrahim Sabek. SEFRQO: A self-evolving fine-tuned RAG-based query optimizer. *Proc. ACM Manag. Data*, 3(6):361:1–361:27, 2025. doi: 10.1145/3769826. arXiv:2508.17556.

[11] Hanwen Liu, Qihan Zhang, Ryan Marcus, and Ibrahim Sabek. SEFRQO-Plus: A self-evolving query optimizer via large language models and retrieval-augmented generation. *The VLDB Journal*, 35(6):58, 2026. doi: 10.1007/s00778-026-00999-z.

[12] David Madras, Toniann Pitassi, and Richard Zemel. Predict responsibly: Improving fairness and accuracy by learning to defer. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, pages 6147–6157, 2018. arXiv:1711.06664.

[13] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proc. 2021 International Conference on Management of Data (SIGMOD '21)*, pages 1275–1288. ACM, 2021. doi: 10.1145/3448016.3452838. arXiv:2004.03814.

[14] OSSC-DB. `pg_hint_plan`: PostgreSQL extension for optimizer hint control. `https://github.com/ossc-db/pg_hint_plan`, 2020. Version 1.3.7, accessed 2026-08-30.

[15] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. Grammar-aligned decoding. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, pages 24547–24568, 2024. arXiv:2405.21047.

[16] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source relational database. `https://www.postgresql.org`, 2024. Accessed 2026-08-30.

[17] Jie Tan, Kangfei Zhao, Rui Li, Jeffrey Xu Yu, Chengzhi Piao, Hong Cheng, Helen Meng, Deli Zhao, and Yu Rong. Can large language models be query optimizer for relational databases? *Proc. ACM Manag. Data*, 3(6):306:1–306:28, 2025. doi: 10.1145/3769771. arXiv:2502.05562.

[18] Zhiming Yao, Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. A query optimization method utilizing large language models. arXiv preprint arXiv:2503.06902, 2025.

[19] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. Lero: A learning-to-rank query optimizer. *Proc. VLDB Endow.*, 16(6):1466–1479, 2023. doi: 10.14778/3583140.3583160. arXiv:2302.06873.

A Experimental Details

Label construction The foreign labels and prompt statistics come from LLMOpt's released files, but the prompt template is our own. Bao's five settings [13] disable, respectively, nothing; `nestloop`; `indexscan` and `mergejoin`; those two plus `nestloop`; and `mergejoin` alone. The first setting is PostgreSQL's own policy. Coverage on JOB test is 113/113/113/30/10 queries per setting. The hint strings released with these labels are the concrete plans the collecting instance materialized under each setting and serialized as a full `Leading()` specification with scan and join methods, so they appear query-specific although the underlying settings are coarse. An oracle over five settings is therefore a lower bound on what hinting can achieve on this instance. The settings are switches over operator classes, and the stored label is whatever plan the collecting optimizer built once those switches were set. LLMOpt collected its labels with a patched `pg_hint_plan` [18]. On our instance the stock extension reproduces each labeled plan's scan and join methods for all 379 frozen strings. Relabeling took 13,342 executions: 2,231 query-plan measurements at one warmup and five repeats, less four executions for each of 11 timeouts, over about 11 hours on the shared machine.

Instance configuration The evaluation machine has an Intel i9-9980XE (18 cores) and 126 GiB of memory. It runs with `enable_bitmapscan`=off, `enable_tidscan`=off, `work_mem`=4 GB, `temp_buffers`=32 GB, `effective_cache_size`=4 GB against `shared_buffers`=32 GB, and `geqo`=off (PostgreSQL's genetic join-order search). Disabling bitmap scans is not symmetric between hinted and unhinted runs. `pg_hint_plan` [14] writes the enable flags when a hint names a scan method, so a hinted plan may use a bitmap scan where the unhinted baseline may not. We measured what that asymmetry is worth, re-running the unhinted baseline on all 113 test queries with the setting on and with it off under the protocol of Section 2. Disabling bitmap scans leaves the unhinted baseline slightly faster: 169.4 s against 174.6 s in total, with a median per-query ratio of $0.996\times$. Running the whole policy with bitmap scans enabled for both baseline and hints gives $1.045\times$ $[0.990, 1.118]$, against $1.081\times$ here. The difference is one query, 29c, whose default plan uses a bitmap scan when allowed (0.87 s against 7.39 s); the gate makes the same decisions under both settings. Under `effective_cache_size`=96 GB, default `geqo`, or both, the gate makes the same decision on 105–112 of 113 queries. The larger cache alone gives $1.087\times$ $[1.020, 1.198]$, and across five GEQO seeds no interval falls below the default.

Measurement noise Repeating a full measurement pass on the same instance changes no single-sample run by more than $0.005\times$. This bounds measurement noise but not generation variance.

Cost and per-query outcomes Including planning time, the gated model runs at $1.092\times$, since a hinted plan narrows the optimizer's search (11.39 s of planning against 14.13 s). With batched generation, preparing eight samples per query breaks even after about 50 passes of the workload. Of the queries the gate hints, 20 run more than 5% faster, 12 within 5%, and 8 more than 5% slower. None time out.

Hint application `pg_hint_plan` [14] applies the parsed prefix of a hint string and silently ignores the remainder, so a hint the parser rejects partway is not equivalent to an unhinted run and does not receive a free $1.0\times$. Only a hint that parses nothing at all falls back to PostgreSQL's plan. We therefore never treat rejection as abstention.

Choosing the tolerance We took T from an earlier experiment on a different set of hints. Taking the peak of the curve of speedup against T on a tuning split puts T one grid point past the value at which catastrophically slow hints begin to be admitted, which returns $0.686\times$ against $1.046\times$ at the frozen value on the same checkpoint and the same split. While developing the gate we read a tolerance off a sweep computed on the same queries we then quoted it against. That reading was biased upward by construction and we discarded it.

Choice of reported configuration We report the ordered model after local adaptation because it is the full configuration, including the ordered base, all 500 local relabeled queries, and the cost gate. The foreign-labeled models and the shuffled model each lack at least one of these, and the models relabeled on 124 and 248 queries were trained afterwards to vary the number of relabeled queries at matched compute. The cost gate, k, the sampling temperature, and T were fixed before any sampled hint was executed. Two other checkpoints reach similar point estimates: $1.075\times$ $[1.006, 1.186]$ with 248 relabeled queries, which does not clear all three matched random controls, and $1.077\times$ $[0.972, 1.226]$ for the shuffled model, whose interval includes never-hinting.

Figure conventions Figure 1a partitions all 113 test queries into the three outcomes the cost gate produces: it hints, it vetoes every candidate, or the model supplies no valid candidate. Vetoed queries appear in two bars: one at the baseline they ran at and one at what their cheapest hint would have cost. Panel (b) forces emission on the $56/113$ queries the ordered model after local adaptation declines with $k{=}1$ sample ($t{=}0.1$), and its control line applies the same procedure to a sibling checkpoint that never declines, where free and constrained decoding give $0.501\times$ vs. $0.498\times$. Intervals in both figures are 95% bootstrap over $10{,}000$ resamples of the per-query totals. In Figure 2 the gate selects by PostgreSQL's cost estimate under a threshold frozen before the run and executes nothing to decide. Each P value is the fraction of resamples in which the matched random control equals or beats the gate: 0.0032 against random-both, 0.0083 against random-queries, and 0.0068 against random-hint. Positive gate headroom first appears between 124 and 248 locally relabeled queries.

Effect of training order Training order is measured twice: -0.013 $[-0.054, +0.036]$ between the foreign-labeled checkpoints, and $+0.137$ $[-0.056, +0.339]$ between the same pair after local relabeling, which is the contrast Section 3 reports. Both intervals include zero. The contrast also depends on the estimator used to summarize it. Across five summaries of the same samples, the relabeled pair spans $+0.137$ (one sample, averaged over eight sampling seeds), -0.108 (one sample, one seed), $+0.004$ (the cost gate over eight), and -0.155 and $+0.166$ (best of eight with and without declining, the last on the 93 queries where both emit a hint). We report the first, as in Section 3.

Why the frozen-plan pool yields no gain The gate admits candidates priced within a tolerance T of the unhinted plan and takes the cheapest, which leaves it hinting on $40/113$ of the test queries. For Bao's five frozen plans we report a plain cost minimum over a pool that already contains the unhinted plan. Under the plain cost minimum, our model's samples reach $1.016\times$ $[1.003, 1.039]$. Under the gate, the frozen pool falls to $0.708\times$ $[0.458, 1.012]$. PostgreSQL's own plan is priced lowest on 98.3% of queries: the unhinted plan on 61.1%, and a frozen copy of that same policy, Bao's first setting that disables nothing, on a further 37.2%. The plain cost minimum therefore keeps PostgreSQL's plan on almost all queries.

Scope of the forced-hint comparison End-of-sequence masking can affect more than the first token, since the mask can prevent termination where free decoding permits ending generation early. The constrained decoder can therefore rewrite hints on queries where the free decoder already emitted a hint. In our runs, it rewrites $30/113$ for the never-declining checkpoint, whose aggregate speedup is unchanged, and 3 of the 57 hints the declining checkpoint did emit. We limit this interpretation to the $56/113$ queries where the free decoder emits nothing and no existing hint can be perturbed.

Table 1: The cost gate by query size, JOB test queries.

Tables joined	n	Hinting rate	Gate	Oracle	Headroom taken	Share of baseline time
≤ 8	62	27%	0.999×	1.236×	−0.4%	64.6%
9–11	31	29%	1.005×	1.164×	+3.3%	20.6%
≥ 12	20	70%	2.004×	2.483×	+67.7%	14.8%

Difficulty groups Table 1 gives the grouping behind the result in Section 3. Queries are grouped by the number of tables joined, which is also the metric our difficulty ordering uses. Queries joining $\geq$ 12 tables carry 14.8% of PostgreSQL's total time and almost the entire aggregate gain. The oracle in each row minimizes over every executed sample from the same pool the gate chooses from, so the Gate and Oracle columns are comparable. Headroom is the fraction of the oracle's available gain that the gate achieves. The same grouping applied to all six checkpoints shows the same concentration. Ranking queries by their own baseline latency instead produces no trend, and the two rankings are nearly independent (Spearman correlation $+0.094$). A per-query median is uninformative for a selective policy, because many queries are not hinted and each scores $1.0\times$.

Replication on another machine To separate hardware from configuration, we repeated the comparison on a second machine with a Xeon E5-2698 v4 and 188 GiB of memory, running the same PostgreSQL version under the same session settings as ours; only the hardware differs. Foreign labels reach $0.397\times$ there, against $0.379\times$ here. Our instance's local argmin labels ($1.503\times$ here), replayed without re-running the model or the gate, reach $1.740\times$ on that machine, against a local argmin of $1.827\times$ computed on it. Matching the settings isolates hardware. Whether our configuration contributes to the failure is not tested here. The StackExchange figure in the conclusion comes from the same machine in a separate session on a different database, with `enable_bitmapscan=on`, and is quoted against the unhinted baseline measured in that session.

Comparison to prior work Candidate count alone does not permit a fair speed comparison: a minimum over N executed plans beats a single draw even when the candidates are drawn from the same distribution. We therefore compare executions spent per decision. The oracles of Section 3 are subject to the same objection and are reported only as diagnostics. Declining is charged at $1.0\times$, not dropped, and every figure covers all 113 test queries. Where a comparison forces a hint instead, it covers only the queries holding an executable candidate, and that count is stated alongside it.

Further limitations The results from LLMOpt we cite were measured on different machines under configurations reported in too little detail to reproduce. Our numbers and theirs share no common axis. Separately, the local supervision that produces our best model is a small fraction of its gradient samples, the remainder being the foreign labels the model was first trained on. That model's performance is therefore not an upper bound for the method. Finally, every abstention and gate result here is JOB on one schema. Whether the abstain action carries the same value on a different workload is untested.