Optimus: Deployable Query Optimization via Novel SQL Rewrites

Raahim Lone

Independent
raahimlone@gmail.com

Abstract

Learned database query optimizers typically optimize over a set of configurations, which limits the attainable plan space. **Optimus** expands the *action space* itself by *mining novel execution plan rewrites* and learns to select among these actions online. Optimus utilizes graph-based inductive matrix completion and a multilayer perceptron with the objective of minimizing latency. Crucially, the system is deployable by design: it requires no engine modifications and its rules include guard-checked compilation. On the extended JOB benchmark, Optimus yields a speedup of $1.16\times$ over vanilla PostgreSQL solely using novel rewrites.

Introduction

Modern data systems transform SQL queries into representations called *query plans*, which dictate how data is retrieved. The component responsible for generating these query plans is known as the *query optimizer*, which typically rely on heuristics and cost models to make decisions (Selinger et al. 1979; Leis et al. 2015). Recent research explores **learned query optimizers**—systems that apply machine learning techniques to help guide the database's built-in query optimizer (Pavlo et al. 2017; Marcus et al. 2019, 2021; Trummer et al. 2019; Khan et al. 2023). In production, these query optimizers must be deployable; they must deliver plan quality alongside safe application, operator control, and auditable behavior.

These learned systems guide the query optimizer by suggesting **query rewrites** in the form of *hints*—statements embedded into an SQL query that tune exposed configuration parameters (knobs, join orders, indexing strategies, and more) (pgh 2025). By tuning these parameters, query plans can process data more efficiently, resulting in reduced SQL *query latency*, or the time taken to run a SQL query within the database (Marcus et al. 2019, 2021; Zhang et al. 2018, 2019, 2024; Yi et al. 2024; Khan et al. 2023; Wilson et al. 2024; Lyndall et al. 2025; Aziz et al. 2024).

This creates a **deployability problem**: because interventions are restricted to global parameters, operators cannot apply or roll back changes at the granularity of individual plans. Furthermore, the lack of explicit, guardable rewrites prevents fine-grained control. This makes it difficult to revert optimizer decisions when service-level objectives (SLOs) degrade.

Concretely, the absence of a novel rewrite surface yields three deployability gaps that practitioners care about:

- **(D1) Object-level scope:** beyond per-query decisions, operators need actions scoped to specific plan decisions to reduce latency.
- **(D2) Safety:** without guard-checked, human-readable rewrites, operators cannot review, whitelist/blacklist, or prove applicability.
- **(D3) Integration without engine changes:** query rewrites often require touching optimizer internals instead of emitting engine-native hints from the SQL layer.

To address the query optimization root cause and its deployability fallout (D1–D3), **Optimus** is presented: a learned system that *mines novel, structural plan rewrites*. Rewrites are encoded in a small, human-readable domain-specific language (DSL). They are admitted only if they compile to engine-native `pg_hint_plan` (pgh 2025) hints under explicit guards. Operating entirely at the SQL layer (no engine modifications), Optimus restores per-query control (addresses D1), provides guard-checked rules (addresses D2), and integrates via engine-native hints (addresses D3).

System overview. Optimus operates within a two-stage pipeline geared towards real-world deployment:

- **Mining:** Optimus uses heuristics and an instruction-tuned large language model (LLM) to generate novel plan rewrites based on observed plans and runtimes. These rewrites are encoded in a DSL in order to provide safety.
- **Prescription/Inference:** Optimus represents query–rewrite pairs in a joint space using inductive graph matrix completion (IGMC) (Zhang and Chen 2020) and a multilayer perceptron (MLP), allowing it to predict which candidate rewrites are likely to improve latency.

In preliminary experiments, Optimus achieves performance exceeding that of PostgreSQL baselines while solely relying on actionable configurations that existing research has yet to cover.

The **contributions** are as follows:

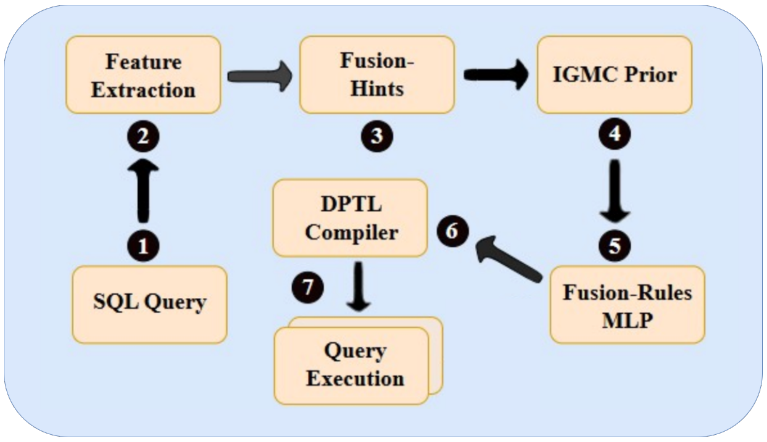


Figure 1: This figure demonstrates Optimus' overall online pipeline upon receiving an SQL query.

- **A novel LLM-guided pipeline** that discovers query rewrites beyond the reach of conventional parameter tuning methods, with a DSL that encodes guards/bindings for auditable application.
- **A learned query optimization method** that leverages inductive matrix completion in combination with a multilayer perceptron to prioritize beneficial structural rewrites.
- **A deployability contract**: guard-checked compilation, enabling operator control with little engine modifications.

Optimus: System Overview

Optimus runs on PostgreSQL via inline `pg_hint_plan` hints—no engine modifications (addresses D3)—and executes only guard-checked rewrites (addresses D2). These hints operate at the plan-subtree level rather than toggling global parameters, enabling per-plan control (addresses D1). For deployability, Optimus runs entirely at the SQL layer and emits only engine-native hints, meaning it can integrate without touching optimizer internals (addresses D3). Critically, each rewrite is encoded in a small, human-readable DSL with per-rule toggles, making rewrites at the granularity operators actually need.

Upon receiving a raw SQL query **(1)**, Optimus follows the flow in Fig. 1. It extracts a baseline plan and encodes it **(2)** using WL-48 and TOK-128 descriptors. An IGMC prior, trained offline on query–hint relations, is approximated via a lightweight ridge projection from the plan features **(3–4)**, avoiding a GNN at inference time. The projected prior is concatenated with the plan descriptors to form x'_q, which feeds the Fusion–Rules MLP scorer **(5)** to rank candidate rewrites mined offline. The top admissible rule is resolved into executable `pg_hint_plan` hints **(6)**. The query is executed on the database server **(7)**. The following sections elaborate on each component.

Notation. Let Q be the number of queries and H the number of traditional hints (knobs). Let $\Omega \subseteq \{1,\dots,Q\} \times \{1,\dots,H\}$ be the set of observed query, hint pairs.

Plan encoders

The encoder module derives query representations from PostgreSQL plans using two complementary features:

WL-48 Sketch: a 48-dimensional Weisfeiler–Lehman graph sketch over the plan's directed acyclic graph (DAG), capturing structural neighborhoods of plan operators (Shervashidze et al. 2011).

TOK-128 bag: a 128-dimensional hashed bag-of-tokens that summarizes key plan components (Akioyamen, Yi, and Marcus 2024).

Both embeddings are concatenated into $x_q \in \mathbb{R}^{176}$, forming the embeddings used in inference. Because encoders operate on planner metadata rather than raw tuples, Optimus avoids data egress and is compatible with restricted environments (addresses D2).

Stage A — Per-hint predictor (Fusion–Residual)

Training data is drawn from the DSB workload (Ding et al. 2021) with measured latencies under traditional hints—standard knobs rather than novel rewrites. A sparse observations matrix $Y \in \mathbb{R}^{Q\times H}$ is constructed, where queries index rows and hints index columns. Each observed query-hint pair $(q,h) \in \Omega$ shares query features x_q and target $y_{q,h}$. x_q is standardized using training-set statistics and projected to a d-dimensional latent $u_q^{\mathrm{FR}} \in \mathbb{R}^d$. Furthermore, each hint h has embedding $e_h \in \mathbb{R}^d$ and bias $b_h \in \mathbb{R}$. Using this information, a bilinear score models interactions, followed by a residual MLP that predicts a correction $\Delta_{q,h}$ and a log-variance term $\log \sigma^2_{q,h}$:

$$d_{q,h} = \langle u_q^{\mathrm{FR}}, e_h \rangle + b_h, \qquad \hat{y}_{q,h} = d_{q,h} + \Delta_{q,h}. \quad (1)$$

Training. A convex combination of Mean Squared Error on the bilinear score $d_{q,h}$ and Gaussian negative log-likelihood (NLL) on the full prediction $\hat{y}_{q,h}$ is optimized (Kendall and Gal 2017):

$$\mathcal{L} = (1-\alpha)\,\tfrac{1}{|\Omega|}\sum_{(q,h)\in\Omega}\big(y_{q,h}-d_{q,h}\big)^2 + \alpha\,\tfrac{1}{|\Omega|}\sum_{(q,h)\in\Omega}\left[\tfrac{1}{2}\log\sigma^2_{q,h} + \frac{\big(y_{q,h}-\hat{y}_{q,h}\big)^2}{2\sigma^2_{q,h}}\right]. \quad (2)$$

Here $y_{q,h}$ are observed *log*-latencies, $\alpha \in [0,1]$ trades off the two terms, and $\sigma_{q,h} > 0$ is the implied standard deviation ($\log \sigma^2_{q,h}$ is parameterized for stability). After training, parameters are frozen. Given a new query q, the output is the per-hint vector $\hat{\mathbf{y}}^{\text{hint}}_q = [\hat{y}_{q,1}, \dots, \hat{y}_{q,H}] \in \mathbb{R}^H$, capturing query specific sensitivity to standard knobs.

Stage B — Mining and representing rewrite rules

Query rewrites are obtained offline from heuristic miners and an LLM (Gemma 3n) (Gemma Team 2025). These processes emit novel rules that introduce changes not captured by existing parameter tuning. Here, *novel rules* refer to rewrites that target a particular plan subtree rather than applying a global toggle (addresses D1). To mine such rules, a sweep across the JOB workload (Leis et al. 2015) yields plan JSONs that the pipeline parses to extract candidate rules. Mining rules offline localizes risk: rewrites are auditable artifacts, making them easier to review (addresses D2).

Heuristic miners. Several heuristics, based on common wisdom within the database community, capture recurring performance pains: (i) remove a redundant *Sort* feeding a *Merge Join*; (ii) replace very large *HashAggregate* with a *GroupAggregate*, and more. These patterns are matched on baseline plans with explicit guards.

These heuristics are human-interpretable and map to concrete plan edits, so operators can whitelist/blacklist individual rules and attach guardrails (e.g. table/size thresholds), providing operator control (addresses D2). This makes changes debuggable in production and compatible with internal change-management policies. Furthermore, these guards ensure that transformations generally apply under explicit preconditions, which simplifies production governance.

LLM augmentation. A summary of the hot plan regions is passed to an instruction-tuned LLM, which proposes canonical actions. Feasibility checks ensure only implementable candidates enter the feasible rule set, ensuring the novelty of the rewrite rules.

To enable downstream selection, each rule t is associated with a descriptor g_t and an embedding v_t used by the scorer. Online, a candidate rewrite t is admissible for query q only if it (i) satisfies guards/bindings and (ii) compiles to valid `pg_hint_plan` hints (addresses D2, D3). The set of all such admissible t defines the feasible set $\mathcal{T}(q)$.

Stage C — IGMC prior and *Fusion–Rules* MLP scorer

IGMC prior (hint level). Trained on the DSB workload, an IGMC model predicts log–latencies for traditional query-hint pairs for each query q. It then yields a hint-ordered vector

$$\mathbf{D}^{\text{IGMC}}_q = \big[\hat{y}^{\,\text{IGMC}}_{q,1}, \dots, \hat{y}^{\,\text{IGMC}}_{q,H_{\text{IGMC}}}\big].$$

To expose this signal at inference without running the computationally heavy GNN, a simple ridge projection is fit from the baseline plan descriptors: $x_q = [\text{WL}_{48};\, \text{TOK}_{128}]$ to $\mathbf{D}^{\text{IGMC}}_q$. At inference time, the IGMC block is reconstructed as $\hat{\mathbf{D}}_q = W x_q + b$ and concatenated with x_q to form $x'_q = [x_q; \hat{\mathbf{D}}_q]$. Exposing the prior via a ridge projection replaces an online GNN with a single matrix–vector multiply, keeping inference overhead negligible under query workloads.

Fusion–Rules MLP (trained on deltas). For each query–rule pair (q,t), the supervised target is the log–delta against the baseline planner, $y_{q,t} = \log \text{ms}(q,t) - \log \text{ms}_{\text{base}}(q)$, so $y_{q,t}<0$ indicates a speedup.

Then, queries and rules are embedded into $u_q, v_t \in \mathbb{R}^r$ with compatibility $d_{q,t} = u_q^\top v_t$. The following feature bundle is formed $z_{q,t} = \big[d_{q,t},\, u_q,\, v_t,\, g_t,\, x'_q\big]$. Subsequently, an MLP f_ϕ maps $z_{q,t}$ to the predicted delta $\hat{y}_{q,t}$. Including (g_t, v_t) in $z_{q,t}$ preserves interpretability since the served recommendation can always be traced to a specific rule/feature slice (addresses D2).

Training minimizes a regression objective with a within-query ranking term:

$$\min_{\phi}\ \underbrace{\tfrac{1}{|I|}\sum_{(q,t)\in I}\rho_\delta(y_{q,t}-\hat{y}_{q,t})}_{\text{Huber loss on log–delta}} + \underbrace{\lambda_{\text{rank}}\sum_{(t^+,t^-)}\max\Big\{0,\ -\big(\widehat{\ell}_{q,t^-}-\widehat{\ell}_{q,t^+}\big)\Big\}}_{\text{pairwise ranking within each query}}, \quad (3)$$

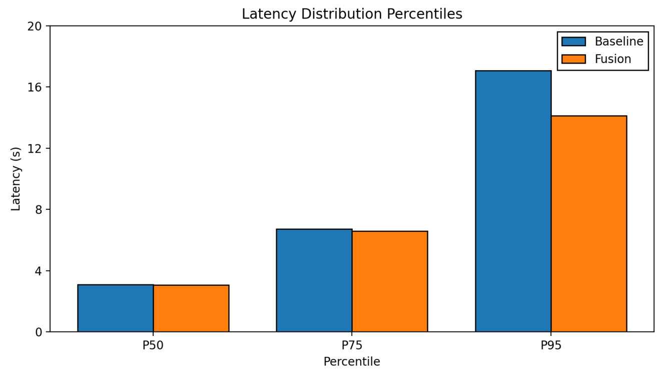
where ρ_δ is the Huber loss and $\widehat{\ell}_{q,t} = \hat{y}_{q,t} + \log \text{ms}_{\text{base}}(q)$ is the induced absolute prediction.

Inference. Absolute latency is recovered by composing the predicted delta with the measured baseline,

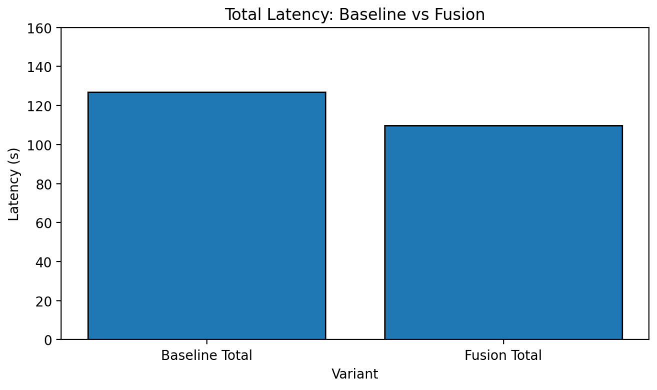
$$\widehat{\log \text{ms}}(q,t) \;=\; \log \text{ms}_{\text{base}}(q) \;+\; \hat{y}_{q,t},$$

and feasible rules from $\mathcal{T}(q)$ are ranked by the implied speedup $\widehat{\Delta}(q,t) = \exp\!\big(\log \text{ms}_{\text{base}}(q) - \widehat{\log \text{ms}}(q,t)\big)$. The top candidate is executed.

Thus, the system requires no engine modifications and uses guard-checked, compilable rules (addresses D2). Because the inference path requires no database engine modifications, the system can be readily deployed in production environments (addresses D3). Each stage of the pipeline contributes to this deployability: the encoders avoid data egress (addresses D2), the Fusion–Residual and IGMC models keep latency negligible, and the mining stage produces auditable rules that can be controlled (addresses D1, D2). Together, these choices make Optimus transparent to operators and compatible with existing PostgreSQL installations (addresses D3).



(a) Latency distributions across percentiles.



(b) Total latency by variant.

Figure 2: Default refers to vanilla PostgreSQL timings while Fusion refers to OPTIMUS.

Preliminary Results

The following research questions are addressed through the preliminary experiments:

1. Can Optimus truly discover impactful query rewrites?
2. Which latency percentile does Optimus impact the most? Does this impact deployability?
3. Is Optimus actually deployable—do the observed gains arise without engine modifications?

Setup. The Extended JOB Benchmark (Marcus et al. 2019) is evaluated over the IMDb database (Leis et al. 2015). It is executed on PostgreSQL 17 (Pos 2025) with `pg_hint_plan` (pgh 2025) enabled. The same server configuration is used across all runs, and warm-cache steady state is established for all queries. For each test query, the vanilla PostgreSQL baseline is measured and the top-ranked Optimus rewrite is executed. For fairness, one warm-up is followed by two measured repetitions per variant. The median is reported.

Overall Latency Reductions. Figure 2(a) shows consistent shifts across percentiles—P50 improves by $1.01\times$, P75 by $1.02\times$, and P95 by $1.21\times$—and Figure 2(b) reports a geometric-mean speedup of $\approx 1.16\times$. These gains are obtained *solely* from the novel generated query rewrites. The results indicate that novel rules alone can yield meaningful reductions to query runtime. Notably, the improvement pattern favors long-tail queries, which can dominate overall workload latency in production systems. This skew suggests that the rule-based rewrites of Optimus are effective in addressing complex plans.

Deployability in Real Systems. A key feature of Optimus is that its performance gains are achieved under conditions that reflect production constraints. It requires no engine modifications and applies only guard-checked rewrites that target specific plan subtrees. Because the entire inference path runs at the SQL layer, Optimus can be integrated into existing PostgreSQL deployments as an extension. The model's architecture ensures inference latency is negligible relative to query execution time, maintaining throughput.

Therefore, Optimus addresses the three deployability gaps while adhering to general deployability constraints. However, it is vital to note that Optimus has not yet been deployed in live production systems, meaning its robustness under concurrent, mixed-query workloads remains to be validated in future work.

Comparison with State-of-the-Art. Although recent learned optimizers often report larger end-to-end gains (Zhang et al. 2024; Yi et al. 2024), it is important to note that Optimus achieves $1.16\times$ end-to-end improvements *without* additional indexes or prior training specific to the tested benchmark. The gains stem purely from structural rewrites. This highlights that there exists an undiscovered set of query rewrites that can yield reliable reductions in latency. Furthermore, unlike many existing systems that modify the query optimizer's internals, Optimus operates as a drop-in layer, which demonstrates that learned reasoning over verified rewrites can produce tangible benefits.

Conclusions and Future Work

Optimus mines guard-checked, executable plan-rewrite rules and ranks them using a fusion of per-hint responses and an IGMC prior. The empirical results indicate that *structural rewrites alone* can reduce total query latency, with the largest benefits concentrated in the high-latency tail. Since Optimus requires no engine modifications and uses a lightweight serve-time path, it fits the constraints of deployable AI. Some exciting future directions for this work include:

- *Machine Learning for Rule Discovery.* A learned model that can mine rules by observing database telemetry yields the potential to highly optimize queries. LLM-based agents for query optimization are highly promising as well, since they can synthesize new transformations.
- *Hybrid tuning.* Combining mined rules and existing configurations holds potential to highly optimize queries. Such hybrid approaches could balance the interpretability of rule-based rewrites with parametric tuning.

References

2025. PostgreSQL Database. http://www.postgresql.org/.

2025. `pg_hint_plan` Documentation (scan/join/order/-parallel/GUC hints). https://pg-hint-plan.readthedocs.io/. Accessed Aug. 26, 2025.

Akioyamen, P.; Yi, Z.; and Marcus, R. 2024. The Unreasonable Effectiveness of LLMs for Query Optimization. To appear, ML for Systems @ NeurIPS 2024. arXiv:2411.02862.

Aziz, M. A.; Shoeb, A. A. M.; Shoeb, A. K. M.; and Hossain, S. 2024. COOOL: A Learning-to-Rank Approach for Query Optimizer and Cost Models. ArXiv:2409.08233.

Ding, B.; Chaudhuri, S.; Gehrke, J.; and Narasayya, V. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proceedings of the VLDB Endowment*, 14(13): 3376–3388.

Gemma Team. 2025. Gemma 3n. https://ai.google.dev/gemma/docs/gemma-3n.

Kendall, A.; and Gal, Y. 2017. What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision? In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, 5574–5584.

Khan, A.; Mahbub, S. M.; Bhatnagar, A.; Atasu, K.; Ştefan, T.-I.; Liang, H.; Kotselidis, C.; and Papadopoulos, N. 2023. fastgres: A High-Performance PostgreSQL for Fast Analytics. *Proc. VLDB Endow.*, 16(11): 2482–2495.

Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P.; Kemper, A.; and Neumann, T. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.*, 9(3): 204–215.

Lyndall, A.; Carrara, N.; Li, Q.; Marcus, R.; and Demir, İ. 2025. Balsa: Learning Query Optimization via Progressive Bayesian Search. ArXiv:2501.12905.

Marcus, R.; Negi, P.; Mao, H.; Tatbul, N.; Alizadeh, M.; and Kraska, T. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, 1275–1288. ACM.

Marcus, R.; Negi, P.; Mao, H.; Zhang, C.; Alizadeh, M.; Kraska, T.; Papaemmanouil, O.; and Tatbul, N. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.*, 12(11): 1705–1718.

Pavlo, A.; Angulo, G.; Arulraj, J.; Lin, H.; Lin, J.; Ma, L.; Menon, P.; Mowry, T. C.; Perron, M.; Quah, I.; Santurkar, S.; Tomasic, A.; Toor, S.; Aken, D. V.; Wang, Z.; Wu, Y.; Xian, R.; and Zhang, T. 2017. Self-Driving Database Management Systems. In *CIDR 2017: 8th Biennial Conference on Innovative Data Systems Research*.

Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; and Price, T. G. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 23–34. ACM.

Shervashidze, N.; Schweitzer, P.; van Leeuwen, E. J.; Mehlhorn, K.; and Borgwardt, K. M. 2011. Weisfeiler–Lehman Graph Kernels. *Journal of Machine Learning Research*, 12: 2539–2561.

Trummer, I.; Wang, J.; Maram, D.; Moseley, S.; Jo, S.; and Antonakakis, J. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, 1153–1170. ACM.

Wilson, J. D.; McLellan, A. D. R.; Supera, S.; and Taccari, L. 2024. GenJoin: Enabling Massive Joins with Probabilistic Recursion. ArXiv:2411.04525.

Yi, Z.; Tian, Y.; Ives, Z. G.; and Marcus, R. 2024. LimeQO: Low-Rank Approximation for Learned Query Optimization. *arXiv preprint arXiv:2407.18223*.

Zhang, B.; Aken, D. V.; Wang, J.; Dai, T.; Jiang, S.; Lao, J.; Sheng, S.; Pavlo, A.; and Gordon, G. J. 2018. A Demonstration of the OtterTune Automatic Database Management System Tuning Service. *Proc. VLDB Endow.*, 11(12): 1910–1913.

Zhang, J.; Liu, Y.; Zhou, K.; Li, G.; Xiao, Z.; Cheng, B.; Xing, J.; Wang, Y.; Cheng, T.; Liu, L.; Ran, M.; and Li, Z. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*, 415–432. ACM.

Zhang, M.; and Chen, Y. 2020. Inductive Matrix Completion Based on Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*.

Zhang, W.; Lim, W. S.; Butrovich, M.; and Pavlo, A. 2024. The Holon Approach for Simultaneously Tuning Multiple Components in a Self-Driving Database Management System with Machine Learning via Synthesized Proto-Actions. *Proc. VLDB Endow.*, 17(11): 3373–3387.