Database Gyms:
Towards Autonomous Database Tuning

Wan Shen Lim

CMU-CS-26-102

February 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andrew Pavlo, Chair
Jignesh Patel
David Andersen
Lin Ma (University of Michigan)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Copyright © 2026 Wan Shen Lim

This research was sponsored by the National Science Foundation under award number 1846158 and 2404373, Snowflake, Alfred P. Sloan Foundation under award number FG201810204, Google, SingleStore, CMU Parallel Data Laboratory, and VMware. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: database management systems, autonomous database tuning, database gyms, tool-based database tuning, machine learning for systems, training data

To my family, my reason for life.

Abstract

Database management systems (DBMSs) are the foundation of modern data-intensive applications. But as more features are developed to support new workloads, they become increasingly complex and difficult to configure. Thus, researchers have invested decades of effort into autonomous DBMS configuration. Recent advances in machine learning (ML) have produced tools that outperform unassisted experts in real-world deployments. However, these tools are advisory and require human expertise for their deployment into database tuning pipelines. Using these tools involves a multi-step process where a human operator (1) determines an optimization objective, (2) selects a suitable tool to improve the objective, (3) sets up and configures the DBMS to run a particular workload, (4) runs the workload to collect telemetry, (5) uses the collected telemetry to calibrate the tool, and (6) operates the tool to obtain recommendations, which the operator must then review and apply. Because of the ad-hoc nature of these pipelines, they require significant human effort to set up, extend, and deploy. Moreover, these tools are difficult to compose and swap. Even if two tools are designed for the same task, differences in their interfaces preclude their interchangeability. Given these challenges, despite the demonstrated ability of database tuning tools to improve performance and lower costs, adoption remains limited due to the substantial human expertise required for their operation.

This dissertation presents the *database gym*, an integrated framework that systematizes and automates the DBMS configuration pipeline. Unlike prior research that focused on improving tool effectiveness with ML, the gym aims to address challenges in the deployment and operation of existing tools by providing a set of reusable, interoperable, and interchangeable components that simplify their development and integration. The gym is designed to address the observation that the bottleneck in the database tuning process has shifted from developing better algorithms for tools to acquiring the training data needed to operate them, elevating training data's importance.

In this dissertation, we demonstrate how the gym's architecture accelerates and adapts tool-based database tuning pipelines through the systematic generation and utilization of their training data, enabling the augmentation and orchestration of tools with end-to-end knowledge. The gym leverages its complete control over the tuning process to enable holistic optimizations that span the entire pipeline. For instance, it reduces step-level overhead by skipping redundant computation during telemetry collection, thus reducing the tuning pipeline's latency. It also eliminates pipeline-level repetition by reusing past experience to adapt a tool's calibrated models to new environments, accounting for semantic differences across software versions and hardware environments.

The techniques in this dissertation show how a training-data-centric approach to tool-based database tuning accelerates tool operation and adapts tool assumptions to new environments, with the database gym architecture providing a framework that simplifies tool development while enabling novel optimizations.

Acknowledgments

When I was an undergrad, I lucked into working with my advisor, Andrew (Andy) Pavlo. I had no research experience when I first met him – or database internals knowledge, for that matter – and he taught me everything I know: identifying problems, scoping out solutions, writing, handling setbacks, and more. In all of our years working together, he constantly provided support and encouragement. I think it is rare that a PhD student can say that they walked out of their weekly meetings happier and more confident! Being advised by Andy is a privilege that I cherish. His passion for research and enthusiasm for life are infectious. Knowing him has inspired me to hedge less and live more.

I am also fortunate to have a great thesis committee: Lin Ma, Jignesh Patel, and David (Dave) Andersen. Lin set the stage for our self-driving database research direction during his time at CMU. He is the reason that the Boot paper exists; when early micro-acceleration results were disappointing, he guided me towards the idea of macro-acceleration. He was a generous senior at CMU and remains a great advisor at Michigan. Jignesh provided precise technical insights that grounded my work. Despite his packed calendar, he always found the time to help out. When Andy was hospitalized, he stepped up as my research advisor and put out my TA course fires. Dave gave me targeted feedback on my thesis proposal and offered valuable perspectives from the world beyond the database community. Their collective support made this thesis possible.

The CMU Database Group has been my home for almost a decade. I especially want to thank two people who actively welcomed me into the group when I was starting out: Tianyu Li and Matthew Butrovich. Tianyu encouraged me to hang out more in the lab and made me feel that I belonged. Matt was my senior and coworker for many years. He created a friendly and collaborative lab culture, fostering a sense of community and readily giving us his advice and support. He is also skilled at proactively writing notes that we juniors would find useful later. Perhaps most importantly, he takes care of my secret coauthors, Scout and Leland, whose paws left their imprints on all of my papers.

The numerous postdocs and PhD students of the database group brightened up my life considerably. William Zhang and I were in the group together since our undergraduate days; our regular chats about research, market news, and offbeat content gave me something to look forward to daily. Hyoungjoo Kim, my internship housemate: thank you for going on adventures together, and get a cat! I also appreciate the many conversations and hours spent together with Samuel Arch, Christos Laspias, Drew Ripberger, Martin Prammer, and Andrew Crotty.

I met too many people from the database group over the years to enumerate here, but I want to try listing everyone with whom I've directly worked, hung out in the lab, and/or stayed in touch: Abhijith Anilkumar, Pulkit Agarwal, Utkarsh Agarwal, Rohan Aggarwal, Dhruv Arya,

Mayank Baranwal, Aditya Bhatnagar, Kyle Booker, Aditya Chopra, Arham Chopra, Te-Yen (Andy) Chou, Jackie Dong, Kyle Dotterrer, Prashanth Duvvada, Emmanuel Eppinger, Jordi Gonzalez, Preetansh Goyal, Garrison Hess, Gautam Jain, Ying Jiang, Katrina Jiao, Lichen Jin, Kunal Jobanputra, Weichen Ke, Mihir Khare, Abigale Kim, Joseph Koshakow, Marcel Kost, Arvind Sai Krishnan, Hugo Latendresse, Jiaying Li, Yuchen Liang, Joyce Liao, Sheng Yong Lim, Yingjie Ling, Xiaoxuan (Lily) Liu, Saransh Malik, Prashanth Menon, Gustavo Angulo Mezerhane, Yashwanth Nannapaneni, Tanuj Nayak, Amadou Latyr Ngom, Tim Lee, Bobby Norwood, Benjamin Owad, Tianlei Pan, Ritu Pathak, Deepayan Patra, Wen Xuan Qiu, Karthik Ramanathan, John Rollinson, Erik Sargent, Alexis Schlomer, Shubham Shastri, Yangjun Sheng, Kushagra Singh, Zhaozhe Song, Sarvesh Tandon, Connor Tsui, Katia Villevald, Patrick Wang, Ruiqi Wang, Wuwen Wang, Xiaohui Wang, Arvin Wu, Chen (Ricky) Xu, Peijing (Mike) Xu, Alex (Chi) Zhang, Ling Zhang, Nevin Zheng, Ricky Zhou. If I have inadvertently missed a name, please forgive me and send me a message; let's catch up! I also got to meet many of my predecessors out there in the world, and it was great talking to them: Dana Van Aken, Joy Arulraj, Matt Perron, Bohan Zhang, Huanchen Zhang. I really enjoyed my time in our group.

In addition to our database group, I was also lucky to be part of the Parallel Data Lab, a steady source of weekly research ideas and fruit intake alike. I am particularly grateful to Greg Ganger, Phil Gibbons, Jason Boles, Chad Dougherty, Karen Lindenfelser, and Joan Digney for their support. I had good conversations with many students, including: Sanjith Athlur, Nirav Atre, Ankush Jain, Timothy Kim, Sara McAllister, Deepanjali Mishra, Kaiyang Zhao.

There are also many good memories from my brief encounters with industry. Johannes Gehrke gave me helpful insights and suggestions for the database gym direction. As a new PhD student, I admired the papers coming out of Microsoft Research's Data Systems Group; I met many authors while interning there and it was an amazing experience. I am especially grateful to my mentors, Surajit Chaudhuri, Vivek Narasayya, Anshuman Dutt, and Arnd Christian König, for explaining to me their thought processes and showing me how they did research. My fellow DSG intern lunchers were also great people to hang out and discuss random topics with. I especially thank Jiashen Cao for being a good friend that I can talk to freely and a fellow lunch organizer. Those summers at Microsoft Research shaped my vision of an ideal workplace. Prior to Microsoft Research, my undergraduate self learned a lot from the mentorship of Justin Levandoski, Eric Boutin, and Niket Goel at Amazon Aurora. I am also grateful to Ippokratis Pandis for his friendly advice when he visited CMU, and to Vaibhav Arora for being someone that I always looked forward to seeing again at each PDL retreat. The database community is full of people who are passionate about their work and generous with their knowledge.

I am grateful to the many CMU faculty, staff, and friends who have greatly enriched my life, either through TAing for them, being on committees together, or chatting with them: Anil Ada, Guy Blelloch, Iliano Cervesato, Tom Cortina, Dave Eckhardt, Michael Erdmann, Charlie Garrod, Robert Harper, Mark Stehlik; Deb Cavlovich, Catherine Copetas, Amanda Hornick, Jenn Landefeld, Angy Malloy, Matthew Stewart, Charlotte Yano; Anup Agarwal, Dorian Chan, Myra Dotzel, Korinna Fragkia, Theo Gregersen, Keerthana Gurushankar, Aditi Kabra, Misha Khodak, Gabriele Oliaro, Eliot Solomon, Yue Yao.

I thank my government for giving me the opportunity to pursue higher education, especially the people at Bahagian Biasiswa and our Embassy in DC. I also thank EducationUSA, Mayra Robles, and Kenny Liew for helping me navigate the process as a first-generation college student.

Beyond my research and CMU roles that culminated in this thesis, I also want to acknowledge the people who have supported me in life.

My parents, Lim Pon Han and Tan Joo Lee, provided me with a loving home and set me up for opportunities that they never had. They tried their best to give their children everything that they could. Their unconditional support is the reason my siblings and I have been able to thrive. My siblings, Lim Chien Hau and Lim Chien Yee, held down the fort at home while I was studying abroad. They have supported me in countless ways, and I would almost certainly have had to quit my PhD if not for them. My brothers-in-law, Chen Weijun and Julian Tan Wee Boon, are the best in-laws I could ask for. My family is my life's greatest blessing.

My undergraduate advisor, Mark Stehlik, continued to be a source of wisdom and guidance throughout my PhD. He taught me to think more flexibly. I aspire to have as much impact on education as he has, while remaining as human and approachable as he is.

Sie Siok Hui was my math teacher in high school, but also so much more. Her belief in me kept me out of trouble and granted me the perseverance to live a clean life. I am lucky to have had so many high school teachers who supported my programming interests and dealt with my antics, especially: Phylliscia Chew, Li Da, Ng Chee Loong, Lim Teck Choow, Lee Chan Lye. Without their guidance, my cybersecurity-filled days would have led me down a different path.

I am also blessed with good friends. To Grace Yu and Pranav Kumar, thank you both especially for supporting me in the moment and the months after my father passed away. And to Christopher Shan, thank you for bringing me along on many adventures. My CMU friends made my days that much brighter, especially: Eric Chen, Gwyneth Chen, Yiyuan Chen, Ian Chiu, Valerie Choung, Ariel Davis, Weihang Fan, Joy Alice Gu, Anne He, Yixin He, Vivian Huang, Thejaswi Kadur, Jessica Annie Lee, Jacob Neumann, Monica Pardeshi, Shivani Prasad, Matias Scharager, Brian Scheuermann, David Sun, Lilia Tang, Ethernet Wang, Cameron Wong, Grant Wu, Patrick Xia, David Xie, Neil Xu, Alexander Yu, Lexin Yuan, Nancy Zhang.

And to my friends from my time in Singapore, especially: Michelle Angelica Anthony, Bohao Guo, Jee Dong Jun, James Chua Wee Kian, Chua Xin Pin, Dawn Oh. We have spent thousands of hours together over the years, and I would gladly spend thousands more. There are many friends from those days with whom I have been worse at keeping in touch, but whose influence has been significant nonetheless: Qingzhuo Aw, Jonathan Chuah, Pei Fang Dee, Franciskus Xaverius Erick Estrada, Ivan Ho, Chua Eu Jing, Eliot Lim, Jireh Ong, Ze Xuan Ong, Muhammad Irham Rasyidi, Andrew Tantri, Daniel Teo. I remember those days fondly.

I expect approximately none of you to read this dissertation or even know that you're mentioned in it, but regardless: thank you all for adding meaning to my life.

Contents

Abstract v

Acknowledgments vii

1 Introduction 1
1.1 Redundant Computation . . . 2
1.2 Discarded Experience . . . 2
1.3 Summary of Contributions . . . 3

2 Background 5
2.1 Self-Driving DBMSs . . . 5
2.2 DBMS Optimization Tools . . . 6
2.3 Tool-Based Database Tuning Pipelines . . . 8

3 Database Gym 11
3.1 Background: Simulation Targets . . . 12
3.1.1 Workload . . . 12
3.1.2 State . . . 13
3.1.3 Actions . . . 16
3.2 Architecture . . . 16
3.2.1 Synthesizer . . . 18
3.2.2 Trainer . . . 19
3.2.3 Planner . . . 19
3.2.4 Decider . . . 20
3.3 Deployment . . . 20
3.4 Initial Implementation and Discussion . . . 21
3.5 Conclusion . . . 23

4 Accelerating Training Data Generation 25
4.1 Background and Motivation . . . 26
4.1.1 Database Gym . . . 26
4.1.2 The Cost of Training Data Generation . . . 27
4.1.3 Exploiting Workload Repetition . . . 27
4.2 Overview . . . 28

4.2.1 Macro-Accelerator (MA) . . . 30
4.2.2 Micro-Accelerator (μA) . . . 30
4.3 Macro-Accelerator (MA) . . . 31
4.3.1 Identifying Similar Queries . . . 32
4.3.2 Adapting to Query Variability . . . 32
4.4 Micro-Accelerator (μA) . . . 33
4.4.1 Stopping Repetitive Operators . . . 33
4.4.2 Sampling for Output Reduction . . . 34
4.5 Engineering . . . 35
4.6 Evaluation . . . 36
4.6.1 Workloads . . . 36
4.6.2 Speed-up vs. Accuracy Measurements . . . 38
4.6.3 Reducing Aborted Queries . . . 43
4.6.4 MA: Different Encoding Strategies . . . 45
4.6.5 MA: Executing Fewer Queries . . . 46
4.6.6 μG: Stopping Operators Early . . . 49
4.6.7 μG: Accelerating Short Workloads . . . 54
4.6.8 Output Sampling Analysis . . . 55
4.7 Conclusion . . . 57

5 Generalizing Training Data Across Heterogeneous Deployments **59**
5.1 Background . . . 60
5.1.1 Upgrading . . . 61
5.1.2 Mitigating Upgrade Issues . . . 66
5.1.3 DBMS Behavior Models . . . 66
5.2 Tool Architecture . . . 67
5.2.1 Limitations of Existing Models . . . 68
5.2.2 Key Insight: Isolate System Characteristics . . . 69
5.3 Implementation . . . 70
5.3.1 Fine-Tuning Behavior Models . . . 70
5.3.2 Composable Fine-Tuning . . . 71
5.3.3 Deciding on Adapters . . . 72
5.4 Case Study: Transformer-Based Models . . . 72
5.5 Evaluation . . . 73
5.5.1 Workload and Experiment Configuration . . . 74
5.5.2 Workload-Level Absolute Prediction . . . 76
5.5.3 Query-Level Relative Predictions . . . 84
5.5.4 Minimizing Upgrade Regret . . . 88
5.6 Takeaways . . . 91
5.7 Conclusion . . . 93

6 Related Work 95
6.1 Autonomous DBMSs . . . 95
6.2 Training Data Generation . . . 96
6.3 Training Data Generalization . . . 97

7 Future Work 99

8 Conclusion 101

Bibliography 103

List of Figures

2.1 **Tool-Based Pipeline** – A pipeline that uses tools to improve DBMS configurations. 8

3.1 **Workload Effects on Modeling Error** – We train two models of QPPNet to predict query elapsed time on PostgreSQL v14 with TPC-H (scale factor 10). The All model enables hash joins for all 22 queries. The First10 model enables hash joins for only the first 10 queries. We leave out bars where both models are below 0.125 relative error for readability. 13
3.2 **Latency vs. Storage Trade-off** – As fill factor decreases, the latency of the NewOrder transaction decreases, but database size increases. 14
3.3 **NewOrder Latency Distributions** – As the fill factor decreases, the latency approaches a Gaussian distribution. 15
3.4 **Database Gym Architecture** – An overview of the database gym's internals. The t subscript indicates how the gym's components produce new data through rounds of iterations. 17
3.5 **Systematizing Pipelines** – Mapping tool-based database tuning pipelines to the database gym's components. 18
3.6 **Replica Resource Availability** – We run SmallBank (scale factor 100, 2000 terminals, 30 min) on PostgreSQL with one primary and one replica. Each datapoint averages across the last 10 readings to reduce noise. 21
3.7 **OpenAI Gym Example** – Code that learns to balance a pole on a moving cart. . 21
3.8 **Database Gym Prototype** – Code that learns to tune a DBMS. 22

4.1 **Architecture** – An overview of Boot's internal components and execution flow. The MA component decides whether to execute a query, and the μA component accelerates the execution of a specific query. 29
4.2 **MA Architecture** – Overview of Figure 4.1's MA. 31
4.3 **Query Plan Behavior** – Distribution of TPC-H `Q9`'s plans and run-time across 1000 invocations (without Boot, SF 100, 1000 seeds). 32
4.4 **Gate** – The Gate's architecture for Listing 4.1's query plan. The orange node is a hash join and the gray nodes are an index scan feeding into a nested loop join. The table shows the effect of introducing the Gate. 34
4.5 **Sampler** – The Sampler's architecture for Listing 4.1's query plan. The orange node is a hash join and the gray nodes are sequential scans. The table shows the effect of introducing the Sampler in addition to the Gate. 35

4.6 **Query Run-time Distribution** – Breakdown of elapsed time for each workload (without any acceleration). 37
4.7 **Collection Time** – The time to generate training data with different modules of Boot active (lower is better). 39
4.8 **Absolute Error** – The absolute error of models that are trained on the individual datasets (lower is better). The red circle shows sample mean and the whiskers extend to 1.5 interquartile range. 41
4.9 **Error Distribution** – The distribution of model error for TPC-H at SF 100, DSB at SF 10, and JOB (closer to 0 is better). For example, an error of -20 s means the model predicted 20 seconds under the actual value. 44
4.10 **Wasted Collection Time** – The percentage of time wasted executing queries that the DBMS aborts after 5 min. We omit TPC-H (SF 10) and JOB as they do not time out. 45
4.11 **Exponential Speedup (Completion Rate)** – Workload completion rate under MA, excluding timeouts. 47
4.12 **Exponential Speedup (Run-time)** – Run-time distributions under MA, excluding timeouts. 48
4.13 **Operator Time Breakdown** – The time spent in each operator (lower is better), excluding queries that timed out in Orig. We show the top 10 operators that Orig executes. 49
4.14 **Speedup Analysis (Absolute)** – Each operator's absolute speedup because of µG in Figure 4.13 (higher is better). 50
4.15 **Speedup Analysis (Relative)** – Each operator's relative speedup because of µG in Figure 4.13 (higher is better). 51
4.16 **Tuple Reduction (Absolute)** – Each operator's absolute reduction in tuples processed because of µG in Figure 4.13 (higher is better). 52
4.17 **Tuple Reduction (Relative)** – Each operator's relative reduction in tuples processed because of µG in Figure 4.13 (higher is better). 53
4.18 **Batching Sensitivity** – Collection time and MAE as we vary µG's hyperparameters on TPC-H (SF 10), 100 queries. 54
4.19 **Sampling Baseline** – Collection time and absolute error with `TABLESAMPLE` and Boot for TPC-H (SF 100). The red circle shows sample mean. The whiskers extend to 1.5 IQR. 55
4.20 **Sampling Rate** – The collection time and MAE when MMA is enabled and sequential scans are sampled at different rates. We exclude queries that timed out for fairness (affecting 4.55% of TPC-H). 56

5.1 **Software Upgrades** – TPC-H performance across consecutive PostgreSQL versions. The colored bar labels are relative to the previous version (green is speedup, red is slowdown). 61
5.2 **Software-Induced Performance Variations** – The performance of TPC-H queries across different PostgreSQL versions on a server-class machine. A `C` above a bar indicates that the query plan changed from the previous version. 62
5.3 **Hardware Choices** – AWS EC2 instance categories and growth. 63

5.4 **Hardware-Induced Performance Variations** – The performance of TPC-H queries across different hardware environments on pg17. A C above a bar indicates that the query plan changed from the smaller hardware environment. 63
5.5 **Hardware Upgrades** – JOB speedup on pg17 when upgrading from Medium to Large hardware, grouped by whether the query plan changed. The red dot is sample mean. Whiskers mark p5 and p95 percentiles. 64
5.6 **Upgrading Hardware for JOB q13b** – The impact of the same hardware upgrade on performance across PostgreSQL versions. 65
5.7 **Possible Combinations** – The number of candidate configurations across major PostgreSQL versions and EC2 instance types. 65
5.8 **Upgrade Tool Architecture** – $^{\text{D}}_{\text{R}}$·DB generates its recommendations by creating models to simulate a workload on different environments. 68
5.9 **Composing Adapters** – Techniques that combine adapter effects. 71
5.10 **Version $\times$ Hardware Runtimes** – The total execution time of the benchmarks across PostgreSQL version and hardware combinations. Each subplot excludes queries that do not complete in time across all versions. 75
5.11 **Version Heatmap Analysis (AE)** – The absolute error of workload run-time predictions for each version-specific model across all training data configurations, where MN refers to the model trained on pgN's data. 78
5.12 **Version Adaptation (AE)** – The shift in model absolute errors for workload run-time predictions when applying $^{\text{D}}_{\text{R}}$·DB's version adapters. The red dot indicates sample mean. The whiskers indicate p5 and p95. 79
5.13 **Hardware Heatmap Analysis (AE)** – The absolute error of workload run-time predictions for each hardware-specific model across all training data configurations, with model hardware as the row label. 81
5.14 **Hardware Adaptation (AE)** – The shift in model absolute errors for workload run-time predictions with $^{\text{D}}_{\text{R}}$·DB's hardware adapters. The red dot indicates sample mean. The whiskers indicate p5 and p95. 82
5.15 **Multi-Δ Adaptation (AE)** – The absolute error distribution of different model families (see Section 5.5.1) when predicting absolute workload run-time. The dashed line separates model families by their Δ classification. 83
5.16 **Version Adaptation (QE)** – The shift in model Q-Error for query latency predictions when applying $^{\text{D}}_{\text{R}}$·DB's version adapters. The red dot indicates sample mean. The whiskers indicate p5 and p95. 85
5.17 **Hardware Adaptation (QE)** – The shift in model Q-Error for query latency predictions when applying $^{\text{D}}_{\text{R}}$·DB's hardware adapters. The red dot indicates sample mean. The whiskers indicate p5 and p95. 86
5.18 **Multi-Δ Adaptation (QE)** – The Q-Error distribution when predicting query latencies with different model families. 87
5.19 **Avoiding Version Upgrade Regret** – The CScore distribution of different model families for version upgrades. 89
5.20 **Avoiding Hardware Upgrade Regret** – The CScore distribution of different model families for hardware upgrades. 90

5.21 **Contrasting Upgrade Recommender Designs** – Pseudocode contrasting the cheapest and most computationally expensive Δ strategies, where `D` = training data, `B` = pre-trained models, `M` = the final models, `F` = foundation model, `VL` = latest version, `HL` = best hardware, `V` = version adapters, `H` = hardware adapters, and + applies adapters (see Section 5.3.2). 92

List of Tables

4.1 **Tuning Times** – Dataset collection and model training times in recent ML approaches. 27
4.2 **Repetitive Queries** – The number of queries and templates in recent workloads used in database tuning. 28
4.3 **Factor Error** – The factor error of the model predictions divided into buckets (lower is better). 42
4.4 **Factor Error** – The factor error of the model predictions divided into buckets for MA.P (lower is better). 46

Chapter 1

Introduction

Database management systems (DBMSs) rely on *actions* that modify their internal state to support the evolving demands of their application workloads. Actions fall into several broad categories [119], including physical design changes (e.g., index recommendation [64]), schema modifications (e.g., altering column types), configuration parameter adjustments (e.g., tuning knobs like buffer pool size [177]), hardware resource allocations (e.g., partitioning [158]), and individual query tuning (e.g., hint selection [132, 199]). A DBMS's *configuration* is the totality of the state resulting from all applied actions. Because these actions interact in complex ways [204] (e.g., the benefit of an index depends on the workload), it is difficult and often NP-hard [47, 102] to identify performant DBMS configurations among trillions of candidates. The process of optimizing a DBMS's configuration is known as *database tuning*.

To manage the complexity of database tuning, researchers have developed *tools* that perform specific tuning tasks. These tools reduce the problem to running the relevant tools at the appropriate times. For example, a database administrator (DBA) can use simple rule-based tools to choose configurations [40] and pick indexes [50] during a deployment's initial setup. They can then invoke more advanced tools that rely on machine learning (ML) for further optimization, such as capacity planning through workload forecasting [124], knob tuning based on workload mapping [177], and holistic optimization through similarity search [204]. Although the simplifying assumptions made by these tools preclude them from finding the most performant configurations, studies show that tool-assisted humans achieve better configurations than unassisted experts in real-world deployments [48].

Despite its potential for increasing DBMS performance and reducing deployment costs, tool-assisted database tuning is not widespread in practice. Adoption is limited not because it is ineffective, but because existing methods still require substantial human involvement and oversight. A human must still determine which tool to use, what inputs to provide the tool, where to deploy the tool, and when to apply the tool's recommendations. The need to establish bespoke tuning pipelines for each tool on every deployment imposes a burden on researchers and human operators alike. The burden of tool deployment is further exacerbated by the lack of tool interchangeability and interoperability. For example, because there is no standardized interface for tools to interact with DBMSs, it is difficult to compare and replace tools even when they are designed for the same task of predicting a SQL query's latency [126, 130]. Additionally, although some tools accept candidate actions as input [180] and other tools generate candidate actions

of their own [204], it is challenging to compose the latter's action generation with the former's action selection. Recent ML-driven tools also require a slow and expensive setup process (e.g., executing 150k SQL queries for telemetry [185]) that may take weeks to complete [130]. As such, despite their effectiveness at improving system performance, tool-based database tuning pipelines remain too slow and operationally burdensome for broader adoption.

The operational challenges that impede the adoption of tool-based database tuning pipelines stem from two key limitations in how tools currently interact with DBMSs. First, existing pipelines are slow because they rely on DBMS mechanisms that were designed for human operation, resulting in excessive redundant computation when repurposed for autonomous operation. Second, because existing tools generalize poorly across new environments, these pipelines must be re-deployed whenever the operating environment changes. These challenges reflect key distinctions between ML for systems and other ML domains. In a systems context, (1) the heterogeneity of systems and absence of generalizable training data require every operator to expensively collect their own data, and (2) system upgrades then invalidate much of that costly data, obsoleting prior efforts and necessitating pipeline re-deployment. We now describe each limitation in more detail.

1.1 Redundant Computation

Existing tuning tools interact with the DBMS through mechanisms that were originally designed for human operators rather than for autonomous operation. This architectural mismatch results in substantial redundant computation, contributing to the extensive setup time of advanced tools [120]. For example, most tools only require system-level metadata such as DBMS query execution telemetry (e.g., operator-level timing information [61]). However, because they obtain this information by instrumenting regular query execution, the DBMS expends significant resources in computing the correct query result which is then discarded by the tool.

What is needed is a dedicated "tool mode" for the DBMS that optimizes for tool interaction. This distinction is analogous to the difference between web scraping and querying a web API: both retrieve the same information, but the former incurs massive overhead in both time and computational resources. By pushing more information about downstream applications into the DBMS, the system can skip over redundant work and accelerate tool operation.

1.2 Discarded Experience

Even after a database tuning tool is set up and deployed, its performance degrades over time as its working environment changes. For example, a tool that predicts query latency must account for how software version upgrades affect the DBMS's query optimization and execution strategies. Other factors that invalidate a tool's calibrated assumptions about system behavior include changes in hardware, workload, and configuration.

However, it is difficult to determine when a tool is operating outside of its calibrated assumptions. In practice, operators are expected to monitor tool output quality for degradation to periodically re-calibrate and re-deploy their tools from scratch whenever necessary. However,

this undermines the tool's value proposition by requiring expertise and effort. Moreover, this approach discards all the expensive data collection and requires more setup work, creating a heavy burden on the operator. These issues are compounded by the need to coordinate multiple tools for better performance [205]. Hence, because tools do not generalize across environments, a tool-based database tuning pipeline has recurring costs: it requires ongoing monitoring and maintenance with a high degree of domain expertise. One solution to this issue is to develop a method for characterizing and calibrating a tool's assumptions about its environment.

1.3 Summary of Contributions

This dissertation investigates pragmatic methods for enabling autonomous DBMS administration on existing deployments by improving and accelerating tool-based database tuning pipelines, pursuing capabilities comparable to theoretically ideal approaches that require a complete redesign of the DBMS (described further in Chapter 2).

This dissertation aims to provide evidence to support the following statement:

Thesis Statement: *Systematically generating and utilizing training data in database tuning pipelines to augment and orchestrate tuning tools with end-to-end knowledge provides an extensible and effective way to accelerate tuning pipelines through reducing redundant computation and reusing past experience in new environments.*

This dissertation makes the following contributions:

Database Gym (Chapter 3)**:** We propose the architecture of the *database gym* [119], a framework that decomposes the database tuning pipeline into modular, interchangeable, and interoperable components that support orchestration without human involvement.

Reducing Redundant Computation (Chapter 4)**:** We demonstrate how the gym's orchestration and end-to-end control of the database tuning process enables software-based acceleration of tool operation [120], sharply reducing the time necessary for tool setup.

Reusing Past Experience (Chapter 5)**:** We demonstrate how the gym's awareness of its operating environment allows it to adapt tools across different software and hardware environments, limiting the need to re-train and re-deploy tools as the environment changes. We further demonstrate that these capabilities enable upgrade recommendation by providing "what-if" analysis.

Chapter 6 discusses related work. Chapter 7 suggests potential directions for future work. We conclude our findings in Chapter 8.

Chapter 2

Background

DBMSs are difficult to configure and optimize because their optimal setting varies with the workload, database contents, and run-time environment. The complexity of this dynamic problem has motivated decades of research into methods for automated DBMS configuration. Recent work has increasingly leveraged ML for database optimization, demonstrating efficacy across problem domains such as query performance prediction [126, 130, 210], query optimization [132], index recommendation [64, 74, 108], knob tuning [115, 177], and partitioning [92].

The goal that unifies these efforts is the creation of an autonomous DBMS that operates without human guidance. Such a system aims to optimize itself automatically for a given objective function (e.g., latency, throughput, cost) and constraints (e.g., cost budget) [151]. It improves this objective by deploying actions (e.g., building an index) with human-understandable explanations in response to anticipated workloads.

In this chapter, we contrast two paradigms for autonomous DBMS optimization. First, we discuss *self-driving* [151] DBMSs that are designed from first principles to operate autonomously without human intervention. Such systems represent the theoretical ideal for what is possible, but they are not applicable to existing DBMSs without significant architectural changes. Second, we examine the dominant paradigm today: *tool-based database tuning*, where human operators orchestrate specialized optimization tools to discover better system configurations. We provide an overview of existing tools and analyze how they integrate into tuning pipelines for existing DBMS deployments. Our analysis identifies the challenges that must be solved to bridge these paradigms and enable self-driving capabilities in existing DBMSs.

2.1 Self-Driving DBMSs

A self-driving [151] DBMS aims to enable completely autonomous DBMS administration that is proactive, efficient, and explainable through three core frameworks in its architectural design.

Workload Forecasting: Existing database tuning tools optimize their input workloads without accounting for how the workload changes over time, leading to sub-optimal configurations. For example, a naive policy of always optimizing for the previous week's workload is ineffective for seasonal demand spikes (e.g., holiday shopping traffic): the system will not be ready to handle

the spike as it happens, and will proceed to waste resources by optimizing for the spike after it has already passed. Thus, a self-driving DBMS aims to observe and forecast patterns in its workload so that it can optimize for the work that it must perform in the near future. It does so through a workload forecasting [124] framework that uses techniques such as clustering, time-series forecasting, and Long Short-Term Memory (LSTM) networks. This framework allows the DBMS to predict workload characteristics such as arrival patterns [124] and query contents [99].

Behavior Modeling: Existing DBMSs are limited in their ability to predict their behavior under different configurations. Although some systems can determine whether they would use a newly created index for specific queries [101, 134], they do not predict the overall run-time improvement from having that index. Additionally, most systems do not track the effects of all their configuration parameters (e.g., sizing hash joins, allocating resources to background tasks), and thus determining the performance impact of modifying those parameters requires expensive trial-and-error experimentation. A self-driving DBMS addresses these limitations through a behavior modeling framework [126] that can predict both the costs and benefits of all its possible actions. The framework's analytical models, ML models, and the design of the self-driving DBMS itself [153] combine to ensure that the effects of all actions can be modeled accurately.

Action Planning: Existing DBMSs do not autonomously modify their configuration even when they suspect that they should (e.g., PostgreSQL's `checkpoint_warning` parameter [30] controls when to output warnings asking a human operator to increase `max_wal_size`). They avoid such changes because they lack a planning process that can determine when, how, and why they should apply their recommended actions. A self-driving DBMS provides such planning capabilities [123] through receding horizon control and Monte Carlo Tree Search (MCTS), allowing it to take actions that stay within its system constraints while pursuing its long-term goals.

The tight integration of the workload forecasting, behavior modeling, and action planning frameworks creates a system design [153] that facilitates holistic and autonomous operation. Such a design represents an ideal scenario for autonomous database management. However, existing DBMSs in real-world deployments were not designed as self-driving systems and lack many of the necessary capabilities. Thus, one challenge is to approximate a self-driving DBMS's functionalities as much as possible within the constraints of existing systems (i.e., retrofitting autonomous capabilities onto systems that were designed with human operators in mind).

2.2 DBMS Optimization Tools

A pragmatic approach for approximating self-driving capabilities in existing DBMSs is to use specialized optimization tools. These tools require minimal modification to a DBMS's core architecture. Instead, they function by observing system characteristics (e.g., performance telemetry) to make their recommendations, which are then either automatically applied or manually reviewed by a human operator. However, because there is no standard interface for how tools integrate with DBMSs or how recommendations are surfaced to operators, it is difficult to com-

pose the functionality of different tools. To illustrate the breadth of optimization problems that tools can address, we describe a few commonly grouped tool categories [204, 205] below.

Knob Tuners: Modern DBMSs expose hundreds [177] of configuration parameters (i.e., *knobs*) that alter the DBMS's behavior, such as its buffer pool size and its maximum degree of parallelism. These parameters cannot be optimized in isolation because they have complex interactions with each other, making it difficult to find performant configurations. Knob tuners simplify the problem by automatically adjusting the DBMS's knobs for a given workload and hardware environment. For example, PGTune [40] is a rule-based knob tuner for PostgreSQL. Given a hardware specification (e.g., CPU, memory, disk type) and expected workload type (e.g., web application, data warehouse), it then applies predefined heuristics to recommend a set of knobs that are generally better than the default PostgreSQL settings. More advanced knob tuners use techniques such as Bayesian optimization [177] and reinforcement learning [180] to discover and recommend better knob settings.

Index Tuners: One effective technique for improving query performance is tuning database physical design through judicious index selection. Database indexes accelerate read queries in exchange for increased write latency and storage consumption. The optimal set of indexes therefore depends on the expected future workload. Both database administrators and index tuning tools leverage historical workload traces to identify beneficial indexes. For example, Dexter [50] is an index tuner for PostgreSQL that analyzes query logs, identifies slow queries, and uses PostgreSQL's hypothetical index API [101] to evaluate potential indexes without physically creating them. It then recommends indexes based on the predicted performance improvement. However, there are a multitude of considerations for each index candidate such as its type (e.g., hash index), physical organization (e.g., fillfactor), predicate constraints (i.e., partial indexes), and attribute composition (e.g., multi-column indexes), making it intractable for human operators to truly reason over all possible indexes. Even a modest schema [17] with 364 attributes across 58 tables generates over 10^{22} indexes when only considering multi-column candidates. Recent tools address this combinatorial explosion of possibilities through similarity search [204], but doing so requires substantially longer tool setup times as training data must be collected to calibrate similarity models.

Query Tuners: A DBMS's optimizer specifies how a query should be executed by producing a query plan. Because plan enumeration is NP-hard [102], query optimizers employ heuristics that may fail to find the optimal plan. Query hints provide a mechanism for external guidance of the optimization process, resulting in alternative plans with different performance characteristics while preserving query semantics. For instance, PostgreSQL supports query hints that disable or mandate specific join types [37] (e.g., hash join, merge join). However, there are too many potential hint combinations to identify each query's optimal hintset through exhaustive enumeration. Query tuners address this problem by suggesting better query hints directly [204] or by providing a framework for intelligent hint exploration. For example, Microsoft SQL Server's Query Hint Recommendation tool [135] automates hint exploration for individual queries. Given a time budget, it intelligently tries new hintsets and prunes candidates that underperform the current

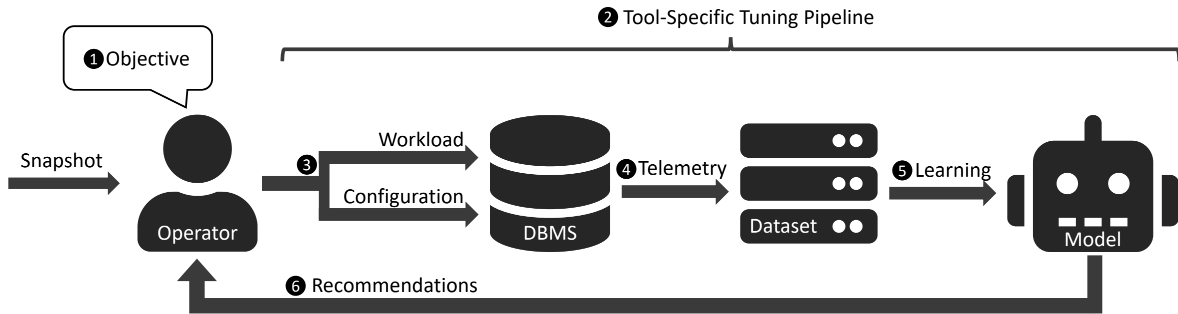


Figure 2.1: **Tool-Based Pipeline** – A pipeline that uses tools to improve DBMS configurations.

best solution. Other approaches employ learning-based techniques [199, 204] to discover better hintsets, but this requires more tool setup time (e.g., collecting telemetry under various hintset configurations to calibrate tool models [132]).

Partitioning: A query's performance depends on the cost of accessing and processing data. This cost is managed by a physical database design's partitioning decisions. Partitioning schemes enable the division of access methods (e.g., tables, indexes) into disjoint sets of data that are physically stored and accessed separately according to a partition function (e.g., hash partitioning, range partitioning). Because partitioning is an NP-hard problem [47], automated partitioning advisors utilize cost-driven search strategies that rely on the DBMS's optimizer [141, 158], analytical cost models [150], or ML models [92]. However, these models lack generalizability and require recalibration to accommodate changes in database schema, workload, DBMS version, and hardware configuration.

Although these tools address different aspects of database tuning, they share common operational requirements. They require integration into an existing DBMS deployment to observe system behavior, evaluate candidate configurations, and apply recommendations. Moreover, many tools require an extensive calibration phase to discover the salient characteristics of each deployment. The next section describes typical integration and calibration procedures.

2.3 Tool-Based Database Tuning Pipelines

Existing tool-based approaches to database tuning assume that a human operator will manually set up, deploy, and operate tools to optimize the DBMS. To understand tool-based optimization, we step through a representative tool-based database tuning pipeline as depicted in Figure 2.1.

Given a snapshot containing a DBMS deployment's workload trace and database contents, ❶ a human operator must first determine what objective to optimize (e.g., p99 latency). Next, ❷ the operator selects a tool that optimizes the objective (e.g., Proto-X [204]) and establishes a tuning pipeline for it. ❸ The operator extracts a representative workload from the snapshot and configures a DBMS to run it on. Typically, operators will avoid running tools on the production DBMS to maintain the production system's stability. Instead, they will set up an offline copy

or forked-off replica [125] with similar data and hardware characteristics, isolating it from the production deployment.

❹ Next, the operator runs the workload against the DBMS to collect telemetry about its behavior. Examples of telemetry [119] include low-level hardware resource metrics (e.g., CPU usage, memory usage) and higher-level observations (e.g., annotated query plans). Although some tools automate the aggregation of their necessary telemetry [61], others require the DBA to manually collect the telemetry into a dataset (e.g., a folder that contains executed `EXPLAIN ANALYZE` PostgreSQL plans) and supply it as input.

❺ The operator then supplies the dataset as input to the tool. Although simple rule-based tools may not require much setup (e.g., PGTune [40] uses if-else rules on hardware characteristics), ML-driven tools must often perform extensive setup to support their core functionality (e.g., QPPNet [130] trains neural networks to model the DBMS's performance). The dataset's expected format and the setup procedures vary widely between tools, even for those that are designed for the same task. As such, an operator must familiarize themselves with the idiosyncrasies of each tool that they want to incorporate into their database tuning process.

❻ After establishing a tool's tuning pipeline, the operator invokes it to obtain recommendations for improving the DBMS's configuration. These recommendations range from knob settings (e.g., buffer pool size) to physical design changes (e.g., creating indexes) to workload-specific optimizations (e.g., query hints). However, the operator must manually review and apply such recommendations to both their test DBMS environment and to the production deployment. Because existing tools do not guarantee that their recommendations are beneficial or even safe to apply, the operator is ultimately responsible for the outcome of this database tuning process.

Given this, it is evident that existing tool-based database tuning pipelines are *human-centric*: they require human operators to be involved at every step of the process. Although tool-assisted database tuning pipelines are effective [48], their deployment is limited by the availability of skilled human operators with sufficient expertise to set up and operate them. This paucity of experts is further exacerbated by two limitations of the tool ecosystem: (1) poor tool interchangeability, which prevents operators from easily substituting one tool for another that performs the same function, and (2) limited tool interoperability, which makes it hard to compose complementary tools into cohesive tuning pipelines.

Thus, one method for bridging the gap between human-centric tool-based tuning and fully autonomous self-driving DBMSs is to minimize the human expertise and involvement necessary to deploy and operate such tool-based pipelines. This middle ground approach aims to maintain compatibility with existing DBMS deployments while conferring the majority of a self-driving DBMS's benefits.

Chapter 3

Database Gym

Prior research on tool-based database tuning has focused on enhancing recommendation quality by augmenting tools with ML-driven models of DBMS behavior [126, 130]. For example, QueryFormer [208] improves index recommendation tools by developing a tree-structured Transformer [178] model that more accurately predicts query plan performance. However, recent advances in ML have diminished the centrality of model architecture design in autonomous DBMS research. First, the emergence of pre-trained models that adapt to a wide variety of tasks (i.e., foundation models [185]) has reduced the need for specialized architectures. Second, automated frameworks [82] have largely eliminated the engineering burden of model development, enabling the creation of effective models with minimal effort or ML expertise [71, 72, 133, 195].

Given this, we contend that the next important challenge in tool-based database tuning is obtaining the *high-quality training data* needed to build these ML models [159]. Other ML-intensive domains, such as robotics [106, 174, 201] and self-driving vehicles [54, 188], obtain training data through software-based simulators. A simulator attempts to approximate the behavior of an entity when it would otherwise be too costly, time-consuming, or dangerous to experiment on the real system. Researchers typically package such simulators into *gyms*, standardized toolkits for developing and evaluating ML models and algorithms. For example, OpenAI's Gym [59] (OpenAI-Gym) is a commonly used platform for reinforcement learning research. The standardization of simulator environments in gyms has accelerated ML research by unifying efforts to build end-to-end pipelines for rapid model prototyping and productionization. Similar gyms have emerged across other research domains, including networking [85] and fintech [49].

Despite the clear benefits of gym environments for ML research, to the best of our knowledge, no one has attempted to build a gym to support autonomous DBMS efforts. One possible reason is that the effort required to build a DBMS simulator is tantamount to building the DBMS itself. Instead, researchers and practitioners have built bespoke training modules for individual DBMS components (e.g., optimizers [128, 132], executors [126]) that do not capture the full complexity of a DBMS's behavior. This fragmented approach reimplements basic infrastructure such as workload capture and analysis. Moreover, it complicates comparing the performance and generalizability of different approaches.

To overcome these problems, the next phase of autonomous DBMS research is to develop a **database gym** that uses a DBMS as its own simulator, rather than building one from scratch. Such a gym addresses the ML and systems challenges of training data acquisition within a unified

framework. Because it manages the entire end-to-end tuning pipeline, it also enables unique opportunities such as pipeline-level optimizations. The gym aims to expedite autonomous DBMS research by providing an extensible platform for obtaining high-quality training data to augment existing tool-based methods, enabling them to approach the capabilities of self-driving DBMSs.

3.1 Background: Simulation Targets

To use a DBMS as its own simulator, the database gym must control the key aspects that govern a DBMS's observable behavior. In a tool-based tuning pipeline, such behavior constitutes the training data that calibrates ML-augmented tools [61] and determines their effectiveness. This training data consists of inputs (i.e., features) and outputs (i.e., labels). A model fits the training data by learning associations between inputs to predict outputs. Achieving the highest level of automated operation requires ML models that capture subtle interactions between different DBMS components. For example, models must account for how background processes (e.g., garbage collection) affect query performance. Learning such interactions requires *high-quality training data* that either (1) represents the interactions as explicit input features or (2) manifests their effects as output labels. The former approach is intractable because there are too many candidate features, many of which are impractical to obtain [61]. The latter approach is more feasible, as it only involves collecting training data under similar conditions to the target DBMS. Therefore, we define high-quality training data as data collected using a workload and state that approximates the target DBMS. This approximation ensures that the DBMS generates training data that captures these subtleties without explicitly modeling all DBMS interactions.

We now describe the aspects of a DBMS simulator that are important for training data quality: (1) *workload* and (2) *state*. We also discuss issues surrounding how the DBMS evaluates *actions* to generate diverse training data. This discussion motivates the design of a *database gym*, which we present in Section 3.2.

3.1.1 Workload

The first aspect of a DBMS that is relevant to training data collection is its workload (i.e., the queries it executes). Both commercial DBMSs (e.g., Oracle [83], Microsoft SQL Server [23]) and popular open-source DBMSs (e.g., PostgreSQL, MySQL) provide tools for workload capture. Previous work uses different representations for captured workloads, including raw SQL [180, 211], query plans [130, 179], system metrics [48, 126, 177], and simple classification (e.g., OLTP vs. OLAP) [111]. We contend that defining the workload as a time series of raw SQL is the best approach because (1) the queries do not change as the system changes, and (2) all other representations derive from them.

Given a historical workload of SQL queries, a DBMS's workload forecasting component seeks to predict and synthesize the future workload. The challenging questions in synthesizing a workload are deciding (1) what queries to consider for execution and (2) how to determine their arrival times. Existing research in workload prediction attempts to forecast volume [124], predict the subsequent SQL query [77, 149], and detect shifts [94]. Furthermore, workload compression [65] identifies important subsets of the workload for monitoring [75] and tuning [166].

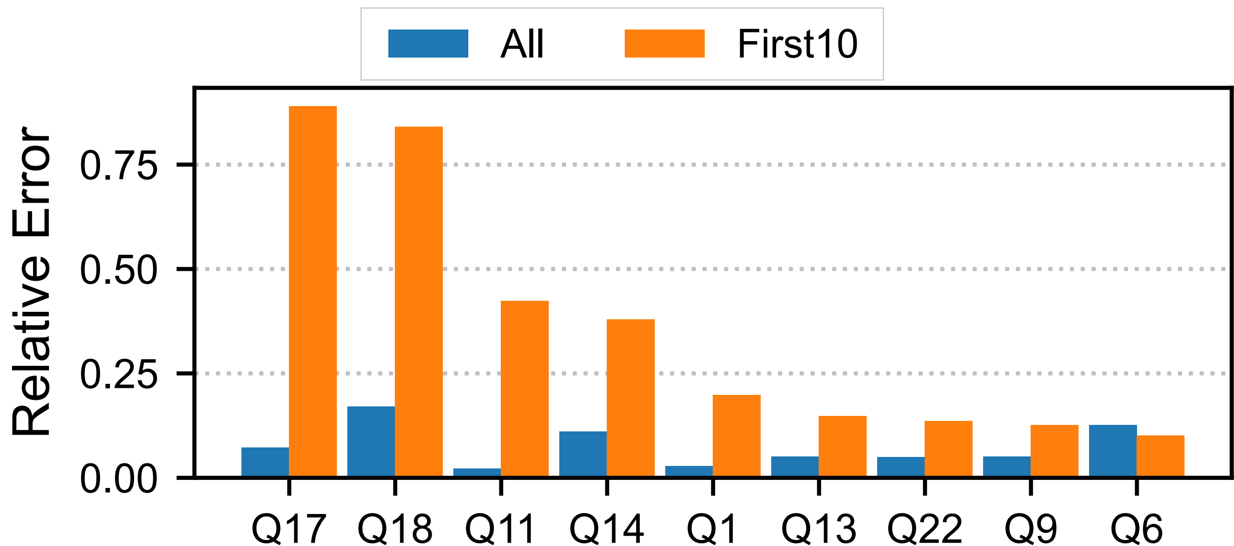


Figure 3.1: **Workload Effects on Modeling Error** – We train two models of QPPNet to predict query elapsed time on PostgreSQL v14 with TPC-H (scale factor 10). The All model enables hash joins for all 22 queries. The First10 model enables hash joins for only the first 10 queries. We leave out bars where both models are below 0.125 relative error for readability.

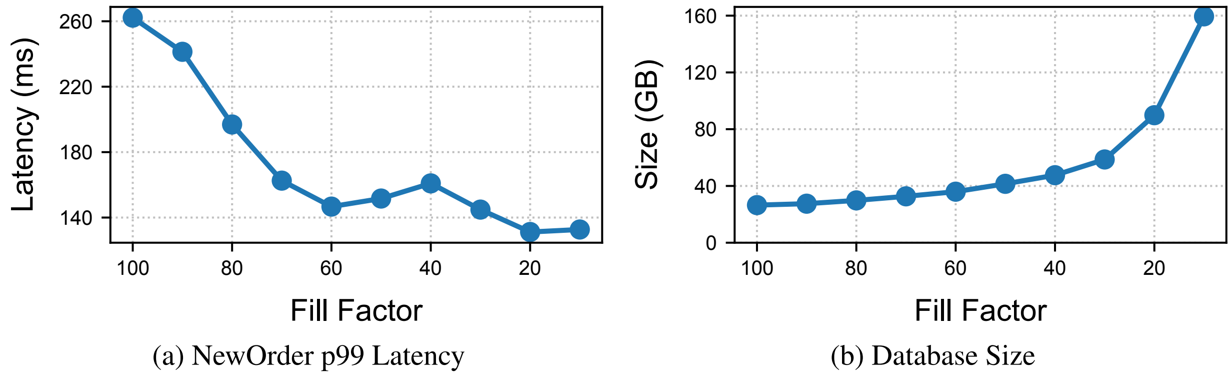
However, most approaches only predict high-level properties or operate on historical workloads. To generate training data, the DBMS requires an executable workload that approximates the target DBMS's future workload (e.g., as a time series of future SQL queries).

To illustrate why this matters, we study QPPNet [130], an ML model for query performance prediction, using the TPC-H benchmark on PostgreSQL v14. QPPNet trains on annotated query plans (i.e., `EXPLAIN ANALYZE`) to predict the execution time of new queries. For this experiment, we simulate workload drift by disabling hash joins for half of TPC-H's queries while collecting training data (e.g., query plan change due to statistics updates). As Figure 3.1 shows, if aspects of the target DBMS's workload are missing in the training data (i.e., some queries switch to using hash joins), then the model has a higher relative error and makes worse predictions. These results suggest that historical workloads are insufficient for training an autonomous DBMS's ML models, necessitating that the system predicts and collects training data for the future workload.

3.1.2 State

A DBMS's state affects its runtime behavior during training data collection. As we now describe, this state comprises the DBMS's *hardware* resources, *logical* contents, and *physical* condition.

Hardware: The state's hardware specification details the resources available to the DBMS. This information includes compute (e.g., number of CPUs, cores, ISAs), storage (e.g., disk, memory), and network topology. These resources can either be the target DBMS's current hardware specification or potential additional resources (e.g., different cloud instance sizes).



(a) NewOrder p99 Latency

(b) Database Size

Figure 3.2: **Latency vs. Storage Trade-off** – As fill factor decreases, the latency of the NewOrder transaction decreases, but database size increases.

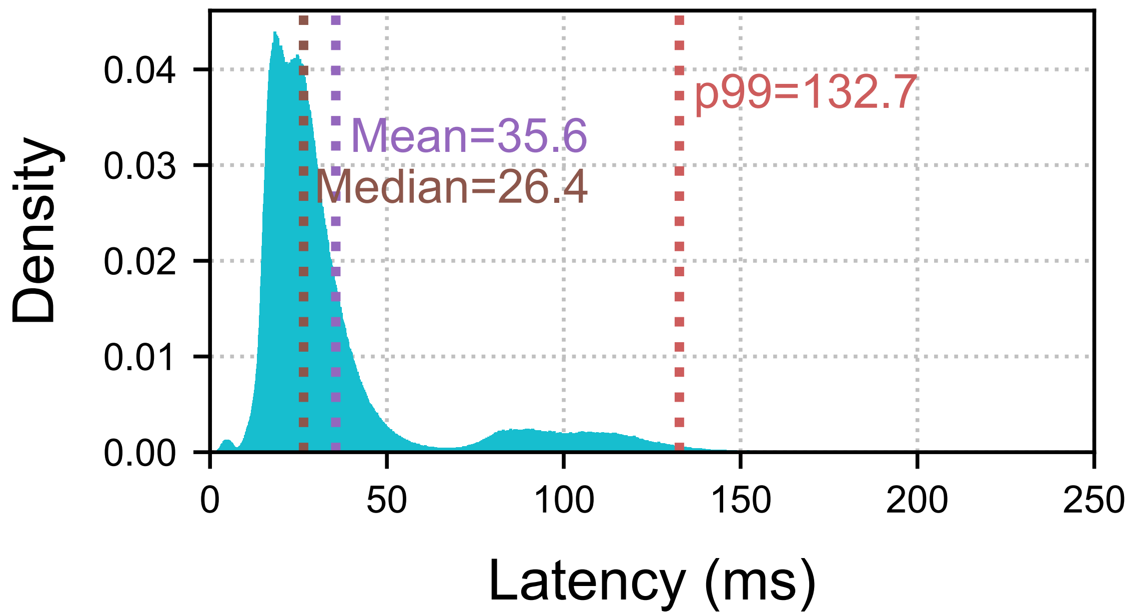
Logical Contents: This component refers to the contents of the database that are externally visible to users and applications. These contents include both the database schema (e.g., columns, views, indexes) and tuples (e.g., values, number of tuples, distributions).

Physical Condition: Lastly, this component includes the aspects of the DBMS that pertain to its physical elements, including (1) the contents of data pages, (2) the layout of auxiliary data structures, and (3) the DBMS's knob configuration.

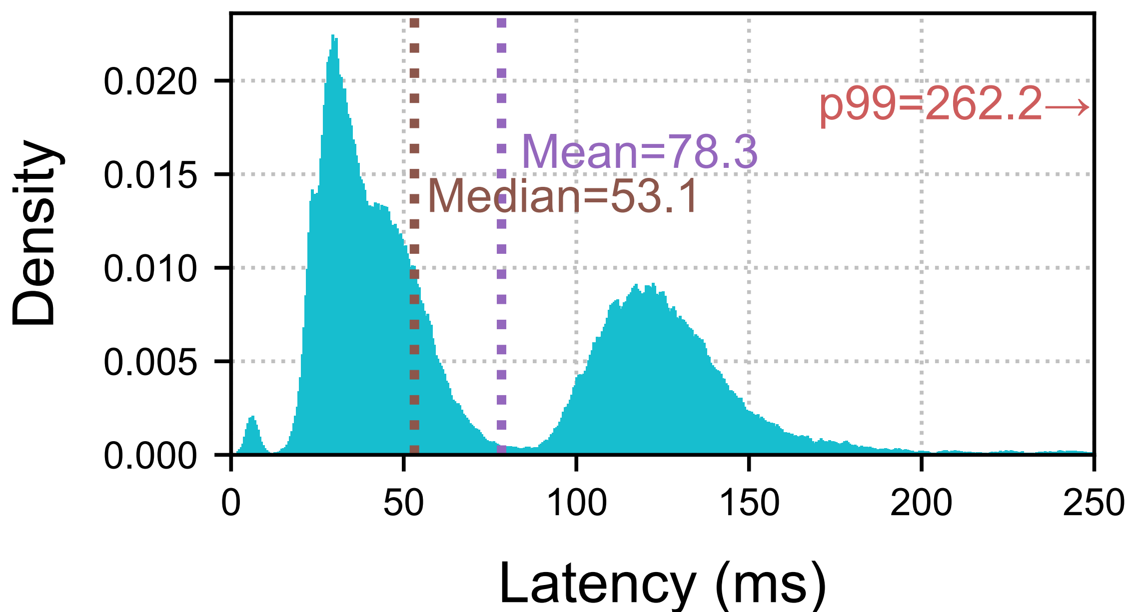
A DBMS's run-time behavior (e.g., query execution performance) obviously depends on its hardware resources and logical contents. Its physical condition is equally important but is often overlooked in training data collection. When a DBMS executes a query that modifies a table, it can alter the table's underlying physical organization in ways that affect the behavior of future queries. For example, because of how PostgreSQL implements multi-versioning [191], updating a tuple may either (1) insert the new version on the same page as the original version or (2) insert the new version on another page. The logical contents in the database are the same in both scenarios, but each result in a different physical condition with varying performance implications.

To demonstrate the importance of accurately simulating a DBMS's physical condition, we execute TPC-C [171] on PostgreSQL v14 while changing how the DBMS stores data. For each trial, we vary the DBMS's knob (`fillfactor`) that determines how much free space the system reserves in a page when inserting new tuples. Reducing the fill factor makes it more likely that the DBMS places a tuple's new version in the same page as the previous version upon update. However, the extra padding also increases the storage size on disk. We load the database with BenchBase [13, 76] and execute each trial for 30 min (scale factor 200, 200 terminals).

Figure 3.2 shows how changing the DBMS's physical condition results in trade-offs between its performance and storage size. As we decrease PostgreSQL's fill factor knob, transaction latency decreases in Figure 3.2a while database storage size increases in Figure 3.2b. These results are unsurprising. However, Figure 3.3 shows that the minimum and maximum fill factor settings (at 10% and 100%, respectively) have different latency profiles, with the latter being pronouncedly bimodal. Moreover, the current knob value may not reflect the actual physical



(a) Fill Factor = 10



(b) Fill Factor = 100

Figure 3.3: **NewOrder Latency Distributions** – As the fill factor decreases, the latency approaches a Gaussian distribution.

condition until a `VACUUM FULL` operation is executed. This makes modeling physical condition difficult as the current knob value may not accurately reflect system state.

This experiment highlights the need to consider a DBMS's physical condition when planning more complex actions (e.g., table fill factor). Therefore, the quality of both training data features and labels depends on achieving a state representative of all three facets of a DBMS.

3.1.3 Actions

Most DBMSs provide programmatic APIs for deploying actions in the following categories: (1) *physical design*, (2) *schema changes*, (3) *knob configuration*, (4) *hardware resource allocation*, and (5) *query plan hints* [152]. The first two categories are changes to the database's physical state (e.g., indexes, views, partitioning) and logical state (e.g., changing column types). The third category consists of optimizations that affect the DBMS's behavior through its configuration knobs (e.g., caching policies). Next, resource allocations determine how the DBMS uses its available hardware to store data and execute queries. The system may either provision new resources (e.g., adding disks, memory, or machines) or redistribute existing resources (e.g., partitioning tables across disks). Lastly, query plan tuning hints are directives that force the DBMS's optimizer to make certain decisions for individual queries (e.g., join orderings). An autonomous DBMS must collect training data on these actions for its ML models.

Generating training data for all these actions is infeasible because too many possible actions exist. Thus, the key challenge is how to identify the actions that are worth considering based on the target DBMS's workload and state. For example, consider building multi-column indexes for a read-heavy workload like Wikipedia [76]. Wikipedia's MySQL schema [17] has 364 attributes across 58 tables. There are $\sum_{i=0}^{m} \frac{m!}{(m-i)!}$ potential multi-column indexes for a table with m attributes, which means Wikipedia has approximately $7 \cdot 10^{22}$ possible indexes. Even if one could obtain training data for each index in a nanosecond, it would still take two million years. This estimation also does not consider index types (e.g., hash vs. B+tree) and index arguments (e.g., split/merge thresholds, page sizes), exacerbating the problem.

Prior work addresses this problem by pruning the action search space with ad-hoc rules. For example, some index selection tools only consider two-column indexes [50], while others prune candidates based on simple criteria (e.g., columns appear in predicates [176]). But such ad-hoc rules will not result in holistic ML models for autonomous DBMSs. These rules must be programmable, enumerable, and integrated into the training data pipeline.

This dissertation focuses on DBMS-scoped actions. Future work may also include tuning OS-level knobs (e.g., page cache policy) or modifying application code (e.g., JDBC/ODBC libraries).

3.2 Architecture

The totality of the above problems impede the collection of the training data needed for an autonomous DBMS's ML-based components. Moreover, these problems reoccur across DBMSs, yet existing solutions are typically DBMS-specific (e.g., buffer pool probes). What is needed is an abstraction layer that synthesizes a future workload and state while also coordinating generation and execution of actions to produce high-quality training data. Such an abstraction would allow for the rapid prototyping and evaluation of ML models. The development of this abstraction motivates our database gym.

A gym is a toolkit that provides a consistent environment for offline training and evaluating ML algorithms that automate tasks on an agent. One of the most influential gyms is OpenAI's Gym (OpenAI-Gym) [59], designed for reinforcement learning (RL) problems. The RL setting places an agent inside an environment that advances in discrete simulation timesteps. At

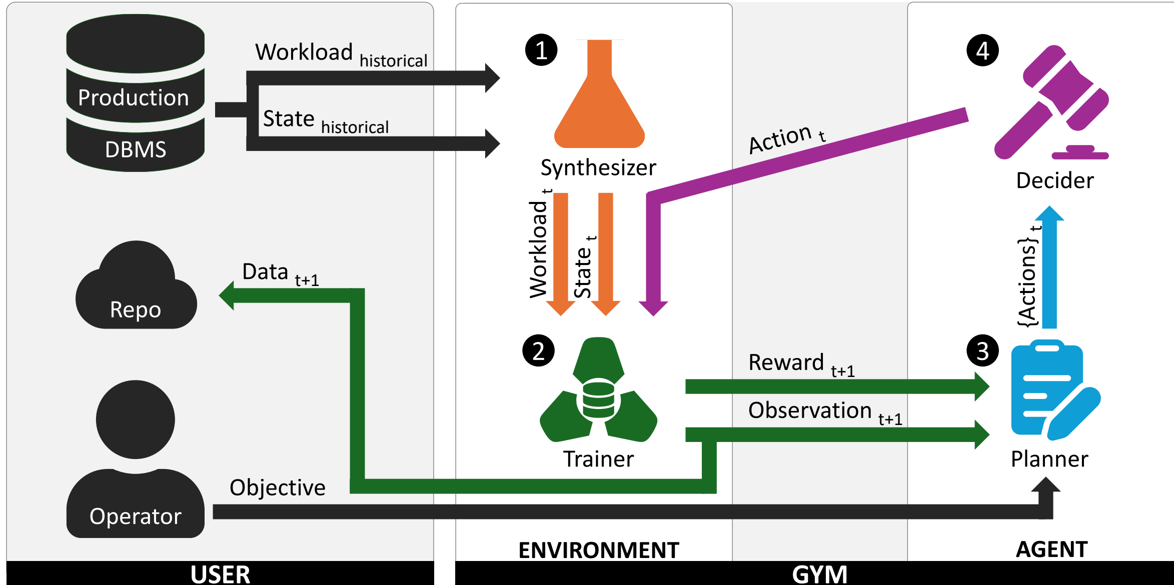


Figure 3.4: **Database Gym Architecture** – An overview of the database gym's internals. The t subscript indicates how the gym's components produce new data through rounds of iterations.

each timestep, the agent performs an action inside the environment that generates an observation and a reward. The agent considers past observations and rewards as it attempts to maximize its expected future reward by formulating an action-picking policy that explores an action space. OpenAI-Gym's key contribution is providing a standardized API for agent-environment communication. This standardization led to the development of different environments (e.g., video games [57], robotic arms [59]) and agents [156, 157] that enable reproducible benchmarking and evaluation. Moreover, OpenAI-Gym is extensible both in core functionality (e.g., distributed training [118], multi-agent environments [169]) and to other research domains [49, 85, 201].

The challenge of developing a *database gym* (Gym) is that simulating the environment is complex and slow. Moreover, unlike other relatively standardized domains (e.g., most cars drive similarly), DBMS deployments are expected to operate across heterogeneous hardware and diverse workloads, which requires heavy customization. The good news is that a perfect simulator exists for every DBMS, namely, the DBMS itself. But one cannot simply wrap a DBMS with a gym API and expect it to be usable. The gym must provide additional mechanisms for establishing the right state in the DBMS and executing workloads to produce high-quality training data. The gym should also generate training data quickly (i.e., faster than real-time query execution).

As Figure 3.4 shows, the gym has three main inputs: (1) the objective function and constraints, (2) the historical workload, and (3) the historical state. The gym uses these inputs and its internal components to produce training data that it stores externally.

The gym's internals are divided into its *Environment* and *Agent*. The Environment consists of two components: ❶ the Synthesizer installs the state and workload of the target DBMS, and ❷ the Trainer coordinates creating an instance of the target DBMS, applying actions, and executing the workload to generate this iteration's observation and reward. The gym collects the Trainer's output as training data, stores it into a repository (Repo), and provides it as feedback to the

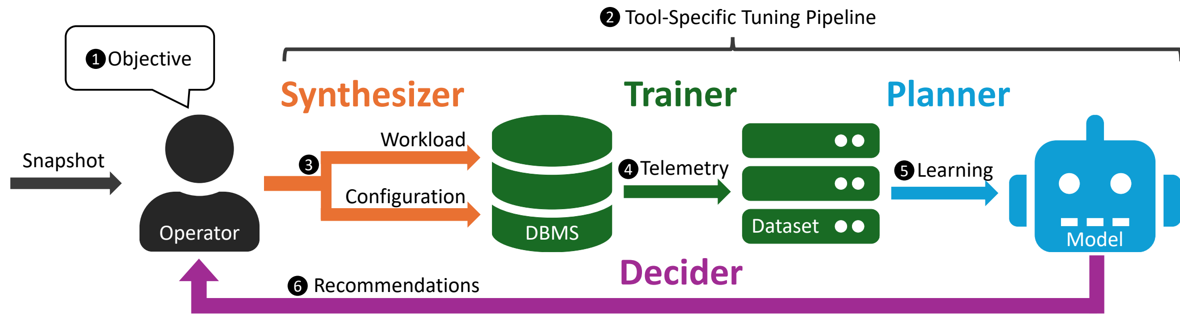


Figure 3.5: **Systematizing Pipelines** – Mapping tool-based database tuning pipelines to the database gym's components.

Agent. The Agent incorporates this feedback into ❸ the Planner to generate candidate actions and uses ❹ the Decider to select actions to deploy. The relation between these components and tool-based database tuning pipelines is shown in Figure 3.5. Through its components, the gym aims to streamline the development and evaluation of database tuning tools.

In the rest of this section, we describe the gym's internals in further detail, focusing first on the Synthesizer and Trainer, followed by the Planner and Decider. In Section 3.3, we discuss problems in system infrastructure related to running the gym.

3.2.1 Synthesizer

The Synthesizer's goal is to produce a workload and state for the Gym's target DBMS. A baseline implementation simply passes through the historical workload and state (i.e., predicts no change). As Section 3.1 describes, although this is not ideal, it suffices under certain assumptions (e.g., static read-only workloads). A better approach is to generate a future workload and state.

Section 3.1.1 introduced the problem of synthesizing the target workload. Generating an executable workload [99] matters because existing tuning tools largely analyze and cost queries in isolation [74, 130] even though queries are not independent and have complex interactions when run together (e.g., lock contention, scan sharing). Furthermore, running a query can produce secondary effects (e.g., checking constraints, generating maintenance work, updating statistics) that are hard to model. Until more advanced models can capture these interactions, the Gym must execute the target workload's SQL queries to generate high-quality training data. Therefore, the Synthesizer needs to support predicting individual SQL queries, which brushes against the limits of ML scalability (e.g., taking 2 ms to produce an executable query limits throughput to 5000 queries per second). Many such open problems remain in making workload synthesis practical.

Section 3.1.2 describes the composition of the target state. The user specifies the hardware resources available, which may differ from the production DBMS. The Synthesizer approximates future logical contents by measuring database growth rates and then scales [161, 168, 196, 202] the contents of tables accordingly. However, we are unaware of any work synthesizing future physical condition. We note that exact target state predictions are unnecessary as long as execution characteristics are similar (e.g., fill factor trends matter more than precise tuple location).

3.2.2 Trainer

The Trainer receives the generated workload and state from the Synthesizer and obtains actions from the Decider. It then installs the relevant state into the target DBMS and applies any necessary actions. The Trainer coordinates executing the synthesized SQL workload on the target DBMS to produce an observation and reward for future rounds of training data generation.

The biggest challenges in this process are minimizing resource usage, accelerating query execution, and achieving the target's physical condition (e.g., Figure 3.3 shows that page layout matters). For example, consider collecting training data on a workload when scaling the database size from 100 GB to 1 TB. The gym can collect this data by (1) synthesizing and loading an additional 900 GB of data, (2) overriding query plan leaf operators to scan "virtual" tuples [162], or (3) downsampling to a representative subset and extrapolating results. The first option is the most accurate but also has the highest storage cost and execution time. The second option saves on storage, but it requires invasive changes to the DBMS and moreover does not speed up query execution. Lastly, the third option saves on storage and execution time, but it may introduce extrapolation errors and have a different physical condition from the target DBMS. Searching for scaling laws to extrapolate from smaller and/or cheaper database instances, and exploring the trade-offs of these approaches, remain open problems.

However, because the Trainer orchestrates the entire end-to-end training data pipeline, it enables the collection of metrics that require coordination across subsystems. For example, the Trainer can place its internal DBMS instance on a custom filesystem that emulates a RAM disk while tracking syscalls. Doing so provides faster than real-time execution for disk-backed DBMSs by accelerating all disk operations. Additionally, the read/write syscall counts provide a logical hardware-independent feature for the workload that generalizes the training data across different hardware environments. This training data improvement is possible because the Trainer completely controls the internal target DBMS instance.

The Trainer's overarching goal is to run workloads much more cheaply than in a real DBMS. For example, UDO [180] distinguishes between cheap and expensive actions to batch and evaluate cheap actions together. Similarly, one possible approach for the Trainer is to share large amounts of state between workload evaluations (e.g., by computing and maintaining small deltas between "adjacent" states).

3.2.3 Planner

The Planner generates candidate actions that it believes are valuable based on its knowledge of prior training outcomes (i.e., the Trainer's past observations and rewards) and the user's objective function (e.g., optimize for p99 latency).

The Planner requires a way to suggest good actions. General database administration guidelines prescribe universal rules (e.g., use PGTune [40]), but these are often not optimal [177]. On the other hand, collecting training data for all possible actions is not feasible (see Section 3.1.3). Therefore, the Planner's key challenge is finding an action representation that allows for programmatic exploration and intelligent pruning of the search space.

Existing research uses more sophisticated methods to prune the action search space. For example, prior work generates initial training data for ML models to select knob configurations

via Latin hypercube sampling [177]. However, the resulting actions do not have a notion of distance from one another (e.g., an action that sets PostgreSQL's working memory to 4 MB should be considered more similar to a 5 MB setting than a 50 MB setting). The Planner could learn these distances with deep action embeddings [80].

3.2.4 Decider

The Decider selects the most promising actions from the list of actions suggested by the Planner and provides them to the Trainer. Learning to pick the most promising action is the purview of RL research. Therefore, the Decider may incorporate multiple learning algorithms with minimal modification (e.g., SB3 [157], UDO [180]).

3.3 Deployment

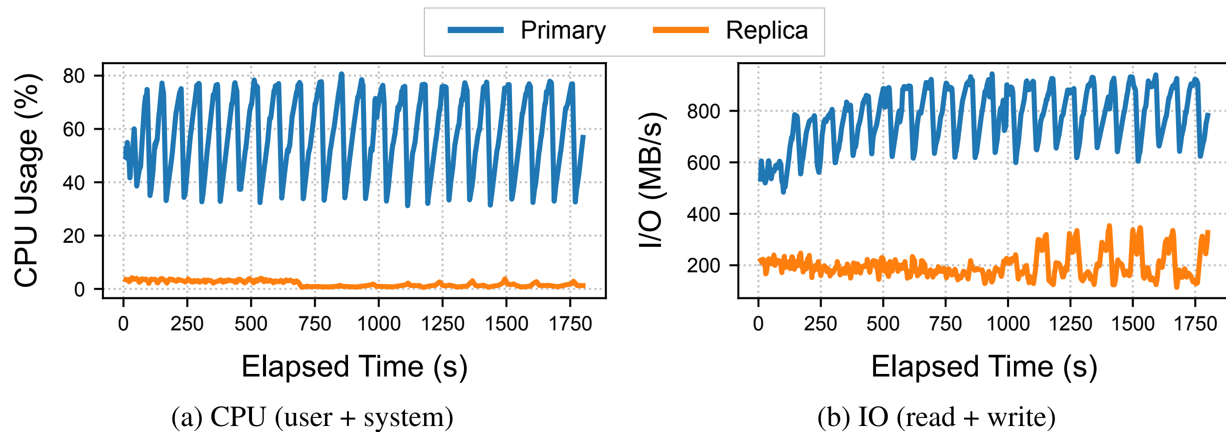
There are two challenges to running the Gym in production: (1) bootstrapping the database contents and (2) bootstrapping the hardware. We focus on the trade-off between the *cost* (i.e., how expensive it is to run the gym) versus *impact* (i.e., the gym's potential disruption to an existing DBMS deployment) of running the gym.

Two obvious considerations are whether the Gym can run on the production DBMS (low cost, high impact) or if it must run on a separate instance (high cost, low impact). The first option requires restricting the gym to avoid violating service-level objectives with bad actions on the production DBMS. For example, Oracle can try index actions in production because it schedules index exploration at idle times and forces index usage with query hints [143]. But some actions are difficult to sandbox (e.g., knob configurations [153]). Hence, we believe that the Gym should not run directly on a production system. The second approach involves creating an exact replica of the production DBMS on the same hardware. Microsoft uses this approach (calling them "B-instances") for auto-indexing in Azure [74, 125]. Such instances are completely isolated, which allows the Gym to explore actions freely, but this approach is prohibitively expensive and infeasible at scale.

Given the above issues, the ideal Gym deployment keeps a B-instance's isolation advantages but without the cost. To achieve this, we can exploit the fact that (1) the Gym can tolerate stale database snapshots, and (2) the DBMS can tear down the non-essential gym at any time. We therefore contend that we should reuse high-availability (HA) replicas to deploy the Gym in a resource-limited container. HA replicas are notoriously idle, allowing the Gym to scavenge [121] unused resources for its training purposes. These containers bootstrap their database contents with incremental snapshots and the DBMS's built-in recovery logic.

To demonstrate the idle resources available for the Gym, we ran SmallBank [13, 76] on PostgreSQL v14 with a primary/replica configuration on two machines while measuring CPU and disk I/O utilization on each machine. Figure 3.6 demonstrates that HA replicas have ample amounts of spare compute and reasonable disk I/O available. However, this approach requires raw access to the DBMS instance, making it difficult in managed cloud environments.

Lastly, although this dissertation restricts its focus to optimizing autonomous single-node systems, real DBMS deployments offer more tuning opportunities (e.g., optimizing replication



(a) CPU (user + system)

(b) IO (read + write)

Figure 3.6: **Replica Resource Availability** – We run SmallBank (scale factor 100, 2000 terminals, 30 min) on PostgreSQL with one primary and one replica. Each datapoint averages across the last 10 readings to reduce noise.

```
1  import gymnasium as gym
2
3  env = gym.make("CartPole-v1")
4  observation, info = env.reset(seed=42, return_info=True)
5
6  for _ in range(1000):
7      action = env.action_space.sample()
8      observation, reward, terminated, truncated, info = env.step(action)
9      if terminated or truncated:
10         observation, info = env.reset()
11
12 env.close()
```

Figure 3.7: **OpenAI Gym Example** – Code that learns to balance a pole on a moving cart.

network topology). Moreover, beyond database tuning, the Gym's organization provides opportunities to replace or learn its components. We defer such considerations to future work.

3.4 Initial Implementation and Discussion

As Section 3.2 describes, OpenAI-Gym is an extensible framework for reinforcement learning research that provides a collection of standardized benchmark environments with a common interface for developing and comparing reinforcement learning algorithms. For example, Figure 3.7 establishes an environment for balancing a pole on a moving cart [56].

The first prototype of the database gym integrated directly into the OpenAI-Gym framework by providing a new environment for database tuning. It used BenchBase [76] for workload and state generation, contained bespoke code for managing a PostgreSQL DBMS deployment, integrated TScout [61] and `EXPLAIN ANALYZE` workflows for telemetry collection, constructed

```python
import gymnasium as gym

# Specify the DBMS tuning environment.
env = gym.make(
    jdbc_url,        # str: The DBMS to connect to and bootstrap from.
    workload_trace, # Path: The historical workload trace.
    horizon,         # timestamp: The target future optimization horizon.
    reward_fn,       # str: Reward function to be optimized (e.g., "p99-latency").
    action_strat,    # str: Strategy for action generation (e.g., "MCTS").
)

# Initialize the DBMS state.
observation, info = env.reset(seed=42, return_info=True)

# Run 1000 tuning steps.
for _ in range(1000):
    # Generate and select the next tuning action. We omit details for brevity.
    action = next_action()
    # Apply the action, run the workload trace, and observe the reward.
    observation, reward, terminated, truncated, info = env.step(action)
    # Reset the DBMS state if necessary.
    if terminated or truncated:
        observation, info = env.reset()

# Clean up the DBMS state.
env.close()
```

Figure 3.8: **Database Gym Prototype** – Code that learns to tune a DBMS.

different models for DBMS behavior (e.g., ModelBot2 [126], QPPNet [130], AutoGluon [82]), applied workload heuristics to generate candidate index recommendations [180], and invoked OpenSpiel [112] for action selection. Figure 3.8 shows how the prototype was invoked.

This prototype enabled the comparison of different DBMS modeling methods by training them on the same dataset. We found that although previous research had emphasized different model architectures [126, 130], automated machine learning (AutoML) techniques such as AutoGluon [82] matched or surpassed the performance of specialized models. For example, on the DSB [78] workload, AutoGluon achieved 1.7 ms error with a few minutes of training, whereas a neural network-based method obtained 15 ms error with a week of training. This better performance was explained by advances in AutoML techniques that include automated feature engineering, neural architecture search, and model ensembling.

We observed that the latency of the database tuning pipeline was dominated by the time required to reset the environment (i.e., loading the relevant state into the DBMS) and running the workload to gather telemetry. The gym can alleviate the cost of the former through aggressive caching and batching [180] of DBMS states. However, it is more difficult to speed up telemetry generation. Existing techniques for telemetry generation are effectively equivalent to running the workload, and the DBMS is already running the workload as fast as it can. Thus, another challenge in improving tool-based database tuning pipelines is accelerating telemetry generation.

3.5 Conclusion

Most of the previous work in using ML for DBMS automation has focused on designing better ML models of DBMS behavior, but recent advances in ML have largely automated model design. The challenge now is to obtain good training data for building these models. This chapter outlines the architecture of the database gym, an integrated environment that generates training data by using the DBMS to simulate itself at the highest possible fidelity. It also discusses how the next challenge in tool-based database tuning is to generate training data faster.

Chapter 4

Accelerating Training Data Generation

Chapter 2 describes how tool-based database tuning pipelines generate training data to construct their behavior models. The pipeline's tools then utilize these behavior models to adjust their recommendations for the system's run-time characteristics. A tacit assumption pervading existing work on tool-based tuning is that this model training data is cheap and easy to obtain (e.g., expecting more data collection on schema change). However, as Chapter 3 describes, advances in ML have improved the quality and reduced the training time of behavior models. Thus, the bottleneck in tool-based database tuning has shifted from designing better behavior models to acquiring training data for them.

Training data is slow and expensive to obtain [93] because it is collected by instrumenting [61] the DBMS as it executes a representative workload (e.g., a trace of SQL queries). In practice, we observe that existing pipelines spend over 93% of their time in collecting training data [78, 126, 206], which can take weeks. Despite this, to our knowledge, there have been no efforts to improve the speed of training data generation for self-driving DBMSs and tool-based database tuning pipelines alike. Some techniques sidestep this issue by learning cardinality estimation models from the data for approximate query processing (AQP) [93, 182, 197]. Although AQP speeds up query execution by estimating results, it does not help with behavior models since they must observe and predict the DBMS's run-time characteristics. Hence, obtaining training data remains a time-consuming bottleneck.

Another problem is that existing methods often have to retrain their models from scratch, either because of new environments (e.g., software updates [126], hardware changes) or model invalidation (e.g., due to dataset growth [130] or workload drift [131]). It is challenging to reuse training data from other deployments when they differ in database contents, DBMS versions, hardware, and system configuration [119].

Given this, what is needed is a way for the DBMS to generate training data faster. Because the DBMS generates training data during query execution, the two processes are coupled: it cannot produce training data faster than it can run the workload.

We now present the **Boot** framework that accelerates training data generation by uncoupling these processes. Boot aims to efficiently *bootstrap* a DBMS's behavior models. To achieve this, we introduce macro- and micro-acceleration (MMA) techniques that leverage the database gym's environment to modify the DBMS's execution semantics for faster query processing. Macro-acceleration decides when to execute a query for its telemetry, and micro-acceleration speeds up

execution by adaptively sampling the DBMS's run-time behavior. The key observation enabling our techniques is that a tradeoff exists between training data quality [119] (i.e., approximated telemetry) and collection time (i.e., how long it takes to obtain telemetry). To our knowledge, we are the first to expedite training data collection by altering query processing behavior.

To evaluate Boot, we integrate it into a database gym (see Chapter 3) for PostgreSQL as a drop-in extension. We assess Boot on its ability to improve training data collection time and the accuracy of its resulting models on both uniform and complex workloads. Our results show Boot achieves up to 268× speedup for a modest increase in model error [209]. We also show that Boot is 3–5× faster than hand-optimized sampling methods, completing in four days what the other techniques complete in three weeks.

Our work makes the following contributions: (1) decoupling training data generation from regular query execution to overcome the training data bottleneck that impacts both self-driving DBMSs and tool-based database tuning pipelines, (2) presenting the MMA techniques for accelerating training data generation, and (3) describing the design and implementation of the new drop-in framework Boot that incorporates MMA in a PostgreSQL database gym.

4.1 Background and Motivation

An autonomous DBMS [151, 153] optimizes itself for a target objective function (e.g., latency). It relies on *behavior models* [126] to predict system performance and identify actions to improve its configuration (e.g., knob settings), but generating the training data for such models is costly. To better understand this problem, we first describe how the *database gym* [119] builds models. Next, we discuss the cost of training data generation and how the gym can exploit query execution characteristics to accelerate it.

4.1.1 Database Gym

As Chapter 3 describes, a database gym [119] is a toolkit for accelerating autonomous DBMS research by orchestrating the construction and utilization of a simulated DBMS environment for training data generation. It first employs workload capture tools [83] to obtain a representative SQL trace [124]. It then uses the DBMS to simulate itself for an ML training environment that is isolated from production [119, 213], such as an offline [126, 177] instance or a high-availability [125] replica. Cloud vendors optimize their fleets with similar environments [74].

The gym executes the workload on the simulation DBMS to collect *training data* about the system's run-time behavior. Such training data contains mappings from input features (e.g., query plans, optimizer estimates) to output labels (e.g., execution time, CPU usage). The gym then provides the DBMS with the training data for building behavior models (i.e., learning functions of the inputs that predict outputs). The DBMS may construct models using bespoke methods (e.g., neural unit [130], operating unit [126]) or automated ML (AutoML) frameworks [119].

After training the behavior models, the gym coordinates the DBMS's *action planning* [153] to discover better configurations. A naïve method is for the gym to apply a candidate configuration, replay the workload, and then measure whether the objective improved. However, such an approach is infeasible because the replay cost compounds with a large number of actions [119].

Table 4.1: **Tuning Times** – Dataset collection and model training times in recent ML approaches.

Model	Collection (h)	Training (h)	Ratio
QPPNet [130]	300	24	0.93
MB2 [126]	9	0.3	0.97
OpAdviser [206]	40	0.05	0.99

Therefore, the DBMS depends on its behavior models to estimate its performance under new configurations without running the workload [130, 204].

Unlike other ML domains that dismiss data collection as a one-time cost (e.g., researchers share LLM model weights because training data does not change [18]), an autonomous DBMS needs training data specific to its workload and configuration. Reusing models from other deployments is challenging because the database's contents, hardware, and system configuration [119] influence data labels. For example, a sequential scan operator's run-time depends on the DBMS's disk I/O speed and buffer pool size. Additionally, even if the DBMS already has models, they may be invalidated because of dataset growth [130], workload drift [131], software updates [126], schema changes [180], or hardware upgrades [119]. This means that collecting the training data necessary [61] to build such models is the most time-consuming part of this tuning process.

4.1.2 The Cost of Training Data Generation

A pernicious assumption in existing modeling DBMS work is that it is easy to collect training data. In practice, a DBMS spends most of its time in the modeling process on executing queries to generate training data [115, 132, 140, 177, 180, 192, 198, 211]. To better understand this problem, we measured how long recent DBMS modeling methods spend on collecting data versus training. Table 4.1 shows that these methods spend over 90% of their time executing queries on the DBMS.

Despite this, there have been no attempts to optimize training data generation for autonomous DBMSs. Prior work reduces instrumentation overhead [61] or employs active learning to sample workloads [125, 179], but no technique increases query execution speed. Training data generation differs from regular query execution because it concerns telemetry (e.g., timing information) and not results (i.e., returned tuples). Because a DBMS does not need to produce exact results during this process, it can speed up collection if it can infer a query's telemetry without fully executing it. This acceleration is possible because queries in a workload can be repetitive, and the operations within a query may also be repetitive.

4.1.3 Exploiting Workload Repetition

The DBMS generates training data telemetry by executing a workload trace. Table 4.2 demonstrates query repetition in real-world traces and synthetic workloads: the DBMS repeatedly executes the same query templates with different input parameters. Furthermore, because the DBMS utilizes a limited number of operator types when formulating a query plan, the repetition of an

Table 4.2: **Repetitive Queries –** The number of queries and templates in recent workloads used in database tuning.

Benchmark	Type	# Queries	# Templates	Repetition
Admissions [130]	Real	2546M	4060	627k×
BusTracker [126]	Real	1223M	334	3.66M×
MOOC [124]	Real	95M	885	107k×
TPC-H [130]	Synthetic	20000	22	909×
DSB [78]	Synthetic	11440	52	220×
Stack [132]	Synthetic	5000	25	200×

operator's execution behavior is even more frequent than that of entire queries. Yet some repetition is necessary because the query's plan and behavior may change depending on parameters and system configurations [103], making it insufficient to execute each query template only once.

Query Repetition: What is needed is a way to determine *when* and *why* the DBMS should execute a query again so that it can execute fewer queries. In the training data context, the DBMS should re-execute a query if it has substantially different run-time telemetry from its past invocations. But the DBMS is collecting training data to bootstrap its models offline, so it has no models to predict a query's characteristics. Therefore, the DBMS must decide whether to re-execute a query based solely on its optimizer estimates and statistics, limiting the available techniques [67]. By skipping queries that do not exhibit new behavior, the DBMS executes fewer queries and thus reduces training data generation time.

Operator Repetition: There are also redundant and unnecessary parts of a query's plan that the DBMS can cut out for training data purposes. This speedup is important for queries that take a long time to complete because of bad configurations. For example, a query that runs slowly because of missing indexes will not get faster halfway through execution. Thus, it is important to reduce the time spent executing operators after they have become predictable; that is, the DBMS should spend the majority of its time exercising "useful" operators.

It is possible to skip redundant operators because of their independence. Each query operator is independent as its behavior only depends on its input tuples. The DBMS relies on this independence to build models from plan telemetry (e.g., `EXPLAIN ANALYZE`) [126, 130]. Our key insight is that integrating such modeling assumptions earlier into training data generation enables early query termination. For example, a sequential scan retrieves tuples by reading from buffer pool pages; the only variation from a telemetry perspective is whether obtaining the tuple involves a disk read. Once the DBMS observes both classes of sequential scan behavior (i.e., with and without disk fetch), it does not need to keep executing the scan.

4.2 Overview

Our analysis above shows that data generation routines are a bottleneck for ML-based DBMS automation because it takes too long to execute queries. But a DBMS does not need to compute

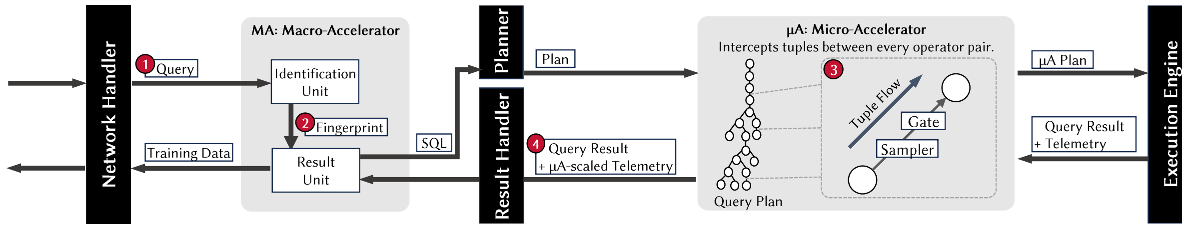


Figure 4.1: **Architecture** – An overview of Boot's internal components and execution flow. The MA component decides whether to execute a query, and the µA component accelerates the execution of a specific query.

```
SELECT nation, o_year, SUM(amount) as sum_profit FROM (
   SELECT n_name as nation, EXTRACT(YEAR FROM o_orderdate) AS o_year,
      l_extendedprice*(1-l_discount)-ps_supplycost*l_quantity AS amount
    FROM part, supplier, lineitem, partsupp, orders, nation
   WHERE s_suppkey = l_suppkey AND ps_suppkey = l_suppkey
     AND ps_partkey = l_partkey AND p_partkey = l_partkey
     AND o_orderkey = l_orderkey AND s_nationkey = n_nationkey
     AND p_name LIKE '%[COLOR]%'
) AS profit GROUP BY nation,o_year ORDER BY nation, o_year DESC;
```

Listing 4.1: TPC-H `Q9`

the correct result for each query in these training data generation scenarios. Instead, the goal is to exercise the system to produce telemetry about its behavior as if it were executing in production.

Given this, we present the **Boot** framework for accelerating training data generation. The high-level idea of Boot is to exploit workload repetition in two ways while being transparent to the downstream ML components and aiming to preserve the accuracy of the ML models. The first method is to execute fewer queries on the DBMS by identifying redundant queries based on their high-level semantics and then reusing previously computed training data instead of running them again (*macro-acceleration*). For the remaining queries that the DBMS does execute, Boot injects special operators into their query plans that (1) dynamically identify redundant computations and then (2) intelligently short-circuits parts of their plans to make them complete faster (*micro-acceleration*). The combination of these techniques supports generating training data at a faster rate than regular query execution allows.

As shown in Figure 4.1, Boot integrates into a DBMS using two modules: (1) the **Macro-Accelerator** (MA) sits in between the DBMS's network handler and query planner and (2) the **Micro-Accelerator** (µA) is embedded in the DBMS's execution engine. Boot's design does not change the DBMS's interface for tuning components. A model training framework still connects to a Boot-enhanced DBMS over standard APIs (e.g., JDBC, ODBC) to execute a workload and collect training data. As such, Boot drops into existing modeling pipelines without any code change. But since Boot circumvents the DBMS's regular query execution, it is unsuitable for production environments. The gym deploys Boot on an offline clone of the production DBMS to avoid application errors.

We now provide an overview of Boot's accelerators. For this and the detailed descriptions in Sections 4.3 and 4.4, we use TPC-H [173] query `Q9` (Listing 4.1) as a running example. Executing `Q9` 1000× at scale factor (SF) 100 on PostgreSQL (pg15) takes 17 hrs. Enabling Boot reduces this time to 1 min with minor degradation in ML model accuracy. We defer discussing our experiments to Section 4.6.

4.2.1 Macro-Accelerator (MA)

Boot's MA module examines each query request as it arrives at the DBMS to determine whether executing it would produce novel training data (i.e., increase the diversity of query plans and operators executed). Novelty is necessary to avoid overfitting models to queries that access specific tables or with particular patterns.

As shown in Figure 4.1, when a SQL query arrives, ❶ the MA computes a fingerprint to identify whether it executed the query before. Since the MA is before the query planning stage, it computes this fingerprint on raw SQL strings. The MA strips out constants from the SQL to produce query templates (similar to prepared statements); this ensures that multiple invocations of a template using different input parameters are considered the same query [124]. In the `Q9` example in Listing 4.1, the MA extracts the constant from the `%[COLOR]%` input parameter and replaces it with a placeholder. We describe the fingerprinting process in Section 4.3.1.

Next, ❷ the MA looks up the query's fingerprint in a *result cache* to determine whether the DBMS executed a similar query with the same fingerprint. This cache maps each fingerprint to a record that contains (1) the query's output and (2) the DBMS telemetry generated while executing the query. The former is necessary because some workload replay and benchmarking tools assume the DBMS returns query results with a particular schema (e.g., typed columns). If the cache does not contain a matching query, the framework sends the request along in the DBMS for processing as usual. If the MA's cache contains a match, then the MA decides whether the system will learn anything new from re-executing it or if it should skip it. For those queries that the MA decides to skip, it returns the cached result and then records that it saw the query again. We discuss the MA's policies for skipping re-execution in Section 4.3.2.

4.2.2 Micro-Accelerator (µA)

Any query that bypasses the MA module then goes to the DBMS's query planner. At this stage, Boot's µA module injects its control methods into the query plan. These components wrap the plan's operators to monitor its run-time execution constantly. When the µA module detects that an operator's behavior has stabilized, it messages the corresponding wrapper to modify the operator's tuple processing (e.g., produce less output). We designed the µA module to support any query processing model (e.g., iterator, materialized, vectorized) and both push- and pull-based execution strategies.

Using the overview diagram in Figure 4.1 again, ❸ the µA embeds each physical plan operator (e.g., scans, joins) with a special wrapper operator that dynamically controls run-time behavior. This wrapper adjusts the sampling rate of its inner operator to change the number of tuples emitted. For example, the µA can change a scan operator to emit only 10% of the tu-

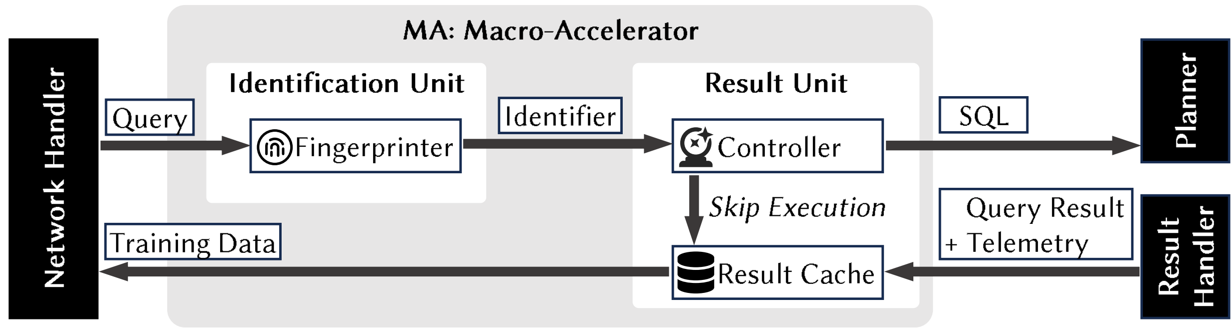


Figure 4.2: **MA Architecture** – Overview of Figure 4.1's MA.

ples it would otherwise produce. The wrapper can also halt an operator's execution when certain conditions are satisfied, such as if µA recognizes that it has enough training data for that operator.

Since the µA may cut off an operator's execution early, ❹ Boot scales each operator's telemetry to approximate what it would have been if the DBMS executed it entirely. For example, suppose that the µA cuts off an operator after processing only 10% of its expected rows. If Boot reports the operator's elapsed time, the operator appears to process all of its rows 10× faster than it did. Therefore, Boot scales the reported time by 10× to help prevent the behavior models from underpredicting queries' execution times.

The advantage of using a wrapper-based approach that modifies a query's physical plan is that it guarantees the DBMS will generate the same plan with and without Boot enabled. As we demonstrate in Section 4.6.8, some DBMSs alter a plan when using SQL-level sampling (e.g., `TABLESAMPLE`), producing different plans and degrading the behavior models' accuracy.

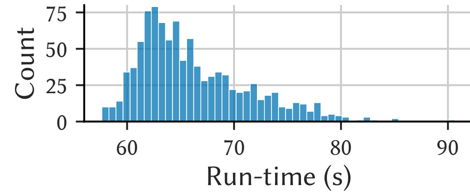
After the DBMS executes the µA-wrapped plan, it sends the estimated query result and telemetry to the MA module. The MA stores these in its result cache for future invocations of similar queries. The MA and µA modules are independent: if either component is disabled, the DBMS processes data with its regular non-accelerated components instead. We show in Section 4.6.2 that the accelerators enhance each other's effects to obtain up to 268× speedup.

4.3 Macro-Accelerator (MA)

We discussed in Section 4.1.3 why a DBMS must rely only on its optimizer when deciding whether to re-execute a query for its telemetry. Previous work showed, however, that DBMSs cannot achieve query progress estimates that are both general and robust [67]. In a worst-case scenario, a DBMS cannot do any better than guaranteeing that a query's progress is between 0% and 100%. Therefore, the MA module employs a heuristic approach that exploits query repetition. The key idea is to skip executing a query if the DBMS believes that the query looks "similar" enough to previous executions that the DBMS will not learn anything new, while adaptively testing the correctness of its beliefs. The functionality that the DBMS needs for this is to determine when an incoming query has been (1) seen before and (2) seen enough. Figure 4.2 shows how the MA achieves this with its Fingerprinter (Section 4.3.1) and Controller components (Section 4.3.2), respectively.

Plan Id	Count	Avg Run-time (s)
P1	95	76
P2	25	66
P3	819	65
P4	61	62

(a) Plan Distribution



(b) Plan Run-time

Figure 4.3: **Query Plan Behavior** – Distribution of TPC-H `Q9`'s plans and run-time across 1000 invocations (without Boot, SF 100, 1000 seeds).

4.3.1 Identifying Similar Queries

The MA module's Fingerprinter assigns identifiers to queries to combine them for training data generation. Because SQL is declarative, the Fingerprinter has many ways to identify a query (e.g., exact SQL text, query template [124], plan shape [130]). However, unlike regular query execution, exact results do not matter for training data. As we now illustrate, the Fingerprinter uses a relaxed comparison method for queries to achieve higher similarity rates.

We execute 1000 instances of Listing 4.1's `Q9` at SF 100 in 17 hr. Figure 4.3b shows the distribution of run-times and plans produced. Although the DBMS produced four different plans, the execution time for these plans exhibits clustering around the average run-time of the most frequent plan (P3). Furthermore, even if the DBMS never executes the slower plans, such as P1, the DBMS may still learn enough about their operators from instances in other plans and queries. For this reason, the Fingerprinter groups queries based on their templates. Doing so may map different query invocations to one cache entry (i.e., approximate matches). But the Fingerprinter's encoding strategy does not have to be static: if a query's parametric behavior is known, the encoder can map each parameter regime to a different fingerprint (e.g., replace `%[COLOR]%` with `[RED,PINK]` if selectivities are similar).

The Fingerprinter also links changes in the DBMS's configuration to the validity of previous executions. For example, adding a new index to a table invalidates the history for all queries that access that table. The Fingerprinter achieves this invalidation by including a hash of the DBMS configuration in each fingerprint. This mechanism enforces an explore/exploit trade-off, although the invalidation overhead depends on the tuning technique used.

4.3.2 Adapting to Query Variability

After the Fingerprinter identifies whether the MA module has seen a query before, the Controller then decides whether it has seen the query enough times to skip future executions. It uses a feedback-driven adaptive algorithm based on binary exponential backoff. For every Fingerprinter entry, the Controller stores a counter (c) that tracks how often it has seen a query to skip future executions. At run time when the Controller receives a query, it decrements the corresponding counter and then does one of the following:

Skip the Query ($c > 0$)**:** The framework does not forward the query for execution and instead returns a cached result. The Controller exploits the training data environment to synthesize results that align with historical data. Although many possible strategies exist (e.g., average the telemetry from previously executed plans, train a model to output similar telemetry [148]), the Controller defaults to repeating the last telemetry observed for that identifier (i.e., naïve forecasting [100]). This approach is fast, avoids the overhead of storing historical plans, and sidesteps environment drift issues.

Execute the Query ($c = 0$)**:** The Controller forwards the query for execution and analyzes the resulting telemetry. If the plan's run-time falls within two standard deviations of the historical mean, the Controller considers the new execution similar and exponentially increases the counter until a threshold. Otherwise, it resets the counter and clears the corresponding execution history.

We illustrate the skipping algorithm by supposing that `Q9` always takes its median run-time (64.8 s) with a threshold of 100 skips. The number of times that the Controller skips the query in between executions is $[1, 2, 4, 8, 16, 32, 64, 100, 100, 100, \dots]$, dropping the time for 1000 executions of `Q9` from 17 hr to 16 min. Should a `Q9` invocation exhibit new behavior, the Controller resets the skipping sequence to sample future instances more frequently. This behavior allows the Controller to adaptively decide a per-query *workload size* for training data collection (i.e., number of executions).

We use run-time to measure similarity to avoid storing and comparing against all executed plans. For each Fingerprinter identifier, the Controller maintains around 10 KB of state: (1) streaming Welford mean [186], (2) query plan, and (3) counter. Table 4.2 shows that workloads with millions of queries reduce to a few thousand plans, so the MA's typical total storage overhead is tens of MBs.

4.4 Micro-Accelerator (µA)

Skipping queries with macro-acceleration allows the DBMS to avoid executing redundant queries. But the DBMS still needs to execute each unique query at least once before it can cache its results, which is still prohibitively expensive. To handle such invocations, we present micro-acceleration techniques that exploit operator repetition (Section 4.1.3) to generate telemetry faster. As with the MA, the DBMS cannot provide optimal guarantees for using operator repetition [67] to accelerate telemetry production. Instead, the µA builds on operator progress estimation [113] to achieve its speedups with its Gate and Sampler submodules.

4.4.1 Stopping Repetitive Operators

The µA module exploits how query processing is performed at an operator's granularity (e.g., Volcano [89] model's `GetNext()`). For every operator, the µA's Gate component wraps this function to override tuple flow and measure repetition by tracking the rate of output tuple production. We note that existing DBMSs already wrap such functions for generating telemetry (e.g., `ExecProcNodeInstr()` in PostgreSQL), providing a natural integration point for the Gate.

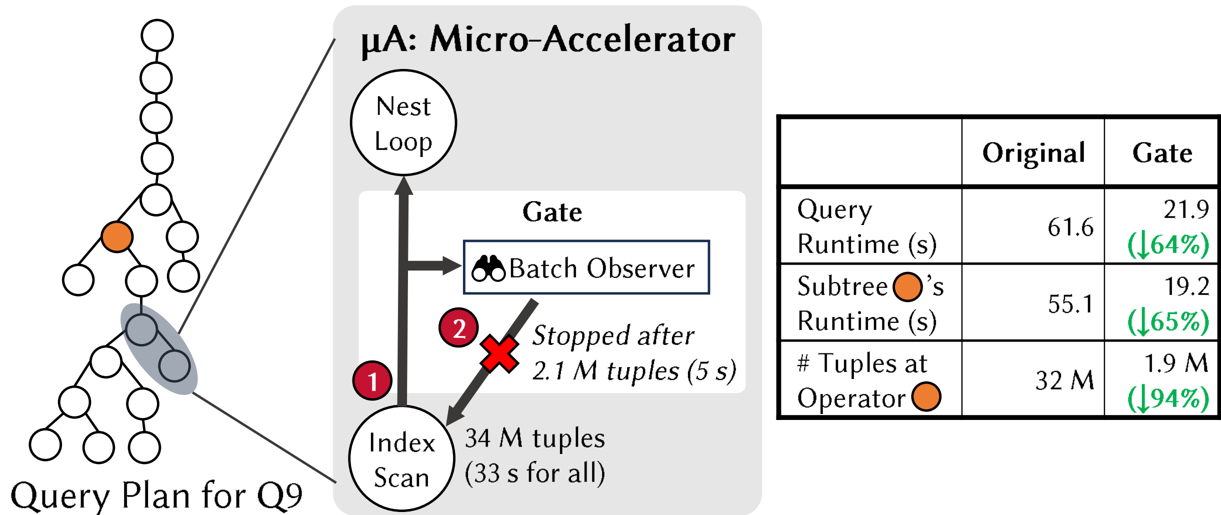


Figure 4.4: **Gate** – The Gate's architecture for Listing 4.1's query plan. The orange node is a hash join and the gray nodes are an index scan feeding into a nested loop join. The table shows the effect of introducing the Gate.

The challenge with an approach that tracks tuple flow is that the DBMS cannot accurately measure telemetry for individual tuples. The difficulty arises from timer overhead and resolution: an operator might take less than 1 μs per tuple. Therefore, the Gate collects telemetry on batches of tuples instead. For a given operator, it monitors the time taken to produce each output row and starts a new tuple batch when both of the following conditions hold: (1) the current batch contains at least 10% of the optimizer's estimated number of output tuples, and (2) the current batch's accumulated time is at least 1 s. The Gate considers the operator repetitive once the total time for its latest batch falls within two standard deviations of the historical mean. When the operator triggers this threshold, the Gate stops the operator from processing new input tuples. We present a sensitivity analysis for these parameters in Section 4.6.7.

We demonstrate the Gate's effects on a query plan for Listing 4.1's `Q9`. We first focus on the highlighted pair of operators in Figure 4.4 that depicts an index scan under a nested loop join. ❶ The index scan sends tuples to the Gate, which defaults to allowing tuples to pass through to the nested loop join. It also adds the tuple's telemetry to its current batch of tuples. When it creates a new batch, it checks whether it is similar to the last 20% of batches. If so, ❷ it stops admitting future tuples from the index scan. For this example, this stops after observing 2.1 M out of a possible 34 M tuples, reducing the time spent in the index scan from 33 s to 5 s. The Gate alone produces a 3× speedup in `Q9`'s run-time.

4.4.2 Sampling for Output Reduction

Whereas the Gate stops an operator's execution, the DBMS may only need to reduce the operator's output (e.g., to exercise less of a particular operator). Because query results do not matter for training data, the μA module samples each operator's output tuples to reduce run-time up in the query plan while maintaining representative behavior. It achieves this by installing a Sampler

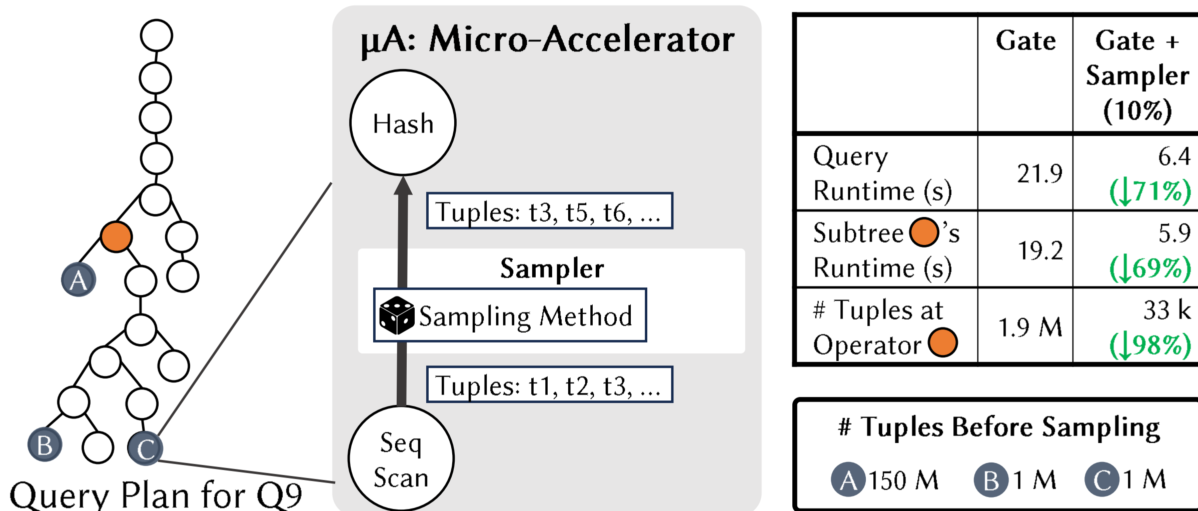


Figure 4.5: **Sampler** – The Sampler's architecture for Listing 4.1's query plan. The orange node is a hash join and the gray nodes are sequential scans. The table shows the effect of introducing the Sampler in addition to the Gate.

component on each operator's output tuple flow.

SQL already provides methods for reducing the number of tuples that certain operators produce. For example, `LIMIT` reduces the number of output tuples, and `TABLESAMPLE` reduces the cardinality of base relations. However, both of these methods are insufficient for our needs. `LIMIT` only applies to the query plan root, so the DBMS may have already done the work to compute the query (e.g., a long-running query that outputs a single tuple). The problem with `TABLESAMPLE` is that the DBMS may select a different query plan. We observed degenerate cases where a query's execution time in PostgreSQL went from 1 min to two days just by adding `TABLESAMPLE`! None of these methods work because the sampling must be hidden from the optimizer to avoid a change in plan. Moreover, the DBMS should be able to sample any operator in its plan (as opposed to only the root or base relations). Therefore, the Sampler exposes a similar interface as `TABLESAMPLE` but at the operator level.

To show how the Sampler effects Listing 4.1's `Q9`, we enable it using Bernoulli sampling at 10% on the operators that produce the most tuples (i.e., the sequential scans labeled as `A`, `B`, and `C` in Figure 4.5). Although it only sampled from these three operators, the effects on the rest of the query plan are significant: the orange node (a hash join) goes from producing 1.9 M tuples to 33 k tuples, and the overall query run-time decreased further from 21.9 s to 6.4 s.

4.5 Engineering

We now describe how we integrated Boot into a database gym as a PostgreSQL DBMS extension.

MA: We implement MA by hooking into the DBMS's traffic control layer. Because PostgreSQL uses the process-per-worker model, workers cannot easily share their execution history

(i.e., queries executed by one worker are not seen by another). Although PostgreSQL coordinates state across processes using shared memory, we avoid this because it interferes with query processing. Instead, MA manages its cache in an external key-value store (Redis).

μA: We implement μA by wrapping PostgreSQL's operators. Specifically, μA overrides every operator's `GetNext()`. While PostgreSQL executes a query, this override constantly analyzes the plan's telemetry to decide what to do. It performs its tasks by further swapping out the function pointer for the wrapper at run-time.

We initially built Boot as a standalone middleware that intercepted queries and polled the DBMS for its currently executing plans. We found that PostgreSQL's existing interfaces did not expose sufficient control. For example, we implemented μA's logic as SQL functions that took operator identifiers as input and polled to determine query progress. This model suffers from non-determinism and creates more work for the DBMS. Integrating Boot directly into the DBMS improves determinism and efficiency.

4.6 Evaluation

We now evaluate the Boot framework's ability to reduce training data generation times for autonomous DBMSs. For our analysis, we integrate Boot into the PostgreSQL (pg15) DBMS. We deploy the DBMS on an Ubuntu 22.04 LTS server with 2×20–core Intel Xeon Gold 5218R CPUs, 188 GB DRAM, and Samsung PM983 SSD, optimizing its configuration with PGTune [40].

We define our workloads and experiment configuration in Section 4.6.1. We then perform an end-to-end high-level analysis of Boot's modules in Section 4.6.2. Next, we identify inefficiencies in training data collection in Section 4.6.3. We describe how Boot addresses these inefficiencies with its MA in Section 4.6.5 and μA in Section 4.6.7. Lastly, we investigate sampling as a technique to further improve Boot's capabilities in Section 4.6.8.

4.6.1 Workloads

We use the database gym [119] to orchestrate the execution of the following workloads. Figure 4.6 shows the run-time distributions of the queries for each workload.

- **TPC-H:** This benchmark models a business analytics workload with eight tables and 22 query templates [173]. We chose this benchmark to represent a workload with a uniform data distribution. We use dbgen [1] to produce a total of 22k queries and execute them on scale factors 10 (∼19 GB with indexes) and 100 (∼192 GB with indexes).
- **DSB:** This is Microsoft's extension of the TPC-DS [172] workload that introduces additional challenges (e.g., complex data distributions, join patterns, skew), with a total of 25 tables and 52 query templates [78]. We use the official generator to produce a total of 10.4k queries and execute them on scale factors 1 (∼5 GB with indexes) and 10 (∼47 GB with indexes).

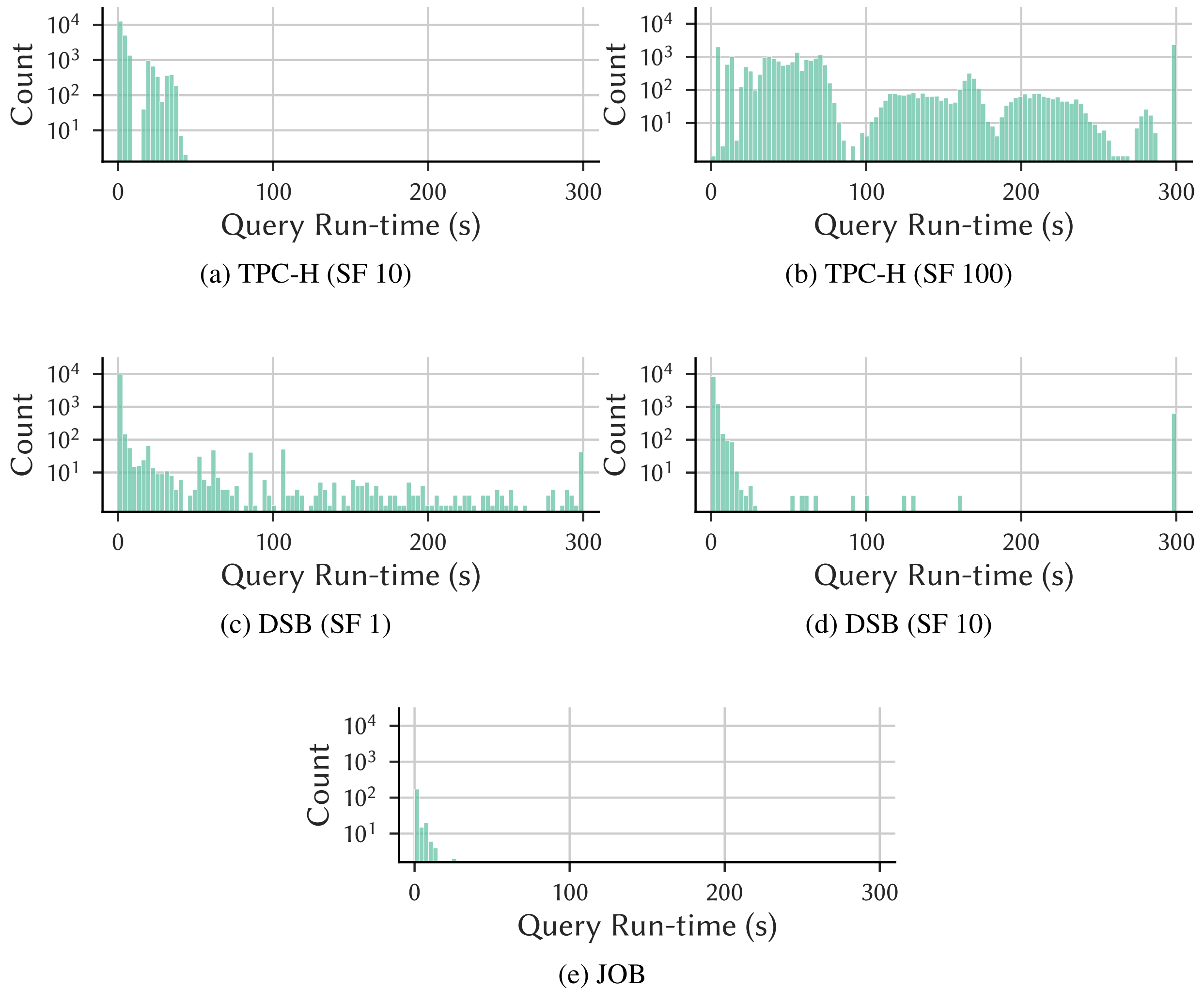


Figure 4.6: **Query Run-time Distribution** – Breakdown of elapsed time for each workload (without any acceleration).

- **JOB:** This benchmark uses IMDB and aims to stress the query optimizer's ability to pick a good join order [114]. It represents the worst-case workload for Boot (i.e., minimal repetition, small workload size, small dataset, short-running queries). There are 113 query instances and 21 tables (∼8.5 GB with indexes).

We set a per-query timeout of 5 min in all experiments. A timeout is necessary in a training data environment as the DBMS is trying to discover better configurations (i.e., it may not be optimally configured). We discuss timeouts further in Section 4.6.3.

To build behavior models using these workloads, the DBMS splits its training data into a train and test dataset as follows: for TPC-H and JOB, the DBMS trains on 80% of the seeds and tests on the remaining 20% [130]; for DSB, the DBMS uses separate seeds for train and test [78]. Next, the DBMS creates behavior models with AutoGluon [82], a state-of-the-art automated ML framework that automatically searches over hyperparameters and network architectures to create a model ensemble (e.g., gradient-boosted trees [69, 104], random forests, neural networks).

4.6.2 Speed-up vs. Accuracy Measurements

We first evaluate Boot's ability to accelerate a DBMS's training data generation process and how it affects the quality of the training data. Since there are no known techniques for measuring the quality of the generated data [46], we use behavior model accuracy as a surrogate metric instead. Thus, this experiment highlights the trade-off between reducing the execution time of queries versus producing models that accurately reflect the DBMS's internal operations.

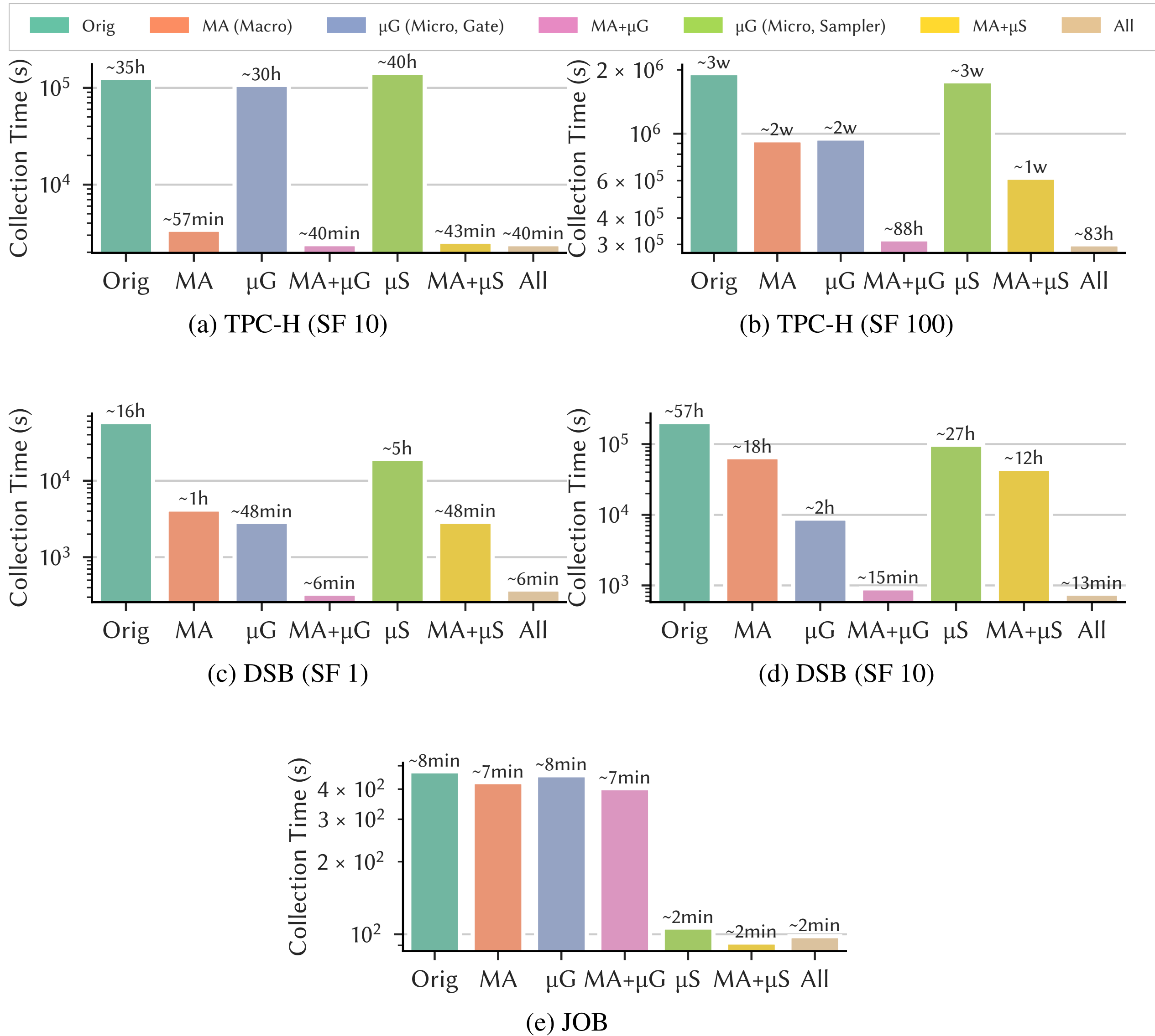
We run the workloads using Boot under seven configurations: (1) the default DBMS without acceleration (**Orig**), (2) only the MA module enabled (**MA**), (3) only the μA's Gate module (**μG**) enabled, (4) the MA and μA's Gate modules enabled (**MA+μG**), (5) only the μA's Sampler module (**μS**) enabled, (6) the MA and μA's Sampler modules enabled (**MA+μS**), and (7) all modules enabled (**All**). These configurations demonstrate the effect of reducing the number of queries (the MA module) and executing each query faster (the μA's modules) in different combinations. We structure our discussions below around the independent (1) MA, (2) μG, and (3) μS modules, followed by (4) the combined configurations (MA+μG, MA+μS, All).

We measure the time that the DBMS takes to generate telemetry for all the queries in a workload (i.e., collection time) as the end-to-end query latency. To measure model accuracy, we adopt two metrics from existing work: (1) *absolute error* [126] and (2) *factor error* [130]. Given a query q that has an actual latency $A(q)$ and a model M that predicts q's latency as $M(q)$, the absolute error is given by $|A(q) - M(q)|$ and the factor error is defined as $R(q) = \max\left(\frac{A(q)}{M(q)}, \frac{M(q)}{A(q)}\right)$. To understand whether these errors are caused by the models under- or over-predicting, we also visualize the error distributions of each model as $M(q) - A(q)$.

Collection Time: Figure 4.7 shows the collection time for the training data configurations across the workloads. We observe that the accelerators always speed up collection time, however, the extent of their benefit varies depending on workload characteristics.

The results in Figures 4.7a and 4.7c show that MA achieves a 13–37× speedup for the smaller SF workloads. However, increasing the SF reduces the speedup to 2–3× in Figures 4.7b and 4.7d. The first reason for this is that because MA does not increase the speed of query execution, it cannot help queries that timed out. Figures 4.6b and 4.6d shows more occurrences of such queries at the 300s mark. The second reason is that MA is more effective for workloads with lower variability in their query run-time. The standard deviation (std) of query run-time is lower in Figures 4.7a and 4.7c (8.42–29.9 s) and higher in Figures 4.7b and 4.7d (70.4–89.3 s). A reduction in std benefits MA because it decides whether to re-execute a query based on the similarity of its run-time to its previous executions. We next observe that the MA achieves only 1.1× speedup in Figure 4.7e for JOB. We expect this result because most of JOB's queries are only executed once (i.e., minimal repetition at the query level). We investigate the MA further in Sections 4.6.4 and 4.6.5.

Figure 4.7 also shows that while μG improves collection time relative to Orig, the increase is not as much compared to MA. For DSB, μG obtains a speedup of 20.3–23.2×, compared to only a 1.18–2.03× speedup for TPC-H. Such improvement depends on workload complexity because larger query plans introduce more opportunities for micro-acceleration (e.g., more leaf nodes, longer running operators). For example, μG sped up DSB's `query001` by 14× (117 s to



(a) TPC-H (SF 10) (b) TPC-H (SF 100)

(c) DSB (SF 1) (d) DSB (SF 10)

(e) JOB

Figure 4.7: **Collection Time** – The time to generate training data with different modules of Boot active (lower is better).

8 s), but only achieves a 2× speedup for most TPC-H queries. We next observe that Figure 4.7d is the only instance where µG is more effective than MA. In addition to query complexity, this is because µA enables the DBMS to complete queries that would otherwise time out (e.g., all `query032` invocations timed out after 300 s with MA but complete within 10s with µG). Lastly, Figures 4.7a and 4.7e show little improvement with µG alone. Orig's configurations show that the DBMS executed the 22k TPC-H queries in 1 day and 113 JOB queries in 8 min, which means the average query duration is approximately 5 s. Most queries did not run long enough to activate µG; we test more aggressive hyperparameters in Section 4.6.7.

These results show that the Sampler alone provides limited improvements on the collection time, with speedups ranging from 0.88–1.09× for TPC-H and 2.09–3.05× for DSB. The high

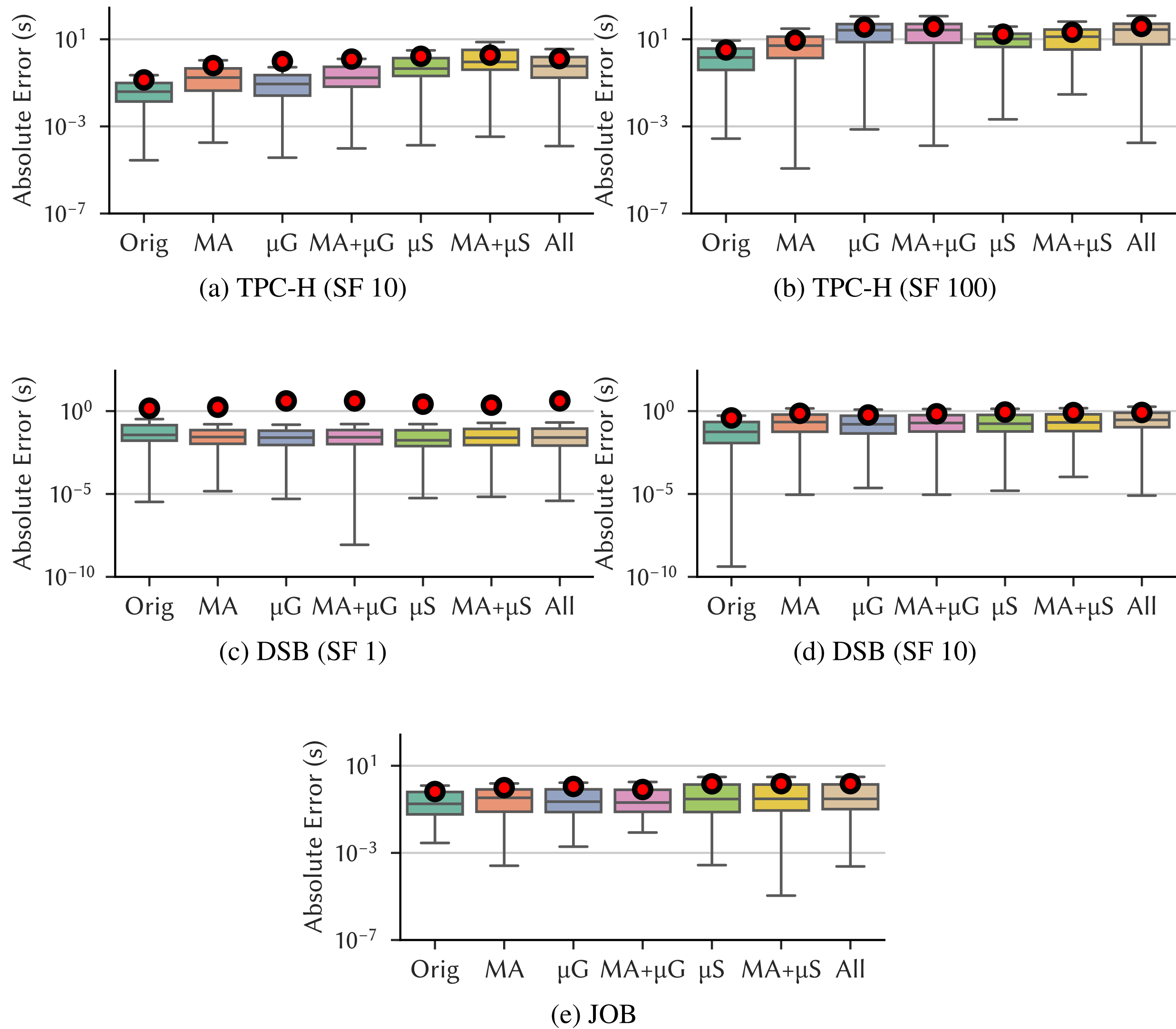
degree of query repetition mutes the Sampler's benefits for both workloads, and its sampling overhead makes TPC-H (SF 10) even slower. But JOB's low query repetition makes the Sampler the only effective technique for it. The Sampler obtains a 4.44× speedup because JOB is dominated by short index scans across multiple queries. These scans are too short for the Gate to accelerate, but a random sample of their tuples reduces the work while maintaining representative behavior. Because real workloads are more repetitive than JOB (e.g., 60% of Redshift's daily queries are exactly the same [193]), we expect other workloads to be more representative of Boot's performance.

The MA boosts the µS's limited efficacy by eliminating query repetition, with the MA+µS configuration obtaining speedups of 3–49× on TPC-H, 4–20× on DSB, and 5× on JOB. In comparison, the MA+µG combination obtains most of Boot's benefits for most workloads. It achieves speedups of 6–52× on TPC-H, 175–226× on DSB, but only 1.17× for JOB. The Gate is more effective than the Sampler because it dynamically stops the rest of an operator's execution, whereas the latter applies a fixed sampling percentage to an operator's output. With all accelerators active, All achieves the best of both worlds and obtains speedups of 6.2–52×, 154–268×, and 4.8× on TPC-H, DSB, and JOB respectively. Figure 4.7d also demonstrates that the modules enhance each other: while MA reaches 3×, µG obtains 23× and µS gets 2× speedup, all modules together obtain 268×. This combination benefits queries that are long-running or often time out. Using DSB `query032` as an example, suppose the DBMS invokes this query 100 times but they all time out at 300 s. MA does not help queries that time out, so MA takes 300×100 = 30000 s to complete. µG next reduces the query's run-time to 10 s, so µG takes at most 10×100=1000 s. Moreover, because the query now completes, MA executes exponentially fewer queries (i.e., 6 instead of 100) and All only takes at most 10×6 = 60 s (over 500× speedup) even before sampling. We examine the time spent in each operator and discuss additional timeout nuances in Section 4.6.3.

Absolute Error: Figure 4.8 shows the behavior models' absolute error when using training data from each configuration. The mean absolute error (MAE) of MA's models ranges from 1.1–4.6× that of the Orig models. The MA models are comparable to the Orig models because their telemetry was produced under similar conditions (i.e., MA only decides whether to execute a query). In contrast, the µG models are worse than Orig because µG terminates execution early and scales the telemetry. This reduced accuracy is reflected in µG's worse MAE of 7–11× for TPC-H and 1.7× for JOB. The latter is less affected because the Gate did not activate as much. For DSB, Figures 4.8c and 4.8d show 2.7× worse error at a smaller SF, but this error improves to 1.5× as the scale increases because Gate allows the DBMS to learn from queries that would otherwise time out.

Compared to the Orig models' MAE, the µS models are 5–12× worse for TPC-H and 1.7–2.3× worse for DSB and JOB, which is comparable to µG. The Gate's early operator termination has similar effects to sampling when it comes to MAE (i.e., observing a subset of execution is similar to sampling the entire execution).

Across all workloads, combining the MA with individual µA modules produces similar errors to the µA module alone. This result is because the MA does not modify query execution itself. When all MA and µA modules are activated, the All models exhibit similar error to the µG



(a) TPC-H (SF 10)

(b) TPC-H (SF 100)

(c) DSB (SF 1)

(d) DSB (SF 10)

(e) JOB

Figure 4.8: **Absolute Error** – The absolute error of models that are trained on the individual datasets (lower is better). The red circle shows sample mean and the whiskers extend to 1.5 interquartile range.

models. Compared to the Orig models, the MAE ranges from 9.4–11.9× for TPC-H, 2.1–2.7× for DSB and 2.29× for JOB. To contextualize these numbers, we sum the prediction errors for `Q1` in Figure 4.8b. Executing these `Q1` instances takes 3.7 hr. Consistent with other state-of-the-art models [126, 130], the Orig models are only off by 5 min, whereas the All models are off by 35 min. Although this is 7× the error, the speedup to obtain All's TPC-H models makes it an acceptable tradeoff (i.e., the DBMS obtains its first models in 4 days instead of 3 weeks, and these models predict 3.1 hr when the actual time is 3.7 hr for all `Q1` executions).

Factor Error: The results in Table 4.3 show how the model prediction error is distributed across queries in the form of factor error (i.e., the multiple that a query's prediction is incorrect).

Table 4.3: **Factor Error** – The factor error of the model predictions divided into buckets (lower is better).

	Factor Error			
	$1.1 \leq$	[1.1,2]	[2,5]	≥ 5
Orig	94%	6%	0%	0%
MA	50%	50%	0%	0%
µG	82%	18%	0%	0%
MA+µG	43%	55%	1%	0%
µS	25%	52%	22%	0%
MA+µS	4%	60%	33%	3%
All	22%	49%	28%	1%

(a) TPC-H (SF 10)

	Factor Error			
	$1.1 \leq$	[1.1,2]	[2,5]	≥ 5
Orig	82%	18%	0%	0%
MA	44%	51%	6%	0%
µG	10%	38%	33%	19%
MA+µG	9%	33%	37%	21%
µS	18%	69%	14%	0%
MA+µS	11%	69%	20%	0%
All	4%	41%	30%	25%

(b) TPC-H (SF 100)

	Factor Error			
	$1.1 \leq$	[1.1,2]	[2,5]	≥ 5
Orig	18%	45%	18%	19%
MA	16%	61%	17%	6%
µG	15%	64%	13%	8%
MA+µG	16%	57%	17%	11%
µS	21%	59%	15%	5%
MA+µS	18%	57%	16%	9%
All	17%	57%	15%	10%

(c) DSB (SF 1)

	Factor Error			
	$1.1 \leq$	[1.1,2]	[2,5]	≥ 5
Orig	52%	42%	4%	1%
MA	14%	64%	17%	4%
µG	18%	71%	10%	2%
MA+µG	15%	64%	16%	4%
µS	15%	63%	19%	3%
MA+µS	15%	59%	21%	5%
All	6%	53%	35%	7%

(d) DSB (SF 10)

	Factor Error			
	$1.1 \leq$	[1.1,2]	[2,5]	≥ 5
Orig	4%	63%	21%	12%
MA	5%	36%	39%	19%
µG	5%	35%	44%	16%
MA+µG	6%	55%	24%	15%
µS	3%	6%	12%	80%
MA+µS	1%	3%	4%	93%
All	1%	0%	1%	98%

(e) JOB

For almost all TPC-H queries, the error for MA's models is at most 2× because of the database's uniform data distribution. Such uniformity means that invocations of the same query but using different input parameters have similar performance. Therefore, even though MA executes fewer queries, the ones that it does execute are enough. However, both MA and Orig models have worse error for DSB and JOB, because these workloads are more complex than TPC-H.

This result is consistent with previous work [130] that found that some queries are harder to model than others. But the difference in the median factor error across all workloads is minimal: MA's error is 1.10–2.22×, and Orig's error is 1.02–1.54×.

Table 4.3 also shows that µG's models have comparable factor error to MA, with the exception of Table 4.3b where it is worse. This increase in error is because PostgreSQL's optimizer underestimates operator selectivities [114]. For µG's models, the median error ranges from 1.05–2.58×, which is up to 2.3× worse than the Orig models. In comparison, µS's models have median errors that range from 1.31–1.44× for every workload except JOB, which is 27×.

The MA+µG models have median factor errors that range from 1.12–2.52×, which is up to 2.5× worse than Orig. With the exception of JOB, we observe similar results for the MA+µS models: the median error ranges from 1.39–1.71×, which is up to 1.4× worse than Orig. The MA+µS models have 5× worse median error for JOB than the Sampler alone, which is up to 78× worse than Orig. Because JOB only has 113 queries, activating the MA removes about 10% (11) of the queries. This missing data is more important in JOB because most queries produce little meaningful data (i.e., the run-time in most operators rounds to zero milliseconds) and the low repetition makes it difficult to learn from other queries.

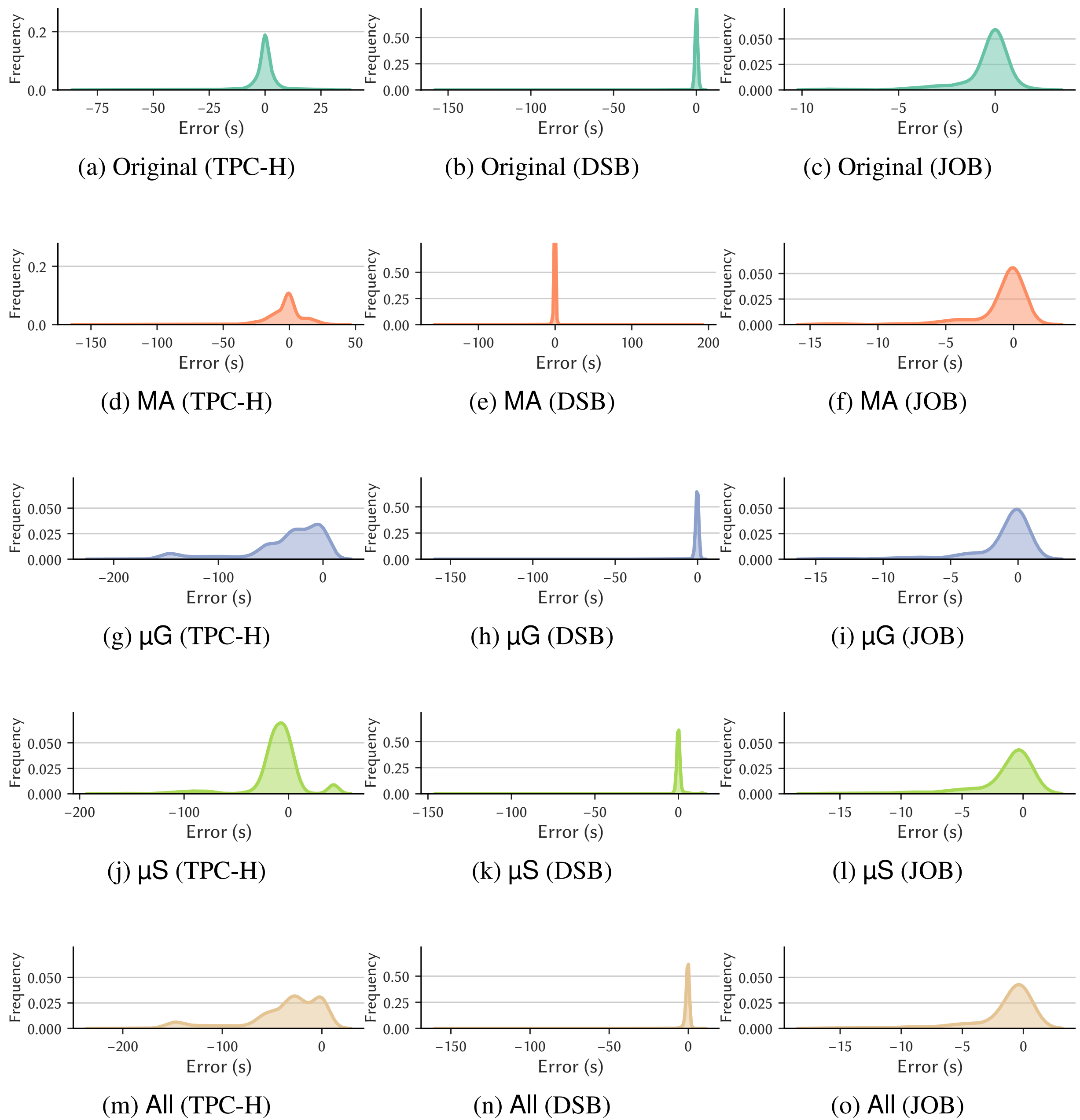
Excluding JOB, the All models have median factor errors ranging from 1.25–2.43×, which is up to 2.4× worse than Orig. JOB observes a median error of 165×, which is 108× worse than the Orig models. However, JOB's queries are shorter, limiting the practical impact of such mispredictions (e.g., query `10a`'s run-time of 0.226 s is underpredicted as 0.0018 s). We investigate the reason in the discussion on error distribution below. Recent research shows that less accurate models are competitive for tasks like index recommendation [209].

Error Distribution: Figure 4.9 shows the distribution of prediction errors for each training data generation configuration on the two larger datasets. The errors for Orig and MA are both uniform peaks with long tails centered at zero (i.e., most queries experience low error), where macro-acceleration has slightly more error. We expect this result in the absence of micro-acceleration as it matches existing work [126, 130] and we use newer modeling techniques.

Figure 4.9g shows the µG models consistently underpredict query latency for a subset of queries. This occurs more for long-running queries because µG expedites their execution and then scales up the telemetry based on optimizer estimates. However, PostgreSQL's optimizer underestimates the result size of multi-join queries [114]. This means that as the optimizer's estimation algorithms improve [197], µG's accuracy improves as well. µS suffers from the same underestimation problem, though its errors are more normally distributed from sampling. All inherits the same underprediction issue from µG and µS, with a slight increase to its error from MA as well.

4.6.3 Reducing Aborted Queries

The collection times in the above section include queries that the DBMS aborts on timeout. Because the DBMS only produces telemetry for completed queries, time is wasted when it executes queries that will abort. Boot reduces this waste by (1) MA probabilistically avoiding such queries and (2) µG allowing queries to finish that would otherwise abort, as described below.



(a) Original (TPC-H) (b) Original (DSB) (c) Original (JOB)

(d) MA (TPC-H) (e) MA (DSB) (f) MA (JOB)

(g) μG (TPC-H) (h) μG (DSB) (i) μG (JOB)

(j) μS (TPC-H) (k) μS (DSB) (l) μS (JOB)

(m) All (TPC-H) (n) All (DSB) (o) All (JOB)

Figure 4.9: **Error Distribution** – The distribution of model error for TPC-H at SF 100, DSB at SF 10, and JOB (closer to 0 is better). For example, an error of -20 s means the model predicted 20 seconds under the actual value.

We now revisit the results from Figure 4.7 by identifying the collection time spent on queries that the DBMS aborts. This analysis reveals the wasted work in the DBMS's training data collection and the extent to which Boot's modules reduce such waste.

Figure 4.10 shows the percentage of collection time that was wasted work. With Orig, the DBMS spends 36% (193 hr) and 29–91% (4.6–52 hr) of its collection time on aborted queries for TPC-H and DSB, respectively. These timeouts are caused by a small fraction of query in-

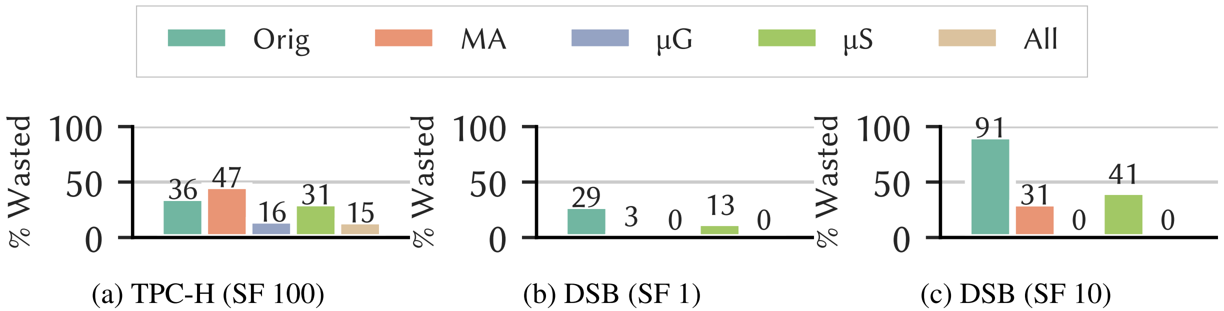


Figure 4.10: **Wasted Collection Time** – The percentage of time wasted executing queries that the DBMS aborts after 5 min. We omit TPC-H (SF 10) and JOB as they do not time out.

vocations from the workload: 2310 (10.5%) for TPC-H SF 100, 56 (1%) for DSB SF 1, and 626 (12%) for DSB SF 10. That is, less than 15% of queries are responsible for 29–91% of the collection time despite producing no telemetry. The DBMS cannot avoid these queries because it is difficult to know how long a query will take before running it. For example, the run-time for DSB's `query102_spj` ranges from 1–26 s based on its parameters.

The MA reduces waste when it substitutes invocations that will abort with historical executions that did not. But Figure 4.10a shows that the MA is unable to reduce TPC-H's waste. Most queries have only a small fraction of their invocations perform poorly relative to the mean (e.g., only 10% of Figure 4.3a's invocations have slower plans). Figures 4.9d and 4.9e show that the DBMS obtains enough data for its models without executing these poor plans. However, the degree of variability between TPC-H's plans was too low for the MA to reduce waste (i.e., the good and bad plans had similar abort behavior). In comparison, DSB's plans had high variability because of its skew and complexity, allowing the MA to reduce waste by 26% at SF 1 (Figure 4.10b) and 60% at SF 10 (Figure 4.10c). The MA obtains less improvement at the higher SF because more query invocations time out, preventing it from avoiding aborts.

The results in Figure 4.10a show that the μG reduces wasted work by 20% for TPC-H and eliminates wasted work entirely for DSB. The μG reduces waste through accelerating long-running queries that would otherwise abort. It does this by identifying operator repetition (see Section 4.1.3) and stopping such operators early, which we investigate further in Section 4.6.6.

Across all workloads, All reduced Orig's wasted work to 32% of the total time (250 hr to 80 hr). This reduction accounted for 30% (170 hr of 560 hr) of All's absolute time improvement.

4.6.4 MA: Different Encoding Strategies

Next, we analyze whether more advanced fingerprinting that includes query plan changes improves the MA's effectiveness. We run the experiments from Section 4.6.2 using a modified Fingerprinter that appends the query plan hash [130] to the query template (**MA.P**), contrasting against the default Fingerprinter (**MA**) and default DBMS configuration (**Orig**).

Comparing MA.P to MA, we observed similar collection times for DSB and JOB. The collection time of TPC-H increased from 57 min to 2 hr for SF 10 and decreased from 2 weeks to 1 week for SF 100 (i.e., MA.P is 2× faster at smaller scale factors and 2× slower at larger scale factors). Both the speedup and slowdown are caused by the MA's adaptive exponential skipping

Table 4.4: **Factor Error** – The factor error of the model predictions divided into buckets for MA.P (lower is better).

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	94%	6%	0%	0%
MA	50%	50%	0%	0%
MA.P	92%	8%	0%	0%

(a) TPC-H (SF 10)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	82%	18%	0%	0%
MA	44%	51%	6%	0%
MA.P	76%	21%	3%	0%

(b) TPC-H (SF 100)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	18%	45%	18%	19%
MA	16%	61%	17%	6%
MA.P	36%	47%	13%	4%

(c) DSB (SF 1)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	52%	42%	4%	1%
MA	14%	64%	17%	4%
MA.P	40%	48%	10%	3%

(d) DSB (SF 10)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	4%	63%	21%	12%
MA	5%	36%	39%	19%
MA.P	5%	40%	36%	19%

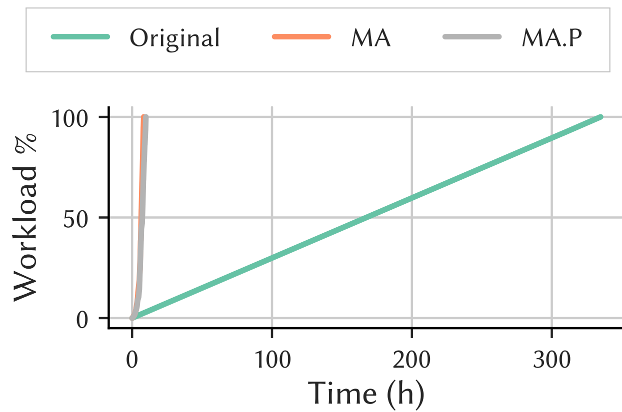
(e) JOB

algorithm. The MA resets a query template's skip counter when a query has different behavior. The MA.P avoids this and obtains speedups at SF 10, but it maintains more skipping sequences (i.e., skip counters are per plan instead of per query). Because the skipping is exponential, MA.P skips fewer executions, resulting in SF 100's slowdown.

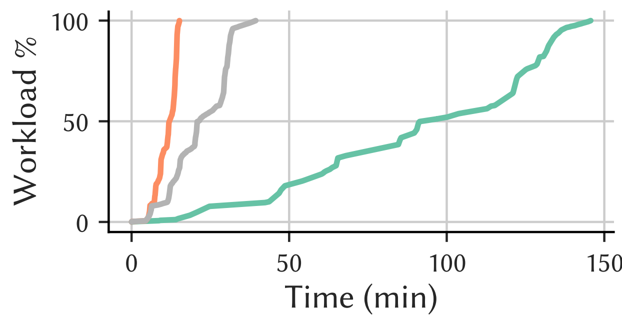
The results in Table 4.4 show that the factor and absolute error improves for all benchmarks with the MA.P. However, unlike the protocol-level MA, the MA.P requires invoking the optimizer and pausing execution after plan generation to check whether to continue execution or to return a cached result. Because optimizer calls are a bottleneck [60] and using plan data increases engineering complexity for modest accuracy benefits, we use MA in our evaluation. We defer more advanced fingerprinting [193] to future work.

4.6.5 MA: Executing Fewer Queries

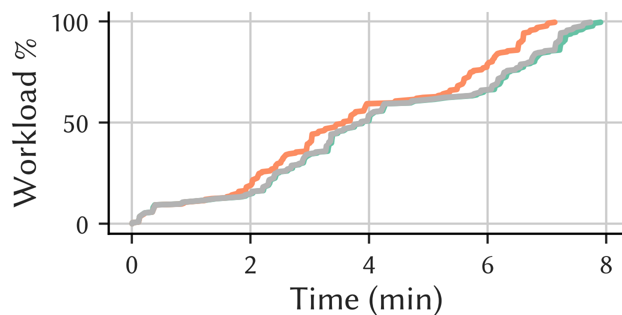
We now evaluate MA's effect on the workload completion rate as a function of the collection time. This analysis shows whether MA obtains its speedup across all queries or only a handful of queries. We analyze the data from Section 4.6.2 by measuring the number of queries completed (Workload %) as a function of elapsed time. We also plot the run-time distributions.



(a) TPC-H (SF 100)



(b) DSB (SF 10)

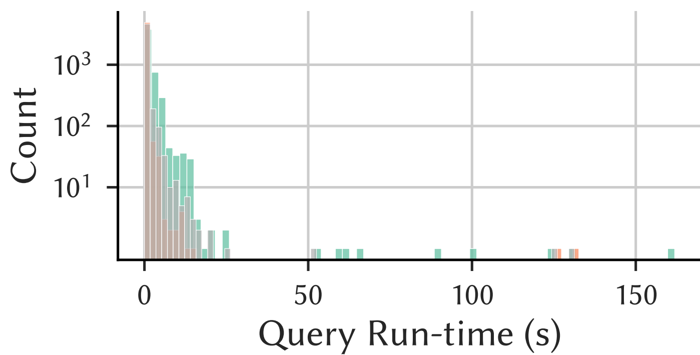


(c) JOB

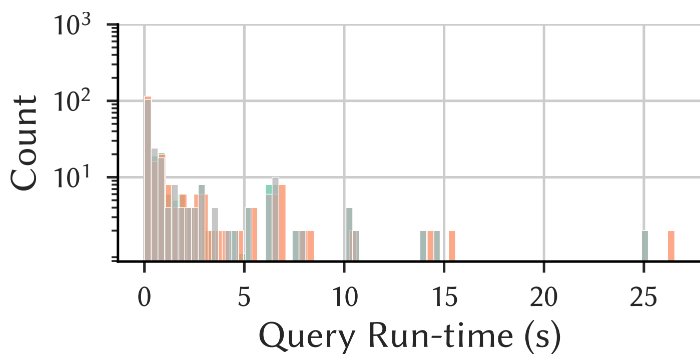
Figure 4.11: **Exponential Speedup (Completion Rate)** – Workload completion rate under MA, excluding timeouts.

Figures 4.11a and 4.11b show that MA's workload completion rate is higher than Orig. Recall that the only difference between these configurations is the number of queries executed: when MA has enough training data, it intelligently skips an exponential number of queries between executions. Therefore, MA's completion improvement also scales exponentially because it executes exponentially fewer queries. Figures 4.12a and 4.12b shows this effect on the distribution of query run-time. The DBMS executes queries with the same complexity under both configu-

(a) TPC-H (SF 100)



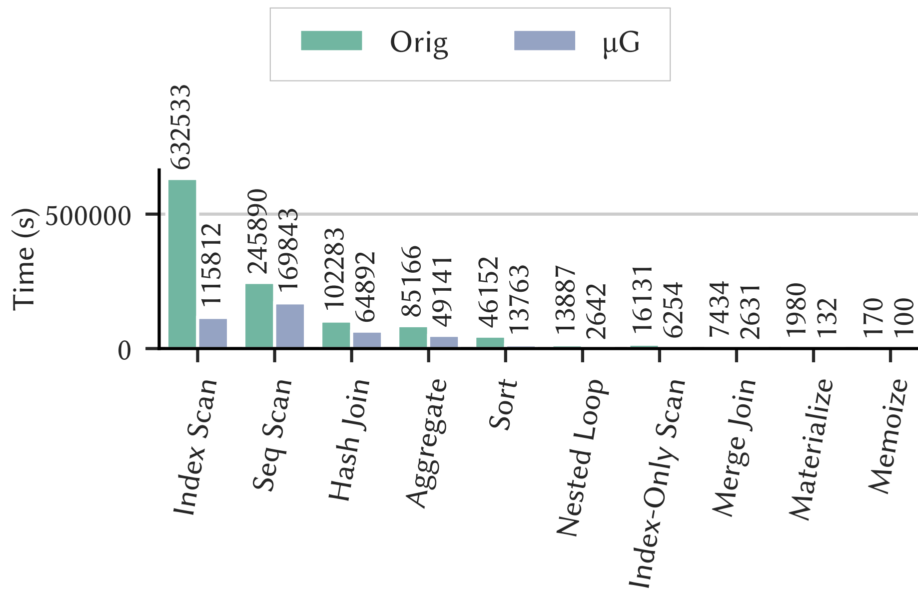
(b) DSB (SF 10)



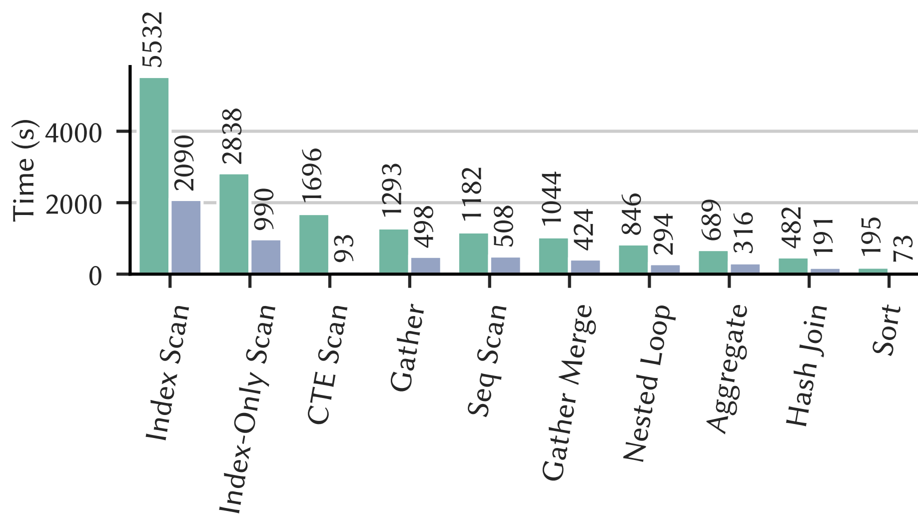
(c) JOB

Figure 4.12: **Exponential Speedup (Run-time)** – Run-time distributions under MA, excluding timeouts.

rations (i.e., similar histogram shapes), but the MA reduces the number of executions for each query (i.e., shorter heights on every bar). Figure 4.11b also shows the reduction in exponential skipping that Section 4.6.4 describes.



(a) TPC-H (SF 100) Timing



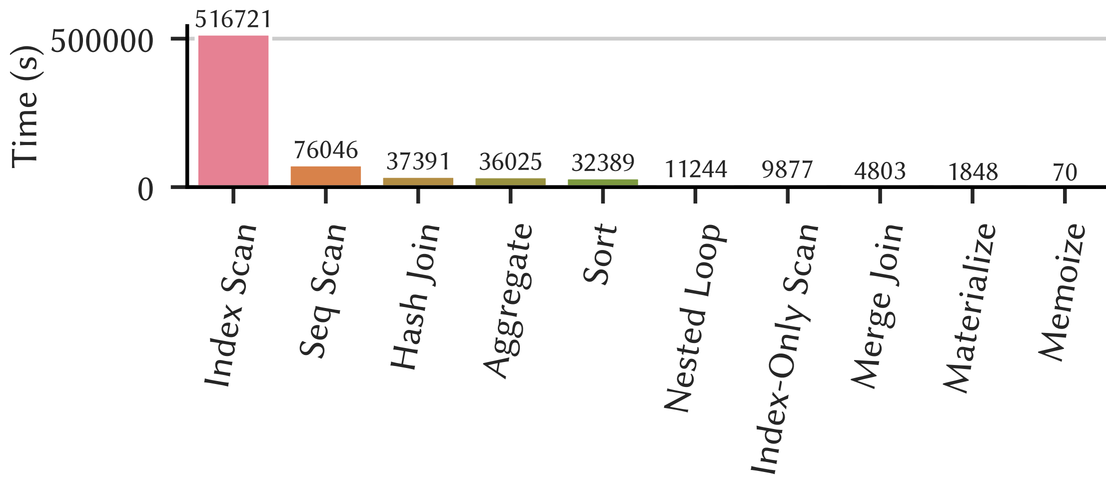
(b) DSB (SF 10) Timing

Figure 4.13: **Operator Time Breakdown** – The time spent in each operator (lower is better), excluding queries that timed out in Orig. We show the top 10 operators that Orig executes.

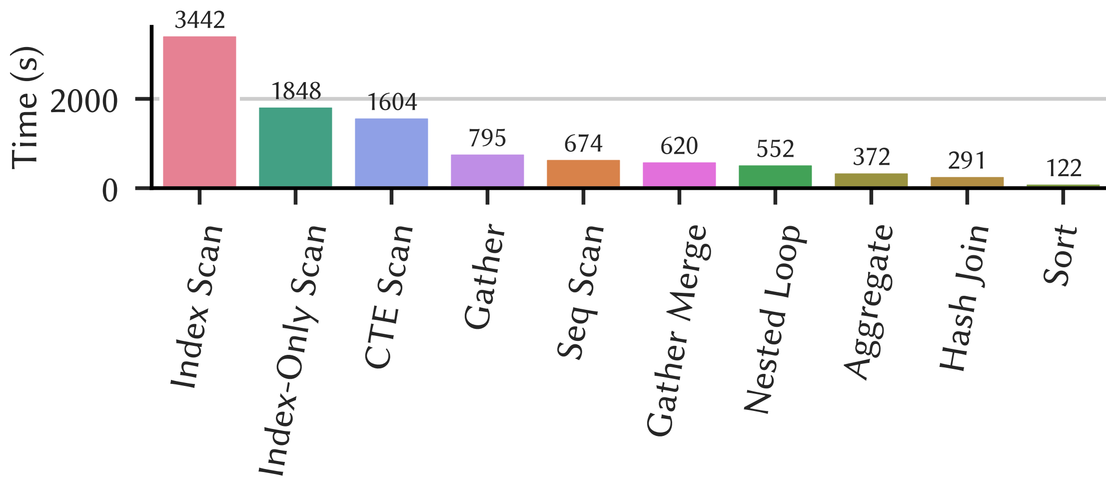
4.6.6 μG: Stopping Operators Early

We revisit μG's ability to reduce collection time by measuring the DBMS's duration in each operator. Our analysis seeks to identify which operators contribute significantly to the collection time and the extent to which Boot speeds them up.

We perform a fine-grained analysis of the data from Section 4.6.2 by breaking down the collection times in Figures 4.7b and 4.7d for the Orig and All configurations into individual op-



(a) TPC-H (SF 100)



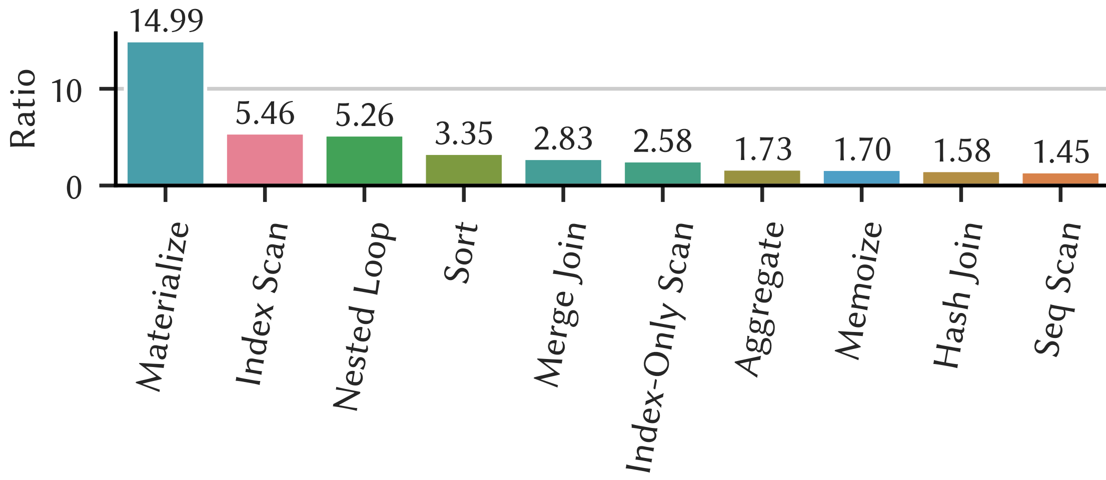
(b) DSB (SF 10)

Figure 4.14: **Speedup Analysis (Absolute)** – Each operator's absolute speedup because of μG in Figure 4.13 (higher is better).

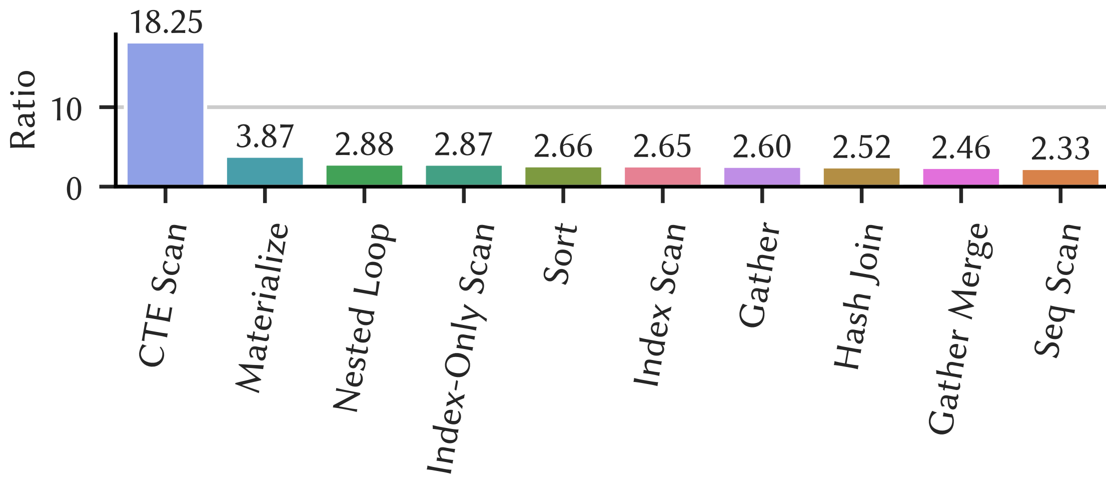
erators. We found that operator timings reported by PostgreSQL are inaccurate [12]. Therefore, we instrument every operator with additional timers [61].

Figure 4.13 shows the distribution of time spent in each operator for both Orig and All configurations. To ensure a fair comparison, we omit timed-out queries. The distribution of operator time is highly skewed, meaning that a few operators are responsible for most of the time spent. We observe that for both workloads (1) the μA reduces the run-time of every operator, (2) scans dominate the original query run-time, and (3) the μA achieves the highest absolute speedups on the longest-running operators.

Figure 4.14a shows that the top three absolute speedups for TPC-H are index scans (175 hr to 32 hr, 5.4×), sequential scans (68 hr to 47 hr, 1.4×), and hash joins (28 hr to 18 hr, 1.5×), whereas for DSB (Figure 4.14b) they are index scans (92 min to 35 min, 2.6×), index-only scans



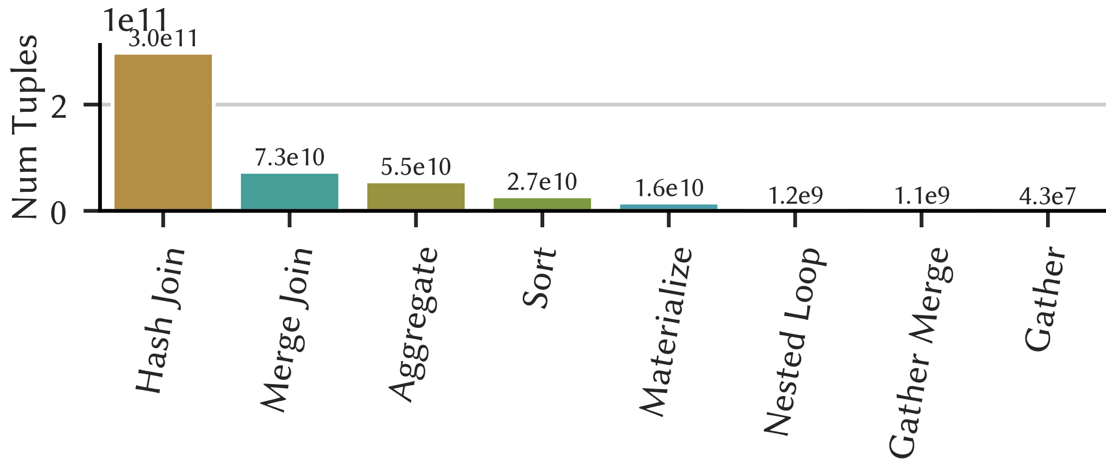
(a) TPC-H (SF 100)



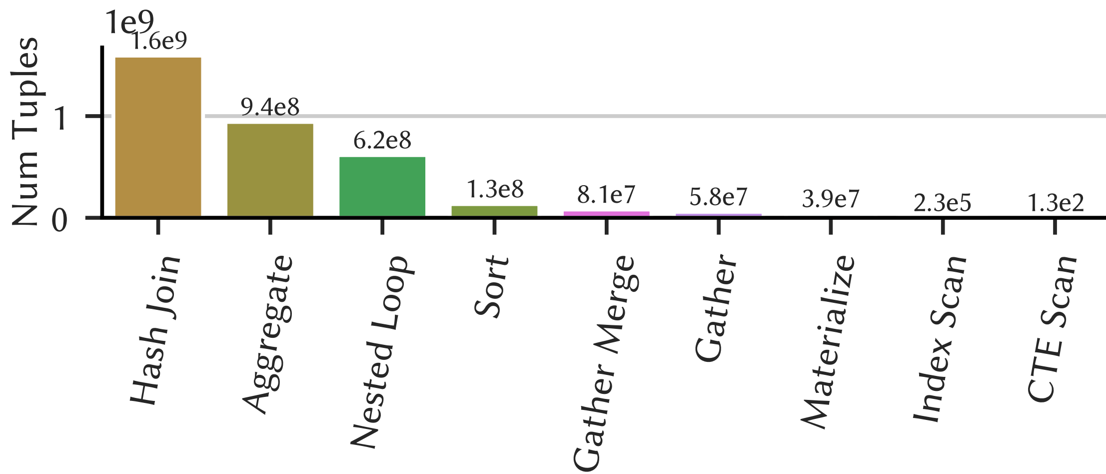
(b) DSB (SF 10)

Figure 4.15: **Speedup Analysis (Relative)** – Each operator's relative speedup because of μG in Figure 4.13 (higher is better).

(47 min to 16 min, 2.8×), and CTE scans (28 min to 90 s, 18×). The disk-based operators (e.g., scans) speed up because μG's early stopping reduces the I/O. The in-memory operators (e.g., hash join, aggregate) in Figure 4.14a obtain their speedup when μG reduces downstream tuples. Figure 4.16a shows that hash joins are 1.5× faster for TPC-H because they process $3{\cdot}10^{11}$ fewer tuples. The reduction in downstream tuples (Figures 4.17a and 4.17b) also explains each operator's relative speedup (Figures 4.15a and 4.15b). The materialize operator has high speedup in TPC-H (14.99×) because it processed 32.29× fewer tuples. Because μG dynamically decides when to stop an operator, the speedup of each operator is difficult to predict. Most speedups and reductions in downstream tuples are 1.4–3.5×.

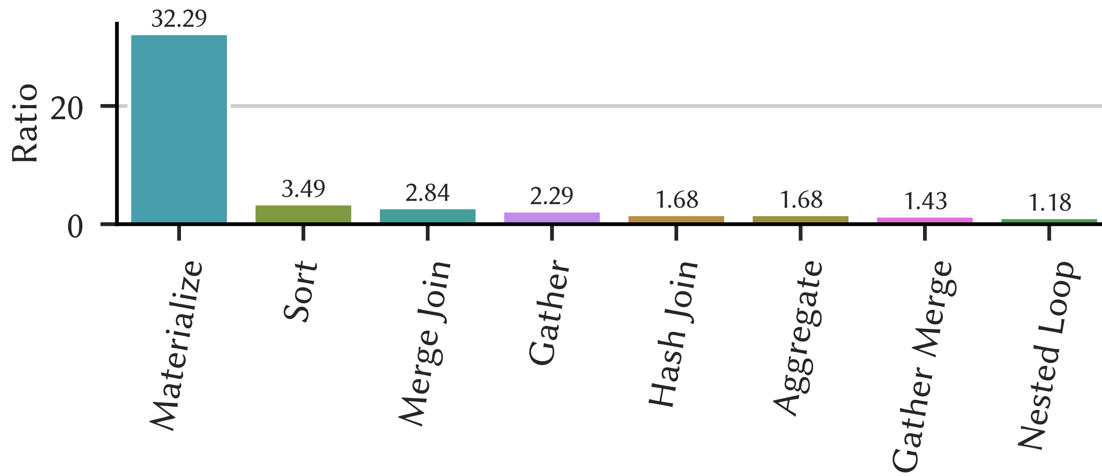


(a) TPC-H (SF 100)

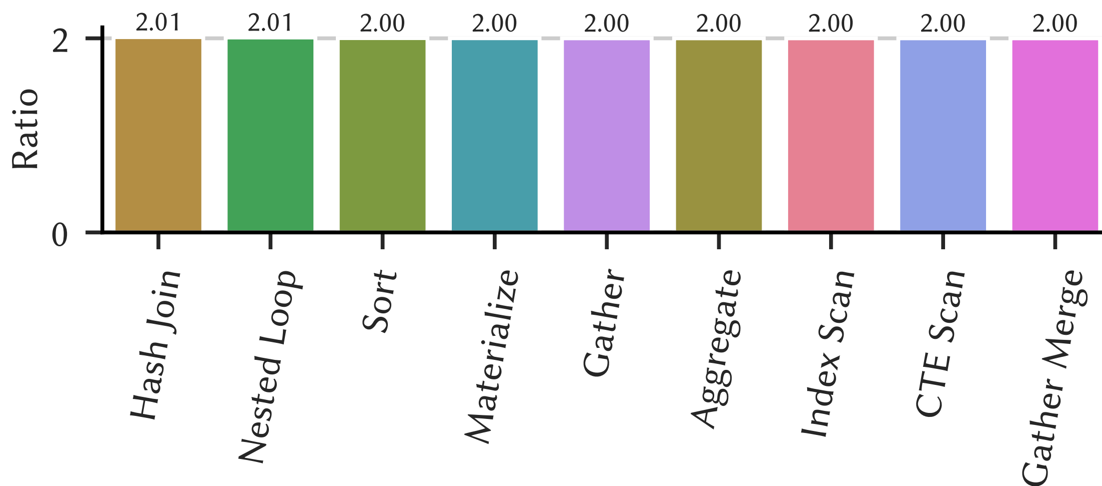


(b) DSB (SF 10)

Figure 4.16: **Tuple Reduction (Absolute)** – Each operator's absolute reduction in tuples processed because of μG in Figure 4.13 (higher is better).

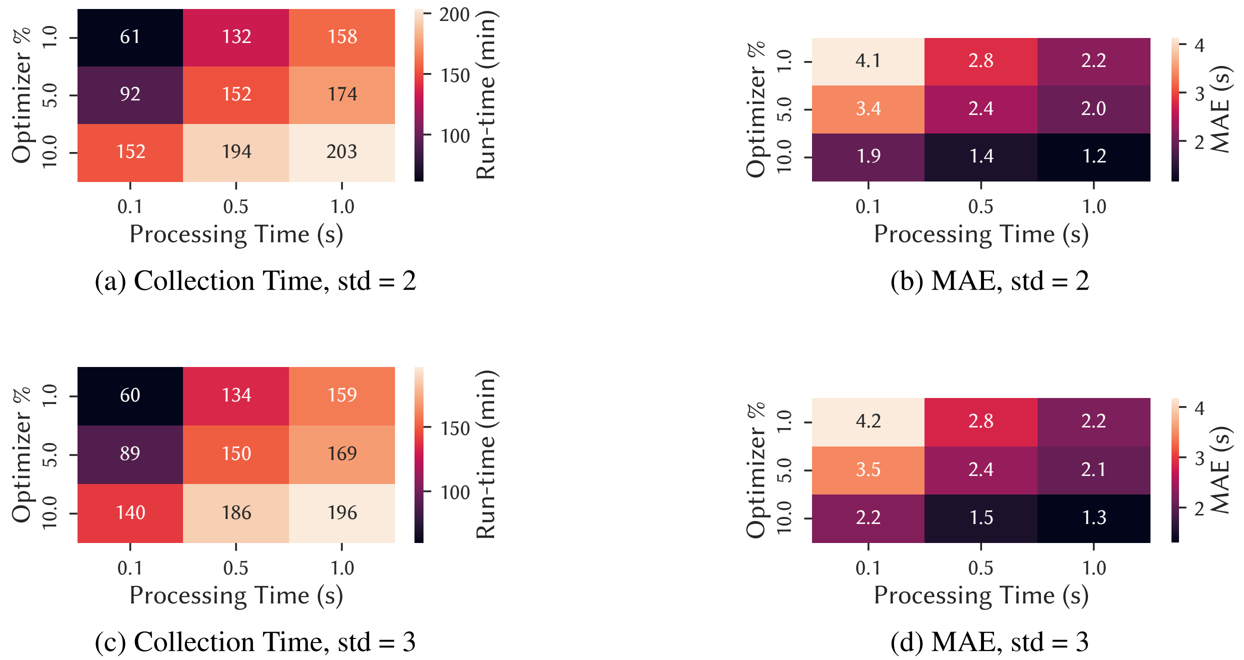


(a) TPC-H (SF 100)



(b) DSB (SF 10)

Figure 4.17: **Tuple Reduction (Relative)** – Each operator's relative reduction in tuples processed because of µG in Figure 4.13 (higher is better).



(a) Collection Time, std = 2

(b) MAE, std = 2

(c) Collection Time, std = 3

(d) MAE, std = 3

Figure 4.18: **Batching Sensitivity** – Collection time and MAE as we vary μG's hyperparameters on TPC-H (SF 10), 100 queries.

4.6.7 μG: Accelerating Short Workloads

We now revisit the experiment from Section 4.6.2 in which we observed μG achieving a speedup over Orig for every workload except TPC-H (SF 10) (Figure 4.7a). This analysis aims to discover why micro-acceleration does not work as well in some scenarios.

We conjectured that micro-acceleration is less effective in Figure 4.7a because the DBMS completed queries too quickly to detect operator repetition. Recall that the μG detects repetition by batching tuple telemetry, which it then uses to determine when an operator should stop processing new input. For every operator, μG creates a new batch whenever the current batch's (1) processing time exceeds 1 s and (2) the number of tuples is greater than the optimizer's estimated tuple count by 10%. μG then stops the operator if the new batch's timing is within two std of the mean on historical data. Therefore, three hyperparameters control μG's detection of operator repetition: (1) processing time, (2) optimizer %, and (3) std.

We perform a sensitivity analysis on μG's hyperparameters using 100 queries from TPC-H (SF 10). We measure the collection time and MAE as we sweep the processing time (0.1 s to 1 s), optimizer % (1% to 10%), and std (2 to 3). These ranges potentially allow for more opportunities for micro-acceleration to optimize this workload.

Figure 4.18 visualizes the three hyperparameters and their target variable (i.e., collection time or MAE) as heatmaps, where darker is better. Figures 4.18a and 4.18c show collection time (darker is faster), whereas Figures 4.18b and 4.18d show MAE (darker is lower). These heatmaps show that increasing the std has no appreciable effect on the collection time and MAE as the top row (std 2) has near-identical values to the bottom row (std 3). Varying the processing time and optimizer % achieves up to a 3× speedup at the cost of up to 3× higher error. For example,

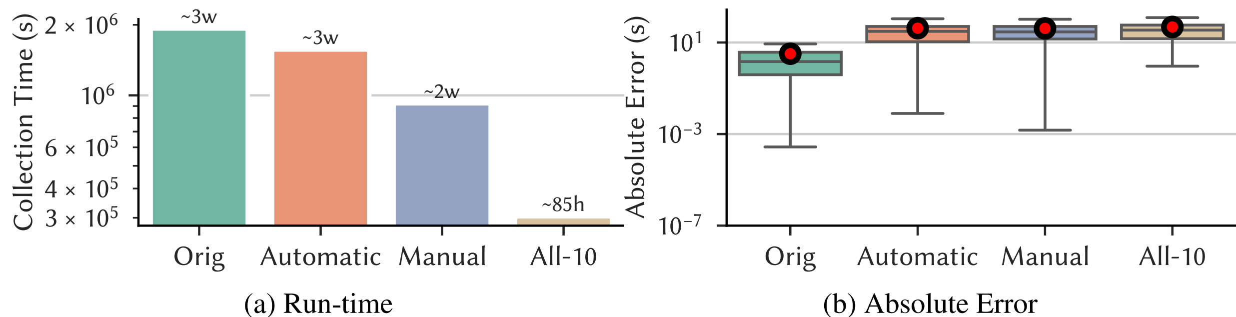


Figure 4.19: **Sampling Baseline** – Collection time and absolute error with TABLESAMPLE and Boot for TPC-H (SF 100). The red circle shows sample mean. The whiskers extend to 1.5 IQR.

reducing the processing time from 1 s to 0.1 s improves collection time by 1.33× (203 s to 152 s) but increases MAE by 1.58× (1.2 s to 1.9 s). Similarly, reducing the optimizer % from 10% to 1% improves time by 2.5× (152 s to 61 s) but increases MAE by 2.2× (1.9 s to 4.1 s).

This result verifies our hypothesis from Section 4.6.2 that µG is ineffective in Figure 4.7a because the queries' average run-time is too short. It also means tuning µA's hyperparameters allows for more aggressive tradeoffs between speedup and error. However, because an optimal progress estimate is impossible, we cannot prescribe a batch size that evenly divides an operator's progress. µG's reduced efficacy is due to the small dataset size, but Boot's combined techniques are still effective. Since Boot already achieves large speedups, we use conservative default settings of 1 s processing time, 10% optimizer cutoff, and two std for all experiments.

4.6.8 Output Sampling Analysis

We next investigate alternative sampling techniques and the Sampler's tradeoffs between collection time and error.

We evaluate Boot's sampling against the DBMS's built-in table sampling (through the TABLESAMPLE SQL modifier) even though Section 4.4.2 outlines qualitative reasons against such an approach. We collect training data for TPC-H (SF 100) as it represents the best case for TABLESAMPLE with its uniform data distribution. We evaluate four training data generation configurations. First is the baseline with neither sampling nor Boot active (**Orig**). The next configuration uses TABLESAMPLE at a rate of 10% on all tables in the query (**Automatic**). Because we found TABLESAMPLE to cause problems with PostgreSQL's optimizer and generate slower plans for some queries, we also manually modify every SQL query to add the TABLESAMPLE clause only for the relations where there is a benefit (**Manual**). Lastly, we collect training data with Boot using a 10% sample rate for all scan operators (**All-10**).

Figure 4.19 shows that both TABLESAMPLE methods reduce collection time by up to 2.1×, though this requires the user to rewrite every query template by hand (Manual). All-10 achieves a 3× faster collection time than both sampling methods with comparable error rate (MAE increases from 12.7× to 14.7×). As described in Section 4.4.2, sampling scans improves the collection time by reducing the number of tuples that the DBMS processes downstream. Moreover, when Sampler only targets less selective operators (e.g., sequential scans), it reduces the risk

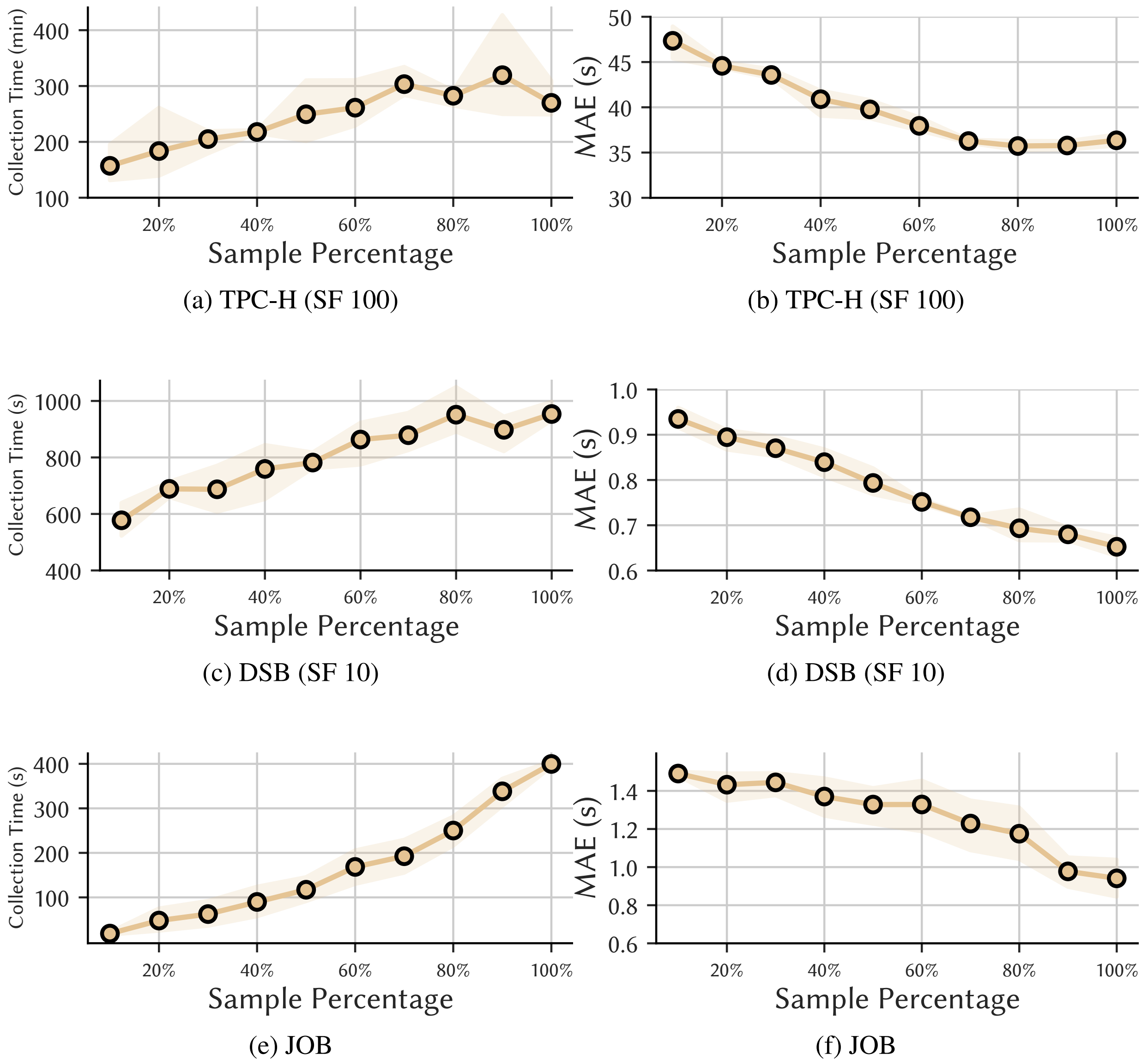


Figure 4.20: **Sampling Rate** – The collection time and MAE when MMA is enabled and sequential scans are sampled at different rates. We exclude queries that timed out for fairness (affecting 4.55% of TPC-H).

of eliminating tuples that were uniquely important for exercising DBMS behavior and therefore limits the effect of sampling on the MAE.

To understand the effect of the sampling percentage, we sweep the rate across trials with different random seeds. Figure 4.20 shows the collection time and MAE as a function of the sampling rate. Their scaling is approximately linear (e.g., for TPC-H, increasing the sampling rate by 10% increases collection time by 10–20 min and decreases MAE by 1–2 s). This demonstrates that the Sampler allows Boot to obtain models even faster, albeit at a proportional model quality cost. Boot defaults to sampling at 50%.

4.7 Conclusion

Autonomous DBMSs use behavior models to evaluate the benefit of candidate configurations while avoiding the workload execution overhead. Collecting the training data required to construct these models is overly time-consuming, making integrating such models into the DBMS's tuning feedback loop infeasible. We introduce two acceleration techniques to expedite the training data collection process by leveraging the unique characteristics of the training data environment. We show how to apply these techniques with our framework Boot that drops into existing database gym pipelines. Experiments across multiple workloads show that our acceleration techniques are well-suited for bootstrapping an autonomous DBMS's models as they produce models in less time (hours instead of weeks) with only a moderate degradation in model accuracy.

Chapter 5

Generalizing Training Data Across Heterogeneous Deployments

As described in Chapter 3, existing DBMS optimization tools rely on models of the DBMS's behavior [126] to generate their tuning recommendations. Such models allow tools to estimate the benefit of candidate configurations without testing them on the DBMS itself, which is computationally expensive. The quality (e.g., accuracy) of these models depends on the training data used for their construction. Because this training data is gathered through workload execution, it implicitly encodes assumptions about the environment in which it was generated. Subsequently, these models learn specific aspects of the training environment (e.g., schema, dataset, version) and struggle to generalize to new environments.

The problem of environment-dependent training data is especially pronounced in database tuning. Unlike other ML domains that have a stable "ground truth", the behavior of a DBMS changes frequently as its software and hardware evolve to incorporate new features (e.g., JSON tables [29]), performance optimizations (e.g., index build parallelism [5], better scan I/Os [11]), security updates (e.g., vulnerability patches [38]), and other improvements. One naive solution for handling this environment dependency is to incorporate the DBMS's software and hardware characteristics as behavior model input features. However, such an approach is computationally prohibitive: it requires obtaining training data that covers the multitude of software and hardware combinations for model calibration. Acquiring this data requires loading and executing workloads across thousands of different environments, which is clearly infeasible. In practice, most models require retraining from scratch whenever their users perceive performance degradation.

The challenges associated with predicting DBMS behavior across heterogeneous environments extend beyond the scope of automated database tuning, sharing similarities with the problem of database upgrades. Although DBMS vendors regularly release upgrades that promise to improve performance and reduce costs, most organizations choose to deploy older DBMS versions [28] and delay upgrades for as long as they can [26]. Upgrade hesitancy exists for a variety of reasons, chief among them the operational risks of upgrading (e.g., downtime [15]) and the fear of performance regressions [27]. In response, the database community has developed processes (e.g., checklists [24]) and tools (e.g., MySQL's Upgrade Checker Utility [25]) to mitigate the operational risks of upgrading. However, such approaches require humans to manually perform the upgrade and execute the actual workload on the new environment [19],

troubleshooting any regressions that arise [21]. When regressions occur, a DBMS may support pinning mechanisms to mitigate them (e.g., query hints [20], plan forcing [8, 23]), but such stop-gap solutions prevent future upgrades from fixing the issue. Furthermore, even if an organization regrets their upgrade and decides to rollback to the previous DBMS version, downgrading is as often operationally burdensome as upgrading itself [36].

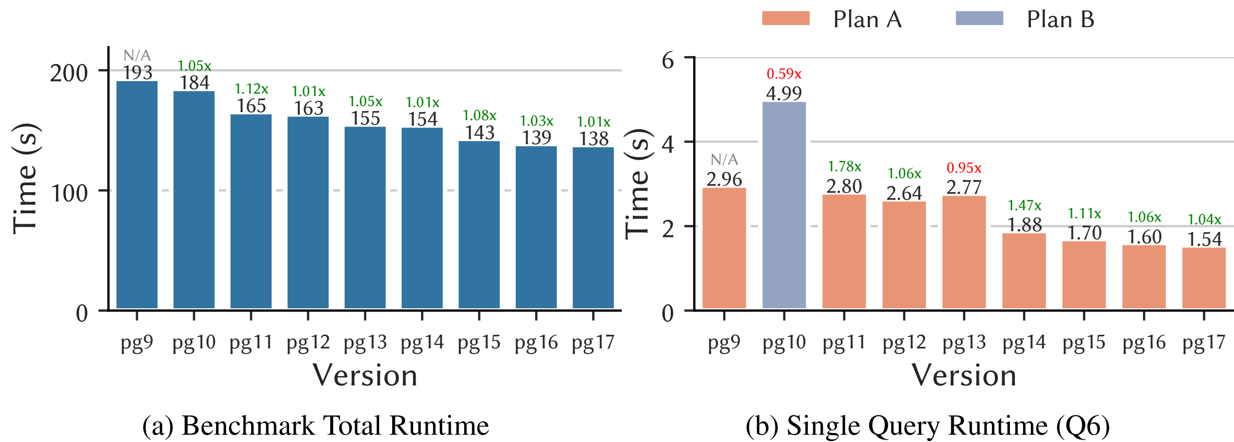
Despite these challenges, behavior models and database upgrades alike must consider the substantial performance impact of changing the *software* or *hardware* for a specific DBMS deployment's workload. Examining software upgrades in isolation, queries from the Join Order Benchmark (JOB) [114] exhibit huge performance variations over the last five years of PostgreSQL releases: an individual query experiences up to 21× speedup or up to 31× regression in p90 latency between consecutive versions [129]. Hardware upgrades introduce even more complexity in reasoning about the trade-off [10] between performance and cost. For example, executing JOB on PostgreSQL version 17 requires 536, 179, or 154 seconds depending on whether a Small, Medium, or Large machine is used (see Section 5.5 for machine details), which is up to a 3.5× difference in performance. However, because the Large machine is 10× more expensive than the Medium machine for only 1.16× speedup, a human may decide to go with the Medium machine for its price-performance. But human operators are expected to discover such tradeoffs by benchmarking their workloads on all the different candidate environments. Given the impractical cost of doing so, many DBMS deployments use sub-optimal software versions and either waste money through overprovisioning or suffer from underprovisioning.

What is needed by both behavior models and database upgrades is a tool that predicts the performance impact of an upgrade **before** it is deployed. Given this, we now present **Dr. Database, MD** ($^{D}_{R}$·DB), an upgrade recommendation tool that adapts existing behavior models to predict the performance impact of software and hardware upgrades. $^{D}_{R}$·DB is usable by both humans and autonomous DBMSs. It applies parameter-efficient fine-tuning techniques [127] to capture a DBMS behavior model's *implicit* features into independent *adapters* that are swappable at run-time. Furthermore, these adapters are composable across different dimensions (e.g., software, hardware), significantly reducing the training data collection required to achieve good coverage. Our empirical results show that $^{D}_{R}$·DB improves median model performance by up to 31× and mean model performance by up to 108×. Our adapter-based approach produces models that are typically no worse but often significantly better than the baseline models generated by existing methods, which are instance-specific and difficult to generalize across environments.

This chapter makes the following contributions: (1) highlighting the impact of issues in software and hardware DBMS upgrades, (2) presenting a tool that provides automated upgrade recommendations by extracting the assumptions of existing behavior models into independently distributable and run-time swappable adapters, (3) demonstrating the effectiveness of $^{D}_{R}$·DB in providing performance-related upgrade recommendations, and (4) evaluating techniques for composing adapters across different dimensions.

5.1 Background

We motivate our work by examining how users upgrade their DBMS deployments today. We first discuss the different types of upgrades. We then describe how existing upgrade-related



(a) Benchmark Total Runtime (b) Single Query Runtime (Q6)

Figure 5.1: **Software Upgrades** – TPC-H performance across consecutive PostgreSQL versions. The colored bar labels are relative to the previous version (green is speedup, red is slowdown).

problems are handled with post-upgrade mitigations that do not solve the fundamental issue of unpredictable upgrade behavior, making it hard to justify upgrading. Lastly, we discuss existing research on modeling DBMS behavior.

5.1.1 Upgrading

We begin by describing software upgrades and then discuss the more challenging problem of DBMS hardware upgrades. Next, we discuss how software-hardware interactions make it difficult to consider them in isolation (i.e., they must be optimized together).

Software Upgrades: Actively maintained DBMSs are updated through periodic releases of (1) *major versions* that may introduce breaking changes to support new features, and (2) *minor versions* that provide backward-compatible fixes and optimizations. For example, PostgreSQL releases a new major version every year [39]. Cloud vendors force automatic minor upgrades during maintenance windows [26] or when end-of-life support is reached [44], but organizations are often required to manually perform major upgrades due to potential compatibility issues.

But despite new features and performance optimizations, many organizations are reluctant to upgrade major versions [26]. For example, even though PostgreSQL v9.6 (pg9.6) was released in 2016 and reached end-of-life status in 2021, commercial providers continue to offer extended support until 2028 [34]. One justification for running decade-old DBMSs is that the upgrade process has an asymmetric risk-reward profile: performance improvements are only known post-upgrade, whereas upgrade failure risks are widely known [207]. Although there are other externalities in determining whether to upgrade (e.g., operational constraints), they are difficult to quantify and thus are outside the scope of this paper. Our focus is on the objective measure of query performance.

To demonstrate both the benefits and pitfalls of software upgrades, we compare TPC-H [173] benchmark performance between consecutive PostgreSQL releases from 2016 (pg9) to 2024

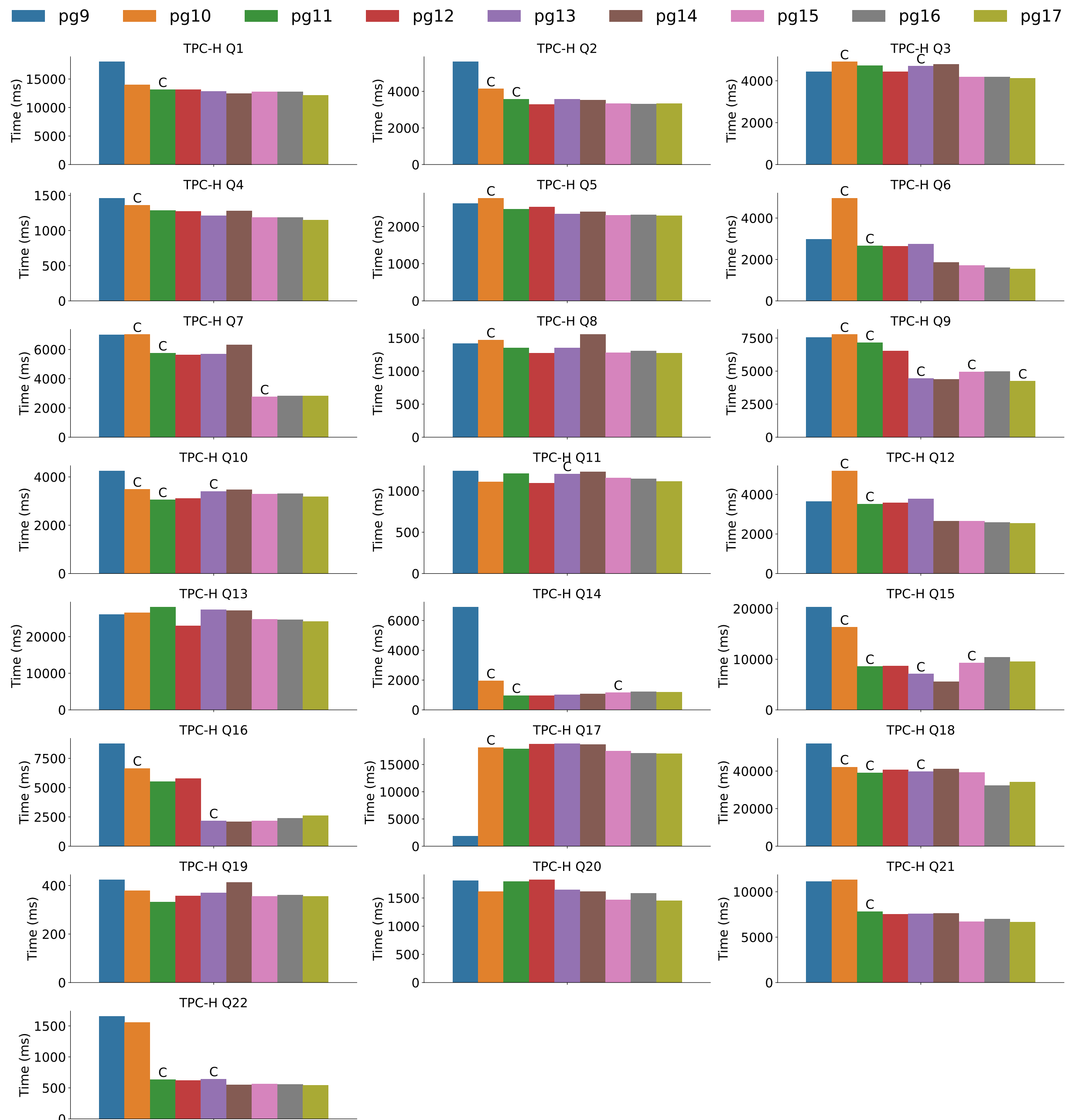
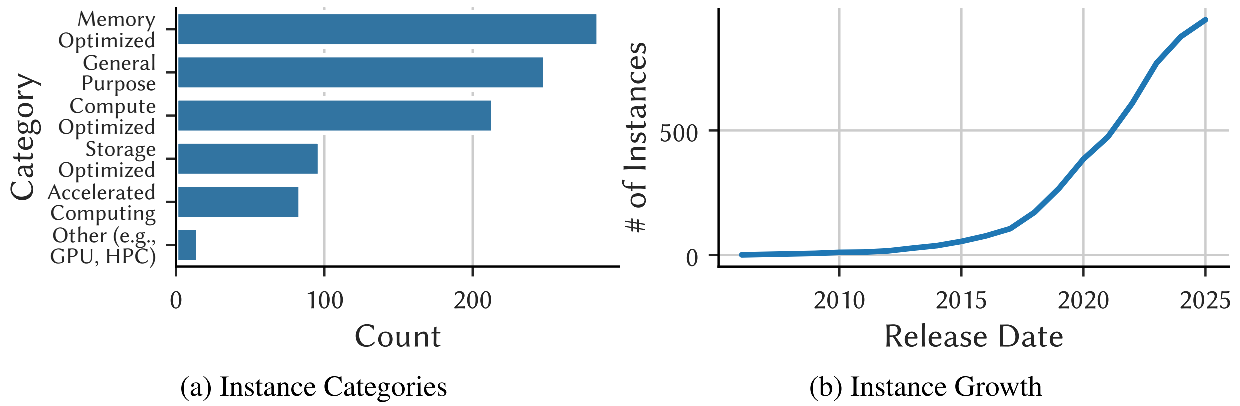


Figure 5.2: **Software-Induced Performance Variations** – The performance of TPC-H queries across different PostgreSQL versions on a server-class machine. A C above a bar indicates that the query plan changed from the previous version.

(pg17) on identical hardware and data. Figure 5.1a shows that the overall performance of TPC-H improves over new versions of PostgreSQL, achieving a cumulative 1.4× speedup from pg9 to pg17. However, analyzing individual queries reveals that TPC-H's per-query behavior does not always get better. For example, even though the overall benchmark is 1.05× faster after upgrading to pg10, Figure 5.1b shows TPC-H Q6 is 1.6× slower because its query plan changed.



(a) Instance Categories

(b) Instance Growth

Figure 5.3: **Hardware Choices** – AWS EC2 instance categories and growth.

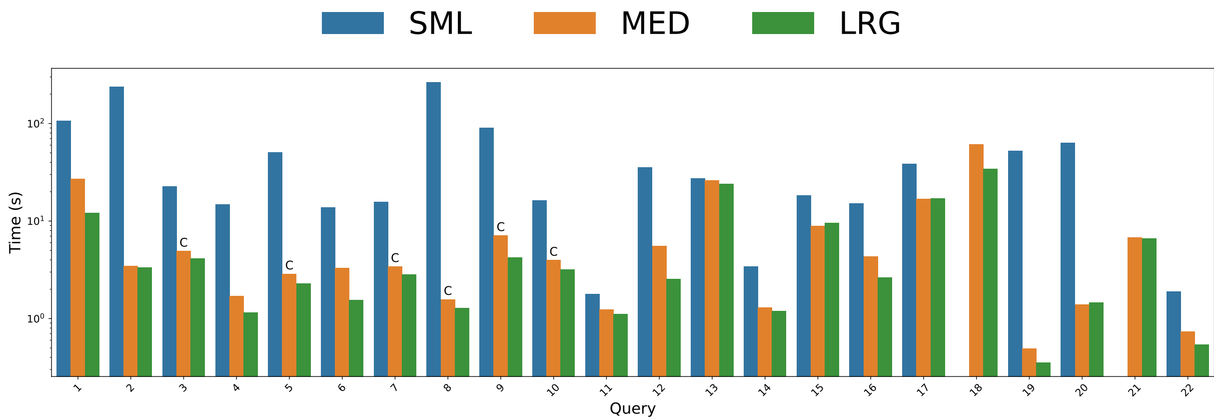


Figure 5.4: **Hardware-Induced Performance Variations** – The performance of TPC-H queries across different hardware environments on pg17. A C above a bar indicates that the query plan changed from the smaller hardware environment.

Similarly, although the overall runtime is approximately the same between pg13 and pg14, Q6 is 1.5× faster on pg14: the query plan remains the same, but it benefits from improved parallel scans [9]. Our investigation in Figure 5.2 finds that such variations are caused by changes in both query plan selection and operator implementations (e.g., faster sequential scans). We observe that the same query plan can improve and degrade in performance over different versions (e.g., Q20's plan never changes despite performance variations). These results illustrate the difficulty of predicting the impact of software versions on query performance.

Hardware Upgrades: To cater to different workloads (e.g., I/O-bound, compute-bound), cloud vendors offer a wide range of instance types that differ in CPU core count, memory capacity, storage performance, and network bandwidth. Figure 5.3a shows the number of instance types available for each workload category in Amazon EC2. There are over 850 EC2 instance types [31] as of 2025, with new hardware classes released multiple times every year [32]. Figure 5.3b shows the growth in instance types over time.

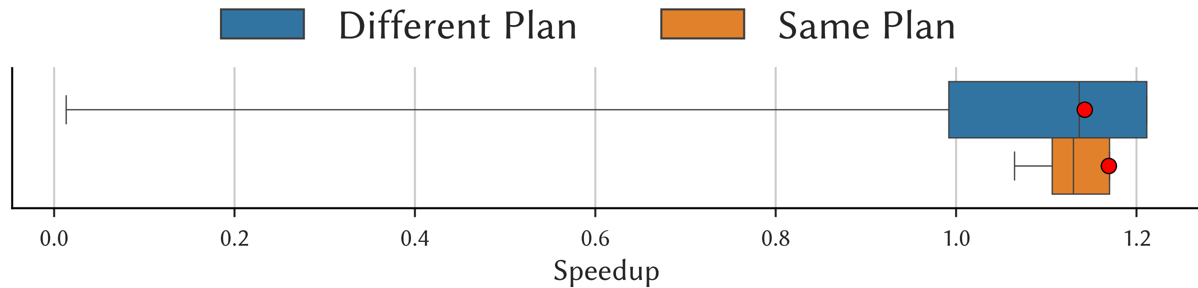


Figure 5.5: **Hardware Upgrades** – JOB speedup on pg17 when upgrading from Medium to Large hardware, grouped by whether the query plan changed. The red dot is sample mean. Whiskers mark p5 and p95 percentiles.

The conventional wisdom is that more expensive hardware provides better performance, and this is typically the case for popular benchmarks where DBMS developers monitor benchmark performance [7]. However, as Figure 5.4 shows, even a hardware upgrade may result in query plan changes and non-uniform speedups. The former occurs because new configurations are necessary to make use of new hardware (e.g., increasing buffer pool size), and the latter is explained by the extent to which existing performance was bottlenecked by the old hardware. More surprisingly, a hardware upgrade may cause regressions for more complex workloads, and because such upgrades are often coarse-grained (e.g., changing cloud instance type), this introduces regression risks to modifying hardware.

To check the generalizability of our findings, we also execute the Join Order Benchmark (JOB) [114] while holding PostgreSQL fixed at pg17 on Medium and Large machines (see Section 5.5 for hardware specifications). JOB stresses the DBMS's ability to pick a good query plan, and Figure 5.5 shows that the magnitude of a hardware upgrade's impact depends heavily on whether the DBMS picks the same query plan. When the query plan is the same, the more expensive machine achieves a modest speedup with a mean improvement of 1.17× and a p99 speedup of 1.77×. However, upgrading the DBMS's hardware may cause it to pick different query plans (e.g., because configuration changes are required to make use of the better hardware), resulting in both larger speedups (e.g., p99 speedup is 3.69×) and unexpected slowdowns (10% of queries experience a slowdown of up to 3.6×). We observed similar behavior across other PostgreSQL versions: with the same query plan, the maximum speedup from a hardware upgrade is modest (up to 2.3×), but the likelihood of performance regressions sharply decreases. In the median case, only 3% of queries regress when the query plan stays the same, as opposed to 28% when the query plan changes. Thus, a hardware upgrade on paper may cause regressions in practice, depending on factors such as whether the query plan remains the same.

Software-Hardware Interactions: Another complication in upgrading is that the performance impact of a software upgrade depends on the hardware environment used and vice versa. For example, a software upgrade that introduces parallel index scans (pg10) will not help if the hardware lacks the necessary cores to make use of it. Such interactions result in non-trivial performance variations across software and hardware combinations.

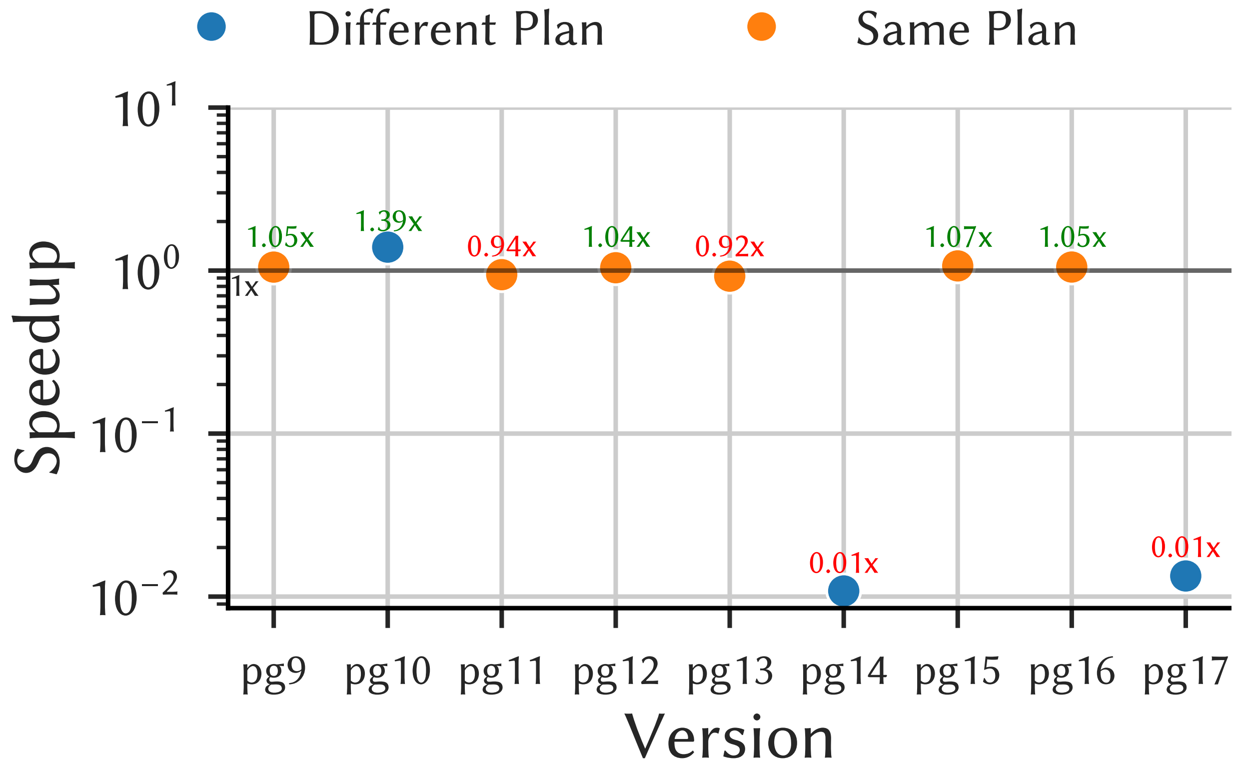


Figure 5.6: **Upgrading Hardware for JOB q13b** – The impact of the same hardware upgrade on performance across PostgreSQL versions.

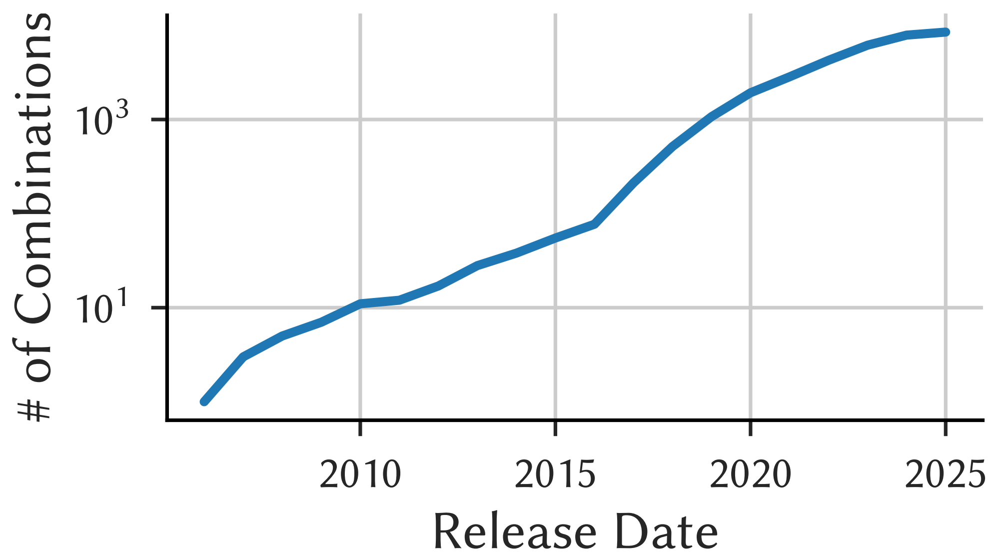


Figure 5.7: **Possible Combinations** – The number of candidate configurations across major PostgreSQL versions and EC2 instance types.

When we examine a single JOB query from Figure 5.5 (q13b) and perform the same upgrade experiment across different PostgreSQL versions, we see that its performance varies when its plan changes. Figure 5.6 shows that depending on the DBMS's version, plan changes result in up to 1.39× speedup and 92× slowdown (e.g., from changing the join order), whereas without plan changes, performance is stable in varying from 1.07× speedup to 1.08× slowdown. Thus, the effect of a hardware upgrade depends also on the software version.

Given this, an organization that wants to pick the best software and hardware for a query must test it against all possible deployment combinations. But as Figure 5.7 shows, doing so is not possible in practice. Even ignoring minor version upgrades, the product of PostgreSQL releases and AWS instance types creates thousands of deployments that a user must load their databases and benchmark their queries on, which is clearly infeasible in both time and cost.

5.1.2 Mitigating Upgrade Issues

In practice, organizations today must nevertheless upgrade. They manage upgrade-related problems through the following techniques:

Mirroring: An organization first deploys a planned upgrade to a test instance that has exactly the same software and hardware as the production instance. They then mirror traffic via a proxy [62] from the production instance to the test instance so that regressions are found and mitigated before they appear in production. However, mirroring is cost-prohibitive and does not provide pre-upgrade visibility into potential upgrade problems, only discovering them post-upgrade. The primary benefit of mirroring is ensuring that upgrade problems manifest first in a safe testing environment. Despite these limitations, mirroring is often the most viable upgrade pathway for organizations today.

Pinning: To prevent query regressions, an organization may pin the DBMS's optimizer version [35], use plan hints [37], or force the use of an old plan [22]. But this approach forgoes the benefits of upgrading: it locks the DBMS into a specific behavior that may be fixed or substantially improved in future versions. Furthermore, when these techniques are defensively employed pre-upgrade, they give up the benefits of upgrading entirely in exchange for stability.

These techniques are insufficient because they do not address the core problem of unpredictable upgrade performance. Thus, what is needed is a way for a user to predict the performance of a particular software and hardware combination without needing to deploy it (i.e., users need a *model* of the DBMS's behavior).

5.1.3 DBMS Behavior Models

Existing research on DBMS modeling exists under the umbrella of automated database tuning. The complexity of DBMSs and workloads today has turned extracting optimal DBMS performance into a dynamic multi-dimensional [204] optimization problem that includes system knobs, query knobs, physical design, and workload characteristics. Because requiring humans

to discover the best configuration is onerous and infeasible, researchers have spent decades on automating this database tuning process.

A goal that unifies such efforts is the development of a self-driving DBMS [151]. A self-driving DBMS [153] automatically determines its optimal configuration as its workload and deployment environment evolve, all without human intervention. It uses machine learning (ML) to construct *behavior models* [126] that predict its performance under different configurations. These behavior models enable individual automated database tuning tasks such as knob tuning [115, 177], query optimization [131, 132, 140], index recommendation [64, 74], and performance prediction [126, 130, 210].

To create such models, a self-driving DBMS executes a representative workload to collect training data [61] (i.e., telemetry), often in the form of annotated query plans (e.g., PostgreSQL `EXPLAIN ANALYZE` output). The DBMS then extracts features X from its training dataset D and uses ML techniques to construct a model of its behavior (e.g., how long it takes to fetch data from disk). A behavior model is a parameterized function $f : X \rightarrow Y$ that maps input features to predicted output (e.g., `EXPLAIN` plan to query runtime), where the system learns the mapping f from D by setting the model's parameters P to weights W. This function supports the DBMS's autonomous decision-making by enabling what-if analysis [64] (e.g., predicting the latency of the DBMS's generated query plan with and without an index). Crucial to the upgrade context, a behavior model supports cheap simulation without needing to execute queries or load data (i.e., by invoking f), allowing one to predict system performance without even deploying the DBMS.

Prior work has focused on *static* behavior models that assume a fixed software and hardware environment. To distinguish between multiple environments, we refer to a behavior model that was trained on data from software SW and hardware HW as $^{\text{HW}}_{\text{SW}}$M, where our software versions are pg10 to pg17, and our hardware environments are {SML, MED, LRG} (see Section 5.5 for details). A model's availability enables upgrade recommendation for its corresponding environment (e.g., $^{\text{LRG}}_{\text{pg17}}$M enables predicting the DBMS's behavior on pg17 and LRG). Therefore, as long as a behavior model exists for each desired upgrade *target* (i.e., specific software and hardware combination), it is possible to find a DBMS's best target configuration prior to performing any upgrades.

5.2 Tool Architecture

Given this, we now present $^{\text{D}}_{\text{R}}$·DB, a tool for automated upgrade recommendation that uses pre-trained behavior models to predict the impact of software and hardware changes **before** they are deployed. The core contribution is a method for modifying behavior models that eliminates the need to pre-train a model for every candidate upgrade configuration by creating new models at run-time instead. We defer technical details to Section 5.3. We first walk through an example of $^{\text{D}}_{\text{R}}$·DB's operation.

Figure 5.8 shows the architecture of $^{\text{D}}_{\text{R}}$·DB. First, ❶ an operator (e.g., a DBA, a database tuning algorithm) asks $^{\text{D}}_{\text{R}}$·DB whether they should upgrade their DBMS's software or hardware for a given workload. In this example, the existing software is pg10 and the hardware is SML. ❷ $^{\text{D}}_{\text{R}}$·DB then begins its diagnosis by preparing a DBMS with the specified characteristics and executing the workload, ❸ obtaining deployment-specific telemetry. Next, $^{\text{D}}_{\text{R}}$·DB ❹ feeds the

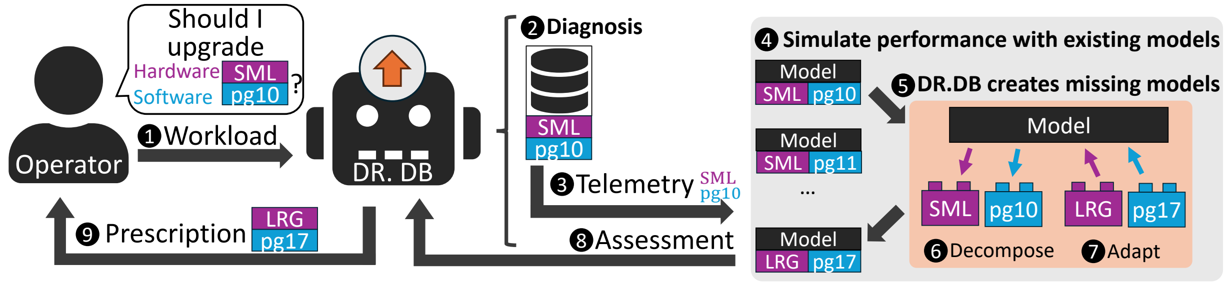


Figure 5.8: **Upgrade Tool Architecture –** $^{D}_{R}$·DB generates its recommendations by creating models to simulate a workload on different environments.

telemetry into its collection of behavior models to simulate the DBMS's performance on different software and hardware combinations. The figure shows that the generated telemetry is for pg10 on a SML machine. Thus, to decide whether to upgrade the hardware to LRG, software to pg17, or both, $^{D}_{R}$·DB must have access to models for $^{LRG}_{pg10}$M, $^{SML}_{pg17}$M, and $^{LRG}_{pg17}$M, respectively. Because existing models are bespoke to their training data, it was previously necessary to pre-train a model for every upgrade target, making it infeasible to support a wide range of environments.

However, $^{D}_{R}$·DB modifies its behavior models to use *adapters*, independent and composable modules that are swappable at run-time (see Section 5.3). Thus, ❺ it can efficiently create new models by switching out a model's adapters. It first selects an available pre-trained model as a starting point. The figure shows $^{D}_{R}$·DB selecting the $^{SML}_{pg10}$M model and ❻ pulling out its software- and hardware-specific adapters. It then ❼ combines the resulting model with its pg17 and LRG adapters, creating a $^{LRG}_{pg17}$M model. $^{D}_{R}$·DB uses this new model to quickly simulate the DBMS's performance on the pg17 and LRG upgrade target. ❽ $^{D}_{R}$·DB repeats this model creation and simulation process for each candidate upgrade configuration to find the best configuration for the provided workload. Lastly, ❾ it recommends the best configuration found to the operator.

In the remainder of this section, we describe why existing behavior models do not work as-is for upgrade recommendations. We then discuss the key insight that makes $^{D}_{R}$·DB's adapter-based approach possible and highlight its qualitative advantages. Our technique adapts existing models (i.e., it does not create new model architectures from scratch). Thus, this paper does not propose a new model for performing upgrade recommendations. Instead, it extends existing behavior models to support upgrades in Section 5.4 and evaluates their improvements in Section 5.5.

5.2.1 Limitations of Existing Models

DBMS behavior models are useful but struggle to support the decision making in the upgrade context. For example, a behavior model cannot predict the impact of a software upgrade if its input features lack software information (e.g., PostgreSQL version as a feature). However, it is not possible to include everything as an input feature: every such feature necessitates collecting training data across its possible values, but training data is difficult and expensive to obtain [119]. We identify two broad limitations of existing behavior models in the context of generating upgrade recommendations.

Generalizability: Because behavior models are trained on historical execution data, they are bespoke to that data's environment and struggle to generalize beyond it. To address the lack of behavior model generalizability, an emerging research direction proposes *foundation models* [185] for databases, which are ML models that are pre-trained to be task- and database-agnostic. Foundation models [58] amortize the cost of training data acquisition and model training across a large variety of downstream tasks This approach has proven successful in domains such as natural language processing and data cleaning [138]. However, unlike such domains, DBMS optimization is a rapidly moving target. For example, although the semantics of English are stable and slow to change, Figure 5.1 demonstrated wide performance variations across PostgreSQL versions over the last five years alone. Because DBMSs evolve rapidly, a new DBMS version may invalidate a behavior model's learnings from previous versions (e.g., new scan operator behavior). Thus, a pre-trained model that does not account for software upgrades will conflate performance characteristics across versions, with no known way of unlearning its biases [58]. Therefore, to obtain foundation models that support upgrades, it is necessary to pre-train models for all possible software and hardware combinations, which is computationally prohibitive.

Featurization: A behavior model's training data environment contains implicit characteristics (e.g., burst I/O performance) that are not explicitly reflected in its input features. A human must decide to include each explicit feature in a model and engineer training data collection that sweeps across the feature's range. But because it is not always evident what features are important (e.g., to our knowledge, we are the first to focus on software version), and sweeping feature ranges is too computationally expensive [192], it is not realistic to assume that a model will include every important characteristic as a feature. Furthermore, the ML community's "bitter lesson" [6] observes that methods based on human knowledge (e.g., manual feature engineering) are less effective than general methods that leverage computation. Instead of squeezing more features into behavior models, what is needed is a general approach that leverages computation to improve behavior model quality.

5.2.2 Key Insight: Isolate System Characteristics

To address the limitations of behavior models in the context of database upgrades, we make a critical observation: behavior models struggle to generalize because they do not isolate system-specific characteristics. For example, there are purely data-driven models [93] that target logical characteristics and therefore do not make physical predictions such as wall-clock time, making them unsuitable for upgrade recommendations. Other models learn both system and data characteristics simultaneously by training on executed queries [61], resulting in difficulties above. However, to our knowledge, there are no models that are purely system-driven (i.e., capture system-specific characteristics independent of the training data workload). Such models would enable upgrade recommendation by simulating performance under different configurations.

$^{\text{D}}_{\text{R}}$·DB's key insight is to isolate the effect of software and hardware from existing behavior models by fine-tuning *adapters* that are individually composable and swappable. For example, exchanging a model's `pg10` adapter with a `pg17` adapter modifies its predictions to assume the latter's characteristics. $^{\text{D}}_{\text{R}}$·DB creates these adapters by leveraging recent developments in fine-

tuning methods for ML models [127], which we describe in Section 5.3.

A key advantage of our adapter-based approach is that it reduces the problem of training a model for every supported upgrade scenario ($\prod_i d_i$ across software and hardware dimensions) to training adapters for each dimension independently ($\sum_i d_i$), reducing both the training data collection and the model training necessary. We provide a concrete example in Section 5.4's case study. Furthermore, model developers are able to pre-train such adapters for distribution as a one-time upfront cost with the release of new software and hardware types (e.g., a cloud provider may distribute a base model and different adapters for customers to choose from, allowing mix-and-match for customer deployments). One limitation of our adapter-based approach is that it only works for deep models (e.g., it does not support tree models such as random forests). However, given current trends towards pre-trained deep models, this is not an impediment in practice.

5.3 Implementation

In this section, we describe the ML techniques behind $^{D}_{R}$·DB's adapter-based approach. First, we describe modern methods for adapting ML models through fine-tuning. We then describe empirical methods for combining fine-tuning artifacts across dimensions (e.g., composing software adaptation with hardware adaptation), a capability that makes them attractive for upgrade recommendation. Lastly, we discuss how to identify suitable fine-tuning candidates in the context of database upgrades.

5.3.1 Fine-Tuning Behavior Models

As Section 5.1.3 discusses, a behavior model's functionality is supported by its parameters P with corresponding weights W. Recent developments in large language models [58] have demonstrated that one effective technique for ML tasks is to pre-train a general-purpose model on a large dataset $D_{general}$ to obtain weights $W_{general}$, and then *fine-tune* the model on a smaller task-specific dataset D_{task} to obtain an updated set of weights W_{task} that have better task-specific performance. Early approaches to fine-tuning updated all parameters P (i.e., they continued training the entire model on D_{task}), but this became computationally infeasible as the number of model parameters grew into the billions. Hence, newer approaches to fine-tuning have focused on only updating a small fraction of the model's parameters, greatly reducing the computational cost of adapting the model to a specific task.

One such approach is low-rank adaptation (LoRA) [98], which freezes the pre-trained model weights W and injects a small number of new trainable parameters P_{new} to obtain a new model with parameters P'. The fine-tuning process then only updates the weights of the new parameters W_{new}, reducing the number of trainable parameters by over 10,000× [98]. LoRA decomposes the weight update $\Delta W \in \mathbb{R}^{p \times q}$ during model training into a much cheaper operation involving two smaller matrices of a reduced dimension r. Specifically, for $A \in \mathbb{R}^{r \times q}$ and $B \in \mathbb{R}^{p \times r}$, it observes that $W' = W + BA$ (i.e., $\Delta W = BA$). We specify a LoRA *adapter* L as its corresponding matrices A and B, i.e., $L = (A, B) \in (\mathbb{R}^{r \times q}, \mathbb{R}^{p \times r})$. This decomposition accelerates fine-tuning for the large billion-parameter that have become common today [58].

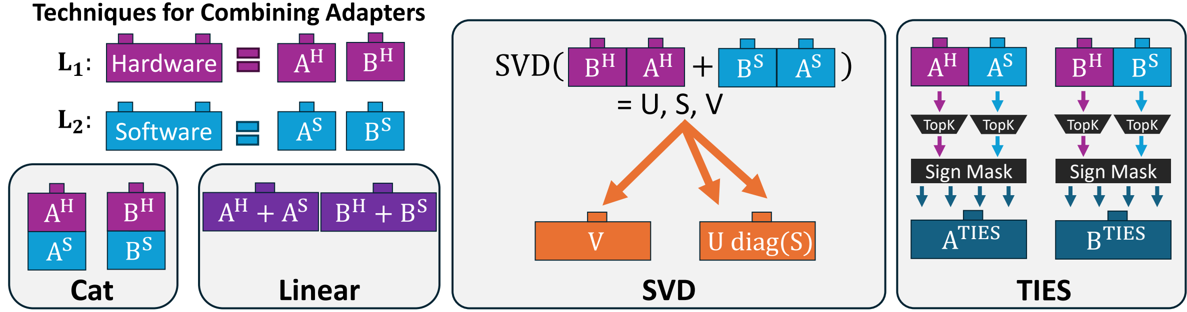


Figure 5.9: **Composing Adapters** – Techniques that combine adapter effects.

As Section 5.1.1 outlines, the difficulty in developing an upgrade recommender today is in collecting training data and building behavior models for the combinatorial number of software and hardware environments available. Two interesting properties of LoRA adapters that mitigate this issue are their **swappability** (i.e., they exist separately from the model) and **composability** (i.e., adapters can be combined). For example, an image generation model that combines an adapter for the concept of a dog and another adapter for the style of watercolor paintings will produce watercolor paintings of dogs [164]. Those adapters exist independently of the image generation model and support cheap run-time loading and unloading. Furthermore, there is no inherent limit to the number of active LoRA adapters [165]. Given this, our tool D R·DB captures dimensional differences in DBMS behavior models through loadable and composable LoRA adapters. In doing so, it achieves a sharp reduction in the training data collection necessary and enable run-time customization of model behavior, with a detailed example in Section 5.4. To our knowledge, we are the first to explore using these LoRA properties in the context of improving DBMS behavior models.

5.3.2 Composable Fine-Tuning

We briefly discuss existing techniques for combining adapters. Given two adapters $L_1 = (A_1, B_1)$ and $L_2 = (A_2, B_2)$, we can combine them with weights w_1 and w_2 to obtain a new adapter $L' = (A', B')$. As techniques for combining adapters remain an active area of research, we limit our discussion to an empirical evaluation of four popular techniques that result in different model performance characteristics. We visualize these methods in Figure 5.9 and describe them briefly below. We evaluate these techniques in Section 5.5.

cat: This method concatenates the two adapters to produce one adapter of increased dimension:

$$A' = \begin{bmatrix} w_1 A_1 \\ w_2 A_2 \end{bmatrix} \in \mathbb{R}^{(r_1+r_2)\times q}$$

$$B' = \begin{bmatrix} w_1 B_1 & w_2 B_2 \end{bmatrix} \in \mathbb{R}^{p\times(r_1+r_2)}$$

linear: This method adds two adapters of the same shape (i.e., it requires $r_1 = r_2$), resulting in one adapter of the same size:

$$A' = \left[(w_1 A_1 + w_2 A_2)\right] \in \mathbb{R}^{r_1 \times q}$$

$$B' = \left[(w_1 B_1 + w_2 B_2)\right] \in \mathbb{R}^{p \times r_1}$$

svd: This method first computes the weighted sum of the two adapters

$$M = \left[(w_1 B_1 A_1 + w_2 B_2 A_2)\right] \in \mathbb{R}^{p \times q}$$

and then takes the singular value decomposition:

$$\text{SVD}\,(M) = USV^{\top}$$

It then reduces the ranks of U, S, and V to rank r', producing the adapter:

$$A' = V[: r', :]$$

$$B' = U[:, : r']diag(S^{[:r']}$$

ties: This method takes the top-k elements of each adapter, elects a sign mask across the adapters, and takes the mean where the sign mask is valid. We defer to the original TIES [194] paper for a full description of their technique.

5.3.3 Deciding on Adapters

Given the ability to create and combine adapters, we now discuss how to decide that an adapter is appropriate. We argue that adapters should capture modeling factors that are easily categorized and changed but are nevertheless difficult to account for.

For example, software version changes are discrete and occur at regular time intervals (e.g., every year). It is possible to quickly determine what version of PostgreSQL a user is running and it is technically possible to perform an upgrade to the next version. However, the exact bundle of features that accompany each version change is not always obvious and whether they impact a workload is unclear. Thus, models should capture software versions as adapters. Hardware upgrades are likewise a good candidate because machine specifications and cloud instance sizes are discrete classes with complex performance characteristics.

5.4 Case Study: Transformer-Based Models

We now demonstrate applying $^{\text{D}}_{\text{R}}$·DB to a specific behavior model. QueryFormer [208] is a tree-structured Transformer-based [178] model that supports predicting a query plan's latency. We chose QueryFormer to represent the Transformer-based models that are prevalent [58] among pre-trained models today. However, $^{\text{D}}_{\text{R}}$·DB's principles apply for any neural network model.

We first describe salient characteristics of QueryFormer's architecture, deferring to the original paper [208] for a complete description. QueryFormer converts a query plan node into a vector embedding by extracting its components (e.g., operator, join type, predicates) and then supplying them into component-specific linear [43] layers, whose outputs are then concatenated into a single vector. It captures the parent-child relationships between plan nodes with height embeddings and intelligent information masking that controls how data flows through the Transformer's self-attention [178] mechanism. For example, it allows for more weight to be placed on a parent node's immediate children than on its distant descendants. It also allows the reducing the weights of cousin nodes (i.e., nodes at the same level from different branches). This process transforms each query plan into a collection of input embeddings X with associated prediction targets Y (e.g., node latency), allowing standard mini-batch model training to learn a QueryFormer model's weights W.

${}^{\text{D}}_{\text{R}}$·DB uses a standard QueryFormer architecture with a modified featurization. We limit our implementation to features that the tool can extract from a DBMS with low overhead (e.g., we removed a predicate bitmap that requires table sampling) and add features that only exist in an upgrade context (e.g., when upgrading, a query's pre-upgrade duration is known). We adjust hyperparameters as necessary to accommodate new workloads and versions (e.g., the number of node types in newer versions of PostgreSQL).

To create an adapter for QueryFormer, we first pre-train a base model using data from a specific software and hardware environment (e.g., pg15 on a LRG machine). We then freeze the weights of this pre-trained model ${}^{\text{LRG}}_{\text{pg15}}\text{M}$ and inject adapter parameters into all self-attention blocks. This process creates approximately 2% more parameters (e.g., a QueryFormer instance with 1.4M parameters gains an additional 30k parameters). During model training, the fine-tuning process only updates these new parameters.

With this setup, we then create version-specific and hardware-specific adapters for different software versions and hardware environments by fine-tuning (see Section 5.3.1) on data that only differs in the desired dimension. Crucially, ${}^{\text{D}}_{\text{R}}$·DB creates independent adapters across each dimension of variation. For example, given nine versions and three hardware environments, we only need to train nine version adapters and three hardware adapters (i.e., 12 adapters) as opposed to training 27 adapters or 27 base models (i.e., nine versions times three hardware environments). Thus, to support the scenario in Figure 5.7, ${}^{\text{D}}_{\text{R}}$·DB only needs $944 + 9 = 953$ adapters instead of $944 \times 9 = 8496$ models, a 8.9× reduction.

To create a customized model, ${}^{\text{D}}_{\text{R}}$·DB applies matching adapters across all dimensions at runtime. Continuing the above example of ${}^{\text{LRG}}_{\text{pg15}}\text{M}$, the ${}^{\text{D}}_{\text{R}}$·DB adapts it for a MED machine by applying a hardware adapter to obtain ${}^{\text{MED}}_{\text{pg15}}\text{M}$, and further specializes it to pg17 by applying a version adapter, resulting in ${}^{\text{MED}}_{\text{pg17}}\text{M}$. The effectiveness of the final model depends on the adapter combination technique used (see Section 5.3.2), with detailed evaluations in Section 5.5.

5.5 Evaluation

We now evaluate ${}^{\text{D}}_{\text{R}}$·DB's effectiveness at adapting DBMS behavior models for different environments (e.g., software, hardware) for automated upgrade recommendations, focusing on the PostgreSQL DBMS.

We built PostgreSQL from source for versions pg9 (released 2016, last updated 2021) to pg17 (released 2024) using the latest tagged releases, covering most versions in active use. We optimize system configurations with PGTune [40] and deploy PostgreSQL on three hardware environments running Ubuntu 22.04 LTS:

- **Large (LRG):** Server machine [45] with 2×20-core Intel Xeon Gold 5218R CPUs, 128 GB of RAM, and a Samsung PM983 SSD.
- **Medium (MED):** Desktop machine [4] with a 4-core Intel Core i7-8650U CPU, 32 GB of RAM, and a Samsung EVO960 SSD.
- **Small (SML):** We mimic a budget cloud instance type for general-purpose compute workloads [33] by using `cgroups` to limit the resources available on the MED environment to two CPU cores, 8 GiB of RAM, and 4750 Mbps of I/O bandwidth.

We first describe our workloads and experiment configuration in Section 5.5.1. We then consider three upgrade scenarios: (1) absolute workload-level run-time prediction in Section 5.5.2, (2) relative query-level latency prediction in Section 5.5.3, and (3) high-level version and hardware upgrade recommendation performance in Section 5.5.4. We summarize our takeaways for using $^{D}_{R}$·DB in Section 5.6.

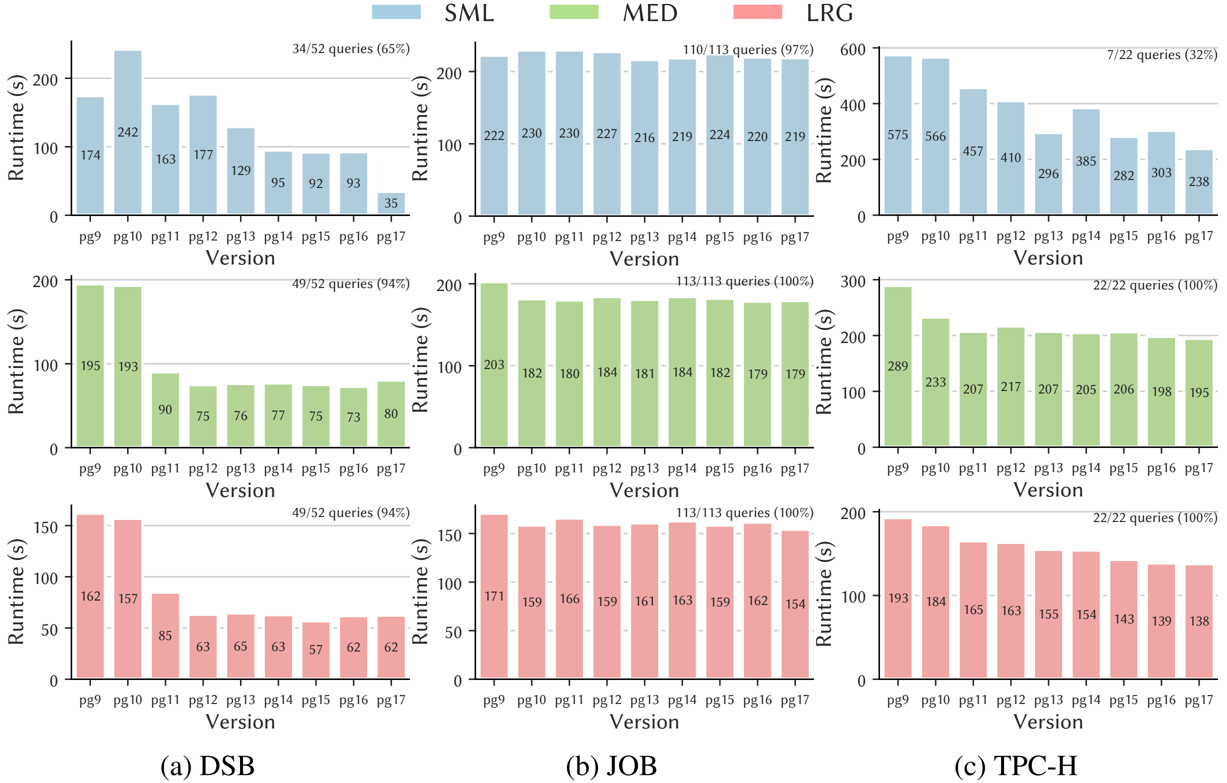
5.5.1 Workload and Experiment Configuration

We execute the following workloads on all 27 combinations of DBMS versions and hardware environments described above.

- **DSB** [78]: This benchmark has 25 tables and 52 query templates. Microsoft created this benchmark by enhancing TPC-DS [172] with complex data distributions and challenging queries. We instantiate DSB with scale factor 10.
- **JOB** [114]: This fixed-size benchmark has 21 tables and 113 queries. It uses the IMDB dataset to stress a query optimizer's ability to pick a good join order. The QueryFormer model [208] that we adapt was initially trained on JOB-light [105].
- **TPC-H** [173]: This benchmark has eight tables and 22 query templates. It models a business analytics workload with a uniform data distribution. We instantiate TPC-H with scale factor 10. We note that PostgreSQL developers monitor TPC-H performance [7], with regressions reported to the mailing list [2].

In total, there are nine DBMS versions, three hardware environments, and three benchmarks, for a total of 81 base experiment configurations. We run each query for five iterations on each configuration and report the median iteration for our evaluations.

We set a per-query timeout of 5 min for all executions. We observe that queries timed out more frequently on older versions and weaker hardware. We visualize how query run-time varies across the different software versions and hardware environments in Figure 5.10 while excluding queries that throw an error or time out in any of the versions (i.e., within a subplot, the number of queries that constitute each bar's run-time is the same for a fair comparison). We observe the largest speedup and slowdown across versions are both for DSB on SML, with a max speedup of 2.67× (pg16 to pg17) and a max slowdown of 1.39× (pg9 to pg10). Many of the larger



(a) DSB (b) JOB (c) TPC-H

Figure 5.10: **Version** × **Hardware Runtimes** – The total execution time of the benchmarks across PostgreSQL version and hardware combinations. Each subplot excludes queries that do not complete in time across all versions.

speedups are from PostgreSQL versions whose release notes highlight general performance improvements and enhanced query parallelism (e.g., pg10 [3], pg11 [5]). However, the results for DSB on the SML hardware (Figure 5.10a) show that even performance-oriented releases may incur regressions on a subset of the workload.

Models Used: We use the workloads' executed plans to train QueryFormer base models and adapters as described in Section 5.4. ${}^{\text{D}}_{\text{R}}$·DB injects the adapters' parameters into the attention mechanism's query, key, and value layers. However, QueryFormer is a schema-specific model (e.g., it requires a training different models for TPC-H and DSB), which precludes a cross-workload evaluation. Thus, we designed a version of QueryFormer without schema dependencies by replacing schema-specific features with a corresponding statistical description (e.g., substituting a table identifier with the table's statistics). We show both per-workload QueryFormer (DSB, JOB, TPC-H) and cross-workload QueryFormer (Combined) models.

Per-Workload Model Setup: We note that QueryFormer observed that some queries are more difficult to predict than others [208] (i.e., model performance is heavily influenced by train-test split). Because we focus on upgrade scenarios where the workload is known, and moreover our

emphasis is on the *relative* improvement from $^{\text{D}}_{\text{R}}$·DB's model adaptation, we show results for training and testing on all data for the per-benchmark models of schema-specific QueryFormer.

Cross-Workload Model Setup: To cover a scenario where a single model needs to generalize across all workloads, we combine all three of our workloads into a single workload and perform a standard 80-20 train-test split. We then train our schema-agnostic QueryFormer on this combined workload and evaluate its performance on the test set. Although this configuration mimics standard ML practice, we note that an upgrade-related interpretation of this setup is to assume that a significant fraction of the workload will shift immediately after the database is upgraded, which is less representative of real-world upgrade scenarios.

Notation: For a workload of SQL queries $Q = \{q_1, \cdots, q_n\}$, the run-time of Q for a version v and hardware configuration h is $T_v^h(Q) = \sum_{i=1}^{n} t_v^h(q_i)$, where $t_v^h(q_i)$ is the run-time of query q_i. Given a model M trained on data from version v and hardware h, we write ${}^{h}_{v}M$ to avoid ambiguity, and write M otherwise to simplify notation. Given a workload Q, M predicts a per-query run-time $M(q_i)$ and a total workload run-time $M(Q) = \sum_{i=1}^{n} M(q_i)$. For a model ${}^{h}_{v}M$, $^{\text{D}}_{\text{R}}$·DB's adapters adapt the version (${}^{h}_{v}M_{v'}$), hardware (${}^{h}_{v}M^{h'}$), or both the version and hardware (${}^{h}_{v}M^{h'}_{v'}$), where $v' \neq v$ and $h' \neq h$.

We describe how far a model is from its evaluation scenario E that uses version v_E and hardware h_E with a custom Δ metric. A model ${}^{h}_{v}M$ is 0Δ from E if it trained on the same version and hardware (i.e., trained from scratch for the target environment, ${}^{h_E}_{v_E}M$). It is 1Δ from E if it requires one adapter to match the target environment (i.e., evaluation is performed with a version-adapted model ${}^{h_E}_{v}M_{v_E}$ or a hardware-adapted model ${}^{h}_{v_E}M^{h_E}$). Lastly, it is 2Δ from E if it requires both adapters to match the target environment (i.e., evaluation with ${}^{h}_{v}M^{h_E}_{v_E}$).

Using these definitions, we define the following *model families*. Given a version and hardware environment, among the family of base models *B* (i.e., models trained on any version and hardware with no adapters), there is a model trained-from-scratch (*TFS*) for that combination that is the optimal 0Δ model. For models that are 1Δ away from optimal (e.g., same hardware but different version), we apply a version adapter or hardware adapter to get the *V* and *H* models. Lastly, for models that are 2Δ away from optimal (i.e., different hardware and version), we combine a version and hardware adapter to get the *cat*, *linear*, *ties*, and *svd* models, depending on the combination technique (see Section 5.3.2).

Given the above definitions, we measure $^{\text{D}}_{\text{R}}$·DB's effectiveness in three upgrade scenarios. First, we focus on a workload-level "will my workload run-time improve?". Next, we examine a query-level "how is each query affected?". Lastly, we address an operational scenario "if I upgrade, will I regret it?". We begin with workload-level absolute run-time predictions.

5.5.2 Workload-Level Absolute Prediction

Because query execution is slow [120], a self-driving DBMS uses its behavior models to predict the latency of a workload instead of executing it. However, as Figure 5.10 shows, the same workload varies in run-time depending on the DBMS's version and hardware, and therefore a

model trained on one environment makes inaccurate predictions in another. Thus, we examine $^{\text{D}}_{\text{R}}$·DB's ability to predict an upgrade's impact on workload-level absolute run-times.

Our evaluation metric is the absolute error (*AE*) of a model M for a workload Q on version v and hardware h, which we define as $AbsErr_v^h(M, Q) = |M(Q) - T_v^h(Q)|$. We measure $^{\text{D}}_{\text{R}}$·DB's effectiveness as whether applying its adapters to M produces models M' with lower absolute error (i.e., $AbsErr_v^h(M, Q) < AbsErr_v^h(M', Q)$ where $M' \in \{M_v, M^h, M_v^h\}$ adapts to the target environment).

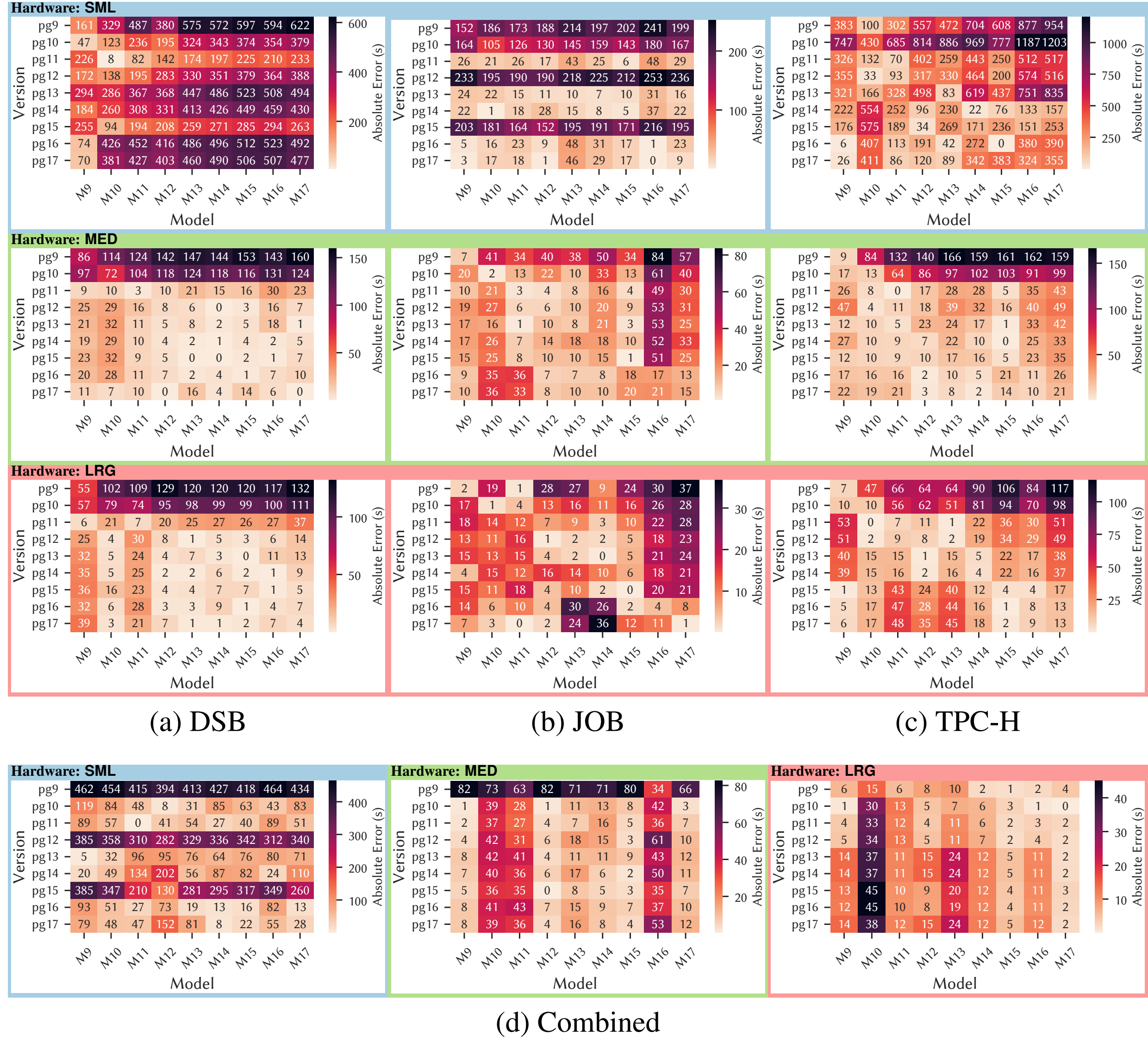
Version Adaptation We first examine $^{\text{D}}_{\text{R}}$·DB when holding the hardware while varying versions. For each benchmark and hardware environment combination, we train nine QueryFormer models M9 to M17 on their corresponding data from pg9 to pg17 (e.g., M9 is only trained on pg9 data) with L1-loss [41] to minimize mean absolute error. We then evaluate the absolute error of using those models to predict the run-time of the workload across all versions for that benchmark and hardware combination.

Figure 5.11 shows the absolute error of each model over the version and hardware combinations. We did not apply $^{\text{D}}_{\text{R}}$·DB's adapters to these models (i.e., the results are the base QueryFormer models without any adaptation).

We first fix the benchmark and hardware environment to compare the absolute error of using a model trained on different PostgreSQL versions against the model trained on the same PostgreSQL version. Across benchmarks, the absolute errors are worse by 3.9–153.3× for DSB, 47.4–77.2× for JOB, and 16.9–246.3× for TPC-H. We find a similar range across hardware environments, with absolute errors worsening by 3.9–47.4× for SML, 49.9–246.3× for MED, and 8.9–77.2× for LRG. Thus, using a model for the wrong version results in absolute errors that are 4–246× worse. In the Combined case, we find that the absolute errors are worse by up to 790× for SML, 10× for MED, and 18× for LRG.

Although using a model with the same version is the best, the randomness of ML-based predictions and error cancellations (e.g., overpredictions cancelling with underpredictions) results in other models performing better. We find that compared to the best model, the absolute error of the same version model is still worse by 10.5–67.9× for DSB, 6.0–44.5× for JOB, and 36.4–1149.7× for TPC-H. When analyzed across hardware environments, the absolute errors are worse by 10.5–1149.7× for SML, 6.0–93.9× for MED, and 6.4–36.4× for LRG. The high 1149× error for SML is from the pg16 TPC-H model, where the same version model has an error of 380s while the best model has an atypically low error of 0.1s. We attribute this to the high frequency of query timeouts for this configuration (see Figure 5.10), resulting in noisy training data. We observe similar outliers in the Combined case, where the best model compared to the base model is up to 15× better for SML, 27× for MED, and 646× for LRG.

To better understand these results, we observe that when Figure 5.11 is viewed as a matrix, the diagonal corresponds to using a model trained on the *same* version, the upper triangular corresponds to using models trained on *newer* versions, and the lower triangular corresponds to using models trained on *older* versions. Thus, we compress this data into boxplots as the *Base* models of Figure 5.12. Next, we apply $^{\text{D}}_{\text{R}}$·DB's version adapter to each Base model to produce version-adapted models *V-qkv*. We visualize the absolute error distribution of both sets of models.



(a) DSB (b) JOB (c) TPC-H

(d) Combined

Figure 5.11: **Version Heatmap Analysis (AE)** – The absolute error of workload run-time predictions for each version-specific model across all training data configurations, where `MN` refers to the model trained on `pgN`'s data.

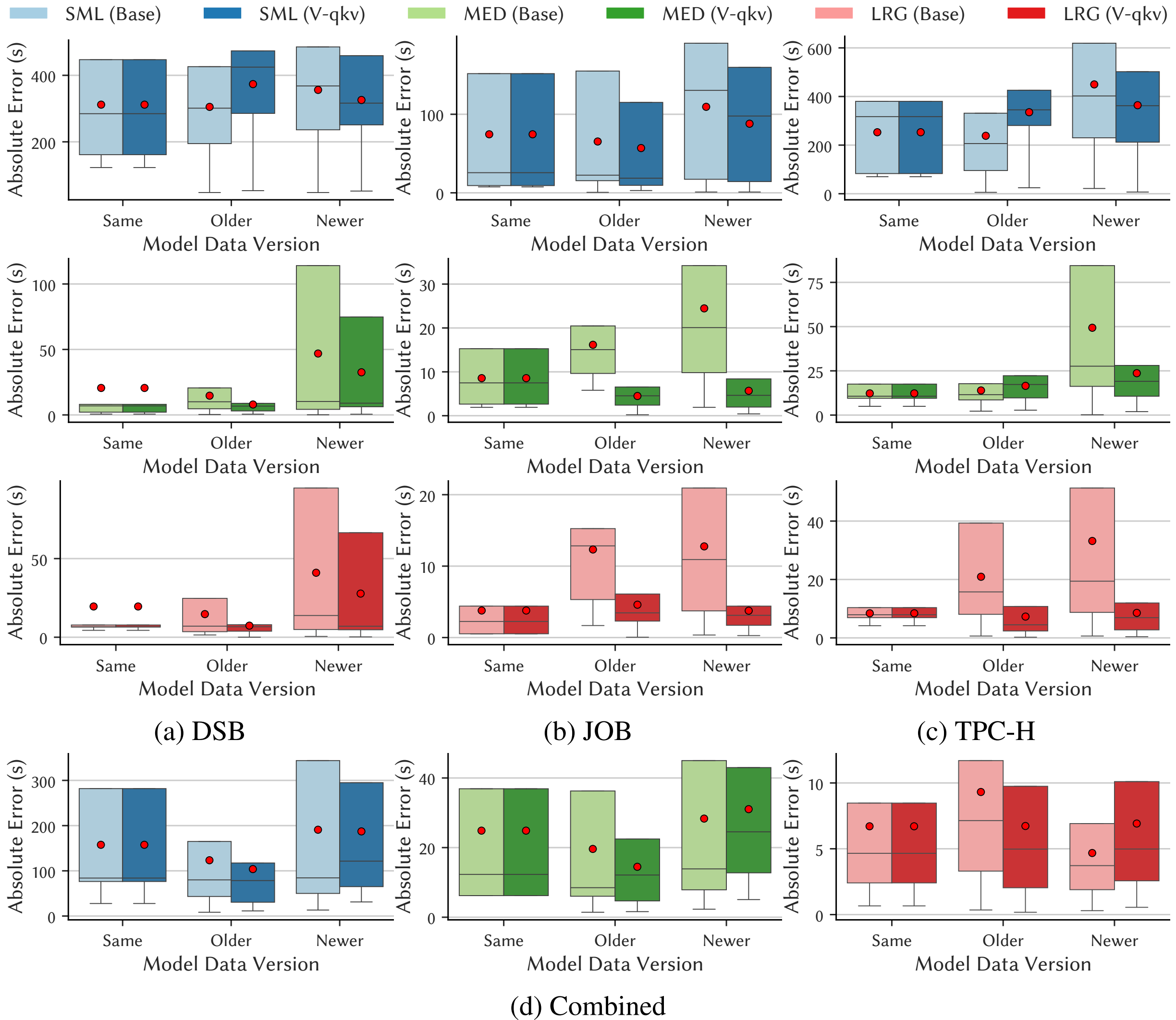


Figure 5.12: **Version Adaptation (AE)** – The shift in model absolute errors for workload runtime predictions when applying ${}^{\text{D}}_{\text{R}}$·DB's version adapters. The red dot indicates sample mean. The whiskers indicate p5 and p95.

Figure 5.12 shows how the absolute error distribution improves by applying $^{\text{D}}_{\text{R}}$·DB's version adapters to convert Base models into version-adapted V-qkv models. For the schema-specific models, $^{\text{D}}_{\text{R}}$·DB improves mean and median error across benchmarks by up to 4.3× on newer models and up to 3.7× on older models, however, it incurs a degradation of up to 1.6× in some of the older models. These results indicate that adaptation is more effective when using newer models on older versions of PostgreSQL. This behavior is expected because the extent of adaptation required is different for both directions: newer models may only need to adjust the performance characteristics of their operators (e.g., "unlearn" performance improvements), whereas older models must reason about new features or operators from scratch. In the Combined evaluation, the train-test split mixes the effect of adaptation with query distribution shift, resulting in many of LRG's Newer models experiencing degraded performance.

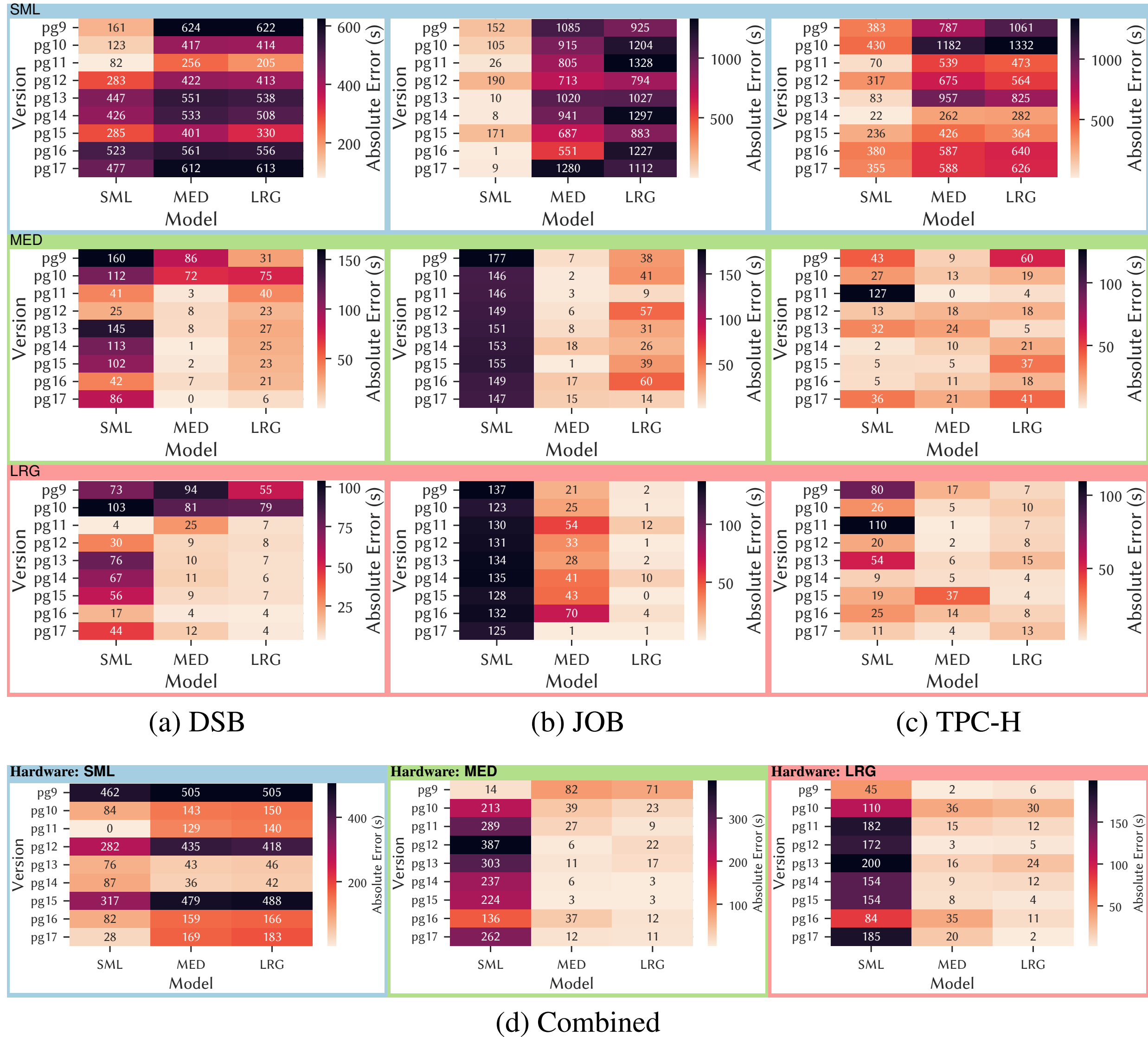
Hardware Adaptation: We now examine $^{\text{D}}_{\text{R}}$·DB in the context of "same version, different hardware". For each benchmark and version combination, we train three QueryFormer models on the different hardware types (SML, MED, LRG) on their corresponding data with L1-loss as before. Next, we compare the absolute error of using a model trained on different hardware against a model trained on the same hardware environment.

Figure 5.13 shows the absolute error of each version-specific model across all hardware environments. The heatmaps are grouped according to hardware environment by row and benchmark by column. Note that none of these models have $^{\text{D}}_{\text{R}}$·DB's adapters applied.

Compared to Figure 5.11, using a model trained on the wrong hardware results in worse absolute errors (up to 1242×) than a model trained on the wrong version (up to 790×), with errors over 100× being more common. Additionally, while a model trained on the same hardware is not always the best model like in the same version case, the model with the same hardware has a lower maximum error across all benchmarks and versions (up to 5.75× vs up to 1149×).

As before, we compress the above data into boxplots as the *Base* models of Figure 5.14. We then apply $^{\text{D}}_{\text{R}}$·DB's hardware adapters to each Base model to produce hardware-adapted models *H-qkv*, visualizing their respective absolute error distributions in Figure 5.14.

Figure 5.14 shows how the absolute error distribution shifts when applying hardware adapters to convert Base models into H-qkv models. Holding the benchmark fixed, (1) the median error on LRG improves by 1.2–3.2× for DSB, 12.3–31.3× for JOB, and 1.2–1.7× for TPC-H, (2) the median error on MED improves by 1.2–2.4× for DSB, 11.7–20.0× for JOB, and 0.8–1.6× for TPC-H, and (3) the median error on SML improves by 4.5–6.2× for DSB, 4.7–6.6× for JOB, and 0.76–0.81× for TPC-H. We observe improvement in almost all cases of hardware adaptation, with the notable exception of adapting SML models for TPC-H. We attribute this to the lack of training data caused by timeouts as mentioned above. Although the median error worsens slightly after hardware-adaptation in some cases (e.g., adapting MED models to LRG for TPC-H), the overall distribution still improves with tighter error percentile bounds. These results suggest that hardware adaptation is almost always desirable. As noted earlier, in the Combined evaluation, the train-test split results in mixing the effect of adaptation with query distribution shift: the adapters reduce median absolute error by up to 2.1× on SML, 11× on MED, and 13× on LRG, but some MED models experience up to 2.5× degradation. However, as the Combined workload's SML and LRG models do not degrade, hardware adaptation remains generally beneficial.



(a) DSB (b) JOB (c) TPC-H

(d) Combined

Figure 5.13: **Hardware Heatmap Analysis (AE)** – The absolute error of workload run-time predictions for each hardware-specific model across all training data configurations, with model hardware as the row label.

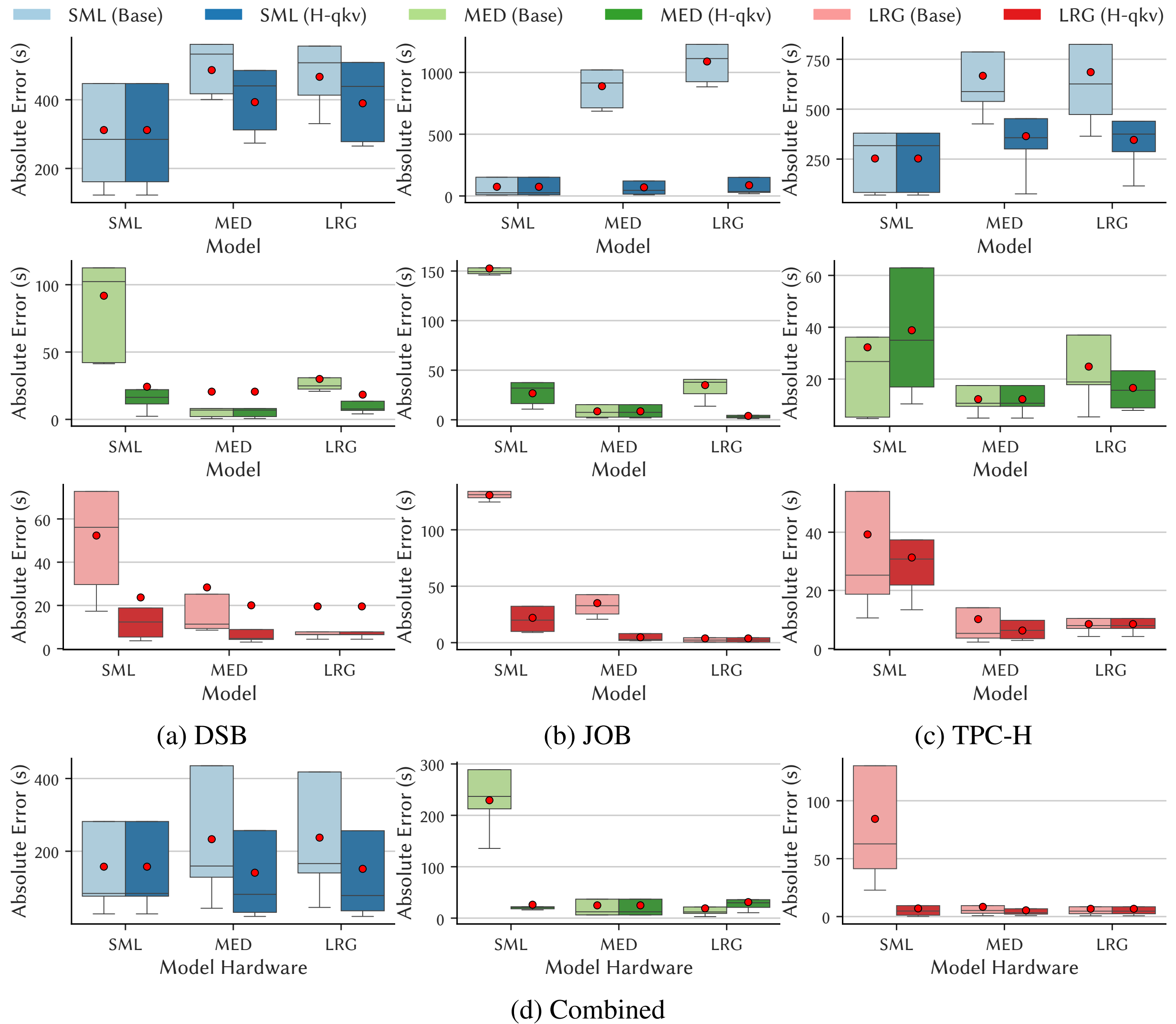


Figure 5.14: **Hardware Adaptation (AE)** – The shift in model absolute errors for workload run-time predictions with $^{D}_{R}$·DB's hardware adapters. The red dot indicates sample mean. The whiskers indicate p5 and p95.

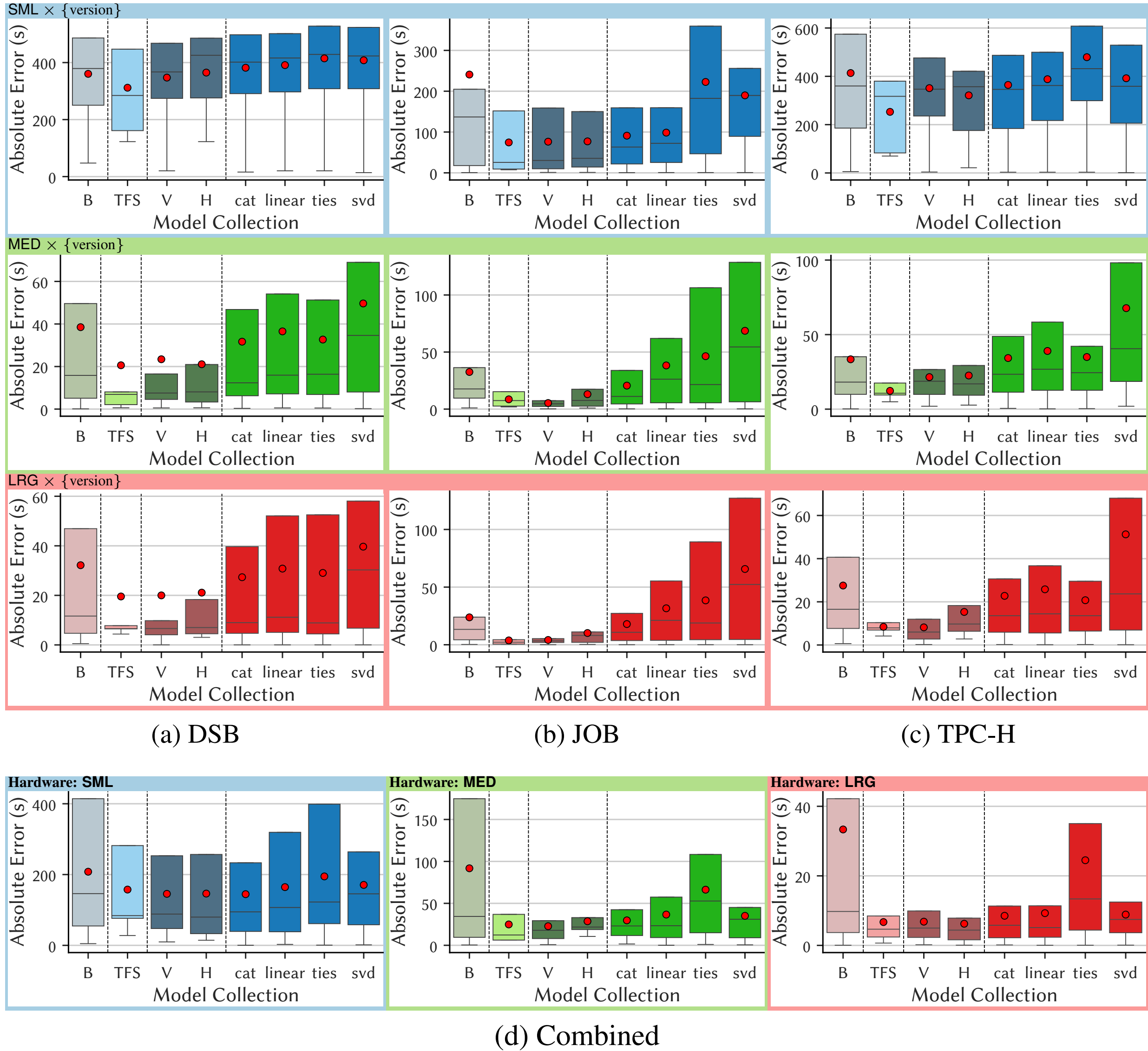


Figure 5.15: **Multi-Δ Adaptation (AE)** – The absolute error distribution of different model families (see Section 5.5.1) when predicting absolute workload run-time. The dashed line separates model families by their Δ classification.

Multi-Δ Adaptation: Lastly, we examine $^{\mathsf{D}}_{\mathsf{R}}$·DB in the context of "different version, different hardware". Recall from Section 5.5.1 that we measure how far away a model is from its evaluation environment using the Δ system and model families.

Figure 5.15 shows the absolute error distribution of the different families of models, where the boxplots indicate the distribution when drawing from a model family. For example, using models that are neither version- nor hardware-adapted will experience errors similar to the *B* category, whereas applying a version adapter results in error distributions in the *V* category. The *TFS* models are the best models, however, it may not be possible to train them for every hardware and software combination. Among the 1Δ models, the *V* models are better than the *H* models (i.e., it is more effective to adapt version than to adapt hardware, suggesting that foundation models should be trained across hardware environments instead of versions). As expected, the 2Δ models are the least effective: some combination techniques (e.g., *svd*) produce worse results than doing no adaptation at all (the *B* models). Hence, the combination technique impacts the effectiveness of the final model, with *cat* offering competitive performance in most scenarios.

To summarize, we find that for workload run-time predictions, the median schema-specific model's absolute error is improved by up to 4.3× through version adaptation and up to 31× through hardware adaptation. We also find that the *cat* combination technique is the most effective, with errors on par or better than the existing *B* models. We observe that the model for the Combined workload occasionally exhibits different behavior from the workload-specific models, which we attribute to QueryFormer's sensitivity to query distribution drift. We discuss the implications of these results in Section 5.6.

5.5.3 Query-Level Relative Predictions

Our previous experiments examined the workload-level absolute error. Although this is a useful metric for deciding whether a workload improves, it is affected by error cancellation (i.e., overprediction negates underprediction). Therefore, even when absolute error improves, the model may be worse at predicting individual queries.

Given this, we now evaluate $^{\mathsf{D}}_{\mathsf{R}}$·DB's ability to make upgrade-related query-level latency predictions. Because of space constraints and similarities in experiment setup to Section 5.5.2, we only describe the experimental differences and then summarize the key findings. We focus on the query-level Q-error [117, 130] (*QE*) instead of the workload-level absolute error. The Q-error is the maximum ratio of the model's prediction to the ground truth. For a given model M, query q, version v, and hardware environment h, we have that $QErr_v^h(M, q) = \max\left(\frac{M(q)}{t_v^h(q)}, \frac{t_v^h(q)}{M(q)}\right)$. We train QueryFormer to optimize for MSE loss [42] to increase outlier penalties.

We first consider version adaptation. Figure 5.16 shows the Q-error distribution before and after applying $^{\mathsf{D}}_{\mathsf{R}}$·DB's version adaptation. We first examine the schema-specific models. We observe that while the median Q-error only improves by up to 1.2× across all configurations, the mean Q-error improves a lot more: up to 3.2× for DSB, 49× for JOB, and 2.0× for TPC-H (i.e., the tail of the distribution is pulled in). We attribute the difference in Q-error across benchmarks to the nature of their queries. In JOB, queries perform superfluous work with bad join orderings and are more efficient with the right join order, and this depends more heavily on the DBMS version. In contrast, DSB and TPC-H perform more consistent amounts of work. For

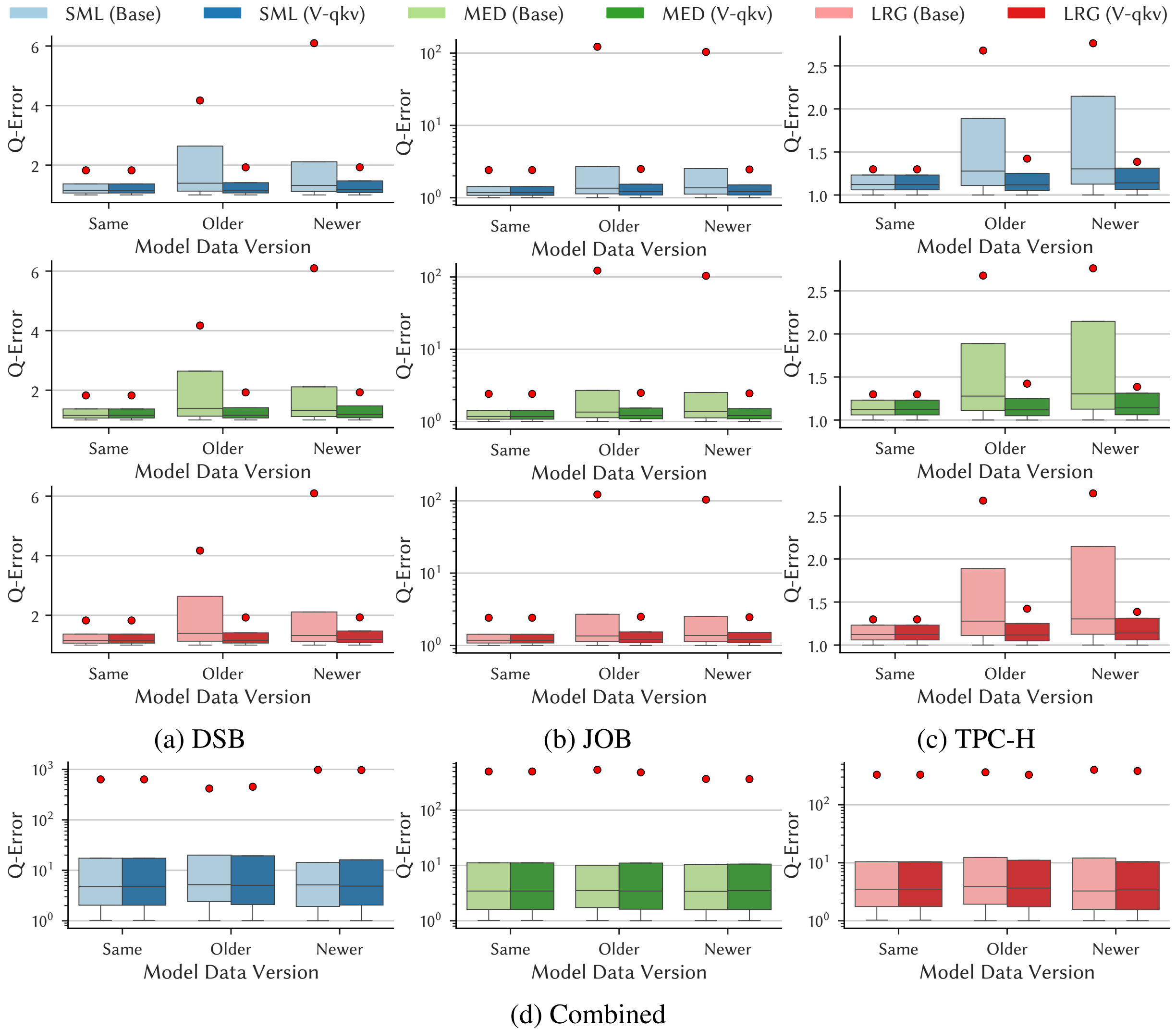


Figure 5.16: **Version Adaptation (QE)** – The shift in model Q-Error for query latency predictions when applying $^{D}_{R}$·DB's version adapters. The red dot indicates sample mean. The whiskers indicate p5 and p95.

(a) DSB

(b) JOB

(c) TPC-H

(d) Combined

Figure 5.17: **Hardware Adaptation (QE)** – The shift in model Q-Error for query latency predictions when applying ${}^{\mathrm{D}}_{\mathrm{R}}$·DB's hardware adapters. The red dot indicates sample mean. The whiskers indicate p5 and p95.

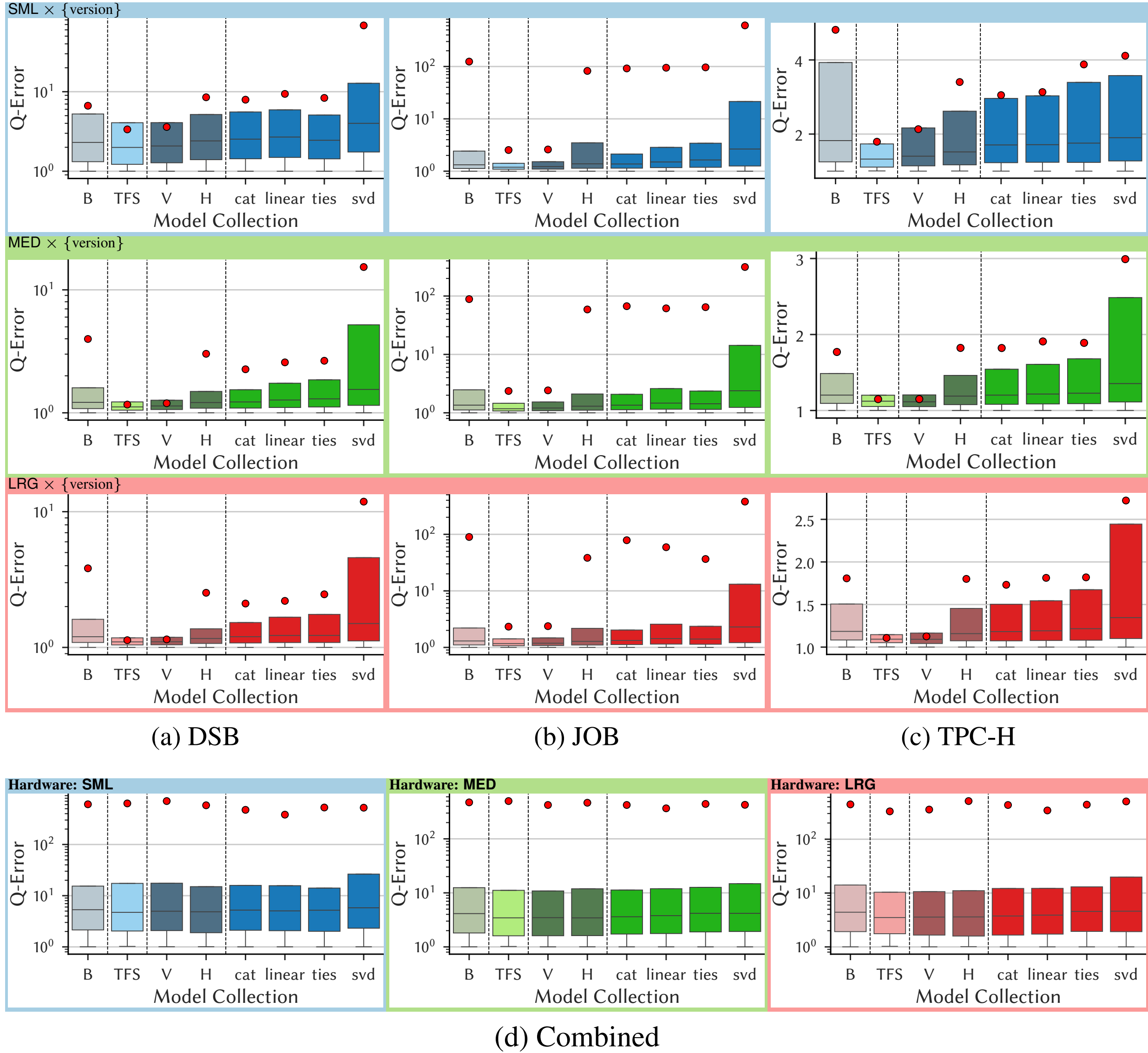


Figure 5.18: **Multi-Δ Adaptation (QE)** – The Q-Error distribution when predicting query latencies with different model families.

the Combined model, we observe that version adaptation has limited effect in either direction: up to 1.11× improvement and up to 1.08× degradation for both mean and median Q-error. As noted previously, we attribute this to query distribution shift.

We then consider hardware adaptation, where Figure 5.17 shows Q-error distributions with and without $^{\text{D}}_{\text{R}}$·DB's hardware adapters. We see higher improvements in Q-error across benchmarks for both the median (1.8–4.1×) and mean (3.4–108×). As with absolute workload-level predictions from before, hardware adaptation almost always improves a model's capabilities. We attribute this to how hardware effects often amount to performance differences in operator behavior, and thus getting a model to recalibrate its operator semantics is a relatively straightforward improvement. In the Combined workload setting, hardware adaptation has modest improvements (1.65× for median, 1.16× for mean) and limited degradation (up to 1.11× for mean, no worsening of median). These results are consistent with findings from the workload-specific models.

Lastly, we consider the same multi-Δ adaptation setup from Section 5.5.2 but with Q-error as the metric. Figure 5.18 shows the error distributions for the model families. We continue to find for the schema-specific models that it is more effective to adapt version than to adapt hardware, and the *cat* combination technique is often the most effective. However, the effect is much more limited for the Combined workload, indicating a dependence on how closely the train set represents the test scenario.

Thus, for query-level latency predictions, $^{\text{D}}_{\text{R}}$·DB's version- and hardware-adaptation shrinks the Q-error distribution's tail for schema-specific models: the median improves by up to 2× but the mean improves by up to 108×. However, the adapters are less impactful for the Combined workload. These results indicate that adapters are better in scenarios where the workload is mostly known (e.g., during an upgrade).

5.5.4 Minimizing Upgrade Regret

In the previous subsections, we saw how $^{\text{D}}_{\text{R}}$·DB enables model adaptation for upgrade recommenders based on workload-level and query-level predictions. We now evaluate the extent to which $^{\text{D}}_{\text{R}}$·DB avoids making bad recommendations (i.e., minimizes regressions when upgrade advice is taken). We consider scenarios in which the DBMS is upgraded to newer versions (i.e., unidirectional version) or better hardware (i.e., only SML to MED and MED to LRG).

We define the *CScore* metric to measure whether a model can detect performance regressions. Given a specific query q, hardware environments h_1 and h_2 where $h_1 < h_2$ (i.e., h_2 is more powerful than h_1), and DBMS versions v_1 and v_2 where $v_1 < v_2$ (i.e., v_2 is newer than v_1), the optimal decision to upgrade (U) or stay (S) versions is given by $O^h_{v_1 \to v_2}(q) = U$ if $t^h_{v_2}(q) \leq t^h_{v_1}(q)$ and S otherwise. A model M's corresponding upgrade advisor A recommends staying or upgrading for a query based on the model's predictions: $A^h_{v_1 \to v_2}(M, q) = U$ if $M^h_{v_2}(q) \leq M^h_{v_1}(q)$ and S otherwise. Given this, there are four possible upgrade advisor outcomes. The model may correctly suggest (1) staying or (2) upgrading (i.e., $A^h_{v_1 \to v_2}(M, q) = O^h_{v_1 \to v_2}(q)$). It may also incorrectly suggest (3) staying when upgrading would have been better, missing an opportunity to improve. Lastly, the model may recommend upgrading when the optimal decision was to stay (i.e., the query regresses). Because the cost of upgrading is much higher than staying, we permit the (3) case and define the CScore as the quantity $\frac{(1)+(2)+(3)}{(1)+(2)+(3)+(4)}$ (i.e., only (4) is penalized). The

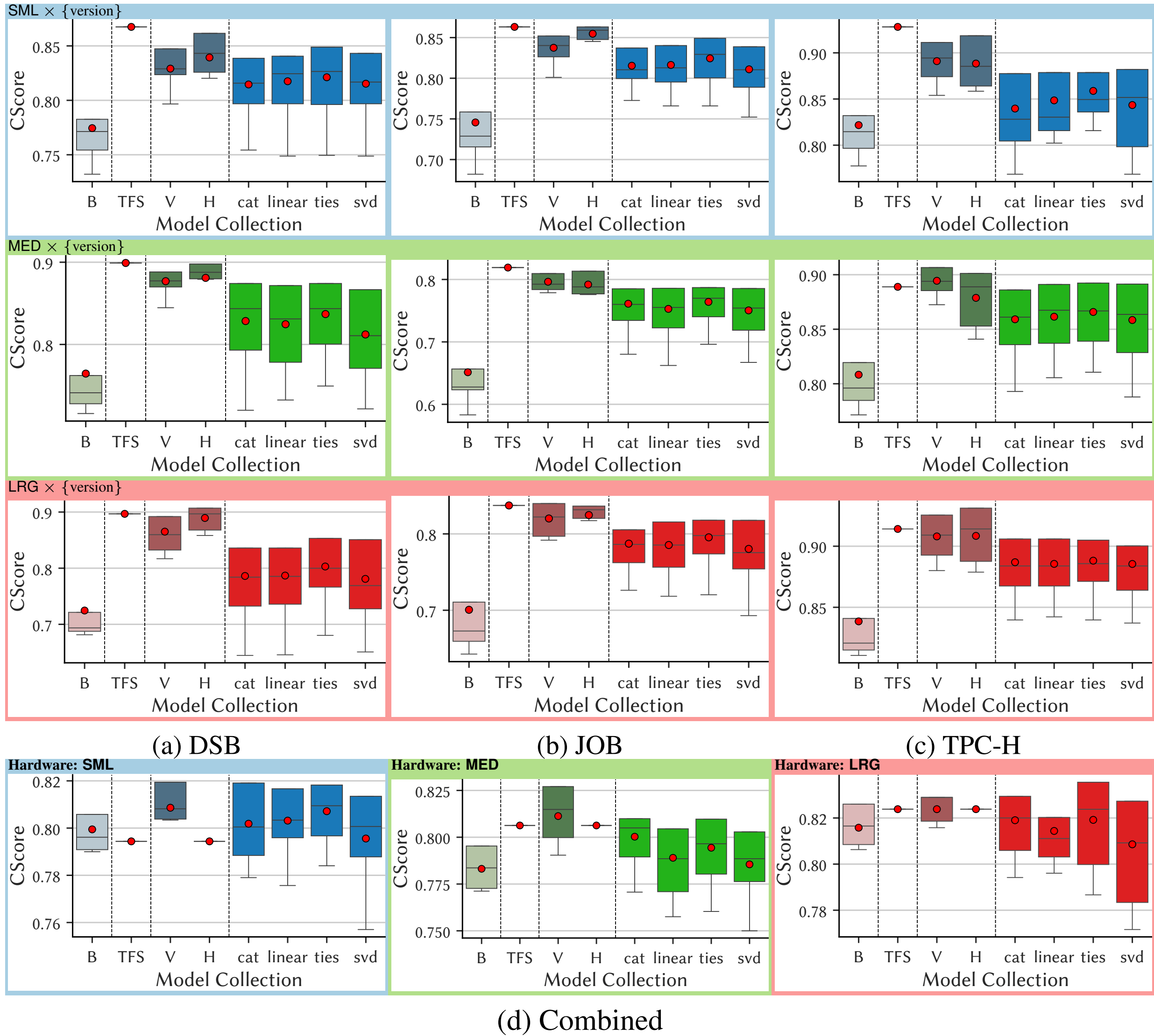


Figure 5.19: **Avoiding Version Upgrade Regret** – The CScore distribution of different model families for version upgrades.

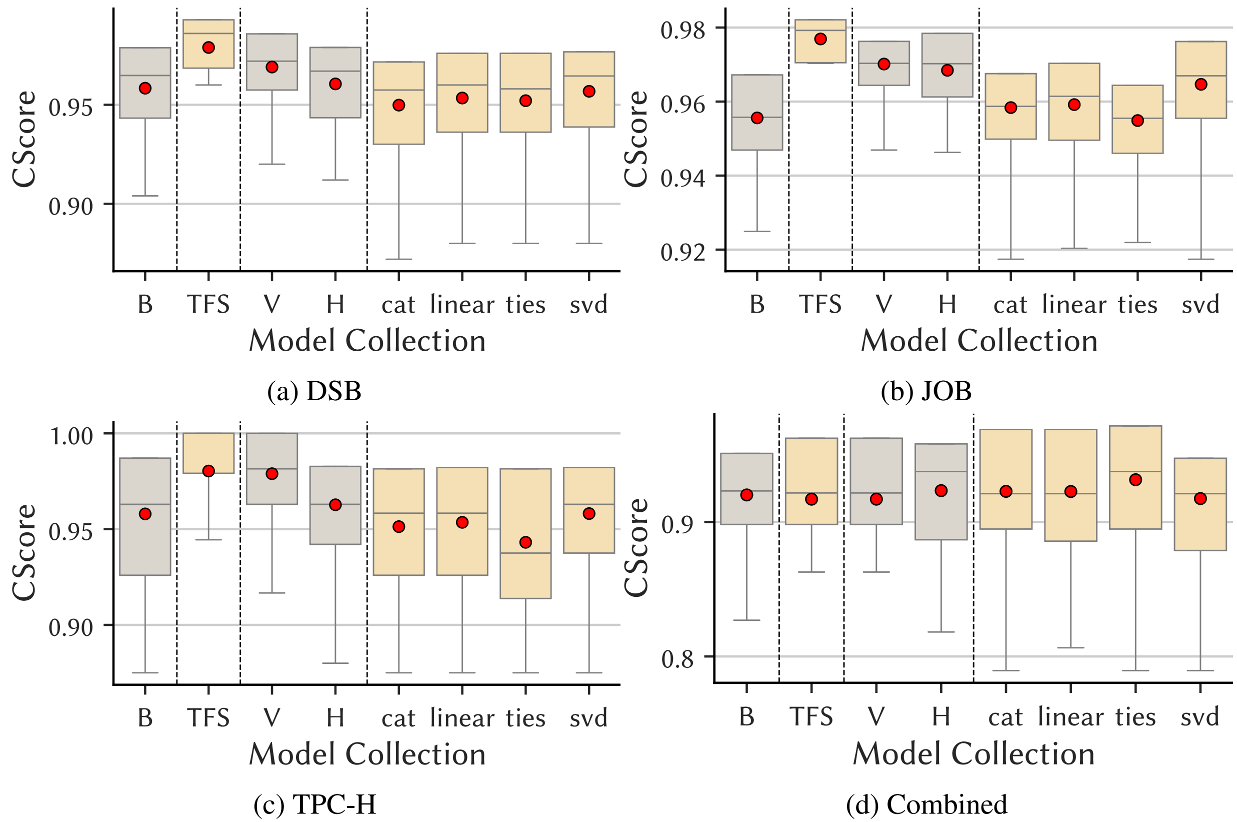


Figure 5.20: **Avoiding Hardware Upgrade Regret** – The CScore distribution of different model families for hardware upgrades.

CScore ranges from zero to one. Thus, we also define the *URegret* as $1 - CScore$: this is the fraction of queries that the upgrade advisor made a big mistake on (i.e., recommended upgrade when the optimal decision was to stay). We elide similar definitions for hardware upgrades. For the rest of this section, we visualize CScore similarly to the Multi-Δ Adaptation sections above. We focus our analysis on a model family's URegret (i.e., the fraction above the median line).

We first consider the CScore for version upgrades for schema-specific models. As Figure 5.19 shows, the URegret for *B* models is up to 20–37% across benchmarks. The 0Δ models approximately halve the URegret (up to 14–18%), with the 1Δ models close behind (up to 17–21%), followed by the less effective *cat* 2Δ models (up to 19–25%). Notably, in the median case we see that $^{\text{D}}_{\text{R}}$·DB is an improvement in all situations, achieving a moderate reduction in URegret for 2Δ and reaching near-optimal 0Δ performance for 1Δ. We see similar results when examining the URegret across hardware environments for version upgrades. In the Combined setting, the median URegret remains similar to the schema-specific models, but the gains from adaptation are more modest.

Next, we consider the CScore for hardware upgrades. We start by examining the schema-specific models. Unlike version upgrades, we see that the URegret is similar across benchmarks for most model families: up to 3.5–4.4% for *B* models, 0–2.1% for 0Δ models, 1.9–3.7% for 1Δ models, and 3.3–6.3% for *cat* 2Δ models. Only the 0Δ models (i.e., those trained for that exact environment) have a noticeable improvement on the URegret. However, the URegret was not high to begin with, which means $^{\text{D}}_{\text{R}}$·DB's hardware upgrade recommendations have little impact but also little harm. Hence, we find that $^{\text{D}}_{\text{R}}$·DB is better at avoiding upgrade regret for version-related regressions than hardware-related regressions. Likewise, the change in URegret for the Combined model is minimal.

5.6 Takeaways

We first recap the results from Section 5.5. We then translate our findings into practical strategies for building an upgrade recommender based on adapting pre-trained models. Lastly, we briefly discuss findings from other experiments.

We evaluated $^{\text{D}}_{\text{R}}$·DB on three upgrade-related concerns: (1) predicting total workload runtime, (2) predicting individual query latency, and (3) avoiding bad recommendations. Focusing on the schema-specific models, we showed that without $^{\text{D}}_{\text{R}}$·DB, models trained on a different environment had significantly worse performance (e.g., up to $246\times$ absolute error). Adapting such models with $^{\text{D}}_{\text{R}}$·DB resulted in errors that were no worse but often much better, with up to $31\times$ improvement in median absolute error and up to $108\times$ improvement in mean Q-error. We also found that adapting a model's hardware environment mattered more than adapting its software version.

The key design decision for an upgrade recommender is choosing what Δ to target (Section 5.5.1) as this determines the computational cost of training data collection and the extent of prediction errors. The status quo (i.e., B) uses models without regard to their version or hardware, which is cheap but inaccurate. A 0Δ strategy (Figure 5.21a) that trains models from scratch for every configuration is the best option, but this is computationally infeasible (see Section 5.2.1). Thus, $^{\text{D}}_{\text{R}}$·DB's main contribution is enabling and evaluating the 1Δ and 2Δ strategies.

```
for v in VERSIONS:
  for h in HARDWARES:
    # Collect training data.
    D = run_benchmark(v, h)
    # Instance-specific training.
    B[v,h] = train_model(D)




Serve M[v,h] = B[v,h].
```

(a) 0Δ (most expensive)

```
D = run_benchmark(VL, HL)
F = train_model(D)

for v in VERSIONS:
    D = run_benchmark(v, HL)
    V[v,HL] = make_lora(D, F)
for h in HARDWARES:
    D = run_benchmark(VL, h)
    H[VL,h] = make_lora(D, F)

Serve M[v,h] = F + V[v,HL] + H[VL,h]
```

(b) 2Δ (cheapest)

Figure 5.21: **Contrasting Upgrade Recommender Designs** – Pseudocode contrasting the cheapest and most computationally expensive Δ strategies, where `D` = training data, `B` = pre-trained models, `M` = the final models, `F` = foundation model, `VL` = latest version, `HL` = best hardware, `V` = version adapters, `H` = hardware adapters, and + applies adapters (see Section 5.3.2).

The 2Δ strategy (Figure 5.21b) requires the least amount of training data collection and model training as adapters are trained and composed independently across the dimensions of variation. Combining adapters with the *cat* strategy produces inexpensive yet effective models. However, given a larger computation and model storage budget, we can boost model performance by going to 1Δ. Our results suggest models should be trained for each hardware environment and adapted across software versions. For example, if there is a widely-used instance type, it is worthwhile to train a base model for that hardware and use adapters to handle DBMS versions. Adapting newer models to older versions is more effective, suggesting that we should periodically retrain and refresh the base hardware model.

Our other experiments showed that adapting different parts of the attention mechanism (e.g., only the query layer) yields similar performance. We also investigated additional train-test splits on JOB, including: 80-20 train-test split, 50-50 train-test split, normalized by query number (e.g., mapping “1a 1b 1c” to “1”), and normalized by suffix (e.g., mapping “1a 1b” to “1-front” and “1c 1d” to “1-back”).

From these experiments, we found that the most important factors were (1) the number of queries in a split (i.e., sufficient training data) and (2) whether a query template was seen before (i.e., the extent to training data is representative of the test data). We find empirically that LoRA improves in-distribution performance at the cost of worse out-of-distribution performance. In the upgrade setting, all queries are known and thus are in-distribution. Additionally, because most queries are repetitive in a real-world setting [193], we believe that training a new adapter periodically to “absorb” all the new query templates may generalize this work beyond upgrades. Given these results, we propose that each adapter is accompanied by a cheap bloom filter of query shapes that were present in the training data. Since $^{\text{D}}_{\text{R}}$·DB supports loading and unloading adapters at run-time, we believe that this will achieve the benefits of adaptation while sharply reducing the risk of degradation.

5.7 Conclusion

Despite rapid improvements to the DBMS software versions and hardware environments available, organizations are reluctant to upgrade. Existing upgrade processes focus on post-upgrade mitigations and do not provide organizations with the information that they need to justify upgrading. To address this, we introduce $^{\text{D}}_{\text{R}}$·DB, a tool for predicting performance changes pre-upgrade. Our tool adapts existing behavior models to new software versions and hardware environments, enabling the simulation of different upgrade scenarios. We show that in the upgrade context, $^{\text{D}}_{\text{R}}$·DB improves model performance by up to 31× for median total run-time prediction error and up to 108× for mean query-level Q-error. We also find empirically that *cat* is a reasonable default technique for combining multiple adapters.

Chapter 6

Related Work

The contributions of this dissertation build on decades of DBMS research: (1) autonomous DBMSs, (2) training data generation, and (3) training data generalization. This chapter surveys and situates our contributions in the context of those areas.

6.1 Autonomous DBMSs

Throughout the decades of autonomous DBMS research, ML models have always required training data. The early 2000s saw tools such as IBM DB2 Advisor [176], but these were often rule-based and system-specific. In recent years, the use of ML for DBMS automation has grown, though the extent of automation varies [153]. Examples include commercial systems such as Amazon Redshift [53, 147], Google AlloyDB [70], Oracle [144, 145], Oracle's MySQL HeatWave [146], Huawei openGauss [116], and Microsoft Azure SQL [74]. They also include academic systems such as NoisePage [151, 153] and SageDB [109]. As these systems depend on their internal ML models to guide operation, they benefit from better training data.

Recent work has focused on individual facets of the training data generation process, including overhead [61], location [74, 125], and quality [179, 184]. However, no existing system integrates these techniques nor solves the research and operational challenges described in Sections 3.1 and 3.3 to achieve a database gym environment that streamlines ML model training and evaluation. The most relevant related work to a database gym today is UDO [180], which we discuss below.

Gym Environment: UDO [180] is a tool that performs offline tuning for DBMSs. Given a workload and an environment, it uses reinforcement learning (RL) to find better DBMS configurations model-free, implementing the OpenAI Gym API to compare different algorithms. UDO focuses on developing novel algorithms [181] like the Decider uses, whereas the Gym synthesizes UDO's input (i.e., workload, state). We now discuss prior synthesis work.

Synthesizing Workloads: To our knowledge, existing research prior to the database gym did not synthesize an executable workload. They focused on individual workload aspects such as auto-scaling workloads for provisioning [73, 88, 160], DBMS performance modeling [137, 139,

200], next SQL statement prediction [79, 95, 96, 97, 149], workload compression [65], and query runtime metrics prediction [84, 90]. QueryBot5000 [124] argues that these are insufficient for predicting workload volume, duration, and change. Instead, it proposed predicting query arrival rates, but these too may be insufficient if raw query contents are required (e.g., for query-level features). Moreover, the Gym approximates future DBMS state by running individual queries, necessitating the synthesis of an executable workload. Recent years have placed more emphasis on synthesizing executable workloads, with Sibyl [99] forecasting sequences of executable queries for various prediction windows.

Synthesizing State: To our knowledge, no work tries to synthesize physical conditions for DBMSs. However, existing research synthesizes logical contents to (1) scale datasets and (2) fake data for testing. Dataset scaling [168] synthesizes new tuples based on either workload [161, 196] or individual table contents [202] while preserving cardinality constraints (e.g., data correlations). Faking data for testing uses techniques such as substitution rules [14, 16], differential privacy [175], and generative networks [170] to improve ML model performance [163] and allay privacy concerns [175]. The gym uses the above techniques to approximate future state with similar properties (e.g., join cardinalities) to improve training data.

Training Data Quality: DataFarm [179, 184] generates synthetic jobs from an input workload. It builds a model to predict job labels and ranks the jobs by the uncertainty of its predictions. Next, it uses active learning or asks a human to pick which job to evaluate. We note that operating at the level of query plans aids synthetic data generation but increases the sensitivity to the DBMS state. The Gym complements DataFarm by providing the input workload and the future state, as well as a location to run in.

Cosine [63] is a self-designing key-value storage engine that gathers high-quality training data to train its concurrency-aware CPU model. The model requires the workload's degree of parallelism to be known. Cosine learns this by sweeping across different factors (e.g., cloud instance type, number of operations, number of CPU cores) as it executes the workload. However, the expense of sweeping workloads and states for database tuning is prohibitive, necessitating the database gym's optimizations for accelerating training data generation.

Location: As Section 3.3 describes, Oracle [143] uses safeguards to try indexes on the primary DBMS, whereas Microsoft [74] tries indexes on B-instances (i.e., independent DBMS copies that receive and replay the primary's workload). The Gym balances between the two approaches by running on high-availability replicas.

6.2 Training Data Generation

To our knowledge, we are the first to accelerate training data generation for autonomous DBMSs by decoupling it from regular query execution. We now discuss areas of related work.

Query Progress Estimation (QPE) The problem of QPE was first formally defined and studied for Microsoft SQL Server [66] and PostgreSQL [122]. Their idea is to decompose the query plan tree into individual pipelines and then estimate overall pipeline progress using the driver nodes [107]. This decomposition made the estimation problem tractable because the true cardinalities of the driver nodes are easier to obtain (e.g., table scan). As behavior models [126, 130] require operator-level data, this early work is not applicable to Boot. Microsoft SQL Server LQS [113] extended QPE to provide logical operator-level completion estimates (e.g., predicting that a hash join operator has made 42% of its total `GetNext()` calls).

Another difference between Boot and prior work in QPE is that the latter aims to help humans debug query performance. The DBMS must still execute the entire query to produce the correct result. In contrast, Boot only need to execute enough of the query for its μG module to generate approximate operator telemetry.

However, key ideas shared by QPE and Boot are (1) viewing the tuples processed so far as a random sample of the tuples available [113] and (2) using execution feedback to refine initial estimates [189]. For example, existing work in QPE and re-optimization uses random sampling to improve uncertainty [55], cardinality [189] and selectivity [91] estimates. These better estimates are complementary to Boot, which operates in a new training data context.

Approximate Query Processing (AQP) AQP provides faster query results by sampling data [68, 136, 142]. Unlike AQP, Boot does not care about the correctness of the query result. However, AQP techniques are useful for obtaining more accurate execution behavior (e.g., the selectivity of a join operator's predicate) and correcting cardinality estimates (e.g., μG's telemetry scaling). Although Boot and AQP are complementary, we envision that ideas from AQP will find new applications in accelerating training data generation.

Training Data Collection Prior work that improves the training data collection process focuses on optimizing instrumentation overhead [61] or reducing the quantity of training data that needs to be collected. For example, using active learning [179] or index-aware similarity [166, 167] reduces the number of queries executed, budget-aware tuning [183, 190] reduces the number of optimizer what-if calls, and incremental model construction [81] allows stopping training data collection early. However, to our knowledge, no techniques exploit the training data setting to accelerate query execution itself.

6.3 Training Data Generalization

To our knowledge, we are the first to focus on automated upgrade recommendations for self-driving DBMSs and likewise the first to use fine-tuning techniques to adapt DBMS behavior models across upgrade environments. We now discuss related topics.

DBMS Upgrades: Existing upgrade tools only provide operational safeguards (e.g., deprecated types [25]) and post-upgrade mitigations (e.g., query result equivalence [155], detecting regressions [83]). To use these tools, operators must provision a staging environment, capture

and replay workloads, and then triage any regressions post-upgrade. Existing tools do not help an operator decide whether they should upgrade. Our work complements these tools by predicting performance changes pre-upgrade.

Behavior Models: Behavior models predict DBMS performance under the different configurations that are supported by their input features (see Section 5.1.3). For example, behavior models replace expensive query execution with predicted latencies [110], enabling tasks such as automated database tuning [204]. However, existing models are instance-specific [185] and do not support "what-if" scenarios beyond those explicitly captured by their input features (see Section 5.2.1). Subsequently, existing models often require expensive retraining [126, 130] to handle new environments. Our work provides a method for adapting models across multiple dimensions without needing to modify their input features, reducing the need to retrain from scratch. Our work does not contribute a new type of model, but rather enhances existing behavior models for new use cases.

Mitigating Learned Component Regressions: Learned query optimizers [131, 132, 140, 212] use ML to replace or augment the traditional optimizer, often generating much better plans at the cost of potential regressions. To mitigate such regressions, the system may execute new plans offline to verify their superior performance [51, 199, 203] or switch back to the traditional plan in high-risk out-of-distribution scenarios [187]. Such efforts combine judicious re-training, offline execution, and careful fallbacks to non-learned logic to reduce or eliminate regressions entirely. Our work is orthogonal to these efforts as we focus on adapting models across environments to improve prediction accuracy.

Chapter 7

Future Work

In Chapter 3, we proposed the database gym, a framework that accelerates the construction and deployment of tool-based database tuning pipelines. Chapter 4 then examined techniques for accelerating the database tuning process through faster training data generation. Chapter 5 subsequently proposed techniques that enabled reusing training data to adapt tools across different deployment environments. Together, these techniques allow human operators to invoke individual database tuning tools quickly and efficiently with the assistance of the database gym.

Given the preceding chapters, the remaining bottleneck in the database tuning pipeline is the human operator's involvement in tool selection. The database gym makes it simpler and faster to construct and deploy a tool's tuning pipeline after a tool has been chosen, but a human operator must still manually (1) decide on an objective to optimize and (2) select a tool to invoke. We propose to leverage recent developments in agentic artificial intelligence (AI) to enable autonomous tool orchestration by substituting the human operator with an AI agent, eliminating human involvement in selecting and invoking tools. Our vision is to enable human operators to focus on high-level database tuning guidance (e.g., "reduce storage costs") instead of laboriously coordinating the minutiae of low-level tool operation.

Recent research on using large language models (LLMs) for database tuning has used LLMs to debug certain performance regressions [86] and optimize individual queries [87], demonstrating the potential of LLM-based reasoning. Building on these results, we envision an approach that operates on a higher level of abstraction than the current database gym, in which an agent is responsible for autonomously creating optimization sub-objectives and deploying the appropriate tools. To do so, we must address the following challenges:

Objective and Tool Identification: Unlike existing LLM-based tuning work that optimizes the DBMS with the LLM directly, we aim to use the LLM to identify suitable sub-objectives and select appropriate tools. However, there is no consensus on how to determine when a DBMS has been sufficiently optimized. Commercial vendors have attempted to dynamically compute *health scores* [154], which aggregate multiple factors and compare them to other similar database deployments. Inspired by this idea, one potential approach is to develop specialized environments within the gym with known maximum "health scores", allowing an AI agent to learn when a deployment has reached the objective for a given environment.

Semantic Tool Interfaces for AI-native Orchestration: Existing tools were designed for human operators and therefore assume that the operator will understand the context in which the tool should be used. Thus, for an AI agent to orchestrate them effectively, it must understand their semantics (e.g., capabilities, limitations, expected runtime). We propose investigating the use of the recently developed Model Context Protocol (MCP) [52] as a method for exposing the semantics of tuning tools to AI agents.

Safety Through Simulation: Existing tools make recommendations without ensuring that they are safe or beneficial, leaving those responsibilities to the operator. Thus, an AI agent requires guardrails to automatically apply recommendations in a safe manner. We propose building these guardrails by leveraging the database gym, which uses a DBMS to safely and quickly simulate the database system's behavior under various configurations. For example, our extensions to PostgreSQL allow generating query plans with only schema and statistics information (i.e., without loading any data). Future work can combine this with the gym's adapted behavior models from Chapter 5 to identify queries where performance is likely to change significantly.

Chapter 8

Conclusion

In this dissertation, we examined the gap in autonomous capabilities between self-driving DBMSs and tool-based database tuning pipelines. We showed that although existing tools are effective, they require significant human effort to configure and deploy. Furthermore, because existing tools are designed for human operators, their semantics are inefficient for autonomous operation. Thus, we proposed the database gym, a framework that decomposes and systematizes the database tuning pipeline to enable efficient autonomous orchestration of these tools. Our initial implementation revealed that autonomous tuning pipelines were bottlenecked by training data generation and reuse, motivating a training-data-centric approach to database tuning.

We developed two novel optimizations that leverage training data's unique properties to accelerate database tuning within the database gym. First, Boot accelerates training data generation by decoupling telemetry collection from ordinary query execution, reducing redundant computation to expedite tool calibration. Next, ${}^{\mathsf{D}}_{\mathsf{R}}$·DB generalizes training data across environments by extracting salient characteristics into composable and swappable adapters, eliminating the need to recalibrate tools from scratch whenever a deployment's software version or hardware configuration evolves. These optimizations accelerate the tuning loop and improve the efficiency of autonomous tool pipelines.

Taken as a whole, the work presented in this dissertation demonstrates the performance and generalization benefits of systematically organizing the database tuning pipeline around training data management within the database gym. By elevating training data from a tuning byproduct to a first-class systems concern, we show how to accelerate the deployment and use of existing tools to impart autonomous capabilities without requiring a complete redesign of the DBMS.

Bibliography

[1] TPC-H dbgen, 2011. URL https://github.com/electrum/tpch-dbgen. 4.6.1

[2] pgsql-hackers: TPC-H Q20 from 1 hour to 19 hours!, 2017. URL https://www.postgresql.org/message-id/CAOGQiiMsnM-cGjCfnpWQNyrzjf%3DFM0s_3hn1guqficSmwWWVfw%40mail.gmail.com. 5.5.1

[3] PostgreSQL Release 10, 2017. URL https://www.postgresql.org/docs/10/release-10.html. 5.5.1

[4] INTEL NUC KITS NUC7i7DNKE & NUC7i7DNHE, 2018. URL https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/nuc-kit-nuc7i7dnke-nuc7i7dnhe-board-nuc7i7dnbe-brief.pdf. 5.5

[5] PostgreSQL Release 11, 2018. URL https://www.postgresql.org/docs/11/release-11.html. 5, 5.5.1

[6] The Bitter Lesson, 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html. 5.2.1

[7] TPC-H performance since PostgreSQL 8.3, 2020. URL https://www.enterprisedb.com/blog/tpc-h-performance-postgresql-83. 5.1.1, 5.5.1

[8] How to avoid performance regression after upgrading Oracle Database, 2020. URL https://blogs.oracle.com/optimizer/post/upgrade-to-oracle-database-12c-and-avoid-query-regression. 5

[9] Allocate consecutive blocks during parallel seqscans, 2020. URL https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=56788d215. 5.1.1

[10] Right Sizing: Provisioning Instances to Match Workloads, 2020. URL https://docs.aws.amazon.com/whitepapers/latest/cost-optimization-right-sizing/cost-optimization-right-sizing.html. 5

[11] PostgreSQL Release 14, 2021. URL https://www.postgresql.org/docs/14/release-14.html. 5

[12] pgmustard: Calculating per-operation times in EXPLAIN ANALYZE, 2021. URL https://www.pgmustard.com/blog/calculating-per-operation-times-in-postgres-explain-analyze. 4.6.6

[13] BenchBase – Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase, 2022. 3.1.2, 3.3

[14] clickhouse-obfuscator. https://clickhouse.com/docs/en/operations/utilities/clickhouse-obfuscator/, 2022. 6.1

[15] PostgreSQL Upgrades Are Hard, 2022. URL https://andreas.scherbaum.la/post/2022-06-22_postgresql-upgrades-are-hard/. 5

[16] Replibyte – Seed your development database with real data. https://www.replibyte.com/docs/introduction, 2022. 6.1

[17] Wikipedia: Database schema. https://en.wikipedia.org/wiki/Wikipedia:Database_download#SQL_schema, 2022. 2.2, 3.1.3

[18] Llama 2: Inference code for LLaMA models, 2023. URL https://github.com/facebookresearch/llama. 4.1.1

[19] SQL Server Distributed Replay overview, 2023. URL https://learn.microsoft.com/en-us/sql/tools/distributed-replay/sql-server-distributed-replay. 5

[20] Join containment assumption in the New Cardinality Estimator degrades query performance, 2023. URL https://learn.microsoft.com/en-us/troubleshoot/sql/database-engine/performance/cardinality-estimator-degrades-query-performance. 5

[21] Decreased query performance after upgrade from SQL Server 2012 or earlier to 2014 or later, 2023. URL https://learn.microsoft.com/en-us/troubleshoot/sql/database-engine/performance/decreased-query-perf-after-upgrade. 5

[22] Optimized plan forcing with Query Store, 2024. URL https://learn.microsoft.com/en-us/sql/relational-databases/performance/optimized-plan-forcing-query-store?view=sql-server-ver17. 5.1.2

[23] Monitor performance by using the Query Store, 2024. URL https://learn.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store. 3.1.1, 5

[24] Plan and test the Database Engine upgrade plan, 2024. URL https://learn.microsoft.com/en-us/sql/database-engine/install-windows/plan-and-test-the-database-engine-upgrade-plan?view=sql-server-ver16. 5

[25] MySQL Upgrade Checker Utility, 2024. URL https://dev.mysql.com/doc/mysql-shell/8.4/en/mysql-shell-utilities-upgrade.html. 5, 6.3

[26] Why Does Everyone Run Ancient Postgres Versions?, 2024. URL https://neon.tech/blog/why-does-everyone-run-ancient-postgres-versions. 5, 5.1.1

[27] Clock's ticking on PostgreSQL 12, but not everyone is ready to say goodbye, 2024. URL https://www.theregister.com/2024/10/22/postgresql_12_eol/. 5

[28] How Postgres is Misused and Abused in the Wild, 2024. URL https://karenjex.blogspot.com/2024/07/how-postgres-is-misused-and-abused.html. 5

[29] PostgreSQL Release 17, 2024. URL https://www.postgresql.org/docs/17/release-17.html. 5

[30] PostgreSQL: Documentation: 17: 28.5 WAL Configuration, 2024. URL https://www.postgresql.org/docs/17/wal-configuration.html. 2.1

[31] Amazon EC2 Instance Type Explorer, 2025. URL https://aws.amazon.com/ec2/instance-explorer/. 5.1.1

[32] Document history for the Amazon EC2 Instance Types Guide, 2025. URL https://docs.aws.amazon.com/ec2/latest/instancetypes/doc-history.html. 5.1.1

[33] Amazon EC2 T3 Micro Instance, 2025. URL https://aws.amazon.com/ec2/instance-types/t3/. 5.5

[34] Google Cloud SQL for PostgreSQL: Database versions and version policies, 2025. URL https://cloud.google.com/sql/docs/postgres/db-versions. 5.1.1

[35] OPTIMIZER_FEATURES_ENABLE, 2025. URL https://docs.oracle.com/en/database/oracle/oracle-database/21/refrn/OPTIMIZER_FEATURES_ENABLE.html. 5.1.2

[36] Downgrading Oracle Database to an Earlier Release, 2025. URL https://docs.oracle.com/en/database/oracle/oracle-database/26/upgrd/downgrading-oracle-db-after-upgrade.html. 5

[37] pg_hint_plan, 2025. URL https://github.com/ossc-db/pg_hint_plan. 2.2, 5.1.2

[38] PostgreSQL Security Information, 2025. URL https://www.postgresql.org/support/security/. 5

[39] PostgreSQL Versioning Policy, 2025. URL https://www.postgresql.org/support/versioning/. 5.1.1

[40] pgtune - tuning PostgreSQL config by your hardware, 2025. URL https://github.com/le0pard/pgtune. 1, 2.2, 2.3, 3.2.3, 4.6, 5.5

[41] L1Loss, 2025. URL https://docs.pytorch.org/docs/stable/generated/torch.nn.L1Loss.html. 5.5.2

[42] MSELoss, 2025. URL https://docs.pytorch.org/docs/stable/generated/torch.nn.MSELoss.html. 5.5.3

[43] torch.nn, 2025. URL https://docs.pytorch.org/docs/stable/nn.html. 5.4

[44] Upgrading a DB instance engine version, 2025. URL https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_

UpgradeDBInstance.Upgrading.html. 5.1.1

[45] SuperServer 2029BT-HNTR, 2025. URL https://www.supermicro.com/products/system/2U/2029/SYS-2029BT-HNTR.cfm. 5.5

[46] DCAI 2021. NeurIPS Data-Centric AI Workshop, 2021. URL https://datacentricai.org/neurips21/. 4.6.2

[47] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating Vertical and Horizontal Partitioning intoAutomated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 359–370, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138598. doi: 10.1145/1007568.1007609. URL https://doi.org/10.1145/1007568.1007609. 1, 2.2

[48] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.*, 14(7):1241–1253, 2021. URL https://db.cs.cmu.edu/papers/2021/p1241-aken.pdf. 1, 2.3, 3.1.1

[49] Selim Amrouni, Aymeric Moulin, Jared Vann, Svitlana Vyetrenko, Tucker Balch, and Manuela Veloso. ABIDES-Gym: Gym Environments for Multi-Agent Discrete Event Simulation and Application to Financial Markets. In *ICAIF'21*, pages 30:1–30:9. ACM, 2021. 3, 3.2

[50] ankane. Dexter: The automatic indexer for Postgres, 2025. URL https://github.com/ankane/dexter. 1, 2.2, 3.1.3

[51] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.*, 16(12):3515–3527, August 2023. ISSN 2150-8097. doi: 10.14778/3611540.3611544. URL https://doi.org/10.14778/3611540.3611544. 6.3

[52] Anthropic. Model Context Protocol, 2025. URL https://modelcontextprotocol.io. 7

[53] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. Amazon Redshift Reinvented. In *SIGMOD '22*, pages 2205–2217. ACM, 2022. 6.1

[54] Aurora. Scaling Simulation. https://aurora.tech/blog/scaling-simulation, 2021. 3

[55] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 107–118, New York, NY, USA, 2005. Association for Comput-

ing Machinery. ISBN 1595930604. doi: 10.1145/1066157.1066171. URL https://doi.org/10.1145/1066157.1066171. 6.2

[56] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077. 3.4

[57] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *J. Artif. Intell. Res.*, 47: 253–279, 2013. 3.2

[58] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ B. Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri S. Chatterji, Annie S. Chen, Kathleen Creel, Jared Quincy Davis, Dorottya Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren E. Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, Rohith Kuditipudi, and et al. On the Opportunities and Risks of Foundation Models. *CoRR*, abs/2108.07258, 2021. URL https://arxiv.org/abs/2108.07258. 5.2.1, 5.3.1, 5.4

[59] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016. 3, 3.2

[60] Matteo Brucato, Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. Wred: Workload Reduction for Scalable Index Tuning. *Proc. ACM Manag. Data*, 2 (1 (SIGMOD)), February 2024. doi: 10.1145/3639305. URL https://doi.org/10.1145/3639305. 4.6.4

[61] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 617–630, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3517845. URL https://doi.org/10.1145/3514221.3517845. 1.1, 2.3, 3.1, 3.4, 4, 4.1.1, 4.1.2, 4.6.6, 5.1.3, 5.2.2, 6.1, 6.2

[62] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A Database Proxy That Bounces With User-Bypass. *Proc. VLDB Endow.*, 16(11):3335–3348, 2023. doi: 10.14778/3611479.3611530. 5.1.2

[63] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. *Proc. VLDB Endow.*, 15(1):

112–126, 2021. 6.1

[64] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 146–155. Very Large Data Bases Endowment Inc., 1997. 1, 2, 5.1.3

[65] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. Compressing SQL workloads. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 488–499. ACM, 2002. 3.1.1, 6.1

[66] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for SQL queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 803–814, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138598. doi: 10.1145/1007568.1007659. URL https://doi.org/10.1145/1007568.1007659. 6.2

[67] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. When can we trust progress estimators for SQL queries? In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 575–586, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930604. doi: 10.1145/1066157.1066223. URL https://doi.org/10.1145/1066157.1066223. 4.1.3, 4.3, 4.4

[68] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 511–519, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3056097. URL https://doi.org/10.1145/3035918.3056097. 6.2

[69] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785. 4.6.1

[70] Google Cloud. Introducing AlloyDB for PostgreSQL: Free yourself from expensive, legacy databases. https://cloud.google.com/blog/products/databases/introducing-alloydb-for-postgresql, 2022. 6.1

[71] Google Cloud. AutoML. https://cloud.google.com/automl/, 2022. 3

[72] Piali Das, Nikita Ivkin, Tanya Bansal, Laurence Rouesnel, Philip Gautier, Zohar S. Karnin, Leo Dirac, Lakshmi Ramakrishnan, Andre Perunicic, Iaroslav Shcherbatyi, Wilton Wu, Aida Zolic, Huibin Shen, Amr Ahmed, Fela Winkelmolen, Miroslav Miladinovic, Cédric Archambeau, Alex Tang, Bhaskar Dutt, Patricia Grao, and Kumar Venkateswar. Amazon SageMaker Autopilot: a white box AutoML solution at scale. In *DEEM@SIGMOD 2020*, pages 2:1–2:7. ACM, 2020. 3

[73] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *SIGMOD Con-*

ference 2016, pages 1923–1934. ACM, 2016. 6.1

[74] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 666–679, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi: 10.1145/3299869.3314035. URL https://doi.org/10.1145/3299869.3314035. 2, 3.2.1, 3.3, 4.1.1, 5.1.3, 6.1, 6.1

[75] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey F. Naughton, and Stratis Viglas. Comprehensive and Efficient Workload Compression. *Proc. VLDB Endow.*, 14(3):418–430, 2020. 3.1.1

[76] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.*, 7(4):277–288, 2013. 3.1.2, 3.1.3, 3.3, 3.4

[77] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-Example: An Automatic Query Steering Framework for Interactive Data Exploration. In *SIGMOD 2014*, pages 517–528. ACM, 2014. 3.1.1

[78] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.*, 14(13):3376–3388, September 2021. ISSN 2150-8097. doi: 10.14778/3484224.3484234. URL https://doi.org/10.14778/3484224.3484234. 3.4, 4, 4.2, 4.6.1, 5.5.1

[79] Naiqiao Du, Xiaojun Ye, and Jianmin Wang. Towards workflow-driven database system workload modeling. In *DBTest 2009*. ACM, 2009. 6.1

[80] Gabriel Dulac-Arnold, Richard Evans, Peter Sunehag, and Ben Coppin. Reinforcement Learning in Large Discrete Action Spaces. *CoRR*, abs/1512.07679, 2015. 3.2.3

[81] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. Efficiently Approximating Selectivity Functions using Low Overhead Regression Models. *Proc. VLDB Endow.*, 13(12):2215–2228, July 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407820. URL https://doi.org/10.14778/3407790.3407820. 6.2

[82] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *arXiv preprint arXiv:2003.06505*, 2020. 3, 3.4, 4.6.1

[83] Leonidas Galanis, Supiti Buranawatanachoke, Romain Colle, Benoît Dageville, Karl Dias, Jonathan Klein, Stratos Papadomanolakis, Leng Leng Tan, Venkateshwaran Venkataramani, Yujun Wang, and Graham Wood. Oracle Database Replay. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1159–1170, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581026. doi: 10.1145/1376616.1376732. URL https://doi.org/10.1145/1376616.1376732. 3.1.1, 4.1.1, 6.3

[84] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE 2009*, pages 592–603. IEEE Computer Society, 2009. 6.1

[85] Piotr Gawlowicz and Anatolij Zubow. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *MSWiM 2019*, pages 113–120. ACM, 2019. 3, 3.2

[86] Victor Giannakouris and Immanuel Trummer. DBG-PT: A Large Language Model Assisted Query Performance Regression Debugger. *Proceedings of the VLDB Endowment*, 17(12):4337–4340, 2024. 7

[87] Victor Giannakouris and Immanuel Trummer. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proceedings of the ACM on Management of Data*, 3(1):1–26, 2025. 7

[88] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *CNSM 2010*, pages 9–16. IEEE, 2010. 6.1

[89] Goetz Graefe. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994. ISSN 1041-4347. doi: 10.1109/69.273032. URL https://doi.org/10.1109/69.273032. 4.4.1

[90] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. PQR: predicting query execution times for autonomous workload management. In *ICAC 2008*, pages 13–22. IEEE Computer Society, 2008. 6.1

[91] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Comput. Syst. Sci.*, 52(3):550–569, June 1996. ISSN 0022-0000. doi: 10.1006/jcss.1996.0041. URL https://doi.org/10.1006/jcss.1996.0041. 6.2

[92] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 143–157, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389704. URL https://doi.org/10.1145/3318464.3389704. 2, 2.2

[93] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.*, 13(7):992–1005, 2020. doi: 10.14778/3384345.3384349. URL http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf. 4, 5.2.2

[94] Marc Holze and Norbert Ritter. Towards workload shift detection and prediction for autonomic databases. In *PIKM 2007*, pages 109–116. ACM, 2007. 3.1.1

[95] Marc Holze and Norbert Ritter. Autonomic Databases: Detection of Workload Shifts with n-Gram-Models. volume 5207 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 2008. 6.1

[96] Marc Holze, Claas Gaidies, and Norbert Ritter. Consistent on-line classification of dbs

workload events. In *CIKM 2009*, pages 1641–1644. ACM, 2009. 6.1

[97] Marc Holze, Ali Haschimi, and Norbert Ritter. Towards workload-aware self-management: Predicting significant workload shifts. In *ICDE 2010*, pages 111–116. IEEE Computer Society, 2010. 6.1

[98] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. LoRA: Low-Rank Adaptation of Large Language Models. *ICLR*, 1(2):3, 2022. 5.3.1

[99] Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesús Camacho-Rodríguez, and Yuanyuan Tian. Sibyl: Forecasting Time-Evolving Query Workloads. In *SIGMOD*, June 2024. URL `https://www.microsoft.com/en-us/research/publication/sibyl-forecasting-time-evolving-query-workloads/`. 2.1, 3.2.1, 6.1

[100] Rob J Hyndman and George Athanasopoulos. Forecasting: Principles and Practice, 3rd edition, 2021. URL `https://otexts.com/fpp3/`. 4.3.2

[101] HypoPG. hypopg: Hypothetical Indexes for PostgreSQL, 2025. URL `https://github.com/HypoPG/hypopg`. 2.1, 2.2

[102] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, September 1984. ISSN 0362-5915. doi: 10.1145/1270.1498. URL `https://doi.org/10.1145/1270.1498`. 1, 2.2

[103] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric Query Optimization. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, page 103–114, 1992. ISBN 1558601511. 4.1.3

[104] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: a highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3149–3157, 2017. ISBN 9781510860964. 4.6.1

[105] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL `http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf`. 5.5.1

[106] Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *IROS 2004*, volume 3, pages 2149–2154. IEEE, 2004. 3

[107] Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. A Statistical Approach Towards Robust Progress Estimation. *Proc. VLDB Endow.*, 5(4):382–393, December 2011. ISSN 2150-8097. doi: 10.14778/2095686.2095696. 6.2

[108] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of In-

dex Selection Algorithms. *Proc. VLDB Endow.*, 13(12):2382–2395, jul 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407832. URL https://doi.org/10.14778/3407790.3407832. 2

[109] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A Learned Database System. In *CIDR 2019*. www.cidrdb.org, 2019. 6.1

[110] Tim Kraska, Tianyu Li, Samuel Madden, Markos Markakis, Amadou Ngom, Ziniu Wu, and Geoffrey X. Yu. Check Out the Big Brain on BRAD: Simplifying Cloud Data Processing with Learned Automated Data Meshes. *Proc. VLDB Endow.*, 16(11): 3293–3301, July 2023. ISSN 2150-8097. doi: 10.14778/3611479.3611526. URL https://doi.org/10.14778/3611479.3611526. 6.3

[111] Eva Kwan, Sam Lightstone, K. Bernhard Schiefer, Adam J. Storm, and Leanne Wu. Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. volume P-26 of *LNI*, pages 620–629. GI, 2003. 3.1.1

[112] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A Framework for Reinforcement Learning in Games. *CoRR*, abs/1908.09453, 2019. URL http://arxiv.org/abs/1908.09453. 3.4

[113] Kukjin Lee, Arnd Christian König, Vivek Narasayya, Bolin Ding, Surajit Chaudhuri, Brent Ellwein, Alexey Eksarevskiy, Manbeen Kohli, Jacob Wyant, Praneeta Prakash, Rimma Nehme, Jiexing Li, and Jeff Naughton. Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1753–1764, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2903728. URL https://doi.org/10.1145/2882903.2903728. 4.4, 6.2

[114] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal*, 27(5):643–668, October 2018. ISSN 1066-8888. doi: 10.1007/s00778-017-0480-7. URL https://doi.org/10.1007/s00778-017-0480-7. 4.6.1, 4.6.2, 4.6.2, 5, 5.1.1, 5.5.1

[115] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.*, 12(12):2118–2130, August 2019. ISSN 2150-8097. doi: 10.14778/3352063.3352129. URL https://doi.org/10.14778/3352063.3352129. 2, 4.1.2, 5.1.3

[116] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing

Wang, and Shifu Li. openGauss: An Autonomous Database System. *Proc. VLDB Endow.*, 14(12):3028–3041, 2021. 6.1

[117] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *Proc. VLDB Endow.*, 5(11):1555–1566, July 2012. ISSN 2150-8097. doi: 10.14778/2350229.2350269. URL https://doi.org/10.14778/2350229.2350269. 5.5.3

[118] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray RLLib: A Composable and Scalable Reinforcement Learning Library. *CoRR*, abs/1712.09381, 2017. 3.2

[119] Wan Shen Lim, Matthew Butrovich, William Zhang, Andrew Crotty, Lin Ma, Peijing Xu, Johannes Gehrke, and Andrew Pavlo. Database Gyms. In *CIDR 2023, Conference on Innovative Data Systems Research*, 2023. 1, 1.3, 2.3, 4, 4.1, 4.1.1, 4.6.1, 5.2.1

[120] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel I Arch, and Andrew Pavlo. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. *Proc. VLDB Endow.*, 17(11):3680–3693, 2024. URL https://www.vldb.org/pvldb/vol17/p3680-lim.pdf. 1.1, 1.3, 5.5.2

[121] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988. 3.3

[122] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 791–802, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138598. doi: 10.1145/1007568.1007658. URL https://doi.org/10.1145/1007568.1007658. 6.2

[123] Lin Ma. *Self-Driving Database Management Systems: Forecasting, Modeling, and Planning*. PhD thesis, Ph. D. Dissertation. Carnegie Mellon University, 2021. 2.1

[124] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 631–645, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3196908. URL https://doi.org/10.1145/3183713.3196908. 1, 2.1, 3.1.1, 4.1.1, 4.2, 4.2.1, 4.3.1, 6.1

[125] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. Active Learning for ML Enhanced Database Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 175–191, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389768. URL https://doi.org/10.1145/3318464.3389768. 2.3, 3.3, 4.1.1, 4.1.2, 6.1

[126] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. MB2: Decomposed Behavior Modeling for Self-

Driving Database Management Systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1248–1261, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457276. URL https://doi.org/10.1145/3448016.3457276. 1, 2, 2.1, 3, 3.1.1, 3.4, 4, 4.1, 4.1.1, 4.1, 4.2, 4.1.3, 4.6.2, 4.6.2, 4.6.2, 5, 5.1.3, 6.2, 6.3

[127] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft, 2022. 5, 5.2.2

[128] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Bojja Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Mohammad Alizadeh. Park: An Open Platform for Learning-Augmented Computer Systems. In *NeurIPS 2019*, pages 2490–2502, 2019. 3

[129] Ryan Marcus. Ten years of improvements in PostgreSQL's optimizer, 2024. URL https://rmarcus.info/blog/2024/04/12/pg-over-time.html. 5

[130] Ryan Marcus and Olga Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, July 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342646. URL https://doi.org/10.14778/3342263.3342646. 1, 2, 2.3, 3, 3.1.1, 3.2.1, 3.4, 4, 4.1.1, 4.1, 4.2, 4.1.3, 4.3.1, 4.6.1, 4.6.2, 4.6.2, 4.6.2, 4.6.2, 4.6.4, 5.1.3, 5.5.3, 6.2, 6.3

[131] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342644. URL https://doi.org/10.14778/3342263.3342644. 4, 4.1.1, 5.1.3, 6.3

[132] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1275–1288, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3452838. URL https://doi.org/10.1145/3448016.3452838. 1, 2, 2.2, 3, 4.1.2, 4.2, 5.1.3, 6.3

[133] Microsoft. AutoML. https://www.microsoft.com/en-us/research/project/automl/, 2022. 3

[134] Microsoft. Automatic tuning, 2024. URL https://learn.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning. 2.1

[135] Microsoft. Overview of the Query Hint Recommendation Tool, 2025. URL https://learn.microsoft.com/en-us/ssms/query-hint-tool/hint-tool-overview. 2.2

[136] Barzan Mozafari. Approximate Query Engines: Commercial Challenges and Research Opportunities. In *Proceedings of the 2017 ACM International Conference on Manage-*

ment of Data, SIGMOD '17, page 521–524, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3056098. URL https://doi.org/10.1145/3035918.3056098. 6.2

[137] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD 2013*, pages 301–312. ACM, 2013. 6.1

[138] Avanika Narayan, Ines Chami, Laurel J. Orr, and Christopher Ré. Can Foundation Models Wrangle Your Data? *Proc. VLDB Endow.*, 16(4):738–746, 2022. doi: 10.14778/3574245.3574258. URL https://www.vldb.org/pvldb/vol16/p738-narayan.pdf. 5.2.1

[139] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *MASCOTS 2005*, pages 239–248. IEEE Computer Society, 2005. 6.1

[140] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2557–2569, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457568. URL https://doi.org/10.1145/3448016.3457568. 4.1.2, 5.1.3, 6.3

[141] Rimma Nehme and Nicolas Bruno. Automated Partitioning Design in Parallel Database Systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1137–1148, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306614. doi: 10.1145/1989323.1989444. URL https://doi.org/10.1145/1989323.1989444. 2.2

[142] Supriya Nirkhiwale, Alin Dobra, and Christopher Jermaine. A Sampling Algebra for Aggregate Estimation. *Proc. VLDB Endow.*, 6(14):1798–1809, September 2013. ISSN 2150-8097. doi: 10.14778/2556549.2556563. URL https://doi.org/10.14778/2556549.2556563. 6.2

[143] Oracle. Autonomous Indexing. https://blogs.oracle.com/connect/post/autonomous-indexing, 2019. 3.3, 6.1

[144] Oracle. Autonomous Database. https://www.oracle.com/autonomous-database/, 2022. 6.1

[145] Oracle. Oracle ExaData. https://www.oracle.com/engineered-systems/exadata/, 2022. 6.1

[146] Oracle. Oracle HeatWave. https://www.oracle.com/mysql/heatwave/, 2022. 6.1

[147] Ippokratis Pandis. The evolution of Amazon Redshift. *Proc. VLDB Endow.*, 14(12):3162–3163, 2021. 6.1

[148] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, and Youngmin Kim. Data Synthesis based on Generative Adversarial Networks. *Proc.*

VLDB Endow., 11(10):1071–1083, June 2018. ISSN 2150-8097. doi: 10.14778/3231751.3231757. URL https://doi.org/10.14778/3231751.3231757. 4.3.2

[149] Andrew Pavlo, Evan P. C. Jones, and Stanley B. Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *Proc. VLDB Endow.*, 5(2): 85–96, 2011. 3.1.1, 6.1

[150] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 61–72, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312479. doi: 10.1145/2213836.2213844. URL https://doi.org/10.1145/2213836.2213844. 2.2

[151] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-Driving Database Management Systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017. 2, 2.1, 4.1, 5.1.3, 6.1

[152] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Engineering Bulletin*, pages 32–46, June 2019. 3.1.3

[153] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. *Proc. VLDB Endow.*, 14(12):3211–3221, July 2021. ISSN 2150-8097. doi: 10.14778/3476311.3476411. URL https://doi.org/10.14778/3476311.3476411. 2.1, 2.1, 3.3, 4.1, 4.1.1, 5.1.3, 6.1

[154] Andy Pavlo. Why Machine Learning for Automatically Optimizing Databases Doesn't Work, 2023. URL https://db.cs.cmu.edu/files/slides/2023/why-ml-doesnt-work-jotb2023.pdf. 7

[155] Percona LLC. pt-upgrade — Percona Toolkit Documentation. URL https://docs.percona.com/percona-toolkit/pt-upgrade.html. 6.3

[156] Antonin Raffin. RL Baselines3 Zoo. https://github.com/DLR-RM/rl-baselines3-zoo, 2022. 3.2

[157] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *J. Mach. Learn. Res.*, 22:268:1–268:8, 2021. 3.2, 3.2.4

[158] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating Physical Database Design in a Parallel Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 558–569, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134975. doi: 10.1145/564691.564757. URL https://doi.org/10.1145/564691.564757. 1, 2.2

[159] Christopher Ré. Is Data Management the Beating Heart of AI Systems? In *SIGMOD '22*, page 3. ACM, 2022. 3

[160] Nilabja Roy, Abhishek Dubey, and Aniruddha S. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *CLOUD 2011*, pages 500–507. IEEE Computer Society, 2011. 6.1

[161] Anupam Sanghi, Raghav Sood, Dharmendra Singh, Jayant R. Haritsa, and Srikanta Tirthapura. HYDRA: A Dynamic Big Data Regenerator. *Proc. VLDB Endow.*, 11(12):1974–1977, 2018. 3.2.1, 6.1

[162] Anupam Sanghi, Shadab Ahmed, and Jayant R. Haritsa. Projection-Compliant Database Generation. *Proc. VLDB Endow.*, 15(5):998–1010, 2022. 3.2.2

[163] Scale AI, Inc. Scale Synthetic. https://scale.com/synthetic, 2022. 6.1

[164] Donald Shenaj, Ondrej Bohdal, Mete Ozay, Pietro Zanuttigh, and Umberto Michieli. LoRA.rar: Learning to Merge LoRAs via Hypernetworks for Subject-Style Conditioned Image Generation. *arXiv preprint arXiv:2412.05148*, 2024. 5.3.1

[165] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. *Proceedings of Machine Learning and Systems*, 6: 296–311, 2024. 5.3.1

[166] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 660–673, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526152. URL https://doi.org/10.1145/3514221.3526152. 3.1.1, 6.2

[167] Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning. *Proc. VLDB Endow.*, 15(10):2019–2031, June 2022. ISSN 2150-8097. doi: 10.14778/3547305.3547309. URL https://doi.org/10.14778/3547305.3547309. 6.2

[168] Y. C. Tay. Data Generation for Application-Specific Benchmarking. *Proc. VLDB Endow.*, 4(12):1470–1473, 2011. 3.2.1, 6.1

[169] Justin K. Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S. Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, Niall L. Williams, Yashas Lokesh, and Praveen Ravi. PettingZoo: Gym for Multi-Agent Reinforcement Learning. In *NeurIPS 2021*, pages 15032–15043, 2021. 3.2

[170] The Synthetic Data Vault. SDV – The Synthetic Data Vault. https://sdv.dev/, 2022. 6.1

[171] The Transaction Processing Council. TPC-C Benchmark (Revision 5.11), February 2010. URL https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf. 3.1.2

[172] The Transaction Processing Council. TPC-DS Benchmark (Revision 3.2.0), June 2021. URL https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf. 4.6.1, 5.5.1

[173] The Transaction Processing Council. TPC-H Benchmark (Revision 3.0.1), April 2022. URL https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf. 4.2, 4.6.1, 5.1.1, 5.5.1

[174] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A Physics Engine for Model-based Control. In *IROS 2012*, pages 5026–5033. IEEE, 2012. 3

[175] Tonic. Tonic.ai – The fake data company. https://www.tonic.ai/, 2022. 6.1

[176] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE 2000*, pages 101–110. IEEE Computer Society, 2000. 3.1.3, 6.1

[177] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3064029. URL https://doi.org/10.1145/3035918.3064029. 1, 2, 2.2, 3.1.1, 3.2.3, 4.1.1, 4.1.2, 5.1.3

[178] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *Advances in neural information processing systems*, 30, 2017. 3, 5.4

[179] Francesco Ventura, Zoi Kaoudi, Jorge Arnulfo Quiané-Ruiz, and Volker Markl. Expand your Training Limits! Generating Training Data for ML-based Data Management. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1865–1878, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457286. URL https://doi.org/10.1145/3448016.3457286. 3.1.1, 4.1.2, 6.1, 6.1, 6.2

[180] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. UDO: Universal Database Optimization using Reinforcement Learning. *Proc. VLDB Endow.*, 14(13):3402–3414, September 2021. ISSN 2150-8097. doi: 10.14778/3484224.3484236. URL https://doi.org/10.14778/3484224.3484236. 1, 2.2, 3.1.1, 3.2.2, 3.2.4, 3.4, 3.4, 4.1.1, 4.1.2, 6.1, 6.1

[181] Junxiong Wang, Debabrota Basu, and Immanuel Trummer. Procrastinated Tree Search: Black-Box Optimization with Delayed, Noisy, and Multi-Fidelity Feedback. In *IAAI 2022*, pages 10381–10390. AAAI Press, 2022. 6.1

[182] Kaiwen Wang, Junxiong Wang, Yueying Li, Nathan Kallus, Immanuel Trummer, and Wen Sun. Joingym: An efficient query optimization environment for reinforcement learning, 2023. 4

[183] Xiaoying Wang, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. Wii: Dynamic Budget Reallocation In Index Tuning. *Proc. ACM Manag. Data*, 2(3), may 2024.

doi: 10.1145/3654985. URL https://doi.org/10.1145/3654985. 6.2

[184] Robin Van De Water, Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. DataFarm: Farm Your ML-based Query Optimizer's Food! - Human-Guided Training Data Generation -. In *CIDR 2022*. www.cidrdb.org, 2022. 6.1, 6.1

[185] Johannes Wehrstein, Carsten Binnig, Fatma Özcan, Shobha Vasudevan, Yu Gan, and Yawen Wang. Towards Foundation Database Models. In *CIDR 2025, Conference on Innovative Data Systems Research*, 2025. 1, 3, 5.2.1, 6.3

[186] B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3):419–420, 1962. doi: 10.1080/00401706.1962.10490022. 4.3.2

[187] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. Eraser: Eliminating Performance Regression on Learned Query Optimizer. *Proc. VLDB Endow.*, 17(5):926–938, January 2024. ISSN 2150-8097. doi: 10.14778/3641204.3641205. URL https://doi.org/10.14778/3641204.3641205. 6.3

[188] Kelvin Wong, Qiang Zhang, Ming Liang, Bin Yang, Renjie Liao, Abbas Sadat, and Raquel Urtasun. Testing the Safety of Self-driving Vehicles by Simulating Perception and Prediction. volume 12371 of *Lecture Notes in Computer Science*, pages 312–329. Springer, 2020. 3

[189] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *29th IEEE International Conference on Data Engineering*, ICDE 2013, pages 1081–1092. IEEE Computer Society, 2013. doi: 10.1109/ICDE.2013.6544899. 6.2

[190] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. Budget-aware Index Tuning with Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1528–1541, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526128. URL https://doi.org/10.1145/3514221.3526128. 6.2

[191] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.*, 10(7): 781–792, March 2017. 3.1.2

[192] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. A Unified Transferable Model for ML-Enhanced DBMS. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL https://www.cidrdb.org/cidr2022/papers/p6-wu.pdf. 4.1.2, 5.2.1

[193] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Murali Narayanaswamy, and Tim Kraska. Stage: Query Execution Time Prediction in Amazon Redshift. 2024. URL https://www.amazon.science/publications/stage-query-execution-time-prediction-in-amazon-redshift. 4.6.2, 4.6.4, 5.6

[194] Prateek Yadav, Derek Tam, Leshem Choshen, Colin Raffel, and Mohit Bansal. TIES-merging: Resolving interference when merging models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=xtaX3WyCj1. 5.3.2

[195] Anatoly Yakovlev, Hesam Fathi Moghadam, Ali Moharrer, Jingxiao Cai, Nikan Chavoshi, Venkatanathan Varadarajan, Sandeep R. Agrawal, Tomas Karnagel, Sam Idicula, Sanjay Jinturkar, and Nipun Agarwal. Oracle AutoML: A Fast and Predictive AutoML Pipeline. *Proc. VLDB Endow.*, 13(12):3166–3180, 2020. 3

[196] Jingyi Yang, Peizhi Wu, Gao Cong, Tieying Zhang, and Xiao He. SAM: Database Generation from Query Workloads with Supervised Autoregressive Models. In *SIGMOD '22*, pages 1542–1555. ACM, 2022. 3.2.1, 6.1

[197] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.*, 14 (1):61–73, September 2020. ISSN 2150-8097. doi: 10.14778/3421424.3421432. URL https://doi.org/10.14778/3421424.3421432. 4, 4.6.2

[198] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 931–944, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3517885. URL https://doi.org/10.1145/3514221.3517885. 4.1.2

[199] Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. Low Rank Learning for Offline Query Optimization. *Proc. ACM Manag. Data*, 3(3), June 2025. doi: 10.1145/3725412. URL https://doi.org/10.1145/3725412. 1, 2.2, 6.3

[200] Dong Young Yoon, Ning Niu, and Barzan Mozafari. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *SIGMOD Conference 2016*, pages 1599–1614. ACM, 2016. 6.1

[201] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernández Cordero. Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo. *CoRR*, abs/1608.05742, 2016. 3, 3.2

[202] J. W. Zhang and Y. C. Tay. Dscaler: Synthetically Scaling A Given Relational Database. *Proc. VLDB Endow.*, 9(14):1671–1682, October 2016. ISSN 2150-8097. doi: 10.14778/3007328.3007333. URL https://doi.org/10.14778/3007328.3007333. 3.2.1, 6.1

[203] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. Deploying a Steered Query Optimizer in Production at Microsoft. In *2022 International Conference on Management of Data*, pages 2299–2311, July 2022. URL https://www.microsoft.com/en-us/research/publication/deploying-a-steered-query-optimizer-in-production-at-microsoft/. 6.3

[204] William Zhang, Wan Shen Lim, Matthew Butrovich, and Andrew Pavlo. The Holon Ap-

proach 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, 2024. URL https://www.vldb.org/pvldb/vol17/p3373-zhang.pdf. 1, 2.2, 2.2, 2.2, 2.3, 4.1.1, 5.1.3, 6.3

[205] Xinyi Zhang, Zhuo Chang, Hong Wu, Yang Li, Jia Chen, Jian Tan, Feifei Li, and Bin Cui. A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning. *Proc. ACM Manag. Data*, 1(2), June 2023. doi: 10.1145/3589331. URL https://doi.org/10.1145/3589331. 1.2, 2.2

[206] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. An Efficient Transfer Learning Based Configuration Adviser for Database Tuning. *Proc. VLDB Endow.*, 17(3):539–552, November 2023. ISSN 2150-8097. doi: 10.14778/3632093.3632114. URL https://doi.org/10.14778/3632093.3632114. 4, 4.1

[207] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 116–131, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483577. URL https://doi.org/10.1145/3477132.3483577. 5.1.1

[208] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proceedings of the VLDB Endowment*, 15(8):1658–1670, 2022. 3, 5.4, 5.5.1, 5.5.1

[209] Yue Zhao, Zhaodonghui Li, and Gao Cong. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *Proc. VLDB Endow.*, 17(4):823–835, March 2024. ISSN 2150-8097. doi: 10.14778/3636218.3636235. URL https://doi.org/10.14778/3636218.3636235. 4, 4.6.2

[210] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. Query Performance Prediction for Concurrent Queries Using Graph Embedding. *Proc. VLDB Endow.*, 13(9):1416–1428, May 2020. doi: 10.14778/3397230.3397238. 2, 5.1.3

[211] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. A Learned Query Rewrite System Using Monte Carlo Tree Search. *Proc. VLDB Endow.*, 15(1):46–58, September 2021. ISSN 2150-8097. doi: 10.14778/3485450.3485456. 3.1.1, 4.1.2

[212] 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, February 2023. ISSN 2150-8097. doi: 10.14778/3583140.3583160. URL https://doi.org/10.14778/3583140.3583160. 6.3

[213] Yiwen Zhu, Yuanyuan Tian, Joyce Cahoon, Subru Krishnan, Ankita Agarwal, Rana Alotaibi, Jesús Camacho-Rodriguez, Bibin Chundatt, Andrew Chung, Niharika Dutta, Andrew Fogarty, Anja Gruenheid, Brandon Haynes, Matteo Interlandi, Minu Iyer, Nick Jurgens, Sumeet Khushalani, Brian Kroth, Manoj Kumar, Jyoti Leeka, Sergiy Matusevych, Minni Mittal, Andreas Mueller, Kartheek Muthyala, Harsha Nagulapalli, Yoonjae Park, Hiren Patel, Anna Pavlenko, Olga Poppe, Santhosh Ravindran, Karla Saur,

Rathijit Sen, Steve Suh, Arijit Tarafdar, Kunal Waghray, Demin Wang, Carlo Curino, and Raghu Ramakrishnan. Towards Building Autonomous Data Services on Azure. In *Companion of the 2023 International Conference on Management of Data*, SIGMOD '23, page 217–224, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450395076. doi: 10.1145/3555041.3589674. URL https://doi.org/10.1145/3555041.3589674. 4.1.1