On Holistic Database Optimization via Leveraging Similarity Across Actions, Workloads, Configurations, and Scenarios

William Zhang

CMU-CS-26-100

February 2026

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

Thesis Committee:
Andrew Pavlo, Chair
Jignesh Patel
Vincent Conitzer
Immanuel Trummer (Cornell)

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

This research was sponsored by the National Science Foundation under award number 2404373, SingleStore, Google, PDL Consortium, and a MongoDB Fellowship. 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, holistic tuning, transfer learning, agentic database anomaly mitigation, machine learning for systems

To my partner, Jessica

Abstract

Modern database management systems (DBMSs) have evolved to support increasingly sophisticated data-intensive applications, at the cost of substantial complexity to configure them for two reasons. First, DBMSs expose a vast configuration space with trillions of possibilities that encompass system knobs, physical design (e.g., indexes), and query options, among others. Second, these applications are constantly evolving with changes in data access patterns, query types, load intensities, hardware, and data distributions that necessitate continuous re-optimization.

To address these challenges, decades of autonomous DBMS optimization research have produced specialized tuning tools to assist human operators. Deploying these tools involves a complex multi-step workflow where an operator (1) observes the DBMS's behavior, (2) selects tools based on the objectives and their expertise, (3) configures them with an isolated environment, (4) orchestrates their execution to obtain recommendations, and (5) reviews those recommendations before deployment. This cumbersome process results in suboptimal configurations and slow adaptation to evolving applications' workloads due to isolated specialized tools, inefficient reuse of prior tuning knowledge, and the fallible human factor.

In this dissertation, we present techniques for addressing those limitations with *similarity* to enable holistic database optimization. First, we present a holistic tuning tool that optimizes multiple DBMS aspects simultaneously by using action similarity to organize actions into neighborhoods conducive to exploration. We then present a framework that assists tuners in adapting to environment changes by leveraging workload and configuration similarity to re-mix historical knowledge. Lastly, we present a system that transforms the human-centric tuning workflow into an agentic process by using scenario similarity to link the deployment context with semantic tool interfaces to optimize the deployment.

The techniques and associated *similarity* definitions presented in this dissertation enable agentic holistic DBMS optimization over a deployment's lifetime, improving the deployment's performance and reducing time taken to adapt to changes in upstream user applications.

Acknowledgments

I am very fortunate to have had the opportunity to be advised by Andrew (Andy) Pavlo throughout my PhD journey. I am grateful for Andy's initial push and advice to pursue database research back when I was an undergraduate. His enthusiasm, passion, and dedication to the field have been a source of inspiration. He has taught me the crux of being a researcher: identifying interesting problems, relentlessly pursuing them, navigating technical challenges, and framing discoveries for the larger scientific community. I am deeply grateful for the experience of working with and learning from Andy, from the early in-the-weeds discussions of building systems through the ups and downs of database research.

I also want to thank Jignesh Patel, Vincent Conitzer, and Immanuel Trummer for serving on my thesis committee. I could not have asked for a more supportive committee. Jignesh provided numerous insights that informed my work's operational assumptions and made them more realistic. Vincent's numerous insightful and thought-provoking questions about broader applicability and the rapidly evolving field of AI were of tremendous help in shaping the research direction. Immanuel's deep knowledge in database tuning provided invaluable feedback and grounding on the technical aspects of my work.

I overlapped with many talented researchers and PhD students in the database group during my time at CMU. Lin Ma mentored and guided me when I first started doing research as an undergraduate, for which I am deeply grateful. Matthew Butrovich provided invaluable advice and feedback early on in my PhD journey. Wan Shen Lim has been with me since our undergraduate days after a run-in while also taking business law; beyond being an amazing collaborator, he has been a great friend and I've enjoyed our chats about world developments, numerous food runs, and more over the years. Sam Arch, Christos Laspias, and Hyoungjoo Kim are PhD students who started after me and with whom I have had the joy of numerous engaging discussions and lunches.

I would also like to extend my thanks to my mentors and amazing researchers during my internship at Gray Systems Lab: Yiwen Zhu and Subru Krishnan. Although they were not directly involved in my thesis's work, their astute questions around database tuning and the research skills they taught me have significantly helped and shaped my work on adapting tuner knowledge and agentic DBMS anomaly response.

I also want to thank the numerous B.Sc., M.Sc., and visiting scholars that I interacted with in the database group from the early dev lunches during terrier/NoisePage days to recent proxy-related research: Tianyu Li, Erik Sargent, Katrina Jiao, Vivian Huang, Wuwen Wang, Abigale Kim, Deepayan Patra, Chi Zhang, Ruiqi Wang, Bohan Zhang, Lichen Jin, Patrick Wang, Xiaohui Wang, and numerous others. Tianyu first introduced me to the database group and Andy.

Patrick built out the first prototype of the database gym that incorporates Proto-X and provided insightful feedback on extending it into live environments (Chapter 3). Xiaohui provided an initial design and early validation on an agentic proxy for live workload manipulation that helped shape the framing and form factor of the agentic anomaly response framework (Chapter 5).

I have been fortunate to have also been a member of the Parallel Data Lab (PDL). PDL has taught me to be a better speaker and presenter through sharing my work and research ideas with the broader research community and consortium partners. In particular, I want to thank Valerie Choung, Hojin Park, and Greg Ganger. I would also like to thank all the administrative staff who have worked behind the scenes to make all of this work possible: Karen Lindenfelser, Joan Digney, Chad Dougherty, Catherine Copetas, and Matthew Stewart.

Lastly, I want to thank my family. My parents have provided unconditional support and understanding since I started pursuing research. I am eternally grateful to my partner Jessica for being by my side as we navigated graduate school and for the enormous joy, excitement, and wisdom she has brought to my life. Through the rollercoaster of ups and downs that is pursuing a PhD, she has provided me with unconditional support, understanding, and love. This dissertation would not have been possible without her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Six decades after the first database management system (DBMS) was developed, DBMSs now form the backbone of modern data-intensive applications. To support their workloads' growing demands, DBMS vendors continue to release new features and provide more opportunities for end users to optimize their systems for specific deployments through database tuning. Database tuning involves reasoning over the DBMS's configurable options that include system knobs [109] (e.g., parallelism, optimizer), physical design (e.g., indexes, partitioning), and query tuning (e.g., query rewriting, hints). Furthermore, these options exhibit complex, subtle interactions, creating trillions of possibilities [68] that make it difficult to discover performant configurations.

To manage this complexity, human operators (e.g., DBAs) employ a multi-step workflow to discover performant configurations [7]. At the start of the workflow, the operator first observes the DBMS's behavior and telemetry, along with any alerts or triggers from a monitoring system (e.g., Amazon CloudWatch). Based on their observations and domain expertise, the operator then contextualizes and interprets the observation to identify issues and potential solutions. They then deploy *tuning pipelines* to generate recommendations, validate them, and apply them to the deployment. These tuning pipelines often comprise one or more specialized tools that target specific aspects of the DBMS. The operator first instantiates a sandbox environment containing a snapshot or isolated clone of the deployment and a workload trace to optimize. They then configure each tool in the pipeline with the appropriate inputs (e.g., tuning objectives, constraints, tunable aspects) and run it in the sandbox environment. These tools range from simple rule-based tools to more sophisticated machine learning (ML)-based tools that can optimize system knobs [60, 109], indexes [11, 23], query tuning [77], and across multiple aspects [111, 134]. Within a single pipeline, the operator manually orchestrates the tools by determining the correct sequence for invoking them and how to combine their respective tool recommendations together.

Although this human-centric workflow has demonstrated potential to improve deployment performance, it cannot achieve a fully autonomous self-driving DBMS [86]. Given user objectives and constraints (e.g., minimize p99 latency within specific service-level agreement (SLA) requirements), a self-driving DBMS holistically optimizes itself over its lifetime. This lifetime holistic optimization involves (1) finding performant configurations for a point-in-time snapshot of the deployment and workload, (2) rapidly adapting to changes caused by upstream applications, and (3) intervening in response to any anomalies. We discuss each of these aspects in more detail, along with the limitations of the existing human-centric workflow and existing tuning pipelines.

1.1 Holistic Optimization for a Point-in-Time Snapshot

Given a point-in-time snapshot of the deployment and workload, a self-driving DBMS must find performant configurations that maximize the user's objectives subject to their constraints. This optimization process is challenging due to the large search space and complex interactions between the configuration spaces. For instance, a deployment may expose hundreds of system knob settings, hundreds of candidate indexes, and thousands of possible query hints to apply. This combined space is exponentially large, making it computationally difficult to reason over.

No existing tuning pipeline or individual tool holistically optimizes the entire configuration space due to this complexity. Each tool focuses on a local aspect of the DBMS (e.g., only knobs) and makes simplifying assumptions or requires operator guidance to make the problem tractable. For instance, some tools use heuristics to prune the search space [134] while others require a human operator to curate their actions [111]. These tools then defer multi-faceted optimization (e.g., jointly optimizing knobs and indexes) to the human operator who uses their judgement to invoke the correct tools in the proper sequence to obtain a performant configuration. Not only is this process time-consuming and cumbersome, but it is also fundamentally flawed. As these tools were designed to operate in isolation, they are unaware of other tuners and cannot compose together to optimize a given deployment holistically. For instance, a knob tuner may choose to turn off index scans to improve the DBMS's performance, at the cost of preventing the index tuner from discovering any beneficial indexes.

To remedy this coordination challenge, we require a holistic tuner that can jointly optimize the entire configuration space simultaneously by reasoning about the interactions between the various configuration spaces and traversing the high dimensional combined space. To accomplish this, the tuner needs to intelligently reason about the relationships among different actions to effectively prune and explore promising regions for performant configurations.

1.2 Rapid Adaptation to Application Changes

Even though a DBMS may be optimized for a given point-in-time snapshot, the environment often is not static. Due to changing user patterns and business requirements, applications' workloads exhibit changes in data access patterns, query types, load intensities, and data distributions [110]. Whenever these changes occur or are detected by an external monitoring system due to an anomaly, an operator must manually re-optimize the DBMS by deploying a tuning pipeline to ensure the DBMS remains in an optimal configuration.

Despite the prevalence of these changes and the substantial knowledge available (e.g., past pipelines, expertise), existing tuning tools are unable to efficiently adapt to applications' changes due to their design limitations. In the case of heuristics-based and cost-based tuners, their algorithms do not readily support historical knowledge because they are fixed algorithms [11, 23, 60]. Even though some ML algorithms theoretically support incorporating prior experience through pre-training [66] or fine-tuning [47], both are expensive in terms of recalibrating historical observations for the current environment and consider only a workload-level granularity. Fundamentally, a framework is needed to reason about individual query semantics and different configurations to accelerate deployment re-optimization for existing and future tools.

1.3 Rapid Intervention for Anomalies

Regardless of how optimized a deployment is, anomalies occur in production deployments [46, 144]. These anomalies are typically detected from a monitoring system (e.g., Amazon CloudWatch) that notifies an operator. Existing "copilot" systems are available that focus on assisting an operator in diagnosing and identifying the root cause of a given anomaly [24, 84, 144].

However, these diagnostic systems are unable to actively intervene. They are unable to *act* and reflect on their actions to improve the deployment. Instead, they rely on a human operator to evaluate their analysis and apply the appropriate corrective actions. There is a lack of standardized interfaces for optimization tools that are conducive to orchestration by a reasoning agent (e.g., a tool-calling large language model). Optimization tools rely on the operator to intuit their semantics and behavior rather than providing explicit guidance. As such, we require a system that can contextualize the deployment state and orchestrate a toolbox of optimization tools to actively intervene and mitigate anomalies.

1.4 Summary of Contributions

This dissertation addresses the challenges of holistically optimizing a DBMS deployment over its lifetime through distinct notions of *similarity*. It explores multiple similarity definitions that facilitate efficient reasoning over (1) actions for tuning a deployment's point-in-time snapshot, (2) workloads and configurations to adapt a deployment to changes in user applications, and (3) scenarios for intervening in ongoing deployment events or anomalies.

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

Thesis Statement: *Identifying and leveraging similarities among tuning actions, workloads, configurations, and operational scenarios enables holistic optimization of a database management system throughout its lifetime that improves performance and reduces adaptation time.*

This dissertation makes the following contributions:

Action Similarity for Holistic Tuner (Chapter 3): We present a holistic tuner that optimizes multiple DBMS aspects simultaneously for a point-in-time snapshot of a given deployment and workload. The holistic tuner leverages a notion of action similarity to structure an extremely high-dimensional action space into related neighborhoods conducive to exploration.

Workloads-Configurations Similarity for Adaptation (Chapter 4): We introduce a framework that enables tuners to adapt an existing deployment to changes in applications' semantics and workloads. The framework leverages workloads-configurations similarity to identify and compose relevant historical contexts into distilled findings that are injected into the assisted tuner to improve the final configuration's performance and reduce time to convergence.

Scenario Similarity for Interventions (Chapter 5): We present a system that transforms the human-centric diagnostics-based workflow into an agentic process capable of directly intervening in response to anomalies encountered over the deployment's lifetime. The system leverages scenario similarity to link the deployment context with a toolbox exposed through semantic tool interfaces to repair the deployment.

We provide more detailed background into DBMS optimization, workload adaptation, and encountered deployment anomalies in Chapter 2. We provide detailed related work in Chapter 6 and possible areas of future work in Chapter 7. We conclude our research findings in Chapter 8.

Chapter 2

Background

In this chapter, we provide background on automated DBMS optimization. We primarily approach this discussion within the broader context of *self-driving* [86] DBMSs that are designed to manage themselves autonomously without human intervention (i.e., holistic optimization across time). We then discuss automated tuning tools and their challenges from both efficacy and deployment perspectives, followed by the ability of these existing tools to adapt to environment changes. Lastly, we conclude with a discussion of existing agentic diagnostic systems and their shortcomings with respect to actively intervening to mitigate anomalies.

2.1 Self-Driving DBMSs

A self-driving [86, 87] DBMS aims for completely autonomous operation that is proactive and efficient. To achieve this, a self-driving DBMS relies on three core architectural components: (1) a forecaster, (2) behavior models, and (3) a planner.

Forecaster: This component is responsible for predicting workloads that the DBMS might have to execute in the future at various horizons (e.g., next hour, next day). Forecasting relies on various techniques such as clustering [71] and Long Short-Term Memory networks [45] to predict workload characteristics such as query templates, arrival patterns, and query parameters. A self-driving DBMS relies on its forecaster to guide the planning process, preparing it for upcoming scenarios. For instance, if the forecaster predicts a workload spike in the next hour, the DBMS can proactively scale up to prepare.

Behavior Models: The simplest method for a self-driving DBMS to evaluate an action's impact is to change the DBMS's configuration and run the workload. However, this approach is infeasible due to computational overhead. As such, a self-driving DBMS relies on its behavior models [73] to predict the costs and benefits of all possible actions. Developers must carefully co-design the behavior modeling framework and the self-driving DBMS architecture to model all actions' effects accurately. For instance, if the DBMS were to change a configuration parameter (e.g., background task resources), the behavior model must be able to predict how that change may impact both immediate and future queries.

Planner: Even though existing DBMSs may suggest configuration changes, they rely on a human operator to evaluate and apply them. The self-driving DBMS's planner uses the forecasted workload and its behavior models to identify promising configuration changes (e.g., knobs to set, indexes to build). The planner then uses authorized time windows (e.g., maintenance window, under-utilized instance) to deploy those changes autonomously.

The tightly coupled forecaster, behavior modeling, and planner components form the bedrock of a self-driving DBMS design that facilitates holistic DBMS optimization over its lifetime. However, existing DBMSs often lack many of these autonomous capabilities. Instead, they resort to a human operator augmented with tuning tools to approximate the planner. We center our discussion on these tuning tools next.

2.2 Automated Database Tuning

As Section 2.1 describes, existing DBMSs rely on a human operator augmented by tuning tools to approximate the planner. These bespoke tools specialize in optimizing specific facets of the DBMS. Before deploying a tool, the operator creates an isolated environment containing a snapshot of the deployment and a workload trace to optimize. This isolated environment allows the tool to freely explore configurations, restart the DBMS, and take potentially destructive actions (e.g., dropping indexes) without affecting the production environment [72]. Depending on high-availability constraints, the operator may temporarily detach (i.e., fork off) a read replica to serve as the sandbox [68, 72], while also mirroring production traffic to observe its behavior. If the optimization proves fruitful, the control plane can directly promote the now-optimized read replica to the new primary or otherwise merge it back into the production environment.

2.2.1 DBMS Optimization Tools

To simplify the overhead for an operator using a tool, existing tools are designed from the outside in. They function by observing the deployment's characteristics (e.g., schema, query runtimes, metrics) and then use their internal algorithm or models to generate recommendations. We describe a few common aspects targeted by different DBMS optimization tools.

Knobs: Modern DBMSs expose hundreds of system knobs that alter the DBMS's behavior [109]. For instance, these knobs can alter the query optimizer's behavior (e.g., join method), the maximum degree of parallelism for scans, and resource settings (e.g., buffer pool size), among others. Knob tuners aim to reason over the interactions between different knobs to provide a performant configuration. These knob tuners range from simple heuristic-based rules [60] to more sophisticated machine learning (ML)-based tools (e.g., Bayesian optimization [109], reinforcement learning [130], LLM-based [47]).

Physical Design: Optimizing the physical design is critical for improving query performance. Physical design includes index selection [11, 23] and materialized views [126], among others. Physical design tuners rely on a workload trace to identify beneficial structures. For instance,

an index tuner may extract query predicates from the workload trace to constrain the set of interesting columns to index [23, 29]. Often, physical design tuners rely on the DBMS to support a "what-if" [22] mechanism to evaluate the impact of different changes without actually applying them, due to overhead. However, as each index candidate has a multitude of considerations, such as index type (e.g., B+Tree, Hash), index columns (e.g., single-column, multi-column), predicate constraints (e.g., partial indexes), and include columns (e.g., covering indexes), existing tuners are unable to reason about all of these considerations. They instead aggressively prune the search space using heuristics or operator guidance to make it more tractable.

Queries: The DBMS's optimizer determines how a query should be executed (e.g., its query plan). Because finding the optimal query plan is NP-hard, the optimizer may fail to find one for a given query. Externally-oriented query tuners aim to improve on the existing query optimizer's behavior by guiding the optimizer. In its simplest form, these tuners may rewrite the query to make planning easier [67]. Other tuners recommend query hints that the optimizer understands to control its exploration strategies (e.g., consider only hash joins) [12, 77]. Internally-oriented query tuners aim to augment the core optimizer to generate more performant plans [76] directly at the cost of higher initial training and calibration. For federated systems or DBMSs that support multiple query engines, middleware (e.g., proxy) may selectively route queries to the most performant execution engine [125].

2.2.2 Sequential Tuning

Existing optimization tools often target only a specific facet of the DBMS (e.g., only knobs or only indexes) in isolation for tractability, as considering all DBMS facets simultaneously yields trillions of possibilities that are computationally difficult to reason about and explore [68]. *Sequential tuning* [134] describes the practice of running multiple optimization tools in sequence and composing each tool's recommendations along the way to derive an optimal configuration. For example, a human operator or framework can first tune knobs, then indexes, and finally query hints. Sequential tuning frameworks are built on top of a pool of existing tools and focus on scheduling their execution (e.g., round-robin, future reward).

However, these frameworks are limited by coordination challenges across individual tools. Each tool runs in complete isolation and is unaware of when other tools will be run. As such, a tool may make a locally optimal decision that is not globally optimal considering the joint space. For instance, when tuning a new deployment from scratch that lacks the appropriate indexes, a knob tuner may turn off index scans to improve the workload's performance. However, this prevents an index tuner from discovering any beneficial indexes since the query optimizer will not consider them.

2.2.3 Tool Interfaces

Beyond the challenges of coordinating individual tools, existing tools lack a uniform interface that allows for automated orchestration. As each tool was initially designed for a human operator, they defer understanding its nuances to the operator. For instance, a knob tuner's efficacy depends on the number of samples [53] and the quality of prior observations [47]. However,

this information is not made readily available and instead depends on human expertise to operate effectively. Furthermore, because tool developers do not provide an "API"-like interface, operators are forced to understand each tool's individual inputs and outputs rather than allowing an automated reasoning process to select the best tool and inputs depending on the current environment and objectives.

2.3 Adaptive Automated Tuning

Beyond the aforementioned challenges around their efficacy, existing tools are also unable to adapt to environment changes. As tools optimize deployments, they acquire experience that includes the tool's reasoning, explored configurations, and observed DBMS behavior. *Adaptivity* captures a tool's ability to reuse prior experience when tuning new scenarios. These scenarios encompass changes in the workload (e.g., parameters, templates, volume), underlying data (e.g., `BULK INSERT`), hardware (e.g., instance upgrades), and schema (e.g., new column) [65, 110]. These environment changes are common in practice, with Redshift observing 50% of their production clusters have 50% of queries repeating exactly (i.e., same query template and parameters) daily [110]. We discuss three approaches to enable adaptive tuning, along with the respective limitations inherent to each approach within existing tuning frameworks.

Foundation Database Models: This approach proposes creating a foundation database model [115] that captures the database's behavior. By training the model on a large dataset of configurations, workloads, and observed telemetry, the model can reason about the database's behavior and generate recommendations that are effective for the current environment. However, this approach has extremely high sampling complexity and is prohibitively expensive [69]. Within a more limited scope, researchers have focused on generalizable cost models [43] for estimating a query plan's runtime without executing it. However, these cost models still require observing substantial query plan telemetry under a range of configurations to be effective.

Bootstrapping from History: Analogous to transfer learning, this approach leverages historical knowledge to accelerate the current tuning process. For most existing tuning tools that capture schema and workload elements to represent the DBMS's state [58, 134], they can only continue optimizing from the historical configuration if the target deployment's representation is closely aligned with the historical deployment's representation. However, this is often not the case as schema migrations occur or the workload evolves over time in unpredictable ways.

To work around this, techniques have proposed using prior tuning sessions' models as expert prior signals [134] about the optimization space, initializing the initial state and guiding future suggested configurations based on incremental nearest-neighbor matching of DBMS telemetry [109], and pre-training the neural networks for evaluating configuration efficacy [66]. However, pre-training remains difficult, as changes in the environment (e.g., queries, schemas) may invalidate previously observed objective values. For instance, a workload shift may introduce a new query that makes a prior configuration less desirable.

LLM-based Adaptation: Recent techniques have focused on leveraging the emergent capabilities of large language models (LLMs) to reason about workloads and configurations [38, 47]. These approaches generate prompts that include instructions for the tuning task and relevant information about the workload. The optionally fine-tuned LLM generates candidate configurations, which are then ranked. Although these techniques are promising, challenges remain regarding granularity. For instance, prompting the LLM on individual queries may allow it to generate more effective configurations, but their composition may result in query regressions. Furthermore, it has yet to be explored how to incorporate query-level historical tuning insights into the prompt. For instance, if a query was already tuned before, the tool could augment the prompt with past attempts (e.g., query hints, indexes) as additional context to assist the LLM in generating more effective configurations (i.e., RAG) [24, 64].

Although there is a range of approaches, existing tuning tools still treat adapting a previously tuned deployment as a tuning-from-scratch process. This is in part due to either inherent limitations in their algorithm (e.g., cost-based search [11], heuristics [60]) or the mismatch between historical observations and the current target environment. This leaves considerable room for improvement in both the efficacy of the final re-optimized configuration and the time it takes for tools to re-optimize the deployment.

2.4 Agentic Diagnostics Systems

In operations, monitoring systems (e.g., Amazon CloudWatch [9]) observe deployment and workload telemetry to identify anomalies [46]. Once these anomalies are detected, a corresponding alert is raised, which pages an on-call operator to investigate and mitigate them. To assist in this process, on-call operators often rely on automated diagnostic systems to provide insights into the anomaly [84]. At the simpler end, these systems are decision trees or causal models based on curated rules [124] to provide root cause analysis. Recent work has proposed more advanced agentic systems based on large language models (LLMs). These agentic systems can use diagnostic tools (e.g., get query plan, retrieve CPU usage) and knowledge base retrieval (e.g., documentation search) to provide further insights into the anomaly and propose possible mitigations. However, these agentic systems are unable to actively intervene. These systems do not include any optimization tools with verifiable outcomes. For instance, even though they may be able to identify a missing index, they are unable to deploy and verify the missing index's efficacy. In some scenarios, even though the workload may appear to exhibit higher p99 latency due to a missing index, the actual cause may be that the system is under heavy load.

Chapter 3

Action Similarity for Holistic Tuner

Existing ML approaches to automatically optimize a DBMS only target a single configuration space at a time (e.g., knobs, query hints, indexes). However, tuning individual configuration spaces is inadequate to achieve a fully autonomous (i.e., self-driving [87]) DBMS.

Simultaneously tuning multiple configuration spaces is challenging due to the combined space's complexity. Applying existing techniques directly to the combination of spaces (i.e., the holistic configuration space) suffers from the *curse of dimensionality* [134]. In a holistic space, the landscape of beneficial configurations becomes sparser. Due to this, these approaches degrade and spend too much time evaluating unpromising solutions. As a result, previous tuning methods work around this by sequentially tuning individual spaces with a pool of tuners. However, these approaches struggle to coordinate their tuners and get stuck in local optima.

In this chapter, we present the Proto-X tuning tool that holistically optimizes a DBMS's configuration across multiple configuration spaces. The key idea of Proto-X is to identify similarities between actions across multiple spaces. Proto-X then uses those similarities to structure the high-dimensional action space into related neighborhoods. Finally, Proto-X synthesizes "proto-actions" to actively explore the resulting space for beneficial configurations.

The rest of the chapter is organized as follows. We provide the key ideas behind action similarity in Sections 3.1.2 and 3.1.3, followed by Proto-X's architecture in Section 3.2, and an evaluation of Proto-X against state-of-the-art DBMS tuning tools on tuning PostgreSQL for analytical and transactional workloads in Section 3.5. By holistically optimizing across spaces that are more complex in terms of quantity and variety than prior tuning tools, Proto-X discovers configurations that improve PostgreSQL's performance by up to 53% over the next best approach.

3.1 Background: Sequential Tuning and Proto-Actions

A self-driving DBMS optimizes itself within user-defined constraints (e.g., resource limits) without human intervention [86, 87]. The DBMS first performs workload forecasting to predict future workloads (e.g., time series of SQL queries) [71]. It then builds *behavior models* [73, 75] to estimate the user's objective function (i.e., performance) for a given configuration. The DBMS's *tuning agent* then chooses *actions* to maximize the objective function on the future workload. For example, an action can build an index, change a table knob, or modify a query plan.

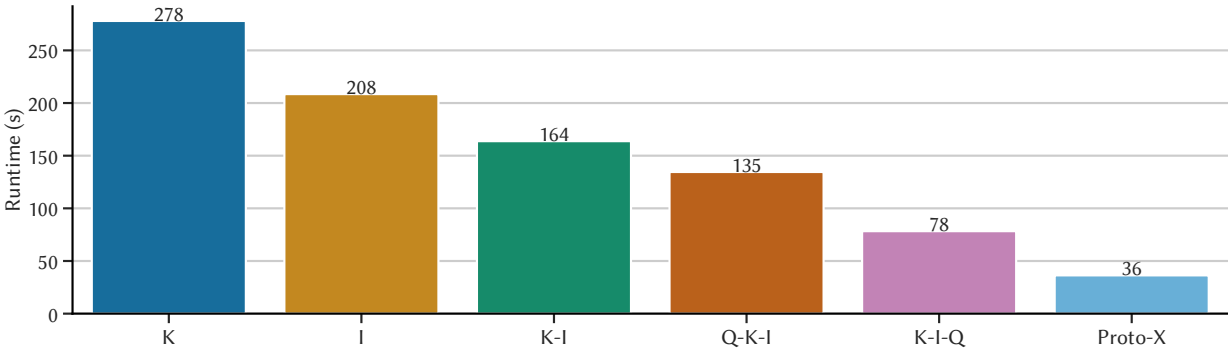


Figure 3.1: Motivating Example – JOB workload runtime when running a knob tuning agent (K), an index tuning agent (I), and a query knob tuning agent (Q) in isolation, sequentially, and our technique Proto-X that holistically optimizes knobs, indexes, and query knobs/hints for 30h.

Existing work tunes a specific category of DBMS options (i.e., a *configuration space*) in isolation. For instance, different tuners target system knob tuning [109, 130], query knob tuning [12, 77], and physical design [97]. As we now discuss, running these tuners one at a time is not optimal due to interactions between the spaces.

3.1.1 Sequential Tuning and Coordination

DBMS tuners [111, 134] that support multiple configuration spaces repeatedly invoke the same steps: (1) choose a single space to tune, (2) fix other spaces to their current configuration, and (3) tune the target space for a fixed amount of time. For example, UniTune [134] may tune knobs for an hour, then indexes, and then back to knobs.

As sequential tuning cannot consider actions across spaces and fails to find optimal solutions, deployments incur wasted resources from repeating workloads [117] or undesirable latency [12]. Figure 3.1 illustrates the runtime on the JOB workload [63] of the best configuration discovered by sequential methods and our holistic technique Proto-X in 30h. Proto-X finds the best configuration (36s), which executes 42s (53%) faster than that found by the next best.

We analyze JOB's Q26c to illustrate this problem in more detail. From an initial configuration, we alternate index and query tuners to obtain a search tree of <index set, query knobs> configurations along with Q26c's corresponding performance in Figure 3.2. Each tuner locally maximizes its component and prunes suboptimal actions (the red X). This process simplifies the search. However, the pruning prematurely eliminates paths to the global optimum <I2,H2>.

3.1.2 Holistic Optimization and Action Similarity

To our knowledge, no prior approach optimizes the entire configuration space simultaneously due to the resulting holistic space's complexity [68, 97, 134]. For instance, there are at least 2^{46} candidate indexes from TPC-DS [106]. This complexity further compounds when considering other spaces in conjunction, such as query knobs.

Although the number of unique actions in a space is large, many share properties such that it may not be necessary to consider each one individually. Two actions are *similar* if they target

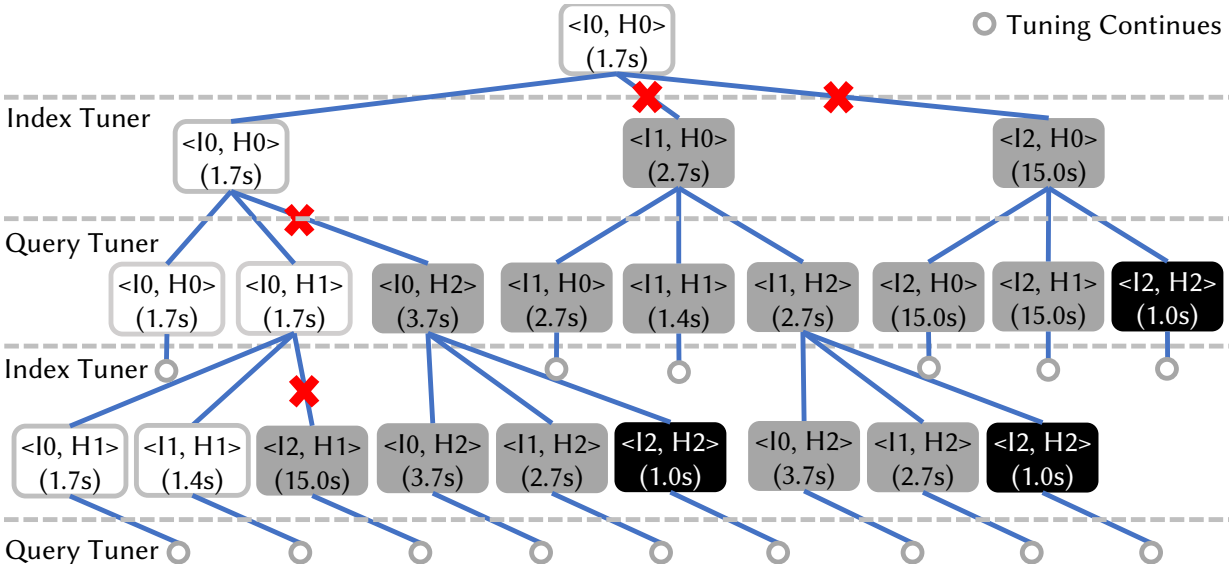


Figure 3.2: JOB Q26c Sequential Tuning Tree – Search tree explored by sequentially tuning indexes and query knobs. Each node represents a <index set, query knob> configuration with Q26c's corresponding performance. The tuners prune branches indicated in red (X). Sequential tuning finds a local optimum at <I1,H1> but will not find the global optimum in black <I2,H2>.

the same database objects and have roughly the same effect on the agent's objective function. Consider two candidate actions that add indexes to the TPC-H `LINEITEM` table [107]: (1) Cand-A builds an index on `(l_partkey)`, and (2) Cand-B builds an index on `(l_partkey,l_commitdate)`. Both actions have similar expected changes to the DBMS's performance and are structurally similar (i.e., the indexes target the same table and share the first key).

Such similarity allows a tuning agent to infer the performance of one action from a similar action. Rather than running the workload once each for Cand-A and Cand-B, the agent can estimate Cand-B's benefit by only evaluating the workload with Cand-A and vice versa. This similarity suggests that agents should not consider actions alone. Instead, an agent should consider an action and its neighborhood of similar actions together.

3.1.3 Proto-Actions and Neighborhoods

We combine actions that modify the DBMS's configuration into a *holon* composed of multiple fields [59, 98]. Each holon field is an independent action corresponding to a configuration space under tuning. We illustrate an example *holon* in Figure 3.3 ❶. `K` sets a system knob that sets the buffer pool size (`shared_buffers`); `Q-c` sets a query knob that disables nested loop joins on Q1 of the workload; `Q-e` sets a query hint that forces Q2 to scan `region` in parallel; `I` builds an index on `LINEITEM (l_partkey,l_suppkey)`.

An encoder (e.g., neural network) maps the holon to a point in a multidimensional continuous *latent space*. We shape the latent space by training the encoder to place holons of similar structure or expected performance nearby (i.e., close to each other) [74]. The latent space's dimensionality captures a trade-off between accuracy of the holon representation and ease of exploration.

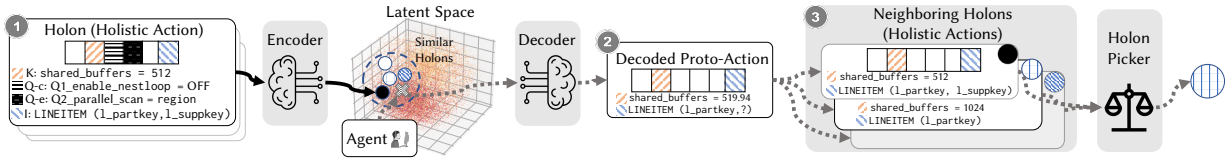


Figure 3.3: Proto-Actions and Neighborhoods – A holon passes through an encoder (e.g., neural network) to obtain a point (filled-in circle) in latent space where holons of similar performance or structure are nearby. A proto-action is a point (**X**) in latent space chosen by a tuning agent. Decoding the proto-action does not necessarily result in a valid holon. Instead, the agent searches the proto-action's neighborhood for a valid holon and selects the most promising one.

Unlike other tuners that directly suggest deployable actions, Proto-X's agent suggests a latent space point (i.e., a *proto-action*). A proto-action represents a neighborhood of holons with similar structure or performance. Directly decoding the proto-action may not yield a valid holon (e.g., ② the proto-action sets an integer knob `shared_buffers` to 519.9). Instead, the agent decodes and ③ searches the proto-action's neighborhood to obtain a candidate set, from which it selects the most promising holon to evaluate [34].

3.2 Architecture

We present the architecture of the Proto-X automated tuning framework in Figure 3.4. Like other offline tuning tools [73, 77, 109, 111, 130, 134], Proto-X assumes access to a representative or historical workload sample [29, 117], an isolated environment for tuning [68], and infrastructure (e.g., proxy) for applying query options, such as those exposed by PostgreSQL's *pg_hint_plan* [1]. Proto-X operates in two phases. First, it creates the latent space based on the DBMS schema, sample workload, and target configuration spaces. Then, in the second phase, Proto-X exploits the created latent space to tune the DBMS. We now describe each phase in more detail.

3.2.1 Phase I: Latent Space Creation

Proto-X begins by collecting each configuration space's metadata: (1) *system knobs*, (2) *table knobs*, (3) *indexes and per-index knobs*, (4) *query knobs*, and (5) *query hints*. A query knob alters the DBMS's behavior by changing a system knob for a specific query (e.g., hash join memory allocation). A query hint injects commands to control how the DBMS generates a plan (e.g., index scan for a table).

Proto-X obtains the tunable system knobs either from the user, through a DBMS-specific method, or from externally derived knob constraints [135]. For each system knob, Proto-X obtains the min/max values and whether the knob should be restricted to a limited value range [87] (e.g., quantization). Next, it connects to the DBMS to obtain the database schema to identify candidate table knobs. Proto-X defines its candidate index domain by extracting all referenced attributes from the predicates of the user's workload [97] and obtains tunable per-index knobs from the user's specification.

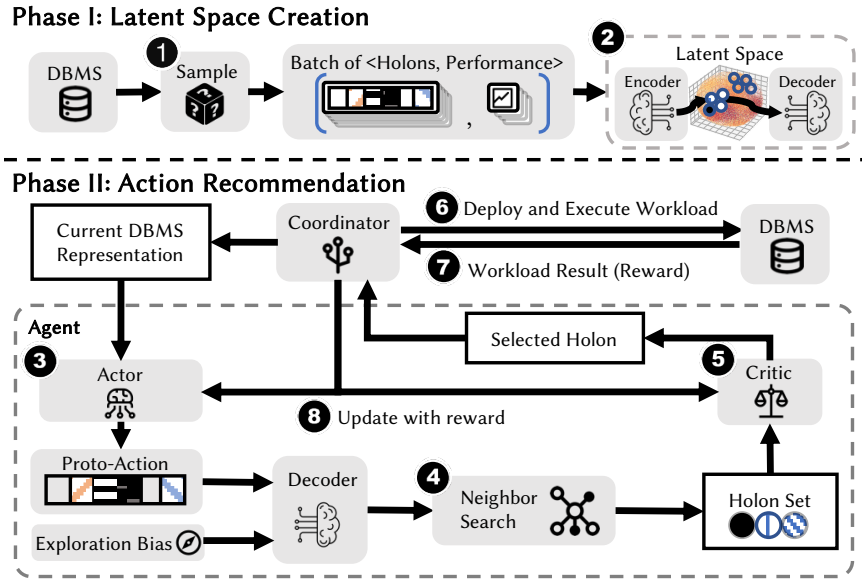


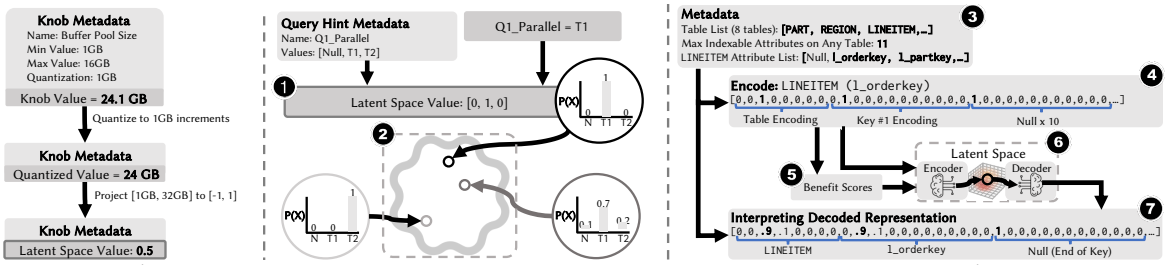
Figure 3.4: Architecture – An overview of the two phases of Proto-X. In Phase I, Proto-X creates the necessary latent space. In Phase II, Proto-X recommends holons to optimize the DBMS.

Lastly, Proto-X parses the workload to identify query knobs and hints. For each query, Proto-X identifies (1) query knobs based on a user-supplied list or by mining the system knobs for applicable knobs, (2) a query hint for each referenced table that forces a specific access method, and (3) a query hint specified once per-query that forces a parallel scan on a selected table. For example, TPC-H [107]'s Q14 accesses the `LINEITEM` and `PART` tables, and thus Proto-X generates the `Q14_lineitem_scan` and `Q14_part_scan` hints that control whether the DBMS should use an index scan for those tables. Proto-X also identifies a `Q14_Parallel` hint. Setting this hint forces the DBMS to scan the referenced table in parallel.

After obtaining metadata, in Figure 3.4, Proto-X ❶ samples a batch of holons and their estimated benefits for the target workload. ❷ Using the batch, Proto-X trains the latent space to place holons of similar benefit at nearby locations. For example, if an index `t (a,b,c)` has approximately the same benefit as an index `t (a,c,b)`, their corresponding actions will be close to each other in the latent space. Proto-X repeats this process several times to refine the latent space. We elaborate further in Section 3.3.

3.2.2 Phase II: Holon Recommendation

Proto-X then instantiates an agent to tune the user's DBMS. The coordinator handles interfacing with the DBMS, from deploying holons to evaluating the user's objective function on the sample workload. The agent is based on the *Wolpertinger* Architecture [34], which uses an actor network to emit a proto-action, refines the proto-action to obtain a neighborhood of candidate holons, and employs a critic network to select the most promising holon.



(a) Knobs **(b)** Query Hints **(c)** Discrete Indexes

Figure 3.5: Latent Space Creation – Illustrates how Proto-X defines and creates the latent space for all supported configuration spaces depending on the space's type. Knobs includes system knobs, table knobs, and query knobs. In Figure 3.5b, "P(X)" refers to the probability function. We illustrate indexes in Figure 3.5c using an example from TPC-H [107].

In Figure 3.4 ❸, Phase II starts with the agent passing the current DBMS representation through its actor network to obtain a proto-action. The representation is either the deployed configuration (e.g., knobs, indexes) or internal metrics data (e.g., tuples processed, pages read) [130]. The agent then augments the proto-action with an exploration bias that Proto-X tweaks with small adjustments over time to encourage the exploration and coverage of less promising regions. We provide a further analysis of this in Section 3.6.6.

The agent then ❹ decodes the augmented proto-action, searches its neighborhood (see Section 3.4.2) to obtain a holon set, and ❺ uses its critic network to select the most promising candidate. ❻ The coordinator deploys the selected holon, runs the sample workload, and ❼ evaluates the user's objective function to obtain a reward. ❽ The coordinator updates the actor and critic networks with the reward and advances the agent to the next tuning step. The agent and coordinator repeat this process until the agent reaches an illegal state (e.g., an invalid configuration) or a fixed number of steps has elapsed. At this point, the coordinator resets the environment to the initial or a previously discovered configuration before resuming.

The coordinator always creates a candidate index. Existing tuning frameworks [29, 111, 134] could drop indexes by treating them as discrete actions, with a bit vector entry that indicates existence, or by using workload statistics. Even though Proto-X could use these patches, an agent should instead discern an index's value across all explored configurations and leverage configuration neighborhoods to find opportunities to drop. We defer this as future work.

3.3 Latent Space Creation

Proto-X constructs separate smaller latent spaces to exploit the nature of each configuration space with its own estimated benefits and to avoid sampling directly from high-dimensional space. Proto-X stitches these smaller spaces together into a holistic space through proto-actions in Phase II to explore and refine its decision-making. We now discuss the spaces' construction for each action type: (1) knobs, (2) query hints, and (3) discrete (e.g., indexes).

3.3.1 Knobs

This type covers system, table, and query knobs. Existing DBMSs expose knobs of different types: integral (e.g., buffer pool size), float (e.g., page fetch cost), and bool (e.g., enable hash joins). Proto-X treats them all as *continuous-type* knobs. Prior techniques observed that similar knob values produce similar observations [53, 87, 135]. For example, a buffer pool size of 1024 MB will have comparable performance to 1025 MB.Proto-X quantizes a knob's value range into equal-width buckets (default is 100 buckets). For knobs with large ranges and non-linear performance (e.g., optimizer cost knobs, memory-related knobs), Proto-X log transforms and quantizes the range to better represent the space. Increasing the quantization trades convergence speed for a finer granularity. Figure 3.5a shows how Proto-X obtains each knob's latent space value. Using the knobs' metadata from the DBMS (see Section 3.2.1), Proto-X quantizes and normalizes each knob's values into `[-1,1]` to ensure those knob values are nearby in latent space.

3.3.2 Query Hints

Figure 3.5b illustrates how Proto-X represents query hints. Proto-X first obtains the query hint's metadata (see Section 3.2.1), which includes the allowed values. ❶ It then represents the hint in latent space as a probability distribution. This representation allows the agent to express preference towards each value, such that ❷ more similar preferences are placed nearby.

We also observe that query hints can interfere with the optimizer's ability to generate optimal plans [87]. For instance, TPC-H [107] Q13 normally executes with parallelism in PostgreSQL. However, hinting to force a parallel scan (e.g., *pg_hint_plan* [1]) causes Q13 to execute without parallelism. To address this, Proto-X adds a special "Null" value to each query hint that the agent can pick to instruct the coordinator to omit the query hint.

3.3.3 Discrete

These objects require a specification (e.g., index keys) to create in the DBMS. Proto-X currently supports indexes. We defer more complex discrete types (e.g., materialized views, partial indexes) for future work. Prior approaches represent indexes with encoding schemes [111] (e.g., one-hot). These encodings are decoded into `CREATE INDEX` when deployed. However, these schemes do not guarantee that indexes of comparable performance have nearby representations. Therefore, we construct a learned latent space to enforce this [21, 76]. We illustrate this process in Figure 3.5c.

Proto-X uses a custom encoding scheme to support arbitrary indexes. This scheme allows Proto-X to consider orders of magnitude more varied indexes (see Table 3.1) than prior techniques that use a pre-curated list [111, 134]. ❸ Using the database's metadata (see Section 3.2.1), Proto-X first augments each table's indexable attribute list with a "Null" value. ❹ Proto-X then encodes each index by one-hot encoding the table, followed by one-hot encoding each key.

❺ Proto-X then obtains the index's *benefit score*, which ranks candidates [31] on their potential improvement to the target workload. Proto-X uses the workload's estimated cost as the benefit score and estimates it for read-only OLAP workloads through the query optimizer's `what-if` mechanism [22]. For OLTP workloads, Proto-X employs behavior models [73] since the optimizer does not account for maintenance operations (e.g., index inserts).

With the index representations and associated benefit scores, ❻ Proto-X then trains an encoder-decoder [44] network. The encoder maps representations (e.g., 140-dim for TPC-H) to points in a low-dimensional (e.g., 32-dim for TPC-H) latent space, and the decoder transforms points back to their original representations. To force indexes with similar scores closer, we introduce a *band constraint* [54] that restricts each index's latent space point to a range of values based on its score. This constraint improves Proto-X's efficacy by placing candidates in bands based on their estimated benefit, with more beneficial candidates close to the origin.

❼ Proto-X decodes the raw representation vector from the decoder to obtain a deployable index. Proto-X interprets each index component as a probability distribution and selects the most probable option. Proto-X first decodes the table (e.g., to `LINEITEM`) and then keeps decoding key columns until reaching the "Null" value.

3.4 Holon Recommendation

After building its latent space in Phase I, Proto-X moves on to Phase II. In this phase, Proto-X actively recommends holons to tune the DBMS. We first describe how the agent represents the current DBMS configuration to output a proto-action. We then discuss how the agent selects a holon and conclude with optimizations to assist the agent in finding promising configurations.

3.4.1 State Representation

In Figure 3.4 ❸, Proto-X outputs a proto-action from its actor network based on the current DBMS configuration. There are two choices for how to represent this information:

Telemetry Representation: This approach uses the DBMS's internal performance counters [18] (e.g., pages fetched) that it generates while executing the workload as a substitute for the DBMS configuration (e.g., knobs). Prior work commonly employs this representation [109, 134].

Structural Representation: This directly embeds the DBMS's configuration [134]. For example, fields in the representation contain system knobs' values. However, the challenge with this representation is variable-length sets (e.g., existing indexes). Prior work [134] uses a bit vector to encode whether a given object (e.g., index) exists from a pre-determined list, which Proto-X does not use. Concatenating each object's representation is also inadequate, as it requires limiting the set size to ensure a fixed-length representation. Instead, Proto-X constructs a set's representation by obtaining each object's latent space representation and averaging them [128].

Proto-X supports either representation to find promising configurations. The telemetry-based representation is simpler and more readily exploitable by an agent than the structural representation. However, the structural representation is more resilient to the DBMS's background processes, which might distort the collected metrics. We elaborate further in Section 3.6.4.

3.4.2 Candidate Neighborhood Generation

Using the DBMS's state representation, the agent passes it through its actor network (Figure 3.4 ❸) to obtain a proto-action. The agent then breaks the proto-action into *slices* and independently generates a neighborhood for each using similarity. Each slice corresponds to some configuration space (e.g., knobs, query hints, discrete). We next discuss how each space is processed below.

Knobs: Recall from Section 3.3.1 that Proto-X handles all knobs as continuous-type knobs. Based on the knob's metadata, the agent projects the proto-action slice out of the latent space $[-1, 1]$ and quantizes the result to obtain a *neighborhood center*. Consider again PostgreSQL's `shared_buffers` system knob. Assume that this knob only takes on size values (in GB) within the set $\{1, 2, 3, 4, 5\}$ and that its neighborhood center is 3 GB. Then 2 and 4 GB lie within a radius $r = 1$ neighborhood, while 1 and 5 GB have a radius $r = 2$. Using an offset -1, we obtain a similar candidate 2 GB. To generate k candidates within a radius r, the agent generates k integral offsets between $[-r, r]$. The agent shifts the center with each offset to obtain a candidate value. We found a radius of 1–3 and candidate size $\{10{,}100\}$ to balance coverage and how quickly the configuration can change. We provide a sensitivity analysis in Section 3.6.5.

Query Hints Recall from Section 3.3.2 that Proto-X represents query hints in the latent space as a probability distribution. As such, the agent first interprets the proto-action slice as a probability distribution over the value set. The agent then takes the most likely value from the distribution as the *center* and samples the distribution to obtain a neighborhood of k candidates.

Discrete: The agent passes the proto-action slice through the decoder and selects the most probable discrete action (e.g., index) representation as the neighborhood *center*. The agent then applies domain knowledge rules to obtain a neighborhood. For indexes, these *structural rules* generate candidates based on the index's definition (e.g., table, key) and do not imply similar expected performance. We found that rules that guarantee similar performance enable the agent to find better configurations more readily. We illustrate four example rules below.

(*Rule 1*) The first is the *leading prefix* rule. From an example center index of `(a,b,c)`, we obtain candidate indexes `(a,b)` and `(a)` by incrementally relaxing the index's specificity until we reach a single-column index [17]. The DBMS can use each candidate index to satisfy the same queries, albeit with fewer index predicates.

(*Rule 2*) The second is the *index type* rule that controls what data structure to use for an index. By default, Proto-X generates only B+tree indexes. With this rule, Proto-X generates candidate indexes of different types (e.g., hash table, block range index) that share the same key as the center B+tree index.

(*Rule 3*) The third rule targets index `INCLUDE` columns. With this feature, the DBMS stores non-key attributes in an index to increase the likelihood of not having to retrieve tuples for some queries (i.e., covering indexes). Proto-X analyzes the workload to obtain jointly accessed attribute sets and attaches them to applicable indexes. Consider a center index `(a,b)` with co-accessed attribute sets `(a,b,c)` and `(a,d,e)`. Proto-X will then generate two additional candidates: "`(a,b) INCLUDE (c)`" and "`(a,b) INCLUDE (d,e)`".

(*Rule 4*) The fourth rule targets per-index knobs that the DBMS exposes to alter a specific index's behavior. For example, PostgreSQL supports specifying a B+Tree's `fillfactor`, which controls how fully the DBMS packs each B+Tree index page. From each candidate index, Proto-X generates additional candidates with different knob settings specified a priori by the user. Consider a candidate index `(a,b)`. Proto-X will then generate an additional candidate "`(a,b) WITH (fillfactor=100)`".

3.4.3 Candidate Holon Selection

After the agent generates a neighborhood for each proto-action slice, it combines all the neighborhoods using a Cartesian product to produce a candidate holon set. However, this candidate set may have holons with different performance characteristics. Using each holon's latent space representation, the agent's critic network selects the most promising holon to evaluate.

3.4.4 Agent Optimizations

Although the agent finds promising configurations, its search process is inherently noisy. Proto-X employs three optimizations to guide exploration with prior experience.

Maximal Query Optimization: Proto-X's tuning space becomes drastically more complex when considering all query options for a workload due to the increased number of query plans, range of performance outcomes, and sparsity of optimal plans. The coordinator assists the agent's decision-making at each step by supplementing the agent's selected query options with query option sets from the DBMS's query optimizer or prior experience. The coordinator then maximizes over those query option sets under the same global configuration (e.g., system knobs, indexes). Proto-X only applies this optimization to OLAP and not OLTP workloads, as the framework assumes that OLAP queries are more complex than OLTP queries (e.g., point-lookup queries) and that OLAP workloads have no dependencies between queries (e.g., `INSERT` followed by `SELECT`). We provide a sensitivity analysis of this in Section 3.6.2.

Exploiting Resets To balance exploration and exploitation, Proto-X's coordinator periodically resets the DBMS environment after a fixed number of steps to some known configuration. Restoring the initial configuration maximizes exploration by allowing the agent to explore completely different trajectories. In contrast, restoring to a known configuration (i.e., a *checkpoint*) [51] forces the agent to continue exploiting its neighborhood to discover a better configuration. Proto-X supports resetting to the initial or best-discovered configuration. This tweak allows the agent to build upon prior configurations more readily to discover more complex configurations.

Timeouts As DBMS tuning tools explore configurations, they often encounter suboptimal ones. Proto-X minimizes the time spent there with two strategies using the application's SLA requirements: (1) query timeout [77, 111] limits a bad query's runtime, and (2) workload timeout [134] limits a suboptimal configuration's time. Proto-X decreases the workload timeout as it discovers better configurations. Based on empirical trials and prior work [111, 134], we utilize query timeouts of 15–30s and workload timeouts of 5–10m.

	JOB				TPC-H			
	Knob	Index	Q.Knobs	Q.Hints	Knob	Index	Q.Knobs	Q.Hints
P+DTA-S+A	N/A	73	1356	N/A	N/A	76	264	N/A
P+DTA-F+A	N/A	201	1356	N/A	N/A	183	264	N/A
UDO	24	181	N/A	N/A	24	65	N/A	N/A
UniTune	61	59	N/A	N/A	61	53	N/A	N/A
Proto-X	45	2740	1356	1090	45	2^{28}	264	108

	DSB				TPC-C		
	Knob	Index	Q.Knobs	Q.Hints	Knob	Index	Table Knobs
P+DTA-S+A	N/A	183	588	N/A	N/A	9	N/A
P+DTA-F+A	N/A	542	588	N/A	N/A	36	N/A
UDO	24	4561	N/A	N/A	33	80	N/A
UniTune	61	263	N/A	N/A	47	91	9
Proto-X	45	2^{47}	588	426	47	217	9

Table 3.1: Configuration Space Size – Number of choices considered by each tuning method in our evaluation. "N/A" indicates that the method does not support those options. "Q.Knobs" refers to query knobs, and "Q.Hints" refers to query hints.

3.5 Evaluation

We evaluate Proto-X's ability to optimize a DBMS's configuration for analytical and transactional workloads. We target PostgreSQL v15.1 running on a server with two Intel Xeon Gold 5218R CPUs (20 cores) and a 960 GB Samsung NVMe SSD. We restrict the DBMS to 32 GB of RAM and 20 worker processes. To support tuning indexes and query options in PostgreSQL, we install the *HypoPG* [49] v1.4 and a patched version of *pg_hint_plan* [1] v1.6 extensions.

We evaluate three OLAP workloads and configure all agents to minimize the overall workload runtime. **JOB** [63] is a benchmark that stresses the query optimizer with 21 tables and 113 queries. **TPC-H SF10** [107] models a business analytics workload with eight tables and 22 queries. **DSB SF10** [32] is Microsoft's extension of TPC-DS [106] that introduces additional challenges (e.g., data distributions, join patterns) with 25 tables and 53 queries. We omit four queries (Q18, Q32, Q81, Q92) due to PostgreSQL's query optimizer's limited ability to unnest subqueries [36]. We unsuccessfully attempted to rewrite those queries with Apache Calcite [14] so that PostgreSQL could complete them in a reasonable time.

We also evaluate **TPC-C** [104] SF100 with 40 terminals but modify it to remove all secondary indexes and the unique index on the `OORDER` table. We sample 1min runs through BenchBase [30] and configure agents to maximize throughput.

We first discuss the configurations for Proto-X (Section 3.5.1) and other baselines (Section 3.5.2). We then present our results in Sections 3.5.3 to 3.5.5. We defer sensitivity studies into Proto-X to Section 3.6.

3.5.1 Proto-X Configuration

We configure Proto-X to tune indexes and global knobs for all the workloads. We allow Proto-X to extract indexable attributes from workload predicates [97] and to generate arbitrary index candidates. We discuss OLAP- and OLTP-specific workload settings below:

OLAP Proto-X tunes 45 system-wide knobs and tunes indexes using all structural rules from Section 3.4.2. We allow Proto-X to build hash, B+Tree, and block range indexes. For B+Tree indexes, it tunes how full the DBMS packs each page by setting `fillfactor` to 90 (default) or 100 (full). For block range indexes, it tunes the summarization granularity (`pages_per_range`) by setting it to 64, 128 (default), or 256 pages. Proto-X tunes for each query 12 optimizer-related knobs[1], per-table access methods[2], and a query hint instructing which table to scan in parallel[3]. For our maximal query optimization (Section 3.4.4), the coordinator runs supplemental query option sets in the following order: (1) the prior step's, (2) the optimizer's, (3) the agent's, and (4) optional domain knowledge-based sets (e.g., index nested loop join, table scans into hash joins).

OLTP Proto-X tunes 47 system-wide knobs in PostgreSQL and a `fillfactor` table knob that controls how much space the DBMS leaves in a page for updates. Proto-X does not enable the index type or include rules (Section 3.4.2) for two reasons: (1) PostgreSQL cannot build multi-column hash indexes, and (2) attaching `INCLUDE` columns to an index orchanging its type to prevent point-lookups (e.g., block range index) degrades OLTP performance. Proto-X tunes how full the DBMS packs each B+Tree index page by setting `fillfactor` to 90 (default) or 100 (full). Some PostgreSQL knobs have externalities that agents cannot capture [87, 109]. For instance, increasing `max_wal_size` will improve throughput but also increase recovery time. As such, we set `commit_delay` to 0 to make transactions immediately durable, restrict `max_wal_size` to 16 GB, and do not tune autovacuum knobs due to sampling constraints.

Before tuning, Proto-X creates the latent space for discrete indexes. We sample B+tree indexes without `INCLUDE` columns and aim for reasonable coverage of the space. Using 40 parallel instances and the target workload, we sample each benchmark as follows:

- **JOB**: 2048 per-table.
- **TPC-H**: 2048 per-table, except $2^{(16-1)}$ for `LINEITEM`.
- **DSB**: 1024 on small (less than seven indexable attributes) tables. Otherwise, we generate 8192 per (table, attribute).
- **TPC-C**: 1024 per-table using behavior models [73] to account for maintenance costs from index updates.

We build latent spaces that balance *reconstruction loss* and similarity to the benefit score distribution. The reconstruction loss measures how accurately the decoder reconstructs actions

[1]Query Knobs: `enable_sort`,`enable_memoize`,`enable_hashjoin`,`enable_nestloop`,`enable_mergejoin`, `enable_gathermerge`,`enable_hashagg`,`enable_material`,`enable_parallel_hash`,`random_page_cost`, `seq_page_cost`,`hash_mem_multiplier`

[2]Query Hint: `NoSeqScan(t)`,`SeqScan(t)`

[3]Query Hint: `Parallel(t)`

from latent space points. Proto-X estimates distribution similarity by sampling 8192 points from each latent space band, decoding them, and measuring the `(table,key)` distribution. We set all agent parameters (e.g., networks, learning rate) based on prior techniques and existing ML literature [10, 75, 130]. We disclose them here [2].

3.5.2 Other Tuning Frameworks

Next, we describe the other state-of-the-art automated tuning frameworks we use in our comparison with Proto-X.

PGTune+Dexter (P+D) This baseline first runs PGTune [60], a heuristics-based knob tuner. We then run Dexter [11] to recommend indexes based on HypoPG [49] and PostgreSQL optimizer's workload costs. For TPC-C, we provide a representative query trace to Dexter.

PGTune+DTA-S+AutoSteer (P+DTA-S+A) This runs PGTune, followed by Microsoft's Anytime Database Tuning Advisor (DTA) algorithm [23]. DTA-S uses Hyrise's implementation with their settings [57]: (1) no storage budget, (2) 30m tuning budget, and (3) only two-column indexes. After DTA-S finishes, we run AutoSteer [12], which tunes query knobs by greedily toggling and merging boolean knobs. We configure AutoSteer to tune the same knobs as Proto-X and add toggles to infuse the prior domain knowledge utilized by our maximal query optimization.

PGTune+DTA-F+AutoSteer (P+DTA-F+A) We replace DTA-S in P+DTA-S+A with DTA-F. In DTA-F, we allow DTA to consider more indexes without a time limit. Due to OOM issues and PostgreSQL's inability to optimize queries with 1000s of hypothetical indexes, we limit DTA-F to indexes with three, five, three, and four columns for TPC-H, JOB, DSB, and TPC-C, respectively.

UDO This is a holistic framework that supports tuning both system knobs and indexes [111]. We use UDO's open-source implementation to construct each benchmark's candidate index list and configure the agent with their default parameters. To ensure a fair comparison, we extended UDO's PostgreSQL knob list to include (1) 24 knobs related to query optimization and resource management and (2) `fillfactor` table knobs for TPC-C. We alter UDO to obtain TPC-C samples through BenchBase [30] and employ 30s, 30s, and 60s query timeouts for JOB, DSB, and TPC-H, respectively. We poll UDO every 15m for the best configuration.

UniTune Lastly, we compare against Alibaba's UniTune framework [134], modified to support PostgreSQL and obtain samples through BenchBase. We use the same parameters released by the authors. UniTune optimizes the same system knobs as Proto-X, and **UniTune+QOpts** uses the same query options as Proto-X with the same prior domain knowledge. We also enable UniTune's query rewriting via Calcite to match their paper. UniTune only constructs single-column indexes because complex indexes lead to a combinatorial explosion in its one-hot representation. We made two improvements to UniTune. First, we run queries serially and set the target objective to minimize workload runtime. We set UniTune's space budget to 2 TB for comparative fairness and to allow it to more consistently find promising configurations.

3.5.3 OLAP Performance Comparison

We start by comparing Proto-X to the other frameworks regarding their ability to optimize OLAP workloads. We run the frameworks on the same hardware for multiple *trials* (i.e., independent tuning periods) with different random seeds. At the start of each trial, we initialize PostgreSQL with its default configuration and load the database. We then run each agent for 30h to tune the DBMS. As agents utilize timeouts during exploration, we evaluate each discovered configuration without timeouts to obtain their actual performance. After deploying each configuration, we empty the OS page cache, run the workload three times, and report the min [62].

We run four trials of each agent and report results for all trials. In Figure 3.6, we plot the mean performance of each agent's best configuration. We also report the best and worst performance achieved by any agent's trial in Table 3.2. The baseline for all trials is PGTune+Dexter (P+D), as this represents the easiest and fastest method for tuning a DBMS since there is no learning. Our analysis focuses on whether a method generates better configurations than P+D.

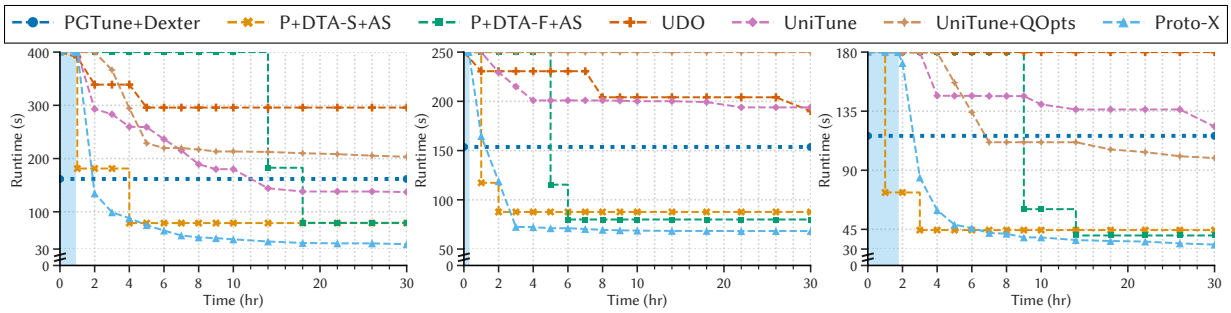
JOB In Figure 3.6a, all methods start with the same performance. UDO flattens after 5h and fails to improve over P+D because it does not pick the correct indexes. Although UDO considers 181 index candidates, only 3–5 indexes improve the workload by a measurable amount. Whereas Proto-X's latent space considers the expected benefit of indexes, UDO initially considers all indexes equally. Thus, UDO is unable to reliably pick the correct indexes.

UniTune builds an average of 29 out of 59 index candidates. In Figure 3.6a, UniTune surpasses P+D after 12h and flattens after 15h due to the limited tuning options available. UniTune+QOpts has more potential as it tunes query options, but it does not manage the additional complexity and fails to surpass P+D.

Proto-X builds its latent space and surpasses P+D within 2h. After 5h, Proto-X matches P+DTA-S+A and P+DTA-F+A and finds better configurations. Proto-X quickly identifies a high-value set of 2–5 indexes and continues to improve it with additional indexes and query options. We attribute the descent smoothness in part to Proto-X's maximal query optimization (see Section 3.4.4): the agent exploits prior knowledge about promising query option sets while simultaneously exploring unevaluated query option sets, indexes, and global knobs. From Table 3.2, both Proto-X's worst (41s) and best (36s) trials yield a 46–53% speedup over the next best P+DTA-S+A.

TPC-H Unlike the previous workload, the results in Figure 3.6b show that only Proto-X, P+DTA-S+A, and P+DTA-F+A surpass the baseline P+D for TPC-H. One of this workload's most important actions is to find the high-value `LINEITEM (l_partkey)` index that all frameworks find. However, UDO and UniTune do not find system knobs better than PGTune. UniTune dedicates more time to query rewriting and indexes when it should tune knobs (e.g., parallelism).

P+DTA-S+A and P+DTA-F+A find their best configuration after 2h and 6h, respectively. Proto-X mostly plateaus after 10h, but Proto-X still outperforms P+DTA-S+A and P+DTA-F+A by tuning a query hint that selects a table to scan in parallel. From Table 3.2, the worst (69s) and best (66s) trial of Proto-X yields a 4–8% speedup over P+DTA-F+A's best run (72s).



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

Figure 3.6: OLAP Performance Comparison – The DBMS's performance achieved using the frameworks' configurations over time on JOB, TPC-H, and DSB. We plot the mean performance obtained by four trials of each agent. The shaded region for Proto-X is the time spent constructing the latent space.

	JOB		TPC-H		DSB	
	Min	Max	Min	Max	Min	Max
PGTune+Dexter	162s	162s	154s	154s	116s	116s
P+DTA-S+AS	77s	79s	81s	93s	43s	46s
P+DTA-F+AS	78s	80s	72s	87s	40s	40s
UDO	189s	400s	144s	250s	188s	200s
UniTune	132s	146s	176s	200s	117s	125s
UniTune+QOpts	126s	237s	250s	250s	98s	100s
Proto-X	36s	41s	66s	69s	31s	35s

Table 3.2: OLAP Performance Spread –The best and worst performance achieved by a framework's four trials in Figure 3.6 on JOB, TPC-H, and DSB.

DSB From Figure 3.6c, UDO and UniTune do not surpass P+D. Similar to before, UDO struggles to identify indexes from 4561 candidates, and UniTune misses knob configurations from misallocating time to index tuning and query rewriting. In this case, UniTune+QOpts improves over P+D by building indexes on `STORE_SALES` and utilizing Proto-X's prior domain knowledge.

P+DTA-S+A and P+DTA-F+A find better configurations after 3h and 14h, respectively, before plateauing. Proto-X spends the first 2h constructing its latent space, surpasses P+DTA-S+A within 6h, maintains its lead over P+DTA-F+A, and continues to find better configurations by building more indexes and tweaking query options. Table 3.2 shows that Proto-X's worst (35s) and best (31s) trial has a 13–23% speedup over P+DTA-F+A's best run (40s).

3.5.4 OLTP Performance Comparison

We next compare Proto-X to the other frameworks on their ability to improve TPC-C for 8h. We report the average of three 1m BenchBase [30] runs for each discovered configuration. We plot the mean performance of the best configuration across the four trials of each agent. We also present each agent's worst, median, and best performance in Table 3.3.

The results in Figure 3.7 show that UDO does not exceed P+D because it does not identify the correct indexes or knob configurations. For OLTP, picking indexes on tables with data actively modified negatively impacts throughput due to additional index maintenance operations. UDO picks indexes that hinder performance, such as indexes on `OORDER` or `ORDER_LINE`.

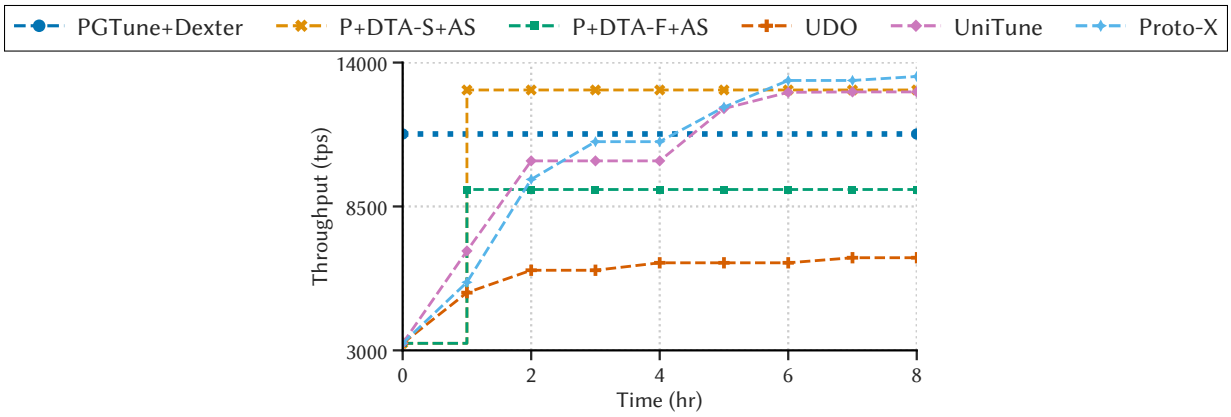


Figure 3.7: OLTP Performance Comparison – The DBMS's performance achieved using the frameworks' configurations over time on TPC-C. We plot the mean performance obtained by four trials of each agent. The shaded region for Proto-X is the time spent constructing the latent space.

	TPC-C		
	Min	Median	Max
PGTune+Dexter	11.3k	11.3k	11.3k
P+DTA-S+A	12.9k	12.9k	12.9k
P+DTA-F+A	9.1k	9.1k	9.1k
UDO	5.7k	6.3k	7.6k
UniTune	12.7k	12.8k	12.9k
Proto-X	13.1k	13.4k	13.8k

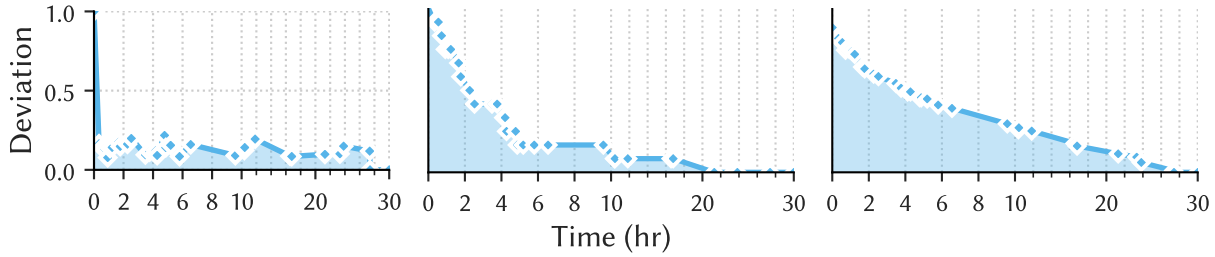
Table 3.3: OLTP Performance Spread of Frameworks – The worst, median, and best performance achieved by a framework's four trials in Figure 3.7.

In comparison, UniTune and Proto-X exceed the performance of P+D by tuning table knobs and avoiding a bad index on `OORDER` that Dexter suggests. From Table 3.3, Proto-X's best run (13.8k) is better than UniTune's (12.9k) by 7%. Proto-X picks an index on `CUSTOMER` that includes `c_last`. This index improves performance in 47% of TPC-C's transactions (i.e., `Payment`, `OrderStatus`). Proto-X maintains this 7% improvement over P+DTA-S+A (12.9k tps). P+DTA-S+A picks the correct `CUSTOMER` index but also picks bad indexes on `OORDER`. This behavior worsens for P+DTA-F+A (9.1k tps), as DTA considers more (13) but mostly bad indexes.

3.5.5 Configuration Time Analysis

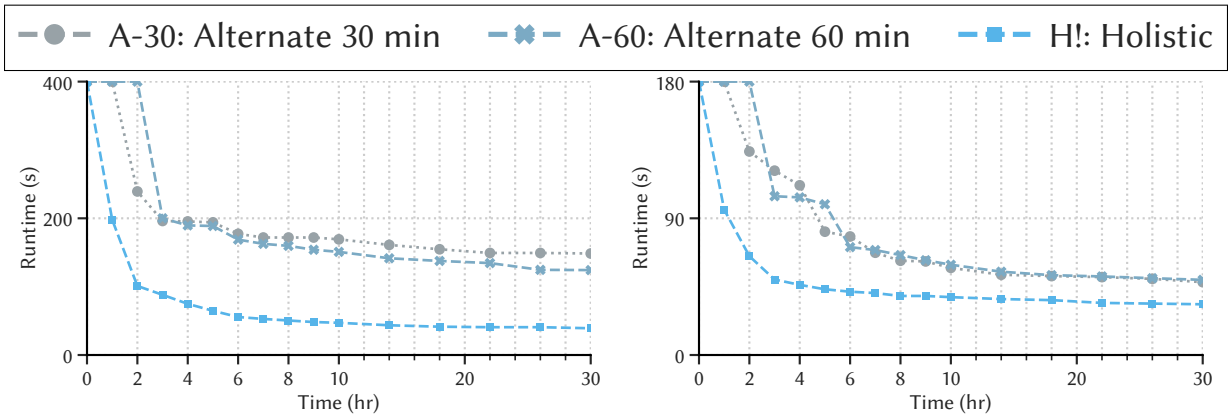
To better understand Proto-X's tuning behavior, we examine the configurations it generates over time. This analysis shows how Proto-X exploits proto-actions to find impactful configurations despite the high dimensionality of the solution space.

Using Proto-X's best-performing trial on JOB from Section 3.5.3, we track the normalized deviation of configurations over time. We compute deviation as the average absolute difference (e.g., L1-distance) between each configuration and the final configuration's values. For indexes, we track the percentage of indexes that are missing from the final configuration. We only consider 12 resource-related knobs (e.g., workers, memory) for the system knob deviation.



(a) System Knobs **(b)** Indexes **(c)** Query Options

Figure 3.8: Configuration Time Analysis – Normalized deviation from the final configuration over time for 12 resource-related system knobs, indexes, and query options using Proto-X's best JOB trial. A configuration's deviation is the average difference of values from the final configuration.



(a) JOB **(b)** DSB

Figure 3.9: Ablation on Holistic vs Sequential Tuning – Performance using holistic versus sequential tuning of components over time on JOB and DSB. We plot the mean performance obtained by four trials of each mode.

We see in Figure 3.8a that the system knobs deviation is about the same after an initial correction in the first hour. Instead of changing them further, the agent focuses on indexes and query options. Figure 3.8b shows that Proto-X finds a preliminary set of good indexes after 6h and comes to a final index set after 21h. Contrast this with its behavior in Figure 3.8c, where it starts with most of the query options being different and gradually alters them over 30h.

3.6 Sensitivity Experiments

We next analyze aspects of Proto-X in more detail. We begin with an ablation study on Proto-X's holistic approach in Section 3.6.1, followed by sensitivity experiments in Sections 3.6.2 to 3.6.6. We conclude with an experiment on generalizing Proto-X's latent space in Section 3.6.7.

3.6.1 Ablation on Holistic vs Sequential Tuning

We first evaluate whether Proto-X outperforms other agents because it holistically reasons across all configuration spaces. We use Proto-X's JOB and DSB configurations from Section 3.5.3 and run three modes: (1) tune each component in 30m intervals (**A-30**), (2) tune each component in 60m intervals (**A-60**), and (3) holistic mode (**H!**). We run each mode four times and plot the mean performance of the best configuration achieved so far over the 30h tuning period in Figure 3.9. We discuss each benchmark below.

JOB In Figure 3.9a, H! (39s) finds configurations that are 73.6% and 68.5% better than those found by A-30 (148s) and A-60 (124s), respectively. Both A-30 and A-60 are prone to missing beneficial indexes, with some trials failing to add any indexes at all. We attribute this to the coordination problem: if the query tuner turns off index scans, then the index tuner will not observe an index's benefit. By considering the spaces holistically, H! avoids this problem.

DSB In Figure 3.9b, H! (33s) finds configurations that are 29.8% and 32.7% better than those found by A-30 (47s) and A-60 (49s), respectively. We note that A-30 outperforms A-60 on DSB but performs worse on JOB. Refining this tuning interval may lead to different outcomes. Next, we analyze the composition of each trial's recommended indexes by examining the number of unique one- and two-column indexes. On average, H! builds 13 single-column and 23 two-column indexes, whereas A-30/A-60 builds eight and 11, respectively. By exploring and constructing more diverse indexes, H! finds better configurations than both A-30 and A-60.

3.6.2 Maximal Query Optimization

The maximal query optimization allows Proto-X to leverage the DBMS optimizer's expertise, past experience, and known priors to guide its recommendations (Section 3.4.4). In this experiment, we consider different modes by combining the following option sets:

- **Agent (A):** Query option set chosen by the agent.
- **Previous (P):** Query option set from the previous time step.
- **Global (G):** Query option set from the query optimizer.
- **Infusion (IV):** Query option sets based on prior domain knowledge about performant query plans to guide the agent. We utilize one option set that enforces index nested loop joins and another that enforces table scans into hash joins.

Using these option sets, we construct the five modes in Table 3.4 by increasing the amount of guidance. We begin with M1 (**A**), which only uses the agent's chosen query options. We augment M1 with **P** (previous) and **G** (query optimizer expertise) to obtain M2 and M3, respectively. M4 combines both M2 and M3. Lastly, we derive M5 by augmenting M4 with **IV** to infuse prior knowledge about performant query plans. We use the same Proto-X configuration for tuning TPC-H from Section 3.5.3. As Proto-X's search plateaus in Figure 3.6b after 10h, we run each mode for only 10h (four trials each).

Table 3.4 shows that executing only the agent's query option set (M1) does not find a good configuration. In contrast, M2, M3, and M4 find configurations outperforming the baseline

Mode	Min	Mean	Max
M1 (**A**)	250s	250s	250s
M2 (**P,A**)	74s	84s	98s
M3 (**G,A**)	91s	93s	94s
M4 (**P,G,A**)	69s	74s	84s
M5 (**P,G,A,IV**)	66s	68s	69s

Table 3.4: Maximal Query Optimization – Proto-X's worst, mean, and best TPC-H performance under different maximal optimization modes.

P+D (154s). Some M4 configurations (69s) yield a 4% improvement over P+DTA-F+A (72s) in 8h. Using the **G** and **P** sets, the agent leverages the expert query optimizer and past experience to guide its recommendation of query options (e.g., per-table access methods). This optimization further allows us to imbue the agent with priors from domain knowledge. From Table 3.4, M5's use of query option sets (**IV**) enables the agent to reduce the best trial's runtime from 69s to 66s and the range of workload runtimes from 15s to 3s.

3.6.3 Actor-Critic Sensitivity

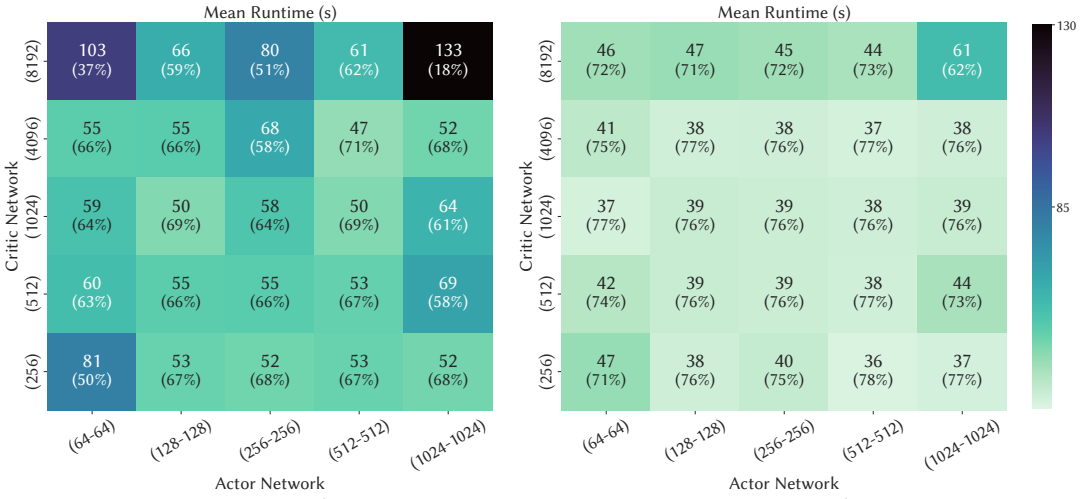
As discussed in Section 3.2, the agent's actor network outputs a proto-action, and the critic network selects the best holon. To understand their impact, we vary five actor and five critic networks of increasing complexity [7, 10, 75, 130]. We run four 30h trials for each pair using Proto-X's JOB configuration from Section 3.5.3. We report the mean performance and percent improvement over P+D in Figure 3.10 at 8h and 30h.

From the 8h grid in Figure 3.10a, the lower-center network pairs have a similar 64–69% improvement over P+D. However, for small (64-64) and large actor networks (1024-1024) and large critic networks (8192), the performance is more varied, with ranges of 37–66%, 18–68%, and 18–62%, respectively. We attribute these differences primarily to network stochasticity due to limited samples.

By 30h in Figure 3.10b, all pairs achieve at least 71% improvement over P+D with a 7% spread, except for (1024-1024, 8192)'s 62%. These results illustrate that the agent's performance tends to worsen for overly simple (e.g., actor 64-64) or complex networks (e.g., critic 8192). Whereas overly simple networks lack reasoning ability, larger networks have more parameters that require more exploration data to learn. Although Proto-X finds promising configurations with a range of actor-critic networks given enough time, Proto-X may benefit from hyperparameter optimization [10] in situations with limited tuning budgets and available idle instances [72].

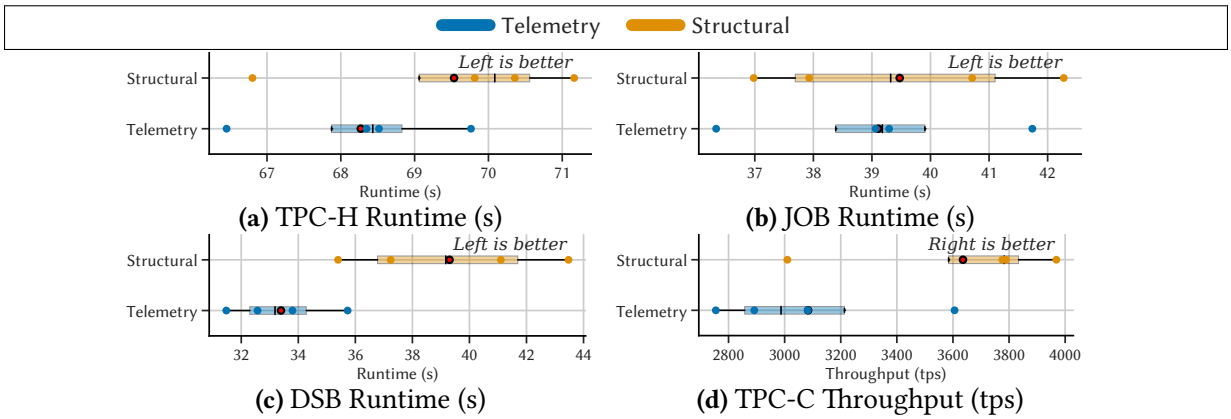
3.6.4 State Sensitivity

We next analyze the impact of Proto-X's state representation. Recall from Section 3.4.1 that Proto-X utilizes either a telemetry (e.g., workload metrics) or structural (e.g., embed knob values and indexes) representation. Better representations enable the agent to understand the current DBMS configuration and make more targeted recommendations. We run four 30h trials of Proto-X using each representation. In Figure 3.11, we report the performance of the best configuration discovered by Proto-X from all trials.



(a) JOB at 8h **(b)** JOB at 30h

Figure 3.10: Actor-Critic Sensitivity – Proto-X's performance on JOB with different actor-critic network pairs at 8h and 30h. Each cell contains the mean performance and percent improvement over P+D across four runs. Lighter colors correspond to lower workload runtimes.



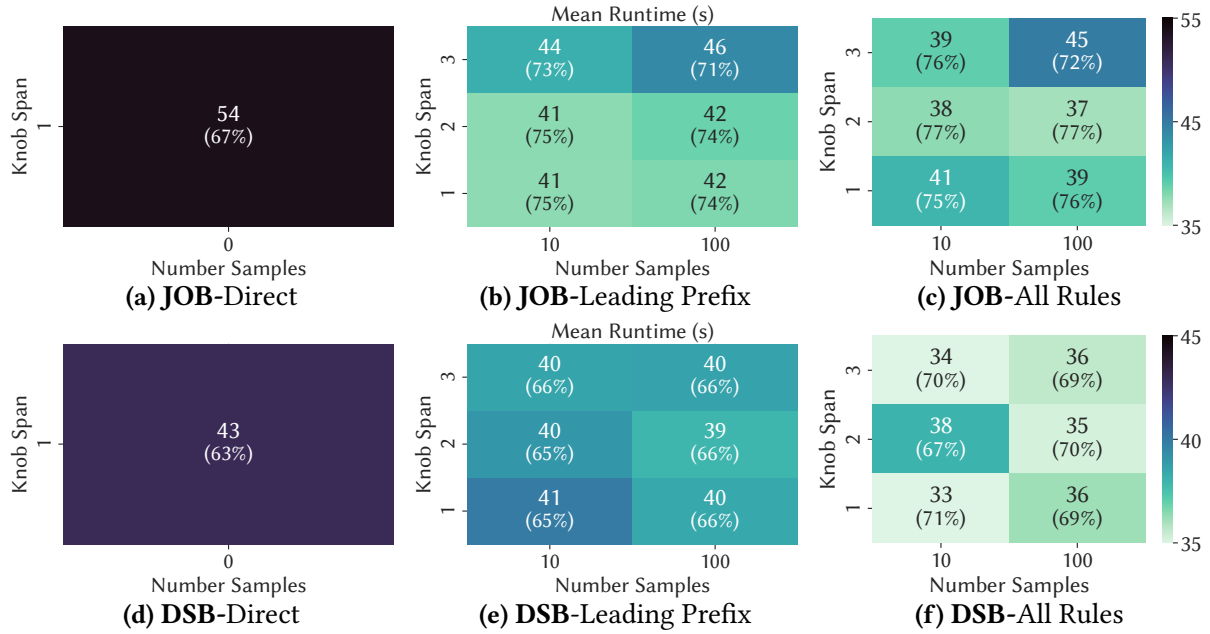
(a) TPC-H Runtime (s) **(b)** JOB Runtime (s)

(c) DSB Runtime (s) **(d)** TPC-C Throughput (tps)

Figure 3.11: State Sensitivity – Workload performance after 30h with four runs of each agent using different state representations. The box plot represents min/max and the red dot indicates the mean.

On average, for TPC-H (Figure 3.11a) and JOB (Figure 3.11b), telemetry performs marginally better than structural by 1s (1.4%) and 0s, respectively. For DSB, Proto-X with telemetry representation finds a configuration 4s (11%) faster than any found by structural, which we attribute to the simpler telemetry-based representation.

Figure 3.11d shows that Proto-X's state representation matters the most for TPC-C. Although both representations find good configurations, telemetry has a wider range (2.2k tps) than the structural representation (0.5k tps). When executing OLTP workloads, we do not have fine-grained control over when PostgreSQL runs its background autovacuum worker. Thus, the telemetry representation may be unstable due to these background processes.



(a) JOB-Direct **(b) JOB-Leading Prefix** **(c) JOB-All Rules**

(d) DSB-Direct **(e) DSB-Leading Prefix** **(f) DSB-All Rules**

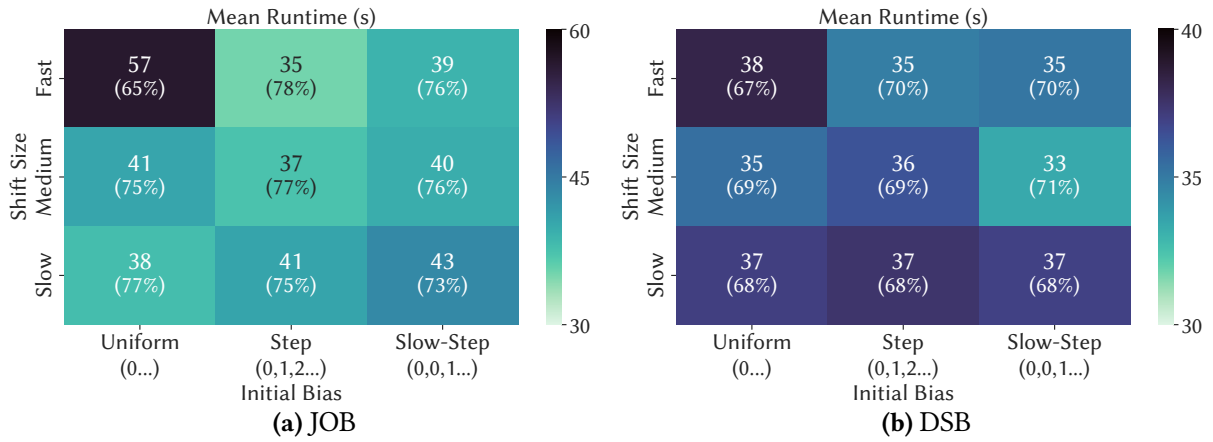
Figure 3.12: Neighborhood Sensitivity – Performance of Proto-X on JOB and DSB with different neighborhood construction parameters. Each cell contains the mean performance and percent improvement over P+D across four runs. Lighter colors correspond to lower runtimes.

3.6.5 Neighborhood Sensitivity

We next examine Proto-X's neighborhood exploration to find valid candidate holons. To manage combinatoric overhead, Proto-X approximates the neighborhood with three parameters: (1) knob span or radius (1, 2, 3), (2) number samples (10, 100), (3) enabled structural rules (leading prefix only, all rules). We introduce a base case that emits the closest direct action, which sets the knob span to 0, samples to 1, and disables structural rules. We run four 30h tuning trials for each configuration using Proto-X's JOB and DSB configurations from Section 3.5.3. We report the mean performance of each neighborhood and its percent improvement over P+D in Figure 3.12.

In Figure 3.12, picking the closest direct action is not the best decision. Constructing a neighborhood allows Proto-X to discover better configurations with further improvements over P+D of up to 10% (JOB) and 8% (DSB). Comparing Figure 3.12c to Figure 3.12b for JOB and Figure 3.12f to Figure 3.12e for DSB, enabling more structural rules tends to further improve performance over P+D: 1–3% (JOB) and 2–7% (DSB). Additional structural rules help facilitate a richer selection of candidate actions (e.g., `INCLUDE` columns, index type). Proto-X benefits from more structural rules, particularly when they produce candidates with better performance.

For knob span and number of samples in Figure 3.12, Proto-X's performance on JOB and DSB slightly degrades as the knob span increases for a given sample size. Increasing knob span makes the neighborhood sparser, which increases the agent's stochasticity and hinders materializing beneficial candidates. Although increasing the sample size can counteract this effect, it makes it more difficult for the critic to differentiate between choices early on. For our experiments, we use a span of 1 across TPC-H, DSB, and JOB. We use 10 samples for TPC-H and DSB due to the smaller query knob space and 100 samples for JOB to ensure reasonable coverage.



(a) JOB **(b)** DSB

Figure 3.13: Exploration Sensitivity – Performance of Proto-X on JOB and DSB with different initial bias settings and increment speeds. Each cell contains the mean performance and percent improvement over P+D in parentheses across four runs. Lighter colors correspond to lower runtimes.

3.6.6 Exploration Sensitivity

Proto-X augments its proto-action with a bias to encourage exploration (Section 3.2). To measure its impact, we vary the bias's two parameters: (1) initial distribution over the starting episode and (2) shift size at the end of each episode. We choose shift sizes of **slow** (1), **medium** (2), and **fast** (5) to capture a range of speeds and initial distributions of **uniform** (e.g., 0...), **step** (e.g., 0,1,2,...), and **slow-step** (e.g., 0,0,1,...). We run four 30h trials with Proto-X's JOB and DSB configurations from Section 3.5.3. We report the mean performance against P+D in Figure 3.13.

The (Fast, Uniform) bias setting in Figure 3.13a stands out, with its mean runtime being >14s worse than all other. Setting the bias to (Fast, Uniform) prevents the agent from sufficiently covering the latent space, thus leading it to miss important candidate indexes. The remaining bias settings result in similar improvements over P+D, with 73–77% for JOB and 67–71% for DSB. A more targeted bias setting allows the agent to aggressively exploit latent space regions while ignoring irrelevant ones. We use slower shift sizes in our experiments for space coverage.

3.6.7 Generalization

In real-world deployments, the DBMS environment under tuning in Phase II may differ from that in Phase I due to dataset drift, workload drift, or schema changes. We now analyze whether Proto-X's latent space effectively guides the agent to promising configurations even when its benefit scores are outdated. We first discuss dataset/workload drift, followed by schema changes.

Dataset/Workload Drift: We use DSB for this experiment as it supports dataset scaling (scale factor) and has more complex parameter distributions. We use Proto-X's configuration from Section 3.5.3, with a latent space built on SF10 and the workload's original 49 queries. We simulate dataset drift by evaluating on SF1/SF20 and workload drift by generating 2× and 3× the original workload. We run four 30h trials for each scenario. We use P+DTA-F+A, the next best agent from Section 3.5.3, as the baseline.

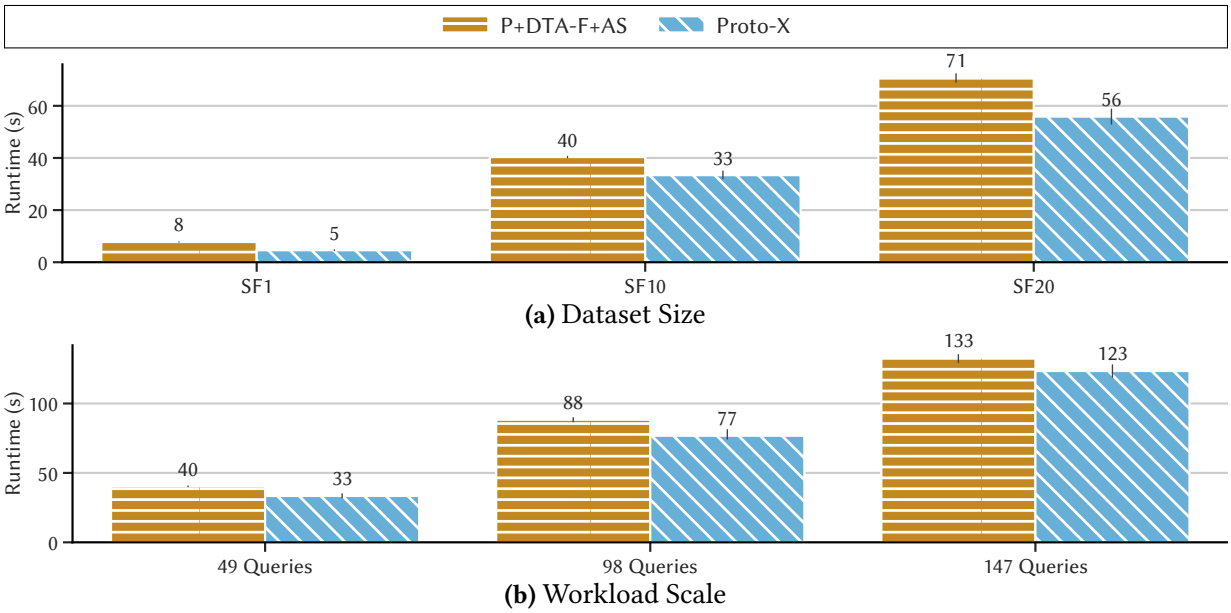


Figure 3.14: Dataset/Workload Drift – Using a latent space built on benefit scores from DSB SF10 with the 49 query workload, Proto-X tunes different scale factors (dataset drift) and workload scales (workload drift). We plot the mean performance of the best configuration discovered from four trials of P+DTA-F+A and Proto-X after 30h.

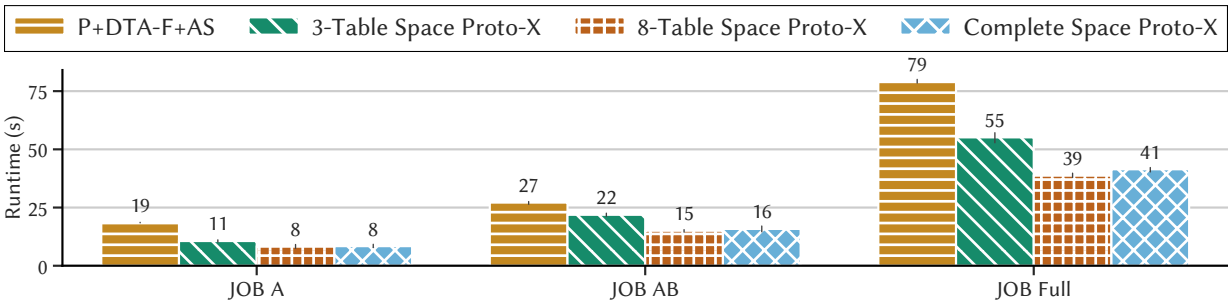


Figure 3.15: Schema Changes – Mean performance on different JOB workloads by four trials of P+DTA-F+A and Proto-X in 30h. Proto-X trains its latent space based on the workload of JOB AB (66 queries) and considers either 3-tables, 8-tables, or all tables as candidate latent spaces.

From Figure 3.14a, Proto-X is better than P+DTA-F+A across all scale factors, with improvements of 38% (SF1), 17.5% (SF10), and 21.1% (SF20), respectively. Indexes that are beneficial at SF10 remain beneficial at SF1/SF20. This continuity allows Proto-X to exploit the SF10 latent space to improve at smaller and larger sizes.

In Figure 3.14b, Proto-X uses a latent space built from a smaller workload to tune more complex ones. Proto-X finds better configurations than P+DTA-F+A at all workload scales, with improvements of 17.5% (49 queries), 12.5% (98 queries), and 7.5% (147 queries). Although more complex workloads distort the benefit scores of multi-column indexes, Proto-X is hindered more by increased workload runtime, partly due to our maximal query optimization.

Schema Changes We use JOB to evaluate Proto-X's resilience to schema changes, as its optimal index set touches more tables than the other workloads (Section 3.5.3). We divide the JOB workload into two sets. The first set (**JOB A** / 33 queries) contains all queries in the benchmark whose name ends with "a" (e.g., Q1a). The second set (**JOB AB** / 66 queries) comprises the queries that end in either "a" or "b". We simulate schema changes by constructing the latent space in Phase I on smaller sets of tables. We build three spaces using the JOB AB workload: 3-Table[4], 8-Table[5], and Complete. We then run four 30h trials and configure Proto-X to tune JOB A, JOB AB, and JOB Full. We present the mean performance of the best-discovered configuration for each scenario in Figure 3.15 compared to P+DTA-F+A.

From Figure 3.15, Proto-X finds better configurations than P+DTA-F+A in all cases, with improvements of 42–58% (JOB A), 19–44% (JOB AB), and 30–51% (JOB Full). We focus on the differences between the three latent spaces. Figure 3.15 shows that 3-Table performs worse due to insufficient coverage of the database's schema (e.g., `MOVIE_COMPANIES`), which causes Proto-X to miss beneficial indexes. Active learning [72] could ensure that the latent space benefit scores remain current as the schema changes.

8-Table and Complete yield similar outcomes in Figure 3.15, which indicates the limited benefit of incorporating excess information into the latent space. We note that 8-Table (39s) is competitive, and Complete (41s) performs marginally worse than the latent space built on JOB Full from Figure 3.6a (39s). We attribute this to agent's stochasticity in building the latent space.

3.7 Future Work

This work presents the first step towards a holistic or "universal" approach to DBMS optimization. There remain multiple opportunities for further enhancement on top of the core Proto-X framework: (1) online adaptation of the latent space in response to schema changes and drift, (2) incorporating less accurate feedback signals (e.g., partial workload execution), (3) handling user constraints by re-using the agent's exploration trajectories, and (4) improving the agent's introspection abilities to explain its decisions.

3.8 Conclusion

Recent work has focused on combining individual DBMS tuning agents to find better configurations. However, these approaches struggle to coordinate their individual agents. We introduce the framework Proto-X that holistically tunes across multiple configuration spaces. Proto-X organizes the configuration space into a latent space and uses proto-actions to navigate through neighborhoods of similar configurations. Evaluating against other state-of-the-art methods on their ability to tune PostgreSQL for OLAP and OLTP workloads, Proto-X discovers configurations that improve performance by up to 53% over the next best approach.

[4] `CAST_INFO`, `MOVIE_INFO`, `MOVIE_KEYWORD`

[5] 3-Table, `AKA_NAME`, `MOVIE_COMPANIES`, `MOVIE_INFO_IDX`, `PERSON_INFO`, `TITLE`

Chapter 4

Workloads-Configurations Similarity for Adaptation

As shown in Chapter 3, optimizing a DBMS for a fixed workload has been well explored by the academic community. Early research focused on heuristics and cost-based search techniques [11, 22, 23, 60, 99]. The last decade saw the rise of using ML-based techniques to tune aspects of the DBMS (e.g., knobs [47, 53, 109, 130], indexes [57, 97, 116]) or across all aspects of the DBMS [132]. Recent advances have focused on large language model (LLM)-based methods [39, 47].

Another key aspect to enable holistic DBMS optimization over its lifetime is being able to adapt its configuration effectively in response to changes in the deployment. For instance, the workload's queries may evolve over time, the database size may grow over time, or the application's schema may change. Existing automated database tuners struggle to adapt and re-optimize database management systems (DBMSs) in response to these changes due to their brittle design and reliance on workload-level observations. For example, heuristic tuners [11, 60] cannot incorporate historical knowledge due to their fixed algorithms. ML-tuners [58, 132, 134] struggle as environment changes (e.g., Q1 no longer exists) corrupt their internal representations (e.g., how they represent Q1's tunables). These design issues prevent tuners from exploiting query-level semantics (e.g., plans) to learn from historical knowledge. Consequently, these tuners take longer to re-optimize after environment changes and end up stuck in subpar configurations.

In this chapter, we present the framework Booster that assists existing tuners in adapting to environment changes. Booster first structures historical artifacts into query-configuration contexts. On a per-query basis, Booster then identifies the most relevant historical contexts as references and then prompts large language models (LLMs) to suggest configurations based on those references. Next, Booster composes the query-level suggestions into a holistic configuration [132] with beam search. Booster then provides its findings to the tuner being assisted for further refinement.

The rest of the chapter is organized as follows. We provide the key ideas behind Booster in Section 4.1.4 and its architecture in Section 4.2. We then provide an evaluation of Booster's ability to assist different state-of-the-art tuners in adapting to environment changes in Section 4.6. By composing recommendations derived from query-level insights, Booster assists tuners in discovering configurations that are up to 74% better and in up to 4.7× less time than continuing to tune from historical configurations.

4.1 Background: Adaptivity and LLM-based Adaptation

An autonomous self-driving DBMS [56, 86, 142] optimizes itself without human intervention. Guided by the user's objectives (e.g., minimize runtime), the DBMS optimizes itself for a given workload by deploying *tuners* to find optimal *holistic configurations* [132] that encapsulate all of the DBMS's tunable facets. For instance, a single *holistic configuration* can prescribe system knobs to set, indexes to build, and hints to apply for each query in the workload. To find these holistic configurations, the DBMS's tuners iteratively explore and experiment with different configurations to the best of their individual capabilities. These include cost-based search and ML-based tuners that target individual DBMS aspects (e.g., knobs [53, 60, 109, 130], physical design [31, 57, 97, 116]), holistic tuners that reason across multiple DBMS aspects [132], and large language model (LLM)-based tuners [39, 47, 123]. We first define adaptivity for tuners and then discuss existing tuners along with their challenges in adapting to different deployments.

4.1.1 Adaptivity for Tuners

As a tuner optimizes DBMSs, it acquires experience that includes the tuner's reasoning, explored configurations, and observed behavior of the DBMS. *Adaptivity* reflects a tuner's ability to reuse prior experience when tuning new scenarios, broadly categorized as *transfer* and *drift*. We will discuss each separately.

Transfer Scenario This addresses the case where a tuner transfers experiences from past DBMS deployments to a previously unseen deployment. This ranges from cases where the historical and target deployments are the same to cases where the schemas differ. For example, a tuner transfers its experiences from tuning a TPC-H [107] instance to a TPC-DS [106] instance.

Drift Scenario This covers scenarios where a tuner optimizes a DBMS deployment whose characteristics change over time [65, 110, 111]. These include changes to query parameters, query templates, query volume (i.e., load spikes), hardware (e.g., instance upgrades), and data (e.g., `BULK INSERT`). These drifts are common, with Redshift observing that 50% of their production clusters have 50% of queries repeating exactly (i.e., same query template and parameters) daily [110].

4.1.2 Existing Tuning Frameworks

We next provide an overview of existing state-of-the-art tuners, which we group into three categories: (1) heuristic and cost-based, (2) ML-based, and (3) LLM-based.

Heuristic and Cost-based Tuners These tuners execute a fixed algorithm to explore configurations [11, 12, 23, 60] obtained with heuristics. They then rely on query plan costs via a "what-if" mechanism [22] to guide the search process.

ML-based and Holistic Tuners These tuners rely on ML techniques to tune the DBMS. While some of these tuners only target specific aspects (e.g., knobs, indexes) of the DBMS [58, 97, 109, 130], holistic tuners [132] reason across all aspects. These tuners rely on an internal model to find beneficial configurations through trial and error. These models capture deployment aspects (e.g., workload, schema) as bits in the models' representation. For instance, a specific bit output by the model may instruct the tuner to build an index [58, 111], set a system knob [109, 130], or apply a query hint [132].

LLM-based Tuners Recent large language model (LLM)-based tuners rely on off-the-shelf models [39] (e.g., GPT-4o [81]) or models fine-tuned on prior experience [47]. They then instruct the LLM with *prompts*. For instance, a prompt can instruct the LLM to assume the role of a database administrator and output a JSON configuration that optimizes the workload. These tuners sample multiple configurations from the LLM and select the best one.

4.1.3 Tuner Adaptivity Challenges

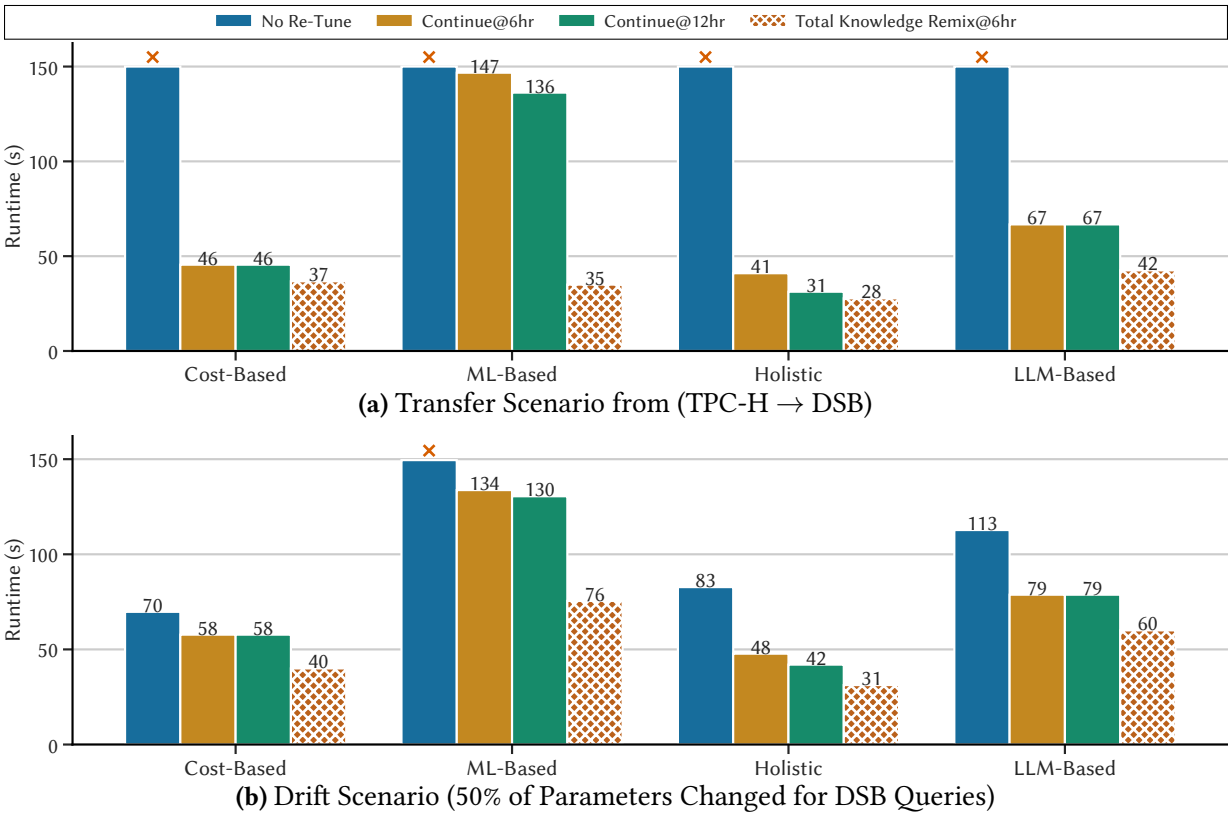
Despite the prevalence of transfer and drift scenarios in practice, existing tuners are unable to adequately adapt to environment changes due to their designs. As we now discuss, their design limitations are due to two core issues:

Rigidity and Brittleness This issue is present in cost-based and ML-based tuners. For both, identifying relevant experiences is critical to discovering beneficial configurations [67, 109, 133, 135]. Even if relevant experience is accurately identified, problems remain.

Cost-based tuners lack a mechanism to apply prior insights to guide the search. For instance, if prior experience reveals an adverse interaction between two indexes, the tuner should prioritize exploring configurations that exclude those two indexes.

ML-based tuners must re-optimize whenever their internal models' representations change due to the environment (e.g., new query, new indexable column). To compensate for this, researchers proposed using relevant experience to pre-train the internal model [66]. However, pre-training remains infeasible due to potential differences with the target environment and the overhead of re-evaluating historical experiences (i.e., configurations) to obtain performance values for the target schema and workload.

Workload Granularity Existing tuners adapt at a workload granularity. Rather than tune from scratch, tuners use workload-level telemetry (e.g., DBMS metrics) to identify and start from some historical configuration (i.e., workload mapping [109]). However, this mapping prevents combining individual query insights across configurations. For instance, configurations C1 and C2 may optimize different workload queries more effectively. Without a query-level adaptation method based on query semantics (e.g., plan) [16, 66], tuners cannot compose the best aspects of C1 and C2 together.



(a) Transfer Scenario from (TPC-H → DSB)

(b) Drift Scenario (50% of Parameters Changed for DSB Queries)

Figure 4.1: Tuner Adaptivity Challenges – PostgreSQL runtimes achieved by tuners in transfer and drift scenarios for three configurations: (1) *no re-tune*, (2) *continue* tuning from the best historical configuration for 6h and 12h, and (3) *remix* prior knowledge and then tune for 6h. Each tuner has access to historical artifacts: TPC-H (transfer) and prior DSB (drift).

LLM-based tuners ignore opportunities to augment the prompt with targeted experience to guide the model's suggestion process on a query-by-query basis. For instance, if a query has been tuned before, the tuner could augment the prompt with past attempts (e.g., query hints, indexes) to provide the LLM with further context to generate more effective configurations [24, 64, 67].

To illustrate how these problems hinder DBMS tuners, we ran workloads on PostgreSQL with two scenarios: (1) an experience transfer from TPC-H [107] to DSB [32] and (2) a previously tuned DSB workload undergoing a parameter drift (i.e., 50% of queries have different parameters). We deploy a (1) cost-based tuner [12, 23, 60], (2) ML-based tuner [134], (3) holistic tuner [132], and (4) LLM-based tuner [12, 39]. Each tuner has access to their historical artifacts: TPC-H for transfer scenario and the previous DSB for drift scenario.

As shown in Figure 4.1a and Figure 4.1b, both scenarios benefit from continuing to tune from the best historical configuration compared to **No Re-Tune**. However, **Continue** falls short of what is achievable by **Total Knowledge Remix**. By fully remixing historical tuning knowledge, tuners discover configurations that are **13–74%** better than those found by **Continue**. To achieve this for a range of tuners, we propose leveraging recent advances in LLMs to reason about individual queries and adapt them to different environments.

4.1.4 LLM-based Query Adaptation

Recent research in LLMs has shown their capabilities in optimization [39, 67, 103, 123] and schema understanding [102, 118, 136]. However, providing curated information to the LLM remains unsolved [37, 78, 131]. We provide an exposition into incorporating tuning knowledge into LLMs through four techniques: (1) workload-level prompts, (2) workload-level fine-tune, (3) query fine-tune, and (4) combining enriched query-level prompts with a composition mechanism.

Workload-Level Prompts This technique (**WL-Prompt**) invokes an off-the-shelf LLM with a workload-level prompt that describes the tuning task, the workload, and additional information (e.g., telemetry, workload summary) as context [64]. This technique utilizes the LLM's innate abilities and pre-trained knowledge to generate configurations rather than historical knowledge.

Workload-Level Fine-Tune This technique (**WL-FT**) fine-tunes an LLM by using historical experiences to alter the LLM's weights [47]. Similarly to **WL-Prompt**, this technique then invokes the fine-tuned LLM to obtain candidate configurations. By fine-tuning, this technique improves the LLM's ability to generate more effective configurations for the target workload.

Query Fine-Tune In comparison to **WL-FT**, this technique (**Q-FT**) fine-tunes an LLM on queries. Although this allows the LLM to relate queries to configurations, it requires prompting the LLM with a query prompt. As such, rather than generating workload-level configurations, the LLM generates configurations for each query that are then combined into a holistic configuration [132].

Enriched Query-Level Prompts and Composition This technique (**Enrich Q+Compose**) is agnostic to whether the LLM is fine-tuned or not. On a per-query basis, this technique enriches the query prompt with prior tuning attempts based on the query (e.g., SQL text, plan) and then instructs the LLM to generate configurations with those historical references [24, 64, 131]. The technique then uses a composition mechanism to combine the query-level configurations into a holistic configuration while resolving conflicts (e.g., different knob values, conflicting indexes). In doing so, this technique achieves **Total Knowledge Remix** by reasoning over all available historical knowledge (e.g., past configurations, Internet).

To understand their efficacy, we first obtain historical experience by using a holistic tuner to optimize a DSB workload for 12h. We then evaluate the previous techniques on using this experience to adapt to a drifted workload where 50% of the queries have different parameters. As shown in Figure 4.2, **WL-Prompt** performs the worst as it does not use prior experience. **WL-FT** and **Q-FT** have limited improvement over **WL-Prompt** due to issues around granularity and combining query configurations, respectively. In contrast, **Enrich Q+Compose** finds drastically better configurations by exploiting relevant knowledge on a per-query basis and resolving conflicts with a composition mechanism. **Enrich Q+Compose** forms the basis of our method to assist existing tuners in adapting to environment changes by remixing historical knowledge.

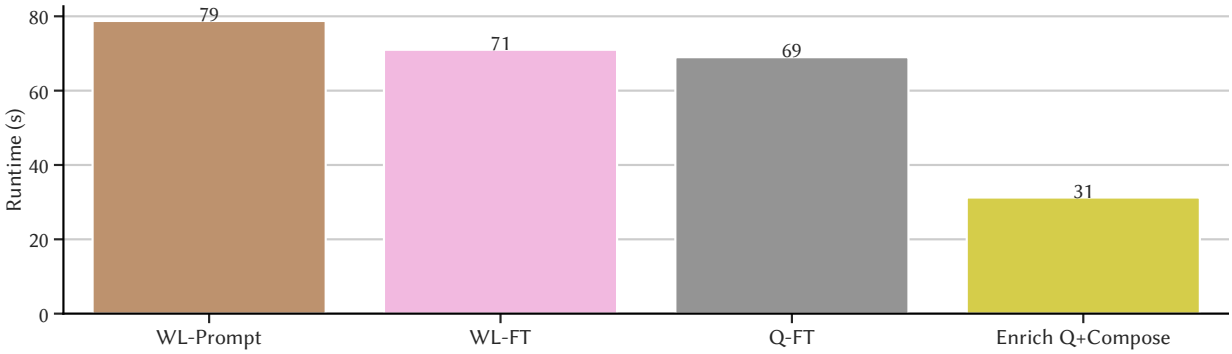


Figure 4.2: LLM-based Query Adaptation – DSB workload runtime achieved by prompting an off-the-shelf LLM (**WL-Prompt**), fine-tuning an LLM with historical knowledge at workload (**WL-FT**) and query (**Q-FT**) granularities, and **Enrich Q+Compose** that combines query prompts enriched with historical attempts and a composition mechanism. All techniques utilize experience generated over 12h by a holistic tuner tuning a DSB workload where 50% of queries have different parameters.

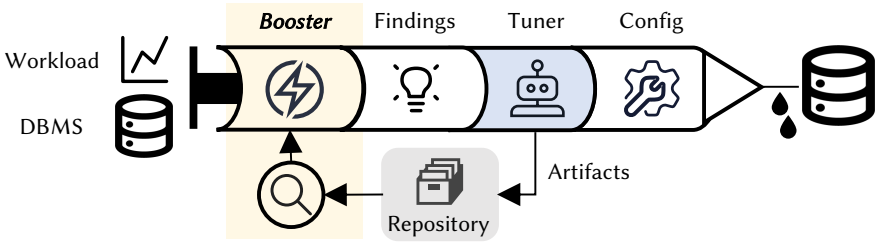


Figure 4.3: Booster Overview – The framework integrates with an existing tuner to improve its adaptivity to environment changes. Booster analyzes the artifact repository and injects its findings (e.g., start configuration) into the tuner. The "injected" tuner then refines the configuration and stores its artifacts into the repository.

4.2 Overview and Architecture

We present how the Booster framework integrates with an existing tuner to improve its ability to adapt to environment changes in Figure 4.3. Based on the workload and DBMS, Booster analyzes the repository of artifacts and injects its findings (e.g., start configuration) into the tuner. The injected tuner then further refines the configuration and updates the repository.

We next describe how Booster utilizes this repository to derive a holistic configuration [132] (i.e., its findings) for injecting into the tuner. As shown in Figure 4.4, Booster's operation is divided into three phases. In Phase I, it monitors the repository for new artifacts generated by the tuner and analyzes them for insights (Section 4.2.1). When the DBMS environment changes, Booster uses these insights to optimize the DBMS. In Phase II, Booster receives the user's target workload and the target DBMS (Section 4.2.2). For each query in the workload, Booster identifies relevant experiences and utilizes them to generate candidate configurations with an LLM. Then in Phase III, Booster combines these query-level candidate configurations with a constrained composition mechanism (Section 4.2.3). Lastly, Booster provides its findings to the tuner to refine. We discuss each in more detail.

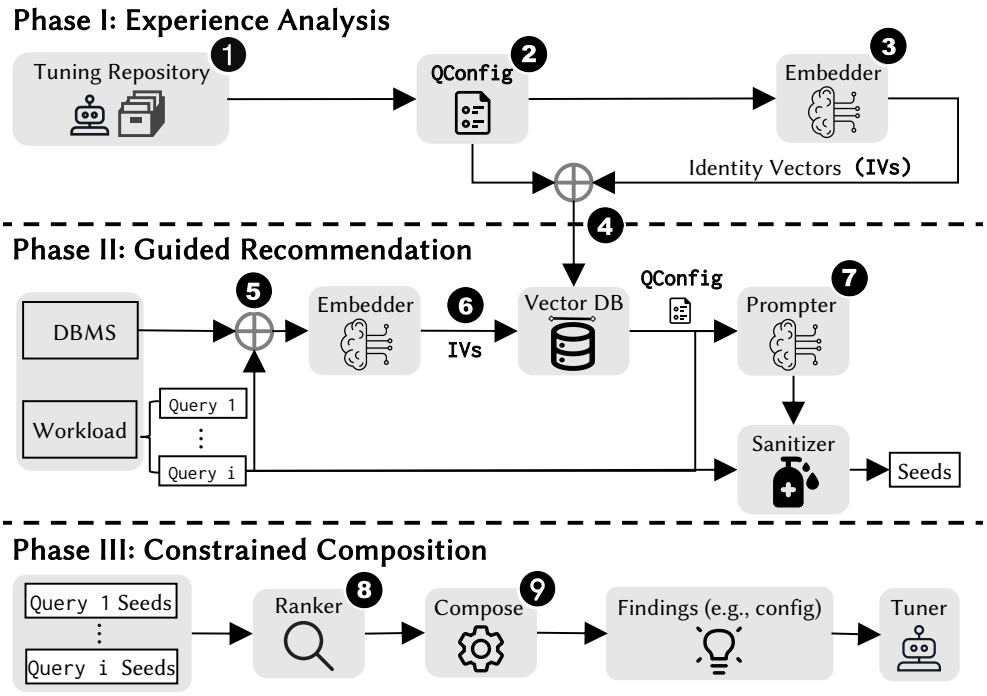


Figure 4.4: Booster Architecture – An overview of the framework's three phases. In Phase I, Booster analyzes historical tuning artifacts. In Phase II, Booster generates candidate configurations (i.e., *seeds*) for each query with an LLM. In Phase III, Booster then composes each query's seeds into a holistic configuration that it then provides to the tuner being assisted.

4.2.1 Phase I: Experience Analysis

In this phase, ❶ Booster monitors a repository of tuning experiences. This repository accumulates artifacts generated by a tuner as it optimizes a deployment. Although these artifacts contain substantial information about the DBMS (e.g., workloads, configurations' efficacy), they are not readily searchable and differ across tuners. To standardize these artifacts into a searchable format, Booster first ❷ parses them into query-configuration (QConfig) objects that represent how a query behaves under a specific configuration. Each QConfig includes but is not limited to the relevant schema, the query's execution plan, the configuration (e.g., knobs, indexes), and links to other related QConfigs for structural organization. For instance, Booster can link QConfigs together in chronological order if the QConfigs refer to the same query.

With these QConfigs, Booster next makes them searchable by leveraging recent advances in LLMs' ability to understand tabular data, schemas, and SQL queries [67, 102, 118, 136]. For each QConfig, ❸ Booster utilizes an embedder (e.g., Voyage 3 Large [4]) to generate fixed-length identity vectors (i.e., embeddings [92]) based on text documents derived from the QConfig. For instance, it can derive them by combining the schema and query plan or by combining the anonymized schema and SQL text. Booster combines these identity vectors with the QConfig and ❹ loads them into a vector database for querying in Phase II (see Section 4.3).

4.2.2 Phase II: Guided Recommendation

At the start of this phase, Booster is now optimizing the DBMS. Booster receives the target workload and a connection to an offline environment for safe exploration [68, 72]. For each query in the workload, ⑤ Booster obtains relevant information from the DBMS (e.g., schema, plan) and uses an embedder to generate identity vectors in the same manner as Phase I. With each query's identity vectors, Booster ⑥ retrieves relevant historical QConfigs from the vector database, combines them with the target query into the prompt, and ⑦ invokes the LLM to recommend configurations based on the QConfigs [24, 64]. As the LLM may suggest invalid configurations (e.g., illegal knob value, invalid index), Booster passes these suggested configurations and the relevant QConfigs through the Sanitizer to obtain configurations for each query that can be deployed. As these configurations target a single query, we refer to them as *query seeds*. We elaborate further in Section 4.3.

4.2.3 Phase III: Constrained Composition

After obtaining the seeds for each query, Booster then combines them into a holistic configuration [132]. However, seed configurations may specify different parameters (e.g., number of parallel workers, indexes) that conflict when combined. To mitigate this, Booster adopts a two-step process to generate holistic configurations from multiple query seeds. ⑧ Booster first ranks all the query seeds and then ⑨ runs a beam search algorithm, a variant of best-first search [25, 88], to identify performant holistic configurations. This algorithm terminates when a terminal condition (e.g., elapsed time) is reached. We elaborate on this further in Section 4.4. Finally, Booster injects its discoveries into the tuner being assisted.

4.3 Experience Analysis & Recommendations

Booster uses QConfigs to guide how its LLM recommends configurations. We now discuss (1) how the framework constructs QConfigs, (2) how it augments the prompt with relevant QConfigs, and (3) how Sanitizer produces seed configurations for each query.

4.3.1 QConfig Construction

During Phase I, Booster mines the repository of artifacts generated by the tuner. Booster analyzes the configurations explored over time (i.e., *trajectory*) to identify interesting configurations (e.g., ones that improve the user's objective function). On a per-query basis, Booster constructs a QConfig from each interesting configuration. As shown in Figure 4.5, this QConfig contains both the DBMS's configuration (e.g., knobs, indexes) and query-specific information (e.g., SQL text, plan). Each QConfig contains a link to the QConfig of the same query that is built from the trajectory's next configuration (i.e., discovered by the tuner chronologically in the future). In Figure 4.5's example, Q1-C0 links to Q1-C2, which then links to Q1-C5.

However, Booster cannot directly use these QConfigs for vector-based similarity search [35]. Instead, Booster must first derive a fixed-length vector representation from each QConfig before it can use distance functions (e.g., cosine distance, Euclidean distance) for similarity search.

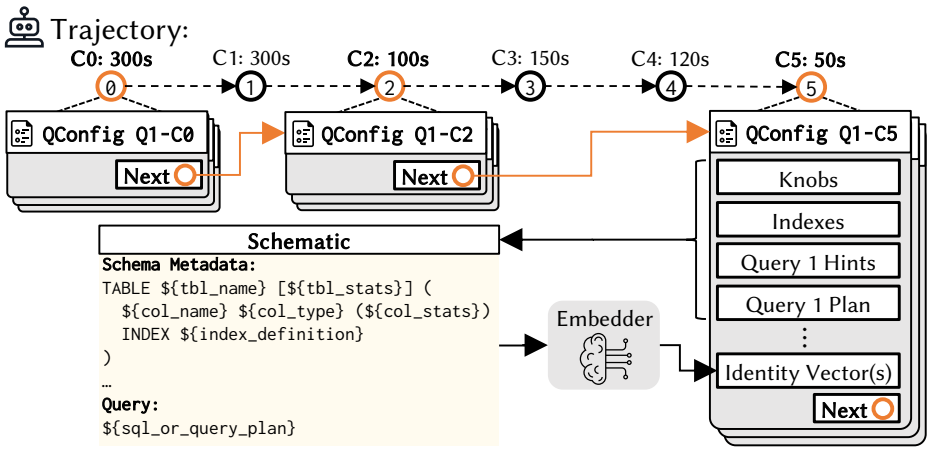


Figure 4.5: QConfig Construction – Booster generates QConfig objects from interesting configurations (e.g., configurations that improved the user's objective function): C0, C2, and C5. As Q1-C5 illustrates, each QConfig contains information (e.g., knobs, plan) about a specific query in a configuration, a link to a downstream configuration (e.g., Q1-C2 to Q1-C5), and multiple identity vectors (i.e., embeddings) obtained by passing different schematics through an embedder.

Recent research has proposed using embedders (e.g., Voyage 3 Large [4]) for natural language to SQL [35] and root cause analysis [84]. These embedders map texts to fixed-length vectors where related texts are located nearby [24]. In this way, Booster obtains a QConfig's identity vectors by constructing representative texts (i.e., *schematics*) and passing them through an embedder.

As shown in Figure 4.5, Booster builds multiple schematics from each QConfig. These schematics capture a query's semantics, ranging from high-level information about what data the query accesses to low-level details of how the DBMS will execute it. Each schematic comprises two parts: (1) schema metadata and (2) the query. The schema metadata describes the referenced relations and columns, utilized indexes, and additional statistics (e.g., number of relation tuples, number of distinct values for each column). The query component describes what the DBMS executes, which can take the form of the SQL query, the query template, or query plan. Booster generates schematics based on all three, along with variants derived by anonymizing table and column names. It then passes each schematic through an embedder to obtain an identity vector (i.e., embedding) and loads the assembled QConfig into a vector database.

4.3.2 QConfig Guided Recommendation

With the vector database from Phase I, Booster in Phase II obtains candidate configurations with an LLM following the process in Figure 4.6. For a given user query, Booster generates schematics and computes identity vectors (IVs) using the same process and embedder as in Phase I. With each query's IVs, Booster retrieves similar QConfigs by minimizing the Euclidean distance to each QConfig's IV of the same schematic type. For example, assume Booster uses the anonymized query plan to build schematic S and obtains IV1. When searching the vector store with IV1, Booster only considers each QConfig's IV that is built with schematic S. This ensures that Booster does not erroneously rank QConfigs based on differences in schematic type.

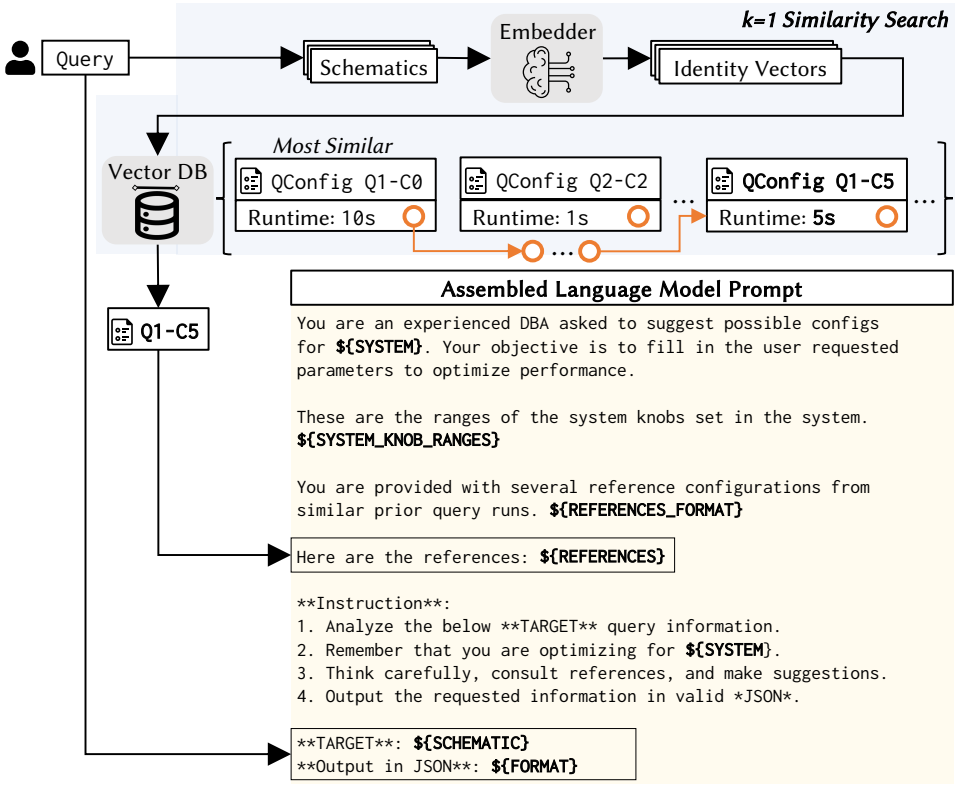


Figure 4.6: Prompt Augmentation with k=1 Relevant QConfig – Based on the query, Booster derives identity vectors in a similar process to Figure 4.5. It then retrieves a ranked list of QConfigs based on similarity (i.e., Euclidean distance), takes the most similar QConfig, and follows its links to obtain the most performant downstream QConfig. Booster then enriches the prompt with the downstream QConfig (QConfigQ1-C5) and prompts the LLM for suggested configurations.

However, the most semantically similar QConfig may not be the best reference. Consider the case where the DBMS is in the stock configuration and Booster is analyzing a previously tuned query. In this case, the most semantically similar historical QConfig′ is derived from the same query in the same stock configuration. Yet, this QConfig′ lacks any guidance (e.g., query knobs to set, indexes to build) to extract. Instead, a more relevant QConfig* with guidance exists downstream of QConfig′ by following the links. Based on this, we make a core change to retrieval: from each retrieved QConfig, Booster follows its links to fetch the most performant downstream QConfig.

Due to LLM context window limitations (i.e., prompt and response length) [52, 70], Booster truncates the references to the top-k QConfig, packages the reference QConfigs into the prompt, and instructs the LLM to reason through those references (Instruction #3 in Figure 4.6) [55, 64] to suggest configurations. Based on empirical trials, Booster sets k to a default value of 2.

4.3.3 Recommendation Sanitization

Prior techniques [39, 47] use the configurations suggested by the LLM "as-is". Booster forces the LLM to output valid JSON to avoid the complexity of extracting relevant configuration snippets.

However, these suggestions may be incomplete, contain invalid indexes and parameters (e.g., a knob from a different DBMS), or request to match parameter values to the references. To correct these errors, Booster's Sanitizer cleans the suggested configurations using each query's QConfig references. Sanitizer first constructs a preliminary list from three sources: (1) the LLM's suggestions, (2) the QConfig references, and (3) additional configurations obtained by filling in any missing parameters in the LLM's suggestions with values from QConfig references.

As this set may occupy a locally suboptimal region, Sanitizer introduces a limited degree of exploration. From each preliminary configuration, Sanitizer derives diverse candidates in two ways: (1) augmenting indexes based on static analysis of the query's predicates and (2) permuting the query knobs (e.g., turn off sorting). For example, turning off sorting in PostgreSQL while preserving other query knobs (e.g., access method hints) tends to different yet performant plans. We provide a sensitivity analysis in Section 4.7.2.

Sanitizer then removes invalid selections (e.g., unknown columns, invalid hints) from each candidate configuration before obtaining its minimal form. For example, Sanitizer uses a "what-if" mechanism [22] to identify indexes used by each candidate and then removes the unused indexes. Another example is query hints that allow multiple possibilities. For instance, PostgreSQL's `NoSeqScan(t)` [1] hint requests the DBMS to use an index scan if possible. Sanitizer modifies these hints based on the query plan to their more specific form (e.g., `IndexScan(t)`). Finally, it preserves all configurations that result in unique query plans as the query's seeds.

4.4 Constrained Composition

Once Booster obtains unique seeds for each query in Phase II (see Section 4.3.2), it combines them into a holistic configuration through a *rank*-and-*compose* process. We discuss each next.

4.4.1 Ranking Seeds

As a query's seeds can have diverse performance outcomes, ranking them is crucial for composing into a holistic configuration. The simplest approach is to use the DBMS's estimated plan cost. However, plan cost is not necessarily correlated with plan quality [15], particularly when query hints are involved that distort the cost. Alternatively, there is extensive literature in learned cost models [43, 73, 75] that build models to predict plan runtime. However, they require substantial representative training data and are shown to have limited efficacy for ranking query plans [42].

Booster instead executes each seed to estimate its quality. Deploying all seeds (e.g., building all indexes) induces significant overhead. Thus, Booster derives alternate indexes that cover the original indexes [100]. To estimate each seed, Booster executes it after replacing the original plan's indexes with alternate indexes that upper bound the original plan's runtime. For example, consider three seeds: (1) Q1 with I1 `t(a,b)`, (2) Q2 with I2 `t(a)` and Q2 does not access `b`, and (3) Q3 with I3 `t(a)` and Q3 does access `b`. For Q1 and Q2, Booster builds I1 and forces both to use it. However, if Booster forces Q3 to use I1, then Q3 may avoid heap fetches with the covering index I1 and execute faster than otherwise possible. To avoid this, Booster builds I3 and forces Q3 to use I3.

Algorithm 1 Beam Search

Input: Query-Configs $QC = \{\ldots, (q_i, [c_{i1}, \ldots, c_{ij}])\}$
Output: Target Configuration
$seeds = \{(q_i, best(c_i)) \forall (q_i, c_i) \in QC\}$ ▷ 1
$cands = \texttt{Merge}(seeds)$ ▷ 2
$best = \texttt{Evaluate}(cands)$ ▷ 3
while budget not exhausted **do**
    $q_{next} = \texttt{Select}(best)$ ▷ 4
    $alt\_q\_seeds = \texttt{Rollout}(q_{next}, best, QC)$ ▷ 5
    $cands = \texttt{Merge}(best, q_{next}, alt\_q\_seeds)$ ▷ 6
    $best = \texttt{Evaluate}(cands)$
end while
Return $best$

Booster executes all seeds in order of increasing plan cost. If the DBMS distorts the cost due to query hints (e.g., turn off sorting when the query requires sorting), Booster computes a hypothetical cost without distortions. To control query execution overhead, Booster adopts a simple per-query timeout strategy. Although Booster could prune seeds with rules or based on clustering analysis of query plans (e.g., common access paths, similar partial join order), there are no established strategies to do so, particularly when queries time out and do not return any execution information (e.g., EXPLAIN ANALYZE). We defer a principled investigation of pruning to future work.

4.4.2 Composition Algorithm

After ranking seeds, Booster composes a holistic configuration. Inspired by relaxation-based physical design [17], Booster greedily constructs an initial configuration and then uses beam search (a variant of best-first search [25, 88]) to refine it. In the best case, combining each query's best seed results in a near-optimal configuration. If conflicts arise (e.g., a query's runtime degrades from its seed's runtime), Booster alters the configuration by targeting those conflicting seeds.

Algorithm 1 shows the beam search algorithm that runs until it exhausts the time budget. ❶ Booster obtains each workload query's best seed, ❷ composes them into holistic candidates (**Merge**), ❸ evaluates them, and selects the best one. From the best candidate, ❹ Booster selects a query seed to refine (**Select**), ❺ rolls out alternate seeds for the selected seed (**Rollout**), and ❻ repeats from ❷ by replacing the selected query seed with alternate ones. Our evaluation in Section 4.7.3 shows that composing for 1.5 hours produces good results. We next discuss these three steps below.

Merge: In this step, Booster composes the seed for each query into a holistic configuration. As these seeds may have incompatible system knobs (e.g., different parallel workers) or physical design structures, Booster must reconcile them. For system knobs, Booster generates configurations based on the minimum, median, and maximum across the seeds. For physical design structures, Booster takes the union and removes identical ones. It then runs these holistic configurations on the DBMS with a cache to eliminate identical plan invocations [68].

Select: In this step, Booster picks the next query to refine. Booster first fixes conflicts (i.e., degraded queries) from merging into a holistic configuration. During this phase, Booster greedily picks a query that performed worse than estimated. Afterwards, Booster greedily picks the query with the highest runtime to refine. Booster tracks selected queries in query-agnostic configurations (i.e., only system knobs and indexes) to avoid re-selecting Q1 if Q2 only undergoes a local change (e.g., query hints) that does not impact Q1.

Rollout: In this step, Booster generates alternate seeds for the selected query. It then merges each alternate seed with the other queries' seeds into holistic configurations and evaluates them. Booster generates these alternate seeds with the following constrained exploration mechanisms (**M1–4**). We analyze these further in Section 4.7.5.

- **M1 Query Knob Permutation**: Similar to in Phase II (see Section 4.3.3), this mechanism permutes the seed's query knobs to generate diverse plans (i.e., seeds). Common variations include deferring to the optimizer [26, 132] or forcing a nested loop join [63, 77].
- **M2 Plan Repair**: This corrects the query seed's plan under the holistic configuration to match its plan in isolation. Booster analyzes both plans and then generates alternate seeds by modifying query hints to restore or eliminate nodes. For example, if the holistic configuration's plan contains a `Sort` node that is not present in isolation, Booster creates an alternate seed with sorting disabled. Booster attempts three repairs: (1) enable optimizer flags corresponding to used nodes, (2) turn off nodes not in the isolated seed's plan, and (3) enable nodes in the isolated seed's plan.
- **M3 Hidden Indexes**: Prior work notes that combining index sets together may impair cost-based query optimizers [17]. This mechanism aims to uncover alternative performant plans. After obtaining the indexes used by the query seed, Booster constructs alternate seeds by selectively omitting those indexes.
- **M4 Ranked Seeds**: Each query has many ranked seeds that may compose with the other queries' seeds into different holistic configurations. To improve the query, Booster only selects ranked seeds with estimated runtimes less than the present query seed's runtime.

Due to the above mechanisms, Booster may generate and evaluate a large number (e.g., $\approx$100) of holistic configurations. To account for this, Booster adopts a 5-step rollout: (1) **Local**, (2) **25% Seed**, (3) **50% Seed**, (4) **75% Seed**, and (5) **100% Seed**. In **Local**, Booster uses **M1,M2** to try query-local alternative seeds that are fast to evaluate with per-query timeouts to bound the runtime. In the other steps, Booster explores seeds from **M3,M4** in slices based on estimated runtime with a workload timeout. Thus, Booster avoids suboptimal seeds if it discovers a performant one in an earlier step.

4.5 Integration

We now discuss how Booster integrates with existing tuners. As Booster is tuner-agnostic, developers can plug it into a tuner through API commands: (1) `Parse`, (2) `Link`, and (3) `Digest`.

`Parse:` As tuners generate artifacts in various formats, this command enables Booster to decipher them and extract their key insights. This process reconciles differences in tunables (e.g., knobs, query hints) between what a tuner supports and what Booster reasons over. For example, some tuners only consider whether sequential scans or index scans are enabled system-wide [111, 134] or on a per-query [12, 77] basis. By contrast, Booster reasons over access methods at a finer table-level granularity for each query.

`Link:` Developers implement how Booster links QConfigs. Recall that the QConfig retrieval process uses these links to retrieve semantically relevant and performant QConfigs (see Section 4.3.2). We propose linking based on the temporal nature of tuning steps. By default, Booster links QConfig objects based on explored trajectories: C1 links to C2 if C2 is reached (i.e., explored) from C1.

`Digest:` Lastly, once Booster completes its composition, Booster passes its findings to the tuner for further refinement. By default, Booster exposes its best holistic configuration to the tuner through `Digest`. However, Booster could expose additional artifacts (e.g., explored configurations) or guide further tuning focus (e.g., target specific queries). We envision that future tuners will support the ingestion of this data. We defer this to future work.

We next discuss how we envision operators will deploy Booster. An operator must provide Booster with an offline environment [72] and API access (or local GPU) to its embedder and prompter LLMs. As Booster and, by extension, downstream tuners are unable to estimate the benefit of further tuning, we rely on the operator to judge when to invoke Booster. Upon starting a new tuning session, we expect that the operator will first invoke Booster, then run any downstream tuners, and finally ensure that Booster analyzes any new artifacts. We defer mechanisms that balance cost and the benefit of further tuning to future work.

4.6 Evaluation

We evaluate Booster's ability to accelerate the convergence of existing tuners when confronted by different drifts and transfer scenarios. We target PostgreSQL v15.1 running on a server with two Intel Xeon Gold 5218R CPUs (20 cores) and a 960 GB Samsung NVMe SSD. We restrict the DBMS to 32 GB of RAM and 20 worker processes. To support Booster's operations in PostgreSQL, we install *HypoPG* [49] for the what-if mechanism [22], *pg_hint_plan* [1] for query tuning, and a custom *HypoExec* extension for re-costing query plans and swapping indexes during execution.

We primarily evaluate with three OLAP workloads. **JOB** [63] (6.7 GB) is a benchmark that stresses the query optimizer with 21 tables and 113 queries. **TPC-H SF10** [107] (14.3 GB) models a business analytics workload with eight tables and 22 queries. **DSB SF10** [32] (22.5 GB) is Microsoft's extension of TPC-DS [106] that introduces additional challenges (e.g., data distributions, join patterns) with 25 tables and 53 queries. We omit four queries (Q18, Q32, Q81, Q92) due to PostgreSQL's query optimizer's limited ability to unnest subqueries [36].

	# QConfigs	# Million Tokens	Embedder Cost (Upper Bound)
DSB (49 Queries)			
Proto-X	1852	17.8	$3.2
P+DTA+AS	3451	38	$6.8
UniTune	185	1.7	$0.3
λ-Tune+AS	1295	13.6	$2.4
TPC-H (22 Queries)			
Proto-X	373	1.6	$0.3
P+DTA+AS	4662	5.9	$1.1
UniTune	94	0.4	$0.1
λ-Tune+AS	592	2.9	$0.5
JOB (113 Queries)			
Proto-X	3325	25.2	$4.5
P+DTA+AS	4628	44.5	$8
UniTune	791	6.1	$1.1
λ-Tune+AS	1732	15.5	$2.8

Table 4.1: Phase I: Experience Analysis Overhead – We present the mean # QConfigs, # tokens, and upper bound the embedder cost. Booster invokes the Voyage 3 Large API at $0.18/million tokens with a rate limit of 3 million tokens/min [5]. Booster caches the API call's inputs and outputs to reduce cost.

4.6.1 Experiment Setup

We deploy four state-of-the-art DBMS tuners:

PGTune+DTA+AutoSteer (P+DTA+AS): This first runs PGTune [60], a heuristics-based knob tuner, followed by Microsoft's Anytime Database Tuning Advisor (DTA) algorithm [23]. We use Hyrise's implementation of DTA [57] with an unlimited tuning budget. After DTA finishes, we run AutoSteer [12] to tune query knobs by greedily toggling and merging boolean knobs.

UniTune: Alibaba's coordinating tuning framework that targets system knobs, indexes, and limited query rewriting through Calcite [134]. We adopt the same settings as their paper, but modify it to run queries serially and to minimize the workload runtime.

Proto-X: CMU's holistic tuner that targets system knobs, indexes, and query options supported by pg_hint_plan [132]. We adopt the same parameters from their paper and extend it to suggest index or bitmap scans and common table expressions (CTEs) inlining.

λ-Tune+AS +AutoSteer (λ-Tune+AS): This first runs Cornell's LLM-based tuner (λ-Tune+AS [39]) that analyzes the workload, formulates a prompt, and generates configurations with GPT-4o [81]. After obtaining configurations, it then runs AutoSteer [12] to tune query knobs.

We next briefly discuss Booster's standard configuration for all experiments.[1] For local LLM inference, Booster uses an RTX3080 GPU with 10GB of RAM, an Intel Xeon W1350 CPU (12 cores) and 32GB of RAM. Booster uses Voyage 3 Large [4] for its embedding model to generate identity vectors (i.e., embeddings) and Llama 3.1-8B-Instruct Q4-K-M [40] to suggest configurations. For consistency, we invoke the LLM with a temperature of 0, a context window of 16384 tokens, and maximum output of 4096 tokens.

[1]Source Code: `https://github.com/17zhangw/booster`.

	# Queries	# Tokens	Runtime	Embedder (Voyage 3 Large) Cost
DSB	49	450k	3.4min	$0.08
TPC-H	22	92k	1min	$0.02
JOB	113	837k	5.6min	$0.15

Table 4.2: Exact Transfer Phase II: Guided Recommendation Embedder Costs – We present the mean embedder (Voyage 3 Large) costs. These costs depend only on the workload and not the tuner.

	# Input Tokens	# Output Tokens	Runtime	Est. Power	Est. OpenRouter Cost
DSB					
Proto-X	261k	39k	9.8min	0.05kWh	$0.005
P+DTA+AS	278k	37k	9.8min	0.05kWh	$0.005
UniTune	270k	36k	9.3min	0.05kWh	$0.005
λ-Tune+AS	275k	40k	10.2min	0.05kWh	$0.01
TPC-H					
Proto-X	68k	14k	3min	0.02kWh	$0.001
P+DTA+AS	71k	12k	3min	0.02kWh	$0.001
UniTune	68k	13k	2.9min	0.02kWh	$0.001
λ-Tune+AS	70k	13k	2.9min	0.02kWh	$0.001
JOB					
Proto-X	537k	80k	20min	0.11kWh	$0.01
P+DTA+AS	510k	88k	20min	0.11kWh	$0.01
UniTune	542k	86k	21min	0.11kWh	$0.01
λ-Tune+AS	518k	96k	22min	0.12kWh	$0.01

Table 4.3: Exact Transfer Phase II: Guided Recommendation Prompter Costs – We present the mean prompter (Llama 3.1-8B-Instruct Q4-K-M) costs across each tuner's trials. We provide estimates assuming a 320W load on our local RTX3080 and estimate cloud inference costs from OpenRouter [3].

In Phase I, we configure Booster to reason over configurations that span Proto-X's options: system knobs, indexes, and query options for each query that cover optimizer switches, access method selection (e.g., sequential, index, bitmap), and whether to materialize or inline CTEs. In Phase II, Booster uses all permutation strategies to generate query seeds (Section 4.3.3). In Phase III, Booster runs its constrained composition for 1.5h (Section 4.4.2).

We populate a tuning repository for each tuner by running four trials with random seeds on the same hardware. At the start of each trial, we initialize PostgreSQL with its default configuration. We then run each tuner for 12h to tune the DBMS. While these tuners run, Booster in Phase I (Section 4.3.1) analyzes their artifacts and generates QConfigs that are available once the tuners finish. We present the overhead of the phase in Table 4.1.

For tuners *continuing from history*, we start each tuner from one of the best configurations in the repository and allow it 12h to optimize the DBMS further. When accelerating a tuner with Booster, each Booster invocation runs Phases II and III with access to all artifacts from that tuner's repository (e.g., four trials of Proto-X). After each tuner's trial, we re-evaluate discovered configurations without timeouts in a warm cache to obtain their actual performance [62, 132].

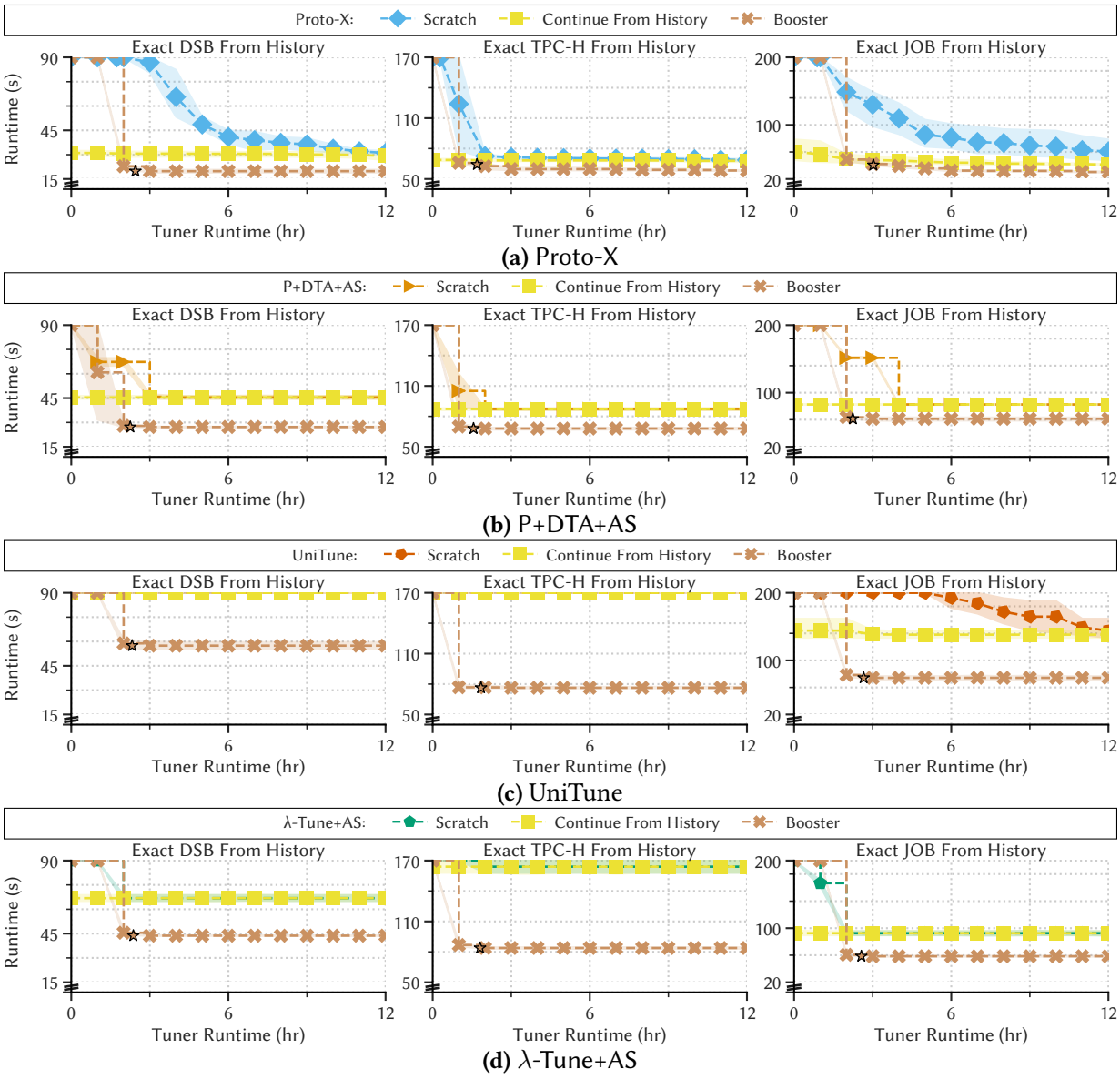


Figure 4.7: Exact Transfer – The DBMS's performance achieved by each framework on DSB, JOB, and TPC-H when tuning from scratch, from the best historical configuration, and accelerated with Booster. We plot the mean performance obtained by four trials of each tuner, with the error band representing the 95% confidence interval. For Booster, the ⋆ illustrates when it finishes composing a holistic configuration.

4.6.2 Exact Transfer

We first evaluate Booster's ability to accelerate tuners when tuning the same historical deployment. We present the embedder and prompter costs incurred by Booster in Tables 4.2 and 4.3. We then compare three tuner variations: (1) tuning from scratch, (2) continuing from the best historical configuration, and (3) accelerated with Booster. We report each tuner's best configuration mean performance, the 95% confidence interval band, and the ⋆ to indicate when Booster completes. We also show each tuner's worst, mean, and best performance in Table 4.4.

Benchmark	Scratch			Continue			Booster		
	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
DSB→DSB									
Proto-X	29s	31s	33s	26s	29s	32s	**19s**	**20s**	**21s**
P+DTA+AS	44s	45s	47s	44s	45s	47s	**27s**	**27s**	**27s**
UniTune	125s	136s	148s	125s	136s	148s	**55s**	**57s**	**60s**
λ-Tune+AS	64s	67s	70s	64s	67s	70s	**44s**	**44s**	**44s**
TPC-H→TPC-H									
Proto-X	68s	69s	71s	66s	68s	69s	**58s**	**58s**	**59s**
P+DTA+AS	86s	87s	88s	86s	87s	88s	**66s**	**68s**	**69s**
UniTune	199s	200s	201s	176s	194s	200s	**76s**	**76s**	**78s**
λ-Tune+AS	156s	167s	181s	156s	167s	181s	**83s**	**84s**	**85s**
JOB→JOB									
Proto-X	47s	61s	88s	40s	41s	42s	**29s**	**30s**	**32s**
P+DTA+AS	82s	83s	83s	82s	83s	83s	**57s**	**61s**	**64s**
UniTune	132s	144s	170s	132s	138s	146s	**72s**	**74s**	**78s**
λ-Tune+AS	89s	93s	97s	89s	93s	97s	**58s**	**58s**	**60s**

Table 4.4: Exact Transfer Performance Spread – The worst, mean, and best result for the tuners' trials in Figure 4.7 on DSB, TPC-H, and JOB when tuning from scratch, from history, and with Booster.

As shown in Figure 4.7, **Continue From History** is generally not an effective strategy. P+DTA+AS and λ-Tune+AS (Figures 4.7b and 4.7d) exhibit no improvement, whereas UniTune (Figure 4.7c) achieves only 3–4% mean improvement on TPC-H and JOB. Table 4.4 indicates that although Proto-X achieves only 1% and 6% improvement on TPC-H and DSB, respectively, its JOB configuration is 33% better.

These results show Booster enables tuners to discover configurations that are better than tuning from scratch or continuing from history. Booster helps to find configurations that have a mean improvement of 16–51% (Proto-X), 22–40% (P+DTA+AS), 49–62% (UniTune), and 34–50% (λ-Tune+AS) over tuning from scratch. Booster achieves this by analyzing artifacts for query-level insights and composing those insights together with constrained exploration. It extends the set of tunables (e.g., query hints) supported by the tuners and also breaks their search algorithms out of local optima.

4.6.3 Parameter Drift

We next evaluate Booster's ability when the workload undergoes a parameter drift, whereby only the parameters of the queries change. We generate a set of DSB queries with a different seed and replace portions of the historical workload: **25%**, **50%**, **75%**, and **100%**. We run four 12h trials for each drift percentage and plot the mean performance along with 95% confidence interval in Figure 4.8.

As shown in Figure 4.8 with the ⋆ marker, Booster consistently outputs a holistic configuration in under 3h. With Booster, all frameworks find configurations with mean improvements of 23–27% (Proto-X), 31% (P+DTA+AS), 42–64% (UniTune), and 24–31% (λ-Tune+AS) over those found by continuing to tune from historical configurations. Booster re-mixes query seeds based on query semantics, rather than the entire workload. For repeated queries, Booster extracts beneficial seeds "as-is". For queries with changed parameters, Booster generalizes from historical seeds with the same query template. Thus, it exploits the observation that queries with the same templates may benefit from similar optimizations. Booster assists in finding these configurations

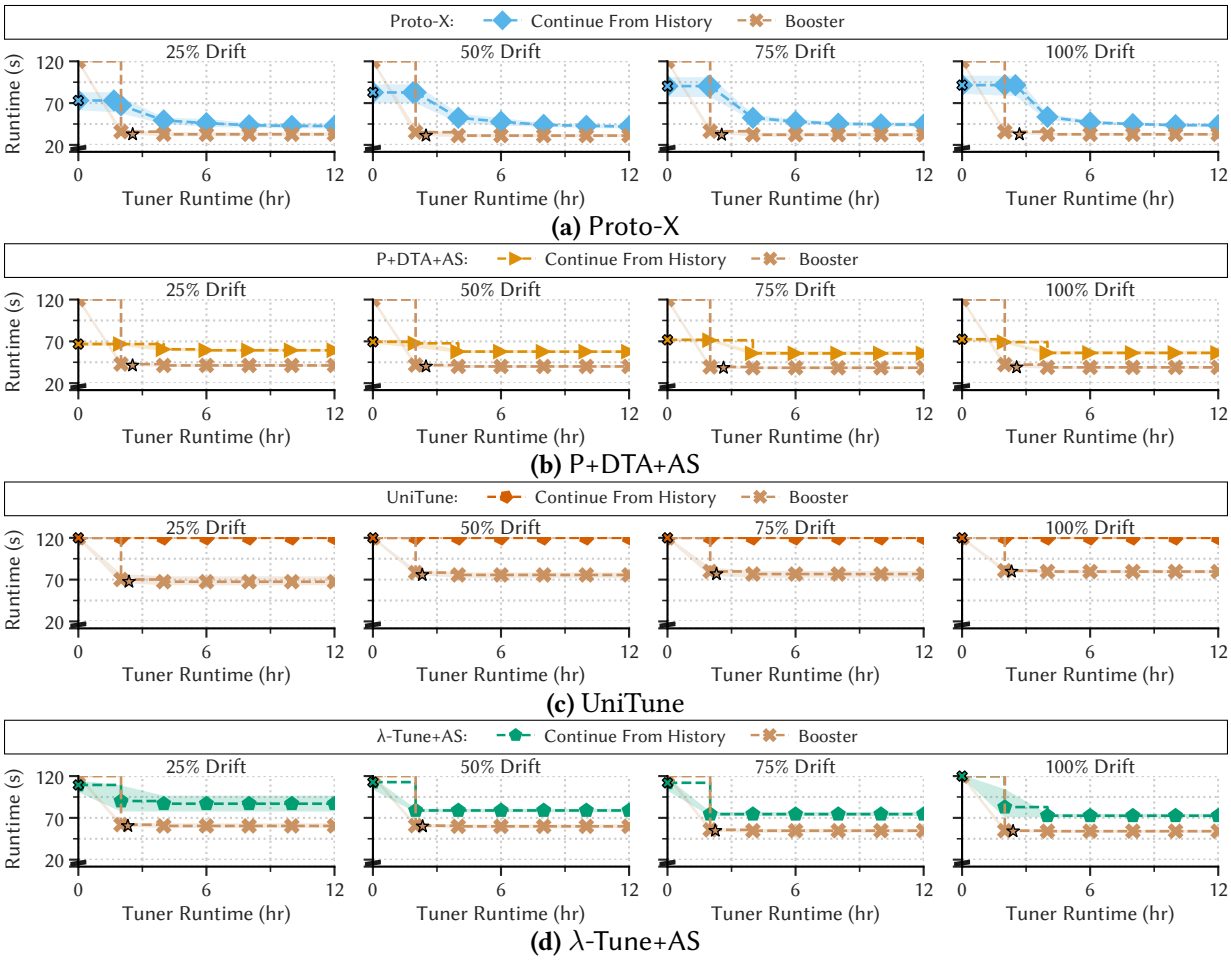


Figure 4.8: Parameter Drift – The DBMS's performance achieved by each framework in response to workload parameter drift. We plot the mean performance with a 95% confidence interval obtained by four trials of each tuner. The **X** represents the starting point when continuing from history. For Booster, the ⋆ illustrates when it finishes composing a holistic configuration.

up to 3.6× (Proto-X), 1.9× (P+DTA+AS), and 3.6× (UniTune) faster. λ-Tune+AS finds configurations 10–30min faster than when using Booster. Booster spends ∼10min longer on LLM inference, as it prompts the LLM on a query rather than workload granularity. Nevertheless, λ-Tune+AS with Booster still finds configurations that are 24–31% better than without.

4.6.4 Template Drift

We now assess how Booster manages workloads that experience template drifts. We generate TPC-DS [106] queries that are not also DSB templates and replace portions of the historical workload by **25%**, **50%**, **75%**, and **100%**. We run four 12h trials per percentage and plot the mean performance along with 95% confidence interval.

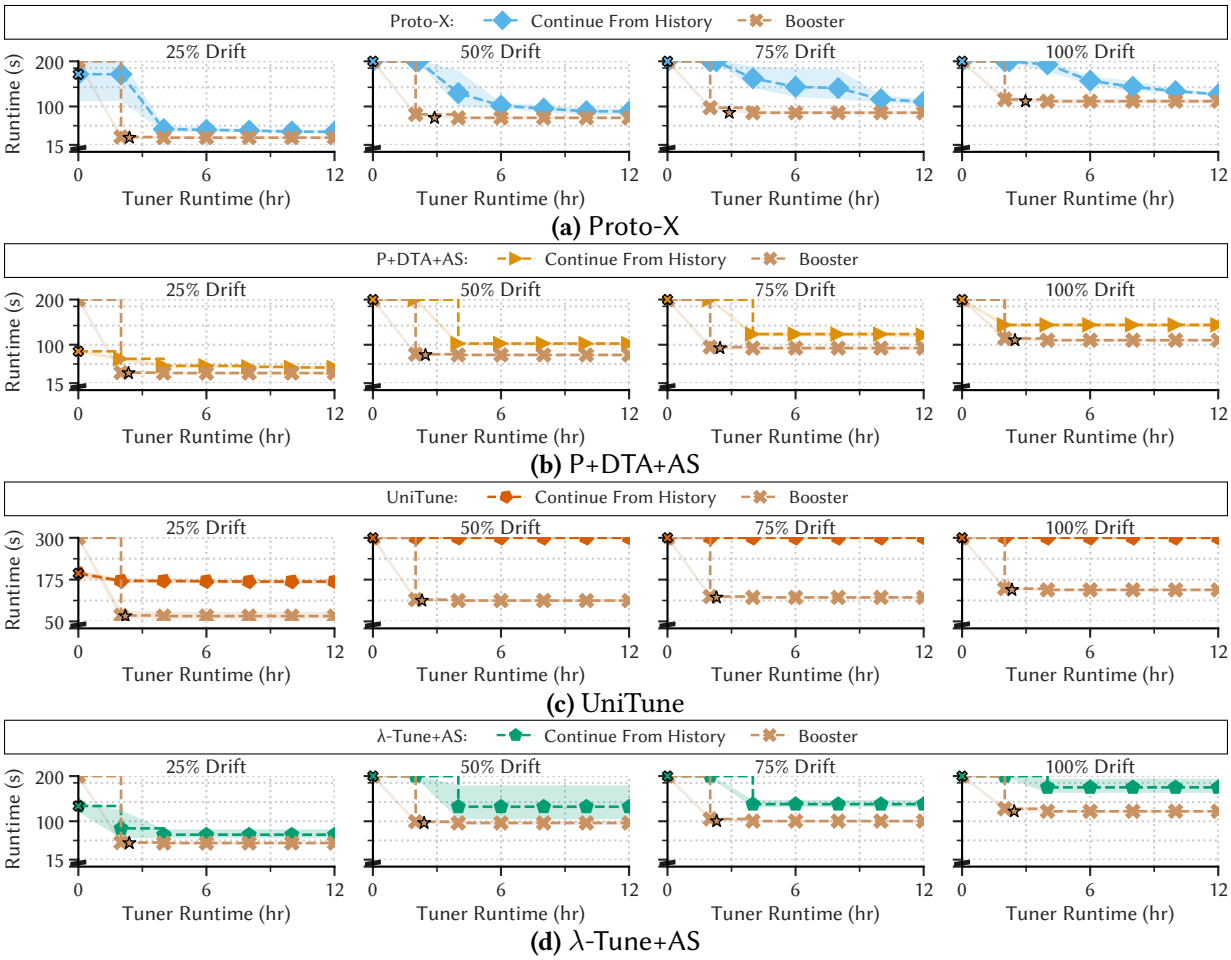


Figure 4.9: Template Drift – The DBMS's performance achieved by each framework in response to workload template drift. We plot the mean performance with a 95% confidence interval obtained by four trials of each tuner. The **X** represents the starting point when continuing from history. For Booster, the ⋆ illustrates when it finishes composing a holistic configuration.

Similar to the previous experiment, Figure 4.9 shows that Booster outputs a holistic configuration in under three hours (⋆ marker). Booster enables the tuners to find configurations with mean improvements of 12–30% (Proto-X), 24% (P+DTA+AS), 52–63% (UniTune), and 26–39% (λ-Tune+AS) over those found by tuning from historical configurations. Booster assists in finding these configurations up to 4.7× (Proto-X), 3.4× (P+DTA+AS), 2.7× (UniTune), and 1.2× (λ-Tune+AS) faster. Booster handles unseen queries through two strategies. It first generalizes from similar historical queries. In cases where the LLM generates a locally suboptimal configuration, Booster uses its constrained exploration mechanism to derive similar, more performant configurations (Section 4.3.3).

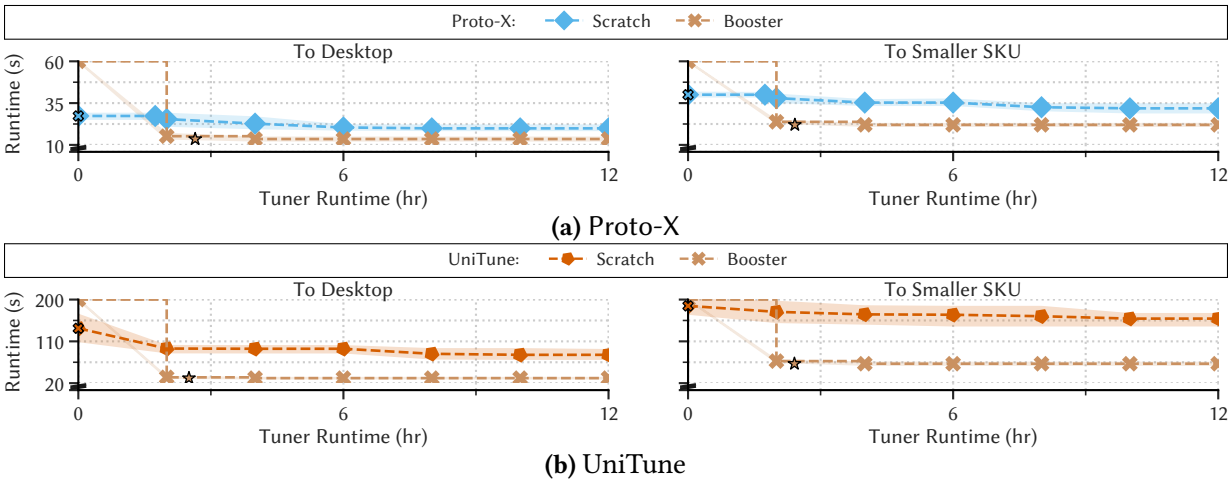


Figure 4.10: Machine Transfer – The DBMS's performance achieved by each framework in response to machine transfer. We plot the mean performance with 95% confidence interval obtained by four trials of each tuner. The **X** represents the starting point when continuing from history. For Booster, the ⋆ illustrates when it finishes composing a holistic configuration.

4.6.5 Machine Transfer

We next evaluate Booster's ability when the environment exhibits a hardware change. We evaluate two scenarios moving from a high-performance server to a *weaker* (Intel Xeon Silver 4114) and *desktop* (Intel Xeon W-1350) machine with the same historical DSB workload. We evaluate the best- and worst-performing tuners from Section 4.6.2 using four 12h trials for each.

As shown in Figure 4.10, all tuners with Booster find configurations with mean improvements of 31–32% (Proto-X) and 61–62% (UniTune) over those found by tuning from historical configurations. Furthermore, Booster enables tuners to find these configurations up to 3.1× (Proto-X) and 2.2× (UniTune) faster. As the target workload is the same as the history, Booster recognizes that each query's best historical seed is a promising starting point. It then leverages this to assist both tuners in finding better configurations more quickly.

4.6.6 Dataset Growth

Another challenge for tuners is when the database grows over time or when switching from development to production instances. To measure this effect, we generate three DSB variations: **SF1→10**, **SF1→20**, and **SF10→20**. Following the process in Section 4.6.1, we first construct the artifact repository from tuning **SF1**. We then evaluate the best- and worst-performing frameworks from Section 4.6.2 using the same workload for four 12h trials per scenario.

Figure 4.11's ⋆ marker indicates that Booster takes longer on **SF20** due to the higher query execution overhead. As shown in Figure 4.11b, Booster assists UniTune in finding configurations with mean improvements of 63% (**SF1→10**), 55% (**SF1→20**), and 53% (**SF10→20**) over those found by tuning from historical configurations. In contrast, in Figure 4.11a, Proto-X shows modest improvements of 6% (**SF1→10**), 7% (**SF1→20**), and 29% (**SF10→20**). To investigate the difference between adapting to **SF20** from **SF1** versus **SF10**, we sweep each SF from 1 to 10 and

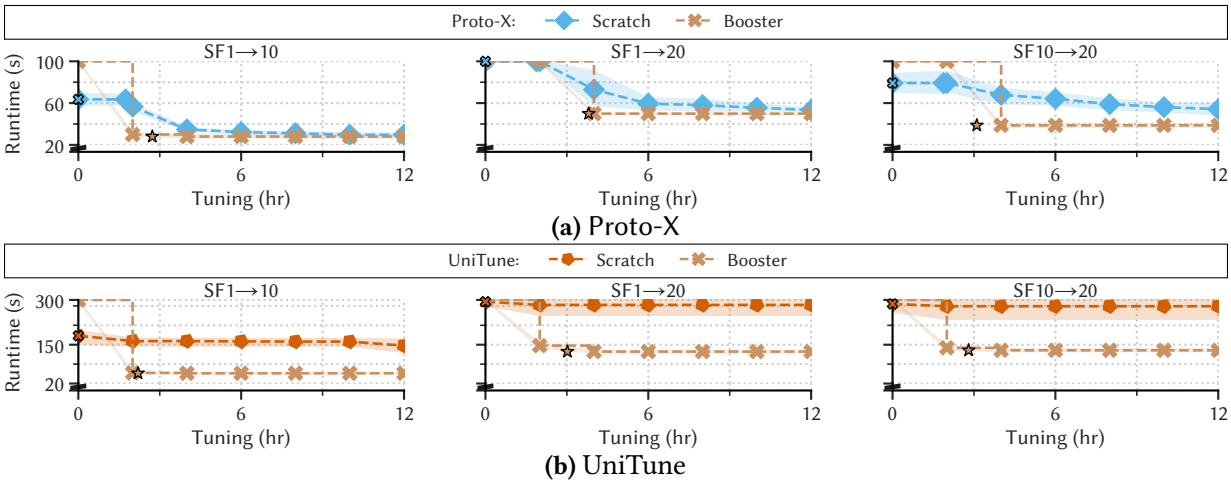


Figure 4.11: Dataset Growth – The DBMS's performance achieved by each framework in response to dataset growth. We plot the mean performance with a 95% confidence interval obtained by four trials of each tuner. The **X** represents the starting point when continuing from history. For Booster, the ⋆ illustrates when it finishes composing a holistic configuration.

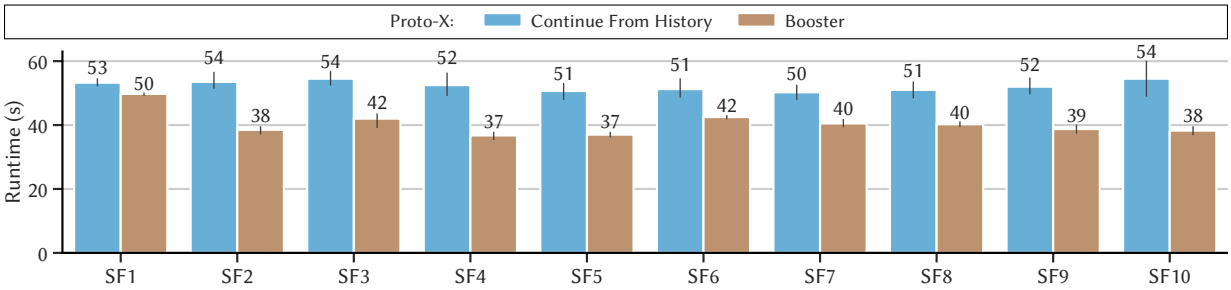


Figure 4.12: Dataset Growth Sensitivity – Mean performance with 95% confidence interval obtained by four trials of Proto-X. Evaluates the dataset growth scenario from different DSB scale factors to SF20.

evaluate with the same methodology as above. We run four 12h trials for each scenario and plot the mean performance along with 95% confidence interval in Figure 4.12. As Figure 4.12 shows, adapting from SF1 stands out, with the others resulting in configurations that are 18–30% better.

Upon further inspection, we find that a single query accounts for ≈7s of the 12s difference between SF1→SF20 and SF2–10 → SF20. The issue begins when Proto-X overfits to SF1 and selects holistic configurations (e.g., turn off hash aggregates, ignore covering indexes) that improve performance by milliseconds without considering stability or generalizability. When adapting to SF20, Booster is trapped by those SF1 configurations in local optima and cannot break out without an expensive combinatorial search. In practice, changes to plans or plan performance happen for various reasons, such as software upgrades or dataset growth. As Booster effectively reuses historical artifacts to accelerate re-tuning, we expect operators to run Booster more frequently to correct those changes. We defer building tuners with an explicit stability or generalizability objective, in addition to a performance objective, for future work.

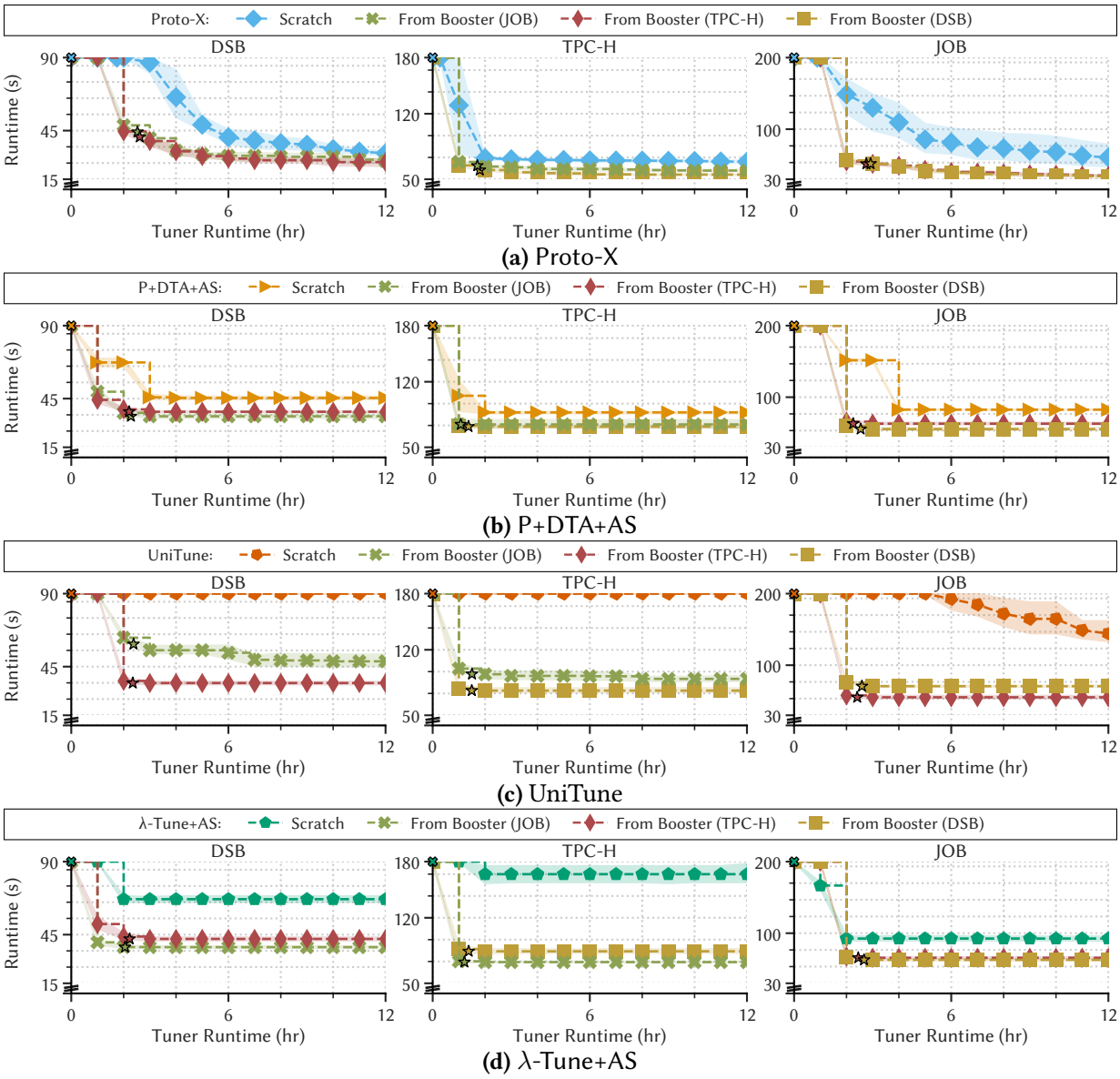


Figure 4.13: Cross-Schema Transfer – The DBMS's performance achieved by each framework on DSB, JOB, and TPC-H when tuning from scratch and accelerated with Booster from other benchmarks' artifacts. We plot the mean performance obtained by four trials of each tuner, with the error band representing the 95% confidence interval. For Booster, the ⋆ illustrates when it finishes composing a holistic configuration.

4.6.7 Cross-Schema Transfer

We next evaluate Booster's ability to accelerate tuners when tuning a new DBMS using artifacts from a different application's database. We run four trials for each tuner variation across all workloads. We again plot the mean performance of each tuner's best configuration, the 95% confidence interval band, and use ⋆ to indicate when Booster completes. We also report the worst, mean, and best performance achieved by any framework's trial in Table 4.5.

Benchmark	Scratch			Booster (DSB)			Booster (TPC-H)			Booster (JOB)		
	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
DSB												
Proto-X	29s	31s	33s	-	-	-	22s	**26s**	28s	25s	**26s**	27s
P+DTA+AS	44s	45s	47s	-	-	-	36s	37s	37s	32s	**34s**	36s
UniTune	125s	136s	148s	-	-	-	34s	**35s**	36s	44s	48s	55s
λ-Tune+AS	64s	67s	70s	-	-	-	41s	42s	44s	36s	**37s**	38s
TPC-H												
Proto-X	68s	69s	71s	54s	**55s**	56s	-	-	-	58s	59s	61s
P+DTA+AS	86s	87s	88s	70s	**72s**	74s	-	-	-	71s	74s	82s
UniTune	199s	200s	201s	74s	**76s**	80s	-	-	-	85s	89s	92s
λ-Tune+AS	156s	167s	181s	82s	84s	88s	-	-	-	72s	**73s**	73s
JOB												
Proto-X	47s	61s	88s	30s	**34s**	37s	34s	35s	35s	-	-	-
P+DTA+AS	82s	83s	83s	54s	**55s**	57s	62s	63s	65s	-	-	-
UniTune	132s	144s	170s	69s	71s	72s	53s	**55s**	58s	-	-	-
λ-Tune+AS	89s	93s	97s	61s	**63s**	64s	64s	66s	67s	-	-	-

Table 4.5: Cross-Schema Transfer Performance Spread – The worst, mean, and best performance achieved by a framework's four trials in Figure 4.13 on DSB, TPC-H, and JOB when tuning from scratch, and with Booster from other benchmarks' repository artifacts.

As indicated in Figure 4.13, tuning from the best historical configuration is identical to tuning from scratch. Due to schema differences, existing tuning frameworks cannot transfer the historical configurations. By contrast, Booster obtains holistic configurations for the target DBMS that are inspired by artifacts, with configurations that leverage `INCLUDE` columns (i.e., covering index) and index knobs (e.g., page fillfactor). Thus, Booster assists tuners in discovering configurations with a mean improvement of 14–44% (Proto-X), 15–33% (P+DTA+AS), 51–74% (UniTune), and 29–56% (λ-Tune+AS) over starting with the best historical configuration. Table 4.5 shows that there is no dominant benchmark (i.e., artifact repository) to initialize Booster with. Instead, Booster can use diverse artifacts to accelerate tuners in finding better configurations.

4.7 Sensitivity Experiments

We next analyze Booster's aspects in more detail. We begin with an ablation on fine-tuning in Section 4.7.1, query knob permutation strategy in Section 4.7.2, the composition phase's search time in Section 4.7.3, the embedders and prompters in Section 4.7.4, the composition phase's rollout policy (Section 4.4.2) in Section 4.7.5, and input data size in Section 4.7.6.

4.7.1 Fine-Tuning

An alternative to Booster's approach of enriching the prompt with historical references is fine-tuning an LLM on (prompt, configuration) pairs [47]. We study whether fine-tuning alone is sufficient. Using the most performant tuner Proto-X's DSB artifacts in the transfer, parameter, and template experiments (Sections 4.6.2 to 4.6.4), we extract each trial's steps into workload-level training data pairs. For each step, we set the output to the trial's best configuration and construct a prompt that contains tuning instructions, a workload summary, DBMS metrics, and query plans truncated to specific context lengths (i.e., **FT-8192**, **FT-16384**). We explore a query variant (**FT-Q**) that replaces the workload summary with the SQL query.

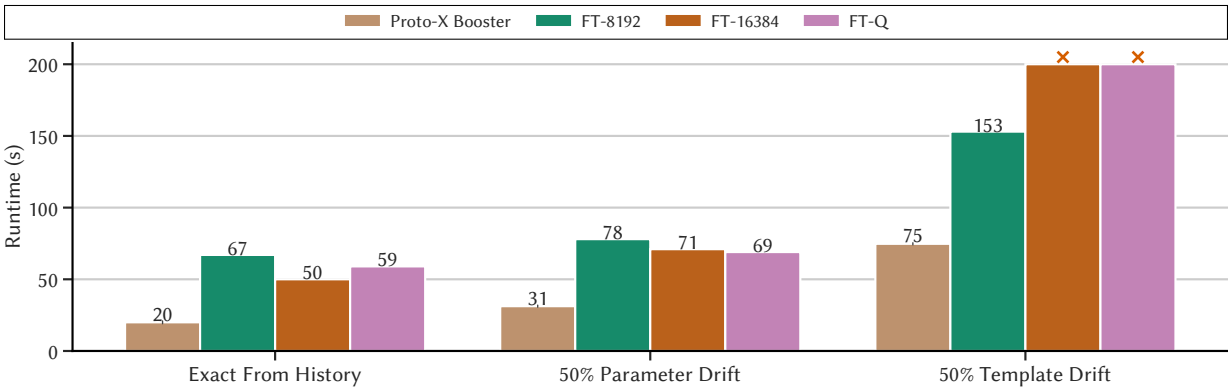


Figure 4.14: Fine-Tuning – The DBMS's performance achieved by each technique across three scenarios based on Proto-X's historical DSB tuning artifacts. We plot the mean performance with a 95% confidence interval obtained from four trials of each technique *without* further refinement.

We fine-tune Booster's LLM (Llama 3.1-8B-Instruct) with LLaMa-Factory [139], FlashAttention-2 [28], and DeepSpeed [91]. For all, we use a learning rate of 2e-5 for 8 epochs [47]. We use 4 RTX3090 for **FT-8192** and **FT-Q** and 2 RTXA6000 for **FT-16384**. During inference for **FT-8192** and **FT-16384**, we sample 8 configurations at a temperature of 0.2 [47] and select the best. For **FT-Q**, we sample three configurations for each query with a temperature of 0.2, select each query's best config, and then combine them together.

We consider three scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, and **50% Template Drift**. We run four trials for each technique *without* further refinement. We plot the mean performance along with a 95% confidence interval in Figure 4.14. Across all scenarios, fine-tuning performs worse. For **FT-8192** and **FT-16384**, truncating to context lengths restricts learning to the prompt's queries. For **FT-Q**, query configs conflict when combined, as seen in **50% Template Drift**.

4.7.2 Query Knob Permutation Strategy

Booster permutes query knobs to generate similar query seeds (Section 4.3.3). We consider five strategies: **None**, **Join Types** (i.e., hash, merge, nested loop), **Access Methods** (i.e., seq-, bitmap-, or index-scan per-table), **Sort** turns off sorting, and **All**. We limit the study to the most performant tuner Proto-X and evaluate three scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, and **50% Template Drift**. We run four trials for each strategy *without* further refinement by Proto-X. We plot the mean performance along with a 95% confidence interval.

As shown in Figure 4.15, the strategy does not matter for **Exact From History**, as Booster exploits the 1:1 mapping from the target workload to historical query seeds. For drifts, **Join Types**, **Access Methods**, and **Sort** strategies enable Booster to improve over **None** by allowing it to locally explore during composition to fix conflicts (i.e., degraded queries) and break out of suboptimal solutions. Booster with the **All** strategy finds the best holistic configuration, as it uses all opportunities to derive diverse plans from each query seed.

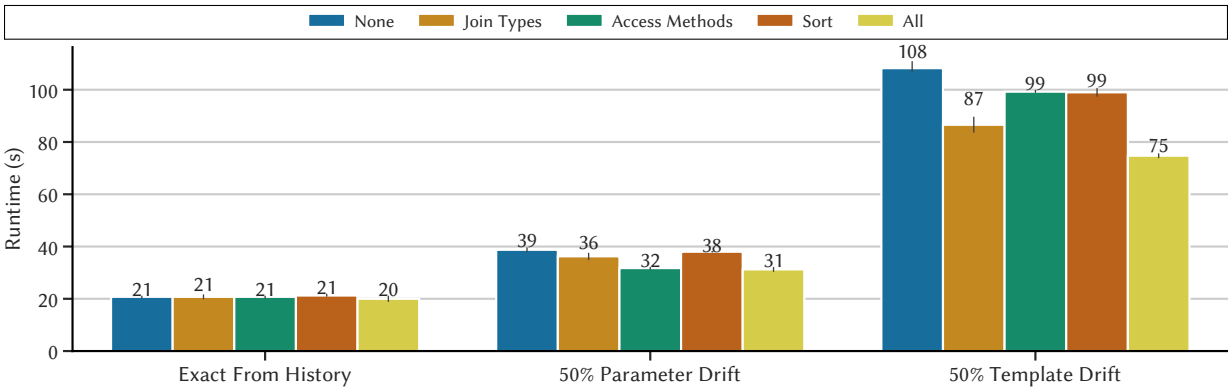


Figure 4.15: Query Knob Permutation Strategy – The DBMS's performance achieved across three scenarios based on Proto-X's historical DSB tuning artifacts when varying Booster's query knob permutation strategy. We plot the mean performance with a 95% confidence interval obtained from four trials of each technique *without* further refinement.

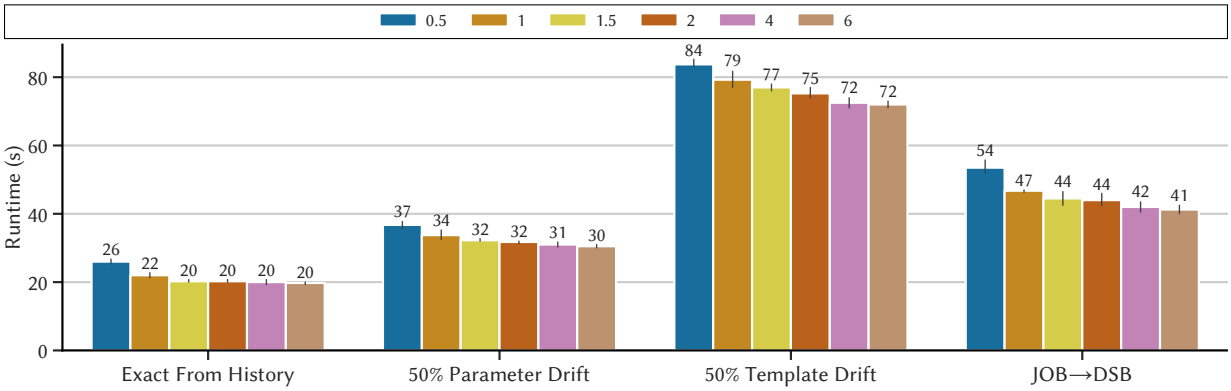


Figure 4.16: Search Time – The DBMS's performance achieved across four scenarios based on Proto-X's historical DSB tuning artifacts when varying Booster's search time. We plot the mean performance with a 95% confidence interval obtained from four trials of each variant *without* further refinement.

4.7.3 Search Time

We next vary the amount of search time allocated to Booster's constrained composition algorithm (Section 4.4.2): 0.5h, 1h, 1.5h, 2h, 4h, and 6h. We limit the study to the most performant tuner Proto-X and evaluate four scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, **50% Template Drift**, and **JOB→DSB**. We run four trials for each variation *without* further refinement by Proto-X. We report the mean performance along with a 95% confidence interval.

Figure 4.16 shows that the trend across all four scenarios aligns with common tuning trends [109, 132]. Providing Booster more time allows it to provide better holistic configurations to the assisted tuner, with diminishing returns. Although **Exact From History** stabilizes after 1.5h, the other scenarios continue to show marginal improvement.

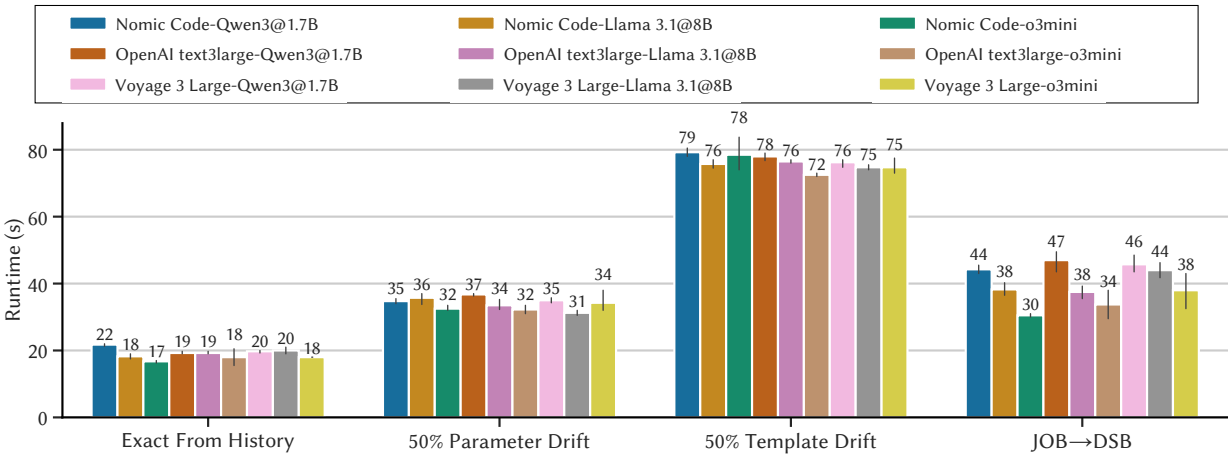


Figure 4.17: Embedder-Prompter Models – The DBMS's performance achieved across four scenarios based on Proto-X's historical DSB tuning artifacts when varying the embedder and prompter LLM. We plot the mean performance with a 95% confidence interval obtained from four trials of each embedder-prompter variant *without* further refinement.

4.7.4 Embedder-Prompter Models

We next investigate the embedder and prompter LLMs used by Booster. We select three embedders based on the Massive Text Embedding Benchmark [79]: Nomic Code [101], text-3-large [82], and Voyage 3 Large [4]. We then select three prompters of different scales: Qwen3-1.7B [119], Llama 3.1-8B-Instruct [40], and o3-mini [83]. We limit the study to the most performant tuner Proto-X and evaluate four scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, **50% Template Drift**, and **JOB→DSB**. We run four trials for each pair *without* further refinement by Proto-X.

Examining Figure 4.17, we first notice that the embedder-prompter matters more when the target workload differs from historical workloads. All pairs are comparable (≈5s) for **Exact** but have larger differences for **JOB→DSB** (≈17s). Second, increasing the complexity of the prompter LLM (i.e., # of parameters) enables Booster to find better configurations. For instance, Booster with o3-mini finds configurations that are 32% better than using Qwen3-1.7B for **JOB→DSB**. However, these improvements come with higher inference costs that are not necessary for some scenarios (**Exact**, **50% Parameter Drift**). We leave the selection of the optimal LLM based on generalizability (e.g., cross-schema) and cost requirements for future work [90].

4.7.5 Rollout Policy

We next study Booster's rollout policy during composition (Section 4.4.2) and consider the following variations: **M1 Query Knob Permutation**, **M2 Plan Repair**, **M3 Hidden Indexes**, **M4 Ranked Seeds**, **M1,M2** (i.e., query-local changes), **M3,M4** (i.e., physical changes), and **All**. We limit the study to the most performant tuner Proto-X and evaluate three scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, and **50% Template Drift**. We run four trials for each variation *without* further refinement by Proto-X. We plot the mean performance along with a 95% confidence interval.

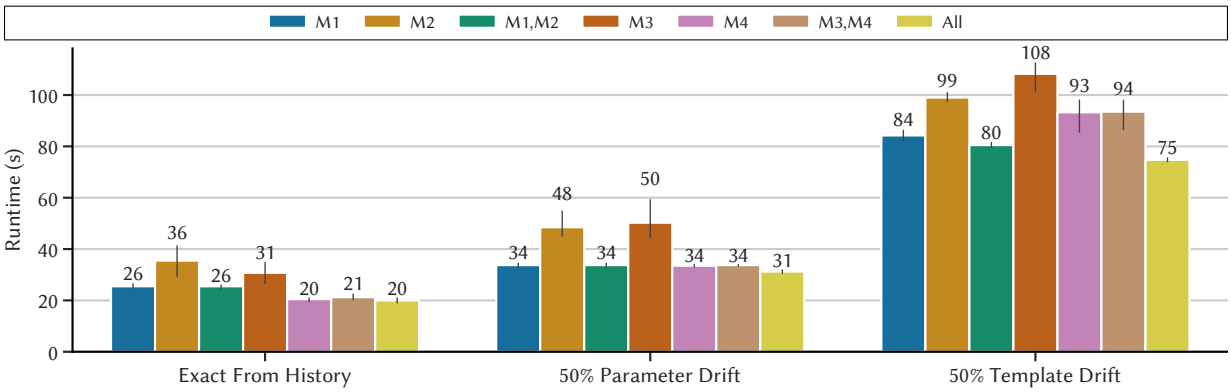
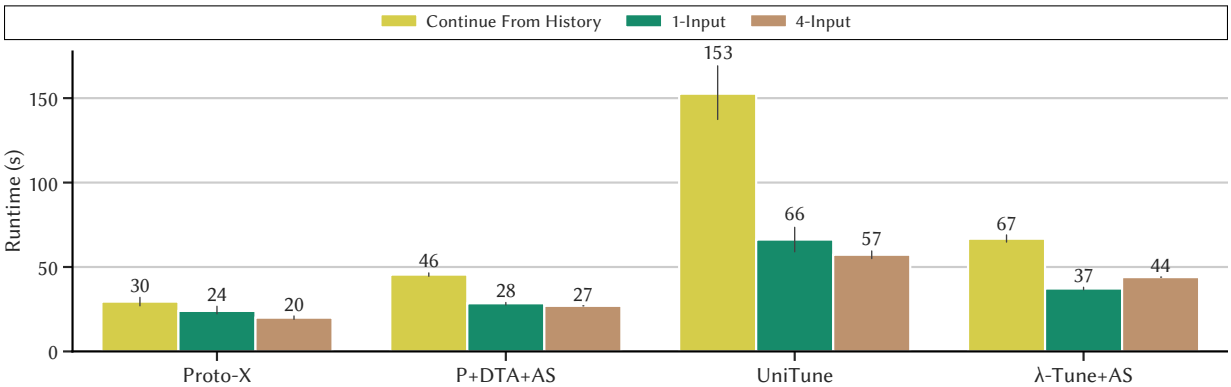


Figure 4.18: Rollout Policy – The DBMS's performance achieved across three scenarios based on Proto-X's historical DSB tuning artifacts when varying Booster's rollout policy during composition. We plot the mean performance with a 95% confidence interval obtained from four trials of each variant *without* further refinement.



(a) Exact From History

Figure 4.19: Input Data – The DBMS's performance achieved by different tuners when transferring the same workload as history. We evaluate whether Booster has access to one artifact or all four artifacts from each tuner. We plot the mean performance with a 95% confidence interval obtained by four trials of each technique *without* further refinement.

As shown in Figure 4.18, we find that **M2** and **M3** alone have limited effectiveness due to their limited exploration. **M1** and **M4** are more effective by providing a greater degree of diverse composition opportunities. We also observe that although **M3,M4** (i.e., physical changes) finds better configurations in **Exact**, **M1,M2** (i.e., query-local changes) outperforms in **50% Template Drift**. Booster with **All** outputs the best configuration by leveraging all opportunities.

4.7.6 Input Data

Recall from Section 4.6.1 that Booster ingests artifacts from all four tuner trials. We next measure the efficacy of Booster when only using one artifact or all four artifacts for each tuner. We run four trials for each variation *without* further refinement. We plot the mean performance along with a 95% confidence interval.

As shown in Figure 4.19, Booster with **1-Input** performs worse than the **4-Input** for Proto-X and UniTune. By contrast, Booster with **1-Input** performs better for λ-Tune+AS. In the case of Proto-X and UniTune, providing Booster with more data allows it to leverage query-level insights across a broader scope and avoid being overly restricted by a single trial's local optima. For λ-Tune+AS, we find that Booster with **1-Input** augments its prompt with reference QConfigs from more diverse query templates (Section 4.3.2) that lead to better outcomes. We defer selecting the number of QConfigs to enrich the prompt based on each query to future work.

4.8 Conclusion

Despite advances in tuners' ability to find performant configurations, existing tuners remain unable to adapt to environment changes (e.g., workload drifts, cross-schema transfers) due to their design. To remedy this, we present the Booster framework that exploits query-level insights from history to assist tuners in adapting. Booster organizes historical artifacts into structured insights, obtains query-level configurations by prompting an LLM with relevant experiences, and composes them into a holistic configuration with beam search. We evaluate Booster's ability to assist state-of-the-art cost-/ML-/LLM-based tuners in adapting to new environments for OLAP workloads on PostgreSQL. Compared to the alternative of continuing to tune from historical configurations, Booster assists tuners in finding configurations that improve DBMS performance up to 74% in up to 4.7× less time.

Chapter 5

Scenario Similarity for Interventions

In Chapters 3 and 4, we discussed holistically optimizing a deployment for a point-in-time snapshot and adapting to application changes. We shift our focus to optimizing the deployment when anomalies occur. To detect these anomalies, the deployment relies on performance monitors to identify issues based on telemetry [145], diagnostic components to reason about possible root causes [84, 144], and tuners to optimize aspects of the DBMS deployment (e.g., the DBMS itself [132] and the proxy [19]).

The fundamental challenge preventing active intervention is the ambiguity and potential errors in reasoning about root causes. For instance, when a p99 latency spike is detected, it may be due to a range of causes (e.g., missing index, traffic surge, query regression) that require different mitigation strategies (e.g., index tuning, traffic control, query tuning). In addition, existing tools often lack situational awareness or deployment context. For instance, an index advisor may not know to avoid suggesting expensive indexes when the system is already under load. Consequently, a human operator (e.g., DBA) bridges this gap by manually evaluating the diagnostic component's analysis, invoking appropriate optimization tools, and deploying the validated mitigations. Although this operator-in-the-loop approach provides greater observability and reliability by allowing the operator to leverage organizational knowledge (e.g., playbooks) and prior experience, it slows response times.

Given this, we introduce the AgenticOperator framework to intervene in a DBMS's operations and maintenance, bridging the gap between monitoring alerts and automated remediation. Upon being triggered by an external performance monitor or anomaly detection system, AgenticOperator reasons about observed telemetry and orchestrates appropriate optimization tools from a "plug-and-play" toolbox to resolve the issues. Similar to how on-call engineers approach debugging, AgenticOperator interleaves information-gathering operations (e.g., retrieving query plans), reasoning about observed telemetry, and optimization tools to iteratively refine the deployment based on the tool's local feedback and internal hypothesis. We present two case studies of AgenticOperator's ability to respond to commonly encountered anomalies (e.g., traffic load, incorrect physical design) [46].

5.1 Background

A self-driving DBMS [86, 87] optimizes itself to execute workloads based on the user's directives. These directives range from simple objectives, such as minimizing p99 latency or reducing storage cost, to more complex multi-objective directives. For instance, minimize p99 latency while prioritizing specific users. However, DBMSs operate in dynamic, changing environments (e.g., traffic spikes, schema evolution, workload drift). As such, the DBMS relies on three core subsystems to ensure the DBMS remains in compliance with the user's directives: observability to collect telemetry (e.g., system metrics, plans, transaction latencies) and identify anomalies, diagnostics to reason about the state of the DBMS in relation to the directives, and an actor (e.g., planning and execution) to optimize and alter the DBMS state. We first discuss the existing taxonomy of DBMS anomalies, the diagnostic techniques for root cause analysis, and the challenges in bridging diagnostics to action.

5.1.1 DBMS Anomalies

Existing work on analyzing DBMS anomalies has focused on OLTP systems, industrial deployments, and analyzing user questions (e.g., internal users, Stack Overflow) [46, 112]. They have converged mainly on the following categories: *traffic control*, *misconfiguration*, *physical design*, and *queries*. We will discuss each of these separately.

Traffic Control: This category covers issues related to traffic flow and query routing. For instance, the DBMS may experience increased traffic (e.g., holiday shopping) that manifests in telemetry as higher write-ahead log traffic, serialization failures, query latencies, or all of the above [46]. Query routing focuses on issues related to read replicas. For instance, the DBMS may experience a spike in read traffic, transition into executing HTAP workloads, or a decrease in parallel analytical queries. In those cases, the DBMS may consider scaling out, scaling in, or re-purposing under-utilized replicas in response to changes in read traffic.

Misconfiguration: To simplify the discussion, we distinguish between system knobs, physical design, and query aspects [132]. This category involves cases where the DBMS's knobs are not configured adequately for the workload. For example, the DBMS's buffer pool may be too small, or the number of parallel workers per query is too high. The category also covers cases where the DBMS has just been initialized and is in the DBMS's stock configuration.

Physical Design: This category covers situations where the DBMS's physical design hinders the workload. For example, the DBMS may be missing an index, an existing index may become suboptimal due to a workload drift, or there may be redundant indexes. In other cases, the replicas' schema may drift from the primary's due to schema migrations or different workload analysis. For example, in systems with declarative, predicate-based partitioning schemes such as PostgreSQL, the replica may miss a specific time-ranged partition, resulting in higher query latency due to accessing a larger default partition.

Queries: This category focuses on query-related issues. For instance, the query may be unoptimized (e.g., SQL rewriting, query plan hints [132]) or its performance may have degraded due to a parameter drift, change in statistics, or data growth. In other cases, a query may be stuck due to deadlocks, waiting for a lock (e.g., DDL lock), or fail to finish within a given time.

5.1.2 Diagnostics Frameworks

Once the observability platform detects an anomaly or determines that the DBMS requires maintenance, it triggers a diagnostics subsystem for root cause analysis. Historically, a paged DBA analyzes the DBMS and then produces a hypothesis and mitigation plan. With advances in ML, recent techniques have focused on heuristic lightweight classifiers and diagnostic agents built on top of tool-calling LLMs.

Heuristic Lightweight Classifiers: These approaches focus on using handcrafted rules or lightweight classifiers from labeled training data to produce a ranked list of possible root causes [84, 124]. For instance, if there is a p99 latency spike and the telemetry indicates a high number of checkpoints, then a classifier may identify "checkpointing" as a root cause. This classifier could take the form of a decision tree or knowledge graph of learned rules [140]. For instance, if the alert is a p99 latency spike, a decision tree of metrics to check can be used to identify the root cause.

Diagnostic Agents: With advances in recent tool-calling large language models (LLMs), recent work has proposed augmenting a state-of-the-art LLM (e.g., GPT-5) with tools for analyzing the DBMS. For example, we can provide tools for retrieving the top-k queries by latency, retrieving the system's telemetry (e.g., CPU load average, memory usage, disk statistics), and executing SQL queries (e.g., schema collection), among others [112].

However, a key challenge associated with LLMs is managing the information (i.e., context) provided to them due to model context limits and long context challenges (i.e., lost-in-the-middle problem [70]). To work around this, different techniques have been proposed, such as selectively curating the provided context or using a multi-agent design [112, 140, 144]. For example, if a query regression is detected, only the query's accessed schema (e.g., tables, columns) is added to the context. Multi-agent designs decompose the diagnosis problem into sub-problems, with each sub-problem handled by a distinct sub-agent with access to a subset of information and tools. For example, assume a monitoring system (e.g., CloudWatch [9]) raises a p99 latency alert. The coordinator then spawns sub-agents that analyze different aspects of the DBMS (e.g., host system usage, checkpointing, and queries). Each sub-agent performs its specialized analysis, compiles its findings, and returns them to the coordinator. The coordinator then synthesizes and outputs a final diagnosis [144].

Although these lightweight classifiers are cheaper and faster to deploy, they are more rigid. By contrast, diagnostic agents are more malleable, as they interact with the tools available to them. In doing so, these diagnostic agents can incorporate evolving knowledge bases, adapt to unseen anomaly scenarios, and propose both a root cause analysis and a mitigation plan.

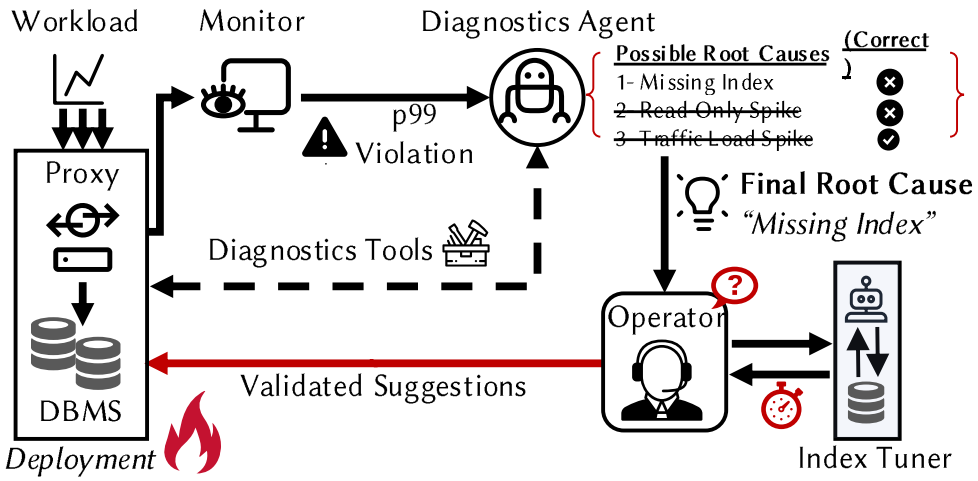


Figure 5.1: Challenges in Diagnostics-Driven Action – The observability system observes telemetry from the deployment and triggers the diagnostics agent with a p99 latency violation alert. The diagnostics agent autonomously invokes diagnostic tools to analyze the DBMS (e.g., get query plan, get most expensive SQL statements) and proposes multiple hypotheses. It then incorrectly concludes that the root cause is due to *missing indexes*, when the actual reason is due to a traffic load spike. Based on this analysis, the operator runs an index tuner and deploys its recommendations, further worsening the deployment.

5.1.3 Challenges in Diagnostics-Driven Action

Despite advances in diagnostic agents, they cannot autonomously translate analysis into remedial actions. Instead, they require an operator to evaluate the analysis, orchestrate the necessary tools, and deploy any discovered mitigations. This dependency stems from two fundamental challenges: (1) hypothesis error and (2) limited tool orchestration. To illustrate these problems, we present a model diagnostics setup in Figure 5.1. The observability system observes the deployment (e.g., primary, replicas, proxy) and triggers the diagnostic agent with discovered anomalies. The diagnostic agent then provides its report to the actor who attempts mitigations.

Hypothesis Error: Due to sampling in frontier LLMs, diagnostic agents can propose noisy and incorrect suggestions. As shown in Figure 5.1, the diagnostic system may produce different and incorrect hypotheses (e.g., missing index, transaction isolation) based on the context provided (e.g., all DBMS metrics and schema). Even though there are expensive workarounds (e.g., majority voting [113]), the diagnostic system is still likely to output an incorrect hypothesis.

Limited Tool Orchestration: Existing optimization tools (e.g., knob tuners [60, 109], index advisors [11, 23]) are designed for human orchestration. There is no prescribed mechanism for exposing tool semantics (e.g., usage rules, limitations) to an agent to orchestrate dynamically. Although an agent could use heuristic rules to select tools, operator-defined rules cannot accommodate the evolving landscape of tools or leverage the contextual understanding ability of LLMs. For example, based on an index advisor's semantics (e.g., tool runtime, whether tool provides estimated index benefits), the LLM can observe that the DBMS is under load and either avoid calling the tool or reject the index recommendations unless they are essential.

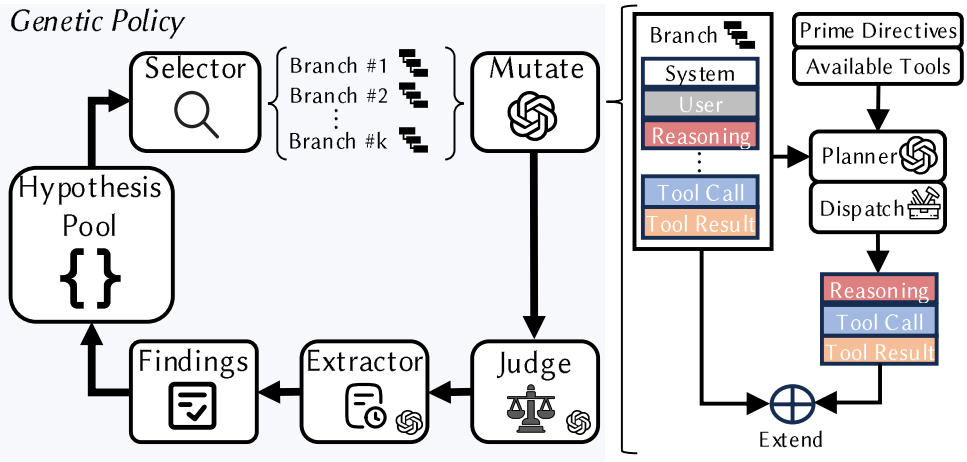


Figure 5.2: Genetic Agentic Hypothesis and Execute – The genetic policy manages the evolution of *branches* that capture a contiguous diagnosis and mitigation pathway. Each iteration selects a subset of branches, mutates them, judges them, and extracts findings. The policy mutates a branch by extending the branch with reasoning, tool call, and tool result blocks from taking a single planning step with an LLM, equipped with tools and the user's prime directives, and executing its chosen tool.

To address the problems presented above, we propose two core pieces. The first is a reasoning agent capable of interleaving research tools (e.g., get a query's EXPLAIN plan, retrieve running queries) and optimization tools (e.g., build index, query tuner) within a single chain. The second is a genetic algorithm to facilitate exploring a range of diverse hypotheses efficiently, while simultaneously learning from all findings.

5.1.4 Genetic Agentic Hypothesis and Execute

Recent LLMs have shown promise in text-to-SQL methods [35] and answering user problems with their deployment [112]. By employing them as a mutation step within genetic algorithms, recent techniques have found state-of-the-art optimization algorithms [27]. We propose viewing the diagnostic and action system analogously, with a genetic algorithm to manage the exploration and synthesis of findings across multiple hypotheses and a reasoning LLM to evolve hypotheses.

We depict this formulation in Figure 5.2. The genetic algorithm's policy manages the complete lifecycle of *hypothesis branches*. Each branch represents a contiguous chain of reasoning [93], comprising the DBMS state, initial hypothesis, and tool invocations. Given an initial population of branches, the policy selects a subset to mutate with a reasoning agent by manipulating the information provided to the agent as context. After providing the agent with both research (e.g., search knowledge base, retrieve actively running queries, retrieve query's explain plan) and optimization tools (e.g., set knob, query tuner), the agent can freely invoke research tools until it emits an optimization tool that ends the mutation step. A framework can then extend and manipulate the branch by appending the tool's results and cross-branch findings before using it again in a future round. In doing so, we enable the agent to both learn across other branches while also facilitating targeted refinement of the current branch, allowing the agentic system to take efficient and proactive action to mitigate ongoing anomalies.

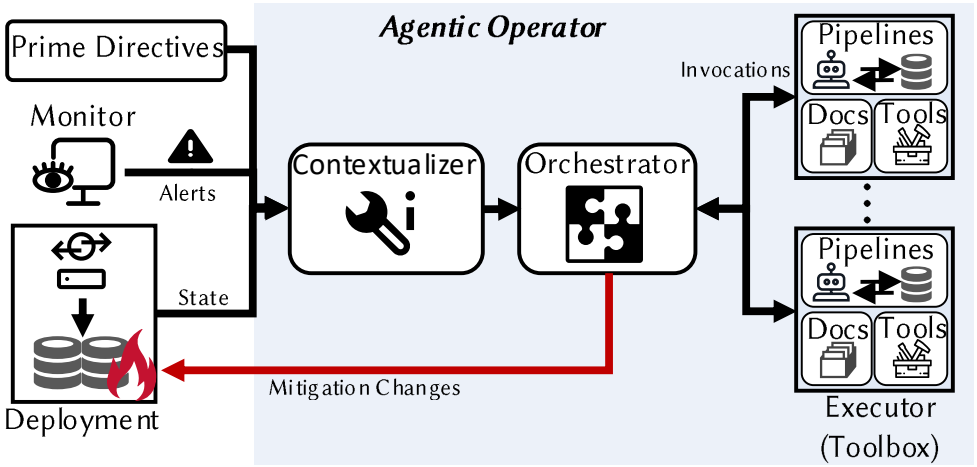


Figure 5.3: Overview – The framework integrates into an existing deployment's monitoring infrastructure. When triggered by an alert, AgenticOperator contextualizes the current deployment state and orchestrates across tools and data sources exposed by executors (e.g., toolboxes). AgenticOperator then applies any discovered mitigations automatically to the deployment.

5.2 Overview

We present the AgenticOperator agentic framework for autonomous DBMS diagnostic and mitigation. We illustrate in Figure 5.3 how AgenticOperator integrates into existing DBMS deployments. We assume the deployment includes a middleware proxy (e.g., pgcat [6]) for traffic control and an anomaly detector (e.g., CloudWatch [9]) capable of generating relevant alert messages (e.g., p99 latency spike, query runtime exceeding a threshold). AgenticOperator uses these alerts and the deployment's telemetry (e.g., active proxy connections, DBMS primary metrics, replica metrics) to construct initial hypotheses and attempt to mitigate the issue in a safe environment [68, 72].

We next describe how AgenticOperator internally processes the input data to mitigate the issue on the production deployment. As shown in Figure 5.4, AgenticOperator's operation is divided into three phases. In Phase I, AgenticOperator assembles its executors. It builds a knowledge base from various data sources (e.g., docs, mailing lists) for research tools and populates its toolbox based on user-provided tool definitions. In Phase II, AgenticOperator uses the alert and deployment telemetry to formulate a set of initial hypothesis branches. In Phase III, AgenticOperator revises its hypothesis branches by incrementally generating each branch's optimization plan, step by step, evaluating all branch steps to obtain per-branch feedback (e.g., tool results, system performance), and propagating summarized findings to all branches. AgenticOperator repeats this until the issue has been mitigated or sufficient time has elapsed. We discuss each in more detail.

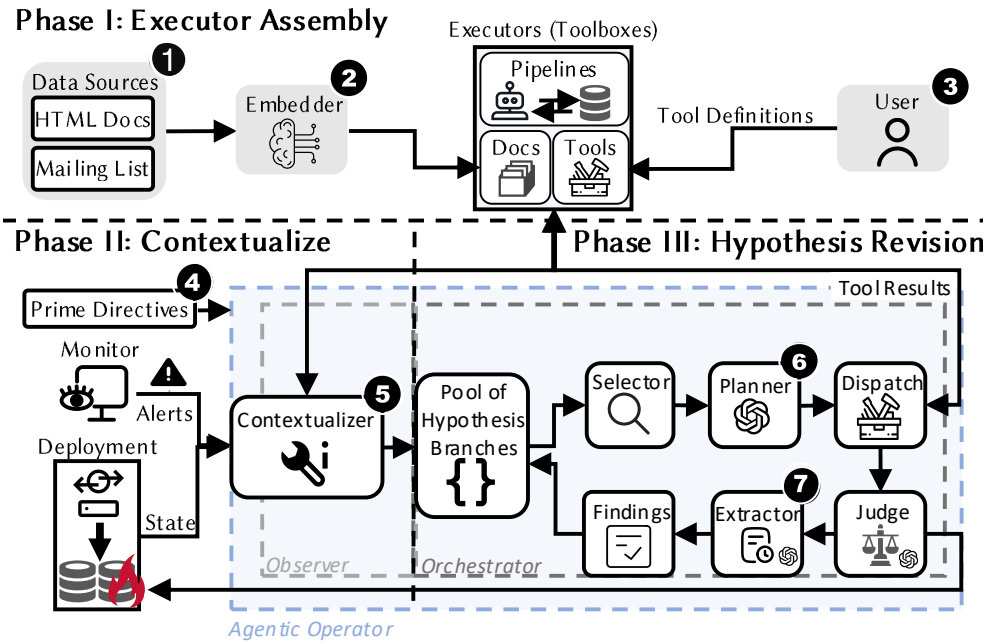


Figure 5.4: AgenticOperator Architecture – An overview of the framework's three phases. In Phase I, AgenticOperator assembles the executors. In Phase II, AgenticOperator observes and contextualizes the deployment. In Phase III, AgenticOperator explores and reasons across diverse hypothesis branches to find and deploy appropriate mitigations.

5.2.1 Phase I: Executor Assembly

In this phase, AgenticOperator assembles its executors (i.e., toolboxes) that it interacts with in Phase II and Phase III. As some executors may expose research-based tools or subagents (e.g., researcher), ❶ AgenticOperator first processes its data sources to provide domain-specific knowledge on-demand. These data sources can be obtained from the DBMS's documentation, mailing lists, forums (e.g., StackOverflow), and internal knowledge bases. ❷ AgenticOperator chunks the documents, obtains embeddings, extracts useful metadata (e.g., DBMS version), and populates a vector store [64]. AgenticOperator exposes both the vector store for semantic similarity and direct access to the underlying files. ❸ AgenticOperator then utilizes the user-provided tool definitions to define all exposed tools, whether they are provided externally or defined internally by AgenticOperator. These tools comprise diagnostic-related (e.g., CPU usage, top SQL statements), research-related (e.g., investigate DBMS behavior), and optimization (e.g., knob tuner, index tuner). We discuss these tool definitions in more detail in Section 5.3.1.

5.2.2 Phase II: Contextualize

AgenticOperator assumes it is connected to another observability and anomaly detection service (e.g., CloudWatch). ❹ AgenticOperator is pre-configured with "prime directives" from users that shape AgenticOperator's behavior. For instance, these directives can instruct AgenticOperator to minimize p99 latency and reduce costs while prioritizing specific client sessions. AgenticOperator incorporates these directives into its reasoning agent's prompt to guide its behavior.

When triggered, ⑤ the Observer receives any alerts generated by the anomaly detection system and the relevant observability telemetry from a recent observation window. This window encapsulates a snapshot that AgenticOperator should focus on analyzing. To construct initial hypotheses, the Observer builds diverse contextual perspectives from the observation window by artificially constraining the available information (e.g., only proxy, only backends). It then passes these different perspectives to its reasoning agent, which generates k initial hypotheses (see Section 5.3). These hypotheses form the initial pool of hypothesis branches for the Orchestrator.

5.2.3 Phase III: Hypothesis Revision

With the initial pool of hypothesis branches, the Orchestrator component evolves these branches through multiple rounds. Orchestrator uses its fitness function (e.g., entire layer, first-N) to select a subset of active hypothesis branches to mutate. ⑥ Orchestrator then uses its planner (e.g., reasoning agent) with its toolbox to reason through the branch so far, gather information as needed, and select an optimization tool. It then dispatches the tool to the relevant Executor and updates the branch with the tool's execution results (e.g., proposed changes, estimated impact).

The Orchestrator then judges all branches and determines whether it wants to apply a specific branch's prospective changes to the live deployment based on the user's prime directives. ⑦ Orchestrator's Summarizer aggregates these branches and their respective tool results into concise findings to guide optimization in future rounds by propagating those concise findings to all active branches in the pool. This cycle continues until some termination condition (e.g., elapsed time, number of rounds, or alerts fixed) is reached. We provide more details about the algorithm in Section 5.4.

5.3 Executor Assembly and Contextualize

For AgenticOperator's reasoning agent to mitigate ongoing issues, it must understand the deployment's state (e.g., traffic, schema). The reasoning agent must also be configured by an operator with a toolbox so that the agent is aware of the possible optimizations it can deploy and can gather more information about the deployment. We will first discuss how Orchestrator contextualizes the deployment, then the toolbox, and finally how the reasoning agent combines them to generate initial hypothesis branches.

5.3.1 Toolbox of Semantic Tools

Beyond the context, AgenticOperator's agent needs to know about the available tools. These tool-calling LLMs form the backbone of AgenticOperator's reasoning agent. Although prior techniques have explicitly incorporated tool-calling guidance into the system prompt [144], this requires potentially rewriting the prompt whenever external tools change. As the landscape of DBMS optimization tools evolves, AgenticOperator directly connects to compliant tool-providing servers via the MCP protocol [13]. AgenticOperator follows a plug-and-play architecture, allowing end-users to customize the provided optimization tools and update them over time.

```
Generate knob configuration recommendations using heuristics.

Usage:
- **Choose this when:** You need fast config-only tuning (< 5 min)...
- Uses well-established heuristics without requiring workload analysis
- Ideal for initial configuration improvements or unknown workload patterns
- Tunes parameters like shared_buffers, work_mem, ...
- Best for systems needing quick wins without deep analysis
- Provides a one-off configuration (only needs to be run once)

Estimated Runtime: Fast (seconds to minutes)

Avoid:
- Using when comprehensive tuning across multiple dimensions is needed (use Proto-X instead)
- Relying on for workload-specific optimizations beyond basic heuristics
```

Listing 5.1: Tool Definition – An example tool definition for a heuristic knob tuner (PGTune [60]). The template includes a high-level statement of the tool, usage guidelines (e.g., when to use and what to use it for), estimated runtime, and when to avoid using the tool.

Under the MCP protocol, AgenticOperator queries an exposed endpoint to retrieve a list of *tool objects*. Each tool object comprises a name, a natural language description of the tool, and a dictionary of arguments that the tool accepts. AgenticOperator's reasoning agent can also call out to specialized sub-agents (i.e., a researcher sub-agent) by presenting it as a "researcher" tool call. These tools are then supplied at inference for the LLM to invoke dynamically. However, for the LLM to correctly orchestrate them based on the provided deployment context, the tool descriptions must expose each tool's semantics (e.g., capabilities, runtime, guidelines).

We provide an example tool definition for a heuristic knob tuner in Listing 5.1. All tools (e.g., sub-agent, optimization, information retrieval) follow the same template. The template provides a high-level statement of the tool, followed by usage guidelines (e.g., when to use and what to use it for). Specifically, for optimization tools, the definition requires an estimated runtime to inform the reasoning agent about the tool's execution behavior and to provide guidelines on when to avoid using the tool. By explicitly containing when a tool should be used or avoided in favor of specific alternatives, the reasoning LLM can select the correct tool based on its complexity for the right situation. Beyond the user's provided tools, AgenticOperator automatically augments the tool list with additional tools to enable the agent to explore the deployment. For example, this includes tools to obtain a query's EXPLAIN plan and a researcher sub-agent.

Researcher Sub-Agent: Similar to other tools in AgenticOperator, the researcher masquerades as another tool with a description following the template in Listing 5.1. The researcher sub-agent resembles existing "deep-research" agents [48], in that it autonomously explores a pre-constructed knowledge base to answer questions. AgenticOperator supports any knowledge base that supports querying the files (e.g., rgrep) and semantic search with filtering on metadata (e.g., DBMS version). By default, AgenticOperator builds a knowledge base that exposes the mailing lists and DBMS docs directly and supports similarity search on chunk embeddings. AgenticOperator is agnostic to how knowledge bases are constructed (e.g., graphs, concepts).

5.3.2 Observer Contextualization

We first discuss how Observer contextualizes the observation window into multiple contexts. Recall that AgenticOperator operates on a trigger-based mechanism. This allows AgenticOperator to side-step the nuanced question of whether the deployment is already optimal. As such, AgenticOperator assumes that the deployment is already encountering an issue.

Before generating contexts, Observer first receives and acquires the necessary telemetry from the given observation window. This includes telemetry from the proxy about connection pools, query routing, proxy wait times, and query or transaction latency distributions on a per-pool basis. On the DBMS side, Observer gathers information (e.g., schema usage statistics, lock contention, backends) from all nodes in the cluster (e.g., primary, replicas). This telemetry is exposed by the DBMS system itself or computed by existing observability tools (e.g., pgmetrics for PostgreSQL). Observer also supports considering historical statistics (e.g., query, mean execution time, number of invocations) from a query store [29] within a sliding window. By incorporating a sliding window, Observer can also look back at past queries, under the assumption that they may repeat [110].

With this gathered information, Observer generates multiple contextual interpretations, each emphasizing different subsystems or perspectives. This allows the reasoning agent to explicitly focus on different aspects of the data, with the ability to collect more information if needed. For instance, we found that when always including system logs about serialization failures, the reasoning agent would periodically pigeon-hole and focus on changing the isolation level rather than other practical changes (e.g., proxy, indexes). Our current implementation defines the following contexts through domain knowledge:

- **Traffic:** This focuses on the workload flowing through the deployment. It includes high-level deployment context (e.g., cluster configuration, version, alerts), host, and proxy information. This context also includes information about the active backends (e.g., backend status, active queries) connected to each cluster node, along with any lock-related information.
- **Buffers / WAL:** Builds upon a high-level deployment context with information targeted towards management of the buffer pool or the write-ahead log. For instance, if DBMS checkpointing is happening too frequently, this context provides the narrowest state descriptor that illustrates the problem.
- **Logical Replication:** Augments high-level deployment context with information about logical replication mechanisms. This context provides a focused representation of whether logical replication is suboptimally configured and may start to lag or impact queries.
- **Schema:** This provides a focus on the database schema: table definitions, index definitions, foreign keys, and usage statistics (e.g., index fetches, live tuples, dead tuples). This context also includes queries that access the schema.
- **Global:** This encompasses the entire observation window's information. We explicitly include a version with and without the system logs to prevent the reasoning LLM from pigeon-holing into the log messages. For instance, an INSERT-heavy workload with many conflicts due to duplicate keys will generate many irrelevant DUPLICATE KEY errors in the log.

```
You are an expert PostgreSQL and proxy administrator specializing in diagnostics and
performance optimization. The system you are maintaining comprises a proxy for traffic
control, the primary DBMS, and replicas. You are allowed to use the optimization tools to re
-configure the proxy and the cluster deployment (e.g., DBMSs, routing) in accordance with
the prime directive. You are *not allowed* to modify the application code, transaction
ordering, application semantics, materialized views, or summary tables. Assume the
application's business logic is correct and not the cause of any issues.

# Your Task
Analyze system context and take optimization actions through iterative diagnosis.

Workflow:
1. Analyze context/results -> Identify actionable issues?
2. Review historical findings if they exist
3. Need more info? -> Use information retrieval tools (...)
4. Ready to act? -> Invoke the declare_hypotheses tool
5. Hypothesis provided? -> Emit AT MOST 1 optimization tool
6. Workflow complete? -> Choose one action: call the reject tool or an optimization tool

# Tool Categories
...
# Calling and Using Optimization Tools
...
# Stopping Condition
...
```

Listing 5.2: Hypothesis Generation System Prompt – The prompt outlines the task, workflow, tool categories, calling and using optimization tools, and stopping condition.

5.3.3 Initial Hypothesis Branch Generation

Armed with each context's description and the toolbox, Observer uses its reasoning agent to generate initial hypothesis branches. We outline the system prompt in Listing 5.2 that also steers the agent in future rounds to generate optimization tools. We provide the LLM with the toolbox following the specific model's conventions. Observer provides guidance on how information gathering tools, meta-tools (e.g., hypothesis tracking), and optimization tools should be called by the LLM, before attaching the context as a user prompt.

During this phase, Observer executes information-gathering tools while rejecting optimization tools. Observer limits the agent to five rounds (e.g., LLM calls). If the agent hits the limit, Observer injects a special prompt to force it to generate hypotheses. This phase terminates in Step #4 of Listing 5.2 when the agent invokes the *declare_hypotheses* meta-tool provided by AgenticOperator to declare k hypotheses. By declaring multiple hypotheses, AgenticOperator avoids the noisy problem of asking the LLM for a single hypothesis while also obtaining variety in analysis. Each hypothesis is a concrete, testable conjecture about the anomaly's cause, often with clear hints on which optimization tool to invoke. With those k hypotheses, Orchestrator then generates k independent hypothesis branches. By viewing each prompt as a series of smaller chunks appended one after another [121], Observer clones the branch and replaces the *declare_hypotheses* tool call with a call to *declare_hypothesis* using each hypothesis.

Algorithm 2 Orchestrator's Evolutionary Algorithm

Input: Initial Population P
Input: Output Channel to Deployment C
while budget not exhausted **do**
    $T = \texttt{Mutate}(\texttt{Select}(P))$ ▷ `Step #1:` Mutation
    $R = \texttt{Execute}(T)$ ▷ `Step #2:` Execution
    $B_{findings} = \texttt{Summarize}(R \cup \{h \in P \mid h.active()\})$ ▷ `Step #3:` Revision
    $P = \{h + B_{findings} \forall h \in P\}$
    $C \leftarrow \texttt{Judge}(P)$ ▷ `Step #4:` Judge
end while

5.4 Hypothesis Revision

Once Observer has generated an initial population of hypothesis branches, AgenticOperator passes control to Orchestrator. We present Orchestrator's algorithm in Algorithm 2. Orchestrator uses an evolutionary algorithm to explore different hypothesis branches across multiple rounds. At a high level, Orchestrator maintains a population P, selects and mutates a subset, executes them for feedback, and then revises the population for the next step. When Orchestrator has reached its budget, it selects the best hypothesis to return as the mitigation plan. We discuss each of these four steps in more detail.

5.4.1 Step #1: Mutation

At the start of each round, Orchestrator selects a subset of active (i.e., not yet rejected) hypothesis branches to mutate. Analogous to the explore-exploit tradeoff, Orchestrator can select hypothesis branches to repeatedly *exploit* or test unexplored branches to *explore*. By default, Orchestrator exhausts a single context's branches at a time (i.e., depth-first), starting from the **Global** context with system logs (see Section 5.3.2).

For the first mutation step, Orchestrator injects additional guidance to focus the LLM on the declared hypothesis rather than any hypotheses in its immutable reasoning traces. Orchestrator executes up to five information-gathering iterations, after which Orchestrator injects a special user message to force a response. The mutation step terminates once the LLM has generated an optimization tool call (e.g., invokes an optimization tool and the call's arguments) or generates a reject tool call to abandon the branch.

5.4.2 Step #2: Execution

After mutating the branches and obtaining a set of optimization tool calls, Orchestrator dispatches the tool calls to their corresponding Executor along with connection details to an isolated replica. This allows the Executor to isolate optimization tool invocations from each other and from the production deployment [68, 72]. Executor executes these tools and returns structured outputs that include errors, recommendations (e.g., knob changes), and performance metrics.

Orchestrator adopts a synchronous execution model. It waits for a dispatched tool call to complete on its Executor before dispatching the next tool call. To eliminate redundant tool invocations, Orchestrator caches non-stochastic tool execution results. For stochastic

tools (e.g., simulated annealing), Orchestrator defers caching to the tool itself (e.g., pre-loading historical evaluations). Prior to dispatching each tool, Orchestrator restores the replica to the configuration obtained by following the hypothesis branch (i.e., restoring to the initial deployment and deploying each optimization tool's changes). Executor then executes the tool and obtains any relevant performance metrics. Executor finishes by building a *tool-use block* from a standardized format that includes errors, deployment changes, and deployment impacts. An alternative approach could stream the tool's outputs (e.g., stdout, stderr) into the branch. However, most of that information will be irrelevant, thus resulting in substantial context growth and rot. By providing only specific deployment changes (e.g., index builds) and how they impacted the deployment, Executor ensures the branch receives the most salient information.

5.4.3 Step #3: Revision

After all tool invocations are complete in the current round, the Summarizer constructs historical summary analysis blocks. The Summarizer builds these summary analysis blocks from both terminated and updated (i.e., branches in the current round with a new tool-use block) branches.

We instruct the Summarizer to emit succinct historical summary analysis blocks to steer other hypothesis branches. To do this, Summarizer presents the previous historical summary analysis, hypotheses, tool call chains, and their impacts. Following its system prompt, Summarizer's LLM identifies viable hypotheses, useful tool-call parameters, and high-level insights. To prevent overloading the LLM with irrelevant information, Summarizer omits information-gathering tools, focusing the historical analysis block solely on hypotheses and specific optimization tool calls to address the anomalies. To facilitate knowledge re-use, Summarizer appends the latest historical analysis block to all active hypothesis branches before the next round, allowing for immediate cross-pollination of information (e.g., index effectiveness). For instance, if a hypothesis branch infers that a traffic spike is the root cause and learns that offloading reads is an incorrect mitigation, another branch that infers the same root cause can decide to try different mitigations (e.g., tuning pool sizes).

5.4.4 Step #4: Judge

After all tool invocations are complete in the current round, Orchestrator judges the population of hypothesis branches to identify whether any of the branches should have their changes applied to the deployment. Due to the need to balance competing user objectives, Orchestrator breaks branches into individual points after each optimization tool invocation. It then computes the Pareto frontier of these points, whereby each point is guaranteed to perform better in at least one measurable performance metric (e.g., throughput, p99 latency, memory). Orchestrator passes these points, along with the user's prime directives, to a judge LLM (e.g., majority voting) to select the point to deploy. When Orchestrator operates on a read replica, the control plane can initiate a live failover from the current primary to the newly optimized read replica.

5.5 Evaluation

We evaluate AgenticOperator's ability to respond to anomalies in a live DBMS deployment. We target PostgreSQL v17 deployed with TPC-E [105], a benchmark modeling a brokerage firm with 20,000 customers, a ∼250 GB database, and 10 concurrent clients. We partition the 20,000 customers across the 10 clients. In our experiments, we designate one *high-priority* client (VIP-) and nine *normal-priority* clients (Normal-). We deploy a middleware proxy (pgcat) between the clients and the primary DBMS, with the VIP-Client connecting through a dedicated high-priority pool (VIP-Pool) and the Normal-Clients connecting through a shared pool (Normal-Pool).

We run on a cluster of four homogeneous servers, with two Intel Xeon Gold 5218R CPUs (20 cores) and a 960 GB Samsung NVMe SSD. We drive the TPC-E benchmark from one server, one server hosts the proxy, and one is the designated primary instance. As we provide a tool for offloading read traffic to a read replica, we keep a provisioned and synced read replica ready that the operator may offload traffic to. We first describe the agents' setup (Section 5.5.1), present two end-to-end scenarios (Sections 5.5.2 and 5.5.3), and then discuss sensitivity studies (Section 5.6).

5.5.1 Experiment Agents' Setup

We compare AgenticOperator against a baseline ReAct agent [122] that observes the context and calls tools in a loop until termination. We provide ReAct with the same Global Context with system logs as AgenticOperator, with the relevant p99 latency alerts injected into the initial observation window. Based on the initial context observation, the ReAct agent iteratively reasons about the next action, invokes a single tool, receives the tool's output, and repeats until some termination condition is met (e.g., calls a termination tool).

In terms of configuration, we provide both AgenticOperator and ReAct with a researcher sub-agent, meta-tools (e.g., *declare_hypotheses*, *revise_hypothesis*, *reject*), diagnostic tools (e.g., *get_full_sql_from_statement*, *get_full_sql_from_log*, *get_full_sql_from_backend*, *get_additional_backends*, *get_additional_statements*, *get_deployed_query_hints*, *get_explain_plan*), knob tuning tools (e.g., *apply_knob*, *pgtune*, *bayesian_knob_tuner*), index tuning tools (e.g., *apply_index_change*, *dexter*, *dta*, *identify_redundant_indexes*), proxy tuning tools (e.g., *tune_pool_sizes*, *apply_pool_sizes*, *offload_reads*, *recentralize_reads*), query hint tuning tools (e.g., *apply_query_hint*, *autosteer*), maintenance tools (e.g., *vacuum_analyze*, *create_statistics*, *pg_repack*), and holistic optimization tools (e.g., *protox*). without additional prompt guidance. We use GPT-5-mini with low reasoning effort. We provide both with the same prime directive: "optimize p99 latency across all users, prioritizing VIP-Client when active, and reduce deployment resources". To evaluate each tool's impacts and provide feedback to the agent, we sample 1min workload observations on a replica to obtain each step's performance metrics.

Specifically for AgenticOperator, we configure it to generate three hypotheses per context and use a depth-first selector (Section 5.4.1) for exploration from the same Global Context with system logs as ReAct. That is, Orchestrator exhausts all hypotheses from the same context before moving on to the next. This depth-first strategy allows AgenticOperator to thoroughly explore related hypotheses (e.g., different pool tuning configurations) before switching to an entirely different diagnostic perspective (e.g., schema-focused analysis).

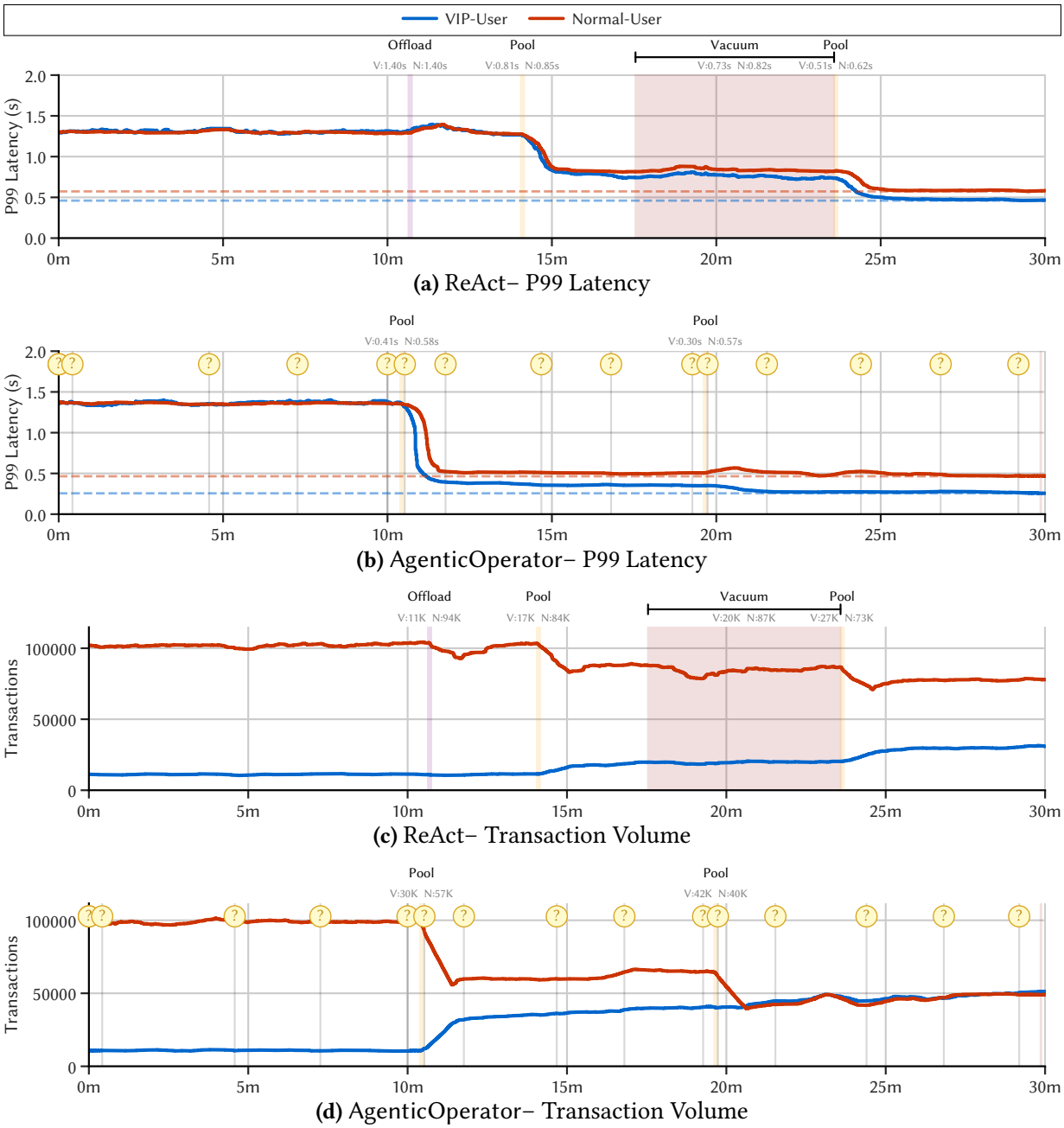


Figure 5.5: Traffic Surge Scenario – P99 latency and transaction volume for the VIP-User and Normal-User over 30 minutes. Each annotation indicates the tool invoked and the resulting p99 latency (V: VIP-User, N: Normal-User). For AgenticOperator, each circled marker denotes an internal thinking step.

5.5.2 Traffic Surge

We first evaluate a scenario where the deployment is overloaded due to a traffic surge. In this scenario, the 10 clients collectively generate enough transactional traffic to saturate and overload the primary DBMS, causing elevated p99 latencies. The root cause is contention at the proxy and DBMS connection level, not a missing index or misconfigured knob.

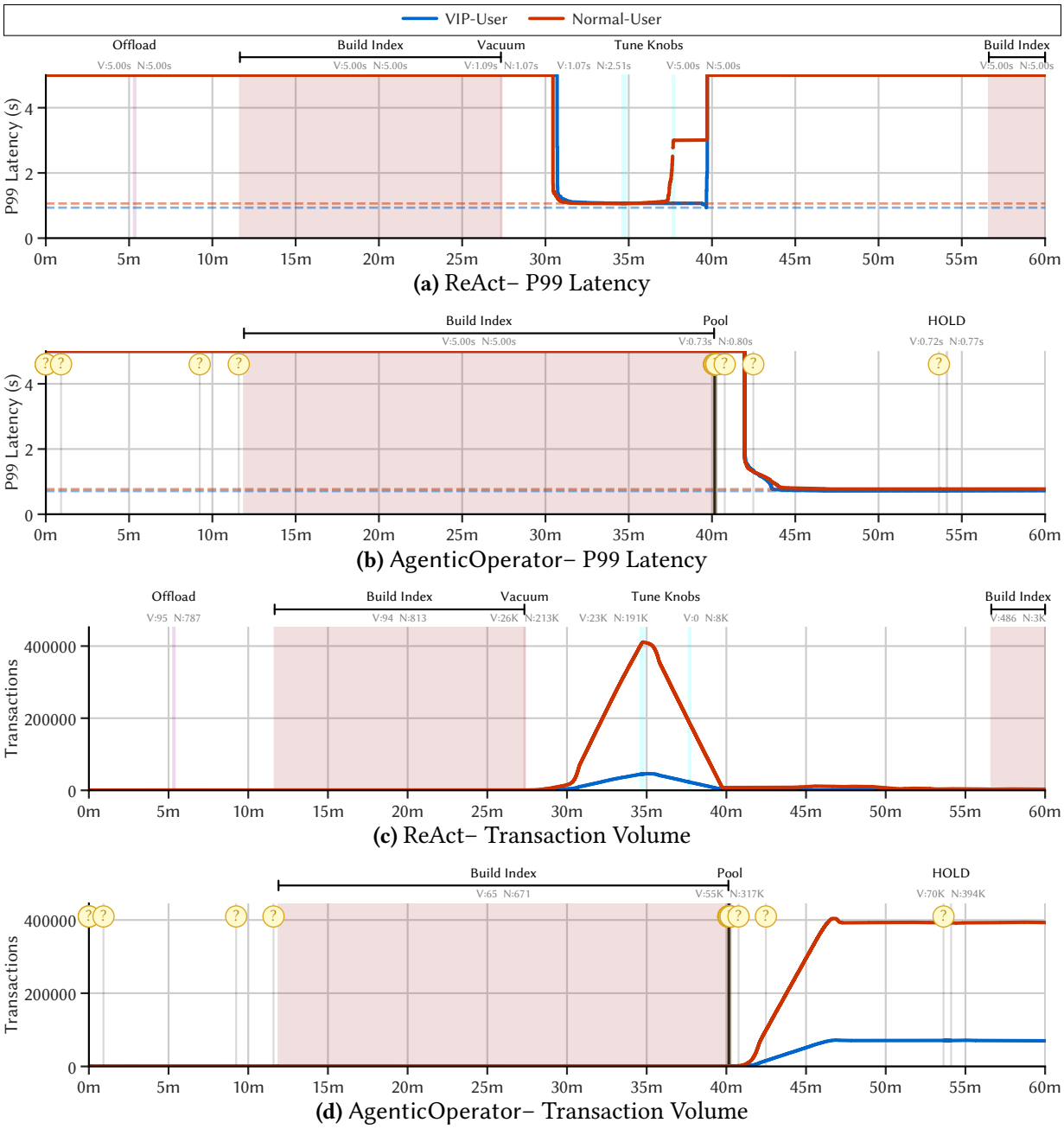
ReAct: As shown in Figure 5.5a, the initial observation window shows elevated p99 latencies (∼1.35s for both user classes). The ReAct agent's first action is to offload read queries to the replica at ∼11m to limited effect (p99 latency remains at ∼1.40s). ReAct then adjusts pool sizes at ∼14m, which yields the first tangible improvement, reducing p99 latency to 0.81s (VIP) and 0.85s (Normal). It then decides to vacuum a table between ∼20–25m and follows up with another pool adjustment. By the 30m mark, ReAct reaches a final p99 of 0.51s (VIP) and 0.62s (Normal) by correctly addressing connection-level contention.

Although the ReAct agent does address the alert and reduces the p99 latency, it does so in a suboptimal way. It took an early offload read traffic action with limited benefit, as the proxy is unable to route mixed read-write transactions to the replica. However, offloading costs the deployment an additional replica's worth of compute and storage that remains largely underutilized, and ReAct fails to decommission the replica. Without the ability to judge the efficacy of a recommendation or tuning decision, the ReAct agent is forced to continue streaming out decisions as it makes them to the live deployment even though they may be tangential or have limited benefit (e.g., vacuuming a table).

AgenticOperator: In Figure 5.5b, AgenticOperator's internal reasoning steps (circled markers) show a fundamentally different approach. Rather than immediately invoking optimization tools, AgenticOperator spends the first ∼10m analyzing the deployment state across multiple hypothesis branches. During this period, Orchestrator generates hypotheses from different contextual perspectives and uses information-gathering tools (e.g., retrieving active backends) to refine its understanding. Orchestrator then explores through its hypothesis branches, only outputting deployment changes to the live deployment when it has a high confidence in a branch's mitigations relative to the prime directive.

In roughly the same time it takes ReAct, AgenticOperator outputs a more effective initial deployment change. AgenticOperator directly targets the connection-level contention that is the actual root cause by tuning pool sizes at ∼11m. For the respective clients, this reduces the p99 latency to 0.41s (VIP) and 0.58s (Normal). AgenticOperator then makes a second pool adjustment at ∼20m, further reducing pool allocations for Normal-Clients to give VIP-Clients better p99 latencies. After 30m, AgenticOperator achieves a final p99 of 0.30s (VIP) and 0.57s (Normal) with limited additional resource usage.

Furthermore, Figure 5.5d shows that AgenticOperator reaches a more balanced transaction distribution without provisioning additional replicas in comparison to ReAct in Figure 5.5c. As AgenticOperator adjusts pool sizes, the VIP-User's transaction volume grows from ∼10K to ∼50K while the Normal-User decreases from ∼100K to ∼50K by the end, reflecting the priority-aware allocation that the prime directive requests. In addition, AgenticOperator achieves a 43% lower VIP-Client p99 latency (0.30s versus 0.51s) and lower overall p99 across the system compared to ReAct. The key difference is that AgenticOperator's hypothesis-driven approach spends time reasoning through diverse potential mitigations before streaming high-confidence changes to the live deployment and allowing branches to learn from each other's successful mitigations. This allows AgenticOperator to not only avoid wasteful actions like offloading reads to a replica or vacuuming a table, but also prioritize continuing to pursue more effective mitigations (e.g., tuning pool sizes again).



(a) ReAct– P99 Latency

(b) AgenticOperator– P99 Latency

(c) ReAct– Transaction Volume

(d) AgenticOperator– Transaction Volume

Figure 5.6: Missing Index Scenario – P99 latency and transaction volume for the VIP-User and Normal-User over 60 minutes. Each annotation indicates the tool invoked and the resulting p99 latency (V: VIP-User, N: Normal-User). For AgenticOperator, each circled marker denotes an internal thinking step.

5.5.3 Missing Index

We next evaluate a scenario where the deployment suffers from a missing index that causes elevated query latencies. Unlike the traffic surge scenario, where the root cause is connection-level contention, this scenario requires the agent to identify and build the correct index.

ReAct: As shown in Figure 5.6a, the initial p99 latencies are at the 5s timeout threshold for both user classes. Similar to the traffic surge scenario, the ReAct agent first attempts to offload reads to the replica, with limited effect due to both the missing index and the proxy's inability to route mixed read-write transactions to the replica. ReAct then correctly identifies that an index is needed and begins building one at ∼12m and completes around ∼30m. Once deployed, the index dramatically reduces p99 latency to 1.09s (VIP) and 1.07s (Normal). However, rather than consolidating and stabilizing the deployment at this improvement, ReAct aggressively invokes a knob tuner at ∼35m that causes a plan regression. ReAct then attempts to compensate by reverting a subset of the knobs and building another index at ∼57m. However, those changes are ineffective, resulting in the deployment remaining in a degraded state at the 60m mark.

This trajectory illustrates a fundamental limitation of ReAct's sequential, greedy strategy. Without the ability to reason about multiple branching hypotheses and the ability to ultimately select between them, ReAct is forced to continue optimizing down its current trajectory with limited ability to course correct by either rolling back entire steps, forking new branches, or restarting from scratch.

AgenticOperator: In contrast, Figure 5.6b shows that AgenticOperator takes a more deliberate approach. During its initial reasoning steps (circled markers), AgenticOperator's hypothesis branches converge on the need for a covering index. The online covering index build starts at ∼12m and completes at ∼40m, taking longer than ReAct's simpler index due to the additional included columns. Once the index is deployed, AgenticOperator modifies pool sizes to target tail latency. The p99 latency improves shortly after to 0.73s (VIP) and 0.80s (Normal) and stabilizes at ∼0.72s (VIP) and ∼0.77s (Normal).

Notably, at ∼55m, AgenticOperator issues a HOLD command on the live deployment even though internal reasoning and refinement continues in the background. The HOLD reflects AgenticOperator's judge (Section 5.4.4) determining that the current deployment state satisfies the prime directive without needing to deploy any additional changes (e.g., offload reads). This HOLD behavior allows AgenticOperator to selectively deploy changes to the live deployment only when the changes are aligned with the prime directive and are effective. As shown in Figure 5.6d, transaction volume ramps up sharply after the index build completes and pool sizes are adjusted, reaching ∼70K (VIP) and ∼394K (Normal) transactions by the end of the scenario. Unlike ReAct, which processes almost no transactions due to repeated regressions, AgenticOperator sustains high throughput once the index is deployed.

Reliability: We next examine the reliability of both approaches across three independent runs of the missing index scenario, measuring the same metrics from different starting conditions. Table 5.1 shows the task-level reliability for each agent across the three runs.

Under ReAct, its trajectories use a range of optimization tool invocations with some degree of variance. For instance, as shown in Table 5.1, all trajectories offload reads to the replica but one run does not tune pool sizes. In the best case, ReAct achieves a p99 of ∼1s by avoiding knobs. In the worst case, it triggers the same query regression, resulting in timeouts. This variance is inherent to ReAct's sequential, greedy strategy. Without the ability to synthesize findings and feedback across different hypotheses, each run will output different optimization sequences.

Optimization Tool	ReAct	Operator
	p99: (1s, ???), Cost: $0.04/$0.004	p99: [0.7s, 1s], Cost: $0.09/$0.006
Create Statistics	N/A	2/3
Index	3/3	3/3
Knobs	2/3	N/A
Offload	3/3	N/A
Query Hints	2/3	N/A
Tune Pool Sizes	2/3	3/3

Table 5.1: Reliability across three independent runs (1 hour each). ReAct exhibits high variance in trajectories and optimization tool invocations. AgenticOperator demonstrates more consistent trajectories, despite higher LLM token costs.

In contrast, AgenticOperator incurs higher LLM costs across the three runs ($0.09/$0.006 vs. $0.04/$0.004 in/out), primarily due to input tokens from maintaining and extending multiple hypothesis branches simultaneously. However, its trajectories are substantially more stable: all three runs converge to p99 latencies between 0.7–1.0s with no additional replicas provisioned with generally the same optimization tool invocations. This stability arises from a core property of the evolutionary algorithm. The cross-hypothesis summarization (Section 5.4.3) propagates findings (e.g., "this specific index improved latency, try it first rather than offloading reads") to all active branches, preventing them from repeating mistakes and hinting at possible mitigations.

5.6 Sensitivity Experiments

The end-to-end scenarios in Sections 5.5.2 and 5.5.3 demonstrate that AgenticOperator mitigates the deployment better than ReAct. As the efficacy of AgenticOperator is highly dependent on the quality of its hypotheses, we next systematically analyze different aspects of AgenticOperator to understand how they impact the generated hypotheses. For all sensitivity experiments, we use GPT-5-mini with the system-wide context with system logs (i.e., comprehensive deployment context) unless specified otherwise. Each scenario in the sensitivity experiments is built with a single intended root cause, covering traffic spikes, idle replicas, missing indexes, extra indexes, mistuned knobs, and read traffic spikes. We evaluate the impact of contextual perspectives (Section 5.6.1), the reasoning model (Section 5.6.2), tool-hypothesis alignment (Section 5.6.3), tool name sensitivity (Section 5.6.4), read replica availability (Section 5.6.5), control plane signals (Section 5.6.6), and directives (Section 5.6.7).

5.6.1 Contextual Perspectives

We first evaluate the impact of the context choice (Section 5.3.2) on the quality of initial hypotheses generated by Observer. Recall that Observer generates diverse contexts by constraining the telemetry visible to the reasoning agent. We consider four contexts: **Global+Logs** (all telemetry including system logs, i.e., comprehensive deployment context), **Global** (all telemetry without

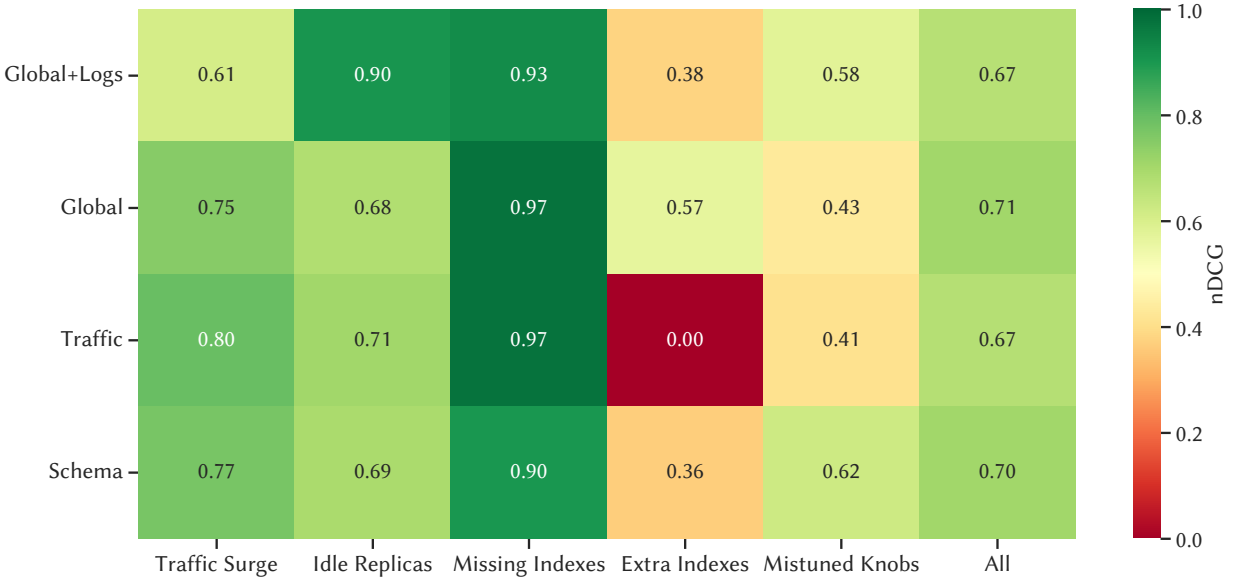


Figure 5.7: Contextual Perspectives – nDCG computed from the three ranked hypotheses generated by each contextual perspective using GPT-5-mini. We evaluate across five classes of anomaly scenarios: traffic surge, idle replicas, missing indexes, extra indexes, and mistuned knobs.

logs), **Traffic** (workload and proxy information only), and **Schema** (database schema and usage statistics). We consider five classes of anomaly scenarios and generate multiple sub-scenarios within each class: traffic surge (**27** sub-scenarios), idle replica (**9** sub-scenarios), missing indexes (**6** sub-scenarios), extra indexes (**6** sub-scenarios), and mistuned knobs (**6** sub-scenarios). For each context-scenario pair, we generate three ranked hypotheses with GPT-5-mini. In Figure 5.7, we report the mean normalized discounted cumulative gain (nDCG) [50] which favors correct hypotheses ranked higher: 1.0 indicates the first hypothesis is correct, $\sim$0.6 the second, and $\sim$0.5 the third. We also present in Table 5.2 the mean input tokens, output tokens, estimated cost, and number of interactions consumed by Observer when generating hypotheses for each scenario based on each context type.

As shown in Figure 5.7, the comprehensive **Global+Logs** context generally performs well across all the various scenarios. Specifically, Observer with the **Global+Logs** context achieves the highest nDCG for idle replicas (0.90) and competitive nDCG for missing indexes (0.93) and mistuned knobs (0.58). For the idle replica and missing index scenarios, Observer correctly identifies that AgenticOperator can decommission idle replicas (e.g., scale-in) or build the appropriate missing index as its first out of three hypotheses. As shown in Table 5.2, Observer generally consumes the most tokens for **Global+Logs** that compounds across multiple rounds.

However, specialized contexts can outperform the **Global+Logs** context in their area of focus. For instance, as seen in Figure 5.7 and Table 5.2, the **Traffic** context achieves a higher nDCG on the traffic surge scenario (0.8 compared to 0.61). It also uses $<$50% of the input tokens and about 2.3 fewer interactions (i.e., concludes in less time). For the missing indexes scenario, Observer with **Traffic** context is able to avoid on average two information-gathering rounds as the agent is able to focus directly on analyzing the problematic queries. By contrast, Observer with **Global+Logs** context spends extra interactions on narrowing the workload.

	Input Tokens	Output Tokens	Estimated Cost	Number of Interactions
Traffic Surge				
Global+Logs	169.7K	4.3K	$0.05	5.9
Global	126.8K	3.6K	$0.04	4.1
Traffic	84.5K	2.1K	$0.03	3.6
Schema	111.6K	2.4K	$0.03	4.2
Idle Replicas				
Global+Logs	83.2K	2.0K	$0.02	2.2
Global	90.0K	2.2K	$0.03	2.6
Traffic	43.8K	1.3K	$0.01	2.2
Schema	72.5K	1.3K	$0.02	2.1
Missing Indexes				
Global+Logs	205.7K	5.6K	$0.06	7.0
Global	132.1K	6.5K	$0.05	5.1
Traffic	75.4K	1.8K	$0.02	4
Schema	121.4K	2.1K	$0.03	5.7
Extra Indexes				
Global+Logs	171.2K	5.3K	$0.05	6.2
Global	122.9K	5.1K	$0.04	4.3
Traffic	77.0K	1.8K	$0.02	4
Schema	66.5K	1.6K	$0.02	2.9
Mistuned Knobs				
Global+Logs	206.0K	7.5K	$0.07	7.4
Global	157.0K	5.8K	$0.05	4.7
Traffic	73.3K	1.8K	$0.02	3.4
Schema	102.3K	1.9K	$0.03	4.4

Table 5.2: Contextual Perspectives: Cost Overhead – Mean input tokens, output tokens, estimated cost, and number of interactions for hypothesis generation from each contextual perspective across five classes of anomaly scenarios using GPT-5-mini. We compute costs without token caching.

Another interesting example relates to system logs. As shown in Figure 5.7, the **Global** context achieves a higher nDCG on the extra indexes scenario (0.57 compared to 0.38) with 28% fewer input tokens and about 2 fewer rounds of interactions. With the **Global+Logs** context, the agent gets distracted by the serialization failure log entries and focuses on methods to change the isolation level. By omitting the system logs, the agent is able to focus instead on addressing actual performance issues such as removing redundant indexes.

These results motivate AgenticOperator's design of generating hypotheses from *multiple* contexts. No single context dominates across all scenarios. In some cases, using an overly specialized **Traffic** context can lead to poor performance for some scenarios, such as extra indexes (nDCG 0.00). By being able to build diverse hypotheses from different contexts, Observer is able to cover a greater range and variety of possible root causes. The evolutionary algorithm Orchestrator is then able to select and explore the most promising hypotheses.

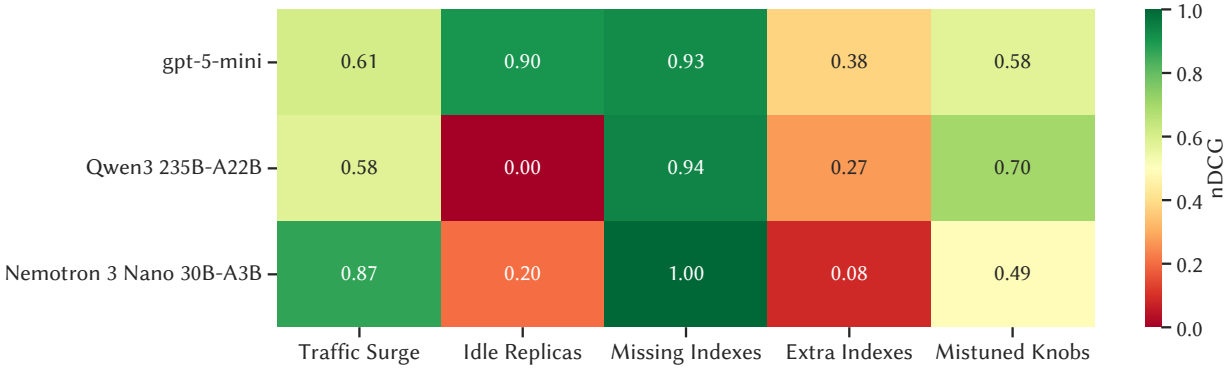


Figure 5.8: Reasoning Model – nDCG of ranked hypotheses generated from the **Global+Logs** context across five anomaly scenarios (traffic surge, idle replicas, missing indexes, extra indexes, and mistuned knobs). We evaluate three reasoning models of different sizes from largest to smallest: **GPT-5-mini**, **Qwen3 235B-A22B**, and **Nemotron 3 Nano 30B-A3B**.

5.6.2 Reasoning Model

We next evaluate the impact of the reasoning model on hypothesis quality. AgenticOperator's plug-and-play architecture (Section 5.3.1) supports any tool-calling LLM as its reasoning agent. We compare three models ordered from largest to smallest effective capacity: **GPT-5-mini**, **Qwen3 235B-A22B**, and **Nemotron 3 Nano 30B-A3B**. For each model, we generate three ranked hypotheses using the **Global+Logs** context across the same five scenarios: traffic surge (**27** sub-scenarios), idle replica (**9** sub-scenarios), missing indexes (**6** sub-scenarios), extra indexes (**6** sub-scenarios), and mistuned knobs (**6** sub-scenarios).

Analyzing Figure 5.8, we note that the smallest model (Nemotron) performs the best for the traffic surge scenario (0.87 nDCG). Although Nemotron uses $\sim 8\times$ more output tokens compared to the largest model (GPT-5-mini) with 0.61 nDCG, it is still cheaper due to lower per-token inference costs. We note that Nemotron is prone to generating substantially more verbose reasoning traces which increases costs. In this case, we attribute its superior performance to the relatively straightforward diagnosis of connection contention, which it is able to quickly pick out from the proxy telemetry. By contrast, we find that Nemotron performs much worse on the extra indexes scenario. In that case, it gets fixated on log-level messages (e.g., duplicate key violations) and attempts to build indexes or change isolation levels to fix the problem.

Further, we observe that both Qwen3 and Nemotron perform poorly on the idle replica scenario (0.00 nDCG and 0.20 nDCG respectively) compared to GPT-5-mini (0.90). These low nDCG scores indicate that both smaller models never output the correct hypothesis. Upon inspection, we find two failure modes for these smaller models. The first is that they fail to act upon their prime directive of reducing resources by scaling-in idle replicas. Rather, they tend to focus on cutting pool connections with limited benefit, instead of more high impact actions (e.g., decommissioning replicas, dropping redundant indexes). Second, they strongly focus on the tool name to relate to the deployment contexts. They interpret the tool name "recentralize_reads" as a request to re-route traffic back to the primary instance, rather than also decommissioning the idle replica as described in the tool description. We explore the nuances of tool naming and how they impact hypothesis quality in more detail in Section 5.6.4.

	Input Tokens	Output Tokens	Estimated Cost	Number of Interactions
Traffic Surge				
gpt-5-mini	152.8K	2.0K	$0.04	4.7
Qwen3 235B-A22B	321.9K	2.5K	$0.03	16.9
Nemotron 3 Nano 30B-A3B	184.1K	19.3K	$0.01	4.3
Idle Replicas				
gpt-5-mini	82.0K	1.2K	$0.02	2.1
Qwen3 235B-A22B	—	—	—	—
Nemotron 3 Nano 30B-A3B	96.8K	9.0K	$0.007	2.3
Missing Indexes				
gpt-5-mini	101.3K	1.6K	$0.03	4.8
Qwen3 235B-A22B	373.7K	2.2K	$0.04	14.7
Nemotron 3 Nano 30B-A3B	203.5K	16.2K	$0.01	4.4
Extra Indexes				
gpt-5-mini	119.6K	1.8K	$0.03	4.3
Qwen3 235B-A22B	330.0K	2.5K	$0.03	15.5
Nemotron 3 Nano 30B-A3B	222.1K	21.0K	$0.02	5
Mistuned Knobs				
gpt-5-mini	136.9K	1.7K	$0.04	4.2
Qwen3 235B-A22B	316.5K	2.2K	$0.03	17.2
Nemotron 3 Nano 30B-A3B	239.5K	22.9K	$0.02	5.2

Table 5.3: Reasoning Model: Cost Overhead – Mean input tokens, output tokens, estimated cost, and number of interactions for hypothesis generation from the **Global+Logs** context across five anomaly scenarios using three reasoning models. "—" indicates the model rejected the scenario and concluded the deployment was operating normally. We compute costs without token caching.

As shown in Figure 5.8, a "one-size-fits-all" approach of using the largest model (GPT-5-mini) may not be the best approach. Depending on the scenario, smaller models perform better and with higher cost efficiency (e.g., traffic surge). We defer automated techniques for routing to the appropriate reasoning model based on the context type and cost constraints to future work.

5.6.3 Tool-Hypothesis Alignment

Beyond hypothesis quality, we measure whether the reasoning agent's tool invocations are consistent with its declared hypothesis. A high-quality hypothesis is necessary but not sufficient: if the agent believes the root cause is a "missing index" but then decides to invoke a knob tuner, the subsequent actions will not address the root cause. Based on domain knowledge, we define a hypothesis-tool mapping from hypothesis categories to tool names. For instance, a "missing index" hypothesis should be followed by an "index tuner" (e.g., "CREATE INDEX" sql, Dexter [11], DTA [23]). We then compute an alignment score between each hypothesis and its subsequent optimization tool call: 1.0 indicates the invocation aligns with the declared hypothesis, while lower scores indicate divergent tool usage.

As the reasoning agent emits its hypothesis in natural language, we employ an automated method to identify the hypothesis category. For each context-scenario, we extract the last hypothesis explicitly generated by the reasoning agent made through a tool call (e.g., *declare_hypothesis*

Figure 5.9: Tool-Hypothesis Alignment – Mean alignment score between hypotheses and subsequent optimization tool invocation across four contextual perspectives (Global+Logs, Global, Traffic, and Schema) and five scenarios (traffic surge, idle replicas, missing indexes, extra indexes, and mistuned knobs). Alignment scores are computed from a golden mapping from hypothesis category to tools.

or *revise_hypothesis*) along with all information-gathering tool calls made. We use a small language model (Qwen3 4B) to identify the hypothesis category from the following set: **proxy sizing**, **proxy routing**, **proxy sizing and routing**, **knob tuning**, **query tuning**, **add index**, and **drop index**. Based on the provided toolbox, we manually construct the golden mapping from those categories to tool names. For instance, a "proxy sizing" hypothesis should be followed by either `apply_pool_sizes` or `tune_pool_sizes`. Similarly, a "knob tuning" hypothesis should be followed by either `apply_knob` or `bayesian_knob_tuner`.

As shown in Figure 5.9, alignment is generally high across all context-scenario pairs. The **Global+Logs** context exhibits the lowest overall alignment (mean 0.70), with weaker scores for mistuned knobs (0.61). We find that including system logs causes the agent to invoke exploratory tools that diverge from its stated hypothesis as it investigates log-specific symptoms and then invokes differing optimization tools. We recognize this is a limitation of the approach that could be mitigated by enforcing stricter tool usage rules (e.g., must explicitly update hypothesis before invoking an optimization tool) in the prompt or harness. We defer this to future work.

Narrower contexts (**Traffic**, **Schema**) maintain strong alignment (means of 0.89 and 0.83 respectively), indicating that focused telemetry helps the agent stay consistent between its analysis and actions. However, when comparing with Figure 5.7, high alignment scores do not necessarily imply high hypothesis quality. The **Traffic** context achieves 1.00 alignment on extra indexes but 0.00 nDCG, meaning the agent consistently invokes tools matching its (incorrect) hypothesis. This experiment reveals that AgenticOperator is, with high fidelity, able to correctly select the right optimization tool based on the hypothesis obtained from analyzing the deployment context. Still, AgenticOperator's multi-context approach is needed to produce a diverse population of hypotheses and potential mitigation strategies.

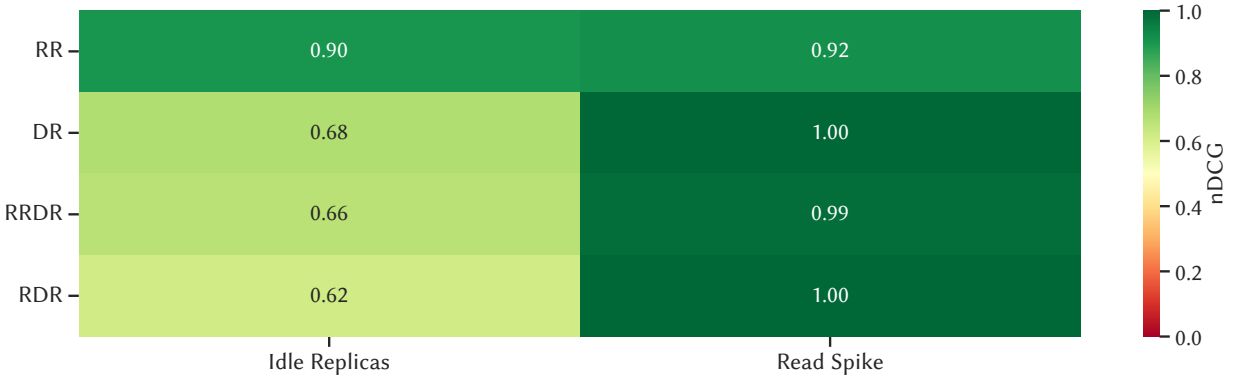


Figure 5.10: Tool Naming – nDCG of ranked hypotheses from GPT-5-mini with the **Global+Logs** context when varying scale-in tool name: **RR** (`recentralize_reads`), **DR** (`decommission_replica`), **RRDR** (`recentralize_reads_decommission_replica`), and **RDR** (`recentralize_decommission_replica`). We consider two scenarios: idle replicas and read spike.

5.6.4 Tool Naming

We next study how tool naming affects hypothesis quality. Since tool definitions are provided by external MCP servers [13] under AgenticOperator's plug-and-play architecture, these factors are not under AgenticOperator's control. For this experiment, we only vary the name of the scale-in tool and not the arguments or description. We consider four names: **RR** (`recentralize_reads`), **DR** (`decommission_replica`), **RRDR** (`recentralize_reads_decommission_replica`), and **RDR** (`recentralize_decommission_replica`). We only consider the idle replicas (9 sub-scenarios) and read spike (6 sub-scenarios) scenarios, as they are the most relevant scenarios to the scale-in tool. For the read spike scenario, we create an OLAP workload driven by multiple normal-priority clients to mimic reporting or analysis queries. We generate all hypotheses using GPT-5-mini with the **Global+Logs** context and compute the nDCG of the three ranked hypotheses.

As shown in Figure 5.10, all names achieve high nDCG ($\geq$0.92) for the read spike scenario, where the need to route traffic to a replica is explicit. With the **RR** name, the agent occasionally outputs a "missing index" hypothesis. By contrast, with the other names, the agent consistently recognizes that there is a HTAP workload and decides to offload read-only traffic to a replica. By explicitly introducing a notion of replicas through the tool name, the agent is more able to relate the `offload_reads` tool with the inverse scale-in tool.

However, for idle replicas, **RR** (0.9) substantially outperforms the alternatives (0.62–0.68). We observe that the agent tends to maintain a sub-goal of preserving deployment slack to handle the workload when load resumes. As such, it is conservative towards infrastructure actions (e.g., decommission replicas) and instead prefers lower-cost actions (e.g., reducing active connections). Despite the "recentralize_reads" description mentioning decommissioning replicas, the agent is better able to connect the observed deployment (e.g., low utilization) to the action of moving traffic back to the primary instance (i.e., no offloading). By contrast, the other names emphasize infrastructure decisions. Although AgenticOperator works with various tool names, the user is responsible for ensuring that tool descriptions are semantically aligned with their directives.

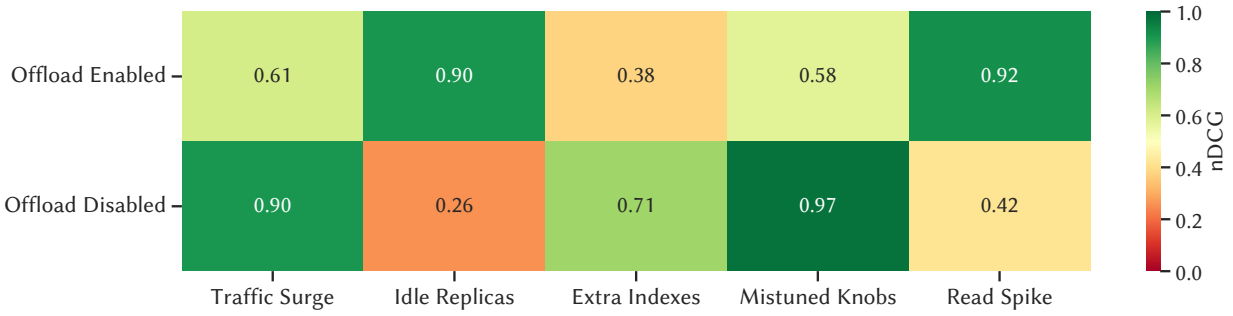


Figure 5.11: Read Replica Availability – nDCG of ranked hypotheses from GPT-5-mini with the **Global+Logs** context when varying offload availability: **Enabled** indicates that there is available read replica capacity, and **Disabled** indicates that there is no capacity. We evaluate across five scenarios: traffic surge, idle replica, extra indexes, mistuned knobs, and read spike.

5.6.5 Read Replica Availability

External to AgenticOperator, there may be higher-level control plane signals (abbreviated hereafter as *control signals*) that can influence or guide the agent's behavior. For instance, hosting providers may expose information indicating whether read replica capacity is available for offloading. We evaluate the impact of varying the availability of read replica capacity through this control signal on AgenticOperator. The `offload_reads` tool remains in the toolbox in both configurations; only the contextual metadata provided changes. This contextual metadata is provided by a higher-order service (e.g., hosting provider) and not necessarily from the user themselves. We evaluate across five scenarios: traffic surge (**27** sub-scenarios), idle replica (**9** sub-scenarios), extra indexes (**6** sub-scenarios), mistuned knobs (**6** sub-scenarios), and read spike (**6** sub-scenarios). We generate all hypotheses using GPT-5-mini with the **Global+Logs** context and compute the nDCG of the three ranked hypotheses.

We first discuss the read spike scenario, as its results are most aligned with expectations. As shown in Figure 5.11, indicating available replica capacity improves nDCG from 0.42 to 0.92. For read spike scenarios, the agent correctly reasons that offloading reads will improve the deployment based on the context. The control signal that indicates available replica capacity further reinforces this hypothesis.

Next, as shown in Figure 5.11, indicating unavailable capacity improves nDCG for these scenarios: traffic surge (0.61 to 0.90), extra indexes (0.38 to 0.71), and mistuned knobs (0.58 to 0.97). We observe that the agent has an implicit bias towards offloading reads when capacity is available. Consequently, with a control signal indicating unavailable capacity, the agent correctly avoids unnecessary offload actions and instead focuses on more appropriate mitigations. In the case of mistuned knobs, the agent now proposes knob tuning as its first hypothesis.

Surprisingly, indicating unavailable capacity substantially degrades nDCG for the idle replicas scenario from 0.90 to 0.26. In this case, the control signal leads the agent down different reasoning trajectories. It either attempts to re-enable offloading reads to the existing replica, shed active connections, or invoke maintenance tools. We infer this is due to how the agent understands its prime directives and its implicit bias towards conservative resource reduction (e.g., reduce connections over decommissioning replicas). Beyond availability of replica capacity, the control

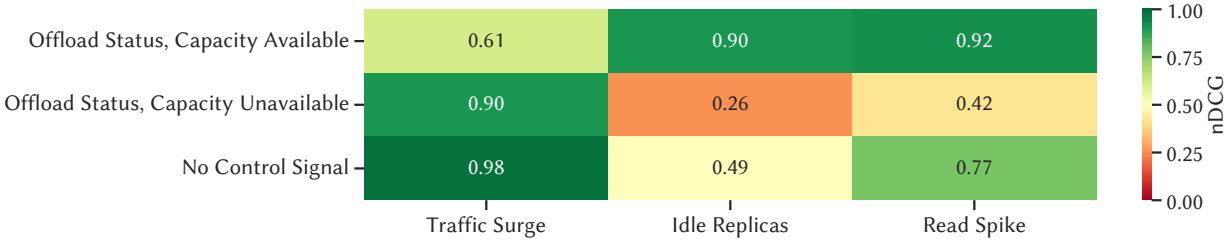


Figure 5.12: Control Plane Signals – nDCG of ranked hypotheses from GPT-5-mini with the **Global+Logs** context across three scenarios: traffic surge, idle replicas, and read spike. We vary whether control plane signals are present and the degree of information injected into the context.

plane can provide other signals (e.g., health-checks, tuned status, forecasted load) to guide the agent's behavior. We defer selecting these control signals, possibly based on prime directives or other service-level objectives, to future work.

5.6.6 Control Plane Signals

In Section 5.6.5, we evaluated the effect of varying replica availability (i.e., available or unavailable). In this experiment, we explore the impact of whether omitting or including control plane signals as a whole has a noticeable impact on hypothesis quality. We look at two signals: whether reads are currently offloaded (Offload Status) and whether there is available read replica capacity (Capacity). We evaluate three scenarios: traffic surge, idle replicas, and read spike.

Similar to the results presented in Section 5.6.5, removing all control plane signals improves nDCG for traffic surge (0.98 versus 0.61), as there is a lack of injected signals that reinforce replica-based mitigations (e.g., offloading reads). However, removing all control plane signals degrades idle replica (0.49 versus 0.90) and read spike (0.77 versus 0.92) scenarios where these additional control plane signals provide useful information.

A particularly interesting observation is that omitting all control signals performs strictly better than injecting the offload status and unavailable replica capacity signals. We believe that the signal indicating no available replica capacity is overly prioritized by or distracting to the agent, which results in worse reasoning trajectories and outcomes. Although this can be mitigated by allowing Observer to create a new context that conditionally injects control plane signals, we believe a more principled approach is required. Further investigation is required to understand how agents use these control plane signals to guide their reasoning.

5.6.7 Directive Sensitivity

We next study how the phrasing of the prime directive affects hypothesis quality. Recall the original prime directive: "optimize p99 latency across all users, prioritizing VIP-Client when active, and reduce deployment resources". We introduce an alternative prime directive: "optimize p99 latency. prioritize high-priority users". This prime directive directly focuses on the VIP-Client and omits mention of resource reduction and the phrasing for "all users". We evaluate across two scenarios: traffic surge (27 sub-scenarios) and read spike (6 sub-scenarios). We

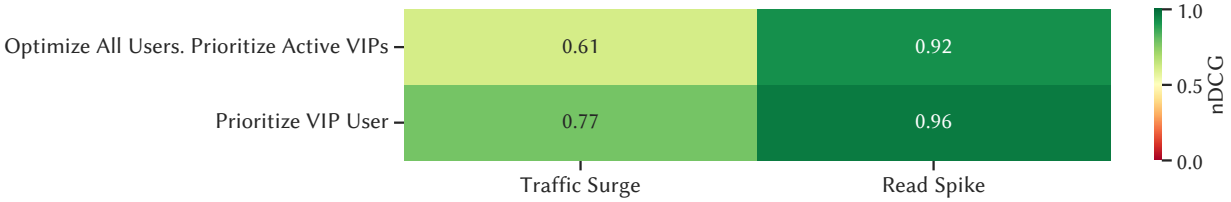


Figure 5.13: Directive Sensitivity – nDCG of ranked hypotheses when varying the phrasing of the prime directive between optimizing for all users versus focusing primarily on the VIP-Client across traffic surge and read spike scenarios. All experiments use GPT-5-mini with the **Global+Logs** context.

generate all hypotheses using GPT-5-mini with the **Global+Logs** context and compute the nDCG of the three ranked hypotheses.

As shown in Figure 5.13, focusing on the high-priority user improves traffic surge nDCG from 0.61 to 0.77 and read spike nDCG from 0.92 to 0.96. For read spikes, this focus more clearly steers the agent towards offloading reads rather than attempting to build indexes on the single primary. In the case of traffic surge, by using a prime directive that more directly focuses on the VIP-Client, the agent focuses more on re-allocating pool resources rather than routing traffic to a replica. The desirability of these more aggressive pool changes is a human-level judgement that lies outside the scope of this work. We defer a more exhaustive study of the impact of the prime directive and whether the range of user objectives (e.g., throughput, latency, resource utilization) can be reduced into a set of canonical prime directives to future work.

5.7 Conclusion

Existing diagnostic systems for DBMS deployments can identify anomalies and propose root causes, but they cannot autonomously translate that analysis into remedial action. The human operator remains the critical bridge between diagnostics and deployment changes, slowing response times. To address this, we present the AgenticOperator framework that autonomously reasons about ongoing anomalies and orchestrates optimization tools to mitigate them. AgenticOperator uses diverse contextual perspectives to generate initial hypothesis branches and an evolutionary algorithm to explore, refine, and cross-pollinate findings across branches before selectively deploying high-confidence mitigations. We evaluate AgenticOperator against a ReAct agent across two case studies (traffic surge and missing index), in which AgenticOperator achieves lower latency and more stable mitigation trajectories. We further present sensitivity studies demonstrating the importance of AgenticOperator's multi-context, plug-and-play design.

Chapter 6

Related Work

We now discuss related work on autonomous DBMS research that complements the background in Chapter 2 and the individual chapters (Chapters 3 to 5). We organize the discussion into tuning agents (Section 6.1), workload forecasting (Section 6.2), behavior modeling (Section 6.3), learned representations (Section 6.4), natural language debugging (Section 6.5), and learned components (Section 6.6).

6.1 Tuning Agents

The literature is rich in ML-based tuning agents. In Chapter 2, we described the broad categories of DBMS optimization tools (i.e., knobs, physical design, and queries) and their interfaces. We now survey the specific ML-based techniques proposed for each category.

Resource Tuning: These agents optimize the DBMS's exposed system knobs, optionally considering the system's resource consumption. Existing approaches utilize either Bayesian optimization [53, 109] or gradient-based techniques [130] to explore and exploit knob configurations. Bayesian approaches model the unknown objective function with a surrogate (e.g., Gaussian process) and use an acquisition function to balance exploration and exploitation. Gradient-based methods instead differentiate through a learned performance model to directly optimize knob values. Both families treat knobs as a standalone search problem and do not consider how knob settings interact with other configuration spaces, such as indexes or query hints. For instance, a knob tuner may disable index scans to improve performance on the current workload, preventing a downstream index tuner from discovering beneficial indexes. Proto-X (Chapter 3) addresses this by embedding knob alongside index and query hint actions in a unified latent space, allowing the agent to reason about cross-space interactions through proto-action neighborhoods.

Capacity Planning: Other agents have focused on capacity planning [20]. These approaches usually involve forecasting to identify future trends or resource requirements. Using forecasts and behavior models to predict resource consumption, an agent allocates tasks or provisions resources to satisfy user-specified constraints (e.g., latency SLOs, cost budgets). Although capacity planning is upstream of the tuning process, its outputs (e.g., allocated hardware,

expected load) directly influence the configuration that a tuner should target. For example, if capacity planning predicts a workload spike, a tuner should proactively adapt the configuration, as explored in the drift scenarios of Chapter 4. Furthermore, its outputs can be used to inform the AgenticOperator about the expected workload and resource requirements, allowing it to reason about potential tools to invoke from its toolbox (Chapter 5) that may also be capacity-related (e.g., offloading reads, scaling up/down resources).

Parametric Query Optimization: Some work focuses on parametric query optimization, identifying the best query plans to keep and reuse for different parameter bindings [108]. These techniques precompute a set of plans that collectively cover the parameter space, switching between them at runtime based on the observed parameter values. This is related to the query hint tuning in Chapter 3, where Proto-X discovers per-query hints that steer the optimizer. By combining these techniques with those in Chapter 3, we can potentially decompose each query based on its set of interesting parameter values and provide targeted hints for each partition. This would improve resilience to plan regressions caused by query hints when parameters change.

Query Tuning: Work on query tuning includes query rewriting [143], which uses a tree search to balance exploring and exploiting known rewriting rules, and steering the query optimizer with a hint set [12, 77] to alter the behavior of individual queries. These approaches tune queries in isolation from other configuration spaces. In contrast, Proto-X (Chapter 3) jointly tunes query hints alongside knobs and indexes, capturing cross-space interactions that isolated query tuners miss. Furthermore, Booster (Chapter 4) exploits per-query historical configurations by enriching LLM prompts with past tuning attempts, enabling more effective adaptation to workload drifts.

Physical Design: Other work has focused on applying machine learning or reinforcement learning techniques to physical design problems. Existing agents are capable of recommending partitioning schemes [129], materialized views [127], and indexes [96, 114, 116]. Despite each tool's sophistication, they consider these aspects in isolation from other system aspects (e.g., system knobs, query hints). As shown in Chapter 3, this isolation leads to suboptimal configurations because the benefit of a physical design depends on system knobs (e.g., whether index scans are enabled) and query hints (e.g., whether the optimizer is forced to ignore indexes).

Joint Configuration Space Tuning: HMAB [89] tunes indexes and materialized views by treating the problem as a two-tiered bandit where the first tier independently selects views and indexes, and the second selects their combination. However, HMAB does not consider system knobs or query hints, limiting its ability to discover configurations that exploit cross-space interactions. UDO [111] tunes indexes and knobs by decomposing the configuration space into subspaces and iteratively optimizing each one while holding the others fixed, resembling the sequential tuning paradigm. UniTune [134] extends this to additionally support query hints, but still relies on round-robin scheduling across subspaces.

More recently, LLM-based approaches [39, 47, 123] have been proposed that instruct a language model to directly generate multi-faceted configurations. These are complementary to

the adaptation framework in Chapter 4, which can augment any tuner (including LLM-based ones) with historical knowledge to accelerate adaptation.

6.2 Workload Forecasting

As noted in Section 2.1, a self-driving DBMS relies on forecasting to predict future workloads and proactively prepare for changes. Existing literature focuses on predicting future queries and loads for DBMS optimization. These techniques include using time-series models to forecast the volume of query templates at different future horizons [71], using learned temporal models to predict parameterized queries [45], and identifying ongoing workload drifts from telemetry [65].

These forecasting techniques are upstream of and complementary to the tuning and adaptation frameworks in this thesis. Once a drift is detected, the human operator or a self-driving DBMS must decide how to adapt. These drift detection mechanisms can be used to trigger the AgenticOperator to intervene, analyze, and re-optimize the deployment if needed. These mechanisms could also be used to proactively invoke Booster and Proto-X to identify the post-drift optimal DBMS configuration.

6.3 Behavior Models

Behavior modeling aims to produce small and accurate models that can infer the DBMS's performance for a given configuration and workload [69, 73, 75]. These models range from query performance predictors (QPP) that estimate the runtime of a single query plan to broader workload-level models that predict aggregate metrics (e.g., total throughput, tail latency) under different configurations. Graph-based models [141] encode the query plan structure and have shown improved generalization across plan shapes, particularly for concurrent queries.

Behavior models serve two primary roles in autonomous DBMS optimization. First, they can substitute for actual DBMS evaluations during tuning, allowing agents to speculatively evaluate candidate configurations without the overhead of running the workload [95]. For instance, an index tuner can use a behavior model to estimate a index's benefit without materializing it. Second, behavior models can prune the action space by filtering out configurations that are predicted to perform poorly [97], reducing the number of expensive evaluations.

These models are complementary to the techniques presented in this thesis. Proto-X (Chapter 3) uses the DBMS's query optimizer cost estimates as a lightweight behavior model during Phase I to estimate the benefit of candidate actions when constructing the latent space. More sophisticated behavior models could replace these cost estimates to improve the quality of the latent space's structure. The Booster framework (Chapter 4) generates substantial DBMS telemetry during its operation (e.g., per-query runtimes under different configurations), which an operator can use to train and refine behavior models for future tuning sessions. The AgenticOperator framework (Chapter 5) could also leverage behavior models to speculatively evaluate the impact of proposed mitigations before deploying them to the production environment, rather than relying on replica-based evaluation.

6.4 Workload and Query Representations

This area focuses on deriving a query [138] or workload [96] representation that is conducive to downstream tasks (e.g., behavior models, tuning). Query-level representations typically encode the query plan tree into a fixed-dimensional vector by learning from plan structure, operator types, and estimated cardinalities. Workload-level representations aggregate individual query representations into a single vector that captures the workload's overall characteristics (e.g., access patterns, resource intensity). Creating discriminative representations allows tuning agents to generalize across workloads by providing a compact summary of the workload's properties that is independent of schema-specific details.

These learned representations represent a step towards a universal "foundation model" for database [115] and are directly applicable to techniques discussed in this thesis. Discriminative learned representations that capture workload semantics and obey the similarity principle (e.g., similar workloads have similar representations) would allow us to condition Proto-X's latent space based on the workload itself (Chapter 3). Alternatively, they could be used as an alternative retrieval mechanism within Booster's pipeline, potentially improving the quality of retrieved historical contexts by better capturing query semantics beyond plan structure.

6.5 Natural Language Debugging

With advances in large language models (LLMs), recent work has focused on debugging SQL issues through a natural language interface. This work ranges from natural language to SQL techniques [35, 137], root cause analysis [84], and debugging user problems with their deployments [112, 144]. These techniques primarily focus on designing retrieval mechanisms (e.g., ranking functions, fine-tuned embedders) for selecting the top-k relevant chunks from a knowledge base to provide to the LLM based on the user's question.

Natural language to SQL techniques [35, 137] translate a user's natural language question into executable SQL, often leveraging schema retrieval and few-shot examples. Root cause analysis frameworks [84] use LLMs to reason about anomaly symptoms and DBMS telemetry to produce ranked lists of possible causes. Multi-agent diagnostic systems [112, 144] decompose the diagnosis problem into sub-problems, with specialized sub-agents analyzing different DBMS aspects (e.g., host metrics, query plans, locks) before a coordinator synthesizes the findings. However, these systems terminate after producing a diagnosis or recommendation, relying on a human operator to evaluate and apply corrective actions. The techniques in Chapter 5 address this limitation by enabling direct action based on the diagnosis.

6.6 Learned Components

These are traditional DBMS components augmented with machine learning. Existing work has focused on instance-optimized layouts [33], learned data structures (e.g., learned trees [41]), learned algorithms (e.g., join [94]), and learned query optimization [8, 76, 77, 120].

Learned query optimizers [76, 120] aim to produce better plans directly, which is complementary to the hint-based or feedback-driven approaches [8, 77]. Both techniques are necessary

to achieve optimal plan quality. Learned query optimization can produce near-optimal or better plans but requires substantial training data and may regress on unseen query patterns. By contrast, hint-based approaches provide a more controlled mechanism for steering plan selection but are constrained by the optimizer's existing plan space. As learned query optimizers mature, they will form another key tool in the DBMS's optimization toolbox (Chapter 5) alongside hint-driven approaches.

Other research has focused on learned cardinality estimation, due to its high impact on query optimizer plan quality [61]. Recent work targets correctly answering cardinality-related queries with techniques resilient to workload and data changes [80]. Accurate cardinality estimation directly reduces the frequency of suboptimal plans caused by the optimizer's reliance on stale or inaccurate statistics. For the techniques in this thesis, improved cardinality estimation would reduce the number of cases where query hints or index recommendations are needed to compensate for poor optimizer decisions. Co-adapting learned components (e.g., cardinality estimators, learned optimizers) and external tuning agents remains an open area for future work.

Chapter 7

Future Work

In the preceding chapters we presented three systems that exploit similarity along different axes to holistically optimize DBMS deployments: actions (Proto-X, Chapter 3), workloads and configurations (Booster, Chapter 4), and scenarios (AgenticOperator, Chapter 5). Each system relies on offline or replica-based evaluation and uses hand-crafted strategies for constructing the information (i.e., context) provided to its reasoning components. We now discuss several open directions, along with the challenges and limitations involved: (1) online operation without replica-based verification (Section 7.1), (2) dynamic specialized context construction (Section 7.2), (3) expanding the tuning boundary (Section 7.3), and (4) explainability (Section 7.4).

7.1 Online Without Replica-Based Verification

A common thread across Proto-X, Booster, and AgenticOperator is their reliance on read replicas or isolated sandboxes to safely evaluate candidate configurations and mitigations before deploying them to production. Proto-X trains and evaluates on a dedicated replica to measure workload performance after each configuration change. Booster similarly evaluates composed configurations on a replica before recommending them to the operator. AgenticOperator executes optimization tools and obtains each tool's impacts on an isolated replica, comparing performance against a workload snapshot before deploying those changes live.

While replicas provide a reliable safety net, maintaining synchronized replicas for the sole purpose of tuning is expensive and operationally burdensome. In many deployments, replicas serve live read traffic and cannot be freely repurposed for experimentation. In other cases, read replicas impose infrastructure overhead that makes them impractical. An appealing alternative is to perform all tuning *online*, without a mandatory replica-based verification step. We outline the key challenges that such a shift would entail.

Conservative Guardrails for Disruptive Changes: Not all configuration changes are equal in their impact on a running system. Lightweight knob adjustments (e.g., changing an optimizer flag) can take effect within a session with negligible overhead, whereas other changes require a full server restart (e.g., buffer pool) or trigger expensive I/O (e.g., building a large B-tree index). An online system must classify changes by their disruptiveness and enforce guardrails accordingly.

For instance, it may reject unsafe actions (e.g., setting buffer pool size larger than the available memory), defer restart-requiring changes to a scheduled maintenance window, or rate-limit I/O-intensive operations (e.g., concurrent index builds) to avoid saturating disk bandwidth during peak hours. This information may also be propagated down to AgenticOperator to proactively constrain the tools invoked. For instance, AgenticOperator may limit Proto-X to optimize only knobs while the system is under high load before switching it into a more aggressive holistic mode after the load subsides. Alternatively, it can constrain the maximal size of the changes proposed in a given optimization step (e.g., can only increase the buffer pool up to 10%).

Accounting for Live Query Load: On a replica, the tuner controls the workload and observes it in isolation, so any change in performance metrics can be attributed directly to the tuner's actions. Since production workloads are externally driven and fluctuate over time, the system requires additional mechanisms for attribution. For instance, if a query hint tuner suggests a change, the system cannot provide feedback unless that query recurs within the observation window. A simple workaround is to extend the observation window; alternatively, the system can reject changes targeting queries that have not been observed recently. Recent research has also proposed mechanisms to incorporate delayed feedback into the tuning process [111].

Furthermore, sampling the live workload risks producing skewed traces that omit infrequent yet important queries [29]. To guard against regressions on high-value templates, the system may need to run sanity queries against the modified configuration. Designing a minimal yet sufficient set of such queries [96] and executing them without materially affecting production remains an open problem.

Rapid Rollback and Regression Detection: Without a replica to fall back to, the system needs a fast rollback mechanism. For knob changes, this can be as simple as restoring the previous value. For physical design changes (e.g., dropping an index), rollback is more expensive and may require rebuilding the original structure, during which the deployment operates in a degraded state. The system must therefore maintain fine-grained checkpoints of its configuration state, keep shadow copies of critical objects (e.g., indexes) [29], and continuously monitor performance against a baseline. A regression detector must be sensitive enough to catch meaningful degradations promptly while tolerating the natural variance of production workloads.

7.2 Dynamic Specialized Context Construction

Both Booster and AgenticOperator construct the information provided to their LLM-based reasoning components through pre-defined strategies. Booster structures historical tuning artifacts into fixed templates of query-level insights (e.g., plan similarities, configuration fragments) that are retrieved based on structural matching. AgenticOperator defines a fixed set of contextual perspectives (e.g., Traffic, Buffers/WAL, Schema, Global; see Section 5.3.2) that emphasize different subsystems of the deployment. In both cases, the context construction strategy is determined by domain knowledge and does not adapt to individual situations. Inspired by memory management approaches for LLMs [85], a future direction is to *dynamically* construct

specialized contexts based on higher-order elements (e.g., alert type, metric profile, workload characteristics) to determine which information blocks are most relevant.

Instead of pre-defined contexts, the system would maintain a library of *context blocks* that are modular, self-contained descriptions of different deployment aspects (e.g., lock contention statistics, index usage, checkpoint frequency, proxy routing tables). A higher-order controller would then select and compose blocks into a context based on the signals and a policy learned from historical sessions. For example, if the alert indicates a p99 latency spike and the metric profile shows elevated WAL write rates, the controller would prioritize WAL-related and checkpoint-related blocks over replication or proxy blocks. This would also allow the controller to swap blocks over mutation steps (Section 5.4.1) to adapt to the evolving situation. For instance, if early rounds reveal that the anomaly is query-related rather than traffic-related, subsequent rounds could swap in more detailed query-level blocks (e.g., per-query plan analyses, index suggestions) while dropping coarse system-level blocks. This adaptive refinement would help the reasoning agent focus on increasingly relevant information as the diagnosis narrows, reducing noise from irrelevant context that can mislead the LLM [70].

However, dynamic context construction introduces its own risks. First, incorrect signal interpretation can lead to systematically excluding relevant information, causing the system to miss the true root cause. For example, if a p99 spike is caused by lock contention during an index build but the controller attributes it to a traffic surge, the assembled context may omit schema and DDL information entirely. Second, the controller overhead must be small relative to the reasoning time it saves. An expensive controller that selects marginally better contexts may not justify the overhead in situations where time is critical.

7.3 Expanding the Tuning Boundary

The systems presented in this thesis optimize within the DBMS itself: knobs, indexes, query hints, and their compositions. In practice, a DBMS deployment is embedded in a larger stack that includes the application above, a middleware proxy alongside, and the operating system and hardware below. Expanding the tuning boundary to encompass these layers opens new optimization opportunities.

Tuning Upwards to Application-Level: Existing tuners treat the application as external and do not attempt to influence how those queries arrive at the DBMS. With visibility into the application's data access layer, tuners could re-order queries within transactions or identify query rewriting opportunities (e.g., filter pushdown). However, application-level changes require understanding application semantics (e.g., which columns are consumed downstream) and respecting correctness constraints (e.g., data freshness) that current tuners cannot reason about.

Tuning Middleware for Engine Routing: The middleware proxy (e.g., pgcat [6]) sits between the application and the DBMS and mediates every query. Although Chapter 5 presented cases for adjusting proxy pool sizes and read traffic routing, it does not perform engine-level routing. For instance, a tuned proxy could route OLAP queries to an engine optimized for analytical workloads.

Alternatively, the proxy could maintain synchronized replicas with different configurations, each optimized for a distinct workload fragment.

Tuning Down to OS and Hardware: Extending the configuration space to include OS-level knobs (e.g., I/O scheduler selection) and hardware settings (e.g., CPU frequency governor) would enable co-optimization across the full stack. This could also unlock opportunities from heterogeneous hardware (e.g., GPU-accelerated query engines, tiered storage devices). Deeper hardware visibility would also help AgenticOperator make more informed decisions. For instance, mitigations depend on whether the hardware is degraded (e.g., disk failure) and on overall system stability (i.e., how trustworthy the observations are).

7.4 Explainability

As these systems make increasingly autonomous decisions, operators need to understand *why* a particular configuration or mitigation was chosen. For post-mortems, specific telemetry evidence can justify why a configuration was chosen or proved suboptimal. The harder problem is explaining why a configuration *may* be beneficial *before* deploying it and observing feedback. Although Proto-X's latent space provides a geometric interpretation (e.g., nearby points correspond to similar configurations), translating distances into human-readable explanations remains non-trivial. Similarly, AgenticOperator's hypothesis branches offer a natural audit trail, but they can be long and interleaved with irrelevant information-gathering steps. Developing concise, actionable summaries that highlight key evidence and decision points would improve operator trust and facilitate debugging when the system makes suboptimal choices.

Chapter 8

Concluding Remarks

In this dissertation, we presented techniques to holistically optimize a database management system over its lifetime. Existing tuning pipelines employ specialized tools that each target a narrow slice of the DBMS's configuration space (e.g., knobs, indexes, query hints) and rely on human operators to orchestrate these tools, interpret their recommendations, and decide when and how to act. Although this workflow can improve performance to a limited degree, it cannot achieve fully autonomous optimization for three reasons. First, because individual tuners cannot reason over the entire configuration space, they produce suboptimal configurations for point-in-time snapshots of the deployment. Second, because tuners cannot transfer experience from prior tuning sessions, they struggle to adapt the DBMS when environment changes happen. Third, when deployment anomalies occur, the workflow relies on a human operator to evaluate root cause analyses from diagnostics and then manually orchestrate tools to mitigate them. Consequently, this workflow cannot directly intervene with mitigations.

This dissertation addresses these limitations through distinct notions of similarity. In Chapter 3, we presented Proto-X, a holistic tuner that simultaneously optimizes system knobs, indexes, and query hints for a point-in-time snapshot. Proto-X leverages *action similarity* to organize the combined configuration space into a learned latent space where similar actions form neighborhoods and synthesizes proto-actions to navigate those neighborhoods efficiently. Then, in Chapter 4, we presented Booster, a framework that assists existing tuners in adapting to environment changes. Booster leverages *workload-configuration similarity* to structure historical tuning artifacts into query-level insights, prompts large language models with the most relevant experiences, and composes per-query suggestions into a holistic configuration via beam search. Lastly, in Chapter 5, we presented AgenticOperator, a system that transforms the human-centric workflow into an agentic process capable of directly intervening in response to anomalies. AgenticOperator incrementally evolves a pool of diverse hypothesis branches and mitigation strategies by leveraging *scenario similarity* to contextualize and relate the deployment's context to specific tools from a plug-and-play toolbox.

Taken together, the contributions of this dissertation provide evidence that identifying and exploiting similarity across actions, workloads, configurations, and scenarios enables holistic DBMS optimization throughout a deployment's lifetime that improves performance and reduces adaptation time to environment changes.

Bibliography

[1] pg_hint_plan. https://github.com/17zhangw/pg_hint_plan/tree/parallel_patch, 2023. 3.2, 3.3.2, 3.5, 4.3.3, 4.6

[2] Proto-X Code. https://github.com/17zhangw/protox, 2024. 3.5.1

[3] Openrouter llama 3.1 8b instruct. https://openrouter.ai/meta-llama/llama-3.1-8b-instruct, 2025. (document), 4.3

[4] voyage-3-large. https://blog.voyageai.com/2025/01/07/voyage-3-large/, 2025. 4.2.1, 4.3.1, 4.6.1, 4.7.4

[5] Voyage AI Pricing. https://docs.voyageai.com/docs/pricing, 2025. (document), 4.1

[6] pgcat. https://github.com/postgresml/pgcat, 2026. 5.2, 7.3

[7] 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, 3.6.3

[8] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. The unreasonable effectiveness of llms for query optimization, 2024. URL https://arxiv.org/abs/2411.02862. 6.6

[9] Amazon Web Services. Amazon CloudWatch. https://aws.amazon.com/cloudwatch/, 2024. Accessed: 2026-02-24. 2.4, 5.1.2, 5.2

[10] Marcin Andrychowicz, Anton Raichuk, Piotr Stanczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters for on-policy deep actor-critic methods? A large-scale study. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=nIAxjsniDzg. 3.5.1, 3.6.3

[11] ankane. Dexter: The automatic indexer for postgres, 2025. URL https://github.com/ankane/dexter. 1, 1.2, 2.2.1, 2.3, 3.5.2, 4, 4.1.2, 5.1.3, 5.6.3

[12] 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, sep 2023. ISSN 2150-8097. doi: 10.14778/3611540.3611544. URL https://doi.org/10.14778/3611540.3611544. 2.2.1, 3.1, 3.1.1, 3.5.2, 4.1.2, 4.1.3, 4.5, 4.6.1, 4.6.1, 6.1

[13] Anthropic. Model context protocol, 2025. URL https://modelcontextprotocol.io. 5.3.1, 5.6.4

[14] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 221–230, 2018. doi: 10.1145/3183713.3190662. 3.5

[15] Rico Bergmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. An elephant under the microscope: Analyzing the interaction of optimizer components in postgresql. *Proc. ACM Manag. Data*, 3(1), February 2025. doi: 10.1145/3709659. URL https://doi.org/10.1145/3709659. 4.4.1

[16] Alexander Bianchi, Andrew Chai, Vincent Corvinelli, Parke Godfrey, Jarek Szlichta, and Calisto Zuzarte. Db2une: Tuning under pressure via deep learning. *Proc. VLDB Endow.*, 17(12):3855–3868, August 2024. ISSN 2150-8097. doi: 10.14778/3685800.3685811. URL https://doi.org/10.14778/3685800.3685811. 4.1.3

[17] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 227–238, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930604. doi: 10.1145/1066157.1066184. URL https://doi.org/10.1145/1066157.1066184. (*Rule 1*), 4.4.2, 4.4.2

[18] 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, pages 617–630, 2022. doi: 10.1145/3514221.3517845. URL https://db.cs.cmu.edu/papers/2022/moddm074-butrovich.pdf. 3.4.1

[19] 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. 5

[20] Joyce Cahoon, Wenjing Wang, Yiwen Zhu, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Chengcheng Wan, Alexandra Ciortea, Sreraman Narasimhan, and Subru Krishnan. Doppler: Automated sku recommendation in migrating sql workloads to the cloud. *Proc. VLDB Endow.*, 15(12):3509–3521, aug 2022. ISSN 2150-8097. doi: 10.14778/3554821.3554840. URL https://doi.org/10.14778/3554821.3554840. 6.1

[21] Yash Chandak, Georgios Theocharous, James Kostas, Scott Jordan, and Philip Thomas. Learning action representations for reinforcement learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 941–950. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/chandak19a.html. 3.3.3

[22] Surajit Chaudhuri and Vivek Narasayya. Autoadmin "what-if" index analysis utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, page 367–378, New York, NY, USA, 1998. Association for Computing

Machinery. ISBN 0897919955. doi: 10.1145/276304.276337. URL https://doi.org/10.1145/276304.276337. 2.2.1, 3.3.3, 4, 4.1.2, 4.3.3, 4.6

[23] Surajit Chaudhuri and Vivek Narasayya. Anytime algorithm of database tuning advisor for microsoft sql server. June 2020. URL https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/. 1, 1.2, 2.2.1, 3.5.2, 4, 4.1.2, 4.1.3, 4.6.1, 5.1.3, 5.6.3

[24] Sibei Chen, Ju Fan, Bin Wu, Nan Tang, Chao Deng, Pengyi Wang, Ye Li, Jian Tan, Feifei Li, Jingren Zhou, and Xiaoyong Du. Automatic database configuration debugging using retrieval-augmented language models. *Proc. ACM Manag. Data*, 3(1), February 2025. doi: 10.1145/3709663. URL https://doi.org/10.1145/3709663. 1.3, 2.3, 4.1.3, 4.1.4, 4.2.2, 4.3.1

[25] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. Loger: A learned optimizer towards generating efficient and robust query execution plans. *Proc. VLDB Endow.*, 16(7):1777–1789, March 2023. ISSN 2150-8097. doi: 10.14778/3587136.3587150. URL https://doi.org/10.14778/3587136.3587150. 4.2.3, 4.4.2

[26] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. Leon: A new framework for ml-aided query optimization. *Proc. VLDB Endow.*, 16(9):2261–2273, May 2023. ISSN 2150-8097. doi: 10.14778/3598581.3598597. URL https://doi.org/10.14778/3598581.3598597. 4.4.2

[27] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How ai is upending systems research, 2025. URL https://arxiv.org/abs/2510.06189. 5.1.4

[28] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135. 4.7.1

[29] 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. 2.2.1, 3.2, 3.2.2, 5.3.2, 7.1, 7.1

[30] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013. 3.5, 3.5.2, 3.5.4

[31] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1241–1258, New York, NY, USA, 2019. Association for Computing Machinery.

ISBN 9781450356435. doi: 10.1145/3299869.3324957. URL https://doi.org/10.1145/3299869.3324957. 3.3.3, 4.1

[32] 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, sep 2021. ISSN 2150-8097. doi: 10.14778/3484224.3484234. URL https://doi.org/10.14778/3484224.3484234. 3.5, 4.1.3, 4.6

[33] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. Instance-optimized data layouts for cloud analytics workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 418–431, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457270. URL https://doi.org/10.1145/3448016.3457270. 6.6

[34] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces, 2016. 3.1.3, 3.2.2

[35] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. Combining small language models and large language models for zero-shot nl2sql. *Proc. VLDB Endow.*, 17(11):2750–2763, July 2024. ISSN 2150-8097. doi: 10.14778/3681954.3681960. URL https://doi.org/10.14778/3681954.3681960. 4.3.1, 5.1.4, 6.5

[36] Kai Franz, Samuel I Arch, Denis Hirn, Torsten Grust, Todd Mowry, and Andrew Pavlo. Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL. In *CIDR 2024, Conference on Innovative Data Systems Research*, 2024. 3.5, 4.6

[37] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024. URL https://arxiv.org/abs/2312.10997. 4.1.4

[38] 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. 2.3

[39] Victor Giannakouris and Immanuel Trummer. λ-tune: Harnessing large language models for automated database system tuning. *Proc. ACM Manag. Data*, 3(1), February 2025. doi: 10.1145/3709652. URL https://doi.org/10.1145/3709652. 4, 4.1, 4.1.2, 4.1.3, 4.1.4, 4.3.3, 4.6.1, 6.1

[40] Aaron Grattafiori et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783. 4.6.1, 4.7.4

[41] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. The rlr-tree: A reinforcement learning based r-tree for spatial data. *Proc. ACM Manag. Data*, 1(1), may 2023. doi: 10.1145/3588917. URL https://doi.org/10.1145/3588917. 6.6

[42] Roman Heinrich, Manisha Luthra, Johannes Wehrstein, Harald Kornmayer, and Carsten

Binnig. How good are learned cost models, really? insights from query optimization tasks. *Proc. ACM Manag. Data*, 3(3), June 2025. doi: 10.1145/3725309. URL https://doi.org/10.1145/3725309. 4.4.1

[43] Benjamin Hilprecht and Carsten Binnig. Zero-shot cost models for out-of-the-box learned cost prediction. *Proc. VLDB Endow.*, 15(11):2361–2374, jul 2022. ISSN 2150-8097. doi: 10.14778/3551793.3551799. URL https://doi.org/10.14778/3551793.3551799. 2.3, 4.4.1

[44] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006. doi: 10.1126/science.1127647. URL http://www.ncbi.nlm.nih.gov/sites/entrez?db=pubmed&uid=16873662&cmd=showdetailview&indexed=google. 3.3.3

[45] 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, 6.2

[46] Shiyue Huang, Ziwei Wang, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. Dbpa: A benchmark for transactional database performance anomalies. *Proc. ACM Manag. Data*, 1(1), May 2023. doi: 10.1145/3588926. URL https://doi.org/10.1145/3588926. 1.3, 2.4, 5, 5.1.1, 5.1.1

[47] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Tieying Zhang, Jianjun Chen, Hong Chen, and Cuiping Li. E2etune: End-to-end knob tuning via fine-tuned generative language model, 2025. URL https://arxiv.org/abs/2404.11581. 1.2, 2.2.1, 2.2.3, 2.3, 4, 4.1, 4.1.2, 4.1.4, 4.3.3, 4.7.1, 6.1

[48] Yuxuan Huang, Yihang Chen, Haozheng Zhang, Kang Li, Huichi Zhou, Meng Fang, Linyi Yang, Xiaoguang Li, Lifeng Shang, Songcen Xu, Jianye Hao, Kun Shao, and Jun Wang. Deep research agents: A systematic examination and roadmap. *arXiv preprint arXiv:2506.18096*, 2025. 5.3.1

[49] HypoPG. hypopg: Hypothetical indexes for postgresql, 2025. URL https://github.com/HypoPG/hypopg. 3.5, 3.5.2, 4.6

[50] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, October 2002. ISSN 1046-8188. doi: 10.1145/582415.582418. URL https://doi.org/10.1145/582415.582418. 5.6.1

[51] Jiechuan Jiang and Zongqing Lu. Generative exploration and exploitation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):4337–4344, Apr. 2020. doi: 10.1609/aaai.v34i04.5858. URL https://ojs.aaai.org/index.php/AAAI/article/view/5858. 3.4.4

[52] Bowen Jin, Jinsung Yoon, Jiawei Han, and Sercan O. Arik. Long-context llms meet rag: Overcoming challenges for long inputs in rag, 2024. URL https://arxiv.org/abs/2410.05983. 4.3.2

[53] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas C. Müller, Carlo Curino, and Shivaram Venkataraman. Llamatune: Sample-efficient dbms configuration tuning. In *VLDB 2022*, March 2022. URL https://www.microsoft.com/en-us/research/publication/llamatune-sample-efficient-dbms-configuration-tuning/. 2.2.3, 3.3.1, 4, 4.1, 6.1

[54] Mahmut KAYA and Hasan Şakir BİLGE. Deep metric learning: A survey. *Symmetry*, 11(9), 2019. ISSN 2073-8994. doi: 10.3390/sym11091066. URL https://www.mdpi.com/2073-8994/11/9/1066. 3.3.3

[55] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088. 4.3.2

[56] Jan Kossmann and Rainer Schlosser. Self-driving database systems: a conceptual approach. *Distrib. Parallel Databases*, 38(4):795–817, December 2020. doi: 10.1007/s10619-020-07288-w. 4.1

[57] 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 index 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. 3.5.2, 4, 4.1, 4.6.1

[58] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. SWIRL: selection of workload-aware indexes using reinforcement learning. In Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang, editors, *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 2:155–2:168. OpenProceedings.org, 2022. doi: 10.48786/edbt.2022.06. URL https://doi.org/10.48786/edbt.2022.06. 2.3, 4, 4.1.2

[59] A. Lawrence. The ghost in the machine, 2002. 3.1.3

[60] le0pard. pgtune, 2025. URL https://github.com/le0pard/pgtune. 1, 1.2, 2.2.1, 2.3, 3.5.2, 4, 4.1, 4.1.2, 4.1.3, 4.6.1, 5.1.3, 16

[61] Kukjin Lee, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. Analyzing the impact of cardinality estimation on execution plans in microsoft sql server. *Proc. VLDB Endow.*, 16(11):2871–2883, aug 2023. ISSN 2150-8097. doi: 10.14778/3611479.3611494. URL https://doi.org/10.14778/3611479.3611494. 6.6

[62] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. Is your learned query optimizer behaving as you expect? a machine learning perspective. *Proc. VLDB Endow.*, 17(7):1565–1577, March 2024. ISSN 2150-8097. doi: 10.14778/3654621.3654625. URL https://doi.org/10.14778/3654621.3654625. 3.5.3, 4.6.1

[63] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, nov 2015. ISSN 2150-8097. doi: 10.14778/2850583.2850594. URL https://doi.org/10.14778/2850583.2850594. 3.1.1, 3.5, 4.4.2, 4.6

[64] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf. 2.3, 4.1.3, 4.1.4, 4.1.4, 4.2.2, 4.3.2, 5.2.1

[65] Beibin Li, Yao Lu, and Srikanth Kandula. Warper: Efficiently adapting learned cardinality estimators to data and workload drifts. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1920–1933, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526179. URL https://doi.org/10.1145/3514221.3526179. 2.3, 4.1.1, 6.2

[66] 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, aug 2019. ISSN 2150-8097. doi: 10.14778/3352063.3352129. URL https://doi.org/10.14778/3352063.3352129. 1.2, 2.3, 4.1.3, 4.1.3

[67] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. Llm-r2: A large language model enhanced rule-based rewrite system for boosting query efficiency. *Proc. VLDB Endow.*, 18(1):53–65, September 2024. ISSN 2150-8097. doi: 10.14778/3696435.3696440. URL https://doi.org/10.14778/3696435.3696440. 2.2.1, 4.1.3, 4.1.3, 4.1.4, 4.2.1

[68] 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. URL https://db.cs.cmu.edu/papers/2023/p27-lim.pdf. 1, 2.2, 2.2.2, 3.1.2, 3.2, 4.2.2, 4.4.2, 5.2, 5.4.2

[69] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel 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, July 2024. ISSN 2150-8097. doi: 10.14778/3681954.3682030. URL https://doi.org/10.14778/3681954.3682030. 2.3, 6.3

[70] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/. 4.3.2, 5.1.2, 7.2

[71] 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, pages 631–645, 2018. doi: 10.1145/3183713.3196908. URL https://db.cs.cmu.edu/papers/2018/mod435-maA.pdf. 2.1, 3.1, 6.2

[72] 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. 2.2, 3.6.3, 3.6.7, 4.2.2, 4.5, 5.2, 5.4.2

[73] 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, pages 1248–1261, 2021. doi: 10.1145/3448016.3457276. URL https://db.cs.cmu.edu/papers/2021/ma-sigmod2021.pdf. 2.1, 3.1, 3.2, 3.3.3, 3.5.1, 4.4.1, 6.3

[74] Ryan Marcus. Learned query superoptimization, 2023. 3.1.3

[75] 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. 3.1, 3.5.1, 3.6.3, 4.4.1, 6.3

[76] 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, jul 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342644. URL https://doi.org/10.14778/3342263.3342644. 2.2.1, 3.3.3, 6.6

[77] 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, 2.2.1, 3.1, 3.2, 3.4.4, 4.4.2, 4.5, 6.1, 6.6

[78] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey, 2025. URL https://arxiv.org/abs/2402.06196. 4.1.4

[79] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark, 2023. URL https://arxiv.org/abs/2210.07316. 4.7.4

[80] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. Robust query driven cardinality estimation under changing workloads. *Proc. VLDB Endow.*, 16(6):1520–1533, apr 2023. ISSN 2150-8097. doi: 10.14778/3583140.3583164. URL https://doi.org/10.14778/3583140.3583164. 6.6

[81] OpenAI. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276. 4.1.2, 4.6.1

[82] OpenAI. New embedding models and api updates. https://openai.com/index/new-embedding-models-and-api-updates/, 2024. 4.7.4

[83] OpenAI. Openai o3-mini. https://openai.com/index/openai-o3-mini/, 2025. 4.7.4

[84] Biao Ouyang, Yingying Zhang, Hanyin Cheng, Yang Shu, Chenjuan Guo, Bin Yang,

Qingsong Wen, Lunting Fan, and Christian S. Jensen. Rcrank: Multimodal ranking of root causes of slow queries in cloud database systems. *Proc. VLDB Endow.*, 18(4): 1169–1182, May 2025. ISSN 2150-8097. doi: 10.14778/3717755.3717774. URL https://doi.org/10.14778/3717755.3717774. 1.3, 2.4, 4.3.1, 5, 5.1.2, 6.5

[85] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560. 7.2

[86] 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. URL https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf. 1, 2, 2.1, 3.1, 4.1, 5.1

[87] Andrew Pavlo, Matthew Butrovich, Lin Ma, Wan Shen Lim, Prashanth Menon, 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, 2021. URL https://db.cs.cmu.edu/papers/2021/p3211-pavlo.pdf. 2.1, 3, 3.1, 3.2.1, 3.3.1, 3.3.2, 3.5.1, 5.1

[88] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., USA, 1984. ISBN 0201055945. 4.2.3, 4.4.2

[89] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. Hmab: Self-driving hierarchy of bandits for integrated physical database design tuning. *Proc. VLDB Endow.*, 16(2):216–229, oct 2022. ISSN 2150-8097. doi: 10.14778/3565816.3565824. URL https://doi.org/10.14778/3565816.3565824. 6.1

[90] Deepak Babu Piskala, Vijay Raajaa, Sachin Mishra, and Bruno Bozza. Optiroute dynamic llm routing and selection based on user preferences: Balancing performance, cost, and ethics. *International Journal of Computer Applications*, 186(51):1–7, November 2024. ISSN 0975-8887. doi: 10.5120/ijca2024924172. URL http://dx.doi.org/10.5120/ijca2024924172. 4.7.4

[91] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3406703. URL https://doi.org/10.1145/3394486.3406703. 4.7.1

[92] Aniket Rege, Aditya Kusupati, Sharan Ranjit S, Alan Fan, Qingqing Cao, Sham Kakade, Prateek Jain, and Ali Farhadi. Adanns: A framework for adaptive semantic search. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 76311–76335. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/f062da1973ac9ac61fc6d44dd7fa309f-Paper-Conference.pdf. 4.2.1

[93] Matthew Renze and Erhan Guven. The benefits of a concise chain of thought on problem-solving in large language models. In *2024 2nd International Conference on Foundation and Large Language Models (FLLM)*, page 476–483. IEEE, November 2024. doi: 10.1109/fllm63129.2024.10852493. URL http://dx.doi.org/10.1109/FLLM63129.2024.10852493. 5.1.4

[94] Ibrahim Sabek and Tim Kraska. The case for learned in-memory joins. *Proc. VLDB Endow.*, 16(7):1749–1762, may 2023. ISSN 2150-8097. doi: 10.14778/3587136.3587148. URL https://doi.org/10.14778/3587136.3587148. 6.6

[95] Jiachen Shi, Gao Cong, and Xiao-Li Li. Learned index benefits: Machine learning based index performance estimation. *Proc. VLDB Endow.*, 15(13):3950–3962, sep 2022. ISSN 2150-8097. doi: 10.14778/3565838.3565848. URL https://doi.org/10.14778/3565838.3565848. 6.3

[96] 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. 6.1, 6.4, 7.1

[97] 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, jun 2022. ISSN 2150-8097. doi: 10.14778/3547305.3547309. URL https://doi.org/10.14778/3547305.3547309. 3.1, 3.1.2, 3.2.1, 3.5.1, 4, 4.1, 4.1.2, 6.3

[98] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, 3 edition, 1996. ISBN 978-0-262-19374-0. 3.1.3

[99] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *VLDB*, pages 1081–1092, 2006. 4

[100] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153, October 2018. ISSN 2150-8097. doi: 10.14778/3282495.3282500. URL https://doi.org/10.14778/3282495.3282500. 4.4.1

[101] Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=iyJOUELYir. 4.7.4

[102] Xiu Tang, Wenhao Liu, Sai Wu, Chang Yao, Gongsheng Yuan, Shanshan Ying, and Gang Chen. Queryartisan: Generating data manipulation codes for ad-hoc analysis in data lakes. *Proc. VLDB Endow.*, 18(2):108–116, 2024. URL https://www.vldb.org/pvldb/vol18/p108-yao.pdf. 4.1.4, 4.2.1

[103] Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. Learned offline query planning via bayesian optimization, 2025. URL https://arxiv.

org/abs/2502.05256. 4.1.4

[104] The Transaction Processing Council. TPC-C Benchmark (Revision 5.11.0), February 2010. URL https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf. 3.5

[105] The Transaction Processing Council. TPC-E Benchmark, April 2015. URL https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-e_v1.14.0.pdf. 5.5

[106] 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. 3.1.2, 3.5, 4.1.1, 4.6, 4.6.4

[107] 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. (document), 3.1.2, 3.2.1, 3.5, 3.3.2, 3.5, 4.1.1, 4.1.3, 4.6

[108] Kapil Vaidya, Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. Leveraging query logs and machine learning for parametric query optimization. *Proc. VLDB Endow.*, 15(3):401–413, 2021. doi: 10.14778/3494124.3494126. URL http://www.vldb.org/pvldb/vol15/p401-vaidya.pdf. 6.1

[109] 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, pages 1009–1024, 2017. URL https://db.cs.cmu.edu/papers/2017/p1009-van-aken.pdf. 1, 2.2.1, 2.3, 3.1, 3.2, 3.4.1, 3.5.1, 4, 4.1, 4.1.2, 4.1.3, 4.1.3, 4.7.3, 5.1.3, 6.1

[110] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. Why tpc is not enough: An analysis of the amazon redshift fleet. *Proc. VLDB Endow.*, 17(11):3694–3706, July 2024. ISSN 2150-8097. doi: 10.14778/3681954.3682031. URL https://doi.org/10.14778/3681954.3682031. 1.2, 2.3, 4.1.1, 5.3.2

[111] 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, 1.1, 3.1.1, 3.2, 3.2.2, 3.3.3, 3.4.4, 3.5.2, 4.1.1, 4.1.2, 4.5, 6.1, 7.1

[112] Pengyi Wang, Sibei Chen, Ju Fan, Bin Wu, Nan Tang, and Jian Tan. Andromeda: Debugging database performance issues with retrieval-augmented large language models. In *Companion of the 2025 International Conference on Management of Data*, SIGMOD/PODS '25, page 243–246, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715648. doi: 10.1145/3722212.3725080. URL https://doi.org/10.1145/3722212.3725080. 5.1.1, 5.1.2, 5.1.4, 6.5

[113] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw. 5.1.3

[114] Zijia Wang, Haoran Liu, Chen Lin, Zhifeng Bao, Guoliang Li, and Tianqing Wang. Leveraging dynamic and heterogeneous workload knowledge to boost the performance of index advisors. *Proc. VLDB Endow.*, 17(7):1642–1654, March 2024. ISSN 2150-8097. doi: 10.14778/3654621.3654631. URL https://doi.org/10.14778/3654621.3654631. 6.1

[115] 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. 2.3, 6.4

[116] 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. 4, 4.1, 6.1

[117] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. Stage: Query execution time prediction in amazon redshift. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, page 280–294, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704222. doi: 10.1145/3626246.3653391. URL https://doi.org/10.1145/3626246.3653391. 3.1.1, 3.2

[118] Mengyi Yan, Yaoshu Wang, Yue Wang, Xiaoye Miao, and Jianxin Li. Gidcl: A graph-enhanced interpretable data cleaning framework with large language models. *Proc. ACM Manag. Data*, 2(6), December 2024. doi: 10.1145/3698811. URL https://doi.org/10.1145/3698811. 4.1.4, 4.2.1

[119] An Yang et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388. 4.7.4

[120] 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. 6.6

[121] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 11809–11822. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/271db9922b8d1f4dd7aaef84ed5ac703-Paper-Conference.pdf. 5.3.3

[122] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629. 5.5.1

[123] Zhiming Yao, Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. A query optimization method utilizing large language models, 2025. URL https://arxiv.org/abs/2503.

06902. 4.1, 4.1.4, 6.1

[124] Dong Young Yoon, Ning Niu, and Barzan Mozafari. DBSherlock: a performance diagnostic tool for transactional databases. In *SIGMOD*, pages 1599–1614, 2016. ISBN 978-1-4503-3531-7. 2.4, 5.1.2

[125] Geoffrey X. Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. Blueprinting the cloud: Unifying and automatically optimizing cloud data infrastructures with brad. *Proc. VLDB Endow.*, 17(11):3629–3643, July 2024. ISSN 2150-8097. doi: 10.14778/3681954.3682026. URL https://doi.org/10.14778/3681954.3682026. 2.2.1

[126] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. Automatic view generation with deep learning and reinforcement learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1501–1512, 2020. doi: 10.1109/ICDE48307.2020.00133. 2.2.1

[127] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. Automatic view generation with deep learning and reinforcement learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1501–1512, 2020. doi: 10.1109/ICDE48307.2020.00133. 6.1

[128] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf. 3.4.1

[129] Eleni Zapridou, Ioannis Mytilinis, and Anastasia Ailamaki. Dalton: Learned partitioning for distributed data streams. *Proc. VLDB Endow.*, 16(3):491–504, 2022. URL https://www.vldb.org/pvldb/vol16/p491-zapridou.pdf. 6.1

[130] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi: 10.1145/3299869.3300085. URL https://doi.org/10.1145/3299869.3300085. 2.2.1, 3.1, 3.2, 3.2.2, 3.5.1, 3.6.3, 4, 4.1, 4.1.2, 6.1

[131] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. Instruction tuning for large language models: A survey, 2024. URL https://arxiv.org/abs/2308.10792. 4.1.4, 4.1.4

[132] William Zhang, Wan Shen Lim, Matthew Butrovich, and Andrew Pavlo. The holon approach for simultaneously tuning multiple components in a self-driving database management system with machine learning via synthesized proto-actions. *Proc. VLDB Endow.*, 17(11):3373–3387, 2024. URL https://www.vldb.org/pvldb/vol17/p3373-zhang.pdf.

4, 4.1, 4.1.2, 4.1.3, 4.1.4, 4.2, 4.2.3, 4.4.2, 4.6.1, 4.6.1, 4.7.3, 5, 5.1.1, 5.1.1

[133] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. Facilitating database tuning with hyper-parameter optimization: A comprehensive experimental evaluation. *Proc. VLDB Endow.*, 15(9):1808–1821, may 2022. ISSN 2150-8097. doi: 10.14778/3538598.3538604. URL https://doi.org/10.14778/3538598.3538604. 4.1.3

[134] 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), jun 2023. doi: 10.1145/3589331. URL https://doi.org/10.1145/3589331. 1, 1.1, 2.2.2, 2.3, 3, 3.1.1, 3.1.2, 3.2, 3.2.2, 3.3.3, 3.4.1, 3.4.1, 3.4.4, 3.5.2, 4, 4.1.3, 4.5, 4.6.1, 6.1

[135] 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, nov 2023. ISSN 2150-8097. doi: 10.14778/3632093.3632114. URL https://doi.org/10.14778/3632093.3632114. 3.2.1, 3.3.1, 4.1.3

[136] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. Reactable: Enhancing react for table question answering. *Proc. VLDB Endow.*, 17(8):1981–1994, April 2024. ISSN 2150-8097. doi: 10.14778/3659437.3659452. URL https://doi.org/10.14778/3659437.3659452. 4.1.4, 4.2.1

[137] Fuheng Zhao, Shaleen Deep, Fotis Psallidas, Avrilia Floratou, Divyakant Agrawal, and Amr El Abbadi. Sphinteract: Resolving ambiguities in nl2sql through user interaction. *Proc. VLDB Endow.*, 18(4):1145–1158, May 2025. ISSN 2150-8097. doi: 10.14778/3717755.3717772. URL https://doi.org/10.14778/3717755.3717772. 6.5

[138] 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. 6.4

[139] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In Yixin Cao, Yang Feng, and Deyi Xiong, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-demos.38. URL https://aclanthology.org/2024.acl-demos.38/. 4.7.1

[140] Wei Zhou, Peng Sun, Xuanhe Zhou, Qianglei Zang, Ji Xu, Tieying Zhang, Guoliang Li, and Fan Wu. Dbaiops: A reasoning llm-enhanced database operation and maintenance system using knowledge graphs, 2025. URL https://arxiv.org/abs/2508.01136. 5.1.2, 5.1.2

[141] 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. ISSN 2150-8097. doi: 10.14778/3397230.3397238. URL https://doi.org/10.14778/3397230.3397238. 6.3

[142] Xuanhe Zhou, Lianyuan Jin, Ji Sun, Xinyang Zhao, Xiang Yu, Jianhua Feng, Shifu Li, Tianqing Wang, Kun Li, and Luyang Liu. Dbmind: a self-driving platform in opengauss. *Proc. VLDB Endow.*, 14(12):2743–2746, July 2021. doi: 10.14778/3476311.3476334. 4.1

[143] 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. 6.1

[144] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. D-bot: Database diagnosis system using large language models. *Proc. VLDB Endow.*, 17(10):2514–2527, June 2024. ISSN 2150-8097. doi: 10.14778/3675034.3675043. URL https://doi.org/10.14778/3675034.3675043. 1.3, 5, 5.1.2, 5.3.1, 6.5

[145] Jiaqi Zhu, Shaofeng Cai, Fang Deng, Beng Chin Ooi, and Wenqiao Zhang. Meter: A dynamic concept adaptation framework for online anomaly detection. *Proc. VLDB Endow.*, 17(4):794–807, December 2023. ISSN 2150-8097. doi: 10.14778/3636218.3636233. URL https://doi.org/10.14778/3636218.3636233. 5