Self-Driving Database Management Systems: Forecasting, Modeling, and Planning

Lin Ma

CMU-CS-21-134

August 2021

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

Thesis Committee
Andrew Pavlo, Chair
Ruslan Salakhutdinov
Gregory Ganger
Christopher Ré, Stanford University

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

This research was sponsored by National Science Foundation award numbers: IIS-1846158, IIS-1423210, and CCF-1438955; by a National Science Foundation Graduate Research Fellowship award; by the Intel Corporation award number 1011856; and by the University Industry Research Corporation award numbers: 1030834 and 1030845. 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. This research was sponsored by Intel ISTC-CC and the National Science Foundation under award numbers CCF-1438955 and IIS-1846158. 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, self-driving databases, machine learning, artificial intelligence

For my family

Abstract

Database management systems (DBMSs) are an important part of modern data-driven applications. However, they are notoriously difficult to deploy and administer because they have many aspects that one can change that affect their performance, including database physical design and system configuration. There are existing methods that recommend how to change these aspects of databases for an application. But most of them require humans to make final decisions on what changes to apply and when to apply them. Furthermore, these previous tuning methods either (1) require expensive exploratory testing, (2) are reactionary to the workload and can only solve problems after they occur, (3) focus only on improving one single aspect of the DBMS, or (4) do not provide explanations on their decisions. Thus, most DBMSs today still require onerous and costly human administration.

In this thesis, we present a novel architecture for a self-driving DBMS that enables automatic system management and removes the administration impediments. Our approach consists of three frameworks: (1) workload forecasting, (2) behavior modeling, and (3) action planning. The workload forecasting framework predicts the query arrival rates under varying database workload patterns using an ensemble of time-series forecasting models. The behavior modeling framework constructs fine-grained machine learning models that predict the runtime behavior of the DBMS. Lastly, the action planning framework generates a sequence of optimization actions based on these forecasted workload patterns and behavior model estimations. It uses receding horizon control and Monte Carlo tree search to approximate the complex optimization problem effectively.

Our forecasting-modeling-planning architecture enables an autonomous DBMS that proactively plans for optimization actions without expensive testing. It automatically applies the actions at proper times, holistically controls all system aspects, and provides explanations on its decisions.

Acknowledgments

I am very fortunate to have Andrew (Andy) Pavlo as my advisor throughout my Ph.D. career. Andy always gave me his best help when I needed them. Andy was never tired of discussing research topics. His passion for databases and humbleness towards science always inspired me. He taught me not only how to navigate technical challenges in database research but also how to write elegant papers, make compelling presentations, light the spirit of students, and manage a professional career. I learned a lot from him. And I am truly grateful for the experience working with him over these years.

I would like to thank all my thesis committee members. Thanks to Ruslan (Russ) Salakhutdinov for the many inspiring discussions we have, especially when we were designing the system's high-level self-driving architecture. Russ always brought genuine opinions and vivid explanations to connect the bridge between system design and ML methods. Thanks to Christopher (Chris) Ré and Gregory (Greg) Ganger for their feedback during the course of this project. Chris has sharp observations. His insights provided exciting possibilities to enhance our self-driving DBMS. Greg shares valuable research experience on the interactions between ML and systems. The constructive suggestions from him shepherded my system design.

I am thankful to all of the collaborators on this project. William Zhang provided vital help in implementing our behavior modeling framework (Chapter 4). Katrina (Jie) Jiao aided me in building the infrastructure for our action planning framework (Chapter 5). Ahmed Hefny guided me through the modeling challenge of our workload forecasting framework (Chapter 3). Thanks to Alex Smola and Geoffrey Gordon for their early feedbacks. Also, thanks to other collaborators on this project: Joy Arulraj, Dana Van Aken, Prashanth Menon, Matt Butrovich, Wan Shen Lim, Lisa Lee, Gustavo Mezerhane, and Wuwen Wang.

I would like to extend my thanks to my mentors during my internship at Microsoft Research: Sudipto Das, Bailu Ding, and Adith Swaminathan. All of them are brilliant researchers. Even though they were not directly involved in this project, the research skills they taught me during our active learning project significantly helped this work on self-driving DBMSs.

I want to thank the many talented researchers and students that overlapped with me at CMU: Laxman Dhulipala, Charles McGuffey, Huanchen Zhang, Tianyu Li, Matthew Perron, Viktor Leis, Marcel Kost, Ziqi Wang, Michael (Tieying) Zhang, Yingjun Wu, Xiaoxuan Liu, Sivaprasad Sudhir, Chad Dougherty, Todd Mowry, David Andersen, Michael Kaminsky, Ricky (Chen) Xu, John Rollinson, Amadou Ngom, Joseph Koshakow, Kyle Dotterrer, Rohan Aggarwal. All of you inspired me to explore further and deeper in my research. I would also like to thank all the administrative staff that helped with many tasks during my graduate school, including Deb Cavlovich, Catherin Copetas, Jessica Parker, Karen Lindenfelser, and Joan Digney.

This thesis would not be possible without my family's unconditional support and love, which I will always remember. I want to dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The idea of using a DBMS to remove the burden of data management from application developers was one of the original selling points of the relational model and declarative query languages from the 1970s [200]. With this approach, a developer only writes a query that specifies what data they want to access. The DBMS then finds the most efficient way to store and retrieve data, and safely interleave operations.

Over five decades later, DBMSs are now the critical part of every data-intensive application in all facets of society, including government, business, and science. These systems are also more complicated now with a long and growing list of functionalities. For example, the number of knobs for different versions of MySQL and Postgres DBMS increased by $3\times$ and $6\times$ respectively over the past 20 years [215]. An effective DBMS deployment requires not only tuning these knobs but also deciding the database's physical design [126], such as indexes and table partitions. Properly tuning these aspects of a DBMS is an involved and complex task. For example, selecting the best set of indexes for a database workload can be NP-complete [164]. Furthermore, it is challenging to decide when and how to apply changes to a DBMS. For example, an index creation may compete for resources with workload queries and slow them down, especially when the workload volume is high. And allocating the appropriate amount of resources for such index creation has an intricate trade-off: using a small amount of resource (e.g., one CPU core) may mitigate the resource contention to the queries but make the index creation slower; using more resources (e.g., eight CPU cores) may increase the contention, but the index creation can finish faster and accelerate the queries sooner. Such complexity makes database tuning a major contributing factor to DBMSs' high total cost of ownership. Personnel is estimated to be almost 50% of the total ownership cost of a DBMS [174]. And a survey suggests that 73% of database administrators (DBAs) [197] think that performance tuning occupies most of their time [170].

Given such complications and costs of deploying a database, people have been dreaming about DBMSs that can automatically optimize themselves for years [128]. Much of the previous work on self-tuning systems focuses on standalone tools that target only a single aspect of the database. For example, some tools can choose the best physical design of a database [45], such as indexes [46, 92, 214], materialized views [18], partitioning schemes [17, 167], or storage layout [24]. Other tools are able to select the tuning parameters for an application [60, 202, 206, 215]. Most of these tools operate in the same way: a DBA provides a sample database and workload trace that guides a search process to find an optimal or near-optimal configuration. This is still an onerous process, however, as it requires laborious preparation of workload samples, spare hardware to test proposed updates, and above all else, intuition into the DBMS's internals. DBAs need to select the best set of updates among all recommendations, decide when to apply them (e.g., during the day or at night) and how to apply them (e.g., using one core or eight cores), and finally apply the changes. All of the major DBMS vendors' tools, including Oracle [62, 123], Microsoft [45, 151], and IBM [201, 211], operate in this manner.

These tools are insufficient for achieving a completely autonomous system because they are (1) external to the DBMS, (2) reactionary, or (3) not able to take a holistic view that considers more than one problem at a time. That is, they observe the DBMS's behavior from outside of the system, so they cannot provide accurate estimations or explanations on the effect of their recommendations. They also only advise the DBAs on how to make corrections to fix only one aspect of the system at a time. Therefore, the DBAs need to repeat the tuning process for all system aspects and address their interactions, such as resource competitions. Moreover, they address problems after they occur but do not proactively prepare the system based on the workload patterns, such as dropping unnecessary indexes ahead of time when the DBMS is approaching the system's storage/memory limit. And these tuning tools assume that the human operating them is knowledgeable enough to update the DBMS during a proper time window when it will have the least impact on applications. For example, the DBAs may need to update the system when the workload volume is low (e.g., during midnight) to avoid interrupting the normal business operation. The database landscape has changed significantly in the last decade: there are an increasing number of types of databases (e.g., transactional, analytical, or hybrid of both) with a growing list of functionalities and configuration options. Thus, one cannot assume that a DBMS is always deployed by an expert that understands the intricacies of database optimization. Furthermore, modern software engineering practices, such as the adoption of DevOps [76], advocate combining the application development and IT operations, including DBMS administrations, to shorten the development life cycle. Therefore, development teams are often comprised of fewer specialized DBAs despite the increasing DBMS complicacy.

There is a recent push from cloud vendors to support more automated tuning that directly applies tuning actions, such as automatic indexing [58, 226], automatic knob tuning [132, 244], and dynamic resource allocation [57, 155]. But these methods again only focus on solving one problem and are mostly reactive. Furthermore, these methods often require lengthy and expensive testing on copies of databases in the cloud (e.g., testing various candidate indexes or knob configurations to find the best one to apply). Most of these methods cannot provide insightful explanations on their decisions either, such as the expected completion time and the side effect to apply a tuning action. Such explanations are crucial for the system's debuggability and transparency, which are essential for real-world applications.

This thesis addresses the challenge of developing completely autonomous DBMSs that automatically optimize themselves without any human intervention using machine learning (ML) and control theory. We present a new type of DBMS architecture called "self-driving" that enables proactive, efficient, and explainable control of all DBMS aspects.

We use an analogy to self-driving cars to explain our design for the self-driving DBMS architecture. A simplified self-driving car architecture consists of (1) a perception system, (2) mobility models, and (3) a decision-making system [156]. The *perception system* observes the

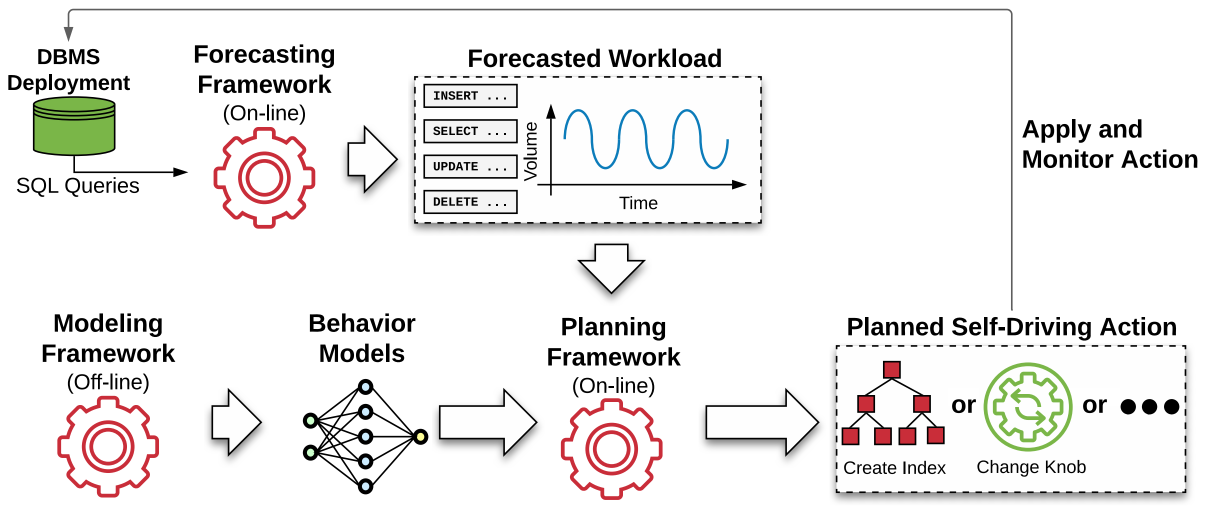


Figure 1.1: Self-Driving DBMS Architecture – Our architecture consists of an on-line workload forecasting framework, an off-line behavior modeling framework, and an on-line action planning framework.

vehicle's surrounding environment and estimates the potential state, such as other vehicles' movements. The *mobility models* approximate a vehicle's behavior in response to control actions in relevant operating conditions. Lastly, the *decision-making system* uses the perception and the models' estimates to select actions to accomplish the driving objectives.

As illustrated in Figure 1.1, our design of a self-driving DBMS architecture consists of three frameworks with analogs to self-driving cars: (1) workload forecasting, (2) behavior modeling, and (3) action planning. The *workload forecasting* and *action planning* are on-line frameworks, whereas the *behavior modeling* is an off-line framework. At runtime, the on-line *workload forecasting* framework helps the system observe and predict the application's future workload (i.e., the arrival rates of the workload's queries over various time horizons). This allows the system to proactively improve the performance before problems occur. The DBMS then uses these forecasts with the behavior models generated by the off-line *behavior modeling* framework to predict the costs and benefits of all types of self-driving actions without expensive exploratory testing. The DBMS provides this information to the on-line *action planning* framework to generate explainable actions (e.g., building an index or changing a knob) that improve the system's target objective function (e.g., latency or throughput). The DBMS then automatically applies and monitors these actions without any human intervention.

This thesis provides evidence to support the following statement:

Thesis Statement: *A self-driving DBMS architecture with workload forecasting, behavior modeling, and action planning frameworks can enable completely autonomous DBMS administration that is proactive, efficient, and explainable.*

We summarize the technical contributions of this thesis as follows:

- We present a workload forecasting framework that predicts the future workload patterns in varying horizons for self-driving DBMSs to proactively choose optimization actions before problems occur (Chapter 3). Our framework uses query templatization, on-line clustering, and ensemble-based time-series forecasting to succinctly predict the future arrival rates of the workload's queries.

- We present a behavior modeling framework that generates behavior models for self-driving DBMSs to estimate the effects of various self-driving actions (Chapter 4). Our framework decomposes the complex DBMS into small and independent tasks to model separately, which reduces the modeling complexity. It generates workload- and dataset-independent models that the self-driving DBMS can use on any production database to avoid the expensive modeling at runtime. These models also provide explanations on the self-driving actions' cost and benefit expectations.
- We present an action planning framework that selects actions for self-driving DBMSs to automatically optimize themselves (Chapter 5). It integrates the workload forecasting and modeling frameworks to generate a sequence of actions tailored to the workload patterns. It uses receding-horizon planning and Monte-Carlo tree search to efficiently approximate the complex optimization problem optimize the system performance holistically.

We next explain the background information of self-driving DBMSs in Chapter 2. We provide a detailed review of related work in Chapter 6 and discuss possible areas for future work in Chapter 7. We discuss our work in the context of the **NoisePage** DBMS [38] that we use to implement our self-driving architecture. NoisePage is an in-memory DBMS that supports hybrid transactional and analytical (HTAP) workloads.

Chapter 2

Background

In this chapter, we discuss our definition of self-driving DBMSs and the related background. We also explain the challenges of building self-driving DBMSs.

2.1 Self-Driving DBMS

The ultimate goal of a self-driving DBMS is to configure, manage, and optimize itself without any human intervention as the database and its workload evolve over time. A key element that guides the DBMS's decision-making is a human-selected *objective function*. An objective function could be either performance metrics (e.g., throughput, latency, availability) or deployment costs (e.g., hardware, hosting, energy). This is akin to a human telling a self-driving car their desired destination. The DBMS must also operate within human-specified constraints, such as memory usage or service-level objectives (SLOs).

The way that the DBMS improves its objective function is by deploying *actions* that it thinks will potentially help it execute the application's workload. These actions control three aspects of the system: (1) *physical design*, (2) *knob configuration*, and (3) *hardware resources*. The first are changes to the database's physical representation and data structures (e.g., indexes, materialized views, partitioning). The second action type are optimizations that affect the DBMS's runtime behavior through its configuration knobs. These knobs can target individual client sessions (e.g., maximum memory consumption allowed per query) or the entire system (e.g., buffer pool size for disk-based systems). Lastly, the resource actions change the hardware resources of the DBMS (e.g., instance type, number of machines); these assume that the DBMS is deployed in an elastic/cloud environment where additional resources are readily available. This thesis focuses on the first two aspects (physical design and knob configuration) because the DBMS prototype that we used to implement our self-driving architecture (NoisePage) does not support resource scaling yet. Nevertheless, we anticipate our proposed architecture can extend to control resource scaling (Section 7.2). Our discussion also focuses on methods that improve the DBMS's efficiency. Automating security protocols in DBMSs is also important, but these techniques are orthogonal to our focus in this thesis.

Although there are recent announcements on self-driving DBMSs [6, 117, 160], there is no standard definition of what it means for a DBMS to be "self-driving". Furthermore, there is con-

Level	Name	Description
0	Manual	The system has no autonomy. The user diagnoses performance problems, creates actions, and applies them manually.
1	Assistant	The system recommends promising actions to the user. The user selects and applies actions.
2	Mixed	The system simultaneously applies actions and notifies the user to make decisions.
3	Local	The system has self-contained components that automatically apply actions for specific aspects of the system.
4	Directed	The system is semi-autonomous. It automatically applies actions but still takes high-level directions from users.
5	Self-Driving	The system is fully autonomous. It applies actions to improve all system aspects holistically without direction, accounts for future workloads, and generates explanations for its decisions.

Table 2.1: Levels of Autonomy – A classification of the levels of autonomy capabilities in DBMSs.

fusion about how the label relates to previous attempts on self-adaptive [93], self-tuning [45], and self-managing [123] DBMSs. To better understand this issue, we next present a taxonomy of the levels of autonomy that a DBMS could support.

2.2 Taxonomy

Achieving full autonomy in a DBMS is years away. There are, however, intermediate levels where a DBMS removes control from humans and takes over more management responsibilities on its own. The amount of autonomy of a self-driving DBMS plays a central role in its design and the user experience. Thus, we propose the following taxonomy on the levels of autonomy that a DBMS can provide. We organize this in increasing levels of complexity of autonomy with decreasing user interaction and control. An overview of these levels is shown in Table 2.1.

Level 0: At the lowest level of autonomy, the system provides an interface for the user to operate in a *manual* fashion. In other words, the DBMS only does exactly what the human instructs it to do. Many open-source DBMSs, such as MySQL [2], PostgreSQL [8], and SQLite [9], do not have built-in functionalities to optimize their configurations. Thus, users need to investigate the DBMS performance and devise actions by themselves unless they use external tools (discussed in higher autonomy levels) for help.

Level 1: The next level of autonomy provides *assistant* tools that recommend improved configurations to the user for some of the DBMS's sub-systems. The user is responsible for (1) selecting which recommendations to apply, (2) deciding when to apply them, and (3) monitoring the DBMS's behavior afterward. These tools often also require the user to manually prepare sample workloads and/or deploy a second copy of the database for the tools to generate new recommendations [48]. All of the major DBMS vendors have their own proprietary tools of this autonomy level. For example, Microsoft's AutoAdmin recommends database physical design structures, including indexes [46], materialized views [18], and partitioning [17], for a given database and workload. IBM's DB2 Performance Wizard tool asks DBAs questions about their application (e.g., whether it is OLTP or OLAP) and then proposes knob settings based on their answers [125]. Oracle also includes a SQL analyzer tool that estimates the impact of configuration modifications [234]. Moreover, there are external tools for open-source databases that give users tuning advice, such as MySQLTuner [3] and PGTuner [7] for knob tuning and Dexter [1] for index selection. Recently, there are attempts to use ML to help configure specific parts of the DBMS, such as indexes [65], knobs [72, 131, 216, 243], and data layouts [68, 236].

Level 2: This level of autonomy assumes a minimal control system that collaborates with the user to configure some sub-systems of the DBMS. The initiative for making decisions is *mixed* between the DBMS and the user. Such autonomy exposes complex problems with the user experience since the user may issue changes that conflict with the DBMS's control component. For example, the user may disable the autonomy of a particular sub-system, and the DBMS must account for this in its models. One example of this type of system was a prototype for IBM DB2 from 2008 [224]. This system used existing tools [125, 201] in an external controller that triggered a change or sent DBA notifications whenever a resource threshold was surpassed (e.g., buffer pool hit ratio). It still required a human to select optimizations and to occasionally restart the DBMS.

Level 3: The next level consists of *local* autonomy where each sub-system can adapt without human guidance, but there is no higher-level coordination between them or long-term planning. The enabling/disabling of autonomy of these components is controlled by the user. For example, some DBMSs support automatic memory allocations (e.g., Oracle [123], IBM DB2 [201]), automatic resource scaling (e.g., Microsoft Azure SQL [57], Amazon Aurora [218]), or automatic creation/drop of indexes (e.g., Microsoft Azure SQL [58]). We also believe that Oracle's autonomous database-as-a-service offerings are an example of this level [6]. The service requires users to select whether their application has either a transactional or analytical workload. It then uses Oracle's existing assistance tools (Level 1) in a control loop to handle common tuning activities, but they are managed in isolation. The DBMS is also unable to predict long-term workload trends.

Level 4: At the *directed* level, the system manages all its sub-systems, and the user only provides high-level direction for global configuration issues. Such direction includes hints about future workload needs so that the DBMS can prepare for long-term decisions (e.g., capacity planning). For example, the user may manually select actions to begin scaling the DBMS's deployment to prepare for an upcoming surge in traffic. We also anticipate that the DBMS still requires session-specific hints from humans at this level. The DBMS is intelligent enough to recognize when to ask a human for help with tuning problems.

Level 5: Lastly, at the highest level of automation, the DBMS is completely independent (i.e., *self-driving*) because it coordinates across all sub-systems without direction from the user. It accounts for future workloads in its decision-making and supports all tuning activities that do not require an external value judgment. The DBMS also tunes individual queries (e.g., join orderings, index selection) and client sessions (e.g., optimizations, knobs). For each deployed action, the DBMS also provides human-readable explanations about why it made the decision to deploy that action. Whether a self-driving DBMS should still allow humans to configure aspects of the system is up for debate. How the DBMS's planning components would respond to such human input is a difficult and unsolved problem.

This thesis presents a DBMS architecture designed for Level 5 autonomy. We view this architecture as the first step towards a self-driving DBMS that is fully autonomous.

2.3 Challenges

There are recent successes in applying ML for automated decision-making problems, such as self-driving vehicles [168], Go games [192], and trading systems [103]. We leverage modern advances in ML and control theory to build the architecture for self-driving DBMSs (Level 5 autonomy). However, ML is not a panacea. There are challenges in applying ML for self-driving DBMSs because of the characteristics of DBMS systems and database workloads. We now discuss these challenges.

Workload Forecasting: A key property to distinguish self-driving DBMSs from DBMSs with lower autonomy levels is the ability to automatically choose when to apply which actions without any human intervention. Achieving this ability requires accurately predicting the workload in the future so that the system can determine the proper time window to make changes. For example, the system should schedule expensive changes (e.g., creating an index) when the workload volume is low to avoid affecting normal workload operations, such as violating the SLOs. Recall from Section 2.2 that DBMSs with Level 5 autonomy should be able to anticipate the future workloads and act accordingly.

However, forecasting database workload in the future is challenging because modern DBMS applications may execute millions of queries per day [137]. Tracking each of these queries and building forecasting models for all of them can be expensive. Database workloads can also have varying patterns, such as cycles [205], growth [137], and evolution [20]. Not only can there be different workload patterns across applications, but there can also be different patterns that simultaneously exist for subsets of queries in a single application. Effective database workload forecasting should capture all these patterns.

Behavior Modeling: The foundation of self-driving DBMSs' decision-making is the ability to estimate the action behavior: how an action helps the workload, how long an action takes, how much resource an action needs, and how applying an action interferes with the system performance. Only high-fidelity models enable effective and robust control. This is similar to self-driving cars that need accurate mobility models to estimate the effect of their actions, such as hitting the brake or turning the steering wheel [156].

But building such behavior models to accurately make these estimations is challenging as DBMSs are complex software systems. There are various components (e.g., storage, query execution, logging) with diverse configuration options and performance characteristics [95]. database components and queries may also execute concurrently in multi-core environments, which multiplicates the number of possible behavioral states that a self-driving DBMS needs to model. Moreover, modern ML models (e.g., neural networks) need high-quality training data to make accurate inferences [173]. However, collecting training data for self-driving DBMSs can be expensive since many actions can take a long time to finish and consume a significant amount of resources, such as creating indexes on large tables.

Action Planning: The effectiveness of self-driving DBMSs relies on their ability to plan for proper actions based on the workload forecasts and behavior model estimations. Since actions can be expensive, ineffective actions not only consume time and resources but also can interfere with the system's performance. It is also important to arrange the action sequence appropriately. For example, if two actions have similar improvements for a workload, it is more beneficial to apply the faster action first to accelerate the queries earlier.

It is challenging to plan for such a sequence of actions and when to apply them efficiently. There are many possible actions across various aspects of the systems, such as changing knob values and creating indexes. It may also be beneficial to plan for longer sequences to arrange actions more holistically based on the workload pattern. But the number of possible choices of action sequences increases exponentially as the sequence length increases, which exacerbates the planning difficulty. Furthermore, the planning requires the behavior models to estimate the action effect on the forecasted workload. These estimations can also incur high model inference overhead on the system. Lastly, the planning needs to account for system constraints, such as the maximum available memory.

We now discuss how we address each of these challenges in the following chapters.

Chapter 3

Workload Forecasting

To be fully autonomous, the DBMS must be able to predict what the workload will look like in the future. If a self-driving DBMS only considers the behavior of the application in the past when selecting which optimizations to apply, these optimizations may be sub-optimal for the workload in the near future. It can also cause resource contention if the DBMS tries to apply optimizations after the workload has shifted (e.g., it is entering a peak load period). Instead, a self-driving DBMS should choose its optimizations proactively according to the expected workload patterns in the future. But the DBMS's ability to achieve this is highly dependent on its knowledge of the queries and patterns in the application's workload.

Previous work has studied database workload modeling in different contexts. For example, one way is to model the demands of resources for the system, rather than a direct representation of the workload itself [56, 172]. Other methods model the performance of the DBMS by answering "what-if" questions about changes in OLTP workloads [148, 151]. They model the workload as a mixture of different types of transactions with a fixed ratio. There is also work to predict how the workload will shift over time using hidden Markov models [99, 100, 101, 102] or regressions [77, 144]. Earlier work has also modeled database workloads using more formal methods with pre-defined transaction types and arrival rates [181, 182].

All of these methods have deficiencies that make them inadequate for self-driving DBMSs. For example, some use a lossy compression scheme that only maintains high-level statistics, such as average query latency and resource utilization [56, 148, 151, 172]. Others assume that the tool is provided with a static workload [181, 182], or they only generate new models when the workload shifts, thereby failing to capture how the volume of queries and the workload trends change over time [77, 144, 181, 182]. Lastly, some models are hardware and/or database design dependent [99, 100, 101, 102], which means the DBMS has to retrain them whenever its configuration changes.

In this chapter, we present a method to succinctly forecast the workload for self-driving DBMSs. Our approach continuously clusters queries based on the their arrival rate temporal patterns. It seamlessly handles different workload patterns and shifts. It then builds models to predict the future arrival patterns for the query clusters. Such predictions are necessary to enable an autonomous DBMS's planning module to identify optimizations to improve the system's performance and apply them proactively [160]. The key advantage of our approach over previous forecasting methods is that the data we use to train our models is independent

	Admissions	**BusTracker**	**MOOC**
DBMS Type	MySQL	PostgreSQL	MySQL
Num of Tables	216	95	454
Trace Length (Days)	507	58	85
Avg. Queries Per Day	5M	19.9M	1.1M
Num of `SELECT` Queries	2541M [99.8%]	19.5M [98%]	0.97M [88%]
Num of `INSERT` Queries	1.8M [0.07%]	15K [0.8%]	14K [1.3%]
Num of `UPDATE` Queries	2.6M [0.1%]	22K [1%]	66K [6%]
Num of `DELETE` Queries	0.4M [0.02%]	3K [0.2%]	51K [4.7%]

Table 3.1: Sample Workloads – Summarization of the workload traces collected from the database applications described in Section 3.1.1.

of the hardware and the database design. Thus, it is not necessary to rebuild the models if the DBMS's hardware or configuration settings change.

To evaluate our forecasting models, we integrated this framework into both MySQL [2] and PostgreSQL [8], and measure its ability to model and optimize three real-world database applications. The results demonstrate that our framework can efficiently forecast the expected future workload with only a minimal loss in accuracy. They also show how a self-driving DBMS can use this framework to improve the system's performance.

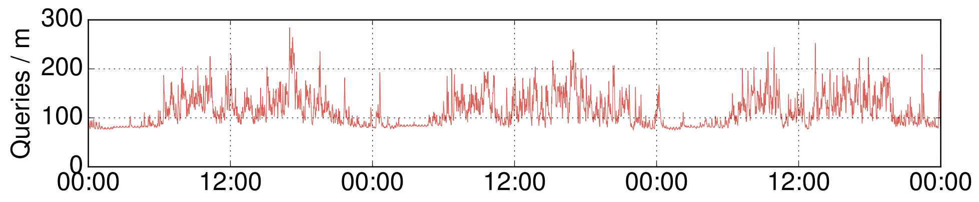
The remainder of this chapter is organized as follows. Section 3.1 discusses common workload patterns in database applications. We next give an overview of our approach in Section 3.2 and then present the details of its components: Pre-Processor (Section 3.3), Clusterer (Section 3.4), and Forecaster (Section 3.5). We provide an experimental analysis of our methods in Section 3.6.

3.1 Background

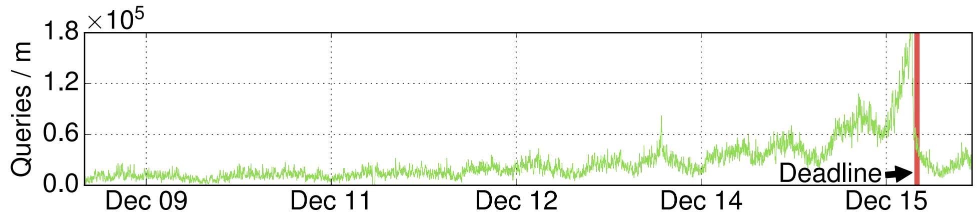
The goal of workload forecasting in an autonomous DBMS is to enable the system to predict what an application's workload will look like in the future. This is necessary because the workloads for real-world applications are never static. The system can then select the optimizations to prepare based on this prediction.

We contend that there are two facets of modern database applications that make robust, unsupervised workload forecasting challenging. The first is that an application's queries may have vastly different arrival rates. Thus, an effective forecasting model must be able to identify and characterize each of these arrival rate patterns. The second is that the composition and volume of the queries in an application's workload change over time. If the workload deviates too much from the past, then the forecasting models become inaccurate and must be recomputed.

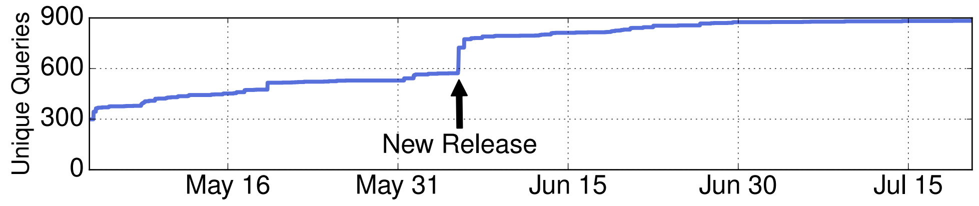
We now investigate the common characteristics and patterns in today's database applications in more detail. We begin with an introduction of the three real-world database application traces used in our discussions and experiments, followed by an overview of the common patterns that these traces exhibit. We then discuss the challenges that effective forecasting models must overcome.



(a) Cycles (BusTracker)



(b) Growth and Spikes (Admissions)



(c) Workload Evolution (MOOC)

Figure 3.1: Workload Patterns – Examples of three common database workload patterns.

3.1.1 Sample Workloads

We now give a high-level description of the three sample workload traces collected from real-world database applications. Table 3.1 provides a more detailed summary of their properties.

Admissions: An admissions website for a university's graduate program. Students submit their application materials to programs in different departments. Faculties review the applications after the deadline and give their decisions.

BusTracker: A mobile phone application for live-tracking of the public transit bus system. It ingests bus location information at regular intervals from the transit system, and then helps users find nearby bus stops and get route information.

MOOC: A web application that offers on-line courses to people who want to learn or teach [4]. Instructors can upload their course materials, and students can check out the course content and submit their course assignments.

3.1.2 Workload Patterns

We now describe the three common workload patterns prevalent in today's database applications. The first two patterns are examples of different arrival rates that the queries in database applications can have. The third pattern exhibits how the composition of the queries in the workload can change over time.

Cycles: Many applications are designed to interact with humans, and as such, their workloads follow cyclic patterns. For example, some applications execute more queries at specific ranges of a day than at others because this is when people are awake and using the service. Figure 3.1a shows the number of queries executed per minute by the BusTracker application over a 72-hour period. The DBMS executes more queries in the daytime, especially during the morning and afternoon rush hours since this is when people are taking buses to and from work. This cycle repeats every 24 hours. Not all applications have such peaks at the same time each day, and the cycles may be shorter or longer than this example.

Growth and Spikes: Another common workload pattern is when the query volume increases over time. This pattern is typical in start-ups with applications that become more popular and in applications with events that have specific due dates. The Admissions application has this pattern. Figure 3.1b shows the number of queries per minute executed over a week-long period leading up to the application deadline. The arrival rate of queries increases as the date gets closer: It grows slowly at the start of the week but then increases rapidly for the final two days before the deadline.

Workload Evolution: Database workloads evolve over time. Sometimes this is a result of changes in the arrival rate patterns of existing queries (e.g., new users located in different time zones start using an application). The other reason this happens is that the queries can also change (e.g., new application features). Of our three applications, MOOC incurs the most changes in its workload mixture. Figure 3.1c shows the accumulated number of distinct queries that the MOOC application executes over time. The graph shows that there is a large shift in the workload after the organization released a new feature in their application in early May.

3.1.3 Discussion

There are three challenges that a forecasting framework must solve to work in real-world DBMS deployments. Foremost is that in order to exploit the various arrival rate patterns in the workloads for optimization planning, there is a need for good arrival rate forecasting models. Not only do different workloads have different patterns, but a single workload can also have separate patterns for sub-groups of queries. Thus, an effective forecasting model must be able to identify and characterize patterns that are occurring simultaneously within the same workload.

The second challenge is that since applications execute millions of queries per day, it is not feasible to build forecasting models for each query in the workload. This means that the framework must reduce the complexity of the workload that it analyzes without severely reducing the accuracy of its predictions.

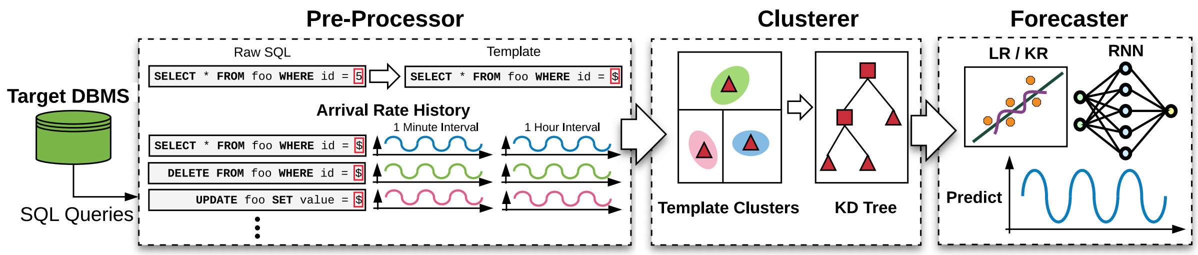


Figure 3.2: QB5000 Workflow – The framework receives SQL queries from the DBMS. This data is first passed into the Pre-Processor that identifies distinct templates in the workload and records their arrival rate history. Next, the Clusterer combines the templates with similar arrival rate patterns together. This information is then fed into the Forecaster where it builds models that predict the arrival rate of templates in each cluster.

Lastly, the framework must handle the changes in the workload's patterns as well as the query mixtures. All of this must be done without any human intervention. That is, the framework cannot require for a DBA to tune its internal parameters or provide hints about what the application's workload is and when it changes.

3.2 Overview

The QueryBot 5000 (**QB5000**) is a workload forecasting framework that runs as either an external controller or as an embedded module. The target DBMS connects to the framework and forwards all the queries that applications execute on it. Decoupling the framework from the DBMS allows the DBA to deploy it on separate hardware resources. Forwarding the queries to QB5000 is a lightweight operation and is not on the query executor's critical path. As QB5000 receives these queries, it stores them in its internal database. It then trains models that predict which types of queries and how many of them the DBMS is expected to execute in the future. A self-driving DBMS can then use this information to deploy optimizations that will improve its target objective (e.g., latency, throughput) [160].

As shown in Figure 3.2, QB5000's workflow is comprised of two phases. When the DBMS sends a query to QB5000, it first enters the **Pre-Processor** and the **Clusterer** components. This is the part of the system that maps the unique query invocation to previously seen queries. This enables QB5000 to reduce both the computational and storage overhead of tracking SQL queries without sacrificing accuracy. The Pre-Processor converts raw queries into generic *templates* by extracting constant parameters out of the SQL string. It then records the arrival rate history for each template.

It is still, however, not computationally feasible to build models to capture and predict the arrival patterns for each template. To further reduce the computational resource pressure, QB5000 then maps the template to the most similar group of previous queries based on its semantics (e.g., the tables that it accesses). The Clusterer then performs further compression of the workload using an on-line clustering technique to group templates with similar arrival rate patterns together. It is able to handle evolving workloads where new queries appear and older ones disappear.

In the final phase, the **Forecaster** selects the largest template clusters (i.e., clusters with the highest query volumes) and then trains forecasting models based on the average arrival rate of the templates within each cluster. These models predict how many queries in each template cluster that the application will execute in the future (e.g., one hour from now, one day from now). This is important because the DBMS will decide how to optimize itself based on what it expects the application to do in the future, rather than what happened in the past. QB5000 also automatically adjust these clusters as the workload changes over time. Every time the cluster assignment changes for templates, QB5000 re-trains its models.

The Pre-Processor always ingests new queries and updates the arrival rate history for each template in the background in real time when the DBMS is running. The Clusterer and Forecaster periodically update the cluster assignments and the forecasting models. When QB5000 predicts the expected workload in the future, it uses the most recent data as the input to the models.

3.3 Pre-Processor

Most applications interact with a DBMS in a programmatic way. That is, the queries are constructed by software in response to some external mechanism rather than a human writing the query by hand. For OLTP workloads, the application invokes the same queries with different input parameters (e.g., prepared statements). For OLAP workloads, a user is often interacting with a dashboard or reporting tool that provides an interface to construct the query with different predicates and input parameters. Such similar queries execute with the same frequency and often have the same resource utilization in the system. Thus, QB5000 can aggregate the volume of queries with identical templates together to approximate the characteristics of the workload. This reduces the number of queries that QB5000 tracks since it only needs to maintain arrival rate information for each template rather than each individual query. Given this, we now describe how QB5000's Pre-Processor collects and combines the queries that it receives from the DBMS.

The Pre-Processor processes each query in two steps. It first extracts all of the constants from the query's SQL string and replaces them with value placeholders. This converts all of the queries into prepared statements. These constants include:

- The values in `WHERE` clause predicates.
- The `SET` fields in `UPDATE` statements.
- The `VALUES` fields in `INSERT` statements. For batched `INSERT`s, QB5000 also tracks the number of tuples.

The Pre-Processor then performs additional formatting to normalize spacing, case, and bracket/parenthesis placement. We use the abstract syntax tree from the DBMS's SQL parser to identify the tokens. The outcome of this step is a generic query *template*.

QB5000 tracks the number of queries that arrive per templates over a given time interval and then stores the final count into an internal catalog table at the end of each interval. The system

	Admissions	**BusTracker**	**MOOC**
Total Number of Queries	2546M	1223M	95M
Total Num of Templates	4060	334	885
Num of Clusters	1950	107	391
Reduction Ratio	**1.3M**	**10.5M**	**0.24M**

Table 3.2: Workload Reduction – Breakdown of the total number of queries that QB5000 must monitor after applying the reduction techniques in the Pre-Processor and Clusterer.

aggregates stale arrival rate records into larger intervals to save storage space. We explain how to choose the time interval in Section 3.5.

QB5000 also maintains a sample set of the queries' original parameters for each template in its internal database. We use reservoir sampling to select a fixed amount of items with low variance from a list containing a large or unknown number of items [219]. An autonomous DBMS's planning module uses these parameter samples when estimating the cost/benefit of optimizations [160].

The Pre-Processor then performs a final step to aggregate templates with equivalent semantic features to further reduce the number of unique templates that QB5000 tracks. Evaluating semantic equivalence is non-trivial, and there has been extensive research on this topic [107, 180, 212]. QB5000 uses heuristics to approximate the equivalence of templates. It considers two templates as equivalent if they access the same tables, use the same predicates, and return the same projections. One could use more formal methods to fully exploit semantic equivalence [50]. We found, however, that heuristics provide reasonable performance without reducing accuracy. We defer investigating more sophisticated methods as future work.

Table 3.2 shows that QB5000's Pre-Processor is able to reduce the number of queries from millions to at most thousands of templates for our sample workloads.

3.4 Clusterer

Even though the Pre-Processor reduces the number of queries that QB5000 tracks, it is still not feasible to build models for the arrival patterns of each template. Our results in Section 3.6.5 show that it can take over three minutes to train a single model. Thus, we need to further reduce the total number of templates that QB5000 forecasts.

The Clusterer component combines the arrival rate histories of templates with similar patterns into groups. It takes templates in a high-dimensional feature space and identifies groups of comparable templates using a similarity metric function. To support modeling an application in a dynamic environment (i.e., one where the workload, database physical design, and the DBMS's configuration can change), the clustering algorithm must generate stable mappings using features that are not dependent on the current state of the database. This is because if the mapping of templates to clusters changes, then QB5000 has to retrain all of its models.

We now examine the three design decisions in our implementation of QB5000's clustering phase. We begin with a discussion of the features that it extracts from each template. We then describe how it determines whether templates belong to the same cluster. Lastly, we present

QB5000's clustering algorithm that supports incremental updates as the application's workload evolves, and how the framework quickly determines whether to rebuild its clusters.

3.4.1 Clustering Features

There are three types of features that the framework can derive from templates: (1) *physical*, (2) *logical*, and (3) *arrival rate history*. Physical features are the amount of resources and other runtime metrics that the DBMS used when it executed the query, such as the number of tuples read/written or query latency. Previous clustering algorithms for database applications have used physical features for query plan selection [84], performance modeling [148], and workload compression [48]. The advantage is that they provide fine-grained and accurate information about an individual query. But they are dependent on the DBMS's configuration and hardware, the database's contents, and what other queries were running at the same time. If any of these change, then the previously collected features are useless and the framework has to rebuild its models. Such instability makes it difficult for the DBMS's planning module to learn whether its decisions are helping or hurting the performance.

Another approach is to use the template's logical features, such as the tables/columns it accesses and the properties of the query's syntax tree. Unlike physical features, these logical features do not depend on the DBMS's configuration nor the characteristics of the workload (e.g., which queries execute more often than others). The disadvantage, however, is that they may generate clusters without a discernible workload pattern because there is limited information from the logical feature and thus the forecasting models make poor predictions. The inefficiency of logical features has also been identified in previous work on predicting query runtime metrics [83].

QB5000 uses a better approach to cluster queries based on their arrival rate history (i.e., the sequence of their past arrival rates). For example, consider the cluster shown in Figure 3.3 from the BusTracker application that is derived from four templates with similar arrival rate patterns. The cluster *center* represents the average arrival rate of the templates within the cluster. Although the total volume per template varies at any given time, they all follow the same cyclic patterns. This is because these queries are invoked together as part of performing a higher level functionality in the application (e.g., a transaction). Since templates within the same cluster exhibit similar arrival rate patterns, the system can build a single forecasting model for each cluster that captures the behavior of their queries.

Calculating the similarity between a pair of arrival rate history features is straightforward. QB5000 first randomly samples timestamps before the current time point. Then for each series of arrival rate history, QB5000 takes the subset of values at those timestamps to form a vector. The similarity between the two features is defined as the cosine similarity of the two vectors. If the template is new, we compare its available timestamps with the corresponding subset in the vectors of other templates. Our current implementation uses 10k time points in the last month of a template's arrival rate history as its feature vector. We found that this is enough to capture the pattern of every arrival rate history in our experiments.

Logical features and arrival rate history features express different characteristics of the queries. But as we show in Section 3.6.7, clustering on the arrival rate features produce better models for real-world applications because they capture how queries impact the system's per-

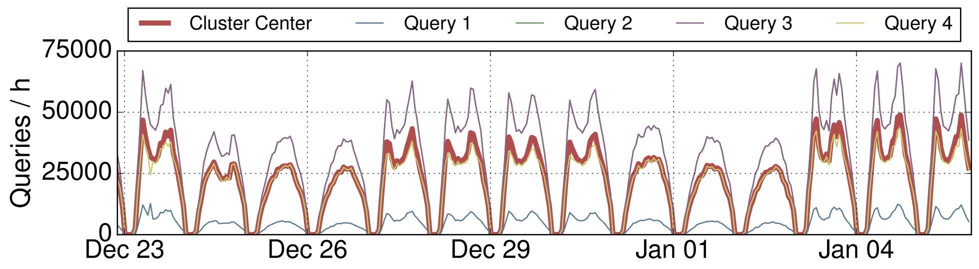


Figure 3.3: Arrival Rate History – The past arrival rates for the largest cluster and the top four queries within that cluster from BusTracker.

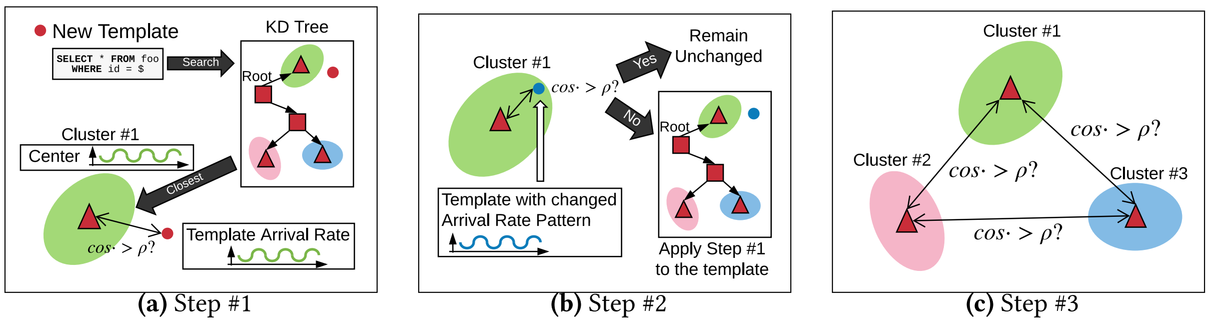


Figure 3.4: On-line Clustering – Clusterer periodically performs three steps to change the templates' cluster assignments incrementally. It first checks the similarities between the arrival rate history features of new templates with the centers of existing clusters. Then it removes the templates from their clusters if the arrival rate patterns of a template and its cluster center have deviated. Finally, it merges the clusters with high similarities between their centers together.

formance. Though using the template's arrival rates avoids rebuilding clusters whenever the DBMS changes, it is still susceptible to workload variations, such as when the system identifies a new template or the arrival rates of existing ones change.

3.4.2 On-line Clustering

QB5000 uses a modified version of **DBSCAN** [78] algorithm. It is a density-based clustering scheme: given a set of points in some space, it groups together points with many nearby neighbors (called *core objects*), and marks points that lie alone in low-density regions as outliers (i.e., points whose nearest neighbors are too far away). Unlike K-means, this algorithm is not affected by the number of small clusters or the cluster densities[1].

The original DBSCAN algorithm evaluates whether an object belongs to a cluster by checking the minimum distance between the object and any core object of the cluster. But we want to assign templates to clusters based on how close they are to a cluster's center and not just any random core object. This is because QB5000 uses the center of a cluster to represent the

[1]We also evaluated K-means clustering, but it has a known problem when the workload has a large number of small clusters, or the clusters have different sizes or densities. These issues have also been observed for previous database workload modeling techniques [48].

templates that are members of that cluster, and builds forecasting models with the center. An on-line extension of the canonical DBSCAN algorithm also has high overhead when updating clusters [79].

Our on-line variant of DBSCAN uses a threshold, ρ ($0 \leq \rho \leq 1$), to decide how similar the arrival rates of the templates must be for them to belong to the same cluster. The higher ρ is, the more similar the arrival rates of the templates within a cluster are, so the modeling result will be more accurate. But the computational overhead will also be higher given the larger number of generated clusters. We conduct a sensitivity analysis on setting this value in Section 3.6.8. As shown in Figure 3.4, QB5000's incremental clustering algorithm periodically performs the following three steps together:

Step #1: For each new template, QB5000 first checks whether the similarity score between its arrival rate history and the center of any cluster is greater than ρ. The template is assigned to the cluster with the highest similarity score that is greater than ρ. We use a *kd-tree* to allow QB5000 to quickly find the closest center of existing clusters to the template in a high-dimensional space [30]. Then QB5000 will update the center of that cluster, which is the arithmetic average of the arrival rate history of all templates in that cluster. If there is no existing cluster (this is the first query) or none of the clusters' centers are close enough to the template, QB5000 will create a new cluster with that template as its only member.

Step #2: QB5000 checks the similarity of previous templates with the centers of the clusters they belong to. If a template's similarity is no longer greater than ρ, QB5000 removes it from its current cluster and then repeat step (1) to find a new cluster placement. Sometimes moving a template from one cluster to another causes the centers of the two clusters to change, and recursively forces other templates from the two clusters to move. QB5000 defers modifying the clusters until the next update period. QB5000 removes a template if it has not received one of its queries for an extended period.

Step #3: QB5000 computes the similarity between the clusters' centers and merges two clusters with a score greater than ρ.

In addition to periodically executing these three steps, QB5000 monitors the new templates in the workload. If the percentage of previously unseen templates is above a threshold, it then triggers these steps to adapt to the workload change. Setting this threshold properly is dependent on the performance attributes of the target DBMS. We defer investigating this problem as future work.

QB5000's incremental algorithm adaptively adjusts the clusters for a dynamic workload without requiring a warm-up period or having prior knowledge of the workload. More importantly, it guarantees that the similarity between a template's arrival rate history and the center of its cluster is smaller than ρ, which improves the accuracy of QB5000's forecasting models. The complexity of these steps is bounded by $O(n \log n)$, where n is the number of templates in the workload. Since Step #2 does not apply cluster changes recursively, this approach does not guarantee the convergence of the clusters at any specific time. Thus, QB5000 might not achieve

	LR	ARMA	KR	RNN	FNN	PSRNN
Linear	✓	✓	✗	✗	✗	✗
Memory	✗	✓	✗	✓	✗	✓
Kernel	✗	✗	✓	✗	✗	✓

Table 3.3: Forecasting Models – The properties of the forecasting models that we investigated for QB5000.

the optimal clustering given the similarity metric. We found, however, this does not affect the efficacy of its forecasting models in practice.

Table 3.2 shows the number of clusters determined by Clusterer and the reduction ratio from the total number of queries in the three workloads. Since QB5000 periodically updates its clustering results, we show the average number of clusters per day.

3.4.3 Cluster Pruning

Even after using clustering techniques to reduce the total number of queries that QB5000 needs to model, real-world applications still tend to have lots of clusters because of the long-tailed distribution of the arrival rate patterns. There are only a few large clusters that exhibit the major workload patterns, but several small clusters with noisy patterns. Those small clusters usually contain queries that only appear a few times and increase the noise in the models with little to no benefit since they are not representative of the application's main workload. QB5000 does not build models for them since their impact on the DBMS's performance is limited. Our experiments in Section 3.6.1 show that the five largest clusters cover up to 95% of the query volume for our three sample workloads.

3.5 Forecaster

At this point in QB5000's pipeline, the framework has converted raw SQL queries into templates and grouped them into clusters. QB5000 is also recording the number of queries executed per minute for each cluster. The final phase is to build *forecasting models* to predict the arrival rate patterns of the clusters' queries. These models allow the DBMS's planning module to estimate the number of queries that the application will execute in the future and select the proper optimizations to meet SLAs [6, 160]. In this section, we describe how QB5000 constructs and uses its forecasting models. We begin with an explanation of its underlying data structures and training methods. We then discuss how QB5000 supports different prediction horizons and intervals for the same cluster over multiple models.

3.5.1 Forecasting Models

There are many choices for forecasting models of the query arrival rates with different prediction properties. Table 3.3 shows a summary of the six models that we considered in the development of QB5000. The first property is whether the model is *linear*, which means that it assumes that there is a linear relationship between the input and output data. Next, a model can retain *memory* that allows it to use both the input data and the information "remembered" from the past observations to predict the future. Lastly, a model can support *kernel* methods to provide another way to model non-linear relationships. In contrast to models that employ non-linear functions, kernel methods achieve non-linearity by using linear functions on the feature maps in the kernel space of the input.

Linear models are good at avoiding overfitting when the intrinsic relationship in the data is simple. They take less computation to build and require a smaller amount of training data. On the other hand, more powerful non-linear models are better at learning complex data patterns. They do, however, take longer to train and are prone to overfitting. Thus, they require more training data. As we will show in Section 3.6.2, linear models often perform better at making predictions in the near future (e.g., one hour) whereas the non-linear models are better at making predictions further out in time (e.g., over a day). But it is non-trivial to determine which type of model to use for different time horizons on different workloads. Likewise, there are also trade-offs between memory-based models. Retaining memory allows the model to exploit the dynamic temporal behavior in an arbitrary sequence of inputs, but this adds training complexity and makes the model less data-efficient.

A well-known solution that works well in other application domains is to use an *ensemble* method (**ENSEMBLE**) that combines multiple models together to make an average prediction. Ensemble methods are used in prediction tasks to combine several machine learning techniques into one predictive model in order to decrease variance or bias (e.g., *boosting*) [154]. Previous work has shown that ensemble methods work well on challenging datasets and they are often among the top winners of data science competitions [165, 250].

We now discuss the two types of forecasting models that QB5000 combines to make its ENSEMBLE model.

Linear Regression (LR): LR models are also known in the statistics and time series prediction literature as linear auto-regressive models. They are simple linear models that have closed-form solutions, which means that they do not require an additional optimization step to find a global optima. In QB5000, the framework regresses the future arrival rate of queries in a cluster based on the arrival rate of the query over a specified period of time in the past. LR has been used for DBMS operator modeling in prior work [23].

Recurrent Neural Network (RNN): Previous work has shown RNNs to be effective at predicting patterns for non-linear systems [225]. It is a class of network where its neurons have a cyclic connection. QB5000 uses a variant of the RNN called *long short-term memory* (LSTM) [98]. LSTMs contain special blocks that determine whether to retain older information and when to output it into the network. This allows the networks to automatically learn the periodicity and repeating trends of data points in a time-series beyond what is possible with regular RNNs [81]. This approach has been used to predict the host-load in data centers [195].

We apply an ensemble method by equally averaging the prediction results of the LR and RNN models. We also tried averaging the models with weights derived from the training history, but that led to overfitting and generated worse results.

Although the ensemble method achieves good average prediction accuracy, we found that it fails to predict the periodic spikes in the workload that are far apart from each occurrence. For example, the December 15th deadline shown in Figure 3.1b occurs each year on the same date, but both the LR and RNN models are unable to predict this annual pattern. This is a common workload pattern, and the ability to predict such spikes ahead of time is necessary for many database optimizations, such as resource provisioning. Thus, we now describe the third forecasting model that we employ in QB5000 that is able to correctly handle this scenario.

Kernel Regression (KR): This is a non-linear variant of LR models that uses the Nadaraya-Watson estimator to achieve its non-linearity without iterative training [33]. The prediction for a given input is a weighted average of training outputs where the weights decrease with distance between the given input and corresponding training inputs.

KR is a non-parametric method, which means that it assumes no particular functional form. It instead only assumes that the function is smooth. This provides it with the necessary flexibility to model different non-linear functions between the inputs and the outputs. Thus, it is able to predict when a spike will repeat in the future even if the spike has only occurred a few times in the past. KR, however, does not extrapolate well with data it has not seen before. As we show in Section 3.6.2, it performs worse than ENSEMBLE in terms of the average prediction accuracy. But it is the only investigated model that is able to handle the yearly spike in Admissions.

Given this, QB5000 uses a hybrid forecast model (**HYBRID**) that we developed to automatically determine when to use predictions from ENSEMBLE versus ones generated from KR. Since KR is good at predicting spikes with a small number observations, if its predicted workload volume is above that of ENSEMBLE by more than a specified threshold, γ ($\gamma \geq 0$), then QB5000 uses the result from KR as its prediction. Otherwise, it uses the result generated from the ENSEMBLE model. In QB5000, we set γ to 150% as this provided the most accurate forecasts for all of the application workloads that we tested. We provide a sensitivity analysis of γ in Section 3.6.10.

3.5.2 Prediction Horizons & Intervals

The scope of a forecasting model is defined in terms of its horizon and interval. How far into the future a model can predict is known as its *prediction horizon*. In general, the longer the horizon is, the less accurate its predictions are expected to be. This is important for self-driving DBMSs since it improves their ability to prepare for immediate workload and resource demand changes. The time granularity at which the model can predict is called its *prediction interval*. For example, a model can predict the number of queries that will execute in one-minute or one-hour intervals. Like the horizon, using a shorter interval often improves the accuracy of its models, but increases the storage and computational overhead.

QB5000 sets the interval at which it records the query arrival rates to be one minute, which is the finest level of prediction that QB5000 is able to provide to the DBMS. When a self-driving DBMS' planning module evaluates potential optimizations, it can decide how to aggregate the per-minute history into longer intervals to train the forecasting models. We evaluate QB5000's performance when aggregating over different time intervals in Section 3.6.4. To predict the spikes, QB5000 trains the KR model used by HYBRID using the entire history of an application

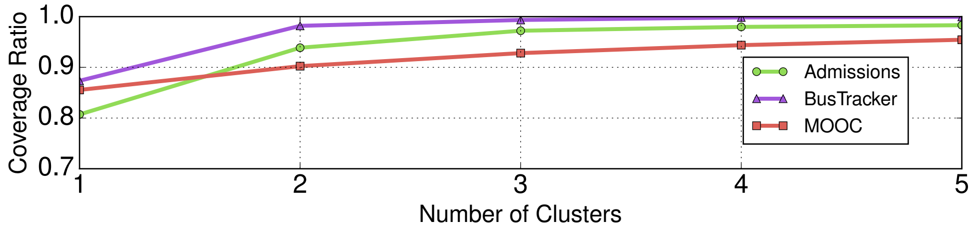


Figure 3.5: Cluster Coverage – The average ratio between the volume of the largest clusters and the total workload volume.

aggregated into one-hour intervals to reduce the computational and storage overhead.

A self-driving DBMS' planning module also decides how far ahead of time its models need to make predictions. QB5000 builds a forecasting model for each required prediction horizon.

3.6 Experimental Analysis

We now present an evaluation of QB5000's ability to model a database application's workload and predict what it will look like in the future. We implemented QB5000's algorithms using `scikit-learn`, `Tensorflow`, and `PyTorch`. We use the three real-world workload traces described in Section 3.1.1. We performed our experiments on a single server with an Intel Xeon E5-2420v2 CPU and 32 GB RAM. Unless otherwise noted, we use a GeForce GTX 1080 GPU with 8 GB RAM for training QB5000's forecasting models.

We first analyze the effectiveness of the Clusterer's compression techniques. We then evaluate the accuracy of the Forecaster's models. We also examine the computation time and storage footprint for QB5000's components. Next, we demonstrate the benefits of QB5000 for self-driving DBMSs for an example application. Lastly, we provide the results of a few sensitivity analysis of QB5000's abilities.

3.6.1 Number of Clusters

The goal of this first experiment is to show that QB5000 can model the majority of a database workload using a small number of the highest-volume clusters. Although modeling more clusters may allow QB5000 to capture more information about the workload, it also increases the computational overhead, data requirements, and memory footprint. We use the method described in Section 3.4 to perform on-line query clustering for all the three workloads. We set QB5000's threshold as ρ=0.8 and the frequency at which it performs incremental clustering algorithm to be once per day. We provide a sensitivity analysis for selecting ρ in Section 3.6.8.

We first calculate the average ratio between the volume of the largest clusters and the total workload volume for each day throughout the entire workload execution. It is calculated by dividing the volume of a given cluster by the total volume of all the clusters for that day. The results in Figure 3.5 show that the highest-volume clusters cover the majority of the queries in the workload. For example, in BusTracker most of the queries come from people checking bus schedules during the morning and evening rush hours. In particular, the five largest clusters

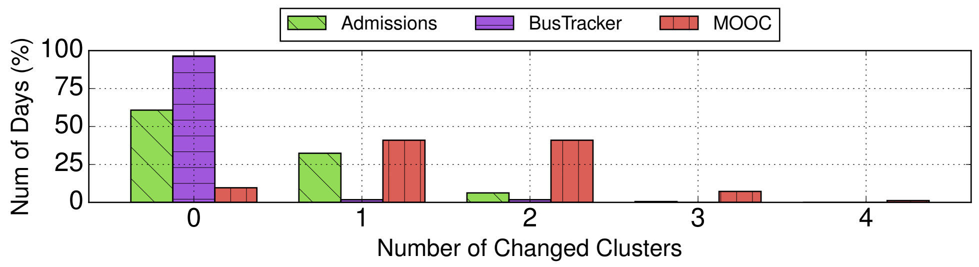


Figure 3.6: Cluster Change – The number of clusters that changed among the five largest clusters between two consecutive days.

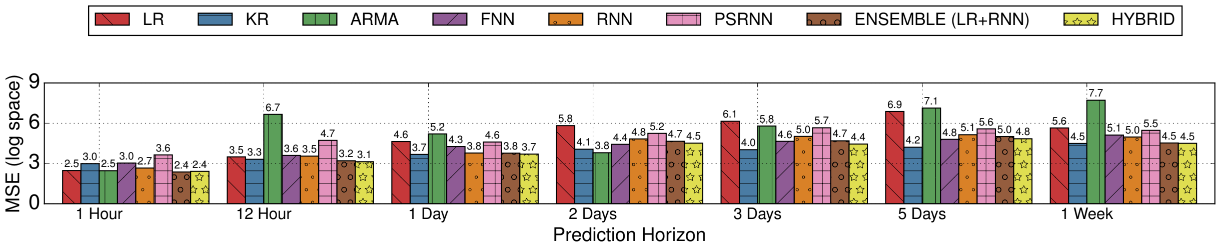
comprise over 95% of the queries for all three workloads. This shows that even though a real-world application may consist of several arrival rate patterns among sub-groups of queries, we can still obtain a good estimation of the workload by modeling only a few of its major patterns. Recall that QB5000 tracks the ratio between the volume of the templates within a cluster.

We then calculate how frequently the five highest-volume clusters change from one day to the next. Figure 3.6 shows the number of days where zero or more changes occurred in the five largest clusters of the three workloads. For Admissions and BusTracker, there is at most one change in the five largest clusters for over 90% of the days. This shows that QB5000's on-line clustering method is not only efficient regarding the characterization of the workload, but is also stable under the usual fluctuations in the workload. The MOOC workload has more cluster changes than the other two because new queries appear as instructors create and launch new classes. This shows that QB5000's incremental clustering algorithm can capture shifts in the application's workload as it changes over time.

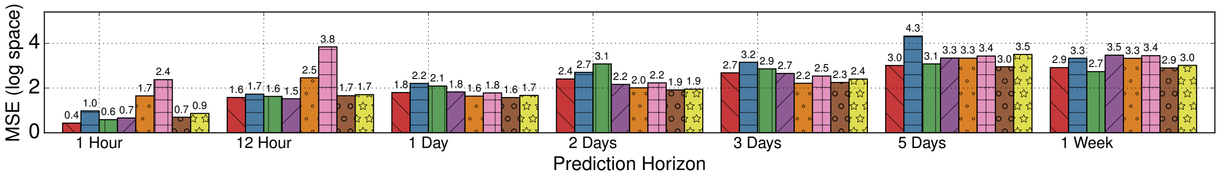
3.6.2 Prediction Accuracy Evaluation

We now evaluate the prediction accuracy of QB5000's forecasting models for different prediction horizons. Recall from Section 3.5.2 that the prediction horizon defines how far into the future a model predicts. In this experiment, QB5000 models the highest-volume clusters that cover more than 95% of the total queries in the workload. Our previous results in Figure 3.5 show that we can achieve this by modeling the three largest clusters for Admissions and BusTracker, and the five largest clusters for MOOC. QB5000 trains a single forecasting model that jointly predicts the query arrival rates for all of the clusters on each prediction horizon. This allows for sharing information across clusters, which improves the prediction accuracy. It uses up to three weeks of the latest query arrival rate data for the training. We use the log of the mean squared error (MSE) as the metric for measuring the accuracy of QB5000's forecasting models. The smaller the MSE, the better the prediction accuracy. We use one-hour prediction horizon in this experiment.

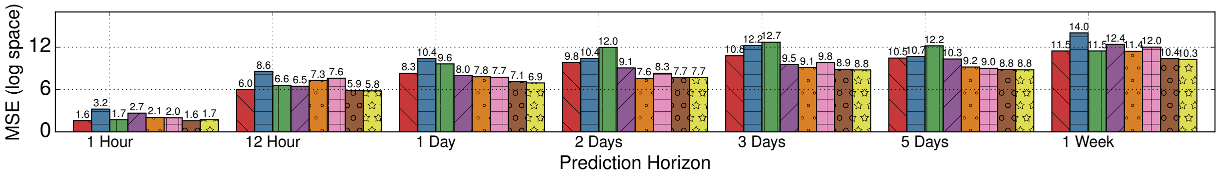
For a self-driving DBMS, a desirable property of its forecasting models is that they are not overly sensitive to their hyperparameters. This is because fine-tuning a model's hyperparameters is by itself a hard optimization task. To evaluate this, we fix the hyperparameters for all models to be the same across the different prediction horizons and workloads. We obtained these settings using cross-validation. Except for the KR used by HYBRID, we use the last day's



(a) Admissions



(b) BusTracker



(c) MOOC

Figure 3.7: Forecasting Model Evaluation – The average prediction accuracy of the different forecasting models over prediction horizons ranging from one hour to one week for the Admissions, BusTracker, and MOOC workloads.

arrival rate as the input for the LR and KR models. For RNN, we use a linear embedding layer of size 25 followed by two LSTM layers each with 20 cells. We take the log of the input before training the models, and convert them back by taking the exponentials of the output.

We compare the accuracy of QB5000's forecasting models discussed in Section 3.5.1 against other models used for forecasting arrival rate patterns. They are used in existing approaches for modeling system workloads and resource usages, as introduced below.

Autoregressive Moving Average (ARMA): ARMA is a generalization of LR models that consists of an autoregressive part and a moving average part acting on residuals. When making predictions, ARMA makes use of all previous observations, either directly or through its residuals. This approach was used for predicting workloads in cloud service environments [27, 176].

Feed-forward Neural Network (FNN): This is a non-linear version of the LR models in which the linear function that approximates the output is replaced by a feed-forward neural network that separates a sequence of linear transformations to the input vector by non-linear activations [207]. FNNs differ from RNNs in that their neurons do not form a cycle that feeds information from all previous observations back into the model. This approach was used to predict the resource usage for transactions in an OLTP DBMS [105].

Predictive State Recurrent Neural Network (PSRNN): This a newer RNN variant that outperforms LSTMs in a variety of prediction tasks [70]. The key advantage of RNNs is that they have an initialization algorithm based on a method of moments that aims to start the optimization process in a better position towards the global optima, as opposed to typical RNN/LSTM initializations.

Figure 3.7 shows the average prediction accuracy of the forecasting models over horizons ranging from one hour to one week for the three workloads. These results exhibit similar trends for how the horizon impacts the prediction accuracy of the models. First, we observe that for shorter horizons, the LR models perform as well as or better than the more complex RNN models. This is because when the horizon is short, the relationship between the arrival rate observed in the recent past and the arrival rate in the near future is more linear than for longer horizons. Thus, a simple model like LR is sufficient for making predictions. In contrast, complex models often overfit the noise in the training data and generate less accurate results when the horizon is short. But as the horizon increases, the relationship between the past and the future also grows in complexity. In this case, more powerful models like RNNs that are adept at learning complex relationships achieve better accuracy. This is consistent with our results; RNN outperforms LR when the horizon is greater than or equal to one day. This effect is also observed in sequence prediction in other domains [34].

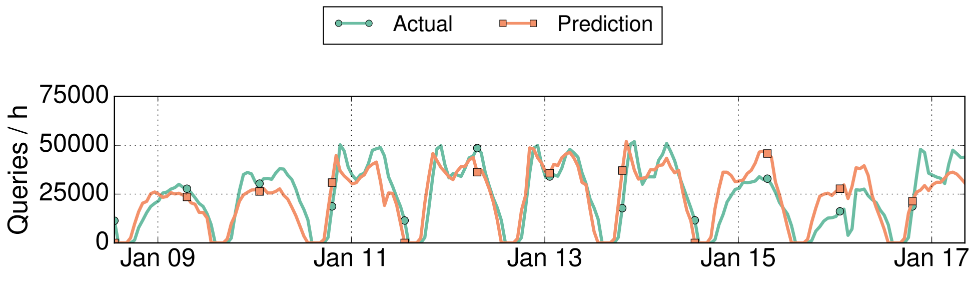
These results also show that the accuracy of ARMA is not stable across the different horizons. For all of the trials in Figure 3.7, it achieves the best performance for only 10% of them, but it has the worst performance 38% of the time. This is because the model is sensitive to its hyperparameters. The optimal hyperparameter settings for ARMA are highly dependent on the statistical properties of the data, such as stationarity and the autocorrelation structure.

The FNN models generally have worse prediction accuracy compared to the RNN models. FNN achieves the best and the worst accuracy in both 5% of the trials. The FNN models cannot remember the state of the workload like RNNs. They also lack the simplicity of LR that protects against overfitting.

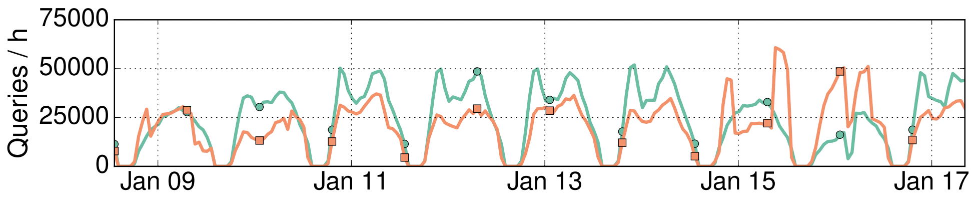
KR has the best performance in 19% of the experiments, but the worst in 24% of the experiments. This model is able to model non-linear functions, but it is prone to error when it has not seen inputs in training that are close to the input to make the prediction with.

PSRNN also performs worse than RNN. It is supposed to have a better initialization than RNN, but this does not always guarantee a better performance because (1) it uses approximation algorithms to find the initialization and (2) its benefit from smarter initialization is restricted when the size of the training data is limited [70]. Since RNN takes less training time than PSRNN and is more available in ML frameworks, we did not use PSRNN in ENSEMBLE.

As shown in Figure 3.7, ENSEMBLE provides the best overall prediction accuracy. It performs better than all the stand-alone models in 61% of the experiments and never has the worst performance. Ensemble methods often have lower variance than their underlying models and produce better results when their models have complementing characteristics [124]. LR and RNN have distinct properties: LR only uses a limited number of observations from the past when making linear predictions, whereas RNN is non-linear and maintains state to memorize



(a) 1-Hour Horizon



(b) 1-Week Horizon

Figure 3.8: Prediction Results – Actual vs. predicted query arrival rates for the highest-volume cluster in the BusTracker workload with prediction horizons of one hour and one week.

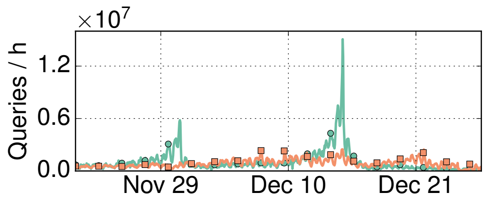
the information from all previous observations. Since LR has comparable performance to ENSEMBLE for horizons shorter than a day, using LR on these short horizons may also be desirable for a DBMS that is short of computational resource.

Even though ENSEMBLE achieves the best accuracy of all the models, recall from Section 3.5.1 that it cannot predict the spikes that repeat infrequently in the workload. HYBRID solves this issue by correcting the result of ENSEMBLE with the help of the prediction from KR. Figure 3.7 shows that HYBRID has little impact on the average prediction accuracy compared to ENSEMBLE.

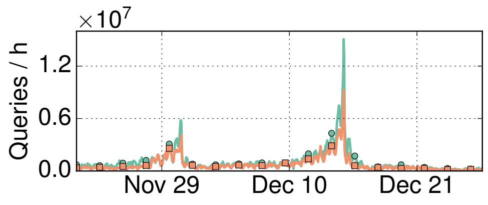
We now demonstrate how QB5000 uses its HYBRID models to predict queries' arrival rates for each cluster. We compare the predicted arrival rates of the queries belonging to the highest-volume cluster in the BusTracker application with their actual arrival rates for one-hour and one-week horizons. Figure 3.8 shows that the one-hour horizon prediction is more accurate than the one-week horizon. This is consistent with Figure 3.7 where the prediction accuracy decreases for longer horizons. The results also show that QB5000's models provide good predictions for both horizons since the predicted patterns of the arrival rates closely mimic the actual patterns.

3.6.3 Spike Prediction Evaluation

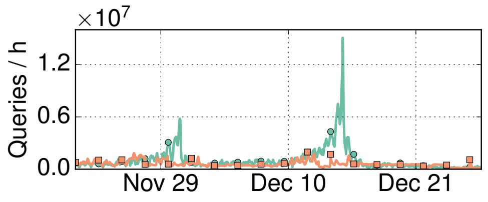
Next, we evaluate QB5000's ability to forecast the growth and spike workload pattern. As mentioned in Section 3.5.2, QB5000 uses the full workload history aggregated into one-hour intervals to predict spikes. It tries to identify workload spikes one week before they will occur. The other settings are the same as in Section 3.6.2. We again evaluated all of the models and present the forecasting results from Nov 15 to Dec 31 (2017), which include two spikes from the admission deadlines on Dec 1 and Dec 15. The Admissions trace contains the similar spikes from the previous year (2016).



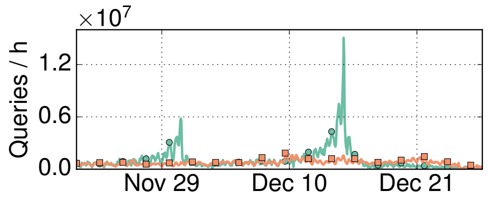
(a) Linear Regression (LR)



(b) Kernel Regression (KR)



(c) Recurrent Neural Network (RNN)



(d) Ensemble (LR + RNN)

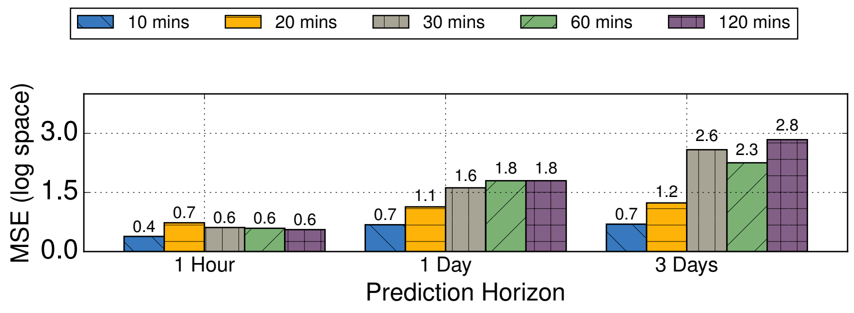
Figure 3.9: Prediction Results – Actual vs. predicted query arrival rates for the combined clusters in the Admissions workload with spike patterns.

The results in Figure 3.9 show that ENSEMBLE and its two base models are unable to predict the spikes in the workload. Although they are not shown here, the other models also perform poorly in this scenario. The linear models' failure is likely due to the scarcity of the spike data and the limited capacity of the models. In contrast, models with higher capacity, like RNNs, may get caught in local optima and thus are also not able to produce good results. KR is the only model that successfully predicts the spikes. This is because its prediction is based on the distance between the test points and training data, where the influence of each training data point decreases exponentially with its distance from the test point. We further demonstrate why KR can separate data points with high query volumes from the other data points in Section 3.6.9.

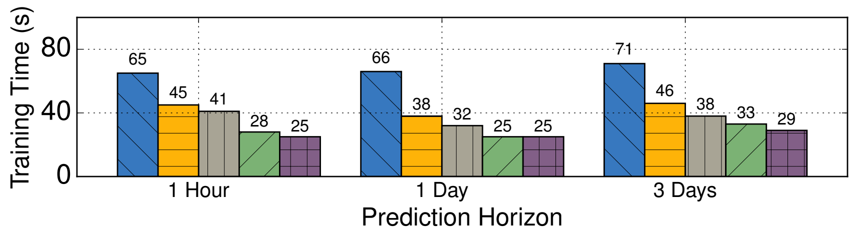
3.6.4 Prediction Interval Evaluation

We next evaluate the prediction accuracy and the training time of our ENSEMBLE forecasting models with varying prediction intervals. Since KR from HYBRID always uses one-hour intervals, it is unnecessary to evaluate it for other intervals. To provide a fair comparison, we compute the total prediction for each hour in the horizon by summing the predictions across the intervals within that hour. We then use these per-hour predictions to compute the MSE. For two-hour intervals, we calculate the prediction for each hour by dividing the interval that contains that hour into two.

The results in Figure 3.10a show that the accuracy of the models increases as the intervals become shorter. This is because shorter intervals provide more training samples and better information for learning the patterns in the data. Shorter intervals are especially beneficial for longer horizons since the relationship between the future and the past arrival rate is more complex. But shorter intervals increase the noise in the data and require more intervals to include the same extent of time. Thus, models trained on shorter intervals are larger (e.g.,



(a) Prediction Accuracy



(b) Training Time (GPU)

Figure 3.10: Prediction Interval Evaluation – The average prediction accuracy and training time with different intervals for BusTracker.

higher input dimension) and more complex.

Figure 3.10b shows the training time for each model at different intervals. It takes less time to train the models on longer intervals, which is expected since these models are smaller and less complex. Increasing the interval from 10 to 120 minutes reduces the training time by roughly 2.5× across all horizons, but decreasing the horizon provides only a minor training time reduction across all intervals.

One must consider these trade-offs when setting the interval, along with the planning capabilities of the target self-driving DBMS. We found that a one-hour interval works well for our operating environment in Section 3.6.6. As such, we use this interval for the remainder of the evaluation. Automatically determining the interval is beyond the scope of this thesis and we leave it as future work.

3.6.5 Computation & Storage Overhead

To better understand the computational and storage overhead of QB5000, we instrumented its code to record the amount of time and space it spends in its four components:

Pre-Processor : The time to templatize a query and update its arrival rate history, and the amount of history data generated daily.

Clusterer : The time to update the clustering results once per day according to the latest history, and the size of the result data.

		Pre-Processor	**Clusterer**	**LR**	**RNN**	**KR**
Computation	Admissions	0.043ms/query	15s/day	GPU:0.3s CPU:0.3s	GPU:9s CPU:58s	GPU:0.16 CPU:0.18s
	BusTracker	0.05ms/query	3s/day	CPU:0.12s GPU:0.13s	GPU:33s CPU:221s	GPU:0.02s CPU:0.02s
	MOOC	0.048ms/query	12s/day	GPU:0.54s CPU:0.51s	GPU:5s CPU:18s	GPU:0.04s CPU:0.04s
Storage	Admissions	1.6MB/day	6.7KB	100B	28KB	11MB
	BusTracker	0.25MB/day	2.2KB	100B	28KB	1.9MB
	MOOC	1.4MB/day	0.8KB	100B	28KB	0.4MB

Table 3.4: Computation & Storage Overhead – The measurements for QB5000's different components.

LR Model : The time to train one LR model, and the size of the learned weights.

KR Model : The time to predict one test point with the KR model, and the size of the historical data maintained for the model.

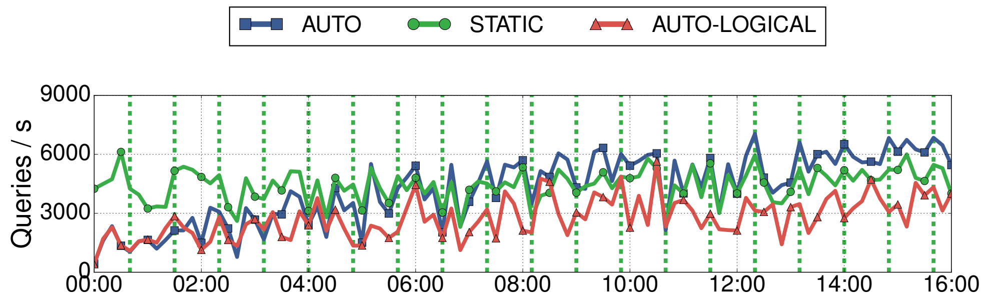
RNN Model : The time to train one RNN model, and the size of the serialized model object from `PyTorch`, which contains both the model parameters and network structure.

For LR, RNN, and KR, we use a one-hour interval and measure the model's time and space requirements over seven horizons. We report the average overhead of these horizons.

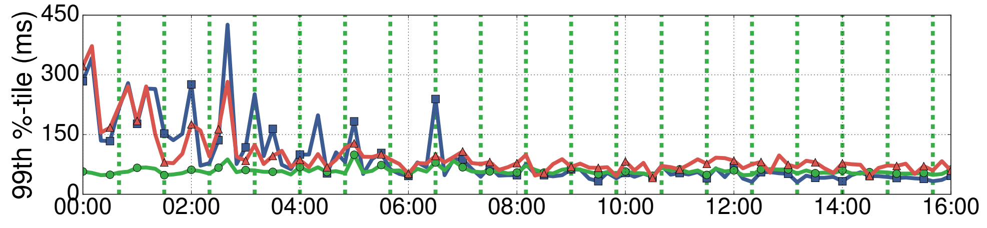
Table 3.4 shows that all of QB5000's components have reasonable storage overhead. The results also show that training the RNN models is the most computationally expensive task. We stop training the RNN models when the validation accuracy stops improving. This means that the number of iterations performed and the training time differ per workload. Using a GPU improves the training time of RNN models by 3.6–7×. The overhead of training LR models is low compared to RNN models, and that the CPU/GPU performances are similar for the LR models since they are so simple. KR requires no training time and little testing time. But its testing time and training-data size increases linearly with the length of the workload history. QB5000 reduces this overhead by always aggregating the training for KR to one-hour intervals (see Section 3.5.2).

3.6.6 Automatic Index Selection

We now demonstrate how self-driving DBMSs can use QB5000's workload forecasts to make proactive optimizations that improve the system's performance. We integrate QB5000 with MySQL [2] and PostgreSQL [8] to process, cluster, and predict SQL workloads to automatically builds indexes for the predicted workload. We select a representative workload for PostgreSQL (BusTracker) and for MySQL (Admissions) for the evaluation. We use a 10 GB database for the



(a) Throughput



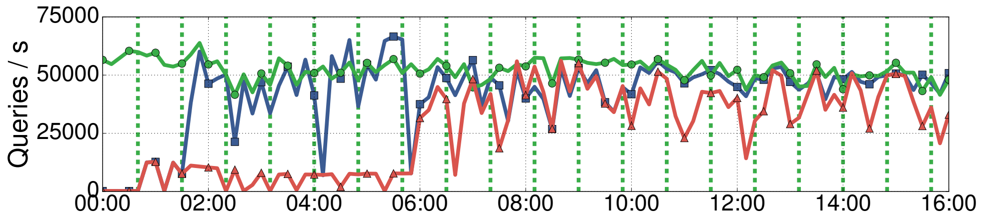
(b) Latency

Figure 3.11: Index Selection (MySQL) – Performance measurements for the Admissions workload using different index selection techniques.

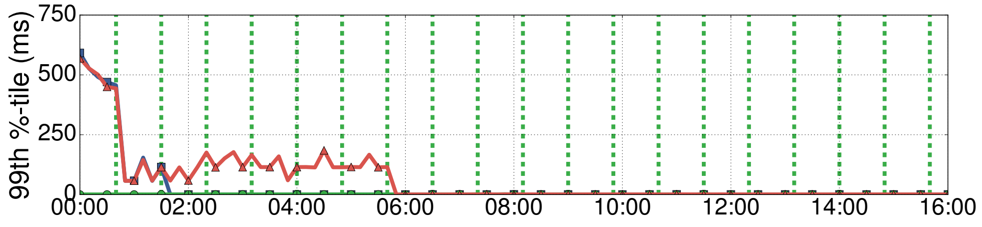
Admissions and a 5 GB database for the BusTracker. We set each DBMS's buffer pool size to be 1/5 of the database size.

We use an index selection technique based on the one introduced by AutoAdmin to generate the set of indexes to build [46]. AutoAdmin first selects the best index for each query in a sample workload to form a candidate set of indexes. It then uses a heuristic search algorithm to find the best-bounded subset of indexes within the candidates. Instead of using a sample workload to generate the candidate indexes, we use the predicted workload of the three largest clusters generated by Clusterer. We note that the purpose of this evaluation is to demonstrate QB5000's ability to dynamically model and predict workloads, and not the efficacy of the index selection algorithm. We compare the performance of the automatic index selection (**AUTO**) against a static index selection method, which uses the same index selection algorithm but applies it to a fixed workload sample over the entire query history that is prepared manually before the start of the experiment (**STATIC**).

In both of the workloads, we initialize the DBMSs with the primary key and foreign key indexes defined in the applications' original schemas but remove all secondary indexes. We choose a random date to start the index selection process and train QB5000's forecasting models from the previous three weeks history. The system builds a new index at hourly intervals based on QB5000's real-time workload predictions for one-hour and twelve-hour prediction horizons. We weight the predictions from the one-hour horizon higher since models for shorter horizons are more accurate. We run the index selection technique for a 16 hour period. To control the total experiment time, we replay the workload 600× faster than the actual workload execution speed. That is, one second in our experiment represents 10 minutes of workload execution in the traces. During the experiment AUTO builds 20 indexes in total, and thus we set STATIC to



(a) Throughput



(b) Latency

Figure 3.12: Index Selection (PostgreSQL) – Performance measurements for the BusTracker workload using different index selection techniques.

also build 20 indexes before the experiment begins.

Figures 3.11 and 3.12 show the performance of MySQL and PostgreSQL using the AUTO and STATIC index selection techniques. The vertical green dotted lines indicate when the DBMS builds a new index. The results in Figure 3.11 show that the throughput and latency of MySQL executing the Admissions workload improve by 5× and 78% over the 16 hour period, respectively. AUTO initially performs worse than STATIC since it has not yet created any secondary indexes, but achieves 28% better throughput and 23% better latency by the end of the experiment. This is because AUTO selects four indexes that were not chosen by STATIC since it is able to leverage QB5000's forecasts. Figure 3.12 shows that PostgreSQL achieves 180× better throughput and 99% better latency for the BusTracker workload over this period. AUTO selects only one different index than STATIC, and thus their final performances are similar.

Automatic index selection is just one example of how QB5000's workload forecasting could be applied in a self-driving DBMS. There are other application scenarios that would serve as more powerful examples, such as resource provisioning. Such scenarios require a planning module for a self-driving DBMS, which requires a forecasting framework like what we present in this chapter.

3.6.7 Logical vs. Arrival Rate History Feature

In this final experiment, we compare the effectiveness of the arrival rate history feature that QB5000's Clusterer uses against the logical feature (**AUTO-LOGICAL**). We repeat the same experiments from Section 3.6.6, except that we group templates based on the similarity of the logical structures of SQL strings. More specifically, the logical feature vector of a templates consist of the query type (e.g., `INSERT`, `SELECT`, `UPDATE`, or `DELETE`), tables that it accesses, the columns that it references, number of clauses (e.g., `JOIN`, `HAVING`, or `GROUP BY`), and number of

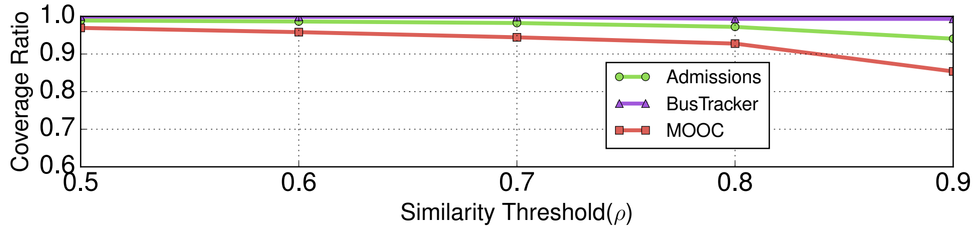


Figure 3.13: Cluster Coverage with ρ – The ratio between the volume of the three largest clusters and the total workload volume with different similarity threshold ρ.

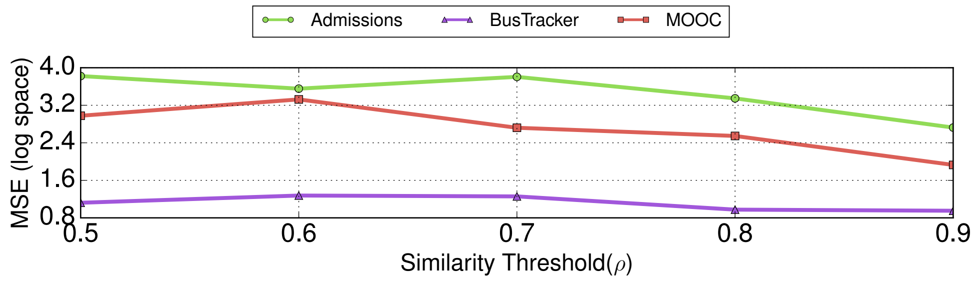


Figure 3.14: Prediction Accuracy with ρ – Normalized prediction accuracy for one-hour horizon with different similarity threshold ρ.

aggregations (e.g., `SUM`, or `AVG`). We use the L2 distance to measure the similarity between two templates. We adjust the threshold ρ so that the volumes of the largest clusters are similar to those generated with the arrival rate history features.

The results in Figures 3.11 and 3.12 show that the DBMSs' throughput is ∼20% slower for AUTO-LOGICAL than for AUTO for both workloads. Figure 3.11 shows that the latencies are similar for AUTO-LOGICAL and AUTO for MySQL running the Admissions workload. But the results in Figure 3.12 show BusTracker in PostgreSQL has 38% higher latency with AUTO-LOGICAL. There are two reasons why logical features lead to worse index selection. The first is that the SQL queries are insufficient for determining whether two templates will have similar impacts on the system. The second reason is that templates within the same logical feature cluster may have multiple arrival rate patterns (including anomalies like one-time queries); this makes it more difficult for the Forecaster to identify these patterns and predict the trends according to the cluster centers.

3.6.8 Sensitivity Analysis of ρ

We now the analyze QB5000's cluster coverage ratio and prediction accuracy for different ρ values. As discussed in Section 3.4.2, ρ is the sensitivity threshold used by the on-line clustering component to determine whether a template should belong to a cluster. In these experiments, we perform on-line clustering and arrival rate forecasting for values of ρ ranging from 0.5 to 0.9 for each of the three workloads. Setting ρ=1.0 results in every template having its own cluster, whereas setting it to 0.0 would group all the templates into a single cluster. We set both the prediction interval and the horizon to be one hour for the forecasting. We then measure the cluster coverage ratio and the prediction accuracy for the three highest-volume clusters.

Figure 3.13 shows the average volume ratio between the three largest clusters and the total workload volume for increasing values of ρ. The higher the ρ is, the more similar the templates within the same cluster are. But the number of templates contained in each cluster is also smaller, which means that the three largest clusters cover less of the workload when ρ is high. The results show that the coverage of the largest clusters is stable when ρ increases from 0.5 to 0.8 for all three workloads, but it drops when ρ reaches 0.9.

Figure 3.14 shows the sensitivity of the prediction accuracy to different values of ρ for the three largest clusters. The results show that the prediction accuracy improves as ρ increases for all three workloads. Since the similarity of the templates within the same cluster increases with ρ, the prediction results also improve since the centers of the clusters provide a more accurate representation of the arrival rate patterns captured by each cluster.

Given this analysis, we set ρ=0.8 in our evaluation since we find that it provides the best balance between the prediction accuracy and the coverage ratio for all three of the workloads.

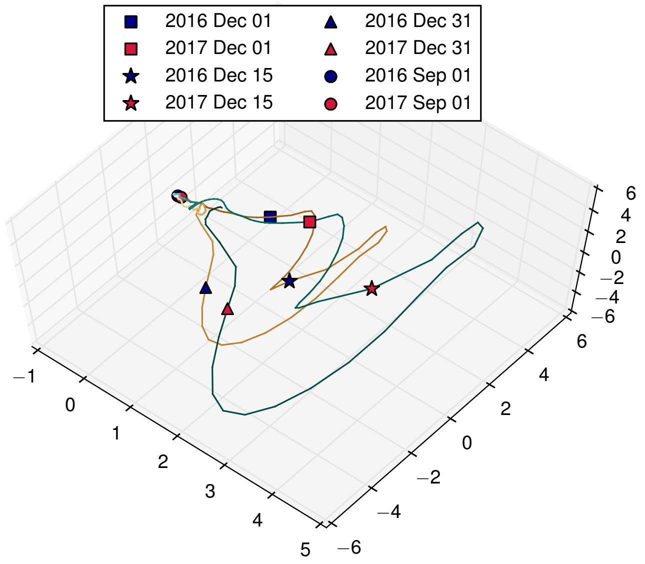
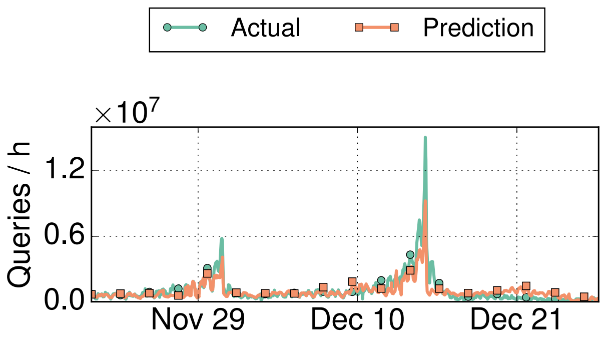


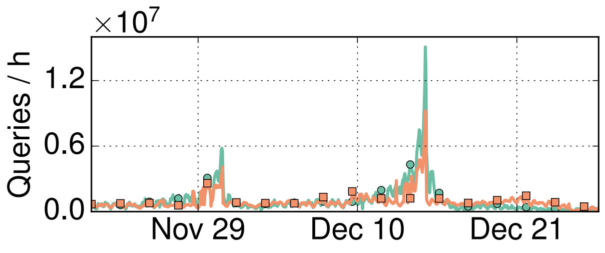
Figure 3.15: Input Space Time-Progress – The 3D-projected input space for the Admissions workload with spikes. The trajectory color follows dates.

3.6.9 Input Space for Spike Prediction

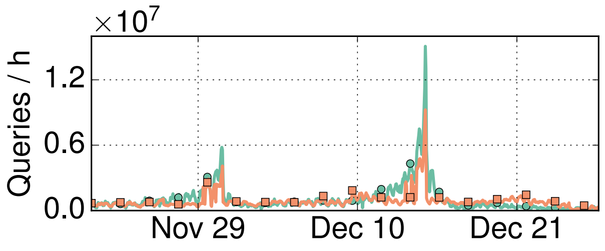
We now demonstrate why KR is able to better predict spikes in a workload compared to the other forecasting models we considered. Recall from Section 3.5.1 that the prediction of KR for a given input is a weighted average of all of the training inputs, where the weights decrease with the distance between the given input and the corresponding training inputs in the kernel space. QB5000 uses the arrival rate history from the previous three weeks (aggregated into one-hour intervals) as the input for KR.



(a) Threshold γ = 100%



(b) Threshold γ = 150%



(c) Threshold γ = 200%

Figure 3.16: Prediction Results – Actual vs. predicted query arrival rates for the combined clusters in the Admissions workload with HYBRID (ENSEMBLE model corrected by KR).

Figure 3.15 shows the inputs for KR at each hour-interval in the Admissions workload trace. The input is projected into 3D space using a common dimensionality reduction technique, PCA [162]. Inputs on nearby dates have similar colors in the trajectory. From this figure, we see the points that correspond to the spike activity are clearly separated from the "normal" activity points. The trajectories from December 1st to December 31st in both 2016 and 2017 travel a much longer distance in the 3D space than in other months. The results also show that inputs that correspond to the same spikes in each of the two years have closer positions in the space. This demonstrates how kernel methods can easily recognize points with high query volumes and predict the future spikes based on the ones observed in the previous year. KR enables QB5000 to predict which dates spikes might occur on throughout the year without explicit domain knowledge.

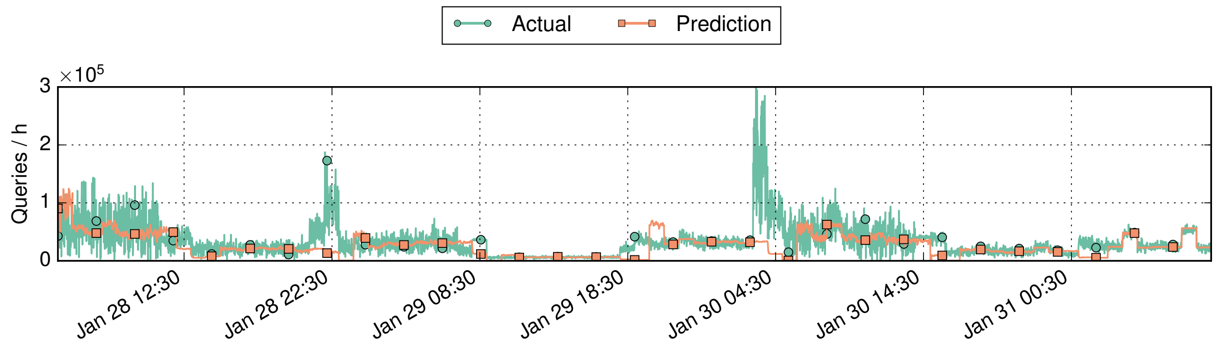


Figure 3.17: Prediction Results – Actual vs. predicted query arrival rates for a synthetic noisy workload.

3.6.10 Sensitivity Analysis of γ

We next analyze the impact of the threshold γ on the prediction performance of QB5000's HYBRID models for workloads with growth and spike patterns. We use the same experimental settings as in Section 3.6.3.

Figure 3.16 shows the combined query arrival rate predictions from all clusters when the threshold γ is set to be 100%, 150%, and 200%. The model is able to predict the major spike patterns for all three settings of γ. When γ is lower, HYBRID uses the result from KR more often than ENSEMBLE, thus improving its ability to predict spikes. However, lower values of γ have increasingly negative impacts on the MSE of the predictions when no spike patterns are present in the workload. In our experiments, we found out that HYBRID has a negligible effect on the MSE of ENSEMBLE when $\gamma > 100\%$. Given this analysis, we set γ to 150% in QB5000.

3.6.11 Predicting Noisy Workloads

In Section 3.1.2, we presented three workload patterns that are prevalent in today's applications. But some workloads may not exhibit strong temporal patterns, and thus their arrival rates from the recent past may differ substantially from the present. This section extends our evaluation of QB5000 by demonstrating its ability to predict the arrival rates of a noisy workload without any temporal patterns to exploit. We use the HYBRID forecast models in this experiment. Such a workload represents a worst-case scenario for QB5000 since the arrival rates are unpredictable.

We constructed a synthetic workload trace that consists of benchmarks from the OLTP-Bench testbed that differ in complexity and system demands [63, 64]. We execute the following eight benchmarks consecutively with varying average arrival rates: Wikipedia, TATP, YCSB, Smallbank, TPCC, Twitter, Epinions, and Voter. Each benchmark is executed for 10 hours. We add white noise to the arrival rate that has a variance set to be 50% of its mean. We also inject random anomalies (i.e., spikes) into the arrival rate of the queries. As we described in Section 3.4.2, QB5000 monitors the ratio of previously unseen templates in the workload, and re-clusters the templates once it detects that the workload has switched from one to another. We set the prediction horizon to be one hour and the prediction interval to be one minute. Since each workload is executed for only 10 hours, QB5000 does not have enough training data to predict long horizons.

The results in Figure 3.17 show the combined predicted arrival rates of all clusters. Each time tick represents when the benchmark shifts from one to another. The results show that QB5000 is able to automatically identify these shifts in the workload and adapt to the new queries. This demonstrates that even when the workload is noisy and the arrival rate is unpredictable, QB5000 is still able to predict the average query volume most of the time.

Chapter 4

Behavior Modeling

An essential component that enables a self-driving DBMS to optimize for the forecasted workload is *behavior models*. These models estimate and explain how the system's performance changes due to a potential *action* (e.g., changing knobs, creating an index). This is similar to how self-driving vehicles use physical models to guide their autonomous planning [156]. But existing DBMSs do not contain the embedded low-level models for self-driving operations, nor do they support generating the training data needed to build such models.

Techniques for constructing database behavior models fall under two categories: (1) "white-box" analytical methods and (2) ML methods. Analytical models use a human-devised formula to describe a DBMS component's behavior, such as the buffer pool or lock manager [148, 151, 239]. These models are customized per DBMS and version. They are difficult to migrate to a new DBMS and require redesign under system updates or reconfiguration. Recent works on using ML methods to construct models have shown that they are more adaptable and scalable than white-box approaches, but they have several limitations. These works mostly target isolated query execution [23, 74, 83, 133, 141]. The models that support concurrent queries focus on real-time settings where the interleaving of queries is known [73, 230, 249], but a self-driving DBMS needs to plan for future workloads without such accurate information [137]. Many ML-based models also rely on dataset or workload-dependent information [73, 141, 203]; thus, a DBMS cannot deploy these models in new environments without expensive retraining.

Given this, we present a framework, called **ModelBot2** (MB2), that generates behavior models that estimate the performance of a self-driving DBMS's components and their interference during concurrent execution. This enables the DBMS's planning components to reason about the expected benefit and the impact of actions. For example, suppose the self-driving DBMS plans to create a new index. In that case, MB2's models can answer how long the index creation is, how the index creation impacts the system performance, and how the new index accelerates the workload's queries.

The main idea of MB2 is to decompose a DBMS's internal architecture into small, independent *operating units* (OUs) (e.g., building a hash table, flushing log records). MB2 then uses ML methods to train an *OU-model* for each OU that predicts its runtime and resource consumption for the current DBMS state. Compared to a single monolithic model for the entire DBMS, these OU-models have smaller input dimensions, require less training time, and provide performance insight to each DBMS component [85]. During inference, MB2 combines

OU-models to predict the DBMS's performance for the future workload (which we assume is provided by workload forecasting techniques [137]) and the system state. To support multi-core environments with concurrent threads, MB2 also estimates the interference between OUs by defining the OU-models' outputs as a set of measurable performance metrics that summarizes each OU's behavior. MB2 then builds *interference models* for concurrent OUs based on these metrics. MB2 also provides a principled method for data generation and training for self-driving DBMSs: developers create *offline* runners that exercise the system's OUs under various conditions, and MB2 uses the runner-produced data to train a set of workload and dataset independent OU-models.

To evaluate our approach, we implement our framework into the **NoisePage** DBMS [38] and measure its ability to model its runtime components and predict its behavior using workload forecasts. We also compare against a state-of-the-art external modeling approach based on deep learning [141]. Our results show that our models support OLTP and OLAP workloads with a minimal loss of accuracy.

4.1 Background

Behavior models are the foundation for building a self-driving DBMS since high-fidelity models form the basis of robust planning and control. We now provide an overview of the salient aspects of modeling for self-driving DBMSs. We then discuss the limitations and unsolved challenges with existing approaches.

4.1.1 Behavior Modeling

Given an action, a self-driving DBMS's behavior models estimate (1) how long the action takes, (2) how much resource the action consumes, (3) how applying the action impacts the system performance, and (4) how the action impacts the system once it is deployed. Such models can also provide explanations about the self-driving DBMS's decisions and debug potential issues.

Analytical models use a human-devised formula to describe a DBMS component's behavior, such as the buffer pool or the lock manager [148, 151, 239]. The unknown variables in the formula generally come from a workload specification. The models (formulas) are disparate for different DBMSs, components, and algorithms. For example, these works have devised drastically different I/O formulas for MySQL and SQL Server [148, 151]. Thus, it is challenging and onerous to revise these models for new DBMSs or DBMS updates.

Recent works show promising results using ML to model query execution in analytical workloads [74, 133, 141]. These ML-based models use query plan information (e.g., cardinality estimates) as input features to estimate system performance metrics (e.g., query latency). Although ML-based models' are potentially easier to transfer to new environments, they still require feature adjustments, data remaking, and retraining. Existing models also do not support transactional workloads, nor do they consider the effects of the DBMS's maintenance operations (e.g., garbage collection).

Some methods also support concurrent queries when the arrival time of each query is known. They derive the concrete interleaving of queries as the input for their analytical or

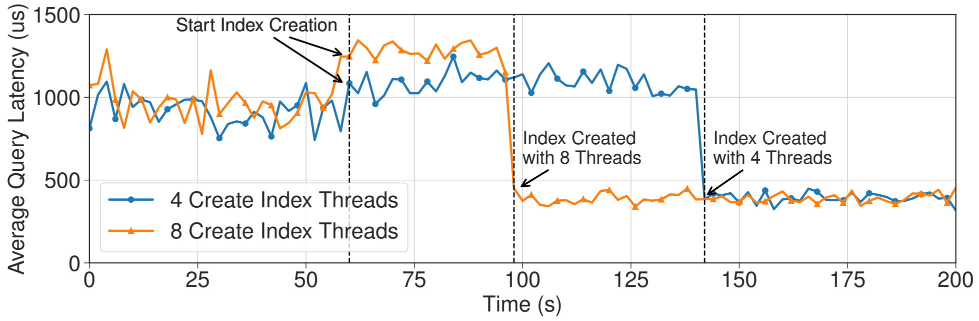


Figure 4.1: Index Build Example – TPC-C query latency running on NoisePage. The DBMS begins index building after 60s using 4 or 8 threads.

ML models [73, 230, 249]. This is insufficient for a self-driving DBMS because it must account for unknown future workloads when planning expensive actions.

To illustrate the difficulties inherent in self-driving DBMS modeling, consider the scenario of when a DBMS must decide whether to build an index. For this example, we use the TPC-C benchmark running on NoisePage; we provide our experimental details in Section 4.7. The results in Figure 4.1 show the query latency for the TPC-C workload when we remove a secondary index on the `CUSTOMER` table. After 60s, the DBMS begins adding that index back to improve the latency. However, before it can start, the DBMS's planning component uses behavior models to identify what benefit (if any) adding that index would provide. The planning component also must select how many threads to use to build the index; using more threads will decrease the build time but degrade the system's performance. For example, Figure 4.1 shows that building with four threads only degrades performance by 25%, but it takes 80s to finish, whereas using eight threads degrades performance by 32% but completes in 40s. A DBMS needs this information to determine which action deployment to choose based on the environment and constraints.

4.1.2 Challenges

Despite the advantages of using ML to build models, there are several challenges in using them in a self-driving DBMS.

High Dimensionality: One approach for behavior modeling is to build a monolithic model that captures all aspects of the DBMS, including its workload, configuration, and actions. Although this is conceptually clean, it incurs the "curse of dimensionality" problem where an ML model's predictive power decreases as the number of dimensions or features increases [210]. Even if the model only targets query execution, a modern DBMS will still have hundreds of plan operator features [23, 141]. Naïvely concatenating these features into the model will lead to sparse input and weak predictive efficacy. Partially because of this, modeling techniques target individual operators instead of the entire query plan [133, 141]. A self-driving DBMS must also consider its runtime state (e.g., database contents, knob configurations), interactions with other components (e.g., garbage collection), and other autonomous actions (e.g., building indexes), which further increases dimensionality.

Concurrent Operations: Since queries and DBMS components that run simultaneously will interfere with each other, a self-driving DBMS needs to model such interference in dynamic environments. A simple approach is to duplicate the input features by the maximum degree of concurrency (e.g., number of threads). This multiplicatively increases feature dimensionality and exponentially increases the possible input features for the models. For example, a workload with 10 different queries running on a 20 core machine has 11^{20} possible input feature combinations. Queries may also incur interference that is not easily identifiable solely on plan features, such as the resource contention between concurrent queries.

Training, Generalizability, and Explainability: Collecting sufficient training data to build ML models for DBMSs is non-trivial when there are many features with large domains. Part of this difficulty is because some DBMS operations take a long time to complete. Thus, collecting this data is expensive, which is a key problem for many ML methods [169]. For example, building indexes can take hours [58], limiting the amount of training data that a DBMS can collect. Most of the previous work on using ML to predict query execution times rely on synthetic benchmarks for training [133, 138, 141, 216]. Although their models perform well for the same workload used for training, they have high prediction errors on different workloads. The behavior models should also be explainable and debuggable to facilitate their practical application [122].

4.2 Overview

MB2 is an embedded behavior modeling framework for self-driving DBMSs. There are two key design considerations: (1) since collecting training data, building models, and diagnosing problems are expensive and time-consuming, MB2 generates models offline in a dataset and workload independent manner. The DBMS then uses the same set of models to estimate the runtime actions' impact on any workload or dataset. (2) MB2's models are debuggable, explainable, and adaptable. These qualities reduce development complexity and provide a view of why the DBMS chooses specific actions.

Figure 4.2 provides an overview of MB2's modeling architecture. The main idea is to decompose the DBMS into independent *operating units* (OUs). An OU represents a step that the DBMS performs to complete a specific task. These tasks include query execution steps (e.g., building join hash tables (JHTs)) and internal maintenance steps (e.g., garbage collection). DBMS developers are responsible for creating OUs and deciding their boundaries based on the system's implementation. We decompose NoisePage naturally following the system's source code organization. For example, NoisePage defines an OU for the function that builds a hash table.

MB2 pairs each OU with an *OU-runner* that exercises the OU's corresponding DBMS component by sweeping the component's input parameter space. MB2 then provides a lightweight data collection layer to transform these OU-runners' inputs to OU features and track the OU's behavior metrics (e.g., runtime, CPU utilization). For each OU, MB2 automatically searches, trains, and validates an *OU-model* using the data collected from the related OU-runner.

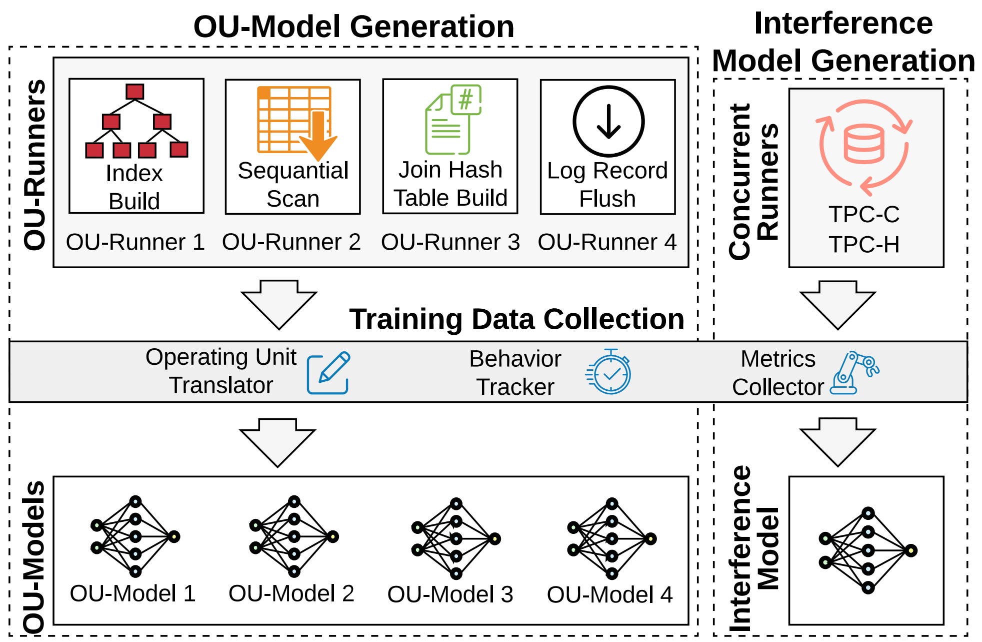
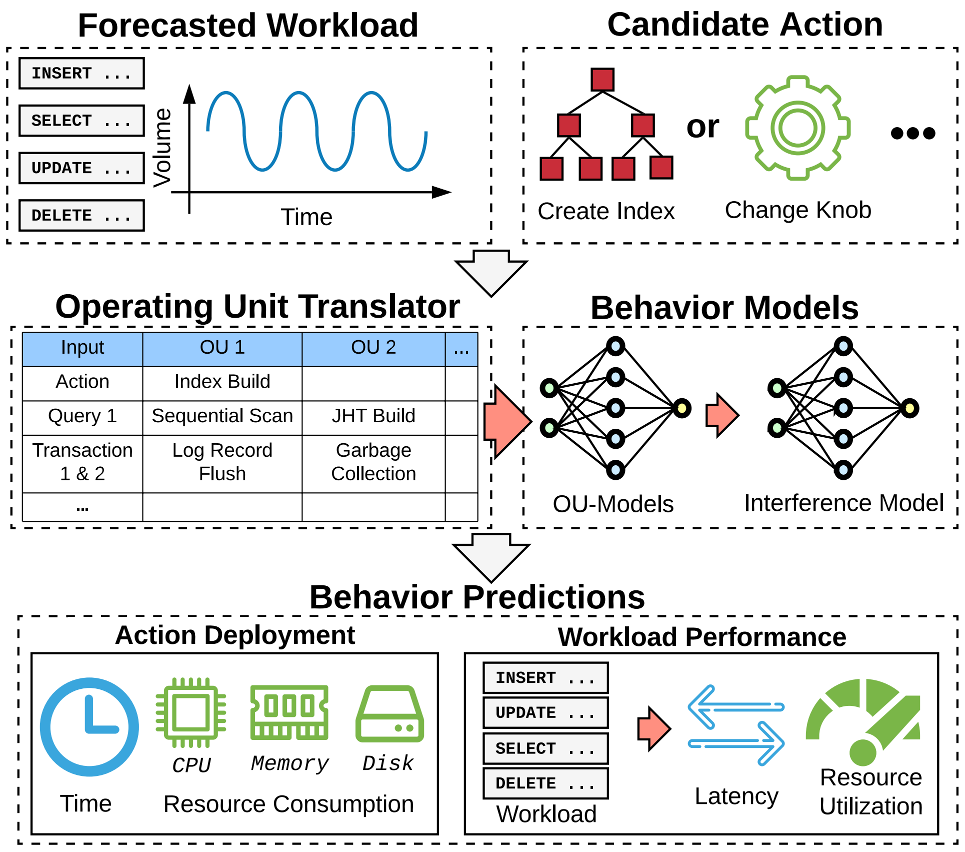


Figure 4.2: System Architecture – MB2 trains OU-models for each OU with the data generated by OU-runners and builds interference models for concurrent OUs based on system resource utilization.

To orchestrate data collection across all OUs and to simulate concurrent environments, MB2 uses *concurrent runners* to execute end-to-end workloads (e.g., benchmarks, query traces) with multiple threads. MB2 uses its training data collectors to build *interference models* that estimate the impact of resource competition, cache locality, and internal contention among concurrent OUs.

With OU-models trained for the entire system, a self-driving DBMS can use them as a simulator to approximate its runtime behavior. Figure 4.3 shows how a DBMS uses MB2's models for inference. The inputs for MB2 are the forecasted workload and a potential action. Employing the same translator infrastructure for training data collection, MB2 first extracts the OUs from the inputs and generates their model features. It then uses OU-models to predict the behavior of each OU, and uses the interference models to adjust the OU-model's prediction to account for the impact of concurrent OUs. Finally, MB2 sums OU's prediction to derive information that guides the DBMS's planning system, such as how long the action takes and its impacts on the forecasted workload's performance.

Returning to our example from Section 4.1.1, before a self-driving DBMS chooses the action to build the index, it can use MB2's models to estimate the time and consumed resources (e.g., CPU, memory) for the action's OUs. MB2 also estimates the effect of building the index on the regular workload by converting its queries into OUs and predicting their performance. The DBMS's planning system then decides whether to build this index and provides explanations for its decision based on these detailed predictions.



Input	OU 1	OU 2	...
Action	Index Build		
Query 1	Sequential Scan	JHT Build	
Transaction 1 & 2	Log Record Flush	Garbage Collection	
...			

Figure 4.3: MB2 Inference Procedure – Given the forecasted workload and a self-driving action, MB2 records the OUs for all input tasks, and uses the OU-models and interference models to predict the DBMS's behavior.

Assumptions and Limitations: We now clarify MB2's capabilities on what it can and cannot do. Foremost is that we assume the framework uses a forecasting system to generate estimations for future workload arrival rates in fixed intervals (e.g., a minute/hour) [137]. The workload forecasting system cannot predict ad-hoc queries it has never seen before. Thus, we assume the DBMS executes queries with a cached query plan except for the initial invocation.

We assume that the target system is an in-memory DBMS; MB2 does not support disk-oriented DBMSs with buffer pools. This assumption simplifies MB2's behavior models since it does not have to consider what pages could be in memory for each query. Estimating cache contents is difficult enough for a single query. It is more challenging when evaluating a sequence of forecasted queries.

MB2 supports both OLTP and OLAP workloads, as well as mixed workloads. Assuming an in-memory DBMS makes it easier to model OLAP workloads since each query's performance is affected by aspects of the system that are observable by MB2: (1) database contents, (2) configuration, and (3) operating environment. Modeling OLTP queries poses additional challenges due to higher variance with short query execution times. Supporting transactions also means that MB2 must consider logical contention (e.g., transactions updating the same record) as well as physical contention for hardware resources. We assume that the DBMS uses MVCC [233] and MB2 supports capturing lock contention. MB2 does not, however, model transaction aborts due to data conflicts because it is challenging to get precise forecasts of overlapping queries.

MB2's OU-models' input features contain the cardinality estimation from the DBMS optimizer, which is known to be error-prone [130]. Our evaluation shows that MB2's prediction is insensitive against cardinality estimation errors within a reasonable range (30%). There are recent works that use ML to improve an optimizer's cardinality estimations [75, 112, 227, 228, 235], which MB2 may leverage.

Lastly, while MB2 supports hardware context in its models (see Section 4.3.2), we defer the investigation on what features to include for different hardware (e.g., CPU, disk) and environments (e.g., bare-metal, container) as future work.

4.3 OU-models

We now discuss how to create a DBMS's OU-models with MB2. The goal of these models is to estimate the time and resources that the DBMS will consume to execute a query or action. A self-driving DBMS can make proper planning decisions by combining these estimates from multiple queries in a workload forecast interval. In addition to accuracy, these models need to have three properties that are important for self-driving operations: (1) they provide explanations of the DBMS's behavior, (2) they support any dataset and workload, and (3) they adapt to DBMS software updates.

4.3.1 Principles

Developers use MB2 to decompose the DBMS into OUs to build explainable, adaptable, and workload independent behavior models. The first step is to understand the principles for dividing the DBMS's internal components into these OUs. For this discussion, we use examples from NoisePage's OUs shown in Table 4.1:

Independent: The runtime behavior of an OU must be independent of other OUs. Thus, changes to one OU do not directly affect another unrelated OU. For example, if the DBMS changes the knob that controls the join hash table size then this does not change the resource consumption of the DBMS's WAL component or the resource consumption of sequential scans for the same queries.

Low-dimensional: An OU is a basic operation in the DBMS with a small number of input features. That is, the DBMS can construct a model that predicts the behavior of that part of the system with as few features as possible. We have found that it is best to limit the number of features to at most ten, which includes any hardware and database state contexts. If an OU requires more than this, then one should attempt to divide the OU into sub-OUs. This limit may increase as ML algorithms and hardware improve over time. Restricting the number of features per OU improves the framework's ability to collect sufficient training data to support any workload.

	Operating Unit	Features	Knobs	Type
Execution	Sequential Scan	7	1	Singular
	Index Scan	7	1	Singular
	Join Hash Table Build	7	1	Singular
	Join Hash Table Probe	7	1	Singular
	Agg. Hash Table Build	7	1	Singular
	Agg. Hash Table Probe	7	1	Singular
	Sort Build	7	1	Singular
	Sort Iterate	7	1	Singular
	Insert Tuple	7	1	Singular
	Update Tuple	7	1	Singular
	Delete Tuple	7	1	Singular
	Arithmetic or Filter	2	1	Singular
Util	Garbage Collection	3	1	Batch
	Index Build	5	1	Contending
WAL	Log Record Serialize	4	1	Batch
	Log Record Flush	3	1	Batch
Txns	Transaction Begin	2	0	Contending
	Transaction Commit	2	0	Contending
NET	Output Result	7	1	Singular

Table 4.1: Operating Unit – Property Summary of OUs in NoisePage.

Comprehensive: Lastly, the framework must have OUs that encompass all DBMS operations which consume resources. Thus, for any workload, the OUs cover the entire DBMS, including background maintenance tasks (e.g., garbage collection) and self-driving actions that the DBMS may deploy on its own (e.g., index creation).

A DBMS will have multiple OU decompositions according to these principles, which allows for flexible implementation choices. Adding more OUs may lead to smaller models with finer-grained predictions and less required training data for each model. But additional OUs may also increase the inference time, model maintenance cost, and stacking of prediction errors. We defer the problem of deriving the optimal OU set for a DBMS as future work.

4.3.2 Input Features

After deciding which OUs to implement, DBMS developers then specify the OU-models' input features based on their *degree of freedom* [69]. These features may contain (1) the amount of work for a single OU invocation or multiple OU invocations in a batch (e.g., the number of tuples to process), (2) the parallel invocation status of an OU (e.g., number of threads to create an index), and (3) the DBMS configuration knobs (e.g., the execution mode). Although humans select the features for each OU-model, the features' values are generated automatically by the DBMS based on the workload and actions. Some features are generic and will be the same

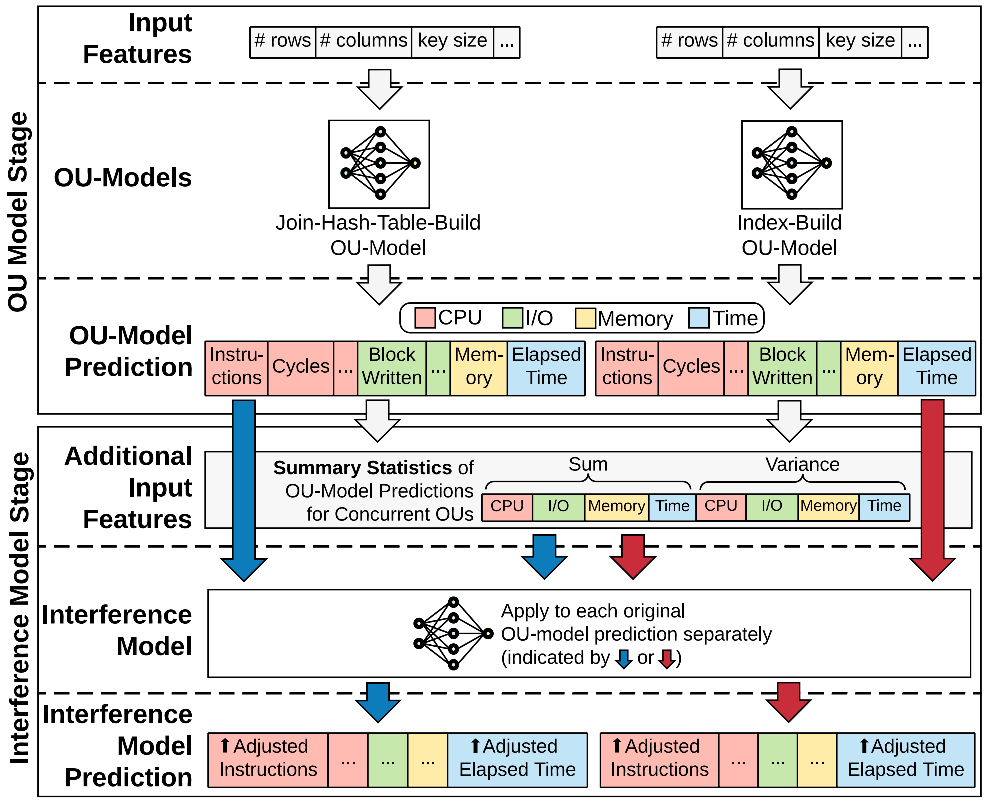


Figure 4.4: OU and Interference Models – MB2 uses OU-specific input features to predict their resource consumption and elapsed time. The interference model uses the summary predictions for concurrent OUs.

across many DBMSs, whereas others are specific to a DBMS's implementation. We categorize OUs into three types based on their behavior pattern, which impacts what information the input features have.

Singular OUs: The first type of OUs have input features that represent the amount of work and resource consumption for a single invocation. These include NoisePage's execution category OUs in Table 4.1. Almost all its execution OUs have the same seven input features. The first six features are related to the relational operator that the OU belongs to: (1) number of input tuples, (2) number of columns of input tuples, (3) average input tuple size, (4) estimated key cardinality (e.g., sorting, joins), (5) payload size (e.g., hash table entry size for hash joins), and (6) number of loops (only for index nested loop joins). The first three features indicate the tuple volume that the OU will process as an estimate of the amount of work it will perform. Likewise, the fourth and fifth features approximate the expected output, which helps determine the intermediate state that the OU maintains during execution. The sixth feature indicates whether the OUs is repeatedly executed in a loop, which helps capture short OUs' caching effect. Lastly, the seventh feature is an execution mode flag that is specific to NoisePage; this indicates whether the DBMS executes a query with its interpreter or as JIT-compiled [116, 146]. NoisePage's networking OU also belongs to this type since network communication is discrete amount of work.

Batch OUs: The second type of OUs have input features that represent a batch of work across multiple OU invocations. It is challenging to derive features for these OUs since a single invocation may span multiple queries based on when those queries arrive and the invocation interval. For example, the OU for log flushes will write the log records generated from queries since the last invocation, the timing of which is unknown. To address this, we define the log flush OU's input features to represent the total amount of records generated by the set of queries predicted to arrive in a workload forecasting interval: (1) the total number of bytes, (2) the total number of log buffers, and (3) the log flush interval. These features are independent of what plans the DBMS chooses for each query.

Contending OUs: The last type of OUs are for operations that may incur contention in parallel invocations. For example, the DBMS can use multiple threads to build an index, but the threads have to acquire latches in the data structure. We include the contention information into these OUs input features. For the index build OU, its input features include: (1) number of tuples, (2) number of keys, (3) size of keys, (4) estimated cardinality of the keys, and (5) the number of parallel threads, which indicates the contention[1]. Similarly, the OUs related to starting and ending transactions also contain information about transactions' arrival rates in the workload forecast interval. The internal contention is orthogonal to the concurrent impact that MB2's interference model captures based on resource consumption that we will discuss in Section 4.4.

A self-driving DBMS must also predict how changes to its configuration knobs impact its OUs. We categorize these tunable knobs into *behavior knobs* and *resource knobs*. The former affects the internal behavior of one or more OUs, such as the execution mode and log flushing interval. As shown in these examples, MB2 appends behavior knobs to the impacted OUs' features. Thus, the OU-models can predict the OU behavior when changing these knobs. MB2 can also append the hardware context to the OU features to generalize the OU-models across hardware. We demonstrate such ability by extending the OU features with one hardware feature, i.e., the CPU frequency (Section 4.7.6). We leave a thorough investigation for the proper hardware context features as future work.

4.3.3 Output Labels

Every OU-model produces the same output labels. They are a vector of commonly available hardware metrics (Section 4.5.1) that approximate the OU's behavior per invocation (i.e., for a single set of input features): (1) elapsed time, (2) CPU time, (3) CPU cycles, (4) CPU instructions, (5) CPU cache references, (6) CPU cache misses, (7) disk block reads, (8) disk block writes (for logging), and (9) memory consumption. These metrics explain what work an OU does independent of which OU it is. Using the same labels enables MB2 to combine them together to predict the interference among concurrent OUs. They also help the self-driving DBMS estimate the

[1]For parallel OUs, including the index build, MB2 uses the max (instead of the sum) predicted elapsed time among each single-threaded invocation as the elapsed time.

impact of knobs for resource allocation. For example, the OU-models can predict each query's memory consumption by predicting the memory consumption for all the OUs related to the query. A self-driving DBMS evaluates what queries can execute under the knob that limits the working memory for each query, and then sets that knob according to its memory budget. Similarly, CPU usage predictions help a self-driving DBMS evaluate whether it has assigned enough CPU resources for queries or its internal components.

Output Label Normalization: We now discuss how MB2 normalizes OU-models' output labels by tuple counts to improve their accuracy and efficiency. DBMSs can execute queries that range from accessing a single tuple (i.e., OLTP workloads) to scanning billions of tuples (i.e., OLAP workloads). The former is fast to replicate with OU-runners, but the latter is expensive and time-consuming. To overcome this issue, MB2 employs a normalization method inspired by a previous approach for scaling query execution modeling [133]. We observe that many OUs have a known complexity based on the number of tuples processed, which we denote as n. For example, if we fix all the input features except for n, the time and resources required to build a hash table for a join are $\mathcal{O}(n)$. Likewise, the time and resources required to build buffers for sorting are $\mathcal{O}(n \log n)$. We have found in our evaluations with NoisePage that with typically less than a million tuples, the output labels converge to the OU's asymptotic complexity multiplied by a constant factor.

Given this, for OU-models that have the number of processed tuples/records as an input feature, MB2 normalizes the outputs. It divides the outputs by the related OU's complexity based on the number of tuples, while the inputs remain intact. A special case is the memory consumption label for building hash tables: NoisePage uses different hash table implementations for joins and aggregations. The hash table pre-allocates memory for joins based on the number of tuples; for aggregations, the hash table grows with more inserted unique keys. Thus, to normalize this label, MB2 divides it by the number of input tuples for the join hash table OU and by the cardinality feature for the aggregation hash table OU.

With such normalization, MB2 only needs to collect training data for OU-models with the number of tuples up to the convergence point. Although previous work on analytical DBMS models also leverages similar database domain knowledge [148, 151], MB2's normalization to OU-models is simple to implement regardless of the OU's implementation and is easy to adapt when there are system updates (Section 4.6). We demonstrate in Section 4.7 that MB2 generalizes to datasets with orders of magnitude higher number of tuples than what exists in the training data using this approach.

4.4 Interference Model

The OU-models capture the internal contention on data structures or latches within an OU (Section 4.3.2). But there can also be interference among OUs due to resource competition, such as CPU, I/O, and memory.[2] MB2 builds a common interference model to capture such

[2]We observe that in corner cases, there can also be resource sharing among OUs, such as the cache locality across consecutively running OUs. MB2 does not model such interference since it has little impact on our evaluation.

interference since it may impact any OU.

As shown in Figure 4.4, MB2 first extracts all OUs from the forecasted workload and the planned action before using OU-models to estimate their output labels. It then applies the interference model to each OU separately to adjust the OU-model's output labels based on the workload forecast's concurrency information and the OU-model predictions for other OUs. Building such an interference model has two challenges: (1) there are an exponential number of concurrent OU combinations, and (2) self-driving DBMSs need to plan actions ahead of time [160], but predicting queries' exact future arrival times and concurrent interleavings is arguably impossible.

To address these challenges, we formulate a model based on *summary statistics* [69] of the unified behavior metrics estimated by the OU-models. Given that all the OU-models have the same output labels, the interference model uses a fixed set of input features that summarize the OU-model predictions regardless of the underlying OUs. Furthermore, the summary statistics aggregate the behavior information of the OUs forecasted to run in an interval, which does not require the exact arrival time of each query. Figure 4.4 illustrates the formulation of the resource competition interference model.

4.4.1 Input Features

The interference model's inputs are the OU-model's output labels for the OU to predict and summary statistics of the OUs forecasted to run in the same interval (e.g., one minute). MB2 adds the OU-models' output labels for the OUs assigned to run on each thread separately and uses the sum and variance for each thread's total labels as the summary. Although MB2 can include other summaries, such as percentiles of concurrent OUs' OU-model predictions, we find the above summary effective in our evaluation. MB2 also normalizes the interference model's inputs by dividing them by the target OU-model's estimated elapsed time to help generalization.

4.4.2 Output Labels

The interference model generates the same set of outputs as the OU-models. Instead of estimating the absolute values of the output metrics, the model predicts the element-wise ratios between the actual metrics and the OU-model's prediction. These ratios are always greater than or equal to one since OUs run the fastest in isolation. We observe that under the same concurrent environment, OUs with similar per-time-unit OU-model estimation (part of the interference model's inputs) experience similar impacts and have similar output ratios regardless of the absolute elapsed time. Thus, the combination of normalizing the inputs by the elapsed time and using the ratios as the outputs helps the model generalize to OUs with various elapsed times.

4.5 Data Generation and Training

MB2 is an end-to-end solution that enables self-driving DBMSs to generate training data from OUs and then build accurate models that predict their behavior. We now describe how MB2 facilitates this data collection and model training. We begin with discussing MB2's internal components that developers must integrate into their DBMS. We then present our OU- and concurrent-runner infrastructure that exercises the system to produce training data.

We again emphasize that this training data collection process uses the DBMS in an offline manner. That is, developers run the system in non-production environments. We defer the problem of how to collect this training data for an on-line system without incurring observable performance degradation as future work.

4.5.1 Data Collection Infrastructure

MB2 provides a lightweight data collection framework that system developers integrate into their DBMS. We first describe how to set up the system to collect the behavior data of OUs (i.e., input features, output labels). We then describe the runtime mechanisms that MB2 uses for tracking each OU's resource consumption. Lastly, we discuss how the framework retrieves this data for model training.

OU Translator: This component extracts OUs from query and action plans and then generates their corresponding input features. MB2 uses the same translator infrastructure for both offline training data collection and runtime inference.

Resource Tracker: Next, MB2's tracker records the elapsed time and resource consumption metrics (i.e., output labels) during OUs execution. The framework also uses this method for the interference model data since it uses the same output labels to adjust the OU-models' outputs. MB2 enables this resource tracking right before the invocation of an OU, and then disables it after the OU completes. The tracker uses a combination of user- and kernel-level primitives for recording a OU's actions during execution. For example, it uses C++11's `std::chrono` high-resolution clock to record the elapsed time of the OU. To retrieve hardware counter information, MB2's uses the Linux `perf` library and the `rusage` syscall. Although these tracking methods do not require the DBMS to run with root privileges to collect the data, they do add some amount of runtime overhead to the system. Investigating more customized and lightweight methods for resource tracking is future work [10, 13].

Metrics Collector: The challenges with collecting training data are that (1) multiple threads produce metrics and thus require coordination, and (2) *resource tracker* can incur a noticeable cost. It is important for MB2 to support low-overhead metrics collection to reduce the cost of accumulating training data and interference with the behavior of OUs, especially in concurrent environments.

MB2 uses a decentralized *metrics collector* to address the first issue. When the DBMS executes in training mode, a worker thread records the features and metrics for each OU that it encounters in its thread-local memory. MB2 then uses a dedicated aggregator to periodically gather this data from the threads and store it in the DBMS's training data repository. To address the second challenge, MB2 supports resource tracking only for a subset of queries or

DBMS components. For example, when MB2 collects training data for the OUs in the execution engine, the DBMS can turn off the tracker and metrics collector for other components.

4.5.2 OU-Runners

An *OU-runner* is a specialized microbenchmark in MB2 that exercises a single OU. The goal of each OU-runner is to reproduce situations that an OU could encounter when the DBMS executes a real application's workload. The OU-runners sweep the input feature space (i.e., number of rows, columns, and column types) for each OU with fixed-length and exponential step sizes, which is similar to the *grid search* optimization [31]. For example, with singular OUs related to query execution, the OU-runner would evaluate different combinations of tuple sizes and column types. Although it not possible to exercise every possible variation for some OU's, MB2's output label normalization technique (Section 4.3.3) reduces the OU-runners' need to consider large data sets.

There are two ways to implement OU-runners: (1) low-level execution code that uses the DBMS's internal API and (2) high-level SQL statements. We chose the latter for NoisePage because it requires less upfront engineering effort to implement them, and has little to no maintenance costs if the DBMS's internal API changes.

MB2 supports modeling OLTP and OLAP workloads. To the best of our knowledge, we are the first to support both workload- and data-independent modeling for OLTP query execution. Prior work either focused on modeling OLAP workloads [133, 141, 230] or assumes a fixed set of queries/stored procedures in the workload [148, 151]. Modeling the query execution in OLTP workloads is challenging for in-memory DBMSs: since OLTP queries access a small number of tuples in a short amount of time, spikes in hardware performance (e.g., CPU scaling), background noise (e.g., OS kernel tasks), and the precision of the resource trackers (e.g., hardware counters) can inflict large variance on query performance. Furthermore, DBMSs typically execute repeated OLTP queries as prepared statements.

To address the first challenge on variability, MB2 executes the OU-runners for the OUs in the execution engine with sufficient repetitions ($10\times$) and applies *robust statistics* [104] to derive a reliable measurement of the OU's behavior. Robust statistics can handle a high proportion of outliers in the dataset before giving an incorrect (e.g., arbitrarily large) result, where such proportion is called the *breakdown point*. MB2 uses the 20% trimmed mean statistics [198], which has a high breakdown point (i.e., 0.4), to derive the label from the repeated measurements. To address the second challenge, MB2 executes each query for five warm-up iterations before taking measurements for the query's OUs, with all executions of a given query using the same query template. MB2 starts a new transaction for each execution to avoid the data residing in CPU caches. For queries that modify the DBMS state, MB2 reverts the query's changes using transaction rollbacks. We find the labels insensitive to the trimmed mean percentage and the number of warm-up iterations.

4.5.3 Concurrent Runners

Since OU-runners invoke their SQL queries one at a time, MB2 also provides *concurrent runners* that execute end-to-end benchmarks (e.g., TPC-C, TPC-H). These concurrent runners provide

MB2 with the necessary data to train its interference model (Section 4.4).

To generate training data with diverse concurrent execution profiles, each concurrent runner executes their workload by varying three parameters: (1) the subsets of queries in the benchmark to execute, (2) the number of concurrent threads that the DBMS uses, and (3) the workload submission rate. The concurrent runners execute their workload with each parameter combination for a brief period of time (e.g., 30s) in correspondence to the short-term prediction intervals used by the DBMS's workload forecasting framework [137]. As discussed in Section 4.4.2, MB2's interference model is agnostic to the OU elapsed time, so the concurrent runners do not need to execute the workloads at different interval lengths.

4.5.4 Model Training

Lastly, we discuss how MB2 trains its behavior models using the runner-generated data. Since OUs have different input features and behaviors, they may require using different ML algorithms that are better at handling their unique properties and assumptions about their behavior. For example, Huber regression (a variant of linear regression) is simple enough to model the filter OUs with arithmetic operations. In contrast, sorting and hash join OUs require more complex models, such as random forests, to support their behaviors under different key number, key size, and cardinality combinations.

MB2 trains multiple models per OU and then automatically selects the one with the best accuracy for each OU. MB2 currently supports seven ML algorithms for its models: (1) linear regression [186], (2) Huber regression [104], (3) support vector machine [194], (4) kernel regression [149], (5) random forest [135], (6) gradient boosting machine [82], and (7) deep neural network [175]. For each OU, MB2 first trains an ML model using each algorithm under the common 80/20% train/test data split and then uses cross-validation to determine the best ML algorithm to use [115]. MB2 then trains a final OU-model with all the available training data using the best ML algorithm determined previously. Thus, MB2 utilizes all the data that the runners collect to build the models. MB2 uses this same procedure to train its interference models.

4.6 Handling Software Updates

DBMSs with an active install base likely have software updates to fix bugs, improve performance, and add new features. If its behavior models cannot keep up with these changes, the DBMS will be unable to adjust itself correctly . To handle software updates, MB2 only needs to run the OU-runners for the affected OUs. This restricted retraining is possible because the OUs are independent of each other. The OU-runners issue SQL queries to the DBMS to exercise the OUs, which means that developers do not need to update them unless there are changes to the DBMS's SQL syntax. Furthermore, MB2 does not need to retrain its interference models in most cases because resource competition is not specific to any OU.

If a DBMS update contains changes that introduces new OU behaviors (e.g., adding a new DBMS component), then MB2 re-runs the concurrent runners to generate the interference models. In NoisePage, we currently use a heuristic for MB2 to retrain the interference models when

Model Type	Runner Time	Data Size	Training Time	Model Size
OUs	514 min	38 MB	18 min	338 MB
Interference	82 min	236 MB	21 min	66 KB

Table 4.2: MB2 Overhead – Behavior Model Computation and Storage Cost.

a DBMS update affects at least five OUs.

4.7 Experimental Evaluation

We now discuss our evaluation of MB2 using the NoisePage DBMS. We deployed the DBMS on a Ubuntu 18.04 LTS machine with two 10-core Intel Xeon E5-2630v4 CPUs, 128GB of DRAM, and Intel Optane DC P4800X SSD.

We use the OLTP-Bench [64] testbed as an end-to-end workload generator for the SmallBank [25], TATP [153], TPC-C [208], and TPC-H [209] benchmarks. SmallBank is an OLTP workload that consists of three tables and five transactions that models customers interacting with a bank branch. TATP is an OLTP workload with four tables and seven transactions for a cellphone registration service. TPC-C is a more complex OLTP benchmark with nine tables and five transactions that models back-end warehouses fulfilling orders for a merchant. Lastly, TPC-H is an OLAP benchmark with eight tables and queries that model a business analytics workload. We use `numactl` to fix the DBMS and OLTP-Bench processes on separate CPU sockets. We also set the Xeon CPUs' power governor setting to "performance" mode to reduce the variance in our measurements.

We use two evaluation metrics. For OLAP workloads, we use the average *relative error* ($\frac{|Actual - Predict|}{Actual}$) used in similar prediction tasks in previous work [133, 141]. Since OLTP queries have short run-times with high variance, their relative errors are too noisy to have a meaningful interpretation. Thus, we use the average *absolute error* ($|Actual - Predict|$) per OLTP query template.

We implemented MB2's models with `scikit-learn` [14]. We use the default hyperparameters except for random forest with 50 estimators, neural network with 2 layers with 25 neurons, and gradient boosting machine with 20 depth and 1000 leaves. We also implemented MB2's OU-runners using Google's Benchmark library [11] with `libpqxx` [12] to connect to NoisePage.

4.7.1 Data Collection and Training

We first discuss MB2's data collection and model training. The results in Table 4.2 show a breakdown between the time that MB2 spends generating data versus the time that it spends training its models. We generated $\sim$1M unique data points for NoisePage's 19 OUs. Exercising the OU-runners is the most costly step because (1) OU-runners enumerate a wide range of feature combinations, and (2) certain data points are expensive to collect (e.g., building a hash table with 1m tuples). The model training step does not take too much time because we designed each OU-model to have a small number of features. Table 4.2 also shows that the concurrent runners produce a larger data set than the OU-runners because they execute multiple OUs at

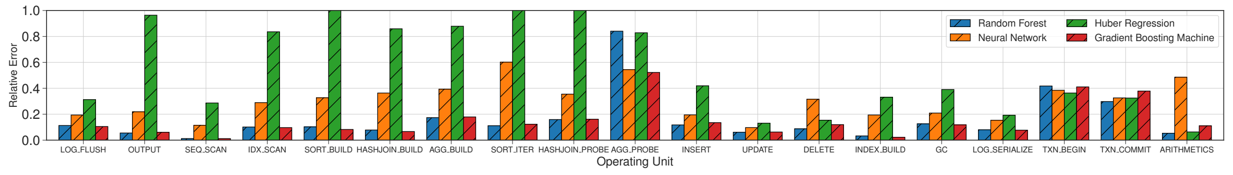


Figure 4.5: OU-model Accuracy (OU) – Test relative error for each OU averaging across all OU-model output labels. OU-models are trained with four ML algorithms: (1) random forest, (2) neural network, (3) Huber regression, and (4) gradient boosting machine.

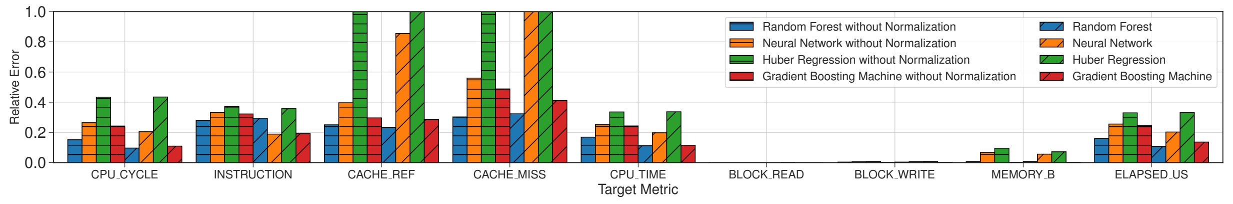


Figure 4.6: OU-model Accuracy (Output Labels) – Test relative error for each output label averaging across all OUs. OU-models are trained with four ML algorithms with and without output label normalization: (1) random forest, (2) neural network, (3) Huber regression, and (4) gradient boosting machine.

the same time. The model is much smaller since, unlike OU-models with one model for each OU, MB2 generated a single interference model that is not specific to any OU.

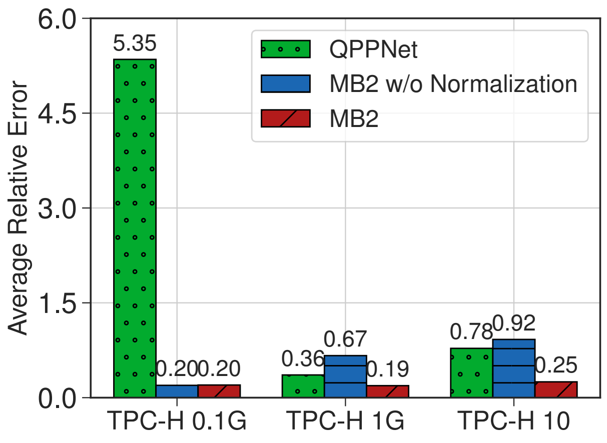
In our experiments, OU translator and OU-model inference for a single query (may contain multiple OUs) on average take 10μs and 0.5ms. Each resource tracker invocation on average takes 20μs.

4.7.2 OU-Model Accuracy

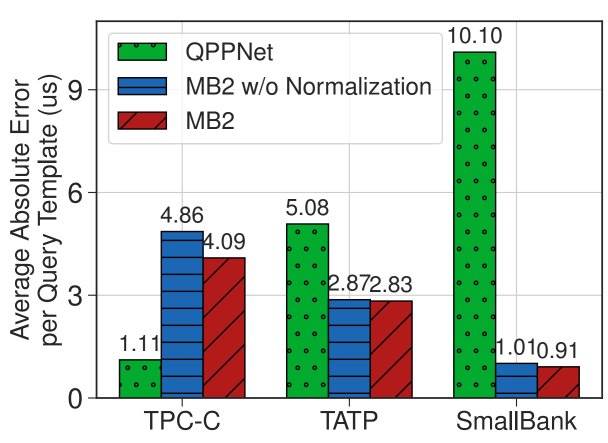
We next evaluate the accuracy of MB2's OU-models, which are the foundation of the self-driving DBMS behavior models. Recall from Section 4.5.4 that we split the data of each OU-runner into 80/20% train/test and build models with seven ML algorithms. From the test result, MB2 selects the best algorithm for each OU and trains the final OU-model using all available OU-runner data. Due to space limitations, we only demonstrate the results for a few more-representative and better-performing ML algorithms.

Figure 4.5 shows the OU-model test relative error averaged across all output labels. More than 80% of the OU-models have an average prediction error less than 20%, which demonstrates the effectiveness of the OU-models. The transaction OU models have higher relative error because most cases have short elapsed times (< 10μs) unless the system is under heavy contention. Similarly, probing an aggregation hash table takes less than 10μs in most cases enumerated by the OU-runner. Thus, for these short-running tasks, small perturbations in the predictions can lead to high relative error.

For most OUs, random forest and gradient boosting machine are the best-performing ML algorithms with sufficient generalizability. Neural networks have higher errors in comparison



(a) OLAP Query Runtime Prediction



(b) OLTP Query Runtime Prediction

Figure 4.7: OU-Model Generalization – Query runtime estimations on different datasets and workloads.

because they are prone to overfitting given the OU-models' low dimensionality. For simple OUs (e.g., arithmetics), less complex models like Huber regression achieve competitive accuracy and are cheaper to train.

In Figure 4.6, we show the predictive accuracy of the OU-models for each output label, averaging across all OUs. Most labels have an average error of less than 20%, where the highest error is on the cache miss label. Accurately estimating the cache misses for OUs is challenging because the metrics depend on the real-time contents of the CPU cache. Despite the higher cache miss error, MB2's interference model still captures the interference among concurrent OUs (see Section 4.7.4) because the interference model extracts information from all the output labels. The results in Figure 4.6 also show the OU-model errors without output label normalization optimization from Section 4.3.3. From this, we see that normalization has minimal impact on OU-model accuracy while enabling generalizability.

4.7.3 OU-Model Generalization

We now evaluate the OU-models' ability to predict query runtime and generalize across workloads. Accurate query runtime prediction is crucial since many self-driving DBMSs' optimization objectives are focused on reducing query latency [161]. As discussed in Section 4.2, MB2 extracts all OUs from a query plan and sums the predicted labels for all OUs as the final prediction.

For a state-of-the-art baseline, we compare against the **QPPNet** ML model for query performance prediction [106, 141]. QPPNet uses a tree-structured neural network to capture a query plan's structural information. It outperforms other recent models on predicting query runtime [23, 133, 231], especially when generalizing the trained model to different workloads (e.g., changing dataset sizes). Since NoisePage is an in-memory DBMS with a fused-operator JIT query engine [152], we remove any disk-oriented features from QPPNet's inputs and adapt its operator-level tree structure to support pipelines. But such adaptation requires QPPNet's training data to contain all the operator combinations in the test data pipelines to do inference with the proper tree structure. Thus, we can only train QPPNet on more complex workloads (e.g., TPC-C) and test on the same or simpler workloads (e.g., SmallBank).

We evaluate MB2 and QPPNet on the (1) TPC-H OLAP workload and (2) TPC-C, TATP, and SmallBank OLTP workloads. To evaluate generalizability, we first train a QPPNet model with query metrics from a 1 GB TPC-H dataset and evaluate it on two other dataset sizes (i.e., 0.1 GB, 10 GB). We then train another QPPNet model with data collected from the TPC-C workload (one warehouse) and evaluate it on the OLTP workloads. For MB2, we use the same OU-models only trained once (Section 4.7.2) for all the workloads.

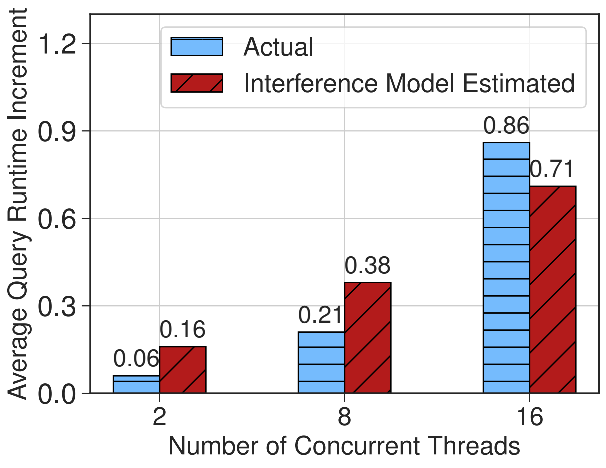
The results in Figure 4.7a show that QPPNet achieves competitive prediction accuracy on the 1 GB TPC-H workload since it is the same as its training dataset. But QPPNet has larger errors for TPC-H on other scale factors. The authors of QPPNet also observed similar generalization challenges for their models despite it outperforming the baselines [141]. In contrast, MB2 achieves up to 25× better accuracy than QPPNet and has more stable predictions across all the workload sizes. We attribute this difference to how (1) MB2 decomposes the DBMS into fine-grained OUs and the corresponding OU-runner enumerates various input features that cover a range of workloads, and (2) MB2's output label normalization technique further bolsters OU-models' generalizability. Even though the 10 GB TPC-H workload has tables up to 60m tuples, which is 60× larger than the largest table considered by MB2's OU-runners, MB2 is still able to generalize with minimal loss of accuracy. MB2 without the output normalization has much worse accuracy on large datasets.

Similarly, Figure 4.7b shows that while MB2 has 4× higher prediction error compared to QPPNet on TPC-C workload where QPPNet is trained, MB2 achieves 1.8× and 10× better accuracy when generalizing to TATP and SmallBank workloads. Such generalizability is essential for the real-world deployment of MB2 for self-driving DBMSs. Since these OLTP queries access a small number of tuples, output normalization has little impact on the model accuracy.

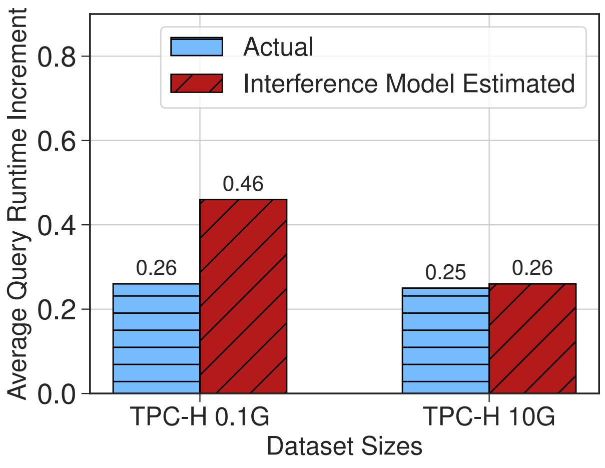
4.7.4 Interference Model Accuracy

We next measure the ability of MB2's interference model to capture the impact of resource competition on concurrent OUs. We run the concurrent runner with the 1 GB TPC-H benchmark since it contains a diverse set of OUs. The concurrent runner enumerates three parameters: (1) subsets of TPC-H queries, (2) number of concurrent threads, and (3) query arrival rate. Since the interference model is not specific to any OU or DBMS configuration, the concurrent runner does not need to exercise all OU-model inputs or knobs. For example, with the knob that controls the number of threads, we only generate training data for odd-numbered settings (i.e., 1, 3, 5, . . . 19) and then test on even-numbered settings. The concurrent runner executes each query setup for 20s. To reduce the evaluation time, we assume the average query arrival rate per query template per 10s is given to the model. In practice, this interval can be larger [137]. Neural network performs the best for this model given its capacity to consume the summary statistics of OU-model output labels.

To evaluate the model's generalizability, the concurrent runner executes queries only in the DBMS's interpretive mode (execution knob discussed in Section 4.3.2), but we test the model under JIT compilation mode. We also evaluate the model with thread numbers and workload



(a) Varying Concurrent Threads



(b) Varying Dataset Sizes

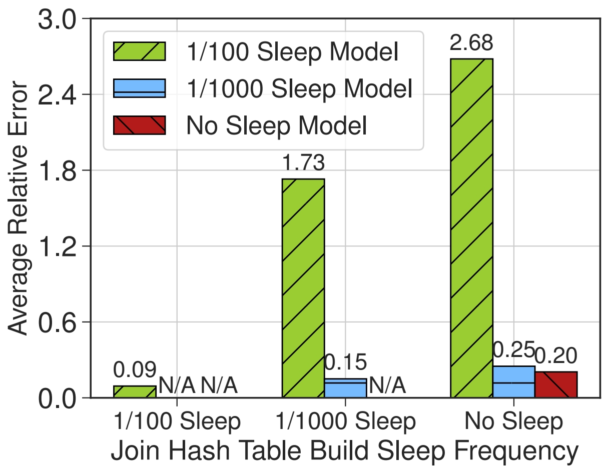
Figure 4.8: Interference Model Accuracy – Model trained with 1 GB TPC-H dataset and odd thread numbers generalizes to other scenarios.

sizes that are different from those used by MB2's concurrent runners. To isolate the interference model's estimation, we execute the queries in both single-thread and concurrent environments and compare the true adjustment factors (the concurrent interference impact) against the predicted adjustment factors.

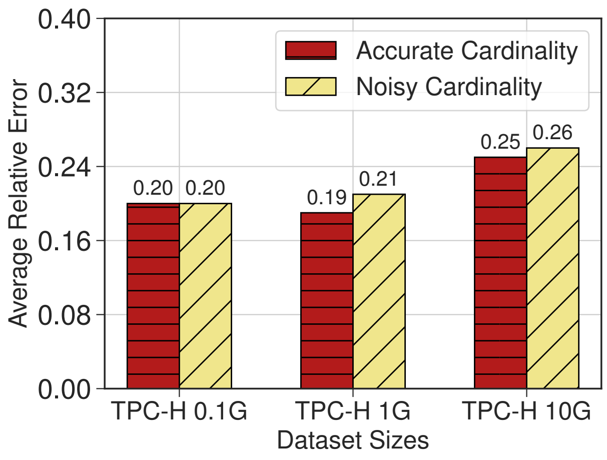
Figure 4.8 shows the actual and interference model-estimated average query run times under concurrent environments. The interference model has less than 20% error in all cases. It generalizes to these environments because (1) it leverages summary statistics of the OU-model output labels that are agnostic to specific OUs and (2) the elapsed time-based input normalization and ratio-based output labels help the model generalize across various scenarios. We also observe that generalizing the interference model to small data sizes result in the highest error (shown under TPC-H 0.1 GB in Figure 4.8b) since the queries are shorter with potentially higher variance in the interference, especially since the model only has the average query arrival information in an interval as an input feature.

4.7.5 Model Adaptation and Robustness

We now evaluate MB2's behavior models' adaptivity under DBMS software updates and robustness against DBMS estimation errors. For the former, we use a case study where we simulate a series of incremental improvements to the join hash table algorithm. To control the amount of change, we inject sleeps (1us) during the hash table creation with different frequencies: no sleep (fastest), sleep once with every 1000 inserted tuples, and sleep once with every 100 inserted tuples (slowest). This change does not affect the behavior of other OUs, such as sequential scans or sorts. Thus, without modifying any other OU-models, we only rerun MB2's OU-runner for hash join and retrain the corresponding OU-model, which takes less than 23 minutes. This is 24× faster than rerunning MB2's entire training process. Figure 4.9a shows the prediction errors for the TPC-H 1 GB workload with the old and updated models. The new models only



(a) DBMS Updates (TPC-H 1 G)



(b) Noisy Cardinality Estimation

Figure 4.9: Model Adaptation and Robustness – Changes in MB2's model accuracy under DBMS updates and noisy cardinality estimation.

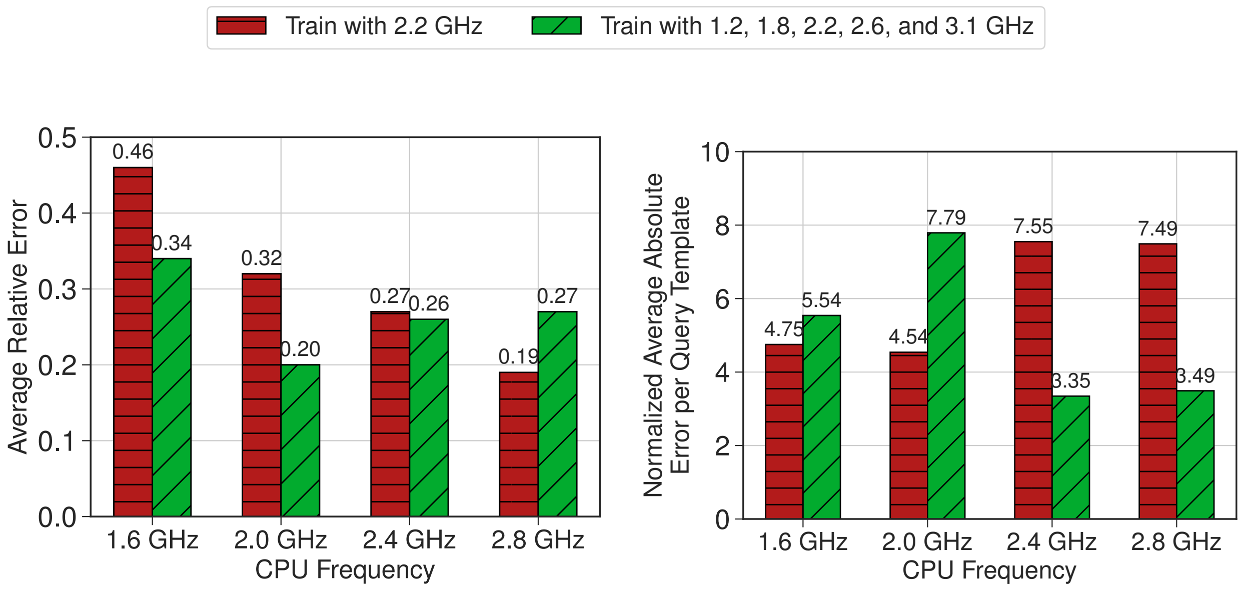
updated with the join-hash-table-build OU have significantly better accuracy compared to the old models prior to the system update. This demonstrates MB2's ability to quickly adapt its models in response to updates. QPPNet, on the other hand, needs full data remaking and model retraining.

Since the OU-models in NoisePage's execution engine use cardinality estimates as input features, we also evaluate the models' sensitivity to noisy estimations. We introduce a Gaussian white noise [69] with 0 mean and 30% variance of the actual value to the tuple number and cardinality features for OUs impacted by cardinality estimation (e.g., joins). Figure 4.9b shows the model's predictive accuracy under accurate and noisy cardinalities with different TPC-H dataset sizes. The models have minimal accuracy loss (< 2%) due to the noise because (1) they are insensitive to moderate noise in the features, and (2) there are many TPC-H queries with high selectivity that reduces the impact of wrong cardinality estimation when executing joins or sorts. Improving the model accuracy by leveraging learned cardinality estimation is left as future work [97, 235].

4.7.6 Hardware Context

Since MB2 generates models offline, their predictions may be inaccurate when the hardware used for training data generation and production differs. Thus, we evaluate MB2's ability to extend the OU-model features to include the hardware context and generalize the models across hardware. We also use a case study where we change the CPU frequency through its power governor. We append the frequency to the end of all the OU-models' input features. We compare the OU-models trained with either only the CPU's base frequency (2.2 GHz) or a range of frequencies (1.2–3.1 GHz), and test the model generalization on a different set of frequencies.

Figure 4.10 shows that extending the OU-model with hardware context improves the prediction in most cases since the context captures the hardware performance differences. A special



(a) TPC-H 1 G Query Runtime Prediction

(b) TPC-C Query Runtime Prediction

Figure 4.10: Hardware Context – Extend MB2's OU-models' input features to include the CPU frequency to generalize to different CPU frequencies.

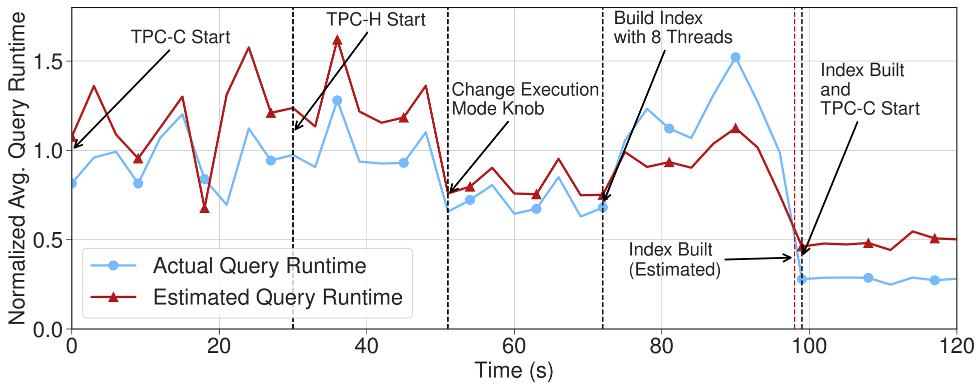
case where including the hardware context performs notably worse is for the TPC-C workload under 2.0 GHz CPU frequency (Figure 4.10b). This is because the models generally overpredict the TPC-C query runtime for a given frequency, and slightly slowing the CPU frequency (2.2 GHz to 2.0 GHz) falsely improves the prediction of the model trained under 2.2 GHz. The results show promise to extend MB2's models with hardware context to generalize across hardware.

4.7.7 End-to-End Self-Driving Execution

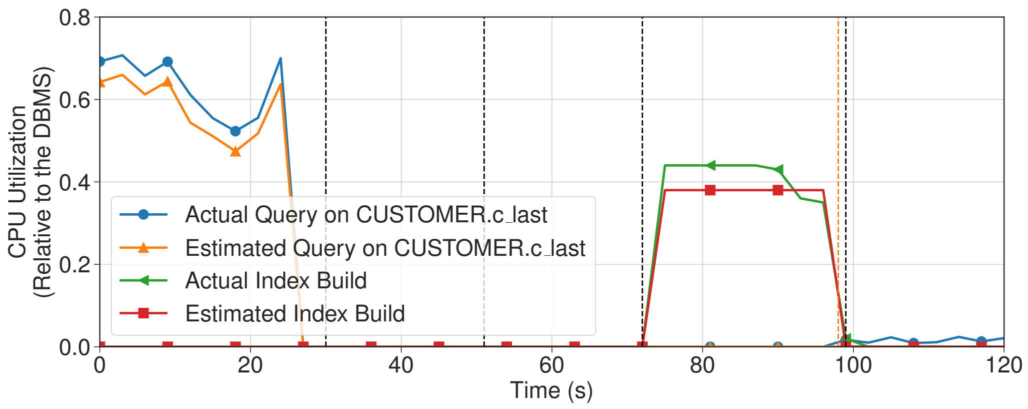
Lastly, we demonstrate MB2's behavior models' application for a self-driving DBMS's planning components and show how it enables interpretation of its decision-making process. We assume that the DBMS has (1) workload forecasting information of average query arrival rate per query template in each 10s forecasting interval, similar to Section 4.7.4, and (2) an "oracle" planner that makes decisions using predictions from behavior models. We assume a perfect workload forecast to isolate MB2's prediction error.

We simulate a daily transactional-analytical workload cycle by alternating the execution of TPC-C and TPC-H workloads on NoisePage. We use the TPC-C workload with 20 warehouses and adjust the number of customers per district to 50,000 to make the choice of indexes more important. For TPC-H, we use a dataset size of 1 GB. We execute both workloads with 10 concurrent threads under the maximum supported query arrival rate.

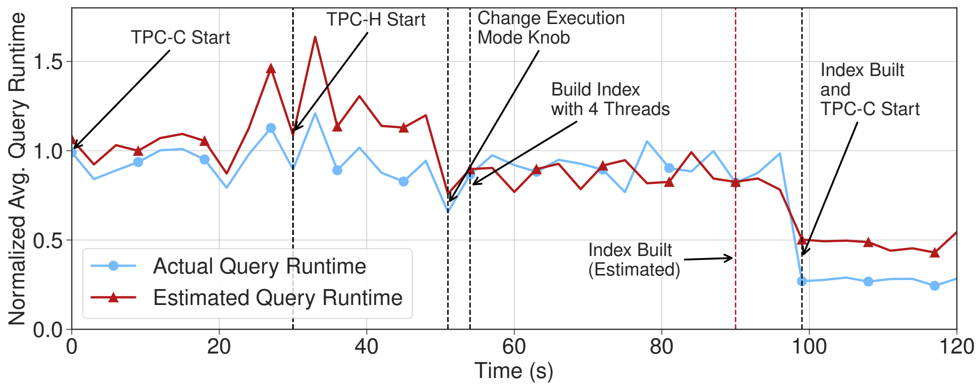
After the DBMS loads the dataset, we execute the workload in NoisePage using its interpretive mode, which is a sub-optimal knob configuration for long-running TPC-H queries. We also remove a secondary index on the `CUSTOMER` table for the (`C_W_ID`, `C_D_ID`, `C_LAST`) columns,



(a) Knob Changing and Index Creation (Eight Create Index Threads)



(b) CPU Utilization Prediction (Eight Create Index Threads)



(c) Knob Changing and Index Creation (Four Create Index Threads)

Figure 4.11: End-to-End Self-Driving Execution – An example scenario where NoisePage uses MB2's OU-models to predict the effects of two actions in a changing workload. The first action is to change the DBMS's execution mode. The DBMS then builds an index with either eight or four threads.

which inhibits the performance of TPC-C. We then let the DBMS choose actions based on the forecasted workload and the behavior model estimations from MB2.

Figure 4.11a shows the actual and estimated average query runtime per 3s interval across the entire evaluation session. We normalized the query runtime against the average runtime under the default configuration to keep metrics within the same scale. The workload starts as TPC-C and switches to TPC-H after 30s. During the first 55s, the DBMS does not plan any actions. Then it plans to change the execution mode from interpretive to compiled with MB2 estimating a 38% average query runtime reduction. The average query runtime drops by 30% after updating this knob, which reflects the models' estimation. After 72s, the DBMS builds the above secondary index on `CUSTOMER` with eight threads before the next time that the TPC-C workload starts. Both the estimated and the actual query runtime increase by more than 25% because of resource competition; and the contention lasts for 27s during the index build with the estimated build time being 26s. After 99s the workload switches back to TPC-C with 73% (60% estimated) faster average query runtime because of the secondary index built by the self-driving DBMS [3]. This example demonstrates that MB2's behavior models accurately estimate the cost, impact, and benefit of actions for self-driving DBMSs ahead of time given the workload forecasting, which is a foundational step towards building a self-driving DBMS [160].

We next show how MB2's behavior models help explain the self-driving DBMS's decision. Figure 4.11b shows the actual and estimated CPU utilization for the queries associated with the secondary index on the `CUSTOMER` and the index build action. Both the actual and estimated CPU utilization for the queries on `CUSTOMER` drop significantly as the TPC-C workload starts for the second time, which is the main reason for the average query runtime reduction. Similarly, the high CPU utilization of index creation, which MB2 successfully predicts, is also the main reason for the query runtime increment between 72s to 99s. MB2's decomposed modeling of each OU is the crucial feature that enables such explainability.

Lastly, we demonstrate MB2's estimations under an alternative action plan of the self-driving DBMS in Figure 4.11c. The DBMS plans the same actions under the same setup as in Figure 4.11a except to build the index earlier (at 58s) with four threads to reduce the impact on the running workload. MB2 accurately estimates the smaller query runtime increment along with the workload being impacted for longer. Such a trade-off between action time and workload impact is essential information for a self-driving DBMS to plan for target objectives (e.g., SLAs). We also observe that MB2 underestimates the index build time under this scenario by 27% due to a combination of OU- and interference-model errors.

[3]The DBMS does not change the execution mode for TPC-C to isolate the action effect.

Chapter 5

Action Planning

We presented techniques to predict the future workload and estimate the behavior of self-driving actions. A self-driving DBMS still needs to decide when to apply what actions given those predictions. Recall that the ability to automatically choose and apply actions without any human intervention distincts self-driving DBMSs from DBMSs with lower autonomy levels (Section 2.2). Proper action planning is essential because actions may (1) take a long time to finish, (2) consume a significant amount of resources, and (3) contend with client queries and slow them down. It is also important to plan a sequence of actions ahead of time because of the potential interactions between actions. For instance, creating one index can make creating another index with overlapping columns unnecessary. Furthermore, as discussed in Chapter 3, database workloads often have cyclical patterns. For example, many workloads have a diurnal pattern that follows human living-cycle [137, 205]. Thus, it is often beneficial to apply expensive actions when the workload volume is low to avoid resource contention.

It is challenging to plan effective actions for self-driving DBMSs efficiently. The sequence of actions may span short and long horizons. Finding the best sequence among various potential actions (e.g., changing knobs or creating indexes) over these horizons is an expensive constrained optimization problem with an exponential search space. The action planning also relies on the behavior models (Chapter 4) to estimate the actions' impact. The model inferences may incur high overhead, which poses more challenges on the planning efficacy.

Most of the previous work on automated database tuning has focused on static workloads (e.g., knob tuning [132, 215, 244], physical design [22, 58, 118], or both [220]). These methods select the best set of knobs or physical design given a static representative workload, typically provided by DBAs. The DBAs then manually decide which action to apply at what time. Some methods focus on database tuning under workload shifts [37, 163, 185]. But most of these works are reactionary: they address the problems after the workload has shifted but do not plan actions proactively based on future workload patterns. To the best of our knowledge, the most recent work that generates a sequence of database tuning actions given the workload patterns is GREEDY-SEQ [20]. This is a heuristic algorithm that recursively merges small action sequences. Though it generates action sequences efficiently, it may miss many optimization opportunities. Moreover, GREEDY-SEQ only supports tuning physical design but not knobs.

We propose an action planning framework for self-driving DBMSs, called **PilotBot0** (PB0), leveraging control theory and recent advances in artificial intelligence (AI). PB0 uses the receding horizon control scheme [145] to plan action sequence across both short and long horizons. It also adapts the Monte Carlo tree search (MCTS) method [51] that has recent successes in AI systems [192, 193], to efficiently approximate the optimization problem. MCTS enables PB0 to effectively select a promising action sequence given any specified time budget. Furthermore, PB0 uses a two-level caching design to accelerate the model inference at both action state and model inference level, which reduces the planning cost.

We first introduce the background information in Section 5.1. We then give an overview of PB0 in Section 5.2. We discuss PB0's MCTS technique and our optimizations in Section 5.3. We explain PB0's caching techniques and implementation choices in Section 5.4 and Section 5.5. And we present our evaluation results in Section 5.6.

5.1 Background

We first discuss the related background information to help understand PB0's mechanism.

5.1.1 Static Workload Tuning

There are many works that help select the best set of changes (e.g., knobs or physical design) for a database given a static workload. For example, many works aim to find optimized index configurations for a workload [46, 58, 59, 183, 214]. Others try to find the best set of knob configurations [132, 215, 244]. These are degenerate cases of self-driving DBMSs' action planning, i.e., the workload only has one uniform pattern. PB0 builds on top of these methods. Instead of searching over the exponential number of all possible changes (actions) for the database, PB0 uses previous methods to generate a small set of candidate actions for the entire set of the forecasted workload. It then searches for the best action sequence from the candidate actions considering the workload patterns.

5.1.2 Receding Horizon Control

Receding horizon control (RHC) is a general-purpose control scheme to solve a constrained optimization problem over a moving time horizon [145]. It uses predictions of future costs/benefits to choose appropriate actions. It has successful applications in various practical scenarios, such as industrial and chemical process control [166], supply chain management [49], and stochastic control in economics [96]. The main advantage of RHC is that it optimizes the current time interval while keeping future times intervals into account. It achieves this by repeatedly optimizing over a finite time horizon but only implementing the action for the current time interval.

5.1.3 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a randomized search algorithm used for decision processes that are difficult or impossible to solve using other approaches [51]. MCTS has been successfully applied to many strategy design problems in computer science, such as Go game [192], chess [193], and poker [178]. MCTS focuses on the analysis of the most promising actions and expands the search tree based on random samplings of the search space. It returns the best sequences of actions explored until the time budget is exhausted. And the quality of the selected actions often increases with time as the algorithm explores more promising actions [43].

MCTS, however, is not a silver bullet. We discuss the additional challenges in applying MCST for self-driving databases that we identify and address in Section 5.3.

5.2 Overview

The DBA must specify the objective (i.e., SLA) before PB0 plans actions, similar to how passengers in a self-driving vehicle need to specify their destinations. For example, the objective can be minimizing the query latency within a storage/memory constraint. PB0 uses the forecasted workload and behavior models to evaluate the effect of self-driving actions relative to the objective and choose the best actions.

There are certain responsibilities that PB0 must fulfill to enable a completely autonomous system:

- Plan actions for both the current and future (forecasted) workloads to address problems before they occur.
- Ensure the satisfaction of all system constraints (e.g., maximum memory consumption).
- Decide when to apply the actions, how to apply them, and apply them automatically without human intervention.
- Provide explanations for the past and future planned actions, which are important for examination, debugging, and auditing.

We now give an overview of how PB0 achieves these goals.

As illustrated in Figure 5.1, PB0 executes the planning on a pilot server that is seperate from the DBMS server. PB0 also uses a model server that maintains the workload forecast and behavior estimation models. Conceptually, the pilot server and the model server can run on any machine. We expect users to deploy the pilot server and model server on a separate machine from the DBMS to avoid resource contention. Furthermore, the model server may run a on machine with hardware accelerators (e.g., GPUs) to further boost the planning speed.

PB0 continuously repeats a planning process with following steps:

- ❶ At the beginning of each time interval, PB0 requests the DBMS to send the query trace (Chapter 3) to the model server to predict the future workload.
- ❷ After the pilot server receives the forecasted workload from the model server, it first uses existing methods (Section 5.1.1) to generate candidate actions for the entire set of the forecasted workload without considering the workload patterns. For example, it uses an algorithm inspired by AutoAdmin from Microsoft [46, 58] to generate the set of candidate indexes that may improve the DBMS's performance on the workload. It uses simple heuristics to generate candidate actions that change the DBMS knobs (e.g., the log serialization interval) with values of different ranges (e.g., increment by 10 or 100).
- ❸ PB0 then plans an action sequence out of the candidates based on the predicted workload patterns. It uses RHC to optimize the objective over a fixed planning horizon with

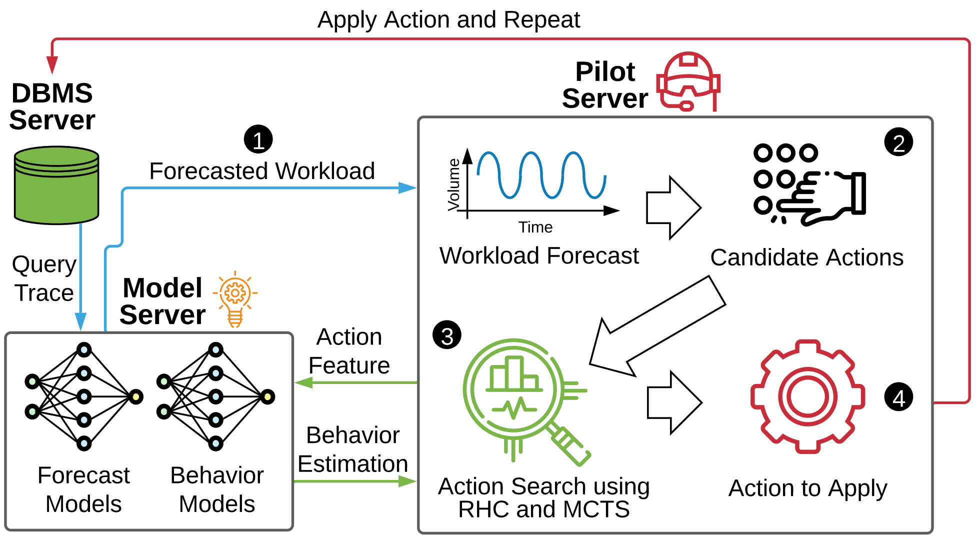


Figure 5.1: Pilot Architecture - PB0 runs the pilot server for planning on a separate machine from the DBMS server. It receives the forecasted workload from a model server to generate candidate self-driving actions. It then uses RHC and MCTS to search for the best action sequence based on the behavior estimations. It applies the first action and repeats the above process.

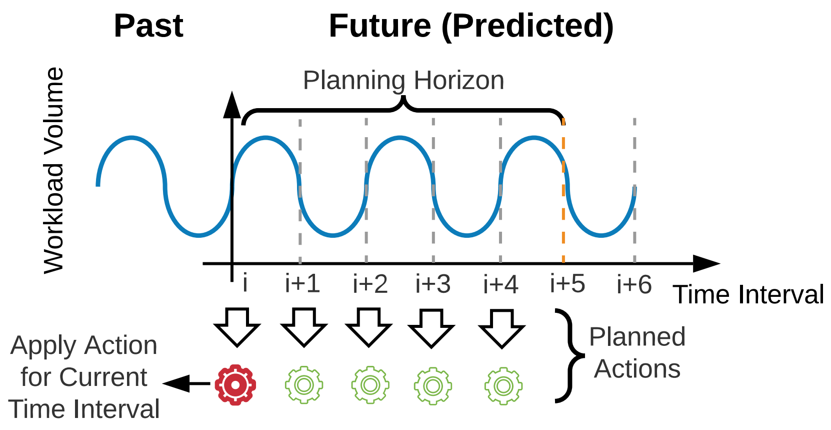


Figure 5.2: Receding Horizon Control - PB0 repeatedly plans actions for a fixed number of time intervals in the future and applies the first action for the current interval.

multiple time intervals. PB0 uses MCTS with the behavior model estimations to solve RHC's optimization problem (details in Section 5.3).

- ❹ Lastly, PB0 requests the DBMS to apply the first action planned for the current time interval and discards the remaining actions. At the beginning of the next time interval, PB0 repeats this optimization process and generates another series of actions optimized for the subsequent forecasted workload.

Figure 5.2 illustrates an example of the RHC process in PB0. The planning horizon has five time intervals. PB0 plans a sequence of five actions with one action for each interval based on the forecasted workload. [1] It then applies the first action for the current time interval and repeats the planning process as the time advances to the next interval.

The behavior models (Chapter 4) serve as the predictive models that the RHC process uses to estimate the action's costs and benefits relative to the self-driving objective. For example, PB0 may need to evaluate the effectiveness of an index creation action based on the objective of reducing the average query latency. In this case, the behavior models estimate how much the forecasted queries' latency increases during the index creation because of the resource competition and how much the forecasted queries' latency decreases after the index creation. PB0 then computes the average query latency by adding these estimations as RHC's cost function. The behavior models also estimate whether the index creation violates any constraints, such as consuming more memory than the system provides. Because PB0 requires these estimations, the pilot server needs to frequently request the model server to perform inferences on the behavior models, which can seriously reduce the planning efficiency. PB0 uses a two-level caching design to accelerate this inference process (Section 5.4).

PB0 also records the forecasted workload and the planned/applied actions in internal tables to serve as an explanation of its decisions. The record for each action contains the action identifier (a monotonically increasing integer), the action command, and the cost/benefit estimation for that action from the behavior models, such as the action's application time, resource consumption, and impact on the self-driving objective. Users or developers can query these tables through the DBMS's SQL interface for debugging, examining, or auditing purposes.

Limitation: We now discuss the assumptions and limitations in PB0. Foremost is that PB0's current implementation only supports reducing the average query latency as the self-driving objective under a memory constraint specified by the user. Nevertheless, we design PB0's architecture and search mechanism to be independent of the self-driving objective. We anticipate that extending PB0 to support other objectives, such as reducing the 95th percentile latency, only involves changing the formula to calculate the initial value number of the MCTS tree node (Section 5.3.1).

We implement PB0 in an in-memory DBMS, NoisePage. Thus, PB0 does not need to support disk-based storage constraints. We anticipate that supporting storage constraints during PB0's planning process is similar to supporting the memory constraint (Section 5.3.4). However, the behavior models need to extend to support estimating the storage overhead for any self-driving actions [139]. NoisePage, as a research prototype, does not support complex database design features (e.g., views, table partitioning) or resource scaling yet. Thus, PB0 focuses on creating/dropping indexes and changing knob values as self-driving actions.

Lastly, PB0 selects the best action sequence from a set of given candidate actions based on the workload patterns but does not generate the candidate actions. As discussed earlier, there are many existing works that PB0 may leverage to further increase the candidate actions' quality. For example, PB0 uses simple heuristics to generate candidate change-knob actions since NoisePage only exposes a small number of configuration knobs. As NoisePage extends its functionalities and uses more knobs, PB0 may use existing methods [215] to pre-select more

[1]For simplicity we assume each action takes at most one interval to finish. We discuss how PB0 handles actions that take multiple intervals in Section 5.3.3.

important knobs to reduce the number of candidate actions and facilitate the planning process.

5.3 Monte Carlo Tree Search and Optimization

The core challenge in PB0's RHC process is to select a sequence of actions that optimize the self-driving objective given a fixed horizon of forecasted workload. The optimization also needs to satisfy any system constraint, such as the maximum memory consumption. We now formally define the optimization problem that PB0 considers.

We denote the total number of time intervals in the planning horizon as n. Recall that the workload forecast predicts the workload w_t for each time interval $t \in \{1, 2, ..., n\}$. And w_t contains the number of arrivals and parameter samples for each query template in that time interval. The set of candidate actions is A. The self-driving objective (e.g., the average query latency) for workload w_t under the changes for a set of actions $A' \subseteq A$ is $g(w_t, A')$. The maximum amount of allowed memory consumption is M and the system memory consumption at time interval t after applying a set of actions $A'' \subseteq A$ is $m(t, A'')$.

Users specify M, and PB0 derives $g(w_t, A')$ based on the user-specified objective using the behavior model estimations. For example, if the objective is to reduce the average query latency, then PB0 uses behavior models to predict the average query latency in w_t after applying A'. The optimization goal is to find a series of d actions $a_1, a_2, ..., a_d$, where $a_i \in A$ and $d \leq n$, and their corresponding application times $t_{a_1}, t_{a_2}, ..., t_{a_d}$, to minimize the total objective over all horizons:

$$\arg\min_{a_1,...,a_d,t_1,...,t_d} \sum_{i \in \{1,...,n\}} g(w_i, \{a_j | t_{a_j} \leq i, 1 \leq j \leq d\})$$

subject to

$$\forall i \in \{1, ..., n\}, m(i, \{a_j | t_{a_j} \leq i, 1 \leq j \leq d\}) \leq M$$

Some actions (e.g., creating an index) may take a long time to finish and cannot improve the workload immediately. We do not include this in the formal problem definition for simplicity. We discuss this in Section 5.3.3.

PB0's constrained optimization problem is expensive to solve as it has an exponential search space of $(n + 1)^{|A|}$. Given this complexity, previous work has resolved to heuristics that can generate an action sequence in a reasonable time but may miss many optimization opportunities [20]. PB0 uses a principled framework (MCTS) to effectively search for promising action sequences given a time budget.

MCTS finds action sequences using a search tree. Each tree level represents a potential action at a specific time interval. Thus, a path from the root to any node of the search tree represents a sequence of d actions at corresponding time intervals, where d equals the node's depth. MCTS maintains a *value number* for each tree node. A node's value number can change during the seach based on how its decendents expand. Therefore, MCTS cannot pre-compute a value number for an action and reuse it across the entire search. MCTS keeps repeating the following search process:

(1) Start from the root to recursively select child with the value number as the sampling weight until reaching a leaf.

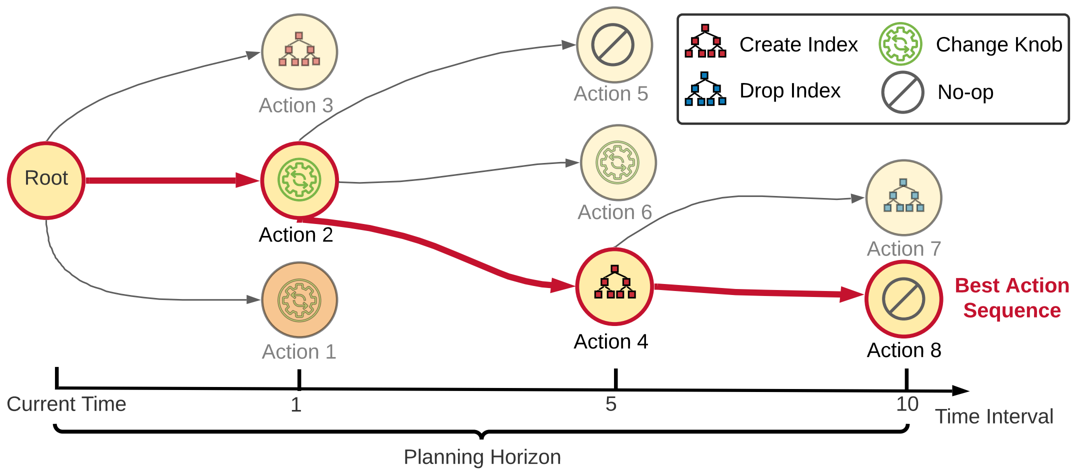


Figure 5.3: Monte Carlo Tree Search - MCTS randomly searches over more promising action sequences under a time budget and returns the best action sequence explored.

(2) If the selected leaf node does not reach the planning horizon, expand the leaf node with one or more children.

(3) Select a child and populate its initial value number (also called *rollout*).

(4) Update the value numbers for the nodes from the root to selected leaf based on the child's rollout result.

The essence of MCTS is a *search strategy* that determines the value numbers, which balance the exploration and exploitation (Section 5.3.2). After the search finishes, PB0 chooses the action sequences represented by nodes with the highest value numbers as its decision. There is a trade-off between the search time budget the planning quality. Such an explicit search process also provides explanations for the cost-benefit estimation of each action in each time interval.

Figure 5.3 illustrates an example of the MCTS result. There are four types of actions in this example, including a no-op action that skips all actions at a time interval. The MCTS explores six action sequences (six leaf nodes) over a planning horizon of ten time intervals. There may not be actions for every time interval, and action sequences may not reach the end of the planning horizon, either. After MCTS exhausts the time budget, it returns the best action sequence explored under the self-driving objective (Action 2→4→8).

There are additional challenges, however, that that we identify in applying MCTS for PB0. We now discuss these challenges and our solutions.

5.3.1 Rollout Challenge

The first challenge to implement MCTS for PB0 is to determine the initial value number for a new leaf node (child of the original leaf) to perform the search. Most existing MCTS methods focus on win/loss games that have a binary outcome, and the canonical method to derive such value is to perform a random rollout: play moves (i.e., apply actions) randomly starting at the new leaf node until a win/loss outcome is determined [114, 192, 193]. This does not directly apply to PB0 since there is no win/loss for a database workload.

Given this, PB0 adapts such canonical rollout method by directly using the objective function g over the entire workload as the initial value number for a newly expanded leaf node l, given the actions from the root to l applied. We denote this rollout value for l as G_l. Furthermore, we observe that since the rollout does not need to determine win/loss, it does not need to apply actions randomly after l, either. This also avoids large oscillations in G_l when the rollout randomly selects a poor action, which may make the search miss promising action sequences. For example, if the rollout randomly drops an important secondary index after l, the average query latency can become drastically higher even though the actions from the root to l have a low G_l. In this case, MCTS is less likely to expand further from l even though actions from the root to l may already benefit the workload without any more actions. Lastly, PB0's MCTS performs rollout for all the new leaf nodes during an expansion together to give the search strategy a better action quality overview.

5.3.2 Search Strategy Challenge

One of MCTS's key challenges is to balance exploration (exploring more action sequences) and exploitation (extending better-performing action sequences). MCTS addresses such challenge using a search strategy that determines each node's value number (i.e., the sampling weight used to select the leaf node to expand). As mentioned in Section 5.3.1, most existing MCTS algorithms focus on win/loss games with a binary outcome. A well-known strategy that most contemporary MCTS search strategies employ is called UCT [114], which calculates the value number for a node o, which can be either an internal node or a leaf node, as:

$$\frac{w_o}{v_o} + \sqrt{\frac{2 \ln V_o}{v_o}}$$

where w_o is the the number of wins for the nodes considered after o, v_o is the number of the times o has been visited, and V_o is the total number of times that o's parent has been visited. The formula's first component corresponds to exploitation that favors moves with a high average win ratio. The second component corresponds to exploration that favors moves with few visits.

Given PB0's rollout method discussed in Section 5.3.1, a natural adaptaion of UCT is:

$$\frac{\text{Average}_{l \in L_{o\text{'s parent}}} G_l}{\text{Average}_{l \in L_o} G_l} + \sqrt{\frac{2 \ln V_o}{v_o}}$$

where L_o is the set of leaves after node o. The formula's first component immitates $\frac{w_o}{v_o}$, which favors nodes with better (lower) average self-driving objective among its peers from the parent.

However, one challenge we observe with this formula is that a single leaf node with an action that negatively affects the workload's queries can drastically change the average objective of all its ancestors. Similar to the rollout challenge, a leaf node that drops an important secondary index can significantly increase the value function of its ancestors. This may lead the search to miss promising action sequences. To address such challenge, PB0 applies a simple yet effective modification to the formula:

$$\frac{\text{Min}_{c \in L_{o\text{'s parent}}} G_c}{\text{Min}_{c \in L_o} G_c} + \sqrt{\frac{2 \ln V_o}{v_o}}$$

Intuitively, this formula uses the best leaf after a node o to represent o's value, which is robust against outlier leaf nodes that may considerably increase the average self-driving objective. And it favors nodes that have a best leaf with low objective value. The first component's range is also between $[0, 1]$, which is the same as the range of the original UCT formula's first component (the win ratio $\frac{w_o}{v_o}$) . We observe that this strategy is more stable and generates better actions under the same time budget than the first adaptation.

5.3.3 Search Interval Challenge

The workload forecasts determine the finest granularity of the planning time interval (e.g., one minute). A straightforward MCTS implementaion for PB0 is to search one action for each time interval during the entire planning horizon (e.g., one day), which can make n large. Thus, the search tree depth can be deep, which exponentially increases the number of possible states. This is challenging for MCTS to generate high-quality actions.

PB0 addresses this challenge by dynamically increasing the number of intervals between actions (search tree levels) during MCTS. The intuition is that actions planned for intervals in the near future may be more important and require finer-grained control since the self-driving DBMS will apply them at first. In contrast, actions for intervals in the far future may be less important since PB0 will re-plan for them later. Thus, PB0 starts its MCTS with a small number of time intervals between actions for fine-grained action optimization. It then increases the number of intervals for each level of the search tree to reduce MCTS's depth and complexity.

In our implementation, PB0 exponentially increases the number of time intervals in every level of MCTS' search (1, 2, 4, 8, 16...). Self-driving DBMS developers may use their domain knowledge to better specify these intervals. For example, many database applications have a diurnal workload pattern that repeats similar workloads in the duration of a day [137, 205]. Thus, developers can set a fixed number of time intervals for the search tree levels corresponding to the first day, so that MCTS can perform fine-grained planning for a full workload cycle. The number of time intervals for the levels after the first day can then increase to reduce the search complexity.

Another difference between PB0's planning and canonical MCTS is that some self-driving database actions (e.g., creating an index) may take a long time to finish. Thus, one action may not finish within the number of time intervals specified for the corresponding search tree level. In this case, PB0 extends that level's number of intervals to the action applying time. Recall that PB0 uses the behavior models (Chapter 4) to estimate the action applying time and the action's interference to the DBMS performance during its application. Only after the action finishes can it help improve the system performance. Thus, PB0 needs to predict the system performance for the workloads before, in-between, and after the action application separately to calculate the objective function g (used in the rollout as discussed in Section 5.3.1).

5.3.4 Memory Constraint Challenge

PB0's MCTS needs to ensure that the explored action sequences do not violate the memory constraint specified by the user. MCTSs designed for games do not support this. PB0 supports ensuring such memory constraint during MCTS by extrapolating the system memory consumption based on the workload forecasts and behavior model estimations.

Recall that the workload forecasts have the query (template) arrival prediction in each time interval in the planning horizon. Thus, PB0 can approximate the table size changes at a given time interval by estimating how many tuples that the workload inserts into/deletes from each table. And it can access the initial system memory consumption for all the tables and indexes at the beginning of MCTS (recall that NoisePage stores tables and indexes in memory as an in-memory DBMS). Thus, PB0 linearly extrapolates the system memory consumption based on the table size changes at any time interval during MCTS. It also uses behavior models to estimate the action memory consumption and extrapolate the value similarly at a given time interval.

If the estimated memory consumption at the time interval of a newly expanded leaf l exceeds the constraint, PB0's MCTS sets l's value number to infinity to avoid future selection of l. It also needs to recalculate the value numbers of all l's ancestors. This is because l's parent may have a high value number before the expansion, but all the new leaves (l's peers) may exceed the memory constraint because of the workload's tuple inserts. In this case, l's parent's original value number becomes invalid, which also impacts all ancestors.

5.4 Inference Cache

MCTS in PB0 makes many inference requests to the behavior models to estimate the action impacts. After a leaf expansion, it estimates the objective function g for both the workloads under the time intervals represented by the current leaf level and the remaining time intervals until the end of the planning horizon. PB0 uses the operating unit (OU) translator (Chapter 4) to generate OUs and corresponding features for these workloads and actions. Then it sends a request to the model server to make inferences on these features, retrieve the estimated values, and sums them up as the inference result. Each query or action may generate multiple OUs and thus require multiple inferences. And every inference takes $\sim$0.5ms based on our experiments. Thus, these inferences add up as significant overhead to the planning.

We observe that there can be repeated inference requests during MCTS, such as estimating the impact of the same action at the same search tree level or invoking the same model from the model server with the same input feature. Thus, PB0 uses a two-level cache to reuse inference results and accelerate the planning.

Action State Cache: The higher-level cache stores the value of the objective function g over a range of time intervals given a set of applied actions. For example, if two leaves at the same level of the search tree have the same set of applied actions along their paths from the root, they share the same value of the objective function g over the remaining time intervals of the workload. Note that the two action sequences (paths) only need to be logically equivalent to reuse such value. For example, if one leaf has actions `CREATE INDEX A`, `CREATE INDEX B`, and `DROP INDEX A` applied, and the other leaf has the action `CREATE INDEX B` applied at the same level, they still share the same value of g for the remaining intervals.

Model Inference Cache: The lower-level cache stores the value of an OU or interference model inference result. For example, suppose PB0 needs to perform a model inference on the

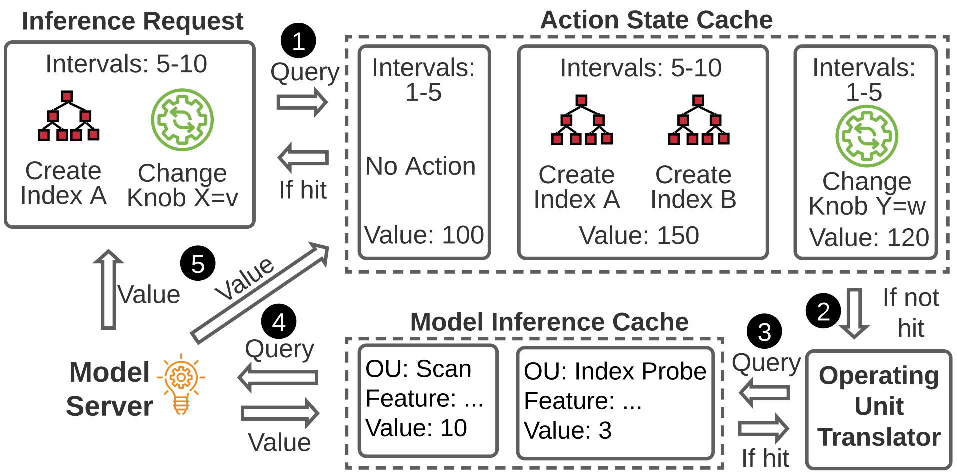


Figure 5.4: Inference Cache - PB0 caches the inference results to accelerate the planning at two levels: the action state level and the model inference level.

same OU with the same feature as a previous inference request. In that case, it can reuse the cached value to avoid the inference call to the model server.

Figure 5.4 illustrates PB0's inference request procedure under the two-level caching:

- ❶ When MCTS expands a new leaf, PB0 first asks the action state cache whether the search has calculated the objective function g for the same range of time interval with the same set of actions. If so, it directly returns the cached value for the leaf.
- ❷ If the request does not hit the action state cache, PB0 uses the OU translator to generate the OUs and features for the workload and action.
- ❸ PB0 then checks the model inference cache for the same inferences performed before. If all the inference requests are in the model inference cache, PB0 calculates the objective function g for the leaf based on the cached values and populates the action state cache.
- ❹ If there are still remaining uncached inference requests, PB0 queries the model server to obtain the estimated values.
- ❺ The model server returns the inference results for PB0 to calculate the leaf's objective function g. PB0 then populates the unfulfilled model inference and action state cache.

Each entry in the action state cache or the model inference cache is smaller than 1 KB. Recall from Section 5.2 that RHC starts one MCTS for each of the repeated planning process. PB0 keeps all the entries in the two caches during a planning process's MCTS. When MCTS finishes, PB0 releases the entire cache and re-create the caches when RHC starts MCTS in the next planning process.

5.5 Implementation Choices

We now describe PB0's several implementation choices.

The first is that the pilot server is a special "replica" node of the database server. It replicates the database catalog and statistics (from `ANALYZE`). Thus, PB0 on the pilot server can generate candidate actions and translate the forecasted workload and actions into OUs with proper features. PB0 uses these features with the behavior models to estimate the action impact. The pilot server does not need to replicate the table contents, which reduces the synchronization overhead.

The second notable aspect of the system is that PB0 uses "what-if" API [44] to help evaluate the action impact. When MCTS expands a new leaf, PB0 applies actions from the root to the leaf under the "what-if" API. It then uses the optimizer to re-generate query plans for the workload's queries under these actions. Such plan re-generation is necessary because (1) the behavior models rely on the query plan to generate OUs and corresponding features to perform inference and (2) there are actions (e.g., creating indexes) that are only useful when the optimizer chooses to use them in the query plan. After one search expansion of MCTS, PB0 restores the applied actions to prepare for the next search.

Lastly, PB0 implements what-if API with NoisePage's transactional catalog. Before MCTS starts, PB0 opens a transaction for each database and applies all the actions within the scope of these transactions. For example, if an action is to create an index, PB0 only inserts the index entry into the catalog without populating the index. The optimizer can decide whether the new query plan uses this index. When MCTS finishes, PB0 aborts all the transactions to revert any temporary changes. Thus, the pilot server isolates the action search against other queries from users or developers. For example, developers can query the pilot server to access the workload forecasts or action history information without interfering with PB0's search process.

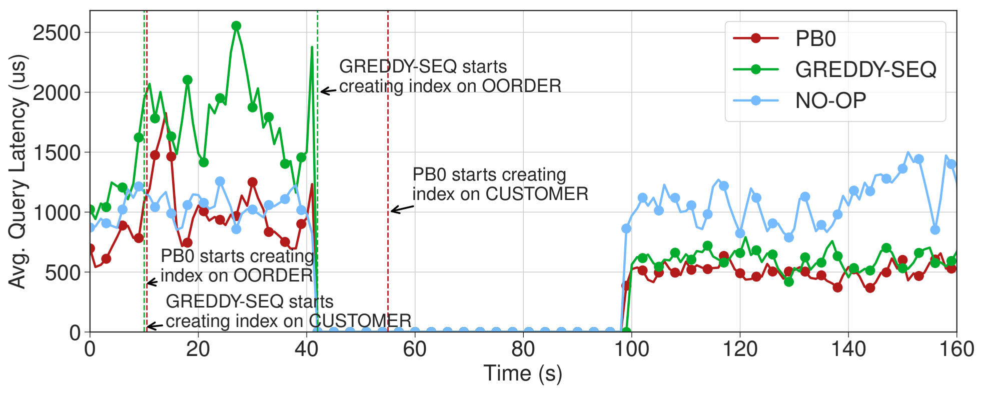
5.6 Evaluation

We now discuss our evaluation of PB0 using the NoisePage DBMS. We deployed the DBMS on a Ubuntu 20.04 LTS machine with two 10-core Intel Xeon E5-2630v4 CPUs, 128GB of DRAM, and Intel Optane DC P4800X SSD.

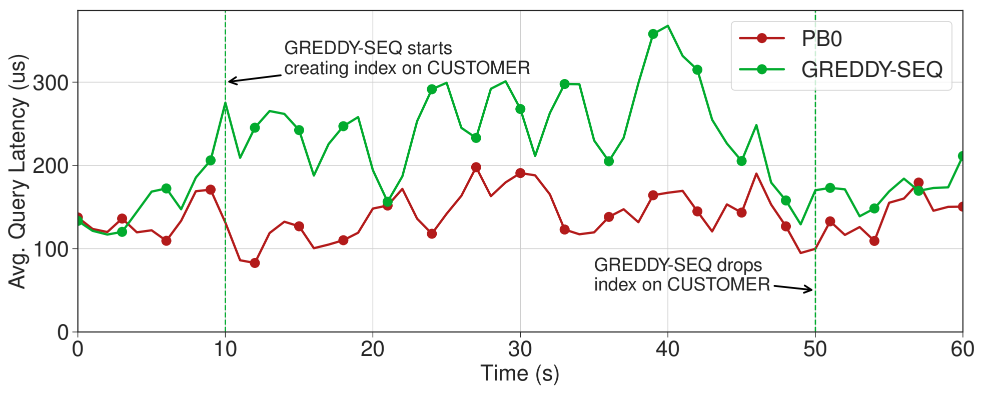
We use the OLTP-Bench [64] testbed as an end-to-end workload generator. We use the TPC-C [208] benchmark that has nine tables and five transaction profiles that model backend warehouses fulfilling orders for a merchant. We use `numactl` to fix the DBMS on one CPU socket and the OLTP-Bench clients and PB0 processes on the other CPU socket. We also use QB5000 to generate the workload forecasts and MB2 to generate the behavior models.

We compare PB0 against a state-of-the-art method that generates a sequence of DBMS tuning actions based on workload patterns, called **GREEDY-SEQ** [20]. Instead of planning for all actions together, GREEDY-SEQ uses a heuristic to reduce the planning complexity by first choosing the best timings to apply each action separately and then merging the actions into a sequence. This method only applies to physical design (e.g., indexes) but not knobs.

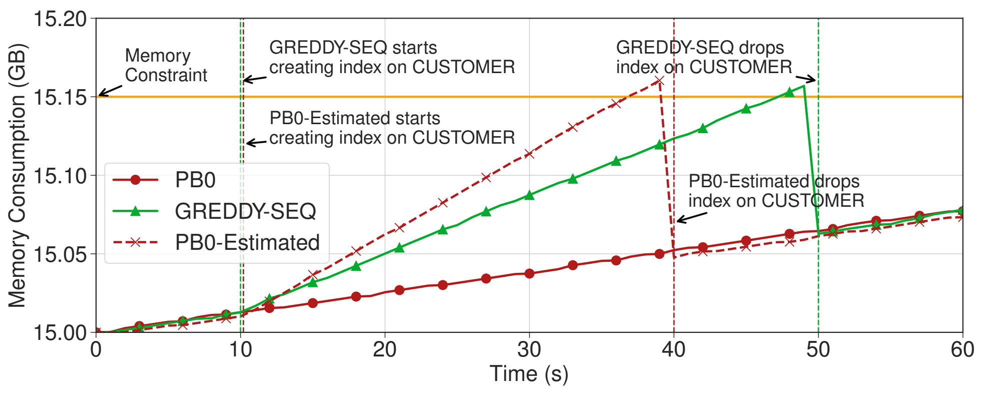
We evaluate PB0's planning quality and the effectiveness of PB0's MCTS optimization and caching design. We use the workload's average query latency as the evaluation metric, which is the same as the self-driving objective.



(a) Diurnal Pattern



(b) Growth Pattern



(c) Growth Pattern Memory Consumption

Figure 5.5: Planning Quality – System performance with the actions chosen by PB0 and GREEDY-SEQ under different workload patterns.

5.6.1 Planning Quality

As discussed in Chapter 3, database workloads often have patterns and choosing action sequences tailored to these patterns is crucial to the system's performance. Thus, we first evaluate PB0's planning quality under two common database workload patterns. We simulate the patterns in short durations to accelerate the evaluation. We use QB5000 to generate workload forecasting information of average query arrival rate per query template in each 5s forecasting interval. Therefore, the smallest planning time interval is 5s. We use a two-minute planning horizon. As discussed in Section 5.3.3, we set PB0 to exponentially increase the number of intervals between MCTS's levels. We set MCTS to execute 80 searches for each workload.

We use the TPC-C workload with 10 warehouses and adjust the number of customers per district to 100,000 to make the choice of indexes more important. We remove the secondary indexes for TPC-C before the experiments. We extend an algorithm inspired by AutoAdmin [46, 58] to generate candidate actions that create secondary indexes for TPC-C with different numbers of threads (1, 2, 4, or 8). We also use heuristics to generate candidate actions that change NoisePage's knobs.

We now discuss PB0's performance for each workload pattern.

Diurnal Pattern: Many applications have a workload pattern that follows the human living-cycle (i.e., large query volume during the day and little to no query volume during the night [137, 205]). We simulate such a pattern by pausing the TPC-C workload for one minute starting at 40s during a 160s period. And we let the self-driving DBMS plan for action sequences starting at 10s. We also added a **NO-OP** baseline that does not apply any actions.

Figure 5.5a shows the average query latency during the workload period for PB0 and GREEDY-SEQ. GREEDY-SEQ and NO-OP's query latencies on average are 49% and 55% slower than PB0. GREEDY-SEQ chooses to build the same pair of secondary indexes as PB0. And both methods use eight threads to create the indexes for fast completion. When the DBMS creates indexes, the queries becomes 63% slower because of the resource competition. Unlike PB0, GREEDY-SEQ's heuristic method does not account for the action application time nor the action's interference to the queries during its application. So it builds the two indexes consecutively, with the `CUSTOMER` index at first, which the behavior models estimate to take 2.5× more time to build but reduces the workload queries' latency 9× more than the `OORDER` index. In contrast, PB0 recognizes that the DBMS cannot finish creating the `CUSTOMER` index before the workload pauses. Thus, it builds the cheaper `OORDER` index first and builds the `CUSTOMER` index after the workload pauses. Therefore, the index creations have minimal interference to the queries and the new indexes significantly accelerate the queries after the workload resumes. [2]

Growth Pattern: There are also applications that have a growth pattern where their database sizes increase over time [137]. But the hardware that runs the DBMS often has a fixed resource limit, such as the maximum memory consumption.[3] We simulate this pattern by increasing the ratio of the `NEWORDER` transaction in TPC-C to 90% and decrease the ratio of other transactions. The workload starts with 15 GB memory consumption, and we set the system's memory constraint to 15.15 GB. We execute the workload for 60s and again let the DBMS plan for action sequences starting at 10s.

The results in Figure 5.5b show that GREEDY-SEQ's query latency on average is 55% higher than PB0. GREEDY-SEQ creates the index on `CUSTOMER` at 10s, but has to drop it right after

[2]NoisePage's knobs do not have measurable performance impact on TPC-C so PB0 does not change them.

[3]NoisePage does not support resource scaling, so PB0 cannot choose to increase the system's resource capacity.

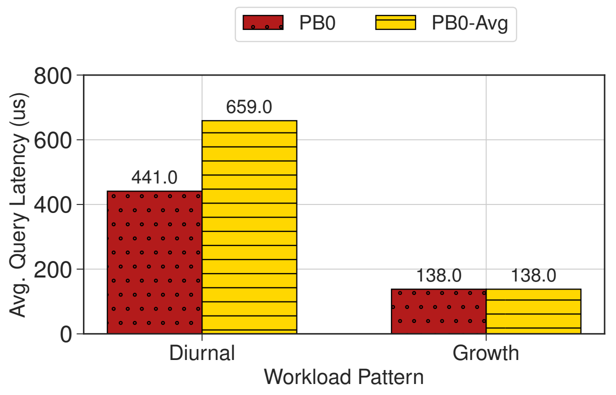


Figure 5.6: MCTS Search Strategy – Average workload query latency with actions chosen by PB0 with different MCTS search strategies.

Strategy	Action	Level	Interval	Min	Max	Avg	Std
PB0	`CREATE INDX ON OORDER`	1	0-2	6.4e8	8.8e8	7.9e8	6.1e7
	NO-OP	2	3-3	6.4e8	8.8e8	7.7e8	7.1e7
	NO-OP	3	5-5	6.4e8	8.3e8	7.5e8	7.8e7
	`CREATE INDX ON CUSTOMER`	4	9-13	6.4e8	8.2e8	7.8e8	6.6e7
PB0-Avg	`CREATE INDX ON CUSTOMER`	1	0-6	6.6e8	9.0e8	8.2e8	6.1e7
	`CREATE INDX ON OORDER`	2	7-9	7.1e8	9.0e8	8.0e8	6.0e7
	NO-OP	3	10-10	7.1e8	9.0e8	7.5e8	6.7e7

Table 5.1: MCTS Search Strategy Statistics – Statistics of the leaf rollout values for the nodes corresponding to the best action sequence under the diurnal workload pattern.

the creation finishes because the system exceeds the memory constraint. PB0, on the contrary, does not build any indexes because its MCTS recognizes that the system memory consumption is increasing (Section 5.3.4) and the index creation exceeds the memory constraint threshold.

Figure 5.5c shows the system's memory consumption over time under the growth workload pattern. The memory consumption with GREEDY-SEQ hits the constraint at the end of the index creation, so the system needs to drop the index immediately. PB0's MCTS has also explored the same action choice as GREEDY-SEQ (denoted as **PB0-Estimated**) during its search before the system applies any action. Though it underestimates the time to create the index by 25% due to imperfect interference estimation, it correctly predicts that the system memory consumption exceeds the threshold before the index creation finishes. Thus, PB0 avoids the unnecessary index creation and resource contention.

5.6.2 MCTS Search Strategy

MCTS's search strategy is the key for finding actions sequences that benefits the system performance as the strategy balances the search's exploration and exploitation. We next evaluate the effectiveness of different MCTS search strategies for PB0. As discussed in Section 5.3.2, a

Workload Pattern	Caching Choice	ASC Hit Rate	MIC Hit Rate
Diurnal	**PB0**	40%	97%
	PB0 w/o ASC	N/A	98%
	PB0 w/o MIC	40%	N/A
Growth	**PB0**	80%	96%
	PB0 w/o ASC	N/A	98%
	PB0 w/o MIC	80%	N/A

Table 5.2: Inference Caching Statistics – Hit rate statistics under different choices of inference caching.

straightforward MCTS search strategy is to use the average rollout value among all leaves as a node's value number (denoted as **PB0-Avg**). PB0 adapts this strategy to use the minimum rollout value among leaves that better suits self-driving DBMSs. We evaluate the search strategies under the same workload settings as Section 5.6.1 with the same candidate actions and environments.

Figure 5.6 shows the measured average query latency over the entire workload. The two strategies make the same decision under the growth workload pattern because the pattern is simple (growing under a fixed rate). However, PB0-Avg makes a similar decision as GREEDY-SEQ under the diurnal workload pattern, which results in 49% higher query latency than PB0. The variance among the leaves' rollout values affects PB0-Avg to choose the best action sequences tailored to a more complex workload pattern.

Table 5.1 shows more details for the nodes corresponding to the best action sequence of the two search strategies under the diurnal workload pattern. For each node, we show its level in the search tree, the estimated time intervals [4] to apply its action, and the statistics of all its leaves' rollout values. These statistics show that there can be large difference (up to 38%) between the min (best) and max (worst) rollout values of a node's leaves. Thus, the outliers (e.g., leaves with the highest values) can affect the average rollout value for a node's leaves. Moerover, the max rollout values of PB0 are lower than PB0-Avg's. And despite PB0-Avg selects actions based on the average rollout values of a node's leaves, its best action sequence's average rollout values are mostly higher than PB0. These statistics suggest that PB0 selects more effective actions than PB0-Avg.

5.6.3 Inference Cache

Lastly, we evaluate the efficacy of PB0's two-level inference cache (Section 5.4). We aim to demonstrate the benefit of each level of the cache and their combined effect. We remove the action state cache (**ASC**) and model inference cache (**MIC**) from PB0 separately, denoted as **PB0 w/o ASC** and **PB0 w/o MIC**. We compare their planning time against PB0 and GREEDY-SEQ under the same workload settings in Section 5.6.1.

The results are shown in Figure 5.7. GREEDY-SEQ has the fasted planning time since it uses a heuristic method without an exhaustive search. And PB0's two caching methods significantly

[4] Recall from Section 5.6.1 that each planning time interval is 5s.

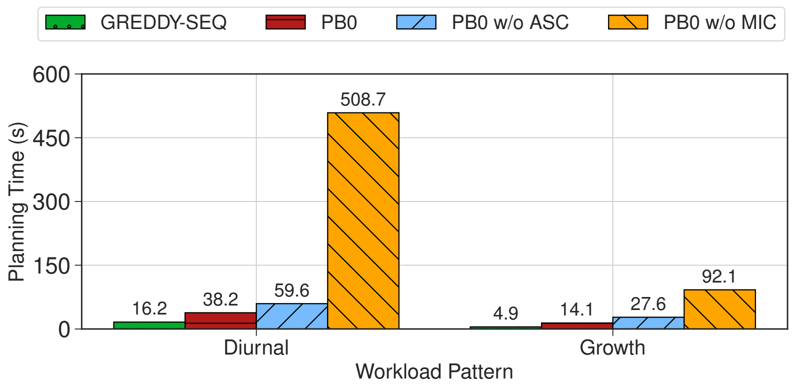


Figure 5.7: Inference Caching – Planning time of GREEDY-SEQ and PB0 with different inference caching methods.

reduce its planning time. The MIC on average reduces the planning time 13× more than the ASC because there are many repeated model inference requests during the search, even among different actions. For example, the workload may contain queries that neither the candidate secondary indexes affect. Thus, PB0 always infer the same OUs for these queries for all the nodes in the same search tree level. And PB0 only needs to make one inference for them with the MIC.

The cache hit rate statistics shown in Table 5.2 verify such analysis. PB0 w/o ASC has on average 38% higher cache hit rate than PB0 w/o MIC over the two workload patterns. PB0 and PB0 w/o MIC has the same cache hit rate for ASC because MIC is at lower level than ASC and does not affect ASC's hit rate. In contrast, PB0 has lower MIC hit rate than PB0 w/o ASC because ASC already removes certain repetitive inference requests. Lastly, PB0 has 2× higher ASC hit rate under the growth workload pattern compared to the diurnal workload pattern because the growth pattern is simpler and thus the explored actions sequences are less diverse.

Chapter 6

Related Work

In this chapter, we discuss the prior work related to self-driving DBMSs. We first explain previous efforts on automated DBMS tuning, which has been studied for decades. We then introduce the methods related to the three frameworks that constitute our self-driving architecture: workload forecasting, behavior modeling, and action planning. Lastly, we briefly discuss the work on learned DBMS components.

6.1 Automated DBMS Tuning

Previous researchers in the last 40 years have studied and devised several methods for automatically optimizing DBMSs [32, 45, 47, 222]. As such, there is an extensive corpus of previous work, including both theoretical [40] and applied research [72, 224, 253]. But no existing system incorporates these techniques to achieve full operational independence without requiring humans to monitor, manage, and tune the DBMS because they (1) cannot predict the future workload to optimize the DBMS proactively and apply changes at proper times, (2) require expensive exploratory testing on a copy of the database, or (3) only target one system aspect and cannot optimize the system holistically. We categorize these techniques based on their tuning targets, including physical design tuning, knob configuration tuning, hardware resource tuning, and autonomous DBMSs.

Physical Design Tuning: Physical design tuning addresses the problem of selecting the best physical design for a given database and its workload. Such tuning optimizes one or more *metrics* for the workload that satisfies *budget constraints*. The metrics are the desired performance outcomes when the DBMS executes the workload, such as minimizing the average query latency. The budget constraints are defined in terms of hardware resources, such as storage (e.g., the maximum amount of memory allowed for creating new indexes) or CPU overhead (e.g., the total time to create the indexes). Previous work on physical design tuning focuses on three areas: index selection, table partitioning, and materialized view selection.

Some early index selection algorithms determine the benefits of indexes using characteristics of the data, such as Drop [223]. In the 1990s, Microsoft's AutoAdmin pioneered the use of leveraging the DBMS's built-in cost models from its query optimizer to estimate the benefits of indexes, which is called "what-if" API [44]. This avoids a disconnection between what the tool chooses as a good index and what the system uses at runtime when it generates query plans. Most contemporary index selection algorithms leverage such "what-if" API [118]. Some algorithms optimize the index benefit per storage unit [36, 59, 214] while others aim to maximize the total benefit under a storage budget [1, 44, 223]. Most of these algorithms support the selection of multi-column indexes and consider the index interaction.

Table partitioning divides a table into multiple disjoint fragments. The values of one or more table columns determine the boundaries of these fragments, typically with either range partitioning or hash partitioning [88, 251]. Previous database partitioning approaches mainly focus on (1) generating the candidate partitioning attributes and (2) searching for the optimal partitioning scheme. The former examines the benefits of each attribute given a sample worklaod [19, 41, 167, 251]. There are techniques to prune the candidate set [252] or combine the candidates to generate multi-attributes for partitioning [17]. The latter searches for the optimal partitioning attributes for each table using greedy strategy [167], approximation method [53], or exhaustive search [251, 252]. There are also search methods that leverage the source code information [213], minimize the contention bottlenecks [157], or adaptively adjust the partitioning scheme [187].

Many materialized view selection algorithms share similar ideas with techniques for multi-query optimization and common subexpression selection [86, 91, 110, 177, 190, 246]. For example, Microsoft SCOPE uses an ILP-based formulation to select sub-expressions for large workloads by finding common parts of query logical plans and materializing them to accelerate subsequent queries [109]. Most commercial database vendors offer automated physical design tools that support materialized view selection. For example, IBM's DB2 Advisor recommends materialized view using an algorithm based on Knapsack problem [253, 254]. The DB2 Advisor also exploits multi-query optimization techniques to construct the candidates [129]. The Oracle Autonomous Database on the Cloud uses an extended covering subexpression algorithm for its materialized view selection [22]. Microsoft's AutoAdmin adopts a three-step search process to deal with the large space of materialized view alternatives [18]. There are also recent attempts that use reinforcement learning for materialized view recommendation [94, 134, 242].

Knob Configuration Tuning: DBMSs' built-in cost models estimate the amount of work for a particular query execution [196]. These models' original intention is for the query optimizer to compare alternative query execution strategies under a specific DBMS configuration. Thus, unlike physical design tuning, knob configuration tuning cannot leverage these built-in models.

All major DBMS vendors provide proprietary tools to help users determine the knob configuration [62, 123, 151]. And these tools have varying amounts of automation [29]. For example, the DB2 Performance Wizard tool released by IBM in the early 2000s asks high-level questions about the application (e.g., whether the application is transactional or analytical) to the users and then recommends knob settings accordingly [125]. IBM engineers need to manually create models used by this tool. A later version of DB2 has a self-tuning memory manager that determines the memory allocation of the DBMS's internal components using heuristics [201, 211]. Oracle has techniques to determine the system bottleneck due to sub-optimal knob configurations [62, 123]. Oracle also develops a SQL analyzer to estimate the impact of the changes in knob configurations [28, 234]. Microsoft's SQL Server has a Resource Advisor that predicts the effect of buffer pool size changes on OLTP workloads [151].

There are also knob configuration tuning tools developed for open-source DBMSs. For example, DBSherlock helps DBAs diagnose performance problems for MySQL by identifying anomalies in the DBMS's performance time-series data [239]. iTuned is a generic configuration used in MySQL or PostgreSQL [72]. It continuously evaluates incremental changes to the knob configurations using a workload sample when the DBMS is under low demand.

Recent efforts have focused on using ML to help tune DBMS configuration knobs. OtterTune is a DBMS tuning service similar to iTuned, except that it leverages data collected from previous tuning sessions for different workloads to train ML models to evaluate configuration changes [216]. Extensions of OtterTune include replacing Ottertune's Gaussian Process Regression model with an actor-critic reinforcement learning model [243] (CDBTune) or a general purpose configurator [191] (using `irace`), incorporating query information in the model input features [131] (QTune), and optimizing resource utilization instead of system throughput [245] (ResTune). These techniques have also been evaluated on workloads from FoundationDB [184] and a multi-national bank [217]. iBTune uses a pairwise deep neural network to recommend individualized buffer pool sizes to database instances in the cloud [206].

Hardware Resource Tuning: Much previous work on hardware resource tuning focuses on the enterprise database workload consolidation. These techniques construct models and estimate the workload resource consumption based on a "representative workload" provided for each application. For example, one technique uses the optimizer ("what-if" API [44]) to estimate the impact of resources to configure virtual machines for database workloads [196]. Kairos executes the representative workloads on a spare hardware to capture their resource utilization characteristics and consolidates these workloads [54]. DBSeer uses analytical models to estimate the resource utilization for a set of transactions to estimate the resource consumption if the arrival rates of these transactions change [148]. Some works also try to automatically scale resources for cloud databases [56, 87, 176] (which we provide more details in Section 6.2) and predicting resource usage for individual queries [21, 23, 73, 83, 133] (which we provide more details in Section 6.3).

Autonomous DBMSs: To the best of our knowledge, there has never been a fully autonomous DBMS that achieves Level 5 autonomy we defined in Section 2.2. In the early 2000s, there were several groundbreaking solutions that moved towards this goal [45], most notably Microsoft's SQL Server AutoAdmin [47] and IBM's DB2 Database Advisor [167, 199, 253]. Others proposed RISC-style DBMS architectures made up of inter-operable components that make it easier to reason about and therefore control the overall system [222]. But over a decade later, all of this research has been relegated to standalone tools that assist DBAs and not supplant them.

The closest attempt to a fully automated DBMS was IBM's proof-of-concept for DB2 from 2008 [224]. This system used existing tools [125] in an external controller and monitor that triggered a change whenever a resource threshold was surpassed (e.g., the number of deadlocks). This prototype still required a human DBA to select tuning optimizations and to occasionally restart the DBMS. And unlike our proposed techniques, it could only react to problems after they occur because the system lacked forecasting.

Automation is more common in cloud computing platforms because of their scale and complexity [55, 111]. Microsoft appears to be again leading research in this area. Their Azure service models resource utilization of DBMS containers from internal telemetry data and automatically adjusts allocations to meet QoS and budget constraints [56]. There are also controllers for applications to perform black-box provisioning in the cloud [15, 39, 172]. The authors in [176] use the same receding-horizon controller model [171] that we propose, but they only perform simple changes to the system. Oracle announced their own "self-driving" DBaaS in 2017 [6]. Users select one of the three types of autonomous DBMS services with varying automation features: data warehouse, transaction processing, or JSON database. Although there is little public information about its implementation, our understanding from their developers is that it runs their previous tuning tools in a managed environment. Some of its automation also require expensive testing of candidate changes (e.g., indexes) on replicas. Oracle also just released the MySQL Autopilot that automates certain aspects for their MySQL Cloud Database Service, including provisioning, data loading, query execution, and failure handling [5].

Recent work explores using reinforcement learning (RL) [193, 204] to tune specific aspect of the DBMS, such as indexes [127, 179, 188], materialized views [94, 134, 242], and knob configurations [89, 131, 243]. These approaches are each limited to a single decision problem in a static operating environment and do not consider long-term forecasts in their models. They also often require expensive tests on data copies to collect a large amount of training data. We use a modularized (forecasting-modeling-planning) design for self-driving DBMSs, which has a different methodology than RL-based approaches. We think this approach provides better data efficiency, debuggability, and explainability, which are essential for the system's practical application. All major organizations working on self-driving cars also use a modularized approach instead of end-to-end RL [136].

6.2 Workload Forecasting

We classify the previous work on workload forecasting and modeling in systems into several categories: resource estimation for auto-scaling, performance diagnosis and modeling, shift detection, and workload characterization for system design.

Resource Estimation: There are works on automatically identifying the workload trends and scaling resources for provisioning. Das *et al.* proposed an auto-scaling solution for database services using a manually constructed hierarchy of rules [56]. The resource demand estimator derives signals from the DBMS's internal latency, resource utilization, and wait statistics to determine whether there is a high or low demand for the resource. Their work focuses on short-term trends and estimates the demand for each resource in isolation. Other works investigated scaling resources proactively in cloud platforms [87, 176]. All of these methods estimate whether there will be a demand for each type of resource in the near future. In contrast, we model the query arrival rates for both short- and long-term horizons to support complex optimization planning decisions.

Performance Diagnosis: There is previous research on the modeling and diagnosis of DBMS performance. DBSeer predicts the answer of "what-if" questions given workload changes, such as estimating the disk I/O for future workload fluctuations [148]. The model clusters the workload based on transaction types and predicts the resource utilization of the system based on transaction mixtures. It is an off-line model that assumes fixed types of transactions. Our work not only looks at the current workload mixtures but also predicts the future workload. DBSherlock is a diagnostic tool for transactional databases that uses causal models to identify the potential causes of abnormal performance behavior and provides a visualization [239]. The modeling in this line of work aims to help the DBAs understand their system and identify bottlenecks. The authors in [151] use analytical models to continuously monitor the relationship between system throughput, response time, and buffer pool size.

Shift Detection: Others have used Markov models to predict the next SQL statement that a user will execute based on what statements the DBMS is currently executing [100, 159]. The authors in [71] combine Markov models with a Petri-net to predict the next transactions. The technique proposed in [100] extends this model to detect workload shifts. Previous work combines these techniques with a workload classification method to model periodic and recurring patterns of the workload [101, 102]. These methods capture certain patterns in a workload, but none of them are able to predict the volume, duration, and changes of workloads in the future.

Workload Characterization: The workload compression technique in [48] shares a similar goal with our work. A set of SQL DML statements are compressed by searching which queries to remove from the workload with an application-specific distance function $D(q_i, q_j)$ for any pair of SQL queries q_i and q_j. This technique does not model the temporal pattern of the queries. Previous works also use set representation to model a database workload [181, 182]. They assume that all transactions arriving at the system belong to a fixed set of pre-defined transaction types. Various collected/aggregated runtime statistics are also used to analyze the structure and complexity of SQL statements and the runtime behaviors of the workload [240].

6.3 Behavior Modeling

We broadly classify the previous work in DBMS modeling into the following categories: (1) ML models, (2) analytical models, and (3) methods for handling concurrent workloads.

Machine Learning Models: Most ML-based behavior models use query plan information as input features for estimating system performance. Techniques range from Ganapathi et al.'s subspace projections [83] to Gupta et al.'s PQR hierarchical classification of queries into latency buckets [90]. Some methods mix plan- and operator-level models to balance between accuracy and generality, compensating further for generalizability issues by adjusting pre-built off-line models at runtime [23] or by building additional scaling functions [133]. QPPNet [141] predicts the latency of query execution plans by modeling the plan tree with deep neural networks.

ML-based models still require developers to adjust features, recollect data, and retrain them to generalize to new environments. Most frameworks also train models on a small number of synthetic benchmarks and thus have high prediction errors on different workloads. Furthermore, these ML-based models do not support transactional workloads and ignore the effects of internal DBMS operations. This is in contrast to MB2 that builds a holistic model for the entire system that accounts for internal operations and handles software updates with limited retraining over SQL.

Analytical Models: Researchers have also built analytical models for DBMS components. DBSeer builds off-line models that predict the resource bottleneck and maximum throughput of concurrent OLTP workloads, given fixed transaction types and fixed DBMS configuration and physical design [148]. Wu et al. tunes the optimizer's cost models for better query execution time predictions through off-line profiling and on-line refinement of cardinality estimates [231, 232]. Narasayya and Syamala provides DBAs with index defragmentation suggestions by modeling workload I/O costs [150].

Concurrent Queries: Due to the difficulty of modeling concurrent operations as described in Section 4.1.2, the aforementioned techniques largely focus on performance prediction for one query at a time. For concurrent OLAP workloads, researchers have built sampling-based statistical models that are fixed for a known set of queries. The predictions have seen applications to scheduling [21], query execution time estimation [73], and overall analytical workload throughput prediction [74]. Wu et al. support dynamic query sets by modeling CPU, I/O interactions, and buffer pool hit rates with queuing networks and Markov chains [230]. More recently, GPredictor predicts concurrent query performance by using a deep learning prediction network on a graph embedding of query features [249]. But it requires the exact interleaving of queries to be known, which is arguably impossible to forecast and thus does not apply to MB2's context. All of these methods also require updating their models whenever the DBMS's software changes.

6.4 Action Planning

We categorize previous work on action planning for DBMS into (1) sequence planning methods, (2) on-line tuning methods, and (3) MCTS applications in DBMS tuning.

Sequence Tuning: As discussed in Section 6.1, most previous work on DBMS tuning focus on static workloads: these techniques recommend a set of changes (actions) for the DBMS given a set of queries (typically provided by DBAs) without any temporal information. In contrast, a self-driving DBMS needs a planning method that not only determines what actions to apply, but also decides when and how to apply the actions given the temporal patterns of the forecasted workload (Section 2.2). To the best of our knowledge, the most recent method that addresses such a planning problem is GREEDY-SEQ [20]. Similar to PB0 (Chapter 5), GREEDY-SEQ first uses previous methods for static workload tuning to generate a set of candidate actions. It then uses heuristics to determine the best timing to apply each action separately given a query sequence. It merges the decision for each individual action to generate the best sequence of actions based on the workload pattern. GREEDY-SEQ does not account for the action application time nor the action's interference to the queries during its application. Its heuristic method may also miss optimization opportunities.

On-line Tuning: There are DBMS tuning methods developed for on-line scenarios. For example, Microsoft proposed a method that continuously modifies the DBMS physical design reacting to changes in the query workload [37]. It generates candidate physical designs that may improve the performance during query optimization. It then keeps identifying the most promising candidates using the statistics gathered from query execution. COLT is another framework that continuously monitors the workload and creates additional indexes [185]. It gathers performance statistics at different levels of detail to allocate more profiling resources to the most promising candidate indexes. DBA bandit learns the benefits of candidate indexes through strategic exploration and performance observation from the workload instead of using the optimizer's estimates [185]. These methods identify promising changes to the DBMS based on the most recent workload but do not plan actions proactively based on future workload patterns.

MCTS Applications: There are also applications of MCTS in DBMS tuning besides action planning. openGauss uses MCTS to rewrite a single query to more efficient equivalent queries [89]. UDO uses MCTS to explore potential configurations for a DBMS given a workload and converge to near-optimal solutions [220]. UDO still focuses on static workloads without temporal information. In contrast, we use MCTS to generate the best sequence of tuning actions considering the workload pattern.

6.5 Learned DBMS Components

Recently, there have been attempts to use ML, especially deep learning [221], to develop "learned" DBMS components that replace their canonical counterparts. For example, a main area of the research focus has been learned cardinality estimators that replace traditional histograms [75, 97, 112, 158, 189, 203, 227, 235, 237, 247]. Learned indexes have been proposed to replace traditional index techniques (e.g., b+-tree, radix tree) [16, 52, 66, 67, 80, 113, 119, 229]. There are also studies on learned query optimizers [121, 140, 142, 143, 228, 241], learned checkpoint optimizers [238], and learned filters [61]. Moreover, there are platforms that integrate these learned components into a single DBMS [89, 120, 248].

These methods are orthogonal to the goal of self-driving DBMSs as the former replaces the heuristics in canonical DBMS components with ML models, and the latter automates the DBMS administrative tasks. Managing the learned components may increase the DBMS administration challenge as users need to decide whether to use such learned components, what learned components to use, when to deploy the learned components, and how to deploy the learned components. For example, learned indexes may replace the traditional B+-tree with ML models, but users still need to determine on which columns to build a learned index, when to build the learned index, and how to build the learned index (e.g., using one CPU core or eight CPU cores). Learned DBMS components and self-driving DBMSs may also complement each other.

For example, our self-driving architecture's behavior modeling framework (Chapter 4) may use learned cardinality estimators to generate more accurate input features for the behavior models.

Chapter 7

Future Work

We believe the efforts towards self-driving DBMSs (Level 5 autonomy as defined in Section 2.2) are just beginning. Thus, we think there are several open questions beyond the topics covered in this thesis that are worth exploring. We now present some of these future research directions, including extensions of our forecasting-modeling-planning architecture to improve its self-driving capabilities and the broader opportunities and challenges for self-driving DBMSs.

7.1 Error Correction and Feedback Control

Although our forecasting-modeling-planning architecture strives to make the self-driving DBMS select the best actions, there may still be applied actions with behavior that does not match the system's expectation. For example, an action may provide less performance benefit than the system predicts or take longer to finish than the system estimates. There are two possible sources for these estimations errors: the workload forecasts and the behavior model estimations. Even when these estimations are accurate, the system may still select sub-optimal actions because the action planning cannot exhaust all possible action sequences. Thus, one important area of further research is to incorporate the feedback from self-driving DBMS deployments to improve the model estimations and the action effectiveness.

The workload forecasting framework may use change point detection techniques [26] to intelligently identify workload changes and proactively re-train its forecasting models. It may also use anomaly detection techniques [42] to exclude outliers from the model training. There are also remaining challenges, such as properly determining the time range of the historical data that the model re-training should use: using more training data may improve the model generalization, but outdated training data may also affect the model accuracy.

The behavior modeling framework may incorporate the action behavior observed in on-line deployments into the models to correct mispredictions. But how to perform such incorporation is still an open question. For example, the modeling framework may combine the off-line training data from the runners with the on-line data to train new behavior models. But the off-line data may quantitatively overwhelm the on-line data, and thus the model training may not fully exploit the information in the on-line data. The modeling framework may also train a separate model only with the on-line data. But it may be challenging to properly combine the predictions from the on-line and off-line models.

The self-driving DBMS may also directly leverage the observed information of the applied actions in its action planning framework to improve the search (MCTS) effectiveness. There are several possible approaches. For example, the action planning framework may use a simple "black-list" to avoid searching over actions that do not improve the system performance. It may also adjust an action's estimated behavior during MCTS with the observed performance metrics for that action. Lastly, it may learn a high-level policy over the applied actions to enhance the action selection during MCTS [192, 193].

7.2 Supporting Complex DBMS Features

Recall from Chapter 5 that we only evaluate our forecasting-modeling-planning architecture with actions that create/drop indexes and adjust knob configurations due to NoisePage's feature limitation. Many real-world applications require more complex DBMS features, such as materialized view [22] or resource scaling [58]. We believe our proposed architecture can extend to automate the tuning for these features.

We do not anticipate that supporting more DBMS features requires changes in the workload forecasting framework, which predicts the query arrival rates that are independent from the DBMS configuration. We also anticipate that the infrastructure in the action planning framework requires minimal extensions, such as generating new types of candidate actions, to support tuning additional features. This shows the flexibility and maintainability of our modularized self-driving architecture. The behavior modeling framework needs to extend the OU-models for the additional DBMS features. It is interesting to evaluate the framework's effectiveness with such extension. While the principle of OU-decomposition can apply to any DBMS features, developers need to decide the boundaries and input features for the new OU-models. Lastly, more DBMS features introduce more candidate actions to the action planning framework, which may enable opportunities to further optimize the action search.

7.3 Self-Driving Aggressiveness Level

Our action planning framework aims to find a sequence of actions that best improve the DBMS performance based on the self-driving objective. However, as discussed in Section 7.1, an applied action's behavior may not match the system's expectation. Thus, it is possible that a self-driving action incurs high overhead that is unnecessary. For example, the workload forecasting framework may significantly over-predict the future workload volume. Thus, a self-driving DBMS may apply actions to provision many unnecessary resources (e.g., machines), which may also negatively impact the system performance because of the data re-arrangement.

One possible solution is to have multiple self-driving aggressiveness levels. For example, the highest aggressiveness level may allow the system to apply any possible actions to optimize the self-driving objective. A lower aggressiveness level may prohibit the system from using actions that drastically impact the system performance (e.g., creating an expensive index or significantly scaling up the resources). Thus, the system may have sub-optimal but more stable performance under the lower aggressiveness level, which may better suit many applications [215, 217]. Furthermore, Furthermore, the system can also use the lower aggressiveness level when the observed action behavior significantly deviates from the expectation, especially before the system can finish incorporating the feedback (e.g., re-training the models).

7.4 Self-Driving across the Stack

DBMSs are typically deployed with one or more frontend/middleware applications (e.g., websites [108], reporting software [147], or in-memory caches [35]). The configurations of these applications may impact the performance of the DBMS. For example, the content in the memory cache may determine what queries the DBMS receives, which affects the DBMS's tuning decisions. Thus, it may be beneficial to extend the self-driving architecture to support tuning software systems across the stack.

There are many open research questions in this direction. For example, there can be diverse software combinations deployed with a DBMS. It is unclear what the appropriate granularity should be to model behavior of all the systems in a deployment. Similar to the challenge of DBMS behavior modeling (Chapter 4), a monolithic model that captures the behavior of all software combinations may be infeasible because of the model complexity. But applying our OU-based decomposed modeling approach on each software system may also be challenging since it requires close familiarity with these systems and source code instrumentation. One may also use off-the-shelf ML models to model each system as a "black-box". But the model dimensionality may still be high, which poses challenges on training data collection and debugging.

7.5 Human-Intelligent-System Interaction

Though a self-driving DBMS can control itself autonomously, there can still be external information humans may provide to the DBMS to further enhance its performance. For example, a human may be aware of an immense workload volume increase in the future that does not occur before, and thus the DBMS cannot infer the increase from the workload history. If the human can provide hints on such an increase, the DBMS can plan appropriate actions proactively. Alternatively, the human may also directly hint the DBMS to scale up the resources before the workload increase.

We believe it is exciting to explore how humans should interact with future intelligent systems, including self-driving DBMSs. Certain information may be straightforward to transfer between humans and the self-driving DBMS. For example, the workload forecasting framework may adjust the query arrival rate predictions according to the workload volume hinted by the human. And the behavior modeling and action planning framework can remain intact. Other information may be more challenging to convey. For example, a specific action hinted at by a human may conflict with the actions planned by the self-driving DBMS, such as competing resources. And the system must properly resolve such conflict even the motivation (e.g., workload increase) for such hinted action may not be revealed.

Chapter 8

Bibliography

[1] Dexter. https://github.com/ankane/dexter. 2.2, 6.1

[2] MySQL. https://www.mysql.com/, . 2.2, 3, 3.6.6

[3] MySQLTuner. https://github.com/major/MySQLTuner-perl, . 2.2

[4] Open Learning Initiative. http://oli.cmu.edu. 3.1.1

[5] MySQL HeatWave. https://www.oracle.com/mysql/heatwave/, . 6.1

[6] Oracle Self-Driving Database. https://www.oracle.com/database/autonomous-database/index.html, . 2.1, 2.2, 3.5, 6.1

[7] PGTuner. https://github.com/jfcoz/postgresqltuner. 2.2

[8] PostgreSQL. https://www.postgresql.org/. 2.2, 3, 3.6.6

[9] SQLite. https://sqlite.org/. 2.2

[10] Bpf compiler collection (bcc). https://github.com/iovisor/bcc/, 2020. 4.5.1

[11] Google benchmark support library. https://github.com/google/benchmark, 2020. 4.7

[12] libpqxx – official c++ client api for postgresql. http://pqxx.org/development/libpqxx/, 2020. 4.7

[13] Processor counter monitor (pcm). https://github.com/opcm/pcm/, 2020. 4.5.1

[14] scikit-learn: Machine learning in python. https://scikit-learn.org/stable/, 2020. 4.7

[15] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. A control-based framework for self-managing distributed computing systems. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, WOSS '04, pages 3–7, 2004. 6.1

[16] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H Chi, Lyric Doshi, Tim Kraska, Andy Ly, Christopher Olston, et al. Learned indexes for a google-scale disk-based database. *arXiv preprint arXiv:2012.12501*, 2020. 6.5

[17] Sanjay Agrawal and *et al.* Integrating vertical and horizontal partitioning into automated physical database design. SIGMOD, 2004. 1, 2.2, 6.1

[18] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. VLDB, 2000. ISBN 1-55860-715-3. 1,

2.2, 6.1

[19] Sanjay Agrawal, Surajit Chaudhuri, Abhinandan Das, and Vivek Narasayya. Automating layout of relational databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 607–618. IEEE, 2003. 6.1

[20] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic physical design tuning: workload as a sequence. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 683–694, 2006. 2.3, 5, 5.3, 5.6, 6.4

[21] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 449–460. ACM, 2011. 6.1, 6.3

[22] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. Automated generation of materialized views in oracle. *Proceedings of the VLDB Endowment*, 13(12):3046–3058, 2020. 5, 6.1, 7.2

[23] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. Learning-based query performance modeling and prediction. In *28th International Conference on Data Engineering*, pages 390–401. IEEE, 2012. 3.5.1, 4, 4.1.2, 4.7.3, 6.1, 6.3

[24] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2o: A hands-free adaptive store. SIGMOD, pages 1103–1114, 2014. 1

[25] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The cost of serializability on platforms that use snapshot isolation. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 576–585. IEEE Computer Society, 2008. 4.7

[26] Samaneh Aminikhanghahi and Diane J Cook. A survey of methods for time series change point detection. *Knowledge and information systems*, 51(2):339–367, 2017. 7.1

[27] Francisco Javier Baldan Lozano, Sergio Ramirez-Gallego, Christoph Bergmeir, Jose Benitez, and Francisco Herrera. A forecasting methodology for workload forecasting in cloud systems. PP:1–1, 06 2016. 3.6.2

[28] P. Belknap, B. Dageville, K. Dias, and K. Yagoub. Self-tuning for SQL performance in Oracle Database 11g. In *ICDE*, pages 1694–1700, 2009. 6.1

[29] Darcy G Benoit. Automatic diagnosis of performance problems in database management systems. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 326–327. IEEE, 2005. 6.1

[30] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. 3.4.2

[31] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012. 4.5.2

[32] Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, HV Jagadish, et al. The asilomar report on database research. *SIGMOD record*, 27(4):74–80, 1998. 6.1

[33] Herman J. Bierens. The nadaraya-watson kernel regression function estimator. In *Topics in Advanced Econometrics: Estimation, Testing, and Specification of Cross-Section and Time Series Models*, pages 212–247. Cambridge University Press, 1994. doi: 10.1017/CBO9780511599279.011. 3.5.1

[34] Byron Boots, Geoffrey J. Gordon, and Arthur Gretton. Hilbert space embeddings of predictive state representations. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013*, 2013. 3.6.2

[35] Dhruba Borthakur. Petabyte scale databases and storage systems at facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1267–1268, 2013. 7.4

[36] 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*, pages 227–238, 2005. 6.1

[37] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 826–835. IEEE, 2007. 5, 6.4

[38] Carnegne Mellon Database Group. Noisepage. https://noise.page/, 2020. 1, 4

[39] Emmanuel Cecchet, Rahul Singh, Upendra Sharma, and Prashant Shenoy. Dolly: Virtualization-driven database provisioning for the cloud. VEE '11, pages 51–62, 2011. 6.1

[40] S. Ceri, S. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng.*, 9(4):487–504, 1983. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1983.234957. 6.1

[41] Stefano Ceri, Shamkant Navathe, and Gio Wiederhold. Distribution design of logical database schemas. *IEEE Transactions on Software Engineering*, (4):487–504, 1983. 6.1

[42] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection for discrete sequences: A survey. *IEEE transactions on knowledge and data engineering*, 24(5):823–839, 2010. 7.1

[43] Guillaume M JB Chaslot, Mark HM Winands, H JAAP VAN DEN HERIK, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008. 5.1.3

[44] Surajit Chaudhuri and Vivek Narasayya. Autoadmin "what-if" index analysis utility. *ACM SIGMOD Record*, 27(2):367–378, 1998. 5.5, 6.1, 6.1

[45] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14, 2007. 1, 2.1, 6.1, 6.1

[46] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. VLDB, pages 146–155, 1997. ISBN 1-55860-470-7. 1, 2.2, 3.6.6, 5.1.1, 5.2, 5.6.1

[47] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB*, pages 1–10, 2000. 6.1, 6.1

[48] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. Compressing sql workloads. In *Proceedings of the 2002 International Conference on Management of Data*, pages 488–499. ACM, 2002. 2.2, 3.4.1, 1, 6.2

[49] Eunkyoung G Cho, Kristin A Thoney, Thom J Hodgson, and Russell E King. Supply chain planning: Rolling horizon scheduling of multi-factory supply chains. In *Proceedings of the 35th conference on Winter simulation: driving innovation*, pages 1409–1416. Citeseer, 2003. 5.1.2

[50] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. Demonstration of the cosette automated sql prover. In *Proceedings of the 2017 International Conference on Management of Data*, pages 1591–1594. ACM, 2017. 3.3

[51] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006. 5, 5.1.3

[52] Andrew Crotty. Hist-tree: Those who ignore it are doomed to learn. In *CIDR*, 2021. 6.5

[53] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. Schism: a workload-driven approach to database replication and partitioning. 2010. 6.1

[54] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 313–324, 2011. 6.1

[55] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1): 5:1–5:45, April 2013. 6.1

[56] Sudipto Das, Feng Li, Vivek R Narasayya, and Arnd Christian König. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1923–1934. ACM, 2016. 3, 6.1, 6.1, 6.2

[57] Sudipto Das, Feng Li, and *et al.* Automated demand-driven resource scaling in relational database-as-a-service. SIGMOD, pages 1923–1934, 2016. 1, 2.2

[58] 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*, pages 666–679, 2019. 1, 2.2, 4.1.2, 5, 5.1.1, 5.2, 5.6.1, 7.2

[59] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *Proceedings of the VLDB Endowment*, 4(6), 2011. 5.1.1, 6.1

[60] B.K. Debnath, D.J. Lilja, and M.F. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In *ICDEW*, pages 11–18, 2008. 1

[61] Kyle Deeds, Brian Hentschel, and Stratos Idreos. Stacked filters: learning to filter by

structure. *Proceedings of the VLDB Endowment*, 14(4):600–612, 2020. 6.5

[62] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. CIDR, 2005. 1, 6.1

[63] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013. 3.6.11

[64] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013. 3.6.11, 4.7, 5.6

[65] 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*, pages 1241–1258. ACM, 2019. 2.2

[66] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020. 6.5

[67] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: a learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment*, 14(2):74–86, 2020. 6.5

[68] 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*, pages 418–431, 2021. 2.2

[69] Yadolah Dodge and Daniel Commenges. *The Oxford dictionary of statistical terms*. Oxford University Press on Demand, 2006. 4.3.2, 4.4, 4.7.5

[70] Carlton Downey, Ahmed Hefny, Byron Boots, Geoffrey J Gordon, and Boyue Li. Predictive state recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 6055–6066, 2017. 3.6.2

[71] Naiqiao Du, Xiaojun Ye, and Jianmin Wang. Towards workflow-driven database system workload modeling. In *Proceedings of the Second International Workshop on Testing Database Systems*, page 10. ACM, 2009. 6.2

[72] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009. 2.2, 6.1, 6.1

[73] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 337–348. ACM, 2011. 4, 4.1.1, 6.1, 6.3

[74] Jennie Duggan, Yun Chi, Hakan Hacigumus, Shenghuo Zhu, and Ugur Cetintemel. Pack-

ing light: Portable workload performance prediction for the cloud. In *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*, pages 258–265. IEEE, 2013. 4, 4.1.1, 6.3

[75] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment*, 12(9):1044–1057, 2019. 4.2, 6.5

[76] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016. 1

[77] Said S Elnaffar and Pat Martin. An intelligent framework for predicting shifts in the workloads of autonomic database management systems. 3

[78] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996. 3.4.2

[79] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, volume 98, pages 323–333, 1998. 3.4.2

[80] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13 (8):1162–1175, 2020. 6.5

[81] Fakultit F/jr Informatik, Y Bengio, Paolo Frasconi, and Jfirgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. 03 2003. 3.5.1

[82] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001. 4.5.4

[83] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *International Conference on Data Engineering*, pages 592–603. IEEE, 2009. 3.4.1, 4, 6.1, 6.3

[84] Antara Ghosh, Jignashu Parikh, Vibhuti S Sengar, and Jayant R Haritsa. Plan selection based on query clustering. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 179–190. VLDB Endowment, 2002. 3.4.1

[85] Leilani H Gilpin, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, pages 80–89. IEEE, 2018. 4

[86] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: a practical, scalable solution. *ACM SIGMOD Record*, 30(2):331–342, 2001. 6.1

[87] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management (CNSM)*, pages 9–16. Ieee, 2010. 6.1, 6.2

[88] Le Gruenwald and Margaret H Eich. Selecting a database partitioning technique. *Journal*

of Database Management (JDM), 4(3):27–39, 1993. 6.1

[89] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. opengauss: An autonomous database system. *Proceedings of the VLDB Endowment*, 14(12):3028–3041, 2021. 6.1, 6.4, 6.5

[90] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. Pqr: Predicting query execution times for autonomous workload management. In *International Conference on Autonomic Computing*, pages 13–22. IEEE, 2008. 6.3

[91] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *International Conference on Database Theory*, pages 453–470. Springer, 1999. 6.1

[92] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for olap. ICDE, pages 208–219, 1997. ISBN 0-8186-7807-0. 1

[93] Michael Hammer and Arvola Chan. Index selection in a self-adaptive data base management system. In *SIGMOD*, pages 1–8, 1976. 2.1

[94] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. An autonomous materialized view management system with deep reinforcement learning. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2159–2164. IEEE, 2021. 6.1, 6.1

[95] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a database system*. Now Publishers Inc, 2007. 2.3

[96] Florian Herzog. *Strategic portfolio management for long-term investments: An optimal control approach*. ETH Zurich, 2005. 5.1.2

[97] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: learn from data, not from queries! *Proceedings of the VLDB Endowment*, 13(7):992–1005, 2020. 4.7.5, 6.5

[98] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 3.5.1

[99] Marc Holze and Norbert Ritter. Towards workload shift detection and prediction for autonomic databases. In *Proceedings of the ACM first Ph. D. workshop in CIKM*, pages 109–116. ACM, 2007. 3

[100] Marc Holze and Norbert Ritter. Autonomic databases: Detection of workload shifts with n-gram-models. In *East European Conference on Advances in Databases and Information Systems*, pages 127–142. Springer, 2008. 3, 6.2

[101] Marc Holze, Claas Gaidies, and Norbert Ritter. Consistent on-line classification of dbs workload events. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1641–1644. ACM, 2009. 3, 6.2

[102] Marc Holze, Ali Haschimi, and Norbert Ritter. Towards workload-aware self-management: Predicting significant workload shifts. In *26th International Conference on Data Engineering Workshops (ICDEW)*, pages 111–116. IEEE, 2010. 3, 6.2

[103] Boming Huang, Yuxiang Huan, Li Da Xu, Lirong Zheng, and Zhuo Zou. Automated

trading systems statistical and machine learning methods and hardware implementation: a survey. *Enterprise Information Systems*, 13(1):132–144, 2019. 2.3

[104] Peter J Huber. *Robust statistics*, volume 523. John Wiley & Sons, 2004. 4.5.2, 4.5.4

[105] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1): 155–162, 2012. 3.6.2

[106] Jie Jao. Qppnet in pytorch. https://github.com/rabbit721/QPPNet. 4.7.3

[107] TS Jayram, Phokion G Kolaitis, and Erik Vee. The containment problem for real conjunctive queries with inequalities. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 80–89. ACM, 2006. 3.3

[108] Anant Jhingran. Moving up the food chain: Supporting e-commerce applications on databases. *ACM SIGMOD Record*, 29(4):50–54, 2000. 7.4

[109] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment*, 11(7): 800–812, 2018. 6.1

[110] Tarun Kathuria and S Sudarshan. Efficient and provable multi-query optimization. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 53–67, 2017. 6.1

[111] Jeffrey O. Kephart. Research challenges of autonomic computing. ICSE '05, pages 15–22, 2005. 6.1

[112] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, SA, January 13-16, 2019, Online Proceedings*, 2019. URL http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf. 4.2, 6.5

[113] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–5, 2020. 6.5

[114] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006. 5.3.1, 5.3.2

[115] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995. 4.5.4

[116] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. In *ICDE*, pages 197–208, 2018. 4.3.2

[117] Jan Kossmann. Self-driving: From general purpose to specialized dbmss. In *Proceedings of the VLDB 2018 PhD Workshop*, 2018. 2.1

[118] 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. *Proceedings of the VLDB Endowment*, 13(12):2382–2395, 2020. 5, 6.1

[119] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018. 6.5

[120] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019. 6.5

[121] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018. 6.5

[122] Max Kuhn, Kjell Johnson, et al. *Applied predictive modeling*, volume 26. Springer, 2013. 4.1.2

[123] Sushil Kumar. Oracle Database 10g: The self-managing database, November 2003. White Paper. 1, 2.1, 2.2, 6.1

[124] Ludmila I Kuncheva and Christopher J Whitaker. Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Machine learning*, 51(2):181–207, 2003. 3.6.2

[125] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. Automatic configuration for IBM DB2 universal database. Technical report, IBM, jan 2002. 2.2, 2.2, 6.1, 6.1

[126] Wilburt Juan Labio, Dallan Quass, and Brad Adelberg. Physical database design for data warehouses. In *Proceedings 13th International Conference on Data Engineering*, pages 277–288. IEEE, 1997. 1

[127] Hai Lan, Zhifeng Bao, and Yuwei Peng. An index advisor using deep reinforcement learning. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2105–2108, 2020. 6.1

[128] Giovanni Lanfranchi, Pietro Della Peruta, Antonio Perrone, and Diego Calvanese. Toward a new landscape of systems management in an autonomic computing environment. *IBM Systems journal*, 42(1):119–128, 2003. 1

[129] Wolfgang Lehner, Bobbie Cochrane, H Pirahesh, and Markos Zaharioudakis. Applying mass query optimization to speed up automatic summary table refresh. In *Proceedings of the International Conference on Data Engineering*, 2001. 6.1

[130] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015. 4.2

[131] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. volume 12, pages 2118–2130. VLDB Endowment, 2019. 2.2, 6.1, 6.1

[132] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12): 2118–2130, 2019. 1, 5, 5.1.1

[133] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *Proceedings of the VLDB Endowment*, 5(11):1555–1566, 2012. 4, 4.1.1, 4.1.2, 4.1.2, 4.3.3, 4.5.2, 4.7, 4.7.3, 6.1, 6.3

[134] Xi Liang, Aaron J Elmore, and Sanjay Krishnan. Opportunistic view materialization with deep reinforcement learning. *arXiv preprint arXiv:1903.01363*, 2019. 6.1, 6.1

[135] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002. 4.5.4

[136] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018. 6.1

[137] 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 ACM International Conference on Management of Data*, SIGMOD '18, 2018. 2.3, 4, 4.2, 4.5.3, 4.7.4, 5, 5.3.3, 5.6.1, 5.6.1

[138] 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*, pages 175–191, 2020. 4.1.2

[139] 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*, pages 1248–1261, 2021. 5.2

[140] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. pages 3:1–3:4, 2018. URL http://doi.acm.org/10.1145/3211954.3211957. 6.5

[141] Ryan Marcus and Olga Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proceedings of the VLDB Endowment*, 12(11):1733–1746, 2019. 4, 4.1.1, 4.1.2, 4.1.2, 4.5.2, 4.7, 4.7.3, 6.3

[142] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11), 2019. 6.5

[143] 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*, pages 1275–1288, 2021. 6.5

[144] Patrick Martin, Said Elnaffar, and Ted Wasserman. Workload models for autonomic database management systems. In *International Conference on Autonomic and Autonomous Systems*, pages 10–10. IEEE, 2006. 3

[145] Jacob Mattingley, Yang Wang, and Stephen Boyd. Receding horizon control. *IEEE Control Systems Magazine*, 31(3):52–65, 2011. 5, 5.1.2

[146] Prashanth Menon and Andrew Pavlo Amadou Ngom, Todd C. Mowry. Permutable com-

piled queries: Dynamically adapting compiled queries without recompiling. *Under Submission*, 2020. 4.3.2

[147] Kristi Morton, Magdalena Balazinska, Dan Grossman, Robert Kosara, and Jock Mackinlay. Public data and visualizations: How are many eyes and tableau public used for collaborative analytics? *ACM SIGMOD Record*, 43(2):17–22, 2014. 7.4

[148] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the 2013 International Conference on Management of data*, pages 301–312. ACM, 2013. 3, 3.4.1, 4, 4.1.1, 4.3.3, 4.5.2, 6.1, 6.2, 6.3

[149] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012. 4.5.4

[150] Vivek Narasayya and Manoj Syamala. Workload driven index defragmentation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 497–508. IEEE, 2010. 6.3

[151] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. Continuous resource monitoring for self-predicting dbms. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 239–248. IEEE, 2005. 1, 3, 4, 4.1.1, 4.3.3, 4.5.2, 6.1, 6.2

[152] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011. 4.7.3

[153] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecommunication application transaction processing (tatp) benchmark description. http://tatpbenchmark.sourceforge.net/, July 2011. 4.7

[154] David W Opitz and Richard Maclin. Popular ensemble methods: An empirical study. 1999. 3.5.1

[155] Oracle. Oracle autonomous database. https://www.oracle.com/autonomous-database/, 2020. 1

[156] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55, 2016. 1, 2.3, 4

[157] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. Plp: page latch-free shared-everything oltp. *Proceedings of the VLDB Endowment*, 4(10):610–621, 2011. 6.1

[158] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. Quicksel: Quick selectivity learning with mixture models. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1017–1033, 2020. 6.5

[159] Andrew Pavlo, Evan P.C. Jones, and Stan Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *Proc. VLDB Endow.*, 5:85–96, October 2011. 6.2

[160] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Toma-

sic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017. 2.1, 3, 3.2, 3.3, 3.5, 4.4, 4.7.7

[161] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. External vs. internal: an essay on machine learning agents for autonomous database management systems. *IEEE bulletin*, 42 (2), 2019. 4.7.3

[162] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901. 3.6.9

[163] R Malinga Perera, Bastian Oetomo, Benjamin IP Rubinstein, and Renata Borovica-Gajic. Dba bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 600–611. IEEE, 2021. 5

[164] Gregory Piatetsky-Shapiro. The optimal selection of secondary indices is np-complete. *ACM SIGMOD Record*, 13(2):72–75, 1983. 1

[165] Robi Polikar. Ensemble based systems in decision making. *IEEE Circuits and systems magazine*, 6(3):21–45. 3.5.1

[166] S Joe Qin and Thomas A Badgwell. A survey of industrial model predictive control technology. *Control engineering practice*, 11(7):733–764, 2003. 5.1.2

[167] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. SIGMOD'02, pages 558–569. 1, 6.1, 6.1

[168] Qing Rao and Jelena Frtunikj. Deep learning for self-driving cars: Chances and challenges. In *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*, pages 35–38, 2018. 2.3

[169] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, volume 11, page 269. NIH Public Access, 2017. 4.1.2

[170] Unisphere Research. The real world of the database administrator, 2015. 1

[171] Jacques Richalet, A Rault, JL Testud, and J Papon. Model predictive heuristic control: Applications to industrial processes. *Automatica*, 14(5):413–428, 1978. 6.1

[172] Jennie Rogers, Olga Papaemmanouil, and Ugur Cetintemel. A generic auto-provisioning framework for cloud databases. In *International Conference on Data Engineering Workshops (ICDEW)*, pages 63–68. IEEE, 2010. 3, 6.1

[173] Yuji Roh, Geon Heo, and Steven Euijong Whang. A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 2019. 2.3

[174] Alex Rosenberg. Improving query performance in data warehouses. *Business Intelligence Journal*, 11, January 2006. 1

[175] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 4.5.4

[176] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *International Conference on Cloud Computing*, pages 500–507. IEEE, 2011. 3.6.2, 6.1, 6.1, 6.2

[177] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 249–260, 2000. 6.1

[178] Jonathan Rubin and Ian Watson. Computer poker: A review. *Artificial intelligence*, 175 (5-6):958–987, 2011. 5.1.3

[179] Zahra Sadri, Le Gruenwald, and Eleazar Leal. Online index selection using deep reinforcement learning for a cluster database. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pages 158–161. IEEE, 2020. 6.1

[180] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)*, 27(4), 1980. 3.3

[181] S Salza and R Tomasso. A modelling tool for the performance analysis of relational database applications. In *Proc. 6th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 323–337, 1992. 3, 6.2

[182] Silvio Salza and Mario Terranova. Workload modeling for relational database systems. In *Database Machines*, pages 233–255. Springer, 1985. 3, 6.2

[183] Rainer Schlosser, Jan Kossmann, and Martin Boissier. Efficient scalable multi-attribute index selection using recursive strategies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1238–1249. IEEE, 2019. 5.1.1

[184] Thomas Schmied, Diego Didona, Andreas Döring, Thomas Parnell, and Nikolas Ioannou. Towards a general framework for ml-based self-tuning databases. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 24–30, 2021. 6.1

[185] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. On-line index selection for shifting workloads. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 459–468. IEEE, 2007. 5, 6.4

[186] George AF Seber and Alan J Lee. *Linear regression analysis*, volume 329. John Wiley & Sons, 2012. 4.5.4

[187] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016. 6.1

[188] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643*, 2018. 6.1

[189] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. Astrid: accurate selectivity estimation for string predicates using deep learning. *Proceedings of the VLDB Endowment*, 14(4):471–484, 2020. 6.5

[190] Yasin N Silva, Paul-Ake Larson, and Jingren Zhou. Exploiting common subexpressions for cloud query processing. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1337–1348. IEEE, 2012. 6.1

[191] Moisés Silva-Muñoz, Alberto Franzin, and Hugues Bersini. Automatic configuration of the cassandra database using irace. *PeerJ Computer Science*, 7:e634, 2021. 6.1

[192] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. 2.3, 5, 5.1.3, 5.3.1, 7.1

[193] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. 5, 5.1.3, 5.3.1, 6.1, 7.1

[194] Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004. 4.5.4

[195] Binbin Song, Yao Yu, Yu Zhou, Ziqiang Wang, and Sidan Du. Host load prediction with long short-term memory in cloud computing. *The Journal of Supercomputing*, pages 1–15, 2017. 3.5.1

[196] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, pages 953–966, 2008. 6.1, 6.1

[197] Bureau Of Labor Statistics. Database administrators. https://www.bls.gov/ooh/computer-and-information-technology/database-administrators.htm, 2019. 1

[198] Stephen M Stigler. The asymptotic distribution of the trimmed mean. *The Annals of Statistics*, pages 472–477, 1973. 4.5.2

[199] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's Learning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 19–28, 2001. 6.1

[200] Michael Stonebraker and Joey Hellerstein. What goes around comes around. *Readings in Database Systems*, 4, 2005. 1

[201] Adam J. Storm, Christian Garcia-Arellano, and *et al.* Adaptive self-tuning memory in DB2. VLDB, pages 1081–1092, 2006. 1, 2.2, 2.2, 6.1

[202] David G. Sullivan and *et al.* Using probabilistic reasoning to automate software tuning. SIGMETRICS, pages 404–405, 2004. 1

[203] Ji Sun and Guoliang Li. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment*, 13(3):307–319, 2019. 4, 6.5

[204] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018. 6.1

[205] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael

Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data*, pages 205–219, 2018. 2.3, 5, 5.3.3, 5.6.1

[206] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proceedings of the VLDB Endowment*, 12(10):1221–1234, 2019. 1, 6.1

[207] Zaiyong Tang and Paul A Fishwick. Feedforward neural nets as models for time series forecasting. *ORSA journal on computing*, 5(4):374–385, 1993. 3.6.2

[208] The Transaction Processing Council. TPC-C Benchmark (Revision 5.11.0). http://www.tpc.org/tpcc/, February 2010. 4.7, 5.6

[209] The Transaction Processing Council. TPC-H Benchmark (Revision 2.16.0). http://www.tpc.org/tpch/, June 2013. 4.7

[210] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern recognition*. Elsevier, 2003. 4.1.2

[211] Wenhu Tian, Pat Martin, and Wendy Powley. Techniques for automatically sizing multiple buffer pools in DB2. CASCON, pages 294–302, 2003. 1, 6.1

[212] Boris A Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70:569–572, 1950. 3.3

[213] Khai Q Tran, Jeffrey F Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. Jecb: A join-extension, code-based approach to oltp data partitioning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 39–50, 2014. 6.1

[214] G. Valentin, M. Zuliani, and *et al.* DB2 advisor: an optimizer smart enough to recommend its own indexes. ICDE, pages 101–110, 2000. 1, 5.1.1, 6.1

[215] 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*, pages 1009–1024, 2017. 1, 5, 5.1.1, 5.2, 7.3

[216] 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. 2.2, 4.1.2, 6.1

[217] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment*, 14(7):1241–1253, 2021. 6.1, 7.3

[218] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Manage-*

ment of Data, pages 1041–1052, 2017. 2.2

[219] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985. 3.3

[220] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. Demonstrating udo: A unified approach for optimizing transaction code, physical design, and system parameters via reinforcement learning. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2794–2797, 2021. 5, 6.4

[221] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, 45(2):17–22, September 2016. URL http://doi.acm.org/10.1145/3003665.3003669. 6.5

[222] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 20–31. Elsevier, 2002. 6.1, 6.1

[223] Kyu-Young Whang. Index selection in relational databases. In *Foundations of Data Organization*, pages 487–500. Springer, 1987. 6.1

[224] David Wiese, Gennadi Rabinovitch, Michael Reichert, and Stephan Arenswald. Autonomic tuning expert: A framework for best-practice oriented autonomic database tuning. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 3:27–3:41, 2008. 2.2, 6.1, 6.1

[225] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989. 3.5.1

[226] Andrew Witkowski, Rafi Ahmed, and Murali Thiyagarajan. Oracle autonomous db: Challenges and some solutions. 2nd International Workshop on Applied AI for Database Systems and Applications, 2020. URL https://sites.google.com/view/aidb2020/home/invited-talks#h.pyh4j69fn27v. 1

[227] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–8, 2019. 4.2, 6.5

[228] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment*, 12(3):210–222, 2018. 4.2, 6.5

[229] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment*, 14(8): 1276–1288, 2021. 6.5

[230] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment*, 6(10):925–936, 2013. 4, 4.1.1, 4.5.2, 6.3

[231] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F

Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1081–1092. IEEE, 2013. 4.7.3, 6.3

[232] Wentao Wu, Xi Wu, Hakan Hacigümüş, and Jeffrey F Naughton. Uncertainty aware query execution time prediction. *Proceedings of the VLDB Endowment*, 7(14):1857–1868, 2014. 6.3

[233] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017. 4.2

[234] Khaled Yagoub, Pete Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. Oracle's sql performance analyzer. *IEEE Data Engineering Bulletin*, 31(1), 2008. 2.2, 6.1

[235] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment*, 13(3):279–292, 2019. 4.2, 4.7.5, 6.5

[236] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 193–208, 2020. 2.2

[237] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment*, 14(1):61–73, 2020. 6.5

[238] Yiwen Zhu, Matteo Interlandi, Abhishek Roy, Krishnadhan Das, Hiren Patel, Malay Bag, Hitesh Sharma, and Alekh Jindal. Phoebe: A learning-based checkpoint optimizer. *Proceedings of the VLDB Endowment*, 14(11):2505–2518, 2021. 6.5

[239] Dong Young Yoon, Ning Niu, and Barzan Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1599–1614. ACM, 2016. 4, 4.1.1, 6.1, 6.2

[240] Philip S. Yu, M-S Chen, H-U Heiss, and Sukho Lee. On workload characterization of relational database environments. *IEEE Transactions on Software Engineering*, 18(4):347–355, 1992. 6.2

[241] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1297–1308. IEEE, 2020. 6.5

[242] 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. IEEE, 2020. 6.1, 6.1

[243] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 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, pages 415–432, 2019. 2.2, 6.1, 6.1

[244] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432, 2019. 1, 5, 5.1.1

[245] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2102–2114, 2021. 6.1

[246] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 533–544, 2007. 6.1

[247] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. Sia: Optimizing queries using learned predicates. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2169–2181, 2021. 6.5

[248] 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. 6.5

[249] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. Query performance prediction for concurrent queries using graph embedding. *Proceedings of the VLDB Endowment*, 13(9): 1416–1428, 2020. 4, 4.1.1, 6.3

[250] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms.* CRC press, 2012. 3.5.1

[251] Daniel C Zilio. *Physical database design decision algorithms and concurrent reorganization for parallel database systems.* Citeseer, 1999. 6.1

[252] Daniel C Zilio, Anant Jhingran, and Sriram Padmanabhan. *Partitioning key selection for a shared-nothing parallel database system.* IBM TJ Watson Research Center, 1994. 6.1

[253] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004. ISBN 0-12-088469-0. 6.1, 6.1, 6.1

[254] Daniel C Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M Lohman, Roberta J Cochrane, Hamid Pirahesh, Latha Colby, Jarek Gryz, Eric Alton, et al. Recommending materialized views and indexes with the ibm db2 design advisor. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 180–187. IEEE, 2004. 6.1