MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems

Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich
Wan Shen Lim, Prashanth Menon, Andrew Pavlo
Carnegie Mellon University
{lin.ma,mbutrovi,wanshenl,pmenon,pavlo}@cs.cmu.edu,{wz2,jiejiao,wuwenw}@andrew.cmu.edu

ABSTRACT

Database management systems (DBMSs) are notoriously difficult to deploy and administer. The goal of a self-driving DBMS is to remove these impediments by managing itself automatically. However, a critical problem in achieving full autonomy is how to predict the DBMS's runtime behavior and resource consumption. These predictions guide a self-driving DBMS's decision-making components to tune and optimize all aspects of the system.

We present the ModelBot2 end-to-end framework for constructing and maintaining prediction models using machine learning (ML) in self-driving DBMSs. Our approach decomposes a DBMS's architecture into fine-grained operating units that make it easier to estimate the system's behavior for configurations that it has never seen before. ModelBot2 then provides an offline execution environment to exercise the system to produce the training data used to train its models. We integrated ModelBot2 in an in-memory DBMS and measured its ability to predict its performance for OLTP and OLAP workloads running in dynamic environments. We also compare ModelBot2 against state-of-the-art ML models and show that our models are up to 25× more accurate in multiple scenarios.

ACM Reference Format:
Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich and Wan Shen Lim, Prashanth Menon, Andrew Pavlo . 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457276

1 INTRODUCTION

A self-driving DBMS can configure, tune, and optimize itself without human intervention, even as the application's workload, dataset, and operating environment evolve [50]. Such automation seeks to remove the complications and costs involved with DBMS deployments. The core component that underlies a self-driving DBMS's decision-making is *behavior models* [51]. 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 [49]. 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 [42, 45, 74]. 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 [9, 17, 20, 34, 40]. The models that support concurrent queries focus on real-time settings where the interleaving of queries is known [16, 68, 72], but a self-driving DBMS needs to plan for future workloads without such accurate information [37]. Many ML-based models also rely on dataset or workload-dependent information [16, 40, 58]; 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 [22]. 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 [37]) 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

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457276

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 [1] 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 [40]. Our results show that our models support OLTP and OLAP workloads with a minimal loss of accuracy.

2 BACKGROUND AND MOTIVATION

A self-driving DBMS's architecture shares similar inspirations from self-driving cars. The commonly adopted self-driving car architecture (simplified for demonstration) consists of (1) a perception system, (2) mobility models, and (3) a decision-making system [49]. The *perception system* observes the 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.

The above components have analogs in a self-driving DBMS: (1) forecasting, (2) behavior models, and (3) planning system [51]. The *forecasting system* is how the DBMS observes and predicts the application's future workload [37]. The DBMS then uses these forecasts with its *behavior models* to predict its runtime behavior relative to the target objective function (e.g., latency, throughput). The DBMS's *planning system* selects actions that improve this objective function.

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.

2.1 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 [42, 45, 74]. 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 [42, 45]. 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 [17, 34, 40]. 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).

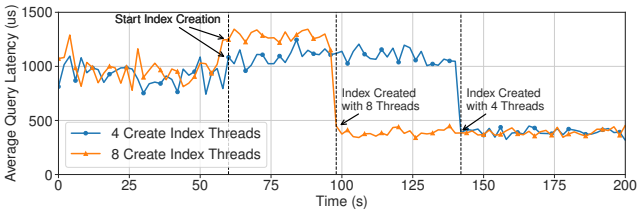


Figure 1: Index Build Example – TPC-C query latency running on NoisePage. The DBMS begins index building after 60s using 4 or 8 threads.

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 ML models [16, 68, 72]. 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 Sec. 8. The results in Fig. 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, Fig. 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.

2.2 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 [62]. Even if the model only targets query execution, a modern DBMS will still have hundreds of plan operator features [9, 40]. 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 [34, 40]. 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.

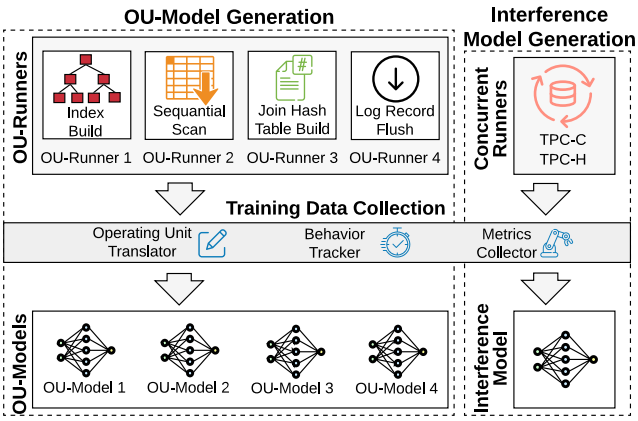


Figure 2: System Architecture – MB2 trains OU-models for each OU with the data generated by OU-runners and builds interference models for concurrent OUs based on system resource utilization.

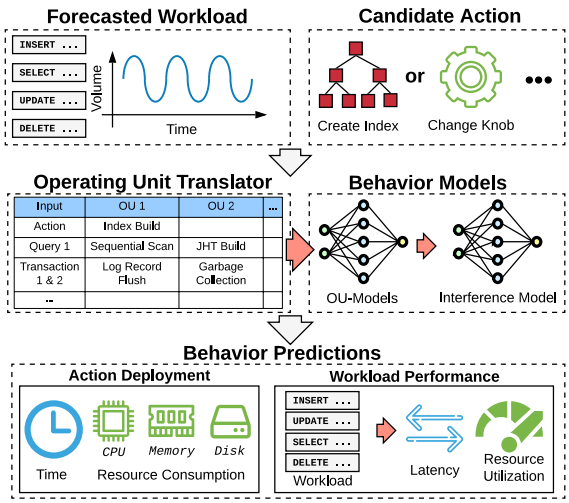


Figure 3: MB2 Inference Procedure – Given the forecasted workload and a self-driving action, MB2 records the OUs for all input tasks, and uses the OU-models and interference models to predict the DBMS's behavior.

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 [52]. For example, building indexes can take hours [12], 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 [34, 38, 40, 63]. 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 [30].

3 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.

Fig. 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.

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. Fig. 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 Sec. 2.1, 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.

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) [37]. 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 [71] 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 [32]. 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 [18, 27, 66, 67, 73], which MB2 may leverage.

Lastly, while MB2 supports hardware context in its models (see Sec. 4.2), 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 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.

4.1 Principles

Developers use MB2 to decompose the DBMS into OUs to build explainable, adaptable, and workload independent behavior models.

	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 1: Operating Unit – Property Summary of OUs in NoisePage.

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 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.

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.

4.2 Input Features

After deciding which OUs to implement, DBMS developers then specify the OU-models' input features based on their *degree of freedom* [14]. These features may contain (1) the amount of work for a single OU invocation or multiple OU invocations in a batch (e.g.,

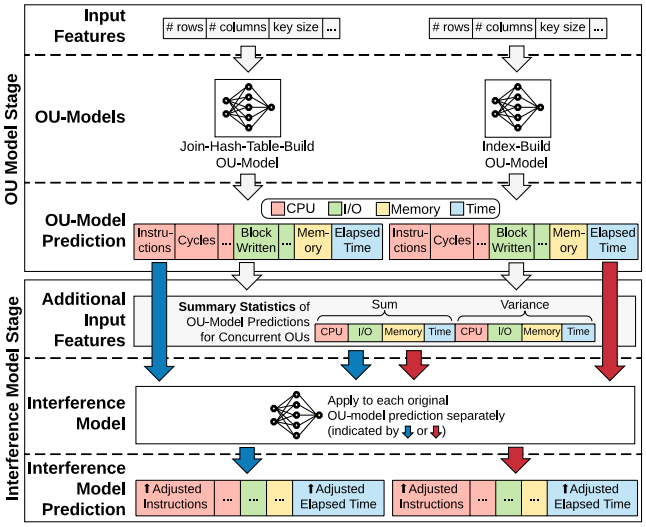


Figure 4: OU and Interference Models – MB2 uses OU-specific input features to predict their resource consumption and elapsed time. The interference model uses the summary predictions for concurrent OUs.

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 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 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 [29, 41]. 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 Sec. 5.

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 (Sec. 8.6). We leave a thorough investigation for the proper hardware context features as future work.

4.3 Output Labels

Every OU-model produces the same output labels. They are a vector of commonly available hardware metrics (Sec. 6.1) 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 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.

[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.

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 [34]. 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 $O(n)$. Likewise, the time and resources required to build buffers for sorting are $O(n \log n)$. 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 [42, 45], 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 (Sec. 7). We demonstrate in Sec. 8 that MB2 generalizes to datasets with orders of magnitude higher number of tuples than what exists in the training data using this approach.

5 INTERFERENCE MODEL

The OU-models capture the internal contention on data structures or latches within an OU (Sec. 4.2). 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 interference since it may impact any OU.

As shown in Fig. 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 [50], but predicting queries' exact future arrival times and concurrent interleavings is arguably impossible.

[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.

To address these challenges, we formulate a model based on *summary statistics* [14] 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. Fig. 4 illustrates the formulation of the resource competition interference model.

5.1 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.

5.2 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.

6 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 online system without incurring observable performance degradation as future work.

6.1 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 [2, 6].

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.

6.2 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 [11]. 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 (Sec. 4.3) 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 [34, 40, 68] or assumes a fixed set of queries/stored procedures in the workload [42, 45]. 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×) and applies *robust statistics* [25] 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 [57], 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.

6.3 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 (Sec. 5).

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 [37]. As discussed in Sec. 5.2, 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.

6.4 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.

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 2: MB2 Overhead – Behavior Model Computation and Storage Cost

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 [54], (2) Huber regression [25], (3) support vector machine [56], (4) kernel regression [43], (5) random forest [35], (6) gradient boosting machine [19], and (7) deep neural network [53]. 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 [28]. 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.

7 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 reruns the concurrent runners to generate the interference models. In NoisePage, we currently use a heuristic for MB2 to retrain the interference models when a DBMS update affects at least five OUs.

8 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 [13] testbed as an end-to-end workload generator for the SmallBank [10], TATP [47], TPC-C [60], and TPC-H [61] 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 [34, 40]. 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` [7]. 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 [3] with `libpqxx` [4] to connect to NoisePage.

8.1 Data Collection and Training

We first discuss MB2's data collection and model training. The results in Table 2 show a breakdown between the time that MB2 spends generating data versus the time that it spends training its models. We generated ~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 2 also shows that the concurrent runners produce a larger data set than the OU-runners because they execute multiple OUs at 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.

8.2 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 Sec. 6.4 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.

Fig. 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 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.

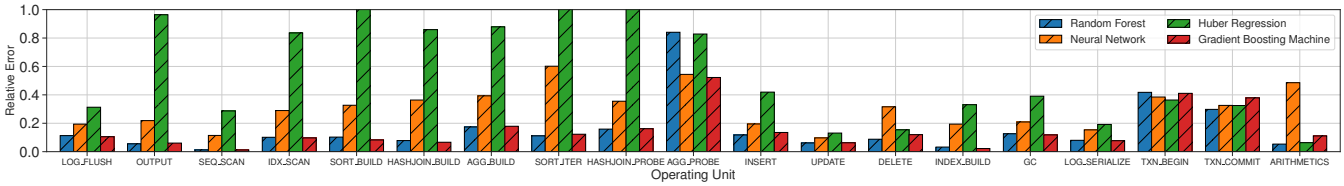


Figure 5: OU-model Accuracy (OU) – Test relative error for each OU averaging across all OU-model output labels. OU-models are trained with four ML algorithms: (1) random forest, (2) neural network, (3) Huber regression, and (4) gradient boosting machine.

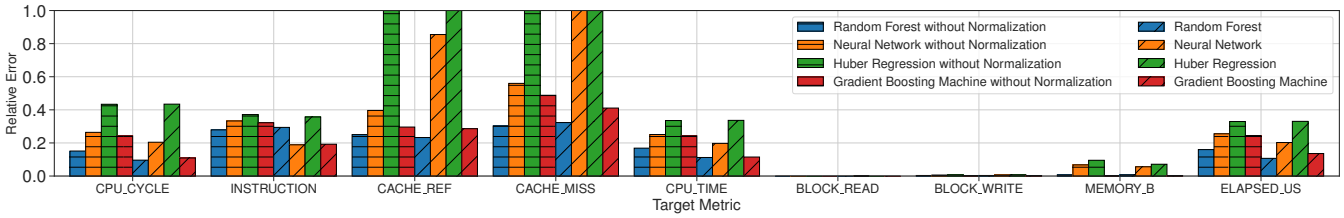
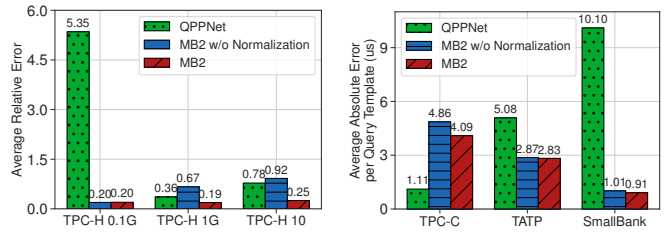


Figure 6: OU-model Accuracy (Output Labels) – Test relative error for each output label averaging across all OUs. OU-models are trained with four ML algorithms with and without output label normalization: (1) random forest, (2) neural network, (3) Huber regression, and (4) gradient boosting machine.



(a) OLAP Query Runtime Prediction **(b)** OLTP Query Runtime Prediction

Figure 7: OU-Model Generalization – Query runtime estimations on different datasets and workloads.

In Fig. 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 Sec. 8.4) because the interference model extracts information from all the output labels. The results in Fig. 6 also show the OU-model errors without output label normalization optimization from Sec. 4.3. From this, we see that normalization has minimal impact on OU-model accuracy while enabling generalizability.

8.3 OU-Model Generalization

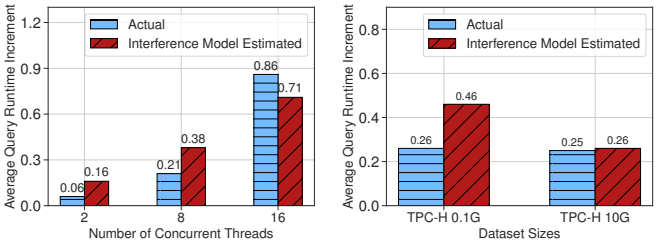
We now evaluate the OU-models' ability to predict query runtime and generalize across workloads. Accurate query runtime prediction is crucial since many self-driving DBMSs' optimization objectives are focused on reducing query latency [51]. As discussed in Sec. 3, MB2 extracts all OUs from a query plan 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 [26, 40]. QPPNet uses a tree-structured neural network to capture a query plan's structural information. It outperforms other recent models on predicting query runtime [9, 34, 69], 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 [46], 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 (Sec. 8.2) for all the workloads.

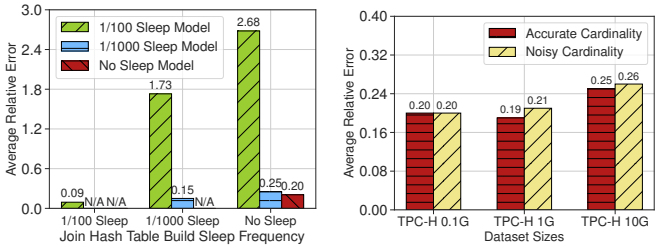
The results in Fig. 7a show that QPPNet achieves competitive prediction accuracy on the 1 GB TPC-H workload since it is the same as its training dataset. But QPPNet has larger errors for TPC-H on other scale factors. The authors of QPPNet also observed similar generalization challenges for their models despite it outperforming the baselines [40]. 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, Fig. 7b shows that while MB2 has 4× higher prediction error compared to QPPNet on TPC-C workload where QPPNet is



(a) Varying Concurrent Threads **(b)** Varying Dataset Sizes

Figure 8: Interference Model Accuracy – Model trained with 1 GB TPC-H dataset and odd thread numbers generalizes to other scenarios.



(a) DBMS Updates (TPC-H 1 G) **(b)** Noisy Cardinality Estimation

Figure 9: Model Adaptation and Robustness – Changes in MB2's model accuracy under DBMS updates and noisy cardinality estimation.

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.

8.4 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 [37]. 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 Sec. 4.2), but we test the model under JIT compilation mode. We also evaluate the model with thread numbers and workload 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.

Fig. 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 Fig. 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.

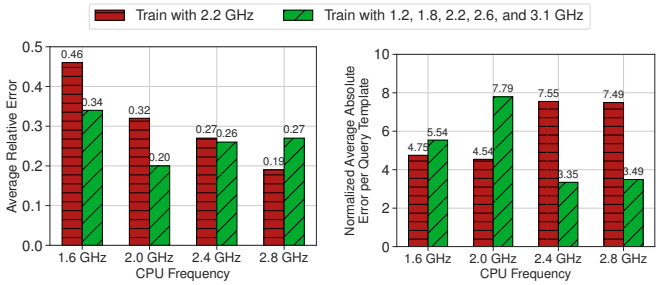
8.5 Model Adaptation and Robustness

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. Fig. 9a shows the prediction errors for the TPC-H 1 GB workload with the old and updated models. The new models only 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 [14] 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). Fig. 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 [24, 73].

8.6 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



(a) TPC-H 1 G Query Runtime Prediction **(b)** TPC-C Query Runtime Prediction

Figure 10: Hardware Context – Extend MB2's OU-models' input features to include the CPU frequency to generalize to different CPU frequencies.

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.

Fig. 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 case where including the hardware context performs notably worse is for the TPC-C workload under 2.0 GHz CPU frequency (Fig. 10b). This is because the models generally over-predict 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.

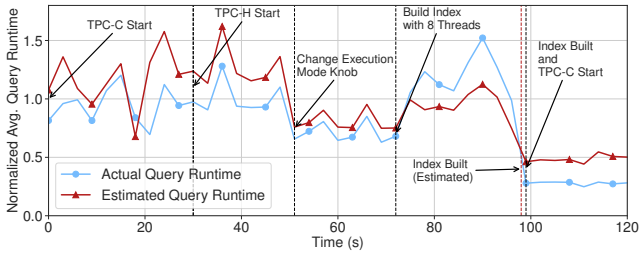
8.7 End-to-End Self-Driving Execution

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 Sec. 8.4, 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.

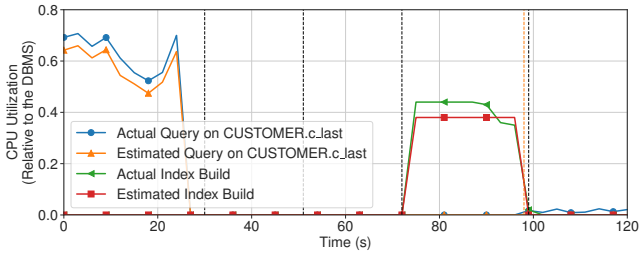
We simulate a daily transactional-analytical workload cycle by alternating the execution of TPC-C and TPC-H workloads on NoisePage. We use the TPC-C workload with 20 warehouses and adjust the number of customers per district to 50,000 to make the choice of indexes more important. For TPC-H, we use a 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, 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.

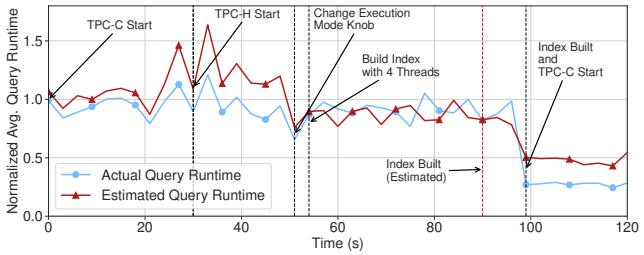
Fig. 11a shows the actual and estimated average query runtime per 3s interval across the entire evaluation session. We normalized the query runtime against the average runtime under the default configuration to keep metrics within the same scale. The workload starts as TPC-C and switches to TPC-H after 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) 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 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.

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 [50].

We next show how MB2's behavior models help explain the self-driving DBMS's decision. Fig. 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,

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

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 Fig. 11c. The DBMS plans the same actions under the same setup as in Fig. 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.

9 RELATED WORK

We broadly classify the previous work in DBMS modeling into the following categories: (1) ML models and (2) analytical models. Related to this are other methods for handling concurrent workloads. We also discuss other efforts on building automated DBMSs and reinforcement learning-based approaches.

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 [20] to Gupta et al.'s PQR hierarchical classification of queries into latency buckets [23]. Some methods mix plan- and operator- level models to balance between accuracy and generality, compensating further for generalizability issues by adjusting prebuilt offline models at runtime [9] or by building additional scaling functions [34]. QPPNet [40] 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 offline models that predict the resource bottleneck and maximum throughput of concurrent OLTP workloads, given fixed transaction types and fixed DBMS configuration and physical design [42]. Wu et al. tunes the optimizer's cost models for better query execution time predictions through offline profiling and online refinement of cardinality estimates [69, 70]. Narasayya and Syamala provides DBAs with index defragmentation suggestions by modeling workload I/O costs [44].

Concurrent Queries: Due to the difficulty of modeling concurrent operations as described in Sec. 2.2, 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 [8], query execution time estimation [16], and overall analytical workload throughput prediction [17]. Wu et al. support dynamic query sets by modeling CPU, I/O interactions, and buffer pool hit rates with queuing networks and Markov chains [68]. More recently, GPredictor predicts concurrent query performance by using a deep learning prediction network on a graph embedding of query features [72]. 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.

Autonomous DBMSs: There is a long history of research on automating DBMS deployments [51]. Weikum et al. provided a summary of existing self-tuning DBMS techniques in the early 2000s [65]. These typically involved queuing theory applied to models with different server types, such as applications, back-end databases, and middleware components [21]. Microsoft Azure SQL Database provides an auto-indexing [12] sub-system that automatically recommends, implements, and validates indexes. Oracle offers an autonomous DBaaS that runs their previous tuning tools in a managed environment [5], which requires expensive testing of candidate changes (e.g., indexes) on replicas without behavior models.

Another target area on using ML for automated DBMSs has been on knob configuration tuning. iTuned [15] and OtterTune [64] use Gaussian Process Regression to model how a DBMS will perform with different knob settings. CDBTune [76] and QTune [33] use actor-critic reinforcement learning to train knob recommendation policies. RelM tunes memory allocation knobs for analytical systems using Guided Bayesian Optimization [31].

Reinforcement Learning (RL) for DBMSs: Recent work explores using RL [55, 59] to enhance individual DBMS components [39, 48, 75], which are independent of MB2's goal to build behavior models for self-driving DBMSs that automate all the administrative tasks. We use a modularized design (Sec. 2) 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 [36].

10 CONCLUSION

Behavior modeling is the foundation for building a self-driving DBMS. We propose a modeling framework ModelBot2 (MB2) that decomposes the DBMS into operating units (OUs) and builds models for each OU with runners that exercise OUs' behavior. Our evaluation shows that MB2's behavior models provide the essential cost, impact, and benefit estimations for actions of self-driving DBMSs and generalize to various workloads and environments.

ACKNOWLEDGEMENTS

This work was supported (in part) by the National Science Foundation (IIS-1846158, SPX-1822933), Google Research Grants, and the Alfred P. Sloan Research Fellowship program. **TKBM**.

REFERENCES

[1] [n.d.]. NoisePage. https://noise.page.
[2] 2020. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc/.
[3] 2020. Google Benchmark Support Library. https://github.com/google/benchmark.
[4] 2020. libpqxx – Official C++ Client API for PostgreSQL. http://pqxx.org/development/libpqxx/.
[5] 2020. Oracle Self-Driving Database. https://www.oracle.com/database/autonomous-database/index.html.
[6] 2020. Processor Counter Monitor (PCM). https://github.com/opcm/pcm/.
[7] 2020. scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/.
[8] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. 2011. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Proceedings of the 14th International Conference on Extending Database Technology*. ACM, 449–460.
[9] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 390–401.
[10] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, 576–585.
[11] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research* 13, 1 (2012), 281–305.
[12] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1–14.
[13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.
[14] Yadolah Dodge and Daniel Commenges. 2006. *The Oxford dictionary of statistical terms*. Oxford University Press on Demand.
[15] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *VLDB* 2 (2009), 1246–1257. Issue 1.
[16] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 337–348.
[17] Jennie Duggan, Yun Chi, Hakan Hacigumus, Shenghuo Zhu, and Ugur Cetintemel. 2013. Packing light: Portable workload performance prediction for the cloud. In *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*. IEEE, 258–265.
[18] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.
[19] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.
[20] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 592–603.
[21] Michael Gillmann, Gerhard Weikum, and Wolfgang Wonner. 2002. Workflow management with service quality guarantees. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 228–239.
[22] Leilani H Gilpin, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. 2018. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*. IEEE, 80–89.
[23] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. 2008. PQR: Predicting query execution times for autonomous workload management. In *Autonomic Computing, 2008. ICAC'08. International Conference on*. IEEE, 13–22.
[24] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proceedings of the VLDB Endowment* 13, 7 (2020), 992–1005.
[25] Peter J Huber. 2004. *Robust statistics*. Vol. 523. John Wiley & Sons.
[26] Jie Jao. [n.d.]. QPPNet in PyTorch. https://github.com/rabbit721/QPPNet.
[27] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf
[28] Ron Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14. Montreal, Canada, 1137–1145.
[29] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*. 197–208.
[30] Max Kuhn, Kjell Johnson, et al. 2013. *Applied predictive modeling*. Vol. 26. Springer.
[31] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1667–1683.
[32] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
[33] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12, 2118–2130.
[34] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for sql queries using statistical techniques. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1555–1566.
[35] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.
[36] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. 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*. 751–766.
[37] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 ACM International Conference on Management of Data (SIGMOD '18)*.
[38] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active Learning for ML Enhanced Database Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 175–191.
[39] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019).
[40] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1733–1746.
[41] Prashanth Menon and Andrew Pavlo Amadou Ngom, Todd C. Mowry. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *Under Submission* (2020).
[42] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the 2013 International Conference on Management of data*. ACM, 301–312.
[43] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.
[44] Vivek Narasayya and Manoj Syamala. 2010. Workload driven index defragmentation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 497–508.
[45] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. 2005. Continuous resource monitoring for self-predicting DBMS. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 239–248.
[46] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.
[47] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. 2011. Telecommunication Application Transaction Processing (TATP) Benchmark Description. http://tatpbenchmark.sourceforge.net/.
[48] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. ACM, 4.
[49] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. 2016. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles* 1, 1 (2016), 33–55.
[50] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR 2017, Conference on Innovative Data Systems Research*.
[51] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *Data Engineering* (2019), 31.
[52] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid training data creation with weak supervision. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 11. NIH Public Access, 269.
[53] Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65, 6 (1958), 386.
[54] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 329. John Wiley & Sons.
[55] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton,

et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.
[56] Alex J Smola and Bernhard Schölkopf. 2004. A tutorial on support vector regression. *Statistics and computing* 14, 3 (2004), 199–222.
[57] Stephen M Stigler. 1973. The asymptotic distribution of the trimmed mean. *The Annals of Statistics* (1973), 472–477.
[58] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment* 13, 3 (2019), 307–319.
[59] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[60] The Transaction Processing Council. 2010. TPC-C Benchmark (Revision 5.11.0). http://www.tpc.org/tpcc/.
[61] The Transaction Processing Council. 2013. TPC-H Benchmark (Revision 2.16.0). http://www.tpc.org/tpch/.
[62] Sergios Theodoridis and Konstantinos Koutroumbas. 2003. *Pattern recognition*. Elsevier.
[63] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024.
[64] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024.
[65] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. 2002. 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*. Elsevier, 20–31.
[66] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–8.
[67] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.
[68] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. 2013. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment* 6, 10 (2013), 925–936.
[69] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 1081–1092.
[70] Wentao Wu, Xi Wu, Hakan Hacigümüş, and Jeffrey F Naughton. 2014. Uncertainty aware query execution time prediction. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1857–1868.
[71] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.
[72] Guoliang Li Jianhua Feng Xuanhe Zhou, Ji Sun. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. In *VLDB 2020*.
[73] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment* 13, 3 (2019), 279–292.
[74] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1599–1614.
[75] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1297–1308.
[76] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 415–432.