Database Gyms

Wan Shen Lim
Carnegie Mellon University
wanshenl@cs.cmu.edu

Matthew Butrovich
Carnegie Mellon University
mbutrovi@cs.cmu.edu

William Zhang
Carnegie Mellon University
wz2@cs.cmu.edu

Andrew Crotty
Northwestern University
andrew.crotty@northwestern.edu

Lin Ma
University of Michigan
linmacse@umich.edu

Peijing Xu
Carnegie Mellon University
peijingx@cs.cmu.edu

Johannes Gehrke
Microsoft Research
johannes@microsoft.com

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Abstract

In the past decade, academia and industry have embraced machine learning (ML) for database management system (DBMS) automation. These efforts have focused on designing ML models that predict DBMS behavior to support picking actions (e.g., building indexes) that improve the system's performance. Recent developments in ML have created automated methods for finding good models. Such advances shift the bottleneck from DBMS model design to obtaining the training data necessary for building these models. But generating good training data is challenging and requires encoding subject matter expertise into DBMS instrumentation.

Existing methods for training data collection are bespoke to individual DBMS components and do not account for (1) how workload trends affect the system and (2) the subtle interactions between internal system components. Consequently, the models created from this data do not support holistic tuning across subsystems and require frequent retraining to boost their accuracy.

This paper presents the architecture of a database gym, an integrated environment that provides a unified API of pluggable components for obtaining high-quality training data. The goal of a database gym is to simplify ML model training and evaluation to accelerate autonomous DBMS research. But unlike gyms in other domains that rely on custom simulators, a database gym uses the DBMS itself to create simulation environments for ML training. Thus, we discuss and prescribe methods for overcoming challenges in DBMS simulation, which include demanding requirements for performance, simulation fidelity, and DBMS-generated hints for guiding training processes.

1 INTRODUCTION

Despite major advancements over the last 50 years, database management systems (DBMSs) remain one of the most difficult software systems to configure and optimize correctly. As such, they have a long history of automated methods for configuring physical design, knob settings, and other setup aspects. There is now significant interest from both academia [40, 45, 48, 76] and industry [9, 19, 21, 54] in applying machine learning (ML) to solve these problems.

The ultimate goal of these ML methods is to create an autonomous DBMS that operates without human guidance (i.e., Level #5 – self-driving DBMS [60]). Such a system aims to optimize itself automatically for a given *objective function* (e.g., latency, throughput, cost) and constraints (e.g., cost budget, SLA) [58]. It improves this objective by deploying actions (e.g., building an index) with human-understandable explanations in response to anticipated workloads.

Most of the previous work on using ML for DBMS automation has focused on enhancing the models for these approaches. However, recent advances in ML have made these efforts moot. Foremost is the emergence of a small number of models (i.e., "foundation" models) that dominate all other models on a wide variety of tasks [7]. Furthermore, automated frameworks have greatly reduced the engineering burden of crafting a model to the point of requiring minimal effort or ML expertise [18, 20, 49, 83].

Given this, we contend that the current challenge facing the database community is obtaining high-quality training data for the models [64]. Other ML-heavy domains, such as robotics [37, 73, 86] and self-driving vehicles [12, 81], obtain training data via software-based simulators. A *simulator* attempts to approximate the behavior of an object when it would otherwise be too costly, time-consuming, or dangerous to experiment on the real system. Researchers typically package such simulators into a *gym*, a toolkit for developing and evaluating ML models and algorithms. For example, OpenAI Gym [14] is a commonly used platform for reinforcement learning research. The standardization of simulator environments in gyms has accelerated ML research by unifying efforts to build end-to-end pipelines for rapid model prototyping and productionization. Other research communities are developing similar gyms for their respective fields, including networking [30] and fintech [10].

Despite the clear benefits of gym environments for ML research, to the best of our knowledge, no one has attempted to build a gym to support autonomous DBMS efforts. One possible reason is that the effort required to build a DBMS simulator is tantamount to building the DBMS itself. Instead, researchers and practitioners have built custom training modules for individual DBMS components (e.g., optimizers [46, 47], executors [45]). This lack of standardization results in reimplementation of basic infrastructure, like workload

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

capture and analysis. Moreover, it complicates comparing the performance and generalizability of ML approaches.

To overcome these problems, we argue that the next chapter of self-driving DBMS research should focus on developing a **database gym**. The gym addresses the problem of obtaining training data to build models for autonomous DBMSs by tackling the ML and systems challenges together. Furthermore, the gym's ownership of the end-to-end pipeline enables unique opportunities for learning across different DBMSs and hardware configurations. The gym aims to expedite research in DBMS automation by providing a turnkey solution to the challenge of obtaining high-quality training data.

2 BACKGROUND

An autonomous DBMS makes decisions using three architectural ML components [60]: (1) *workload forecasting* to predict the future workload, (2) *behavior modeling* to predict the system's response to actions, and (3) *action planning* to decide which actions to apply and when. These ML components typically rely upon models trained on data collected from the *target* DBMS [15]. Such training data consists of inputs (i.e., features) and outputs (i.e., labels). A model fits the training data by learning associations between inputs to predict outputs. For example, a model can predict the benefit of building an index for the future workload.

Achieving the highest level of automated operation requires ML models that capture subtle interactions between different DBMS components. However, the ML models cannot account for these interactions without *high-quality training data* that either (1) represents the interactions as explicit input features or (2) manifests their effects as output labels. However, the former approach is intractable because there are too many candidate features, many of which are impractical to obtain [15]. The latter approach is more feasible, as it only involves collecting training data under similar conditions as the target DBMS. Therefore, we define high-quality training data as data collected using a workload and state that approximates the target DBMS. This approximation ensures that the DBMS generates training data that captures these subtleties without explicitly modeling all DBMS interactions.

We now describe the aspects of a DBMS simulator that are important for training data quality: (1) *workload* and (2) *state*. Related to these, we also discuss issues surrounding how the DBMS evaluates *actions* to generate diverse training data. This discussion motivates the design of a *database gym*, which we present in Section 3.

2.1 Workload

The first aspect of a DBMS that is relevant to training data collection is its workload (i.e., the queries it executes). Both commercial DBMSs (e.g., Oracle [28], Microsoft SQL Server [50]) and popular open-source DBMSs (e.g., PostgreSQL, MySQL) provide tools for workload capture. Previous work uses different representations for captured workloads, including raw SQL [79, 88], query plans [48, 77], system metrics [8, 45, 76], and simple classification (e.g., OLTP vs. OLAP) [39]. We contend that defining the workload as a time series of raw SQL is the best approach because (1) the queries do not change as the system changes, and (2) all other representations derive from them.

Given a historical workload of SQL queries, a DBMS's workload forecasting component seeks to predict and synthesize the future workload. The challenging questions in synthesizing a workload are deciding (1) what queries to consider for execution and (2) how to determine their arrival times. Existing research in workload prediction attempts to forecast volume [44], predict the subsequent SQL query [25, 61], and detect shifts [35]. Furthermore, workload compression [17] identifies important subsets of the workload for monitoring [23] and tuning [69]. However, these approaches only predict high-level properties or operate on historical workloads. To our knowledge, no existing work focuses on synthesizing a future workload that targets training data collection for ML components (e.g., a time series of future SQL queries that the DBMS can execute).

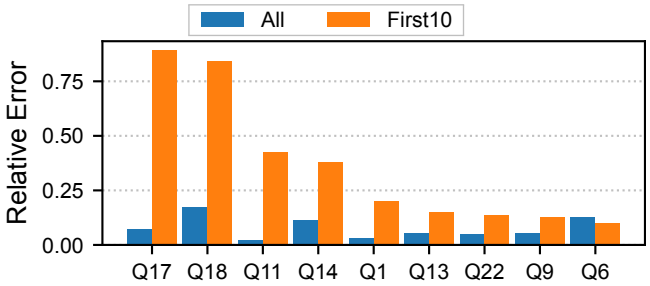


Figure 1: Workload Effects on Modeling Error – We train two models of QPPNet to predict query elapsed time on PostgreSQL v14 with TPC-H (scale factor 10). The All model enables hash joins for all 22 queries. The First10 model enables hash joins for only the first 10 queries. We leave out bars where both models are below 0.125 relative error for readability.

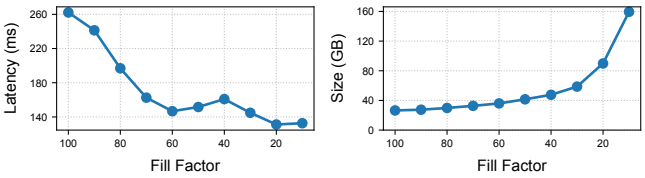
To illustrate why this matters, we study QPPNet [48], a state-of-the-art ML model for query performance prediction, using the TPC-H benchmark on PostgreSQL v14. QPPNet trains on annotated query plans (i.e., `EXPLAIN ANALYZE`) to predict the execution time of new queries. For this experiment, we disable hash joins for half of TPC-H's queries while collecting training data to simulate workload drift (e.g., query plan change due to statistics updates). As Figure 1 shows, if aspects of the target DBMS's workload are missing in the training data (i.e., some queries switch to using hash joins), then the model has a higher relative error and makes worse predictions. These results suggest that historical workloads are insufficient for training an autonomous DBMS's ML models, and therefore the system must predict and collect training data for the future workload.

2.2 State

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

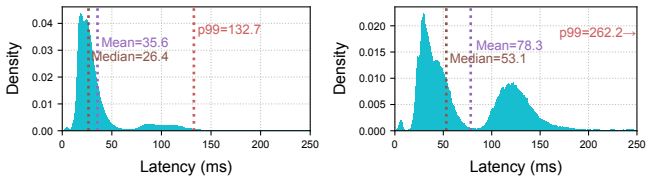
Hardware: The state's hardware specification details the resources available to the DBMS. This information includes compute (e.g., number of CPUs, cores, ISAs), storage (e.g., disk, memory), and network topology. These resources can either be the target DBMS's current hardware specification or potential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

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



(a) NewOrder p99 Latency **(b)** Database Size

Figure 2: Latency vs. Storage Trade-off – As fill factor decreases, the latency of the NewOrder transaction decreases, but database size increases.



(a) Fill Factor = 10 **(b)** Fill Factor = 100

Figure 3: NewOrder Latency Distributions – As the fill factor decreases, the latency approaches a Gaussian distribution.

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

A DBMS's runtime behavior (e.g., query execution performance) obviously depends on its hardware resources and logical contents. Its physical condition is equally important but is often overlooked in training data collection. When a DBMS executes a query that modifies a table, it can alter the table's underlying physical organization in ways that will affect the behavior of future queries. For example, because of the way PostgreSQL implements multi-versioning [82], updating a tuple may either (1) insert the new version on the same page as the original version or (2) insert the new version on another page. The logical contents in the database are the same in these two scenarios, but each will result in a different physical condition with varying performance costs.

To demonstrate the above, we execute TPC-C on PostgreSQL v14 while changing how the DBMS stores data. For each trial, we vary the DBMS's knob (`fillfactor`) that determines how much free space the system reserves in a page when inserting new tuples. Reducing the fill factor makes it more likely that the DBMS will place a tuple's new version in the same page as the previous version upon update. However, the extra padding also increases database size on disk. We load the TPC-C database with BenchBase [1, 24] and execute each trial for 30 minutes (scale factor 200, 200 terminals).

Figure 2 shows how changing the DBMS's physical condition results in trade-offs between its performance and storage. As we decrease PostgreSQL's fill factor knob, transaction latency decreases in Figure 2a while database storage size increases in Figure 2b. These results are unsurprising. However, Figure 3 shows that the min/max fill factor settings (10, 100) have different latency profiles, with the latter being pronouncedly bimodal. Moreover, the current knob value may not reflect the actual physical condition until a `VACUUM FULL` operation. This makes modeling physical condition difficult, as it is not enough to simply scrape the current knob value.

This experiment highlights the need to consider a DBMS's physical condition to accurately support more complex actions, such as tuning the fill factor for individual tables and indexes. Therefore, the quality of both training data features and labels depends on achieving a state representative of all three facets of a DBMS.

2.3 Actions

Most DBMSs provide programmatic APIs for deploying actions in the following categories: (1) *physical design*, (2) *schema changes*, (3) *knob configuration*, (4) *hardware resource allocation*, and (5) *query plan hints* [59]. The first two categories are changes to the database's physical state (e.g., indexes, views, partitioning) and logical state (e.g., changing column types). The third category consists of optimizations that affect the DBMS's behavior through its configuration knobs (e.g., caching policies). Next, resource allocations determine how the DBMS uses its available hardware to store data and execute queries. The system may either provision new resources (e.g., adding disks, memory, or machines) or redistribute existing resources (e.g., partitioning tables across disks). Lastly, query plan tuning hints are directives that force the DBMS's optimizer to make certain decisions for individual queries (e.g., join orderings). An autonomous DBMS must collect training data on these actions for its ML models.

Generating training data for all these actions is infeasible because too many possible actions exist. Thus, the key challenge is how to identify the actions that are worth considering based on the target DBMS's workload and state. For example, consider building multi-column indexes for a read-heavy workload like Wikipedia [24]. Wikipedia's MySQL schema [6] has 364 attributes across 58 tables. There are $\sum_{i=0}^{m} \frac{m!}{(m-i)!}$ potential multi-column indexes for a table with m attributes, which means Wikipedia has $\sim 7 \cdot 10^{22}$ possible indexes. Even if one could obtain training data for each index in a nanosecond, it would still take 2m years. This estimation also does not consider index types (e.g., hash vs. B+tree) and index arguments (e.g., split/merge thresholds, page sizes), exacerbating the problem.

Prior work addresses this problem by pruning the action search space with ad-hoc rules. For example, some index selection tools only consider two-column indexes [3], while others prune candidates based on simple criteria (e.g., columns appear in predicates [75]). But such ad-hoc rules will not result in holistic ML models for autonomous DBMSs. These rules must be programmable, enumerable, and integrated into the training data pipeline.

In this paper, we focus on DBMS-scoped actions. Future work may also include tuning OS-level knobs (e.g., OS page cache policy) or modifying application code (e.g., JDBC/ODBC libraries).

3 DATABASE GYMS

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

A *gym* is a toolkit that provides a consistent environment for offline training and evaluating ML algorithms that automate tasks on an agent. One of the most influential gyms is OpenAI Gym (OA-Gym) [14], designed for reinforcement learning (RL) problems. The RL setting places an *agent* inside an *environment* that advances in discrete simulation timesteps. At each timestep, the agent performs an *action* inside the environment that generates an *observation* and a *reward*. The agent considers past observations and rewards as it attempts to maximize its expected future reward by formulating an action-picking *policy* that explores an *action space*. OA-Gym's key contribution is providing a standardized API for agent-environment communication. This standardization led to the development of different environments (e.g., video games [13], robotic arms [14]) and agents [62, 63] that enable reproducible benchmarking and evaluation. Moreover, OA-Gym is extensible both in core functionality (e.g., distributed training [41], multi-agent environments [71]) and to other research domains [10, 30, 86].

The challenge of developing a *database gym* (DB-Gym) is that simulating the environment is complex and slow. Moreover, unlike other relatively standardized domains (e.g., most consumer cars drive similarly), DBMS deployments are expected to operate across heterogeneous hardware and diverse workloads, which requires heavy customization. The good news is that a perfect simulator exists for every DBMS, namely, the DBMS itself. But one cannot simply wrap a DBMS with a gym API and expect it to be usable. The gym must provide additional mechanisms for establishing the right state in the DBMS and executing workloads to produce high-quality training data. Additionally, the gym should aim to execute workloads faster than in real-time (see Section 3.2).

As Figure 4 shows, the DB-Gym has three main inputs: (1) the objective function and constraints, (2) the historical *workload*, and (3) the historical *state*. The gym uses these inputs and its internal components to produce training data that it stores externally.

The gym's internal components make up the *Environment* and *Agent* in the DB-Gym. The Environment consists of two components: ❶ the **Synthesizer** installs the state and workload of the target DBMS, and ❷ the **Trainer** coordinates creating an instance of the target DBMS, applying actions, and executing the workload to generate this iteration's observation and reward. The gym collects the Trainer's output as training data, stores it into a repository (*Repo*), and provides it as feedback to the Agent. The Agent incorporates this feedback into ❸ the **Planner** to generate candidate actions and uses ❹ the **Decider** to select actions to deploy.

In the rest of this section, we describe the DB-Gym's internals in further detail, focusing first on the Synthesizer and Trainer, followed by the Planner and Decider. In Section 4, we discuss open problems in system infrastructure related to running the DB-Gym.

3.1 Synthesizer

The Synthesizer's goal is to produce a workload and state for the DB-Gym's target DBMS. A baseline implementation simply passes through the historical workload and state (i.e., predicts no change). As Section 2 describes, although this is not ideal, it may still be valuable under certain assumptions (e.g., static read-only workloads). Thus, a better approach is to generate a future workload and state.

Synthesizing the target workload was introduced in Section 2.1. We refine the requirements further by noting that most prior work

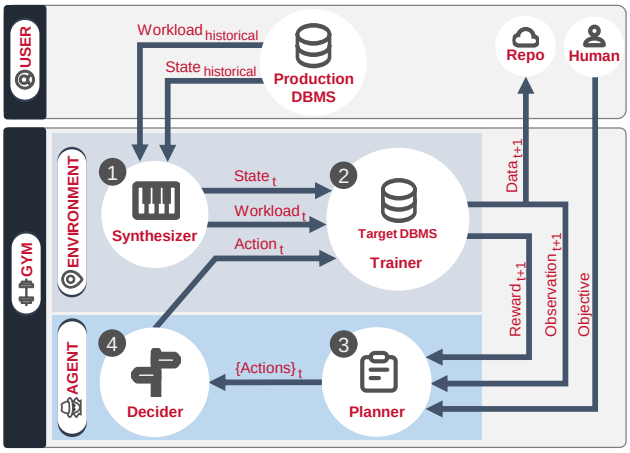


Figure 4: Database Gym Architecture – An overview of the database gym's internals. The t subscript refers to iterations inside the *Gym*.

analyzes and costs queries in isolation [21, 48]. But queries are not independent and have complex interactions when run together (e.g., lock contention, scan sharing). Furthermore, running a query can produce secondary effects (e.g., checking constraints, generating maintenance work, updating statistics) that are hard to model. Until more advanced models can capture these interactions, the DB-Gym must execute the target workload's SQL queries to generate high-quality training data. Therefore, the Synthesizer needs to support predicting individual SQL queries, which brushes against the limits of ML scalability (e.g., taking even 2 ms to predict a query limits throughput to 5000 queries per second). Many such open problems remain in making workload synthesis practical.

Section 2.2 describes the composition of the target state. The user specifies the hardware resources available, which need not be the same as for the production DBMS. The Synthesizer approximates future logical contents by measuring database growth rates and then scales [67, 70, 84, 87] the contents of tables accordingly. However, we are unaware of any work synthesizing future physical condition. We note that exact target state predictions are unnecessary as long as execution characteristics are similar (e.g., precise tuple location matters less than overall fill factor trends).

3.2 Trainer

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

The biggest challenges in this process are minimizing resource usage, accelerating query execution, and achieving the target's physical condition. For example, consider collecting training data on a workload when scaling the database size from 100 GB to 1 TB. The gym can collect this data by (1) synthesizing and loading an additional 900 GB of data, (2) overriding query plan leaf operators to scan "virtual" tuples [66], or (3) downsampling to a representative subset and extrapolating results. The first option is the most accurate but also has the highest storage cost and execution time. The

second option saves on storage, but it requires invasive changes to the DBMS and moreover does not speed up query execution. Finally, the third option saves on storage and execution time, but it may introduce extrapolation errors and have a different physical condition from the target DBMS. Searching for scaling laws to extrapolate from smaller and/or cheaper database instances, and exploring the trade-offs of these approaches, remain open problems.

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

The Trainer's overarching goal is to run workloads much more cheaply than in a real DBMS. For example, UDO [79] distinguishes between cheap and expensive actions to batch and evaluate cheap actions together. Similarly, in future work, we could share large amounts of state between workload evaluations (e.g., by keeping small deltas between "adjacent" states).

3.3 Planner

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

The Planner requires a way to suggest good actions. General database administration guidelines prescribe universal rules (e.g., use PGTune [4]), but these are often not optimal [76]. On the other hand, collecting training data for all possible actions is not feasible (see Section 2.3). Therefore, the Planner's key challenge is finding an action representation that allows for programmatic exploration and intelligent pruning of the search space.

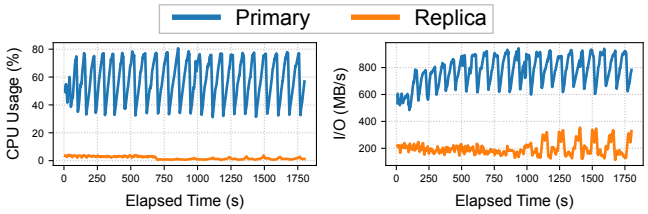
Existing research uses more sophisticated methods to prune the action search space. For example, prior work generates initial training data for ML models to select knob configurations via Latin hypercube sampling [76]. However, the resulting actions do not have a notion of distance from one another (e.g., an action that sets PostgreSQL's working memory to 4 MB should be considered more similar to a 5 MB setting than a 50 MB setting). The Planner could learn these distances with deep action embeddings [27].

3.4 Decider

The Decider selects the most promising actions from the list of actions suggested by the Planner and provides them to the Trainer. Learning to pick the most promising action is the purview of RL research. Therefore, the Decider may incorporate multiple learning algorithms with minimal modification (e.g., SB3 [63], UDO [79]).

4 DEPLOYMENT

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



(a) CPU (user + system) **(b)** IO (read + write)

Figure 5: Replica Resource Availability – We run SmallBank (scale factor 100, 2000 terminals, 30 minutes) on PostgreSQL with one primary and one replica. Each datapoint averages across the last 10 readings to reduce noise.

Two obvious considerations are whether the DB-Gym can run on the production DBMS (low cost, high impact) or if it must run on a separate instance (high cost, low impact). The first option requires restricting the gym to avoid violating SLAs with bad actions on the production DBMS. For example, Oracle can try index actions in production because it schedules index exploration at idle times and forces index usage with query hints [53]. But some actions are difficult to sandbox (e.g., knob configurations [60]). Hence, we believe that usually the DB-Gym should not run directly on a production system. The second approach involves creating an exact replica of the production DBMS on the same hardware. Microsoft uses this approach (calling them "B-instances") for auto-indexing in Azure [21, 43]. Such instances are completely isolated, which allows the DB-Gym to explore actions freely, but this approach is often prohibitively expensive and infeasible at scale.

Given the above issues, the ideal DB-Gym deployment keeps a B-instance's isolation advantages but without the cost. To achieve this, we can exploit the fact that (1) the DB-Gym can tolerate stale database snapshots, and (2) the DBMS can tear down the non-essential gym at any time. We therefore contend that we should reuse high-availability (HA) replicas to deploy the DB-Gym in a resource-limited container. HA replicas are notoriously idle, allowing the DB-Gym to scavenge [42] unused resources for its training purposes. These containers bootstrap their database contents with incremental snapshots and the DBMS's built-in recovery logic.

To demonstrate the idle resources available for the DB-Gym, we ran SmallBank [1, 24] on PostgreSQL v14 with a primary/replica configuration on two machines while measuring CPU and disk I/O utilization on each machine. Figure 5 demonstrates that HA replicas have ample amounts of spare compute and reasonable disk I/O available. However, this approach requires raw access to the DBMS instance, making it impossible in managed cloud environments.

Lastly, although we have restricted our focus in this paper to optimizing autonomous single-node systems, real DBMS deployments offer more tuning opportunities (e.g., optimizing replication network topology). Moreover, beyond database tuning, the DB-Gym's organization provides opportunities to replace or learn individual components. We defer such considerations to future work.

5 RELATED WORK

Throughout the decades of autonomous DBMS research, ML models have always required training data. The early 2000s saw tools such as IBM DB2 Advisor [75], but these were often rule-based and system-specific. In recent years, the use of ML for DBMS automation has grown, though the extent of automation varies [60]. Examples include commercial systems such as Amazon Redshift [11, 57], Google AlloyDB [19], Oracle [54, 55], Oracle's MySQL HeatWave [56], Huawei openGauss [40], and Microsoft Azure SQL [21]. They also include academic systems such as NoisePage [58, 60] and SageDB [38]. As these systems depend on their internal ML models to guide operation, they benefit from better training data.

Recent work has focused on individual facets of the training data generation process, including speed [15], location [21, 43], and quality [77, 80]. However, no existing system integrates these techniques nor solves the research and operational challenges described in Sections 2 and 4 to achieve a database gym environment that supports ML model training and evaluation. The closest thing to a database gym today is UDO [79], which we discuss below.

Gym Environment: UDO [79] is a tool that performs offline tuning for DBMSs. Given a workload and an environment, it uses RL to find better DBMS configurations model-free, implementing the OpenAI Gym API to compare different algorithms. UDO focuses on developing novel algorithms [78] like the Decider uses, whereas the DB-Gym synthesizes UDO's input (i.e., workload, state). We now discuss prior synthesis work.

Synthesizing Workloads: To our knowledge, no prior work synthesizes a complete workload. There are methods to auto-scale for provisioning [22, 31, 65], DBMS performance modeling [51, 52, 85], next SQL statement prediction [26, 33, 34, 36, 61], workload compression [17], and query runtime metrics prediction [29, 32]. Ma et al. argues that these are insufficient for predicting workload volume, duration, and change [44]. Instead, they propose predicting arrival rates, but these too may be insufficient if raw query contents are required (e.g., for query-level features). Moreover, the DB-Gym runs individual queries to approximate future DBMS state.

Synthesizing State: To our knowledge, no work tries to synthesize physical conditions for DBMSs. However, existing research synthesizes logical contents to (1) scale datasets and (2) fake data for testing. Dataset scaling [70] synthesizes new tuples based on either workload [67, 84] or individual table contents [87] while preserving cardinality constraints (e.g., data correlations). Faking data for testing uses techniques such as substitution rules [2, 5], differential privacy [74], and generative networks [72] to improve ML model performance [68] and allay privacy concerns [74]. The gym uses the above techniques to approximate future state with similar properties (e.g., join cardinalities) to improve training data.

Training Data Quality: DataFarm [77, 80] generates synthetic jobs from an input workload. It builds a model to predict job labels and ranks the jobs by the uncertainty of its predictions. Next, it uses active learning or asks a human to pick which job to evaluate. We note that operating at the level of query plans aids synthetic data generation but increases the sensitivity to the DBMS state. The DB-Gym complements DataFarm by providing the input workload and the future state, as well as a location to run in.

Cosine [16] is a self-designing key-value storage engine that gathers high-quality training data to train its concurrency-aware CPU model. The model requires the workload's degree of parallelism to be known. Cosine learns this by sweeping across different factors (e.g., cloud instance type, number of operations, number of CPU cores) as it executes the workload.

Location: As Section 4 describes, Oracle [53] uses safeguards to try indexes on the primary DBMS, whereas Microsoft [21] tries indexes on B-instances (i.e., independent DBMS copies that receive and replay the primary's workload). The DB-Gym balances between the two approaches by running on high-availability replicas.

6 CONCLUSION

Most of the previous work in using ML for DBMS automation has focused on designing better ML models of DBMS behavior, but recent advances in ML have largely automated model design. The challenge now is to obtain good training data for building these models. This paper outlined the architecture of the database gym, an integrated environment that generates training data by using the DBMS to simulate itself at the highest possible fidelity.

Acknowledgments

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

References

[1] 2022. BenchBase – Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase.
[2] 2022. clickhouse-obfuscator. https://clickhouse.com/docs/en/operations/utilities/clickhouse-obfuscator/.
[3] 2022. Dexter – The automatic indexer for Postgres. https://github.com/ankane/dexter.
[4] 2022. PGTune – PostgreSQL configuration wizard. https://github.com/gregs1104/pgtune.
[5] 2022. Replibyte – Seed your development database with real data. https://www.replibyte.com/docs/introduction.
[6] 2022. Wikipedia: Database schema. https://en.wikipedia.org/wiki/Wikipedia:Database_download#SQL_schema.
[7] DCAI 2021. 2021. NeurIPS Data-Centric AI Workshop. https://datacentricai.org/neurips21/.
[8] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253.
[9] Amazon Web Services, Inc. 2020. Amazon Redshift announces Automatic Table Optimization. https://aws.amazon.com/about-aws/whats-new/2020/12/amazon-redshift-announces-automatic-table-optimization/.
[10] Selim Amrouni, Aymeric Moulin, Jared Vann, Svitlana Vyetrenko, Tucker Balch, and Manuela Veloso. 2021. ABIDES-gym: gym environments for multi-agent discrete event simulation and application to financial markets. In *ICAIF'21*. ACM, 30:1–30:9.
[11] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22*. ACM, 2205–2217.
[12] Aurora. 2021. Scaling Simulation. https://aurora.tech/blog/scaling-simulation.
[13] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *J. Artif. Intell. Res.* 47 (2013), 253–279.
[14] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *CoRR* abs/1606.01540 (2016). arXiv:1606.01540

[15] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems *(SIGMOD '22)*.

[16] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. *Proc. VLDB Endow.* 15, 1 (2021), 112–126.

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

[18] Google Cloud. 2022. AutoML. https://cloud.google.com/automl/.

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

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

[21] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *SIGMOD Conference 2019*. ACM, 666–679.

[22] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *SIGMOD Conference 2016*. ACM, 1923–1934.

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

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

[25] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: an automatic query steering framework for interactive data exploration. In *SIGMOD 2014*. ACM, 517–528.

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

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

[28] Leonidas Galanis, Supiti Buranawatanachoke, Romain Colle, Benoît Dageville, Karl Dias, Jonathan Klein, Stratos Papadomanolakis, Leng Leng Tan, Venkateshwaran Venkataramani, Yujun Wang, and Graham Wood. 2008. Oracle Database Replay. In *SIGMOD 2008*. ACM, 1159–1170.

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

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

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

[32] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. 2008. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC 2008*. IEEE Computer Society, 13–22.

[33] Marc Holze, Claas Gaidies, and Norbert Ritter. 2009. Consistent on-line classification of dbs workload events. In *CIKM 2009*. ACM, 1641–1644.

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

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

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

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

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

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

[40] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: An Autonomous Database System. *Proc. VLDB Endow.* 14, 12 (2021), 3028–3041.

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

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

[43] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active Learning for ML Enhanced Database Systems. In *SIGMOD Conference 2020*. ACM, 175–191.

[44] 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 *(SIGMOD '18)*. 631–645.

[45] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems *(SIGMOD '21)*. 1248–1261.

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

[47] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD '21*. ACM, 1275–1288.

[48] Ryan C. Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746.

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

[50] Microsoft. 2022. Monitor performance by using the Query Store. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store.

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

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

[53] Oracle. 2019. Autonomous Indexing. https://blogs.oracle.com/connect/post/autonomous-indexing.

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

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

[56] Oracle. 2022. Oracle HeatWave. https://www.oracle.com/mysql/heatwave/.

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

[58] 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*.

[59] 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. *IEEE Data Engineering Bulletin* (June 2019), 32–46.

[60] Andrew Pavlo, Matthew Butrovich, Lin Ma, Wan Shen Lim, Prashanth Menon, Dana Van Aken, and William Zhang. 2021. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. *Proc. VLDB Endow.* 14, 12 (2021), 3211–3221.

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

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

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

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

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

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

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

[68] Scale AI, Inc. 2022. Scale Synthetic. https://scale.com/synthetic.
[69] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek R. Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *SIGMOD '22*. ACM, 660–673.
[70] Y. C. Tay. 2011. Data Generation for Application-Specific Benchmarking. *Proc. VLDB Endow.* 4, 12 (2011), 1470–1473.
[71] Justin K. Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S. Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, Niall L. Williams, Yashas Lokesh, and Praveen Ravi. 2021. PettingZoo: Gym for Multi-Agent Reinforcement Learning. In *NeurIPS 2021*. 15032–15043.
[72] The Synthetic Data Vault. 2022. SDV – The Synthetic Data Vault. https://sdv.dev/.
[73] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. MuJoCo: A physics engine for model-based control. In *IROS 2012*. IEEE, 5026–5033.
[74] Tonic. 2022. Tonic.ai – The fake data company. https://www.tonic.ai/.
[75] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE 2000*. IEEE Computer Society, 101–110.
[76] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning *(SIGMOD '17)*. 1009–1024.
[77] Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Expand your Training Limits! Generating Training Data for ML-based Data Management. In *SIGMOD '21*. ACM, 1865–1878.
[78] Junxiong Wang, Debabrota Basu, and Immanuel Trummer. 2022. Procrastinated Tree Search: Black-Box Optimization with Delayed, Noisy, and Multi-Fidelity Feedback. In *IAAI 2022*. AAAI Press, 10381–10390.
[79] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization using Reinforcement Learning. *Proc. VLDB Endow.* 14, 13 (2021), 3402–3414.
[80] Robin Van De Water, Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. DataFarm: Farm Your ML-based Query Optimizer's Food! - Human-Guided Training Data Generation -. In *CIDR 2022*. www.cidrdb.org.
[81] Kelvin Wong, Qiang Zhang, Ming Liang, Bin Yang, Renjie Liao, Abbas Sadat, and Raquel Urtasun. 2020. Testing the Safety of Self-driving Vehicles by Simulating Perception and Prediction *(Lecture Notes in Computer Science, Vol. 12371)*. Springer, 312–329.
[82] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (March 2017), 781–792.
[83] Anatoly Yakovlev, Hesam Fathi Moghadam, Ali Moharrer, Jingxiao Cai, Nikan Chavoshi, Venkatanathan Varadarajan, Sandeep R. Agrawal, Tomas Karnagel, Sam Idicula, Sanjay Jinturkar, and Nipun Agarwal. 2020. Oracle AutoML: A Fast and Predictive AutoML Pipeline. *Proc. VLDB Endow.* 13, 12 (2020), 3166–3180.
[84] Jingyi Yang, Peizhi Wu, Gao Cong, Tieying Zhang, and Xiao He. 2022. SAM: Database Generation from Query Workloads with Supervised Autoregressive Models. In *SIGMOD '22*. ACM, 1542–1555.
[85] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *SIGMOD Conference 2016*. ACM, 1599–1614.
[86] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernández Cordero. 2016. Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo. *CoRR* abs/1608.05742 (2016). arXiv:1608.05742
[87] J. W. Zhang and Y. C. Tay. 2016. Dscaler: Synthetically Scaling A Given Relational Database. *Proc. VLDB Endow.* 9, 14 (2016), 1671–1682.
[88] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58.