# Automated Database Tuning vs. Human-Based Tuning in a Simulated Stressful Work Environment

## A Demonstration of the Database Gym


Patrick Wang
phw2@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, USA

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

William Zhang
wz2@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, USA

Samuel Arch
sarch@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, USA

Andrew Pavlo
pavlo@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, USA



## Abstract

Machine learning (ML) has gained traction in academia and industry for database management system (DBMS) automation. Although studies demonstrate that ML-based tuning agents match or exceed human expert performance in optimizing DBMSs, researchers continue to build bespoke tuning pipelines from the ground up. The lack of a reusable infrastructure leads to redundant engineering effort and increased difficulty in comparing modeling methods. This paper demonstrates the database gym framework, a standardized training environment that provides a unified API of pluggable components. The database gym simplifies ML model training and evaluation to accelerate autonomous DBMS research. In this demonstration, we showcase the effectiveness of automated tuning and the gym's ease of use by allowing a human expert to compete against an ML-based tuning agent implemented in the gym.



## CCS Concepts

• **Information systems** → **Autonomous database administration**.

## Keywords

Database Systems; Automated Database Tuning; OpenAI Gym

**ACM Reference Format:**
Patrick Wang, Wan Shen Lim, William Zhang, Samuel Arch, and Andrew Pavlo. 2025. Automated Database Tuning vs. Human-Based Tuning in a Simulated Stressful Work Environment: A Demonstration of the Database Gym. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725083


## 1 Introduction

Database management systems (DBMSs) are among the most challenging software systems to tune and configure effectively. This complexity has driven decades of research into automated approaches for optimizing physical design choices, parameter settings, and system configuration. Recent years have seen growing adoption of machine learning (ML) techniques in both academic research [12, 13, 17–19] and commercial systems [4, 7, 8] to address these optimization challenges.

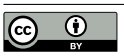
These ML methods aim to create an autonomous DBMS that operates without human guidance (i.e., Level #5 – self-driving DBMS [15]). Such a system aims to optimize itself automatically for a given *objective function* (e.g., latency, throughput, cost) and constraints (e.g., cost budget, SLA) [14]. The autonomous DBMS improves this objective by deploying actions (e.g., creating an index, setting system knobs, or adding optimizer hints to queries) that modify the configuration of the DBMS.

The foundation of such ML-augmented systems are models trained from data collected from the system. This training data provides information about the objective value of the workload over varying database configurations. Each ML-augmented system must collect its own data because the training data is specific to the workload and hardware. The system may collect data before the training process [17] or during the training process itself [18].

Most previous work on using DBMS automation focuses on enhancing ML models [11]. These efforts often reimplement the database tuning pipeline from scratch: workload capture/generation, database setup, training data collection, model creation, and model deployment. Such bespoke pipelines make it challenging to combine techniques even when they should be independent (e.g., using a different operator latency model in a tuning algorithm).

The **Database Gym** (DB-Gym [10]) framework aims to standardize an API for these disparate tasks, allowing researchers to mix and match the different components in the pipeline. We take inspiration from the Gymnasium [16] project (formerly OpenAI Gym [5]), which accelerates the development and comparison of reinforcement learning algorithms by providing a set of agents, environments, and a standardized API for communicating between them. Likewise, we designed the DB-Gym to provide an extensible open-source platform for autonomous DBMS research.

We demonstrate the DB-Gym with a competition between a human and a state-of-the-art tuning algorithm [18]. The backend for this competition is implemented on top of the DB-Gym framework





*SIGMOD-Companion '25, Berlin, Germany*



ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725083

to show its ability to compare two tuning "agents" (one human and one machine) in a standardized environment. The backend's simple implementation (a few lines of code) also demonstrates the ease of setting up an end-to-end tuning pipeline with the database gym.

We lay out the remainder of the paper as follows. We first describe the DB-Gym's architecture in Section 2. We then discuss our demonstration in Section 3.

## 2 Architecture

**Repository**:
⇒ https://github.com/cmu-db/dbgym

We now describe the high-level architecture of a DB-Gym, as shown in Figure 1. A DB-Gym comprises three main components: (1) the **Orchestrator** that handles initialization and setup, (2) the **Agent(s)** that interact with and tune the DBMS, and (3) the **DBMS Process** that evaluates the target workload on different configurations. These three components then coordinate through two standardized interfaces: (1) the *Workspace* and (2) the *Environment*. The DB-Gym exposes these interfaces to the components through its public API. Next, we discuss how these components interact within the gym to tune and optimize a DBMS, followed by an overview of key design decisions in its implementation.

### 2.1 Components and Interfaces

In Figure 1, the Orchestrator first creates its workspace in step (1). The workspace is a filesystem directory that stores the artifacts that the DB-Gym generates during tuning (e.g., queries, training data, models). The gym exposes this directory through the *Workspace*'s Python API. The Orchestrator only interacts with the Tuning Agent through the Workspace interface.

Next, in step (2), the DB-Gym initializes the execution environment based on the workspace's contents. This setup deploys the DBMS with an initial configuration and loads the target database(s). The gym exposes another Python interface (*Environment*) that provides the necessary functionality for DBMS tuning. All of the **Tuning Agent**'s interactions with the DBMS Process (e.g., deploy action, time a query, restart) go through this Environment. This deployment configuration allows the gym to abstract away the specifics of different DBMSs, providing a unified interface for agents.

After the DB-Gym initializes the environment, the Tuning Agent begins the tuning process. For illustrative purposes, we assume an iterative agent implementation (i.e., it deploys one action at a time, observing the outcome to decide what to do next). In step (3a), the Tuning Agent first analyzes the workload (schema, data, and queries) that is stored in the workspace. It then iteratively (3b) deploys actions through the Environment. (3c) For each action, the Environment reconfigures the DBMS, evaluates the workload, and returns the reward to the Tuning Agent. Although our example in Figure 1 illustrates an OLAP setting that aims to minimize query runtime, the gym also supports OLTP scenarios (e.g., maximize workload throughput, minimize p99 latency). (4) The Tuning Agent writes the tuning results to the Workspace.

Lastly, after the Tuning Agent completes its tuning process, in step (5), the Orchestrator then analyzes the tuning results from the Workspace to perform an apples-to-apples comparison between the different agents. The gym framework provides a stable mechanism for comparing the performance of different agents on the same workload and hardware without needing to adjust for agent-specific optimizations (e.g., timeouts, partial workload evaluation) as this is abstracted away by step (3).

### 2.2 Key Design Decisions

We next present motivating rationales behind the DB-Gym and its core design. These design decisions are based on over a decade of experience developing ML-based automated tuning tools for DBMSs. As we now discuss, the DB-Gym's architecture is based on (1) a centralized control structure, (2) tuning artifacts as a sequence of deltas, and (3) Orchestrator-driven replay.

The first goal of the DB-Gym is to encapsulate all DBMS logic through a clean abstraction (i.e., the *Environment*). This centralization allows researchers to focus on creating and improving tuning agents rather than deal with DBMS-specific details around connecting, setting parameters, or deploying actions. For instance, in PostgreSQL, setting the maximum number of worker processes requires a DBMS restart whereas changing an optimizer knob does not require restarting.

The *Environment* also provides a consistent mechanism for researchers to deploy standardized environments (e.g., skewed TPC-H with the same random seeds) and relevant plugins or extensions. For example, our PostgreSQL deployment includes `boot` [11] for accelerating training data generation, `hypopg` [3] for hypothetical indexes, and `pg_hint_plan` [6] for optimizer hints (e.g., query knobs). Additionally, because the Tuning Agent interacts with the DBMS through the Environment API, developers can easily switch reward functions (e.g., from runtime-based to cost-based) without worrying about how to compute those rewards (e.g., through `EXPLAIN`, accelerated query execution by `boot`, or with ML models [12]). Standardizing the reward function also helps to avoid subtle errors in its computation. For example, many DBMSs report misleading operator-level costs and runtimes because they are inconsistent in whether they include children nodes [1, 2].

Our second design decision is to express tuning results as a **sequence** of configuration deltas rather than a single result or workload runtime. This sequence provides researchers with better insight into an Agent's tuning process (e.g., by visualizing its discovered configurations and improvements over time). The configuration deltas also provide a "time-travel" capability for consistent result reproducibility. They allow the Orchestrator to load a specific step (i.e., a particular configuration) by replaying the sequence from the initial configuration.

Lastly, we designed the gym so that the Orchestrator could compare the performance of different agents. The Orchestrator **replays** each agent's configuration deltas on a new DBMS instance through the Environment. The Orchestrator then re-evaluates the workload over each configuration without further input from the agents to ensure a fair comparison.

## 3 Demonstration

**Demo Video**:
⇒ https://cmudb.io/gym-demo-sigmod25

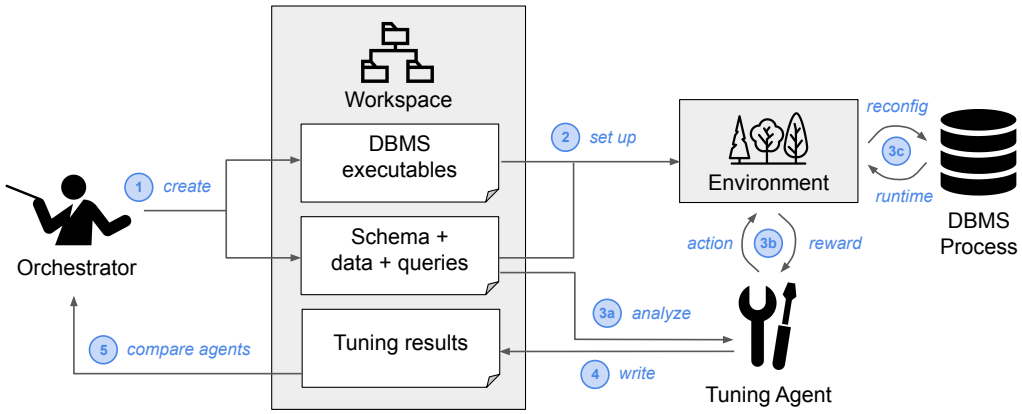



**Figure 1: DB-Gym Architecture** – An overview of the three components of the DB-Gym: (1) *Orchestrator*, (2) *Tuning Agent*, and (3) *DBMS Process*. The DB-Gym also provides two interfaces (unfilled icons in gray boxes) that these components use to interact with each other: (1) *Workspace* and (2) *Environment*. The arrows with blue labels represent the steps in the workflow when using the database gym. First, the Orchestrator creates the DBMS executable and workload files (schema, table data, and queries). Second, the Environment is initialized using these files. Third, the Tuning Agent uses the Environment to tune the DBMS. Fourth, the Tuning Agent writes the results to the Workspace. Lastly, the Orchestrator compares the results across different Tuning Agents.

Our demonstration of the database gym is meant to showcase the state-of-the-art in automated tuning and the need for infrastructure like the DB-Gym to facilitate their development. We will challenge participants to tune PostgreSQL to achieve the best performance for three queries from JOB [9]. As shown in Figure 2, we will provide participants with an immersive interface for database tuning where they have 60 seconds to tune (1) secondary indexes, (2) system knobs, and (3) query knobs. The gym will evaluate and rank the participants' final configurations relative to the best configuration found by Proto-X [18], a state-of-the-art automated database tuner.

① The gym exposes a web interface that simplifies the complexity of database tuning for participants by limiting the number of tuning options. ② After the participant selects their choices, the gym submits their configuration to a separate server to run the workload and measure the DBMS's performance. ③ We chose the JOB workload so that the evaluation process will only take a few seconds, allowing participants to receive synchronous feedback. ④ The gym will also compile the results of all participants into a leaderboard, including the performance of Proto-X.

We now describe the simplified tuning options offered by the demonstration. First, the web interface (shown in Figure 3) only allows the user to select indexes on a single column with up to a single include column. All indexes are restricted to B-Tree indexes with no further options (e.g., `fillfactor`, `pages_per_range`). Additionally, we pre-filter the possible indexes based on the tables and columns that appear in the workload. Next, we limit the configuration of system and query knobs to the ten most important knobs. We also simplify knob tuning by limiting all numerical knobs to five possible values.

For fairness, we modified Proto-X so that its final configuration is fully reproducible using the demonstration's web interface. We deleted all indexes on tables and columns that are not present in the queries being evaluated. We also removed any indexes that were built on more than one column or had more than one include column. We ignore its attempts to set index options (e.g., `fillfactor`, `pages_per_range`, non-B-tree options) and use PostgreSQL's default settings for all indexes. We restrict the set of system and query knobs that Proto-X is allowed to modify and round the value of numerical knobs to the values present in the web interface.

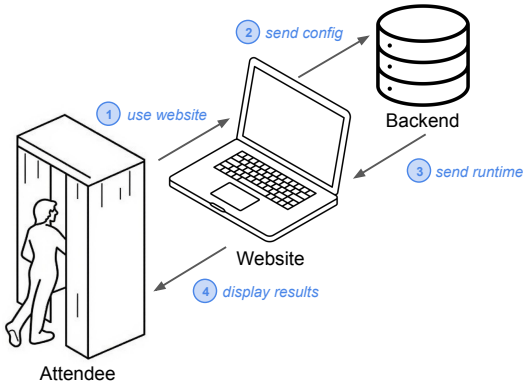



**Figure 2: Demo Flow** – An illustration of the steps attendees will take during the demonstration. First, they will enter a booth with a laptop in it and use the web UI to select a DBMS config. Second, they will submit their DBMS config to the backend server. Third, the backend will configure the DBMS based on that config, evaluate the workload's runtime, and send it back to the user. Fourth, the website will display the user's runtime.

**Demo Takeaways:** The goals of this demo are three-fold. First, we seek to engage the audience with a demonstration of the current state-of-the-art in DBMS tuning. Unlike previous automated database tuners that consider one system aspect (e.g., indexes, system knobs) at a time, Proto-X is *holistic*: it considers all system aspects at once, including query knobs. Second, we will showcase the backend that we implemented with the DB-Gym framework. The simplicity of the backend (two short scripts) demonstrates the flexibility and ease of use of the DB-Gym framework. Lastly, we hope that the game will provide participants with a new-found appreciation for the difficulty of tuning DBMSs under duress.

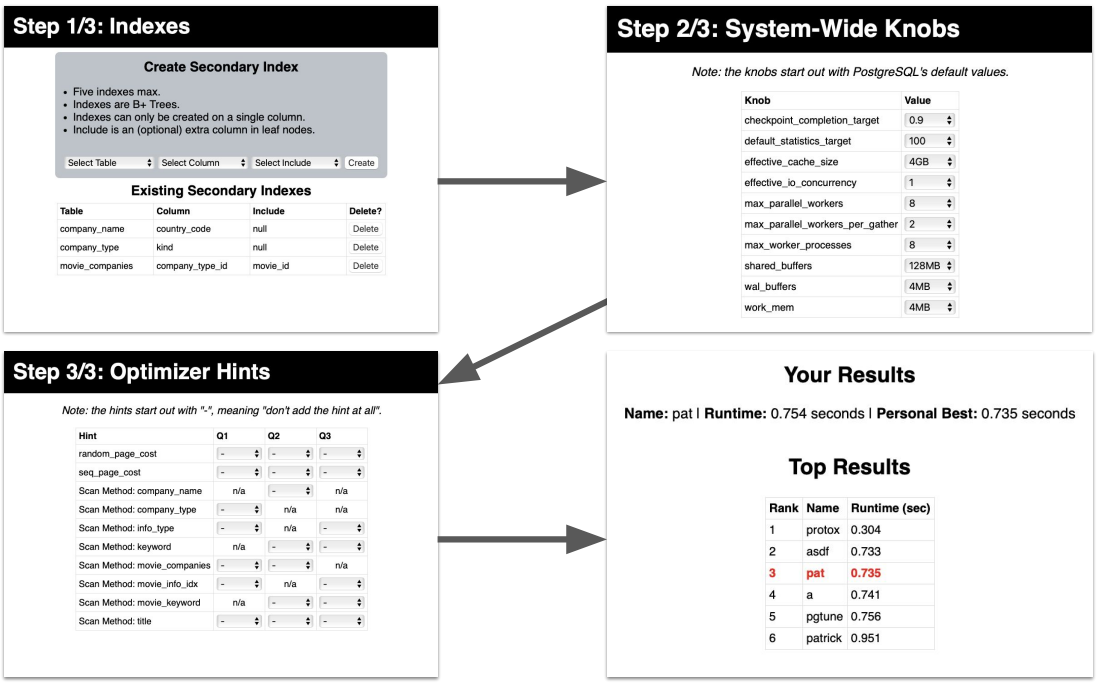



**Figure 3: Demo Web UI** – The figure shows the main pages in the web UI that attendees will interact with during the demo. First, they will select indexes. Second, they will select system knobs. Third, they will select query knobs (e.g., optimizer hints). Lastly, they will see their runtime on a leaderboard.

## 4 Conclusion

Through our human vs. machine tuning competition, we demonstrate how automated agents can effectively optimize database performance in stressful environments where humans struggle. The demo was built using the DB-Gym framework, which provides unified APIs between different components of the tuning pipeline. By providing this extensible open-source platform, we aim to accelerate research in autonomous DBMS tuning by allowing researchers to focus on novel ideas rather than reimplementing basic infrastructure.

## Acknowledgments

The authors would like to give a shout out to Jignesh Patel for going hard in the paint and Kendrick Lamar for his contributions to the CMU Database Group.

## References

[1] 2021. *pgmustard: Calculating per-operation times in EXPLAIN ANALYZE.* https://www.pgmustard.com/blog/calculating-per-operation-times-in-postgres-explain-analyze
[2] 2022. *Understand Your Plan: Reading Operator Times In SQL Server Execution Plans.* https://erikdarling.com/understand-your-plan-operator-times-in-sql-server-execution-plans/
[3] 2025. HypoPG. https://github.com/HypoPG/hypopg.
[4] 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/.
[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *ArXiv* abs/1606.01540 (2016).
[6] NTT OSS Center. 2025. pg_hint_plan. https://github.com/ossc-db/pg_hint_plan.
[7] Google Cloud. 2022. Introducing AlloyDB for PostgreSQL. https://cloud.google.com/blog/products/databases/introducing-alloydb-for-postgresql.
[8] 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.
[9] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal* 27, 5 (October 2018).
[10] Wan Shen Lim, Matthew Butrovich, William Zhang, Andrew Crotty, Lin Ma, Peijing Xu, Johannes Gehrke, and Andrew Pavlo. 2023. Database Gyms. In *CIDR 2023, Conference on Innovative Data Systems Research.* https://db.cs.cmu.edu/papers/2023/p27-lim.pdf
[11] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. 2024. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. *Proc. VLDB Endow.* 17, 11 (July 2024), 14 pages. doi:10.14778/3681954.3682030
[12] 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).* 14 pages.
[13] 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.
[14] Andrew Pavlo et al. 2017. Self-Driving Database Management Systems. In *CIDR 2017.*
[15] 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.
[16] Mark Towers et al. 2024. Gymnasium: A Standard Interface for Reinforcement Learning Environments. *ArXiv* abs/2407.17032 (2024).
[17] 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.
[18] William Zhang, Wan Shen Lim, Matthew Butrovich, and Andrew Pavlo. 2024. The Holon Approach for Simultaneously Tuning Multiple Components in a Self-Driving Database Management System with Machine Learning via Synthesized Proto-Actions. *Proc. VLDB Endow.* 17 (2024), 3373–3387.
[19] Xinyi Zhang, Zhuo Chang, Hong Wu, Yang Li, Jia Chen, Jian Tan, Feifei Li, and Bin Cui. 2023. A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning. *Proc. ACM Manag. Data* 1, 2, Article 186 (jun 2023).