Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems

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

Lin Ma
linmacse@umich.edu
University of Michigan

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

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

Samuel Arch
sarch@cs.cmu.edu
Carnegie Mellon University

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

ABSTRACT

Autonomous database management systems (DBMSs) aim to optimize themselves automatically without human guidance. They rely on machine learning (ML) models that predict their run-time behavior to evaluate whether a candidate configuration is beneficial without the expensive execution of queries. However, the high cost of collecting the training data to build these models makes them impractical for real-world deployments. Furthermore, these models are instance-specific and thus require retraining whenever the DBMS's environment changes. State-of-the-art methods spend over 93% of their time running queries for training versus tuning.

To mitigate this problem, we present the Boot framework for automatically accelerating training data collection in DBMSs. Boot utilizes macro- and micro-acceleration (MMA) techniques that modify query execution semantics with approximate run-time telemetry and skip repetitive parts of the training process. To evaluate Boot, we integrated it into a database gym for PostgreSQL. Our experimental evaluation shows that Boot reduces training collection times by up to 268× with modest degradation in model accuracy. These results also indicate that our MMA-based approach scales with dataset size and workload complexity.

PVLDB Reference Format:
Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. PVLDB, 17(11): 3680 - 3693, 2024.
doi:10.14778/3681954.3682030

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lmwnshn/boot.

1 INTRODUCTION

Database management systems (DBMSs) are difficult to configure and optimize due to shifts in workloads, database contents, and run-time environments. Researchers have spent decades developing methods for automated DBMS configuration (e.g., physical design, knob settings). Recent years have seen a surge of interest in applying machine learning (ML) to this problem, including query performance prediction [34, 37, 64], query optimization [35], index recommendation [12, 14], knob tuning [29], and partitioning [21].

The high-level goal unifying these efforts is the development of an autonomous DBMS that operates without human guidance (i.e., a *self-driving* DBMS [42, 43]). Given a target objective function (e.g., latency, throughput, cost), a self-driving DBMS aims to autonomously configure, tune, and optimize itself as its database contents and workload evolve. Because it is prohibitively expensive to identify beneficial configurations through workload execution, such autonomous DBMSs rely instead on *behavior models* [34] that predict the system's run-time characteristics. For example, a behavior model that uses optimizer statistics to predict a SQL query's latency [37] obviates the need to run the query in the general case; the DBMS validates its predictions through active learning [32].

To build its behavior models, the DBMS requires *training data* [8] comprised of database metadata (e.g., optimizer statistics) and run-time telemetry (e.g., operator latency). The DBMS avoids collecting training data from production to safeguard its stability [30] and performance [8]. Instead, database gyms [30] create a simulation environment for the DBMS to safely develop, build, and evaluate ML models. The DBMS generates training data in a gym by observing itself as it executes a representative workload, such as a trace of SQL queries, and constructs its behavior models based on this data.

A tacit assumption pervading existing work on applying ML for DBMSs is that training data for behavior models is cheap or easy to obtain. In practice, training data generation is slow and expensive because it requires the DBMS to execute workloads [22]. We observe that existing methods spend over 93% of their time preparing, testing, and evaluating these models [15, 34, 61], which can take weeks. Despite this, to our knowledge, there have been no efforts to improve the speed of training data generation for self-driving DBMSs. Some techniques sidestep this issue by learning cardinality estimation models from the data for approximate query processing (AQP) [22, 51, 59]. Although AQP speeds up query execution by estimating results, it does not help with behavior models since they must observe and predict the DBMS's run-time characteristics. Hence, obtaining training data is still a time-consuming bottleneck.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682030

Another problem is that existing methods often have to retrain their models from scratch, either because of new environments (e.g., software updates [34]) or model invalidation (e.g., due to dataset growth [37] or workload drift [36]). It is challenging to reuse training data from other deployments when they differ in database contents, DBMS versions, hardware, and configurations [30].

Given this, what is needed is a way for the DBMS to generate training data faster. Because the DBMS generates training data during query execution, the two processes are coupled: it cannot produce training data faster than it can run the workload.

We now present the **Boot** framework that accelerates training data generation by uncoupling these processes. Boot aims to efficiently *bootstrap* a DBMS's behavior models. To achieve this, we introduce macro- and micro-acceleration (MMA) techniques that leverage the database gym's environment to modify the DBMS's execution semantics for faster query processing. Macro-acceleration decides when to execute a query for its telemetry, and micro-acceleration speeds up execution by adaptively sampling the DBMS's run-time behavior. To evaluate Boot, we integrate it into a database gym based on PostgreSQL. We assess Boot on its ability to improve training data collection time and the accuracy of its resulting models on both uniform and complex workloads. Our results show Boot achieves up to 268× speedup for a modest increase in model error [62]. We also show that Boot is 3–5× faster than hand-optimized sampling methods, completing in four days what the other techniques complete in three weeks.

Our work makes the following contributions: (1) decoupling training data generation from regular query execution to overcome the training data bottleneck in achieving DBMS autonomy, (2) presenting MMA for accelerating training data generation, and (3) the design and implementation of the new drop-in framework Boot that incorporates MMA in a PostgreSQL database gym.

2 BACKGROUND AND MOTIVATION

An autonomous DBMS [42, 43] optimizes itself for a target objective function (e.g., latency). It relies on *behavior models* [34] to predict system performance and identify actions to improve its configuration (e.g., knob settings), but generating the training data for such models is costly. To elaborate, we first describe how the *database gym* [30] builds models. Next, we discuss the training data generation cost and the exploitable repetition of query execution.

2.1 Database Gym

A database gym [30] is a toolkit for accelerating self-driving DBMS research by orchestrating the construction and utilization of a simulated DBMS environment focusing on training data generation. It first employs workload capture tools [18] to obtain a representative SQL trace [33]. It then uses the DBMS to simulate itself for a ML training environment that is isolated from production [30, 65], such as an offline [34, 48] or a high-availability [32] replica. Cloud vendors optimize their database fleets with similar environments [14].

The gym executes the workload on the simulation DBMS to collect *training data* about the system's run-time behavior. Such training data contains mappings from input features (e.g., query plans, optimizer estimates) to output labels (e.g., execution time, CPU/memory usage). The gym then provides the DBMS with the training data for building behavior models (i.e., learning functions of the inputs that predict outputs). The DBMS may construct models using bespoke methods (e.g., neural unit [37], operating unit [34]) or automated ML methods using a model ensemble (AutoML) [30].

After training the behavior models, the gym coordinates the DBMS's *action planning* [43] to discover better configurations. A naïve method is for the gym to apply a candidate configuration, replay the workload, and then measure whether the objective improved. However, such an approach is infeasible because the replay cost compounds with a large number of actions [30]. Therefore, the DBMS depends on its behavior models to estimate its performance under new configurations without running the workload [37, 60].

Unlike other ML domains that dismiss data collection as a one-time cost (e.g., researchers share LLM model weights because training data does not change [3]), an autonomous DBMS needs training data specific to its workload and configuration. Reusing models from other deployments is challenging because the database's contents, hardware, and system configuration [30] influence data labels. Additionally, even if the DBMS already has models, they may be invalidated because of dataset growth [37], workload drift [36], software updates [34], schema changes [50], or hardware upgrades [30]. This means that collecting the training data necessary [8] to build such models is the most time-consuming part of this process.

Table 1: Model Construction – The dataset collection time and model training time for recent ML tuning approaches.

Model	Collection (h)	Training (h)	Ratio
QPPNet [37]	300	24	0.93
MB2 [34]	9	0.3	0.97
OpAdviser [61]	40	0.05	0.99

Table 2: Repetitive Queries – The number of queries and templates in recent workloads used in database tuning.

Benchmark	Type	# Queries	# Templates	Repetition
Admissions [37]	Real	2546M	4060	627k×
BusTracker [34]	Real	1223M	334	3.66M×
MOOC [33]	Real	95M	885	107k×
TPC-H [37]	Synthetic	20000	22	909×
DSB [15]	Synthetic	11440	52	220×
Stack [35]	Synthetic	5000	25	200×

2.2 The Cost of Training Data Generation

In practice, a DBMS spends most of its time in the modeling process on executing queries to generate training data [29, 35, 39, 48, 50, 57, 58, 63]. To better understand this problem, we measured how long state-of-the-art DBMS modeling methods spend on collecting data versus training. Table 1 shows that these methods spend over 90% of their time executing queries on the DBMS.

Despite this, there have been no attempts to optimize training data generation for autonomous DBMSs. Prior work reduces instrumentation overhead [8] or employs active learning to sample workloads [32, 49], but no technique increases query execution speed. Training data generation differs from regular query execution because it concerns telemetry (e.g., timing information) and not results (i.e., returned tuples). Because a DBMS does not need to produce exact results during this process, it can speed up collection if it can infer a query's telemetry without fully executing it. This acceleration is possible because queries in a workload can be repetitive, and the operations within a query may also be repetitive.

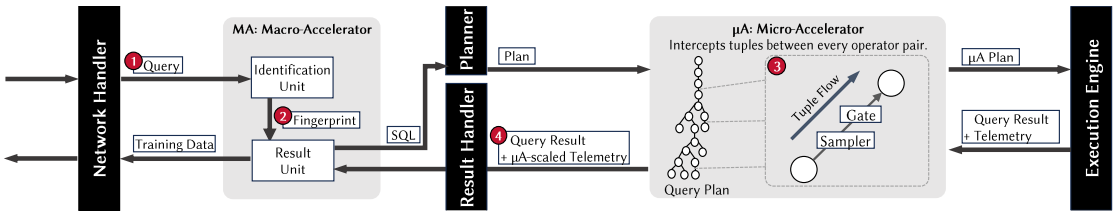


Figure 1: Architecture – An overview of Boot's internal components and execution flow. The MA component decides whether to execute a query, and the µA component accelerates the execution of a specific query.

2.3 Exploiting Workload Repetition

Table 2 demonstrates query repetition in real-world traces and synthetic workloads: the DBMS repeatedly executes the same query templates with different input parameters. Additionally, the DBMS utilizes a limited number of operator types when formulating a query plan. The repetition of an operator's behavior during execution is even more frequent than with the entire queries. Yet some repetition is necessary because the query's plan and behavior may change depending on parameters and configurations [24].

Query Repetition: What is needed is a way to determine *when* and *why* the DBMS should execute a query again so that it can execute fewer queries. In the training data context, the DBMS should re-execute a query if it has substantially different run-time telemetry from its past invocations. But the DBMS is collecting training data to bootstrap its models offline, so it has no models to predict a query's characteristics. Therefore, the DBMS must decide whether to re-execute a query based solely on its optimizer estimates and statistics, limiting the available techniques [10]. By skipping queries that do not exhibit new behavior, the DBMS executes fewer queries and thus reduces training data generation time.

Operator Repetition: There are also redundant and unnecessary parts of a query's plan that the DBMS can drop. This speedup is important for queries that take a long time to complete because of bad configurations. For example, a query that runs slowly because of missing indexes will not get faster halfway through execution. Thus, it is important to reduce the time spent executing operators after they have become predictable; that is, the DBMS should spend the majority of its time exercising "useful" operators.

Each query operator is independent as its behavior only depends on its input tuples. The DBMS relies on this independence to build models from plan telemetry (e.g., `EXPLAIN ANALYZE`) [34, 37]. Our key insight is that integrating such modeling assumptions earlier into training data generation enables early query termination. For example, a sequential scan retrieves tuples by reading from buffer pool pages; the only variation from a telemetry perspective is whether obtaining the tuple involves a disk read. Once the DBMS observes both classes of sequential scan behavior (i.e., with and without disk fetch), it does not need to keep executing the scan.

3 OVERVIEW

Our analysis above shows that data generation routines are a bottleneck for ML-based automation for DBMSs because it takes too long to execute queries. But a DBMS does not need to compute the correct result for each query in these training scenarios. Instead, the goal is to exercise the system to produce telemetry about its behavior as if it was executing in production.

```
SELECT nation, o_year, SUM(amount) as sum_profit FROM (
  SELECT n_name as nation, EXTRACT(YEAR FROM o_orderdate) AS o_year,
      l_extendedprice*(1-l_discount)-ps_supplycost*l_quantity AS amount
   FROM part, supplier, lineitem, partsupp, orders, nation
  WHERE s_suppkey = l_suppkey AND ps_suppkey = l_suppkey
    AND ps_partkey = l_partkey AND p_partkey = l_partkey
    AND o_orderkey = l_orderkey AND s_nationkey = n_nationkey
    AND p_name LIKE '%[COLOR]%'
) AS profit GROUP BY nation,o_year ORDER BY nation, o_year DESC;
```

Listing 1: TPC-H `Q9`

Given this, we present the **Boot** framework for accelerating training data generation. The high-level idea of Boot is to exploit workload repetition in two ways while being transparent to the downstream ML components and without degrading the accuracy of the ML models. The first method is to execute fewer queries on the DBMS by identifying redundant queries based on their high-level semantics and then reusing previously computed training data instead of running them again (*macro-acceleration*). For the remaining queries that the DBMS does execute, Boot injects special operators into their query plans that (1) dynamically identify redundant computations and then (2) intelligently short-circuits parts of their plans to make them complete faster (*micro-acceleration*).

As shown in Figure 1, Boot integrates into a DBMS using two modules: (1) the **Macro-Accelerator** (MA) sits in between the DBMS's network handler and query planner and (2) the **Micro-Accelerator** (µA) is embedded in the DBMS's execution engine. Boot's design does not change the DBMS's interface for tuning components. A model training framework still connects to a Boot-enhanced DBMS over standard APIs (e.g., JDBC, ODBC) to execute a workload and collect training data. As such, Boot drops into existing modeling pipelines without any code change. But since Boot circumvents the DBMS's regular query execution, it is unsuitable for production environments. The gym deploys Boot on an offline clone of the production DBMS to avoid application errors.

We now provide an overview of Boot's accelerators. For this and the detailed descriptions in Sections 4 and 5, we use TPC-H [47] query `Q9` (Listing 1) as a running example. Executing `Q9` 1000× at scale factor (SF) 100 on PostgreSQL (v15) takes 17 hrs. Enabling Boot reduces this time to 1 min with minor degradation in ML model accuracy. We defer discussing our experiments to Section 7.

3.1 Macro-Accelerator (MA)

Boot's MA module examines each query request as it arrives at the DBMS to determine whether executing it would produce novel

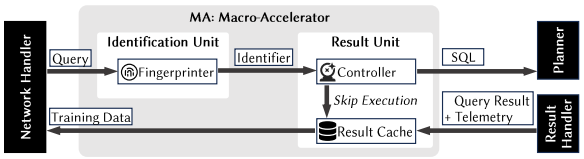
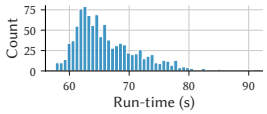


Figure 2: MA Architecture – Overview of Figure 1's MA.

Plan Id	Count	Avg Run-time (s)
P1	95	76
P2	25	66
P3	819	65
P4	61	62

(a) Plan Distribution



(b) Plan Run-time

Figure 3: Query Plan Behavior – Distribution of TPC-H Q9's plans and run-time across 1000 invocations (without Boot).

training data (i.e., increase the diversity of query plans and operators executed). Novelty is necessary to avoid overfitting models to queries that access specific tables or with particular patterns.

As shown in Figure 1, when a SQL query arrives, (1) the MA computes a fingerprint to identify whether it executed the query before. Since the MA is before the query planning stage, it computes this fingerprint on raw SQL strings. The MA strips out constants from the SQL to produce query templates (similar to prepared statements); this ensures that multiple invocations of a template using different input parameters are considered the same query [33]. In the Q9 example in Listing 1, the MA extracts the constant from the `%[COLOR]%` input parameter and replaces it with a placeholder.

Next, (2) the MA looks up the query's fingerprint in a *result cache* to determine whether the DBMS executed a similar query with the same fingerprint. This cache maps each fingerprint to a record that contains (1) the query's output and (2) the DBMS telemetry generated while executing the query. The former is necessary because some workload replay and benchmarking tools assume the DBMS returns query results with a particular schema (e.g., typed columns). If the cache does not contain a matching query, the framework sends the request along in the DBMS for processing as usual. If the MA's cache contains a match, then the MA decides whether the system will learn anything new from re-executing it or if it should skip it. For those queries that the MA decides to skip, it returns the cached result and then records that it saw the query again. We discuss the MA's policies for skipping re-execution in Section 4.2.

3.2 Micro-Accelerator (μA)

Any query that bypasses the MA module then goes to the DBMS's query planner. At this stage, Boot's μA module injects its control methods into the query plan. These components wrap the plan's operators to monitor its run-time execution constantly. When the μA module detects that an operator's behavior has stabilized, it messages the corresponding wrapper to modify the operator's tuple processing (e.g., produce less output). We designed the μA module to support any query processing model (e.g., iterator, materialized, vectorized) and both push- and pull-based execution strategies.

Using the overview diagram in Figure 1 again, (3) the μA embeds each physical plan operator (e.g., scans, joins) with a special wrapper operator that dynamically controls run-time behavior. This wrapper can adjust the sampling rate of its inner operator to change the number of tuples emitted. For example, the μA can change a scan operator to emit only 10% of the tuples it would otherwise produce. The wrapper can also completely halt an operator's execution when certain conditions are satisfied, such as if μA recognizes that it has enough training data for that operator.

Since the μA may cut off an operator's execution early, (4) Boot scales each operator's telemetry to approximate what it would have been if the DBMS executed it entirely. For example, suppose that the μA cuts off an operator after processing only 10% of its expected rows. If Boot reports the operator's elapsed time, the operator appears to process all of its rows 10× faster than it did. Therefore, Boot scales the reported time by 10× to help prevent the behavior models from underpredicting queries' execution times.

The advantage of using a wrapper-based approach that modifies a query's physical plan is that it guarantees the DBMS will generate the same plan with and without Boot enabled. Some DBMSs alter a plan when using SQL-level sampling (e.g., `TABLESAMPLE`), producing different plans and degrading the behavior models' accuracy.

After the DBMS executes the μA-wrapped plan, it sends the estimated query result and telemetry to the MA module. The MA stores these in its result cache for future invocations of similar queries. The MA and μA modules are independent: if either component is disabled, the DBMS processes data with its regular non-accelerated components instead. We show in Section 7.2 that the accelerators enhance each other's effects to obtain up to 268× speedup.

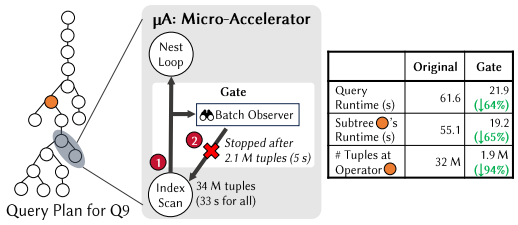
4 MACRO-ACCELERATOR (MA)

We discussed in Section 2.3 why a DBMS must rely only on its optimizer when deciding whether to re-execute a query for its telemetry. Previous work showed, however, that DBMSs cannot achieve query progress estimates that are both general and robust [10]. Therefore, the MA module employs a heuristic approach that exploits query repetition. Figure 2 shows how the MA achieves this with its Fingerprinter (Section 4.1) and Controller components (Section 4.2).

4.1 Identifying Similar Queries

The MA module's Fingerprinter assigns identifiers to queries to combine them for training data generation. Because SQL is declarative, the Fingerprinter has many ways to identify a query (e.g., exact SQL text, query template [33], plan shape [37]). However, unlike regular query execution, exact results do not matter for training data. As we now illustrate, the Fingerprinter uses a relaxed comparison method for queries to achieve higher similarity rates.

We execute 1000 instances of Listing 1's Q9 at SF 100 in 17 hr. Figure 3b shows the distribution of run-times and plans produced. Although the DBMS produced four different plans, the execution time for these plans exhibits clustering around the average run-time of the most frequent plan (P3). Furthermore, even if the DBMS never executes the slower plans, such as P1, the DBMS may still learn enough about their operators from instances in other plans and queries. For this reason, Fingerprinter groups queries based on their templates. Doing so may map different query invocations to one cache entry. But the Fingerprinter's encoding strategy does not have to be static: if a query's parametric behavior is known, the



	Original	Gate
Query Runtime (s)	61.6	21.9 (↓64%)
Subtree's Runtime (s)	55.1	19.2 (↓65%)
# Tuples at Operator	32 M	1.9 M (↓94%)

Figure 4: Gate – The Gate's architecture for Listing 1's query plan. The orange node is a hash join and the gray nodes are an index scan feeding into a nested loop join. The table shows the effect of introducing the Gate.

encoder can map each parameter regime to a different fingerprint (e.g., replace `%[COLOR]%` with `[RED,PINK]` if selectivities are similar).

The Fingerprinter also links changes in the DBMS's configuration to the validity of previous executions. For example, adding a new index to a table invalidates the history for all queries that access that table. The Fingerprinter achieves this invalidation by including a hash of the DBMS configuration in each fingerprint. This mechanism enforces an explore/exploit trade-off, although the invalidation overhead depends on the tuning technique used.

4.2 Adapting to Query Variability

After the Fingerprinter identifies whether the MA module has seen a query before, the Controller then decides whether it has seen the query enough times to skip future executions. It uses a feedback-driven adaptive algorithm based on binary exponential backoff. For every Fingerprinter entry, the Controller stores a counter (c) that tracks how often it has seen a query to skip future executions. At run time when the Controller receives a query, it decrements the corresponding counter and then does one of the following:

Skip the Query (c>0): The framework does not forward the query for execution and instead returns a cached result. The Controller exploits the training data environment to synthesize results that align with historical data. Although many possible strategies exist (e.g., average the telemetry from previously executed plans, train a model to output similar telemetry [41]), the Controller defaults to repeating the last telemetry observed for that identifier (i.e., naïve forecasting [23]). This approach is fast, avoids the overhead of storing historical plans, and sidesteps environment drift issues.

Execute the Query (c=0): The Controller forwards the query for execution and analyzes the resulting telemetry. If the plan's run-time falls within two standard deviations of the historical mean, the Controller considers the new execution similar and exponentially increases the counter until a threshold. Otherwise, it resets the counter and clears the corresponding execution history.

We illustrate the skipping algorithm by supposing that `Q9` always takes its median run-time (64.8 s) with a threshold of 100 skips. The number of times that the Controller skips the query in between executions is $[1, 2, 4, 8, 16, 32, 64, 100, 100, 100, \ldots]$, dropping the time for 1000 executions of `Q9` from 18 hr to 16 min. Should a `Q9` invocation exhibit new behavior, the Controller resets the skipping sequence to sample future instances more frequently. This behavior allows the Controller to adaptively decide a per-query *workload size* for training data collection (i.e., number of executions).

We use run-time to measure similarity to avoid storing and comparing against all executed plans. For each Fingerprinter identifier, the Controller maintains around 10 KB of state: (1) streaming Welford mean [53], (2) query plan, and (3) counter. Table 2 shows that workloads with millions of queries reduce to a few thousand plans, so MA's typical total storage overhead is tens of MBs.

5 MICRO-ACCELERATOR (μA)

Skipping queries with macro-acceleration allows the DBMS avoid executing redundant queries. But the DBMS still needs to execute each unique query at least once before it can cache its results, which is still prohibitively expensive. To handle such invocations, we present micro-acceleration techniques that exploit operator repetition (Section 2.3) to generate telemetry faster. As with the MA, the DBMS cannot provide optimal guarantees for using operator repetition [10] to accelerate telemetry production. Instead, the μA builds on operator progress estimation [27] to achieve its speedups.

5.1 Stopping Repetitive Operators

The μA module exploits how query processing is performed at an operator's granularity (e.g., Volcano [19] model's `GetNext()`). For every operator, the μA's Gate component wraps this function to override tuple flow and measure repetition by tracking the rate of output tuple production.

The challenge with this tuple flow approach is that the DBMS cannot accurately measure telemetry for individual tuples. The difficulty arises from timer overhead and resolution: an operator might take less than 1 μs per tuple. Therefore, the Gate collects telemetry on batches of tuples instead. For a given operator, it monitors the time taken to produce each output row and starts a new tuple batch when both of the following conditions hold: (1) the current batch contains at least 10% of the optimizer's estimated number of output tuples, and (2) the current batch's accumulated time is at least 1 s. The Gate considers the operator repetitive once the total time for its latest batch falls within two standard deviations of the historical mean. When the operator triggers this threshold, the Gate stops the operator from processing new input tuples. We present a sensitivity analysis for these parameters in Section 7.7.

We demonstrate the Gate's effects on a query plan for Listing 1's `Q9`. We first focus on the highlighted pair of operators in Figure 4 that depicts an index scan under a nested loop join. (1) The index scan sends tuples to the Gate, which defaults to allowing tuples to pass through to the nested loop join. It also adds the tuple's telemetry to its current batch of tuples. When it creates a new batch, it checks whether it is similar to the last 20% of batches. If so, (2) it stops admitting future tuples from the index scan. For this example, this stops after observing 2.1 M out of a possible 34 M tuples, reducing the time spent in the index scan from 33 s to 5 s.

5.2 Sampling for Output Reduction

Whereas the Gate stops an operator's execution, the DBMS may only need to reduce the operator's output. Because query results do not matter for training data, the μA module samples each operator's output tuples to reduce run-time up in the query plan while maintaining representative behavior. It achieves this by installing a Sampler component on each operator's output tuple flow.

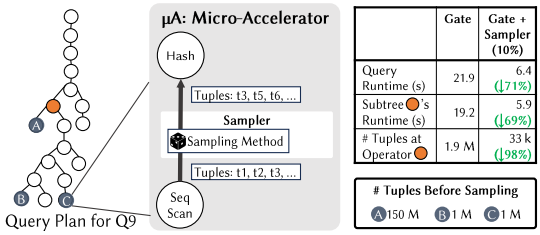


Figure 5: Sampler – The Sampler's architecture for Listing 1's query plan. The orange node is a hash join and the gray nodes are sequential scans. The table shows the effect of introducing the Sampler in addition to the Gate.

SQL already provides methods for reducing the number of tuples that certain operators produce. For example, `LIMIT` reduces the number of output tuples, and `TABLESAMPLE` reduces the cardinality of base relations. However, both of these methods are insufficient for our needs. `LIMIT` only applies to the query plan root, so the DBMS may have already done the work to compute the query (e.g., a long-running query that outputs a single tuple). The problem with `TABLESAMPLE` is that the DBMS may select a different query plan. We observed degenerate cases where a query's execution time in PostgreSQL went from 1 min to two days just by adding `TABLESAMPLE`! None of these methods work because the sampling must be hidden from the optimizer to avoid a change in plan. Moreover, the DBMS should be able to sample any operator in its plan (as opposed to only the root or base relations). Therefore, the Sampler exposes a similar interface as `TABLESAMPLE` but at the operator level.

To show how the Sampler effects Listing 1's `Q9`, we enable it using Bernoulli sampling at 10% on the operators that produce the most tuples (i.e., the sequential scans labeled as `A`, `B`, and `C` in Figure 5). Although it only sampled from these three operators, the effects on the rest of the query plan are significant: the orange node (a hash join) goes from producing 1.9 M tuples to 33 k tuples, and the overall query run-time decreased further from 21.9 s to 6.4 s.

6 ENGINEERING

We now describe how we integrated Boot into a database gym as an extension for the PostgreSQL DBMS.

MA: We implement MA by hooking into the DBMS's traffic control layer. Because PostgreSQL uses the process-per-worker model, workers cannot easily share their execution history (i.e., queries executed by one worker are not seen by another). Although PostgreSQL coordinates state across processes using shared memory, we avoid this because it interferes with query processing. Instead, MA manages its cache in an external key-value store (Redis).

μA: We implement μA by wrapping PostgreSQL's operators. Specifically, μA overrides every operator's `GetNext()`. While PostgreSQL executes a query, this override constantly analyzes the plan's telemetry to decide what to do. It performs its tasks by further swapping out the function pointer for the wrapper at run-time.

We initially built Boot as a standalone middleware that intercepted queries and polled the DBMS for its currently executing plans. We found that PostgreSQL's existing interfaces did not expose sufficient control. This model suffers from non-determinism and creates more work for the DBMS. Integrating Boot directly into the DBMS improves determinism and efficiency.

7 EVALUATION

We now evaluate the capabilities of Boot to reduce the training data generation times for autonomous DBMSs. For our analysis, we integrate Boot into the PostgreSQL (v15) DBMS. We deploy the DBMS on an Ubuntu 22.04 LTS server with 2×20–core Intel Xeon Gold 5218R CPUs, 188 GB DRAM, and Samsung PM983 SSD. We optimize the system's configuration with PGTune [4].

We define our workloads and experiment configuration in Section 7.1. We then perform an end-to-end high-level analysis of Boot's modules in Section 7.2. Next, we identify inefficiencies in training data collection in Section 7.3. We describe how Boot addresses these inefficiencies with its MA in Section 7.5 and μG in Section 7.7. Lastly, we investigate sampling as a technique to further improve Boot's capabilities in Section 7.8.

7.1 Workloads

We use the database gym [30] to orchestrate the execution of the following workloads. Figure 7 shows the run-time distributions.

- **TPC-H:** This benchmark models a business analytics workload with eight tables and 22 query templates [47]. We chose this benchmark to represent a workload with a uniform data distribution. We use dbgen [1] to produce a total of 22k queries and execute them on scale factors 10 (~19 GB with indexes) and 100 (~192 GB with indexes).
- **DSB:** This is Microsoft's extension of the TPC-DS [46] workload that introduces additional challenges (e.g., complex data distributions, join patterns, skew), with a total of 25 tables and 52 query templates [15]. We use the official generator to produce a total of 10.4k queries and execute them on scale factors 1 (~5 GB with indexes) and 10 (~47 GB with indexes).
- **JOB:** This benchmark uses IMDB and aims to stress the query optimizer's ability to pick a good join order [28]. It represents the worst-case workload for Boot (i.e., minimal repetition, small workload size, small dataset, short-running queries). There are 113 query instances and 21 tables (~8.5 GB with indexes).

We set a per-query timeout of 5 min in all experiments. A timeout is necessary in a training data environment as the DBMS is trying to discover better configurations (i.e., it may not be optimally configured). We discuss timeouts further in Section 7.3.

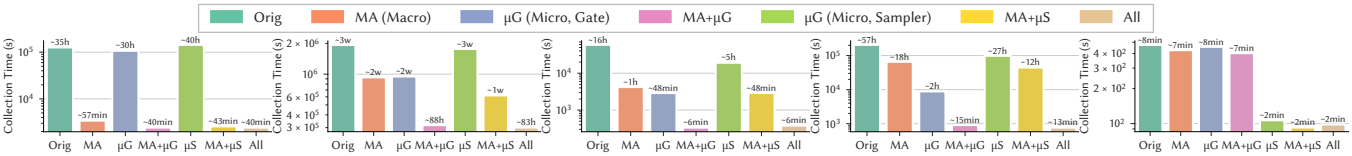
To build behavior models using these workloads, the DBMS splits its training data into a train and test dataset as follows: for TPC-H and JOB, the DBMS trains on 80% of the seeds and tests on the remaining 20% [37]; for DSB, the DBMS uses separate seeds for train and test [15]. Next, the DBMS creates behavior models with AutoGluon [17], a state-of-the-art automated ML framework that automatically searches over hyperparameters and network architectures to create a model ensemble (e.g., gradient-boosted trees [13, 25], random forests, linear models, neural networks).

7.2 Speed-up vs. Accuracy Measurements

We first evaluate Boot's ability to accelerate a DBMS's training data generation process and how it affects the quality of the training data. Since there are no known techniques for measuring the quality

(a) TPC-H (SF 10) (b) TPC-H (SF 100) (c) DSB (SF 1) (d) DSB (SF 10) (e) JOB

Figure 6: Query Run-time Distribution – Breakdown of elapsed time for each workload (without any acceleration).



(a) TPC-H (SF 10) (b) TPC-H (SF 100) (c) DSB (SF 1) (d) DSB (SF 10) (e) JOB

Figure 7: Collection Time – The time to generate training data with different modules of Boot active (lower is better).

of the generated data [5], we use behavior model accuracy as a surrogate metric instead. Thus, this experiment highlights the trade-off between reducing the execution time of queries versus producing models that accurately reflect the DBMS's internal operations.

We run the workloads using Boot under seven configurations: (1) the default DBMS without acceleration (**Orig**), (2) only the MA module enabled (**MA**), (3) only the µA's Gate module (**µG**) enabled, (4) the MA and µA's Gate modules enabled (**MA+µG**), (5) only the µA's Sampler module (**µS**) enabled, (6) the MA and µA's Sampler modules enabled (**MA+µS**), and (7) all modules enabled (**All**). These configurations demonstrate the effect of reducing the number of queries (the MA module) and executing each query faster (the µA's modules) in different combinations. We structure our discussions below around the independent (1) MA, (2) µG, and (3) µS modules, followed by (4) the combined configurations (MA+µG, MA+µS, All).

We measure the time that the DBMS takes to generate telemetry for all the queries in a workload (i.e., collection time) as the end-to-end query latency. To measure model accuracy, we adopt two metrics from existing work: (1) *absolute error* [34] and (2) *factor error* [37]. Given a query q that has an actual latency $A(q)$ and a model M that predicts q's latency as $M(q)$, the absolute error is given by $|A(q) - M(q)|$ and the factor error is defined as $R(q) = \max\left(\frac{A(q)}{M(q)}, \frac{M(q)}{A(q)}\right)$. To understand whether these errors are caused by the models under- or over-predicting, we also visualize the error distributions of each model as $M(q) - A(q)$.

Collection Time: Figure 7 shows the collection time for the training data configurations across the workloads. We observe that the accelerators always speed up collection time, however, the extent of their benefit varies depending on workload characteristics.

The results in Figures 7a and 7c show that MA achieves a 13–37× speedup for the smaller SF workloads. However, increasing the SF reduces the speedup to 2–3× in Figures 7b and 7d. The first reason for this is that because MA does not increase the speed of query execution, it cannot help queries that timed out. Figures 6b and 6d shows more occurrences of such queries at the 300s mark. The second reason is that MA is more effective for workloads with lower variability in their query run-time. The standard deviation (std) of query run-time is lower in Figures 7a and 7c (8.42–29.9 s) and higher in Figures 7b and 7d (70.4–89.3 s). A reduction in std benefits MA because it decides whether to re-execute a query based on the similarity of its run-time to its previous executions. We next observe that the MA achieves only 1.1× speedup in Figure 7e for JOB. We expect this result because most of JOB's queries are only executed once (i.e., minimal repetition at the query level). We investigate the MA further in Sections 7.4 and 7.5.

Figure 7 also shows that while µG improves collection time relative to Orig, the increase is not as much compared to MA. For DSB, µG obtains a speedup of 20.3–23.2×, compared to only a 1.18–2.03× speedup for TPC-H. Such improvement depends on workload complexity because larger query plans introduce more opportunities for micro-acceleration (e.g., more leaf nodes, longer running operators). For example, µG sped up DSB's `query001` by 14× (117 s to 8 s), but only achieves a 2× speedup for most TPC-H queries. We next observe that Figure 7d is the only instance where µG is more effective than MA. In addition to query complexity, this is because µA enables the DBMS to completes queries that would otherwise time out (e.g., all `query032` invocations timed out after 300 s with MA but complete within 10s with µG). Lastly, Figures 7a and 7e show little improvement with µG alone. Orig's configurations show that the DBMS executed the 22k TPC-H queries in 1 day and 113 JOB queries in 8 min, which means the average query duration is approximately 5 s. Most queries did not run long enough to activate µG; we test more aggressive hyperparameters in Section 7.7.

These results show that the Sampler alone provides limited improvements on the collection time, with speedups ranging from 0.88–1.09× for TPC-H and 2.09–3.05× for DSB. The high degree of query repetition mutes the Sampler's benefits for both workloads, and its sampling overhead makes TPC-H (SF 10) even slower. But JOB's low query repetition makes the Sampler the only effective technique for it. The Sampler obtains a 4.44× speedup because JOB is dominated by short index scans across multiple queries. These scans are too short for the Gate to accelerate, but a random sample of their tuples reduces the work while maintaining representative behavior. Real workloads are much more repetitive than JOB (e.g., 60% of Redshift's daily queries are exactly the same [56]), so other workloads are more representative of Boot's performance.

The MA boosts the µS's limited efficacy by eliminating query repetition, with the MA+µS configuration obtaining speedups of 3–49× on TPC-H, 4–20× on DSB, and 5× on JOB. In comparison, the MA+µG combination obtains most of Boot's benefits for most

Table 3: Factor Error – The factor error of the model predictions divided into buckets (lower is better).

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	94%	6%	0%	0%
MA	50%	50%	0%	0%
µG	82%	18%	0%	0%
MA+µG	43%	55%	1%	0%
µS	25%	52%	22%	0%
MA+µS	4%	60%	33%	3%
All	22%	49%	28%	1%

(a) TPC-H (SF 10)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	82%	18%	0%	0%
MA	44%	51%	6%	0%
µG	10%	38%	33%	19%
MA+µG	9%	33%	37%	21%
µS	18%	69%	14%	0%
MA+µS	11%	69%	20%	0%
All	4%	41%	30%	25%

(b) TPC-H (SF 100)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	18%	45%	18%	19%
MA	16%	61%	17%	6%
µG	15%	64%	13%	8%
MA+µG	16%	57%	17%	11%
µS	21%	59%	15%	5%
MA+µS	18%	57%	16%	9%
All	17%	57%	15%	10%

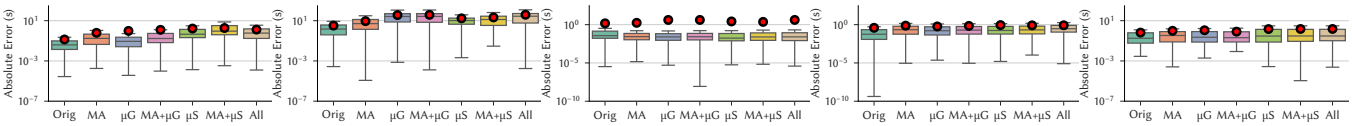
(c) DSB (SF 1)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	52%	42%	4%	1%
MA	14%	64%	17%	4%
µG	18%	71%	10%	2%
MA+µG	15%	64%	16%	4%
µS	15%	63%	19%	3%
MA+µS	15%	59%	21%	5%
All	6%	53%	35%	7%

(d) DSB (SF 10)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	4%	63%	21%	12%
MA	5%	36%	39%	19%
µG	5%	35%	44%	16%
MA+µG	6%	55%	24%	15%
µS	3%	6%	12%	80%
MA+µS	1%	3%	4%	93%
All	1%	0%	1%	98%

(e) JOB



(a) TPC-H (SF 10) **(b) TPC-H (SF 100)** **(c) DSB (SF 1)** **(d) DSB (SF 10)** **(e) JOB**

Figure 8: Absolute Error – The absolute error of models that are trained on the individual datasets (lower is better). The red circle shows sample mean and the whiskers extend to 1.5 interquartile range.

workloads. It achieves speedups of 6–52× on TPC-H, 175–226× on DSB, but only 1.17× for JOB. The Gate is more effective than the Sampler because it dynamically stops the rest of an operator's execution, whereas the latter applies a fixed sampling percentage to an operator's output. With all accelerators active, All achieves the best of both worlds and obtains speedups of 6.2–52×, 154–268×, and 4.8× on TPC-H, DSB, and JOB respectively. Figure 7d also demonstrates that the modules enhance each other: while MA reaches 3×, µG obtains 23× and µS gets 2× speedup, all modules together obtain 268×. This combination benefits queries that are long-running or often time out. Using DSB `query032` as an example, suppose the DBMS invokes this query 100 times but they all time out at 300 s. MA does not help queries that time out, so MA takes 300×100 = 30000 s to complete. µG next reduces the query's runtime to 10 s, so µG takes at most 10×100=1000 s. Moreover, because the query now completes, MA executes exponentially fewer queries (i.e., 6 instead of 100) and All only takes at most 10×6 = 60 s (over 500× speedup) even before sampling. We examine the time spent in each operator and discuss additional timeout nuances in Section 7.3.

Absolute Error: Figure 8 shows the behavior models' absolute error when using training data from each configuration. The mean absolute error (MAE) of MA's models ranges from 1.1–4.6× that of the Orig models. The MA models are comparable to the Orig models because their telemetry was produced under similar conditions (i.e., MA only decides whether to execute a query). In contrast, the µG models are worse than Orig because µG terminates execution early and scales the telemetry. This reduced accuracy is reflected in µG's worse MAE of 7–11× for TPC-H and 1.7× for JOB. The latter is less affected because the Gate did not activate as much. For DSB, Figures 8c and 8d show 2.7× worse error at a smaller SF, but this error improves to 1.5× as the scale increases because Gate allows the DBMS to learn from queries that would otherwise time out.

Compared to the Orig models' MAE, the µS models are 5–12× worse for TPC-H and 1.7–2.3× worse for DSB and JOB, which is comparable to µG. The Gate's early operator termination have similar effects to sampling when it comes to MAE (i.e., observing a subset of execution is similar to sampling the entire execution).

Across all workloads, combining the MA with individual µA modules produces similar errors to the µA module alone. This result is because the MA does not modify query execution itself. When all MA and µA modules are activated, the All models exhibit similar error to the µG models. Compared to the Orig models, the MAE ranges from 9.4–11.9× for TPC-H, 2.1–2.7× for DSB and 2.29× for JOB. To contextualize these numbers, we sum the prediction errors for `Q1` in Figure 8b. Executing these `Q1` instances takes 3.7 hr. Consistent with other state-of-the-art models [34, 37], the Orig models are only off by 5 min, whereas the All models are off by 35 min. Although this is 7× the error, the speedup to obtain All's TPC-H models makes it an acceptable tradeoff (i.e., the DBMS obtains its first models in 4 days instead of 3 weeks, and these models predict 3.1 hr when the actual time is 3.7 hr for all `Q1` executions).

Factor Error: The results in Table 3 show how the model prediction error is distributed across queries in the form of factor error (i.e., the multiple that a query's predicted latency is incorrect).

For almost all TPC-H queries, the error for MA's models is at most 2× because of the database's uniform data distribution. Such uniformity means that invocations of the same query but using different input parameters have similar performance. Therefore, even though MA executes fewer queries, the ones that it does execute are enough. However, both MA and Orig models have worse error for DSB and JOB, because these workloads are more complex than TPC-H. This result is consistent with previous work [37] that found that some queries are harder to model than others. But the difference in the median factor error across all workloads is minimal: MA's error is 1.10–2.22×, and Orig's error is 1.02–1.54×.

Table 3 also shows that µG's models have comparable factor error to MA, with the exception of Table 3b where it is worse. This increase in error is because PostgreSQL's optimizer underestimates operator selectivities [28]. For µG's models, the median error ranges from 1.05–2.58×, which is up to 2.3× worse the Orig models. In comparison, µS's models have median errors that range from 1.31–1.44× for every workload except JOB, which is 27×.

The MA+µG models have median factor errors that range from 1.12–2.52×, which is up to 2.5× worse than Orig. With the exception of JOB, we observe similar results for the MA+µS models: the median error ranges from 1.39–1.71×, which is up to 1.4× worse than Orig. The MA+µS models have 5× worse median error for JOB than the Sampler alone, which is up to 78× worse than Orig. Because JOB

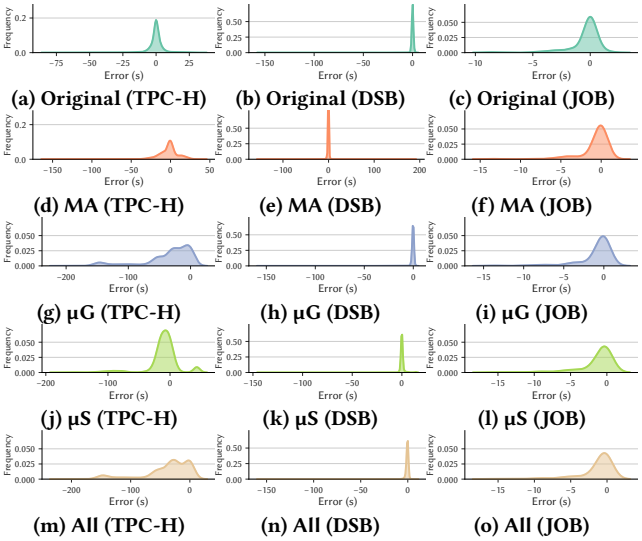


Figure 9: Error Distribution – The distribution of model error for TPC-H at SF 100, DSB at SF 10, and JOB (closer to 0 is better). For example, an error of -20 s means the model predicted 20 seconds under the actual value.

only has 113 queries, activating the MA removes about 10% (11) of the queries. This missing data is more important in JOB because most queries produce little meaningful data (i.e., the run-time in most operators rounds to zero milliseconds) and the low repetition makes it difficult to learn from other queries.

Excluding JOB, the All models have median factor errors ranging from 1.25–2.43×, which is up to 2.4× worse than Orig. JOB observes a median error of 165×, which is 108× worse than the Orig models. However, JOB's queries are shorter, limiting the practical impact of such mispredictions (e.g., query `10a`'s run-time of 0.226 s is underpredicted as 0.0018 s). We investigate the reason in the discussion on error distribution below. Recent research shows that less accurate models are competitive for tasks like index recommendation [62].

Error Distribution: Figure 9 shows the distribution of prediction errors for each training data generation configuration on the two larger datasets. The errors for Orig and MA are both uniform peaks with long tails centered at zero (i.e., most queries experience low error), where macro-acceleration has slightly more error. We expect this result in the absence of micro-acceleration as it matches existing work [34, 37] and we use newer modeling techniques.

Figure 9g shows the µG models consistently underpredict query latency for a subset of queries. This occurs more for long-running queries because µG expedites their execution and then scales up the telemetry based on optimizer estimates. However, PostgreSQL's optimizer underestimates the result size of multi-join queries [28]. This means that as the optimizer's estimation algorithms improve [59], µG's accuracy improves as well. µS suffers from the same underestimation problem, though its errors are more normally distributed from sampling. All inherits the same underprediction issue from µG and µS, with a slight increase to its error from MA as well.

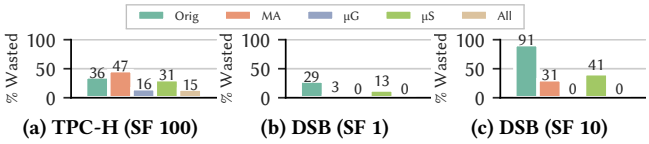


Figure 10: Wasted Collection Time – The percentage of time wasted executing queries that the DBMS aborts after 5 min. We omit TPC-H (SF 10) and JOB as they do not time out.

7.3 Reducing Aborted Queries

The collection times in the above section include queries that the DBMS aborts on timeout. Because the DBMS only produces telemetry for completed queries, time is wasted when it executes queries that will abort. Boot reduces this waste by (1) MA probabilistically avoiding such queries and (2) µG allowing queries to finish that would otherwise abort, which we describe in more detail below.

We now revisit the results from Figure 7 by identifying the collection time spent on queries that the DBMS aborts. This analysis reveals the wasted work in the DBMS's training data collection and the extent to which Boot's modules reduce such waste.

Figure 10 shows the percentage of collection time that was wasted work. With Orig, the DBMS spends 36% (193 hr) and 29–91% (4.6–52 hr) of its collection time on aborted queries for TPC-H and DSB, respectively. These timeouts are caused by a small fraction of query invocations from the workload: 2310 (10.5%) for TPC-H SF 100, 56 (1%) for DSB SF 1, and 626 (12%) for DSB SF 10. That is, less than 15% of queries are responsible for 29–91% of the collection time despite producing no telemetry. The DBMS cannot avoid these queries because it is impossible to know how long a query will take before running it. For example, the run-time for DSB's `query102_spj` ranges from 1–26 s based on its parameters.

The MA reduces waste when it substitutes invocations that will abort with historical executions that did not. But Figure 10a shows that the MA is unable to reduce TPC-H's waste. Most queries have only a small fraction of their invocations perform poorly relative to the mean (e.g., only 10% of Figure 3a's invocations have slower plans). Figures 9d and 9e show that the DBMS obtains enough data for its models without executing these poor plans. However, the degree of variability between TPC-H's plans was too low for the MA to reduce waste (i.e., the good and bad plans had similar abort behavior). In comparison, DSB's plans had high variability because of its skew and complexity, allowing the MA to reduce waste by 26% at SF 1 (Figure 10b) and 60% at SF 10 (Figure 10c). The MA obtains less improvement at the higher SF because more query invocations time out, preventing it from avoiding aborts.

The results in Figure 10a show that the µG reduces wasted work by 20% for TPC-H and eliminates wasted work entirely for DSB. The µG reduces waste through accelerating long-running queries that would otherwise abort. It does this by identifying operator repetition (see Section 2.3) and stopping such operators early, which we investigate further in Section 7.6.

Across all workloads, All reduced Orig's wasted work to 32% of the total time (250 hr to 80 hr). This reduction accounted for 30% (170 hr of 560 hr) of All configuration's absolute time improvement.

Table 4: Factor Error – The factor error of the model predictions divided into buckets for MA.P (lower is better).

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	94%	6%	0%	0%
MA	50%	50%	0%	0%
MA.P	92%	8%	0%	0%

(a) TPC-H (SF 10)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	82%	18%	0%	0%
MA	44%	51%	6%	0%
MA.P	76%	21%	3%	0%

(b) TPC-H (SF 100)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	18%	45%	18%	19%
MA	16%	61%	17%	6%
MA.P	36%	47%	13%	4%

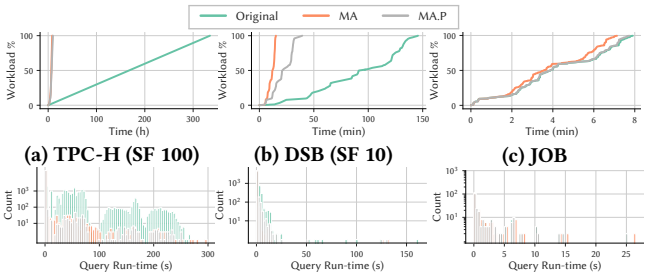
(c) DSB (SF 1)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	52%	42%	4%	1%
MA	14%	64%	17%	4%
MA.P	40%	48%	10%	3%

(d) DSB (SF 10)

	Factor Error			
	1.1 ≤	[1.1,2]	[2,5]	≥ 5
Orig	4%	63%	21%	12%
MA	5%	36%	39%	19%
MA.P	5%	40%	36%	19%

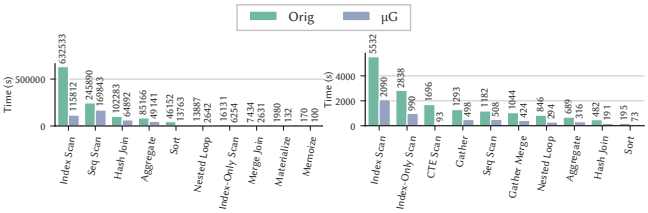
(e) JOB



(a) TPC-H (SF 100) **(b) DSB (SF 10)** **(c) JOB**

(d) TPC-H (SF 100) **(e) DSB (SF 10)** **(f) JOB**

Figure 11: Exponential Speedup – Workload completion rate and run-time distributions under MA, excluding timeouts.



(a) TPC-H (SF 100) Timing **(b) DSB (SF 10) Timing**

Figure 12: Operator Time Breakdown – The time spent in each operator (lower is better), excluding queries that timed out in Orig. We show the top 10 operators that Orig executes.

7.4 MA: Different Encoding Strategies

Next, we analyze whether more advanced fingerprinting that includes query plan changes improves the MA's effectiveness. We run the experiments from Section 7.2 using a modified Fingerprinter that appends the query plan hash [37] to the query template (**MA.P**), contrasting against the default Fingerprinter (**MA**) and default DBMS configuration (**Orig**).

Comparing MA.P to MA, we observed similar collection times for DSB and JOB. The collection time of TPC-H increased from 57 min to 2 hr for SF 10 and decreased from 2 weeks to 1 week for SF 100 (i.e., MA.P is 2× faster at smaller scale factors and 2× slower at larger scale factors). Both the speedup and slowdown are caused by the MA's adaptive exponential skipping algorithm. The MA resets a query template's skip counter when a query has different behavior. The MA.P avoids this and obtains speedups at SF 10, but it maintains more skipping sequences (i.e., skip counters are per plan instead of per query). Because the skipping is exponential, MA.P skips fewer executions, resulting in SF 100's slowdown.

The results in Table 4 show that the factor and absolute error improves for all benchmarks with the MA.P. However, unlike the protocol-level MA, the MA.P requires invoking the optimizer and pausing execution after plan generation to check whether to continue execution or to return a cached result. Because optimizer calls are a bottleneck [7] and using plan data increases engineering complexity for modest accuracy benefits, we use MA in our evaluation. We defer more advanced fingerprinting [56] to future work.

7.5 MA: Executing Fewer Queries

We now evaluate MA's effect on the workload completion rate as a function of the collection time. This analysis shows whether MA obtains its speedup across all queries or only a handful of queries. We analyze the data from Section 7.2 by measuring the number of queries completed (Workload %) as a function of elapsed time. We also plot the run-time distribution for both Orig and MA.

Figures 11a and 11b show that MA's workload completion rate is higher than Orig. Recall that the only difference between these configurations is the number of queries executed: when MA has enough training data, it intelligently skips an exponential number of queries between executions. Therefore, MA's completion improvement also scales exponentially because it executes exponentially fewer queries. Figures 11d and 11e shows this effect on the distribution of query run-time. The DBMS executes queries with the same complexity under both configurations (i.e., similar histogram shapes), but the MA reduces the number of executions for each query (i.e., shorter heights on every bar). Figure 11b also shows the reduction in exponential skipping that Section 7.4 describes.

7.6 µG: Stopping Operators Early

We revisit µG's ability to reduce collection time by measuring the DBMS's duration in each operator. Our analysis seeks to identify which operators that contribute significantly to the collection time and the extent to which Boot speeds them up.

We perform a fine-grained analysis of the data from Section 7.2 by breaking down the collection times in Figures 7b and 7d for the Orig and All configurations into individual operators. We found that operator timings reported by PostgreSQL are inaccurate [2]. Therefore, we instrument every operator with additional timers [8].

Figure 12 shows the distribution of time spent in each operator for both Orig and All configurations. To ensure a fair comparison, we omit timed-out queries. The distribution of operator time is highly skewed, meaning that a few operators are responsible for most of the time spent. We observe that for both workloads (1) the µA reduces the run-time of every operator, (2) scans dominate the original query run-time, and (3) the µA achieves the highest absolute speedups on the longest-running operators.

Figure 13a shows that the top three absolute speedups for TPC-H are index scans (175 hr to 32 hr, 5.4×), sequential scans (68 hr to 47 hr, 1.4×), and hash joins (28 hr to 18 hr, 1.5×), whereas for DSB (Figure 13b) they are index scans (92 min to 35 min, 2.6×), index-only scans (47 min to 16 min, 2.8×), and CTE scans (28 min to 90 s, 18×). The disk-based operators (e.g., scans) speed up because µG's early stopping reduces the I/O. The in-memory operators (e.g., hash join, aggregate) in Figure 13a obtain their speedup when µG reduces

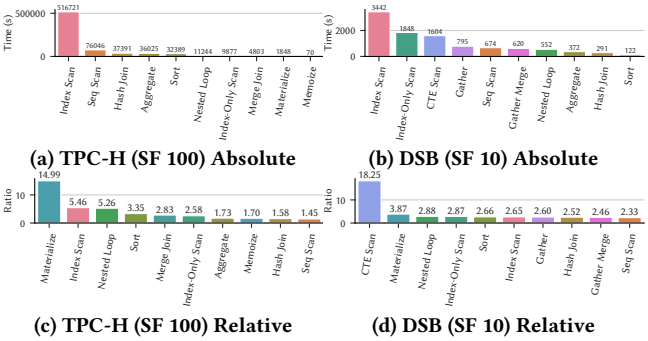


Figure 13: Speedup Analysis – Each operator's speedup because of µG in Figure 12 (higher is better).

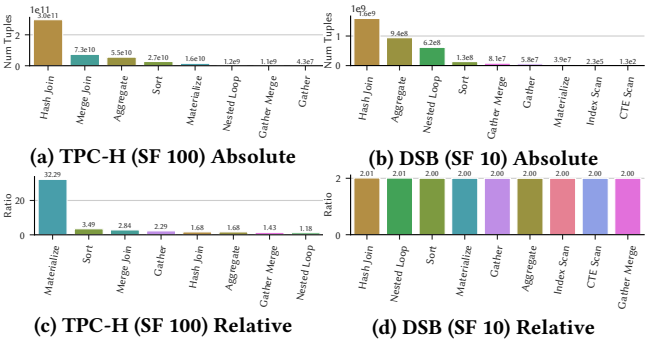


Figure 14: Tuple Reduction – Each operator's reduction in tuples processed because of µG in Figure 12 (higher is better).

downstream tuples. Figure 14a shows that hash joins are 1.5× faster for TPC-H because they process $3{\cdot}10^{11}$ fewer tuples. The reduction in downstream tuples (Figures 14c and 14d) also explains each operator's relative speedup (Figures 13c and 13d). The materialize operator has high speedup in TPC-H (14.99×) because it processed 32.29× fewer tuples. Because µG dynamically decides when to stop an operator, the speedup of each operator is difficult to predict. Most speedups and reductions in downstream tuples are 1.4–3.5×.

7.7 µG: Accelerating Short Workloads

We now revisit the experiment from Section 7.2 in which we observed µG achieving a speedup over Orig for every workload except TPC-H (SF 10) (Figure 7a). This analysis aims to discover why micro-acceleration does not work as well in some scenarios.

We conjectured that micro-acceleration is less effective in Figure 7a because the DBMS completed queries too quickly to detect operator repetition. Recall that the µG detects repetition by batching tuple telemetry, which it then uses to determine when an operator should stop processing new input. For every operator, µG creates a new batch whenever the current batch's (1) processing time exceeds 1 s and (2) the number of tuples is greater than the optimizer's estimated tuple count by 10%. µG then stops the operator if the new batch's timing is within two std of the mean on historical data. Therefore, three hyperparameters control µG's detection of operator repetition: (1) processing time, (2) optimizer %, and (3) std.

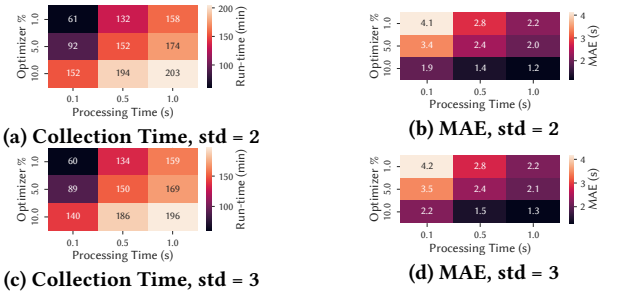


Figure 15: Batching Sensitivity – Collection time and MAE as we vary µG's hyperparameters on TPC-H (SF 10), 100 queries.

We perform a sensitivity analysis on µG's hyperparameters using 100 queries from TPC-H (SF 10). We measure the collection time and MAE as we sweep the processing time (0.1 s to 1 s), optimizer % (1% to 10%), and std (2 to 3). These ranges potentially allow for more opportunities for micro-acceleration to optimize this workload.

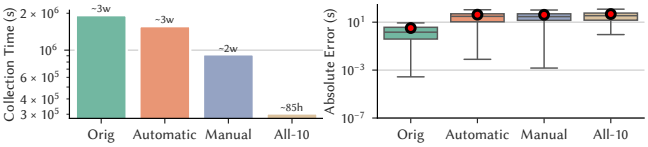
Figure 15 visualizes the three hyperparameters and their target variable (i.e., collection time or MAE) as heatmaps, where darker is better. Figures 15a and 15c show the collection time (darker is faster), whereas Figures 15b and 15d show the MAE (darker is lower). These heatmaps show that increasing the std has no appreciable effect on the collection time and MAE as the top row (std 2) has near-identical values to the bottom row (std 3). Varying the processing time and optimizer % achieves up to a 3× speedup at the cost of up to 3× higher error. For example, reducing the processing time from 1 s to 0.1 s improves the collection time by 1.33× (203 s to 152 s) but increases the MAE by 1.58× (1.2 s to 1.9 s). Similarly, reducing the optimizer % from 10% to 1% improves time by 2.5× (152 s to 61 s) but increases the MAE by 2.2× (1.9 s to 4.1 s).

This result verifies our hypothesis from Section 7.2 that µG is ineffective in Figure 7a because the queries' average run-time is too short. It also means tuning µA's hyperparameters allows for more aggressive tradeoffs between speedup and error. However, because an optimal progress estimate is impossible, we cannot prescribe a batch size that evenly divides an operator's progress. µG's reduced efficacy is due to the small dataset size, but Boot's combined techniques are still effective. Since Boot already achieves large speedups, we use conservative default settings of 1 s processing time, 10% optimizer cutoff, and two std for all experiments.

7.8 Output Sampling Analysis

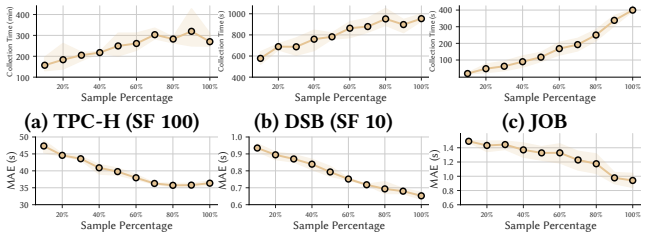
We next investigate alternative sampling techniques and the Sampler's tradeoffs between collection time and error.

We evaluate Boot's sampling against the DBMS's built-in table sampling (through the `TABLESAMPLE` SQL modifier) even though Section 5.2 outlines qualitative reasons against such an approach. We collect training data for TPC-H (SF 100) as it represents the best case for `TABLESAMPLE` with its uniform data distribution. We evaluate four training data generation configurations. First is the baseline with neither sampling nor Boot active (**Orig**). The next configuration uses `TABLESAMPLE` at a rate of 10% on all tables in the query (**Automatic**). Because we found `TABLESAMPLE` to cause problems with PostgreSQL's optimizer and generate slower plans for some queries, we also manually modify every SQL query to



(a) Run-time (b) Absolute Error

Figure 16: Sampling Baseline – Collection time and absolute error with `TABLESAMPLE` and Boot for TPC-H (SF 100). The red circle shows sample mean. The whiskers extend to 1.5 IQR.



(a) TPC-H (SF 100) (b) DSB (SF 10) (c) JOB

(d) TPC-H (SF 100) (e) DSB (SF 10) (f) JOB

Figure 17: Sampling Rate – The collection time and MAE with MMA enabled, sampling sequential scans at different rates. We exclude timed out queries for fairness (4.55% of TPC-H).

add the `TABLESAMPLE` clause only for the relations where there is a benefit (**Manual**). Lastly, we collect training data with Boot using a 10% sample rate for all scan operators (**All-10**).

Figure 16 shows that both `TABLESAMPLE` methods reduce collection time by up to 2.1×, though this requires the user to rewrite every query template by hand (Manual). All-10 achieves a 3× faster collection time than both sampling methods with comparable error rate (MAE increases from 12.7× to 14.7×). As described in Section 5.2, sampling scans improves the collection time by reducing the number of tuples that the DBMS processes downstream. Moreover, when Sampler only targets less selective operators (e.g., sequential scans), it reduces the risk of eliminating tuples that were uniquely important for exercising DBMS behavior and therefore limits the effect of sampling on the MAE.

To understand the effect of the sampling percentage, we sweep the rate across trials with different random seeds. Figure 17 shows the collection time and MAE as a function of the sampling rate. Their scaling is approximately linear (e.g., for TPC-H, increasing the sampling rate by 10% increases collection time by 10–20 min and decreases MAE by 1–2 s). This demonstrates that the Sampler allows Boot to obtain models even faster, albeit at a proportional model quality cost. Boot defaults to sampling at 50%.

8 RELATED WORK

To our knowledge, we are the first to accelerate training data generation for autonomous DBMSs by decoupling it from regular query execution. We now discuss the areas of related work to this problem.

Query Progress Estimation (QPE): The problem of QPE was first formally defined and studied for Microsoft SQL Server [11] and PostgreSQL [31]. Their idea is to decompose the query plan tree into individual pipelines and then estimate overall pipeline progress using the driver nodes [26]. This decomposition made the estimation problem tractable because the true cardinalities of the driver nodes are easier to obtain (e.g., table scan). As behavior models [34, 37] require operator-level data, this early work is not applicable to Boot. Microsoft SQL Server LQS [27] extended QPE to provide logical operator-level completion estimates (e.g., predicting that a hash join operator has made 42% of its total `GetNext()` calls).

Another difference between Boot and prior work in QPE is that the latter aims to help humans debug query performance. The DBMS must still execute the entire query to produce the correct result. In contrast, Boot only need to execute enough of the query for its µG module to generate approximate operator telemetry.

However, key ideas shared by QPE and Boot are (1) viewing the tuples processed so far as a random sample of the tuples available [27] and (2) using execution feedback to refine initial estimates [54]. For example, existing work in QPE and re-optimization uses random sampling to improve uncertainty [6], cardinality [54] and selectivity [20] estimates. These better estimates are complementary to Boot, which operates in a new training data context.

Approximate Query Processing (AQP): AQP provides faster query results by sampling data [9, 38, 40]. Unlike AQP, Boot does not care about the correctness of the query result. However, AQP techniques are useful for obtaining more accurate execution behavior (e.g., the selectivity of a join operator's predicate) and correcting cardinality estimates (e.g., µG's telemetry scaling). Although Boot and AQP are complementary, we envision AQP will find new applications in accelerating training data generation.

Training Data Collection: Prior work that improves the training data collection process focuses on optimizing instrumentation overhead [8] or reducing the quantity of training data that needs to be collected. For example, using active learning [49] or index-aware similarity [44, 45] reduces the number of queries executed, budget-aware tuning [52, 55] reduces the number of optimizer what-if calls, and incremental model construction [16] allows stopping training data collection early. However, to our knowledge, no techniques exploit the training data setting to accelerate query execution itself.

9 CONCLUSION

Autonomous DBMSs use behavior models to evaluate the benefit of candidate configurations while avoiding the workload execution overhead. Collecting the training data required to construct these models is overly time-consuming, making integrating such models into the DBMS's tuning feedback loop infeasible. We introduce two acceleration techniques to expedite the training data collection process by leveraging the unique characteristics of the training data environment. We show how to apply these techniques with our framework Boot that drops into existing database gym pipelines. Experiments across multiple workloads show that our acceleration techniques are well-suited for bootstrapping an autonomous DBMS's models as they produce models in less time (hours instead of weeks) with only a moderate degradation in model accuracy.

ACKNOWLEDGMENTS

This work was supported (in part) by the CMU Parallel Data Laboratory, VMware Research Grants for Databases, and Google DAPA Research Grants.

REFERENCES

[1] 2011. *TPC-H dbgen.* Retrieved 2024-07-04 from https://github.com/electrum/tpch-dbgen

[2] 2021. *pgmustard: Calculating per-operation times in EXPLAIN ANALYZE.* Retrieved 2024-07-04 from https://www.pgmustard.com/blog/calculating-per-operation-times-in-postgres-explain-analyze

[3] 2023. *Llama 2: Inference code for LLaMA models.* Retrieved 2024-07-04 from https://github.com/facebookresearch/llama

[4] 2024. *pgtune - tuning PostgreSQL config by your hardware.* Retrieved 2024-07-04 from https://github.com/le0pard/pgtune

[5] DCAI 2021. 2021. *NeurIPS Data-Centric AI Workshop.* Retrieved 2024-07-04 from https://datacentricai.org/neurips21/

[6] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/1066157.1066171

[7] Matteo Brucato, Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2024. Wred: Workload Reduction for Scalable Index Tuning. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 50 (February 2024), 26 pages. https://doi.org/10.1145/3639305

[8] 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. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 617–630. https://doi.org/10.1145/3514221.3517845

[9] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 511–519. https://doi.org/10.1145/3035918.3056097

[10] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. 2005. When can we trust progress estimators for SQL queries?. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 575–586. https://doi.org/10.1145/1066157.1066223

[11] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. 2004. Estimating progress of execution for SQL queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 803–814. https://doi.org/10.1145/1007568.1007659

[12] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Data Bases* (Athens, Greece) *(VLDB '97)*. Very Large Data Bases Endowment Inc., 146–155.

[13] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[14] 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 *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 666–679. https://doi.org/10.1145/3299869.3314035

[15] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (September 2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

[16] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently Approximating Selectivity Functions using Low Overhead Regression Models. *Proc. VLDB Endow.* 13, 12 (July 2020), 2215–2228. https://doi.org/10.14778/3407790.3407820

[17] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *arXiv preprint arXiv:2003.06505* (2020).

[18] 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 *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 1159–1170. https://doi.org/10.1145/1376616.1376732

[19] Goetz Graefe. 1994. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (February 1994), 120–135. https://doi.org/10.1109/69.273032

[20] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. 1996. Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Comput. Syst. Sci.* 52, 3 (June 1996), 550–569. https://doi.org/10.1006/jcss.1996.0041

[21] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 143–157. https://doi.org/10.1145/3318464.3389704

[22] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (March 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[23] Rob J Hyndman and George Athanasopoulos. 2021. *Forecasting: Principles and Practice, 3rd edition.* Retrieved 2024-07-04 from https://otexts.com/fpp3/

[24] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1992. Parametric Query Optimization. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*. 103–114.

[25] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: a highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. 3149–3157.

[26] Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. 2011. A Statistical Approach Towards Robust Progress Estimation. *Proc. VLDB Endow.* 5, 4 (December 2011), 382–393. https://doi.org/10.14778/2095686.2095696

[27] Kukjin Lee, Arnd Christian König, Vivek Narasayya, Bolin Ding, Surajit Chaudhuri, Brent Ellwein, Alexey Eksarevskiy, Manbeen Kohli, Jacob Wyant, Praneeta Prakash, Rimma Nehme, Jiexing Li, and Jeff Naughton. 2016. Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1753–1764. https://doi.org/10.1145/2882903.2903728

[28] 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), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[29] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (August 2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[30] 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.*

[31] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. 2004. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 791–802. https://doi.org/10.1145/1007568.1007658

[32] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 175–191. https://doi.org/10.1145/3318464.3389768

[33] 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 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3183713.3196908

[34] 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. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1248–1261. https://doi.org/10.1145/3448016.3457276

[35] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[36] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[37] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (July 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[38] Barzan Mozafari. 2017. Approximate Query Engines: Commercial Challenges and Research Opportunities. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for

Computing Machinery, New York, NY, USA, 521–524. https://doi.org/10.1145/3035918.3056098

[39] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2557–2569. https://doi.org/10.1145/3448016.3457568

[40] Supriya Nirkhiwale, Alin Dobra, and Christopher Jermaine. 2013. A Sampling Algebra for Aggregate Estimation. *Proc. VLDB Endow.* 6, 14 (September 2013), 1798–1809. https://doi.org/10.14778/2556549.2556563

[41] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, and Youngmin Kim. 2018. Data Synthesis based on Generative Adversarial Networks. *Proc. VLDB Endow.* 11, 10 (June 2018), 1071–1083. https://doi.org/10.14778/3231751.3231757

[42] 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*.

[43] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. 2021. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. *Proc. VLDB Endow.* 14, 12 (July 2021), 3211–3221. https://doi.org/10.14778/3476311.3476411

[44] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 660–673. https://doi.org/10.1145/3514221.3526152

[45] Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2022. DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning. *Proc. VLDB Endow.* 15, 10 (June 2022), 2019–2031. https://doi.org/10.14778/3547305.3547309

[46] The Transaction Processing Council. 2021. *TPC-DS Benchmark (Revision 3.2.0)*. Retrieved 2024-07-04 from https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf

[47] The Transaction Processing Council. 2022. *TPC-H Benchmark (Revision 3.0.1)*. Retrieved 2024-07-04 from https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf

[48] 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* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[49] 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 *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1865–1878. https://doi.org/10.1145/3448016.3457286

[50] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization using Reinforcement Learning. *Proc. VLDB Endow.* 14, 13 (September 2021), 3402–3414. https://doi.org/10.14778/3484224.3484236

[51] Kaiwen Wang, Junxiong Wang, Yueying Li, Nathan Kallus, Immanuel Trummer, and Wen Sun. 2023. JoinGym: An Efficient Query Optimization Environment for Reinforcement Learning. arXiv:2307.11704 [cs.LG]

[52] Xiaoying Wang, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2024. Wii: Dynamic Budget Reallocation In Index Tuning. *Proc. ACM Manag. Data* 2, 3, Article 182 (may 2024), 26 pages. https://doi.org/10.1145/3654985

[53] B. P. Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* 4, 3 (1962), 419–420. https://doi.org/10.1080/00401706.1962.10490022

[54] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?. In *29th IEEE International Conference on Data Engineering* (Brisbane, Australia) *(ICDE 2013)*. IEEE Computer Society, 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899

[55] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. Budget-aware Index Tuning with Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1528–1541. https://doi.org/10.1145/3514221.3526128

[56] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 280–294. https://doi.org/10.1145/3626246.3653391

[57] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2022. A Unified Transferable Model for ML-Enhanced DBMS. In *CIDR 2022, Conference on Innovative Data Systems Research*.

[58] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 931–944. https://doi.org/10.1145/3514221.3517885

[59] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (September 2020), 61–73. https://doi.org/10.14778/3421424.3421432

[60] 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. In *Under Submission*.

[61] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. 2023. An Efficient Transfer Learning Based Configuration Adviser for Database Tuning. *Proc. VLDB Endow.* 17, 3 (November 2023), 539–552. https://doi.org/10.14778/3632093.3632114

[62] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2024. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *Proc. VLDB Endow.* 17, 4 (March 2024), 823–835. https://doi.org/10.14778/3636218.3636235

[63] 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 (September 2021), 46–58. https://doi.org/10.14778/3485450.3485456

[64] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries Using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (May 2020), 1416–1428. https://doi.org/10.14778/3397230.3397238

[65] Yiwen Zhu, Yuanyuan Tian, Joyce Cahoon, Subru Krishnan, Ankita Agarwal, Rana Alotaibi, Jesús Camacho-Rodriguez, Bibin Chundatt, Andrew Chung, Niharika Dutta, Andrew Fogarty, Anja Gruenheid, Brandon Haynes, Matteo Interlandi, Minu Iyer, Nick Jurgens, Sumeet Khushalani, Brian Kroth, Manoj Kumar, Jyoti Leeka, Sergiy Matusevych, Minni Mittal, Andreas Mueller, Kartheek Muthyala, Harsha Nagulapalli, Yoonjae Park, Hiren Patel, Anna Pavlenko, Olga Poppe, Santhosh Ravindran, Karla Saur, Rathijit Sen, Steve Suh, Arijit Tarafdar, Kunal Waghray, Demin Wang, Carlo Curino, and Raghu Ramakrishnan. 2023. Towards Building Autonomous Data Services on Azure. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 217–224. https://doi.org/10.1145/3555041.3589674