Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems

Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson[♠], William Zhang, Yu Xia[♦], Andrew Pavlo
Carnegie Mellon University, [♠]Army Cyber Institute, [♦]Massachusetts Institute of Technology
{mbutrovi,wanshenl,lin.ma,wz2,pavlo}@cs.cmu.edu,john.rollinson@westpoint.edu,yuxia@mit.edu

Abstract

A self-driving database management system (DBMS) aims to configure, deploy, and optimize almost all aspects of itself automatically without human intervention or guidance. Achieving this high level of automation relies on machine learning (ML) models that predict how a DBMS will behave in different scenarios. This behavior encompasses all DBMS runtime operations, including query execution and maintenance tasks. These ML-based behavior models for a self-driving DBMS require low-level training data about a DBMS's internals. Such training data includes (1) features that describe the workload, environment, and DBMS configuration, and (2) both DBMS- and hardware-level metrics. But it is difficult to collect training data from a DBMS while it is running because it can introduce performance and measurement degradations that hinder the ML models' ability to predict the DBMS's behavior correctly.

We present the TScout (TS) framework for collecting training data from self-driving DBMSs. Our framework is an internal approach where developers annotate a DBMS's source code with hooks to monitor the system's behavior. TS then extracts these hooks and generates a kernel-level program (via Linux's BPF) that efficiently captures metrics from multiple sources (e.g., CPU performance counters, memory allocators). TS combines these metrics with internal DBMS state observations, generating training data for behavior models. We integrated TS in a PostgreSQL-compatible DBMS and measured its ability to collect training data for both OLTP and OLAP workloads. Our results show that TS generates training data for a deployed DBMS to train more accurate models than previous methods with only a 7% performance reduction.

CCS Concepts

• **Information Systems → Autonomous Database Administration**.

Keywords

Database Systems, Training Data, Modeling, Metrics, BPF, Butrovich!

ACM Reference Format:
Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson[♠], William Zhang, Yu Xia[♦], 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 (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517845

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3517845

1 INTRODUCTION

Self-driving DBMSs seek to automate the arduous tuning and optimization tasks for databases [39, 41, 42]. Given a target objective function (e.g., throughput, latency), a self-driving DBMS automatically deploys actions that it deems will help the application's future workload for that objective. These actions control three aspects of the system: (1) physical design, (2) knob configuration, and (3) hardware resources. Although there are existing advisory tools that support individual actions (e.g., index recommendation [11], knob tuning [52]), a self-driving DBMS takes a holistic approach to optimization to include when and how to deploy such actions.

The functionality that enables this is a self-driving DBMS's ability to estimate the cost/benefit of an action using *behavior models* [29]. For example, if an action adds an index, the DBMS's behavior models predict how much CPU and memory the system will consume to execute future queries if the database includes that index. The behavior models also predict the cost of the action's deployment. Using the same index action example, the models predict how much CPU and memory that DBMS will use to create that index and how that will affect queries running simultaneously.

Building these behavior models as the foundation of a self-driving DBMS requires ample *training data*. Such training data comes from the DBMS executing queries and actions and observing the outcome. Training data comprises both high-level observations specific to the DBMS's operations, such as the number of tuples that a query plan operator consumes, and hardware metrics (e.g., CPU instructions, memory allocations).

One problem with existing approaches for training data collection in autonomous DBMSs is that they rely on *offline* DBMS instances. The most common method is to clone the database and simulate the application via a workload trace [13, 30, 52]. To avoid copying the database, another method uses hand-written "runners" that execute queries in an offline environment [29]. Although these offline methods are useful for bootstrapping a DBMS's models, they require significant time and computational resources to train. They can also not try all combinations of physical designs and knob configurations in a DBMS.

It is clear that the solution to this problem is to perform *online* training data collection while the DBMS executes the application's workload and then augment offline data with this new "fresh" data. Incorporating such online data as soon as possible means that the DBMS's behavior models better reflect the system's current workload and configuration. But current online training data collection methods are bespoke and impose unacceptable runtime overhead.

This paper presents the **TScout** (TS) framework for efficient and accurate training data generation in self-driving DBMSs. Our

framework collects metrics using a combination of hardware-level performance counters, kernel-level observations (via BPF [18]), and application-level counters. TS combines these metrics with descriptors of DBMS behavior during query execution and internal maintenance tasks to create training data for machine learning models. TS uses code generation to create a kernel-level program that collects metrics tailored to the DBMS.

We integrated TS with the **NoisePage** DBMS [1] and measured its overhead and accuracy. We also compare with existing approaches from other observation tools for training data collection in NoisePage. Our results show that TS increases the DBMS's collection rate by ∼300% with only a 7% runtime performance overhead. For behavior models that are heavily influenced by the workload, we also find that generating training data while deployed with TS reduces the error for NoisePage by 98% over existing approaches.

This paper is organized as follows. We first discuss in Sec. 2 training data collection challenges. Sec. 3 presents the TS framework, followed by how to collect metrics in Sec. 4 and engineering issues in Sec. 5. Lastly, we present our evaluation in Sec. 6.

2 BACKGROUND

We begin with an overview of self-driving DBMS architectures. Next, we describe the two data sources needed to train models that guide the decision-making processes in a self-driving DBMS. This will motivate the need for non-intrusive methods to generate and collect online training data.

2.1 Self-Driving DBMS

A self-driving DBMS's architecture is comprised of three components: (1) forecasting system, (2) behavior models, and (3) planning system [42]. The *forecasting system* is how the DBMS observes and predicts the application's future workload [28]. The DBMS then uses these forecasts with its *behavior models* to predict its runtime behavior relative to the target objective (e.g., latency, throughput) [29]. The *planning system* selects actions that improve this objective.

A behavior model represents a discrete component in the DBMS. For a set of input features, the model emits metrics that estimate the component's work for those inputs. For example, a DBMS could build a model for a sequential scan where the input is the table name and the number of tuples that the operator will scan, and the output is the expected execution time.

The models' output is comprised of one or more *metrics*. A metric is a low-level measurement of how the DBMS interacts with its underlying hardware: (1) CPU, (2) memory, (3) disk, and (4) network. The DBMS can easily measure some metrics with user-level code (e.g., memory allocated). Other metrics are only observable via methods that are external to the DBMS in either the OS or hardware (e.g., CPU counters), and thus are more challenging to collect. The self-driving components require accurate models. For example, the planning system uses models and the workload forecast to predict future DBMS resource consumption. If those models are inaccurate, the planning components will not optimize the DBMS.

The scope of a model depends on the implementation. In the case of the sequential scan example, the DBMS could use a single model to represent the entire query plan scan operator [30] or multiple models with internal features that represent smaller operating units (OUs). OU-level models are less complex and thus require less training data than monolithic models [29].

To build OU models, a DBMS needs training data of previous examples of the system's operations (e.g., executing queries). Each data point in a training corpus contains input features and its corresponding output metrics that the models will predict. We now describe ways in which a DBMS can generate such training data.

2.2 Input Features Collection

The inputs to an OU behavior model describe the DBMS's task when executing that OU. For example, the features for an index lookup OU contain its schema, data structure type, and the number of index entries. Since workloads, statistics, and configurations fluctuate, such features must reflect DBMS state at the time of execution. One can collect these features from the DBMS either (1) externally via SQL commands or (2) internally within the DBMS.

External: Previous work on modeling query latency extracts features by executing **`EXPLAIN`** for every query [30]. This provides the optimizer's physical plan and cost estimates, which can be decomposed into individual operator features.

However, this approach is not feasible in high-performance scenarios. **`EXPLAIN`** is meant to be an infrequent operation that regenerates the query plan. Furthermore, it presents human-readable information about query execution, and cannot capture a query's interactions with DBMS background tasks, like garbage collection and write-ahead logging (WAL). Lastly extracting the DBMS's configuration and environment requires executing even more SQL queries. This additional work slows down query execution, making it challenging to collect training data in an online setting.

Internal: An alternative approach is to embed feature collection logic inside the DBMS. Since the DBMS already maintains information about query execution, configuration, and environment, it can use its internal APIs to read this data. In addition, the system can tag background operations with identifiers to describe what queries initiated them. This approach is similar to previous network tracing methods that determine causality in opaque systems [15].

Internal methods solve many problems associated with external training data features at the expense of higher software engineering costs. Foremost is that since the DBMS already needs the results of **`EXPLAIN`** for query execution—explicitly invoking it is not necessary. Second, internal collection accurately represents temporal features that fluctuate, like CPU frequency and the number of concurrent workers. Lastly, internal feature collection enables modeling DBMS subsystems beyond the query execution to express causality in background tasks like write-ahead logging. As such, an internal collection method is better suited for self-driving DBMSs.

2.3 Output Metrics Collection

The output of an OU behavior model is the DBMS's estimated metrics, so the training data for these models must come from runtime operations. Therefore, the challenge is how to efficiently measure a DBMS's metrics without altering its observed behavior.

Previous work on modeling query latency collected metrics using **`EXPLAIN ANALYZE`** [30]. In addition to **`EXPLAIN`**'s limitations that we describe in Sec. 2.2, **`EXPLAIN ANALYZE`** does not return query results

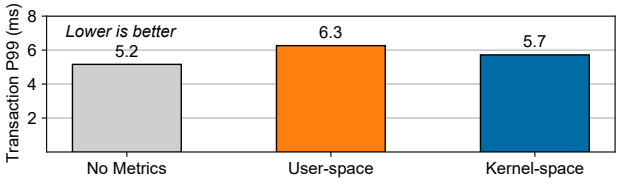


Figure 1: User-space vs. Kernel-space Metrics Collection – Transaction latency of TPC-C with (1) DBMS metrics collection disabled, (2) metrics collected in user-space, and (3) metrics collected in kernel-space using BPF.

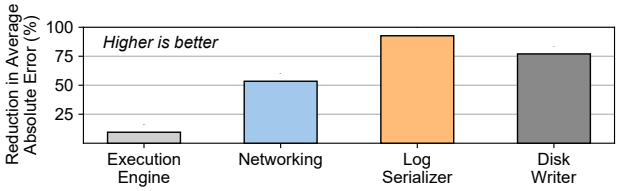


Figure 2: Offline vs. Online Training Data – Accuracy measurements of behavior models trained with offline and online data when predicting the execution time of TPC-C queries.

on the client, and the instrumentation imposes runtime overhead to query execution [45]. It can also not collect metrics for other parts of the system not related to query execution (e.g., maintenance operations). Such issues make **`EXPLAIN ANALYZE`** unsuitable for generating the output metrics for training data.

Given this, there are two approaches for capturing metrics in an internal training data collection framework for a self-driving DBMS: (1) *user-space* or (2) *kernel-space*.

User-space: To collect system metrics from user-space, the DBMS manually invokes the necessary functions in its source code to enable and disable recording metrics. It then manually retrieves the metrics with a final set of function calls.

The problem with this approach is that for some metrics, such as CPU counters, the functions to enable/disable their collection are syscalls to the OS; syscalls are expensive because execution switches into a privileged kernel mode. In some cases, collecting metrics for a single OU requires multiple syscalls per hardware type. Another problem is that the OS may not expose all the metrics via syscalls. Instead, the DBMS developer has to scrape procfs (`/proc`) or other tools, which are slower and less robust than syscalls.

Kernel-space: With kernel-space collection, the DBMS relies on an ancillary program running inside the OS kernel to retrieve metrics for it. This approach means that the program requires only one transition to kernel mode to extract multiple system metrics. Once in kernel mode, the program can read CPU performance counters, network statistics, and any additional system metrics.

Although kernel-space collection is faster than user-space methods, traditional OS kernel modules are notoriously difficult to write and could potentially pose several safety issues. In the last decade, however, the rise of kernel-embedded VMs for specialized programs has simplified executing user code in kernel-space while providing guarantees that the programs cannot harm the OS.

The most well-known of these VMs is Linux's "extended" Berkeley Packet Filter (**BPF**) library [47]. BPF allows developers to write event-driven programs in C-like dialect that run in kernel mode to trace the behavior of other processes. The kernel's BPF validator strictly limits a program's length, permissible operations, and storage allocation. But BPF programs can execute OS-level tasks efficiently due to their privileged execution mode. Developers compile their programs to BPF bytecode and then load them into the kernel. During this loading step, the BPF subsystem verifies the program's safety, just-in-time compiles the bytecode to machine code, and transfers it into the kernel.

The Linux kernel does not continuously run a loaded BPF program. Instead, when an event (e.g., kprobes, uprobes, or tracepoints) occurs in a monitored process, the process switches into kernel mode and executes the BPF code. Unlike some syscalls to retrieve metrics, the OS does not preempt BPF programs, which yields more predictable performance for the DBMS.

When a BPF program runs, it can inspect kernel data structures, explore the DBMS's address space, accumulate metrics into BPF maps, and communicate with the DBMS in user-space. Upon completion, the thread resumes the DBMS's original execution path. BPF metrics collection also minimizes the amount of code that needs to run in a privileged mode; without system modification, a DBMS that collects OS metrics from user-space would need to run as root. The OS sandboxes BPF programs with strict verification requirements, making them safer to use in production environments than a DBMS with elevated privileges in user-space.

To show the benefit of collecting DBMS metrics in kernel-space, we evaluate the average p99 latency of TPC-C transactions on NoisePage in three configurations: (1) one with no metrics collection enabled, (2) one with metrics collection using user-space methods, and a final method relying on BPF. We run the workload with a single client to reduce the effects of contention on any shared data structures. The results in Fig. 1 show how metrics collection using BPF yields more predictable performance. As expected, both methods increase the p99 latency compared to no metrics collection due to the increased amount of work being performed. However, the reduced number of syscalls with BPF reduces the tail latency of queries compared to the user-space approach. The BPF approach generates the same data as user-space syscalls, but with fewer execution mode switches, resulting in better performance.

2.4 Offline vs. Online Training Data

A self-driving DBMS uses two type of data to train its behavior models: (1) *offline data* and (2) *online data*.

Offline Data: A self-driving DBMS generates offline data if it uses a separate environment from the production one. The most common approach is to deploy a clone of the production instance and replay a workload trace [13, 30, 52]. Although creating database snapshots has become easier in recent years, it still requires a new DBMS instance for the clone, which can be prohibitively expensive. Furthermore, one must also capture a workload trace from the production instance to replay on the clone.

An alternative to the clone-based approach is for DBMS developers to create targeted microbenchmarks (i.e., *runners* [29]) that simulate different execution scenarios. Since traces are static and repetitive (especially for OLTP applications), they generate training data that does not provide additional information. Runners target specific DBMS components by sweeping input values to generate unique training data points that yield robust models. Although

these runners require additional engineering for DBMS developers, they reduce redundancy in the offline training data collection.

Regardless of what approach a DBMS uses to generate offline data, the system will need to sweep database configurations and parameters to mimic scenarios in the production DBMS. Depending on the complexity of a DBMS's runner suite, it could take several days or weeks to generate enough offline data to yield robust, generalizable models. Then if the target DBMS's software changes (e.g., upgrading to a new DBMS version) or its hardware changes, the DBMS may have to re-run its offline training data generation methods. It is still impossible to represent all possible configurations and environments.

Online Data: This second data category occurs if the target production DBMS collects the training data as it executes the application's queries. Online data has the advantage that it exactly reflects the DBMS's current workload, database, and environment. This means that as the application's workload evolves or the database grows/shrinks in size, the training data will reflect these changes.

To illustrate the benefit of using online data versus offline data, we compare the accuracy of OU-level models in NoisePage to predict the system's behavior when executing an OLTP workload [29]. We group the OU models by DBMS subsystem because they share the same input features. The offline models use training data collected from NoisePage's built-in runners, while the online models also use training data collected from running the TPC-C benchmark. We then measure the models' average absolute error in predicting the execution time of queries from a TPC-C trace. We hold out 20% of queries (by query template type) from the online training data set, and then evaluate model accuracy on these omitted queries.

The results in Fig. 2 show that online data improves the behavior models' accuracy when predicting OU execution time for previously unseen queries. The models for the WAL subsystem, encompassing the log serializer and disk writer, experience the most improvement because their behavior depends on the workload itself. More accurate OU models allow a self-driving DBMS to better predict its behavior with future workloads and new configurations.

3 FRAMEWORK OVERVIEW

Our analysis in the previous section argues that an ideal training data framework for a self-driving DBMS collects internal input features and kernel-space output metrics from online workloads. Given this, we now present the TScout (TS) training data collection framework. TS facilitates the recording and processing of OU-granular training data from a multi-threaded DBMS to generate training data for the system's behavior models. The framework retrieves training data from the DBMS as it executes queries, and thus it does not require developers to create microbenchmarks to simulate workloads. TS collects output metrics in kernel-space using BPF with minimal performance impact without sacrificing measurement accuracy. When it is not feasible to use kernel-space methods, TS also supports user-space metric collection and makes it easy to combine metrics collected from different approaches.

The architecture overview in Fig. 3 shows that TS's deployment occurs in two stages. In the initial **Setup Phase**, a developer annotates the DBMS's source code with markers to declare OU boundaries and what training data to collect for them. TS then extracts these markers and codegens a custom program for interfacing with the DBMS and collecting training data in the **Runtime Phase**.

We designed TS with two key properties to achieve non-intrusive instrumentation for training data collection. The first is that TS does not produce back pressure on the DBMS. Although there is a small, unavoidable overhead to collecting data, TS does not add blocking synchronization mechanisms on critical paths. The second property is that it supports fine-grained, dynamic collection. TS supports adjustable data collection frequencies per internal subsystem rather than the "all or nothing" approach of many DBMSs.

The following section describes the architecture of the TS framework in more detail. We first describe how developers integrate TS with a DBMS and its codegen process. We then present how the framework operates in its Runtime Phase to retrieve metrics and other data to produce training data for its behavior models. We discuss how TS collects hardware resource data in Sec. 4.

3.1 Setup Phase

TS assumes that the target DBMS uses an internal approach to autonomous planning [29]. Thus, before deploying the DBMS with TS, a developer must first modify the system's source code to indicate when, where, and how TS collects training data. They do this by annotating the DBMS with markers for each OU to specify their locations and subsystem. If the OU belongs to a new subsystem of the DBMS, then the developer also defines the OU's input features and resources for TS to monitor. TS's Compiler uses this information to generate the Collector's BPF code (Sec. 3.2).

Markers: TS provides its markers as statically-defined tracepoint macros [5, 14, 48]. Tracepoints were introduced as part of Solaris' DTrace project for kernel-level debugging and dynamic tracing. At DBMS compilation time, these tracepoint macros generate `NOP` instructions and metadata about their offset location in the program code. The OS replaces these `NOP`s with instructions that enable TS to instrument the OU when the program starts.

BPF supports other invocation methods (e.g., uprobes [51]) that do not require source code annotation. We use tracepoints instead of them for several reasons: (1) compiler optimizations and function name mangling make it difficult to define OU boundaries in machine code, (2) OU behavior can span multiple functions, (3) DBMS control flow can have multiple exit paths from OUs (e.g., query and trigger exceptions), and (4) DBMSs that use JIT compilation [37] further obfuscate system behavior, making instrumentation less reliable.

TS uses markers to identify when the DBMS executes an OU. A developer annotates each OU in the DBMS's source code with a triplet of markers: (1) `BEGIN`, (2) `END`, and (3) `FEATURES`. The first two represent the start (`BEGIN`) and finish (`END`) boundaries of an OU in the system. At runtime, when a DBMS's thread encounters a `BEGIN` marker, this triggers an event that causes TS to enable metrics collection for that thread. The framework then disables collection when that same thread encounters a `END` marker. TS maintains an internal state machine to handle situations when markers are in an unexpected order (Sec. 5.1).

The third marker type (`FEATURES`) is how the DBMS records the input features and any user-level metrics for each OU's behavior model (Sec. 2.1). In most cases, the features of an OU are known before execution (i.e., an OU's inputs describe the amount of work

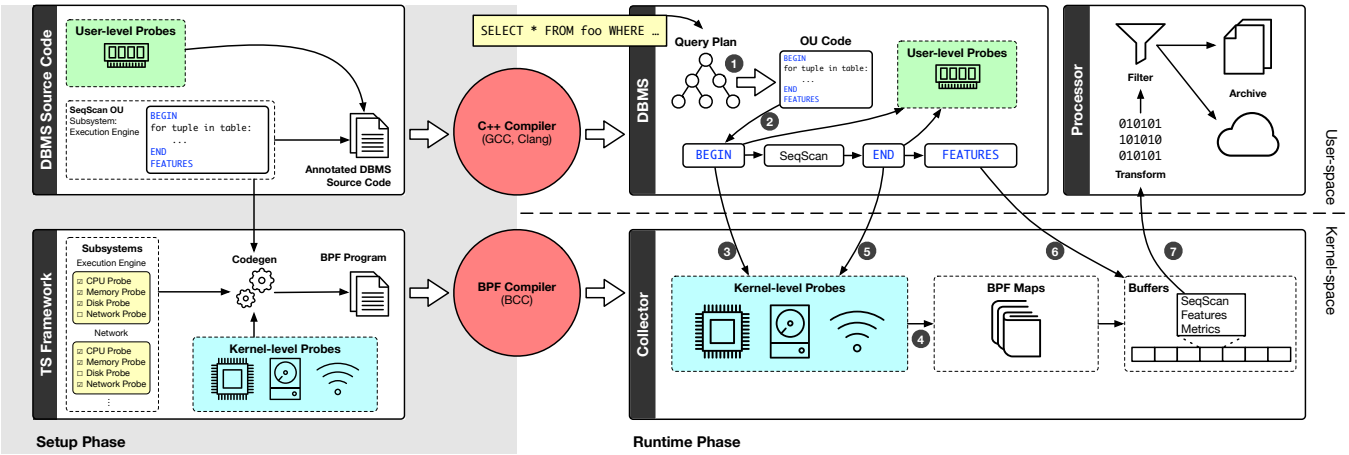


Figure 3: Framework Overview – TScout's architecture is split into two phases. In the Setup Phase, a developer annotates the DBMS's source code. Then, TS uses code generation to create a customized Collector for the Runtime Phase. During the Runtime Phase, the Collector retrieves metrics and the DBMS's OU input features to create training data. Finally, these data are shipped to the Processor for archiving with other training data.

it will perform). However, DBMSs also perform operations that cannot be summarized until completion. One example of this scenario is processing messages from the network layer. PostgreSQL's protocol allows for multiple queries to be sent in a single packet, and only after fully inspecting the buffer can the amount of work be summarized. For this reason, TS treats input feature generation as a separate event after OU execution.

Fig. 3 shows an example of using markers for an OU that performs a sequential scan on a table. The `BEGIN` and `END` markers enclose the loop that reads tuples from the table. The `FEATURES` marker denotes the input parameters for the scan operation, including the table's name and the number of scanned tuples.

TS does not collect training data all the time as continuously monitoring fine-grained OUs would (1) degrade the DBMS's performance and (2) generate a large amount of data that require significant storage resources. Thus, TS needs to toggle collection on and off at runtime on a per query basis. TS wraps markers with lightweight sampling logic that determines whether to collect training data at runtime. Some input features also require the DBMS to aggregate them before sending them to TS (e.g., total memory used by an OU over multiple allocations). TS informs the DBMS that it can bypass this work via a user-space flag that indicates when collection is enabled. We discuss these optimizations in Sec. 5.3.

Codegen: After the developer adds markers to the DBMS's source code, TS then extracts their embedded metadata from the DBMS to determine which metrics it needs to collect per subsystem. For the example shown in Fig. 3, the DBMS's execution engine needs CPU, memory, and disk metrics but not network metrics. This metadata also indicates that CPU and disk metrics will come from TS's built-in probes whereas the memory metrics will come from a developer provided probe. TS then generates the source code for a BPF program to create the Collector component. As we describe in Sec. 3.2, the Collector retrieves the necessary metrics for each marker and aggregates their measurements. The framework then generates kernel-safe bytecode for this program using a BPF compiler (`BCC`).

3.2 Runtime Phase

The administrator deploys the DBMS together with TS on the same server. Using the example in Fig. 3, the application submits a query that performs a sequential scan and then ❶ the DBMS begins executing the query's plan. We assume that the DBMS uses a single thread to simplify our discussion but TS supports multi-threaded execution. ❷ The thread then executes the OU for the sequential scan that the developer annotated with markers in the Setup Phase. ❸ When the DBMS encounters the `BEGIN` marker for this OU at runtime, it triggers TS to enable metrics collection using its probes.

We next describe how TS coordinates the retrieval of metrics from its probes with its Collector component. We then discuss how TS uses its Processor to extract training data from the DBMS.

Collector: The Collector orchestrates training data collection from disparate sources in the DBMS. In addition to the metric-specific code from TS's Codegen component, the Collector also includes data structures to store intermediate metrics.

❹ The Collector uses a BPF map to store a snapshot of probes' results at the `BEGIN` marker. ❺ After completion of the OU and triggering the `END` marker, the Collector retrieves this initial data from the BPF map, invokes the necessary probes again to get a current snapshot, computes the value for each metric, and then stores the final results back in the BPF map. When the DBMS reaches the OU's `FEATURES` marker, ❻ the Collector packages the features and metrics together into a struct (sample data point) and then ❼ sends it to the Processor via a `perf` ring buffer for the final step.

Probes: TS's probes generate metrics from the running DBMS. A probe is a small piece of reusable code that retrieves one or more metrics that the framework uses for multiple OUs. In the example from Fig. 3, the framework contains a probe to retrieve the number of CPU cache misses during the sequential scan OU.

TS supports both user-level and kernel-level probes. In the Setup Phase, the developer specifies which probe to use for each resource category. Both types of probes are necessary because it is more efficient for the DBMS to retrieve some metrics using one type versus another (Sec. 4). TS's markers support passing arguments to

kernel-space probes so the DBMS can provide qualifiers for an OU, such as which file descriptors or network sockets to monitor.

The memory probe in Fig. 3 is an example of a user-level probe because the developer writes the code to perform metrics retrieval from inside the DBMS. TS provides the DBMS with a buffer for storing input features and metrics at runtime. The DBMS's responsibility is to fill that buffer with necessary data and send it to the Collector at the FEATURES marker. For kernel-level probes, TS handles the retrieval and storage of metrics automatically.

Processor: The Processor is the user-space component that extracts and archives training data. Once the Collector sends a completed sample as a `perf` buffer to the Processor, the Processor can do additional cleaning on the data and write it to the appropriate output target (e.g., local disk, cloud storage).

When the Processor receives a sample, it extracts the raw data from the `perf` buffer and transforms it to the correct format. For our implementation in NoisePage, this requires data type conversions, but the final format is configurable based on the system's ML training pipeline. For example, a DBMS with a Volcano-style execution engine [17] with query plan operators that call child operators can leverage the Processor's transformation step to separate the runtime metrics of parent and child operations in the query plan. If the Processor cannot keep up, it has a feedback mechanism to decrease the sampling rate. Alternatively, the Processor can drop data without incurring correctness problems; the Collector's buffer is bounded so that TS will overwrite samples if it is full.

4 RESOURCE PROBES

Probes are reusable code that run at OU boundaries defined by the markers in the DBMS. For each probe, the BEGIN marker starts the measurement, and the END completes the process. TS provides probes for four hardware categories: CPU, memory, network, and disk. There are multiple reasons for this separation. First, the nature of how the DBMS uses these resources is different. The DBMS creates threads or processes that the OS schedules to run on the CPU. For memory, the DBMS is constantly acquiring and releasing memory to execute queries whereas it groups blocking IO calls. The second reason for distinct probe definitions is because the OS exposes different ways to measure each hardware type. The OS provides syscalls to measure usage for some hardware, while others require access to kernel internals. Thus, depending on the hardware category, a probe for TS is either (1) user-level or (2) kernel-level.

TS provides kernel-level probes written in BPF for measuring CPU, network, and disk activity. Sec. 3.2 describes how TS generates this code in the Setup Phase. Memory is the only category that requires developers to implement a user-level probe per DBMS. In this case, TS generates no Collector code for that hardware category. Instead, user-level probes require the DBMS to track its resources during OU execution. The DBMS bundles those metrics at the FEATURES marker before sending them to TS's Collector.

In addition to user-level and kernel-level metrics, hardware devices can expose counters (e.g., CPU PMU, disk SMART stats). TS does not provide probes to collect any of these. The code required to access these counters can be vendor-specific, making it not portable and onerous to implement. For example, Intel's `rdpmc` instruction accesses PMUs, but this requires low-level knowledge of the CPU's ISA and the OS's scheduling algorithm. We now describe how TS's probes extract metrics per hardware category.

4.1 CPU Probe

TS's CPU probe measures the amount of work a CPU performs for the DBMS within the boundaries of an OU. This work includes pipeline information like cycles, instructions, and reference CPU cycles. It also records caching metrics, such as cache references and misses. TS use the `perf_event` functions of BPF to retrieve this information. These functions are a stable API provided by the Linux kernel that supports multiple architectures [54]. The framework could also access `perf_event` in user-space; we measure the tradeoffs between different access methods in Sec. 6.2.

At the BEGIN marker, TS's CPU probe reads `perf_event` counters, normalizes their values, and stores them in the Collector's BPF map. This normalization step is necessary because of the limited PMU registers on the CPU. If the probe enables more `perf` counters than the CPU supports, then the OS samples these values. The probe must normalize raw counters by the elapsed active time in the PMU. TS handles this step transparently.

At the OU's END marker, the CPU probe again reads the `perf` counters and performs the same normalization. It then retrieves the initial values from the BPF map, computes their difference, and stores the final metrics back in the BPF map.

4.2 Memory Probe

TS's memory probe is the only user-level probe in the framework. The developer instruments their OUs to collect their memory consumption for TS. There are several reasons that memory is the only hardware category that follows this model. First, memory allocations in a DBMS have different life cycles that may persist beyond OU execution. Second, the frequency of memory allocations make them impractical to instrument with BPF. Lastly, the abstraction layers of memory allocators and virtual memory make it difficult to assign ownership to an OU from BPF. We now discuss each of these considerations in more detail.

Life Cycles: When a DBMS thread allocates memory, the system's amount of time to use that memory depends on many factors. Some memory is for transient tasks (e.g., C++ object allocation), while some memory is for data structures that last the DBMS process' lifetime (e.g., catalog metadata). But for modeling the DBMS's runtime behavior for self-driving purposes, the only allocations that matter are for buffers during query execution (e.g., query results, WAL redo buffers). Thus, only the DBMS developer knows how long a memory allocation is expected to be needed, and if the ownership will change beyond the OU that initiated the request.

Frequencies: A DBMS performs millions of memory allocations per second. Although allocators keep the majority of them completely in user-space, BPF probes would force every allocation to trap into kernel-space, which would be unbearably slow. TS's query sampling reduces overhead, but it prolongs the time to generate sufficient training data. TS's user-level memory probe allows DBMS developers to instrument allocations optimally at a granularity that makes sense for each OU definition in their system.

Abstractions: Modern allocators (`jemalloc`, `mimalloc`) handle `malloc` calls by over-requesting and then reusing memory to minimize syscalls. These allocators have tunable pools, reuse policies, and other optimizations to avoid fragmentation. As such, these allocators obscure the OS's view of true memory usage. The OS is only aware of a specific memory request from the DBMS if it requests a large allocation that is unsuitable for OU instrumentation.

Within the kernel, virtual memory further complicates determining how much memory an OU actually consumes. First, Linux assigns physical memory to virtual addresses lazily by default, with the kernel relying on page faults to service allocations. This disparity between request time and use time, and some allocators relying on this behavior to minimize system calls, makes it difficult to attribute a request for raw memory to a specific memory allocation by an OU. Lastly, POSIX instrumentation syscalls like `getrusage` are too coarse-grained, and only provide summary statistics for the entire DBMS process.

For the above reasons, TS provides a user-level probe for collecting memory metrics. DBMS developers implement their own memory tracking for each OU, and populate the values at the appropriate `FEATURES` markers. TS bundles these metrics with the results of its kernel-level probes to complete an OU's data. TS takes this approach because DBMSs already track memory to support knobs for join buffer sizes [31] and queries like **`EXPLAIN ANALYZE`** [23].

4.3 Network Probe

TS provides a kernel-level probe that records an OU's network activity (e.g., bytes read/written). One could implement this with a user-level probe, and many DBMSs already keep such statistics by accumulating the results of networking syscalls. But a benefit of TS's approach is separating instrumentation logic from control logic. DBMS networking state machines are complex code that rely on the return value of socket operations. Without TS, a DBMS needs additional code to respond to the socket results and accumulate metrics. With TS, developers only need to wrap socket operations (e.g., `read`, `write`) in markers.

There are user-space sources for network metrics, but they impose a larger overhead than TS's BPF approach. Linux command-line tools (e.g., `netstat`, `ss`) generate socket statistics, but the frequency of OU execution make them infeasible. Linux's `sock_diag` syscall also provides socket-level metrics, but it also has overhead from data copying and parsing. Instead, TS's network probe limits the transition overhead to just the mode-switch and extracts socket statistics by reading kernel data structures (`tcp_sock`).

4.4 Disk Probe

TS's disk probe design is similar to the network probe described above: it is a kernel-level probe that calculates bytes read and written for disk IO. Like for network activity, this probe could track this information in user-space based on the results of IO syscalls, but TS again separates the concerns of instrumentation and control flow. The performance concerns of using command-line tools at OU granularity and syscall overheads apply to measuring disk metrics.

TS's kernel-level disk probe collects disk statistics the same way as the network probe: by reading metrics that the Linux kernel already maintains by traversing the DBMS's `task_struct`. As an optimization specifically for NoisePage, TS's implementation assumes a single open file (e.g., WAL) for the entire DBMS process, which allows the probe to retrieve metrics from the OS's `ioac` IO accounting struct. One can extend the probe to support tracking IO for multiple files by using their descriptors as marker parameters.

5 ENGINEERING

We now present additional details and considerations of TS's implementation that we did not cover in Secs. 3 and 4.

5.1 BPF Development

BPF enables tools to run programs inside the OS without recompiling the kernel or loading privileged modules. BPF currently requires a modern Linux kernel; TS targets kernel version 5.4 due to its wide adoption in the major Linux distributions. As we now describe, BPF imposes unique constraints for these programs.

Most of these limitations come from BPF's *verifier* that checks programs as they are loaded into the kernel. The verifier enforces a program's safety and performance guarantees by building a control flow graph and then checks for unreachable instructions. It allows for loops, but they must be bounded at compile-time. BPF does not allow dynamic allocation other than in BPF maps, and it restricts pointers to a safe API. Because TS reads kernel data structures to extract runtime metrics, TS validates kernel memory accesses before dereferencing. The verifier also restricts the total length of the program to 1m instructions. This is not a problem for TS as its compiled BPF programs only contain 100s of instructions.

BPF tooling is limited, and messages from the BPF compiler and verifier do not always make errors obvious. TS simplifies BPF development by using its Codegen component to generate high-level C code that uses the BCC library to generate BPF code. As described in Sec. 3.1, DBMS developers select the hardware resources to monitor for an OU, and TS automatically composes probe code.

TS benefits from BPF constraints by enforcing a strict state machine. If TS executes in an unexpected order, the Collector resets training data collection, discards any intermediate results, and logs an error message. This scenario can occur in two ways: (1) a DBMS developer placing markers in a sequence that TS deems incorrect (e.g., out-of-order markers) and (2) exceptional control flow due to client request to the DBMS.

5.2 Query Engine Integration

TS's flexibility supports all modern DBMS designs. As we now describe, two cases are challenging to implement OU training data collection in a DBMS's execution engine.

JIT Query Compilation: With this technique, the DBMS converts a query plan into executable code that it then compiles and links into its address space [37]. The problem, however, is that tracepoints in this dynamic code are not known when TS's Codegen component creates its BPF program and will not trigger events at runtime. Instead, one can wrap the location in the DBMS that invokes a compiled query with markers. But in a system using fine-grained OUs, a compiled query will contain multiple OUs. In general, the problem is that one may want to model query execution at a granularity that the DBMS does not support.

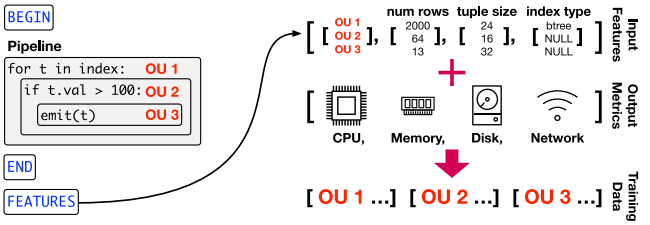


Figure 4: Query Engine Integration – TS's vectorized input features supports collecting training data from a JIT compilation execution engine with a pipeline containing multiple OUs.

To overcome this problem, TS supports recording the input features for multiple OUs that the DBMS executes together. In the example shown in Fig. 4, the compiled query contains three OUs: (1) index lookup, (2) filter, and (3) writing to an output buffer. After invoking the query, the Collector retrieves a single set of metrics from its probes but the FEATURES marker emits vectors of input features for the query's three OUs. Breaking apart which portion of the metrics correspond to which OU happens when the DBMS trains new models, which is outside the purview of TS.

Recursive Operators: Another problem is when an operator invokes itself, causing TS to encounter the same BEGIN marker twice before a END marker. This can occur in a Volcano-style [17] DBMS when an operator higher in the query plan invokes the same operator type. For example, a hash join operator calls *next* on its child operator that happens be the same hash join operator. This situation can also arise in queries with recursive common table expressions (CTEs) where an operator may call its own function.

We solved this problem by modifying TS's Collector to maintain intermediate results using a stack BPF map. For each BEGIN marker, the Collector pushes a new OU invocation entry onto this stack. Then as it encounters FEATURES markers, it pops the last entry from the stack, checks that it matches the expected OU type, and then transmits the data to the Processor.

5.3 Sampling

For training data collection in NoisePage, we implemented adjustable sampling for subsystem events. These events are typically queries, but some subsystems group operations to improve performance. For example, the disk serializer combines query results into buffers, so each sample corresponds to a single buffer.

Per-subsystem sampling reduces overhead at the expense of lower data generation rates. TS maintains a 100-bit field for each subsystem to represent its sampling rate. For example, a sampling rate of 100% has this field to all one, while a rate of 20% will have 20 random bits set to one. The random distribution of ones reduces the burstiness of collection. Without shuffling, a transaction's query sequence may fall entirely within the sampling window, thereby experiencing higher latency than other transactions. Lastly, each thread in NoisePage maintains offsets to index into the bit fields. On a candidate collection event, the thread checks the bit value at its offset, uses the value to enable or disable training data for the event, and then increments the offset until it wraps around to zero.

5.4 Dynamic Feature Selection

A key challenge in a training data collection framework like TS is how to support changing what data to collect. Previous work in DBMS tuning has shown the benefits of automatic feature selection algorithms to simplify modeling [52]. Adding additional collection targets (e.g., user-space probes) to a DBMS requires source code changes. If the DBMS already exposes the necessary data, external approaches can modify what data they retrieve without affecting the system's behavior. But internal approaches may potentially require redeployment and restarting whenever the models require new input features, which is difficult because production DBMSs have high uptime requirements.

If one does not need to modify the DBMS's code, then TS supports dynamic selection of internal features without restarting in two ways: (1) TS's Collector runs in the same address space as the DBMS, so it has access to all information available at deployment. (2) TS can dynamically unload BPF programs, modify them, and reload them. With this capability, developers can change the internal features to collect, restart TS, and generate new models from the training data. We defer the problem of using TS to automatically identify which features to include in a DBMS's OU models as future work.

6 EVALUATION

To evaluate the efficacy of the TS framework, we integrated it in the NoisePage DBMS [1]. NoisePage is a PostgreSQL-compatible DBMS that uses HyPer-style MVCC [38] over Apache Arrow in-memory columnar data [27]. The original version of NoisePage uses OU-level behavior models that it trains via offline runners with user-level probes [29]. We modified NoisePage's source code to introduce TS's markers and probes for its OUs.

We deployed NoisePage on a server with 2×20-core Intel Xeon Gold 5218R CPUs, 196 GB DRAM, and Samsung PM983 SSD. For the experiment in Sec. 6.6 with a smaller hardware configuration, we use a server with a single 6-core Intel Core i7-10710U CPU, 64 GB DRAM, and Samsung 970 EVO+ SSD. Both machines run Ubuntu 20.04 with Linux (v5.4) that supports BPF.

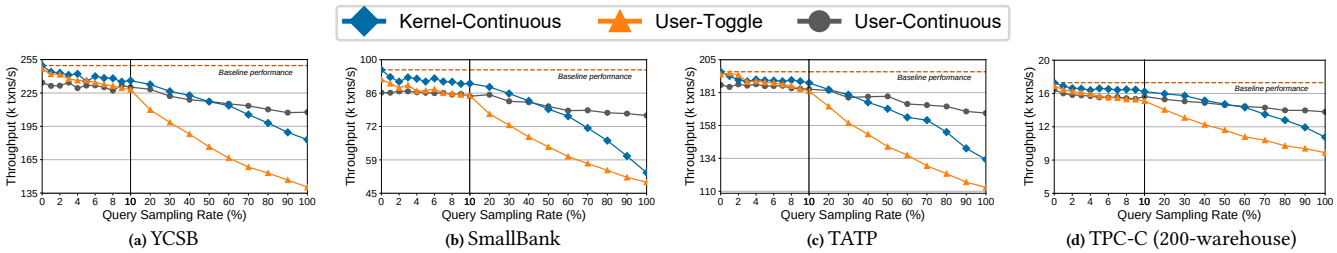
In each experiment, we deploy NoisePage and TS's BPF-based Collector running in the kernel. We also use a single-threaded Processor to extract training data and write it to a file on the server's local disk. Since NoisePage uses operation-fusion in its execution engine, we preprocess the DBMS's online models to break multiple OUs per execution engine operation into per-OU data points using offline models (Sec. 5.2).

When evaluating the performance of OU models trained with data from TS, we report the *average absolute error*. OLTP transactions are short-lived and result in noisy runtime measurements, so we measure the absolute error ($|Actual - Predict|$) for each query template and then compute the average.

6.1 Workloads

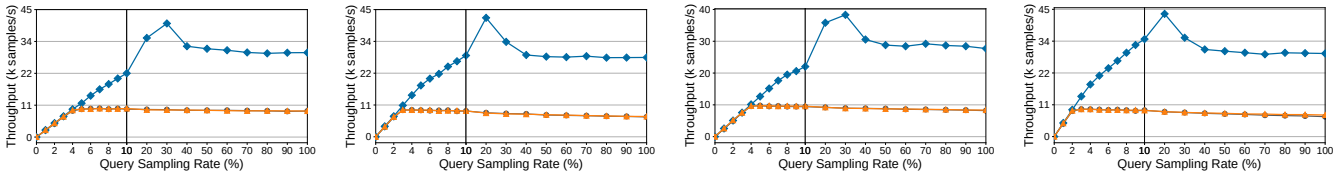
We use the following workloads in our evaluation. All queries execute over JDBC using the BenchBase framework [2, 12].

YCSB: The Yahoo! Cloud Serving Benchmark [10] is a synthetic benchmark modeled after cloud services. To maximize the transaction throughput, we use a read-only configuration for YCSB that only executes queries that retrieve a single tuple using its primary



(a) YCSB (b) SmallBank (c) TATP (d) TPC-C (200-warehouse)

Figure 5: Runtime Overhead (Transaction Throughput) – Impact of query sampling on OLTP transaction throughput, comparing user-space and kernel-space approaches to system metrics collection.



(a) YCSB (b) SmallBank (c) TATP (d) TPC-C (200-warehouse)

Figure 6: Runtime Overhead (Training Data Generation) – Impact of query sampling on OLTP training data generation, comparing user-space and kernel-space approaches to system metrics collection.

key. The YCSB database contains a single table comprised of tuples with a primary key and 10 columns of random data, each 100 bytes in size. Each tuple's size is approximately 1 KB. We use a database with 12m tuples (~13 GB).

SmallBank: This workload models a banking application where transactions perform simple read and update operations on customers' accounts [3, 8]. All transactions involve a small number of tuples retrieved using primary key indexes. In addition to the original six transaction types, we added a transaction that transfers money between two accounts. The database contains three tables with 50m accounts (~10 GB).

TATP: The Telecom Application Transaction Processing benchmark is an OLTP testing application that simulates a caller location system used by telecommunication providers [55]. It has nine transaction types that use either a primary key to find caller records or a secondary index as an indirection look-up to caller records. The database contains 20m tuples (~16 GB) stored across four tables.

TPC-C: This is an order-processing application that contains nine tables and five transaction types [50]. For our experiments, we use a 1-warehouse database (128 MB), a 20-warehouse database (~2.5 GB), and a 200-warehouse database (~25 GB).

CH-benCHmark: This is a hybrid (HTAP) workload comprised of the TPC-C OLTP schema with queries adapted from the TPC-H OLAP benchmark [9]. We use four threads to execute TPC-H queries and 16 threads to execute TPC-C transactions.

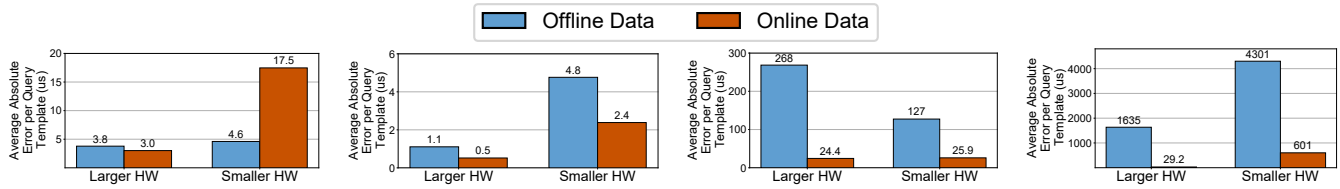
6.2 Runtime Overhead

We first measure to what extent collecting training data with TS reduces the DBMS's performance. We modified NoisePage to use three collection methods with TS: (1) kernel-level probes with continuous `perf` counters (Kernel-Continuous), (2) user-level probes with dynamic `perf` counters (User-Toggle), and (3) user-level probes with continuous `perf` counters (User-Continuous). For the first and last methods, TS enables its probes even if the framework never retrieves the data (e.g., TS toggles on CPU counters for all OUs at DBMS start-up). The second approach dynamically toggles `perf` at the start and stop of each OU. We do not evaluate toggled `perf` in a kernel-level probe as this does not align with BPF's access API.

This experiment enables data collection for all DBMS subsystems to ensure the maximum impact. We sweep the TS's sampling rate from 0% to 100%. We run each rate configuration three times and report (1) the average throughput and (2) how many data samples TS generates. We execute the workloads described in Sec. 6.1 with 20 client threads. We use Kernel-Continuous with 0% sampling rate as the baseline because the only user-space logic is minimized to sampling behavior, which all three methods need.

The results in Fig. 5 show that the DBMS's performance drops as the sampling rate increases for each method. This reduction is because the DBMS, OS, and TS spend more time collecting and processing training data as the rate increases. In the User-Toggle configuration when the sampling rate reaches 100%, the DBMS's throughput drops by almost 50%. The User-Continuous approach reduces the DBMS's throughput by 2–8% even when the sampling rate is 0% because the kernel does more work at each context switch to save the CPU's PMU state. But since User-Continuous only requires a single syscall to retrieve `perf` counters, it incurs at most a 15% reduction compared to the other methods. Kernel-Continuous requires multiple syscalls to read `perf` counters, which slow the system down despite the syscalls occurring under a single switch in execution mode. User-Toggle is the slowest because it requires three syscalls with execution mode switches for each sampled event: one to enable counters, disable counters, and read results.

User-Continuous has the smallest impact on the DBMS's performance at a high sampling rate, but we must also consider the data generation rate. As such, we next compare the number of data points that TS extracts from the DBMS as the sampling rate increases. We measure this from the number of training data samples that the Processor writes to its output source.



(a) Execution Engine (b) Networking (c) Log Serializer (d) Disk Writer

Figure 7: Adapting to Environment Changes – Comparing prediction error of baseline OU models trained with offline runner data against models trained with online TPC-C data.

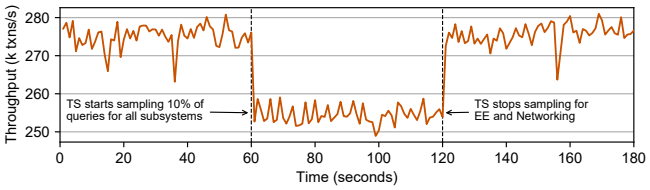


Figure 8: Adjustable Sampling – Impact of training data sampling on YCSB transaction throughput.

Fig. 6 shows TS's data generation throughput increases as the sampling rate increases but only up to a point. For User-Toggle and User-Continuous, the overhead of retrieving data from the user-space probes prevents them from generating more data points beyond a 2–4% sampling rate and the Processor is mostly idle waiting for data from the Collector. On the other hand, TS generates data using Kernel-Continuous at a 3× higher generation rate than the other two methods. Kernel-Continuous achieves peak throughput at a 20–30% sampling; at a higher sampling rate, the overhead of collecting the data from probes slows down query execution, leading to less collected data. We attribute Kernel-Continuous's advantage to its in-kernel data structures with efficient RCU synchronization [33].

The results in Figs. 5 and 6 show that the ideal configuration for TS is to use Kernel-Continuous with a sampling rate of 10%. This setup yields the maximum throughput for both the application's workload and training data generation.

6.3 Adjustable Sampling

The previous experiment evaluated TS's overhead when collecting training data for all the DBMS's subsystems simultaneously. As described in Sec. 5.3, TS also has per-subsystem sampling that allows it to collect training data only for the OUs whose models require refinement. Such flexibility further reduces TS's overhead since it is not an "all or nothing" approach.

To demonstrate how collecting data for individual subsystems affects the DBMS's performance, we run YCSB and then dynamically adjust TS's subsystem sampling rates at runtime. TS starts with a 0% sampling rate. After 60 sec, TS starts sampling 10% of queries across four subsystems: (1) execution engine, (2) networking, (3) log serializer, and (4) disk writer. Then after another 60 sec, TS adjusts the execution engine and networking sampling rates to 0% to simulate a scenario where TS has generated enough data. TS maintains its 10% sampling rate for the other two DBMS subsystems.

As shown in Fig. 8, the DBMS's throughput drops by ~7% when TS starts training data collection for all subsystems. But the DBMS's returns to its original throughput when TS disables collection for the system's execution engine and networking. Although TS keeps collecting data for the DBMS's WAL-related subsystems, the workload is read-only, and TS does not generate much training data and imposes minimal overhead to the DBMS's throughput.

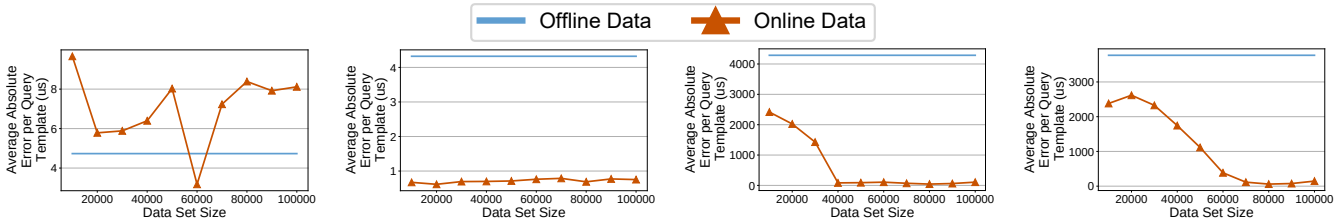
6.4 Adapting to Environment Changes

We next evaluate the ability of the DBMS's models to adapt to changes in its environment. We target the scenario where the DBMS migrates to a new machine with different hardware capabilities. This experiment highlights a key benefit of online training data collection: a self-driving DBMS does not need to redeploy its offline runners to retrain its models after the migration.

We consider two separate scenarios where the DBMS migrates to either (1) a better machine (Larger-HW) or (2) a weaker machine (Smaller-HW). As described above, our more powerful machine has 2×20-cores and the lesser one has 6-cores. For each scenario, we first deploy the DBMS with only offline models on its initial hardware type. We then move the DBMS to the new machine and enable TS's collection for 1 min while executing TPC-C (20 warehouses, one client). During this time, TS generates ~2m training data samples; we tested generating the same data at lower sampling rates with a longer workload time but saw similar results. We then retrain the DBMS's behavior models with the online and offline data combined and evaluate the model accuracy with 5-fold cross-validation.

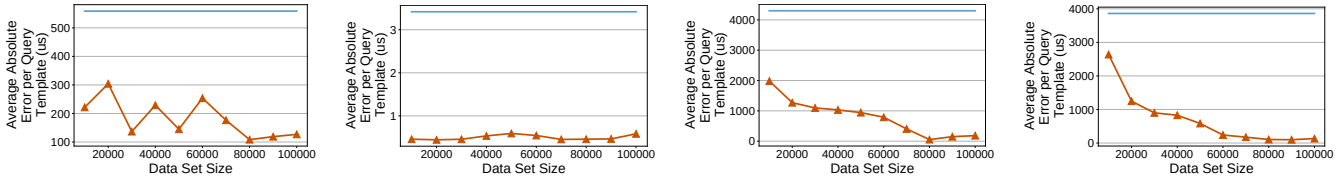
The results in Fig. 7 show that online data helps in nearly every scenario across all subsystems. The largest improvement is NoisePage's disk writer (Fig. 7d), where online data improves the models' accuracy by 98% and 86% when migrating to larger and smaller hardware, respectively. The accuracies of the log serializer models also improve by up to 91% with online data; as we mentioned previously, this gain is because the online data better represents the runtime behavior of the system's group commit implementation. Both the networking and execution engine (Larger-HW only) see modest improvements as well, though the average error already starts small in those scenarios (i.e., <20μs).

The only model whose accuracy does not improve with online data is the execution engine on smaller hardware in Fig. 7a. We see a similar but a less pronounced reduction when running this same experiment on other machines that are slightly less powerful than our 2×20 machine. We believe that this diminution is due to a shortage of features describing the CPU for the models to generalize across architectures. The CPU most impacts query execution time, yet the only hardware context feature is the CPU's clock speed. The larger machine has over twice the amount of L3 cache (27.5 MB vs. 12 MB), significantly affecting query performance. We suspect that features that characterize the CPU beyond clock speed would increase the benefit of online training data in this scenario.



(a) Execution Engine (b) Networking (c) Log Serializer (d) Disk Writer

Figure 9: Model Convergence (TPC-C) – Comparing prediction error of baseline OU models trained with offline runner data against models trained with increasing size online TPC-C data.



(a) Execution Engine (b) Networking (c) Log Serializer (d) Disk Writer

Figure 10: Model Convergence (CH-benCHmark) – Comparing prediction error of baseline OU models trained with offline runner data against models trained with increasing size online CH-benCHmark data.

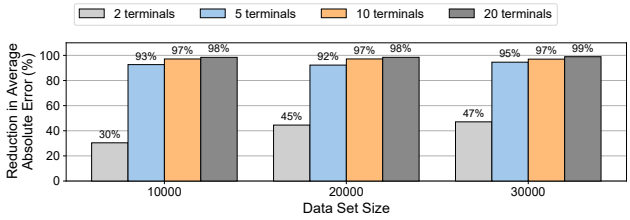


Figure 11: Model Convergence (TPC-C) – OU model accuracy improvement for the execution engine with increasing number of workers. Higher bars show a greater reduction in error.

6.5 Model Convergence

We now measure the convergence time for NoisePage's behavior models to understand how much training data is necessary to generate accurate predictions. As shown in Sec. 6.2, TS generates thousands of data points per second at a low sampling rate. Thus, this experiment informs us on how long to enable collection for a DBMS's current configuration.

We simulate a scenario where the DBMS migrates to a larger server and must refine existing models initially trained on a weaker machine with online data. We execute TPC-C (20 warehouses, one client) for 15 min with TS's metrics collection enabled. We again use 5-fold cross-validation to evaluate the model accuracy. For each fold of test data, we retrain the model with increasingly larger random data samples to evaluate the convergence.

The results in Fig. 9 show the accuracy measurements of the models for NoisePage's four subsystems with larger training data set sizes. The horizontal line in each graph represents the baseline accuracy of NoisePage's offline models. Fig. 9c indicates that the DBMS's log serializer models converge after 40k data points and improve accuracy by up to 98%. This is because NoisePage uses group commit and batches records from different transactions to reduce disk IO. The offline runners target individual OUs, and do not represent the behavior of the end-to-end workload. Similarly, the DBMS's disk writer models converge after 70k data points in Fig. 9d. This subsystem is workload dependent like the log serializer, but also shows how online data benefits scenarios where the IO device changes. The networking models shown in Fig. 9b do not require much data to converge and its error difference from the offline models is small (i.e., <5 μs).

The most interesting result is for the execution engine in Fig. 9a where the online models are less accurate than offline. Although the difference is small (i.e., <1 μs), it contradicts expectations and what we see with the other models. NoisePage's offline runners are heavily influenced by TPC-C's workload with a single client [29], and thus there is not much for online data to improve.

To provide a better use case to show convergence, we run TPC-C scaling the number of clients. We only report in Fig. 11 the results for the execution engine with fewer training data set sizes; the convergence for the other subsystems are not shown, but their behavior follows those of Fig. 9. As the number of clients increases, the offline models are less accurate at predicting execution time. With 20 clients, the offline models' average absolute error is 885 μs. Each query contains multiple OUs, so these errors compound when predicting total DBMS performance. The biggest contributor to this error is contention for resources under heavy load that the offline runners do not capture. For the online models, average absolute error drops to less than 10 μs.

Lastly, we measure model convergence with an HTAP workload. We run CH-benCHmark for 15 minutes with one warehouse and 20 clients. We configure BenchBase to use 16 clients to execute TPC-C and four clients to execute CH-benCHmark queries. We use the same 5-fold cross-validation for evaluation.

The trends for the results in Fig. 10 are similar to our measurements in Fig. 9. The log serializer takes longer to converge with the CH-benCHmark workload but eventually reaches similar performance levels. The disk writer and network subsystems exhibit similar benefits of online modeling. The execution engine is again the most difficult to model. Fig. 10a shows strict improvement with

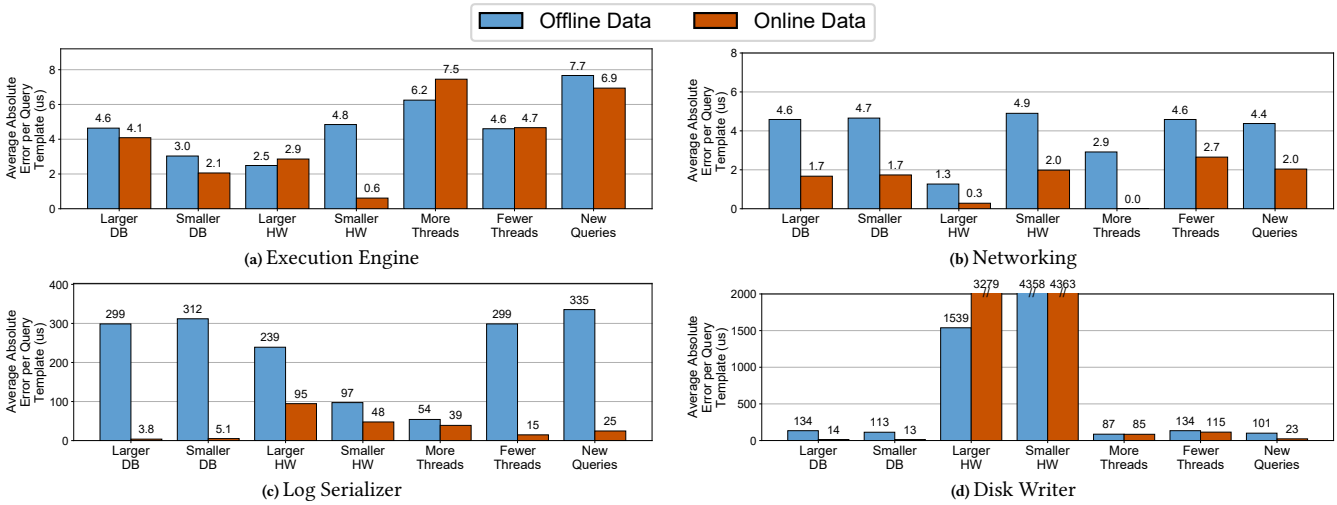


Figure 12: Model Generalization – Comparing prediction error of baseline OU models trained with offline runner data against models trained with online TPC-C data. Online data does not match configuration of test workload.

all data sizes, but the models can perform worse from one data point to the next. This may be due to overfitting and demonstrates that care is needed when training models in an online setting.

TS generates tens of thousands of training data points per second during high-performance workloads. At this rate, TS provides enough training to refine models with minimal overhead quickly.

6.6 Model Generalization

We now evaluate the accuracy of models trained using online data in various scenarios that a self-driving DBMS will encounter in real-world deployments. This experiment aims to demonstrate that the accuracy of the DBMS's behavior models' improves for their current deployment scenario, and ideally even for scenarios that the DBMS has not yet encountered. At a minimum, the online models should perform no worse in these new scenarios than their offline model counterparts. For example, we are interested if the online models generated in Sec. 6.4 overfit to their new environment.

We run the TPC-C benchmark in four scenarios that vary some aspect of the DBMS's environment: (1) database size, (2) hardware, (3) thread count, and (4) workload. For each scenario, we first execute the workload on NoisePage in an initial setting for 1 min with training data collection enabled. Next, we deploy NoisePage again in a new setting and collect more training data. We then compare how well the models trained with the first data set predict the DBMS's behavior as measured in the second set. We use NoisePage's models trained with offline data using the first setting as a baseline. We then switch the settings and repeat the evaluation to test generalization.

Database Size: We vary the size of the database by changing the number of TPC-C warehouses with a single client. The first configuration starts with online data from TPC-C with one warehouse and then evaluates TPC-C with 20 warehouses (Larger-DB). The second configuration starts with online data from 20 warehouses and then evaluates one warehouse (Smaller-DB).

Hardware: We next compare the models using training data on different hardware. For the first configuration, we collect online data from the 6-core machine running TPC-C with 20 warehouses and one client. We then measure the models' accuracy for TPC-C on the 2×20-core machine (Larger-HW). The second configuration reverses the hardware: we train the models from the 2×20-core machine and then evaluate on the 6-core machine (Smaller-HW).

Thread Count: This scenario uses the same database size and hardware, but we increase the number of concurrent threads executing in the DBMS. The first configuration executes TPC-C with 20 warehouses and one client, and then the second configuration uses 20 warehouse and 20 clients (More-Threads). The second configuration swaps the thread count (Fewer-Threads).

New Queries: Lastly, this scenario evaluates the models when the application's workload changes. One potential approach is to train the models on TPC-C in the first configuration but then switch to a different benchmark for the second. Such a setup measures the models' robustness, not the benefit of online data. Instead, we use TPC-C (20 warehouses, one client) for both configurations but train on an 80% sample of its queries (randomly selected based on query templates) in the first configuration. We then evaluate the other 20% queries in the second configuration. We use the same 5-fold cross-validation for evaluation.

Fig. 12 presents the results for these scenarios divided by subsystem. Our first observation is that offline models with small errors (<10 μs) improve in almost all scenarios. For networking models, this is for two reasons: (1) they have a small number of input features and (2) the subsystem's behavior is consistent in all scenarios. The workload does not change the model's error rate (which was already low) and online data only improves them. The execution engine models are more complex, and its models continue their performance with online data, maintaining sub-10 μs average errors. These results show that training with online data does not overfit the models to the workload and still yields robust models that generalize to unseen scenarios.

For offline models with larger errors (>10 μs), online data improves model performance in all scenarios except for two shown in

Fig. 12d. The disk writer models perform about the same when migrating to smaller hardware, but average error increases by 2× with larger hardware because the input features for this model do not capture the storage device's capabilities. This is another example of a model that would improve with hardware context [57].

As in Sec. 6.4, Fig. 12c indicates that the log serializer models improve because its behavior is mostly influenced by query arrival rate due to effects of group commit. Both the DBMS's execution engine and disk writer subsystems also incur lower model accuracy on the larger hardware scenario. The former's accuracy drops by 16% in Fig. 12a, while the latter's accuracy drops by 113% in Fig. 12d. This discrepancy reflects the hardware characteristics of the two machines and how OU input features that reflect their performance are important (i.e., hardware context [57]).

7 RELATED WORK

To the best of our knowledge, there is no prior work on training data collection for self-driving DBMSs. There is, however, an existing corpus on profiling, observability, and modeling for DBMSs and other software systems. We now discuss this prior research.

Profiling: The approaches to collect queries' profiling data differ depending on whether they target administrators versus system developers. For the former, **`EXPLAIN ANALYZE`** annotates query plans with internal metrics, like the elapsed time or the number of accessed tuples per operator. Monitoring tools, such as VividCortex [46] and Amazon's RDS Performance Insights [4], track execution times along with additional DBMS- and OS-level metrics.

Most DBMSs also provide bespoke profiling methods in their source code. PostgreSQL contains DTrace probes similar to what TS uses for query tracing, debugging, and performance analysis [44]. One could use these probes to build markers that communicate with TS's Collector, but they still need additional `FEATURES` markers to capture OU input features. MySQL contained DTrace probes in its source, but these were deprecated in v5.7 and subsequently removed in v8.0 in 2018 [40]. MongoDB has support for creating USDT probes, but this functionality appears unused [34].

In addition to profiling an individual query, most DBMSs track internal performance statistics over time. These statistics represent the user-level behavior of the DBMS across multiple queries, such as data read from or written to disk. To track timing information, DBMSs rely on portable methods using either POSIX builtins (e.g., `clock_gettime`) or language-level high-resolution clocks (e.g., `std::chrono`). Some DBMSs such as PostgreSQL [43] and MySQL [36] capture hardware metrics using tools like `rusage`. As we describe in Sec. 4, there are limitations to using coarse-grained approaches for generating high-quality training data.

Observation Services: System observability is a concept adapted from control theory that refers to the analysis of system runtime telemetry for visualizing and understanding complex software interactions [6, 19]. A common pattern is to collect all available metrics at runtime, identify a relevant subset of the metrics for a given task, and then feed those metrics into a specific model or tool.

ViperProbe [24] is an adaptive BPF-based *gray box* approach to metrics collection for Kubernetes microservices. ViperProbe uses offline analysis to identify relevant metrics for a program and then generates a BPF program to collect those metrics.

Other observability tools typically build on kernel monitoring primitives, such as OS performance counters, Strace, Ftrace, DTrace, Sysdig (which was rewritten to use BPF), or BPF. For example, Slack's Observability Data Management System [20] analyzes metrics to diagnose and resolve disruptions across multiple services. Seer [16], MicroRCA [56], and LOUD [32] are systems that gather all available metrics to identify root causes of QoS violations, locate performance issues in microservices, and localize faults in a cloud setting, respectively. Sieve [49] identifies a salient metrics and associated dependencies, but it analyzes offline. Pythia [7] is a proposed always-on automated instrumentation framework.

In comparison, TS is an adaptive *internal* approach to dynamic metrics collection. Furthermore, we designed TS to facilitate the collection of high-quality training data for self-driving DBMSs.

Modeling: Previous work in DBMS knob tuning takes a similar approach to observation services, using external features and metrics to train models about DBMS behavior. For example, OtterTune [52] and CDBTune [25] both train ML models optimizing DBMS knob configurations by using the external metrics collected from running the target workload. Similarly, other work in DBMS modeling relies on collecting training data from external sources like query logs, query plans, or custom runners. QPPNet [30] extracts execution times and execution plans from PostgreSQL's **`EXPLAIN ANALYZE`** to train models that predict the execution time of queries. ModelBot2 [29] exercises offline runners to train OUs-level models similar to TS. DBSeer [35] extracts SQL query logs to cluster queries based on their runtime behavior. DeepSketch [21] executes training queries to obtain cardinalities. GPredictor [58] extracts query plans and execution performance from the DBMS to train a graph-embedding performance model. UDO [53] evaluates its performance on sample workloads offline. Huawei's openGauss aggregates training data from three external sources: (1) database metrics, (2) SQL query log, and (3) database log [26]. Other learned system proposals like SageDB are not prescriptive in generating training data in online settings [22].

8 CONCLUSION

We presented the TScout framework for training data collection in self-driving DBMSs. We showed how it addresses the engineering challenges of collecting high-quality online training data from production DBMSs, with the flexibility to support any system architecture. With TS, developers modify their DBMS to add markers at points in the code to enable the metrics collection for four hardware categories (CPU, memory, disk, network). TS automatically receives and processes events at these markers using a kernel-space BPF program. In our evaluation using NoisePage, we showed that TS produces better training data that results in more accurate models than previous methods with minimal performance overhead.

Acknowledgments

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

VGltIEtyYXNrYSBCZXRyYXI1ZCBNZQo=

References

[1] 2021. NoisePage – Database Management System Project. https://noise.page.

[2] 2022. BenchBase: Multi-DBMS SQL Benchmarking Framework. https://github.com/cmu-db/benchbase.

[3] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE '08*. 576–585.

[4] Amazon Web Services, Inc. 2021. Amazon RDS Performance Insights. https://aws.amazon.com/rds/performance-insights/.

[5] Chris Andrews. 2017. chrisa/libusdt: Create DTrace probes at runtime. https://github.com/chrisa/libusdt.

[6] Kiam Heong Ang, G. Chong, and Yun Li. 2005. PID control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology* 13, 4 (2005), 559–576.

[7] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. 2019. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *SoCC*. 165–170.

[8] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *Proceedings of the 2008 International Conference on Management of Data (SIGMOD '08)*. 729–738.

[9] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The Mixed Workload CH-BenCHmark. In *DBTest*. Article 8, 6 pages.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.

[11] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *SIGMOD '19*. 666–679.

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

[13] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[14] Facebook. 2021. folly/tracing: Utility for User-level Statically Defined Tracing. https://github.com/facebook/folly/tree/master/folly/tracing.

[15] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*.

[16] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices *(ASPLOS '19)*. 19–33.

[17] G. Graefe. 1994. Volcano: An Extensible and Parallel Query Evaluation System. 6, 1 (Feb. 1994), 120–135.

[18] Brendan Gregg. 2019. *BPF Performance Tools: Linux System and Application Observability* (1st ed.). Addison-Wesley Professional.

[19] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *OSDI '18*. 1–16.

[20] Suman Karumuri, Franco Solleza, Stan Zdonik, and Nesime Tatbul. 2021. Towards Observability Data Management at Scale. *SIGMOD Rec.* 49, 4 (March 2021), 18–23.

[21] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating cardinalities with deep sketches. In *SIGMOD*. 1937–1940.

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

[23] Cockroach Labs. 2021. EXPLAIN ANALYZE | CockroachDB Docs. https://www.cockroachlabs.com/docs/stable/explain-analyze.html.

[24] Joshua Levin and Theophilus A. Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *CloudNet*. 1–8.

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

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

[27] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wes McKinney, and Andrew Pavlo. 2021. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4, 534–546.

[28] 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 *SIGMOD*. 631–645.

[29] 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 *SIGMOD*. 1248–1261.

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

[31] MariaDB. 2021. MariaDB Enterprise Documentation. https://mariadb.com/docs/reference/mdb/system-variables/join_buffer_size/.

[32] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. 2018. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 262–273.

[33] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. 2020. RCU Usage In the Linux Kernel: Eighteen Years Later. *SIGOPS Oper. Syst. Rev.* 54, 1 (Aug. 2020), 47–63.

[34] MongoDB, Inc. 2019. mongodb/mongo: usdt.h. https://github.com/mongodb/mongo/blob/master/src/mongo/platform/usdt.h.

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

[36] MySQL. 2007. mysql/mysql-server: sql_profile.cc. https://github.com/mysql/mysql-server/blob/8.0/sql/sql_profile.cc.

[37] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[38] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*. 677–689.

[39] Oracle. 2021. Self-Driving Database | Autonomous Database Oracle 19c. https://oracle.com/database/autonomous-database/.

[40] Oracle Corporation. 2015. MySQL 5.7 Reference Manual :: 5.8.4 Tracing mysqld Using DTrace. https://dev.mysql.com/doc/refman/5.7/en/dba-dtrace-server.html.

[41] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, et al. 2017. Self-Driving Database Management Systems. In *CIDR*.

[42] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Engineering Bulletin* (June 2019), 32–46.

[43] PostgreSQL. 2002. postgres/postgres: getrusage.c. https://github.com/postgres/postgres/blob/master/src/port/getrusage.c.

[44] PostgreSQL. 2006. postgres/postgres: probes.d. https://github.com/postgres/postgres/blob/master/src/backend/utils/probes.d.

[45] PostgreSQL. 2006. PostgreSQL: Documentation: 14: EXPLAIN. https://www.postgresql.org/docs/current/sql-explain.html.

[46] SolarWinds. 2021. Database Performance Monitor (DPM) | SolarWinds. https://vividcortex.com/.

[47] Alexei Starovoitov. 2013. LKML: Alexei Starovoitov [PATCH net-next] extended BPF. https://lkml.org/lkml/2013/9/30/627.

[48] Sthima. 2017. sthima/libstapsdt: Create Systemtap's USDT probes at runtime. https://github.com/sthima/libstapsdt.

[49] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable Insights from Monitored Metrics in Distributed Systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. 14–27.

[50] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[51] Linus Torvalds. 2012. kernel/git/torvalds/linux.git - Linux kernel source tree. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=654443e20dfc0617231f28a07c96a979ee1a0239.

[52] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. 1009–1024.

[53] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. Demonstrating UDO: A Unified Approach for Optimizing Transaction Code, Physical Design, and System Parameters via Reinforcement Learning. In *SIGMOD*. 2794–2797.

[54] Vince Weaver. 2013. The Unofficial Linux Perf Events Web-Page. http://web.eece.maine.edu/~vweaver/projects/perf_events/.

[55] Antoni Wolski. 2009. TATP Benchmark Description (Version 1.0). http://tatpbenchmark.sourceforge.net.

[56] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of PerformanceIssues in Microservices. In *IEEE/IFIP Network Operations and Management Symposium (NOMS '20)*. IEEE, 1–9.

[57] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD*. 415–432.

[58] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428.