On Embedding Database Management System Logic in Operating Systems via Restricted Programming Environments

Matthew Butrovich

CMU-CS-24-107

May 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by Google, VMware, and the National Science Foundation under award number 1822933. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: database management systems, operating systems, extensibility, eBPF

To my wife, Morgan.

Abstract

The ever-increasing improvement in computer storage and network performance means that disk I/O and network communication are often no longer bottlenecks in database management systems (DBMSs). Instead, the overheads associated with operating system (OS) services (e.g., system calls, thread scheduling, and data movement from kernel-space) limit query processing responsiveness. User-space applications like DBMSs can elide these overheads with a kernel-bypass design. However, extracting benefits from kernel-bypass frameworks is challenging, and the libraries are incompatible with standard deployment and debugging tools.

This dissertation presents an alternative implementation strategy for systems called user-bypass—a design that extends OS behavior for DBMS-specific features, including observability, networking, and query execution. Historically, DBMS developers avoid kernel extensions for safety and security reasons, but recent improvements in OS extensibility present new opportunities. Developers write safe, event-driven programs with user-bypass to push DBMS logic into the kernel and avoid user-space overheads. When a DBMS in user-space invokes these programs, user-bypass provides behavior similar to a new OS system call, albeit without kernel modifications. Alternatively, when an OS thread or interrupt triggers these programs in kernel-space, user-bypass inserts DBMS logic into the kernel stack.

In this dissertation, we will introduce three systems that use the user-bypass method in DBMSs. First, we present an observability framework that employs user-bypass to collect training data for self-driving DBMSs that reduces the number of round trips to kernel-space to retrieve performance counters and other system metrics. Next, we present a database proxy that applies user-bypass to support features like connection pooling and workload replication while reducing data copying and user-space thread scheduling. Lastly, we present an embedded that provides ACID transactions over multi-versioned data in kernel-space.

The techniques in this dissertation show user-bypass benefits across multiple DBMS design disciplines and provide a template for future DBMS and OS co-design.

Acknowledgments

I would not be a database researcher without my research advisor, Andy Pavlo. His enthusiasm, humility, and dedication inspire me as an academic, and Andy's middle-child energy never results in a dull moment. He taught me how to explore research problems, design experiments, wordsmith papers, and present our work to the broader community. Andy left an indelible mark on my life, and I cannot thank him enough for his mentorship. I will value our friendship forever.

My thesis committee comprises remarkable researchers, and I am privileged to benefit from their feedback. Jignesh Patel, Justine Sherry, and Sam Madden are the most supportive committee members I could have hoped for. Justine is a sounding board I have relied on since I came to CMU as an M.Sc. student, and her positivity is infectious. Jignesh arrived at CMU more recently and made a lasting impact on me in that time. Having another database researcher on the 9th floor of Gates is a fantastic resource; he always made time for a quick chat. Andy once described Sam as "one of the best people [he has] ever met.' He is not wrong. This document is much more robust thanks to years of feedback and encouragement from my committee.

CMU is an incredible place to be a researcher. I admire the intelligence, thoughtfulness, and passion of the students, faculty, and staff with whom this environment surrounded me. I benefitted from lessons learned by my senior Ph.D. students in the CMU Database Group: Dana Van Aken, Lin Ma, and Prashanth Menon. The Ph.D. students who arrived after me fostered a sense of community, even through years when we all worked from home: Wan Shen Lim, William Zhang, Sam Arch, and Christos Laspias. Friends in the Ph.D. program who, for some reason, decided that database research is not the ultimate calling nevertheless offered valuable perspectives: Graham Gobieski, Michael Rudow, Jack Kosaian, Nate Chodosh, Giulio Zhou, and Han Zhang.

I overlapped with too many talented M.Sc. and B.Sc. students in the Database Group to enumerate, but I thank all of them for their enthusiasm for database research. Karthik Ramanathan helped me explore the challenges of kernel-bypass in Chapter 4. John Rollinson was a fountain of operating system knowledge for Chapters 3 and 4. Tianyu Li and I spent a summer at CMU building the foundation of the NoisePage DBMS in Chapter 3, and I am a better programmer for it.

The administrators, program coordinators, managers, and other staff at CMU create order out of chaos. I am grateful for their constant patience, flexibility, and resourcefulness: Jessica Canel, Deb Cavlovich, Catherine Copetas, Joan Digney, Tracy Farbacher, Amanda Hornick, Jenn Landefeld, Karen Lindenfelser, Angy Malloy, Matthew Stewart, and Charlotte Yano. The M.Sc. program at CMU is fortunate to have two wonderful Daves: Dave Eckhardt helped me adjust to CMU's academic rigor when I arrived and first pointed me toward Andy as a research mentor. Dave O'Hallaron gave me fantastic advice throughout my years here and always had an open ear for feedback.

In addition to the Database Group, I received enormous support from the Parallel Data Lab (PDL) and Systems, Networking, and Performance (SNAP) Lab. The students, faculty, and staff of those research groups helped refine my writing, code, presentations, and research ideas. Their broad perspectives made me a more well-rounded systems researcher. In particular, Greg Ganger, George Amvrosiadis, Dave Andersen, Phil Gibbons, Chad Dougherty, and Jason Boles of the PDL were invaluable to me.

This document is the culmination of an academic journey I resumed over a decade ago. I could not have returned to school while working full-time at Western Digital without the support of my manager, Greg Nagel, and my friends there: Luca Ben, Evan Schmidt, and Dan Cristea. My community college professors at IVC, Sanjai Gupta, Seth Hochwald, and Chan Loke, brought an infectious energy to teaching. At UCI, Robin Jeffers saw academic potential in my persistence and allowed me to enroll in the School of Engineering after others turned me away. Over my years at UCI, she helped me navigate the challenges of being a part-time student in a program not designed for students like me.

I am so thankful to my family. Their support for my academic pursuits never wavered, even as the goalposts shifted from A.Sc. to B.Sc. to M.Sc. and finally Ph.D., and the years I spent apart from them grew. Their patience, compassion, and encouragement enabled my academic curiosity. I could not have found my path without them.

Finally, what I will treasure most from graduate school is meeting the love of my life, my best friend, and my wife, Morgan. She brings endless enthusiasm to my personal goals, research, and career aspirations. I cannot imagine navigating graduate school without her partnership, and I eagerly anticipate what our future holds. I must also thank our furry friends: our cat, Leland, has been a good little companion to me. Our dog, Scout, has also been a diligent research collaborator and a source of much happiness on the 9th floor of Gates. I must also congratulate Morgan on beating me to become the family's first Dr. Butrovich.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database management systems (DBMSs) form the foundation of modern applications. For example, machine learning (ML) infrastructure uses DBMSs at every stage of their pipeline [77]. Data warehouses accumulate records that originated in transactional databases [68, 92, 122, 162]. Then, analysts query the data warehouse to train a model and store the resulting model in a separate database. Querying these models accesses a feature store, the previously mentioned model database, and a vector database to generate a meaningful answer. Each DBMS's performance, reliability, and resource costs are critical to deploying robust applications.

On a multitasking operating system (OS), DBMSs run as processes with memory contents isolated from other applications through virtual memory [79]. In this scenario, the OS kernel runs as a privileged process that manages the system's memory address space and hardware devices (e.g., CPU, I/O). We refer to this privileged execution mode more generally as *kernel-space*. In contrast, application processes like DBMSs run in *user-space* where the OS provides abstractions to share system resources and maintain system stability. For example, the OS allocates CPU time between applications using processes, threads, and a preemptive scheduler. Virtual memory provides safety and security from one application accessing another's memory contents and avoids the development burden of a fixed memory address space. File descriptors provide a unified abstraction for reading and writing data to I/O devices like disks and network sockets.

The OS's application programming interface (API) for user-space applications are *system calls*—function calls that invoke kernel frameworks. DBMSs use system calls to request resources from the OS and then apply application-specific logic to access those resources optimally. Depending on their performance requirements, DBMSs may attempt to manage a resource from user-space completely, cede control entirely to the OS, or take a balanced approach between the two.

1.1 User-Space Methods

DBMSs rely on operating system services (e.g., file systems, network sockets, threads) to manage hardware resources like CPU, memory, and I/O (e.g., network, disk). Researchers discussed the shortcomings of using OS services to support DBMSs for decades. For example,

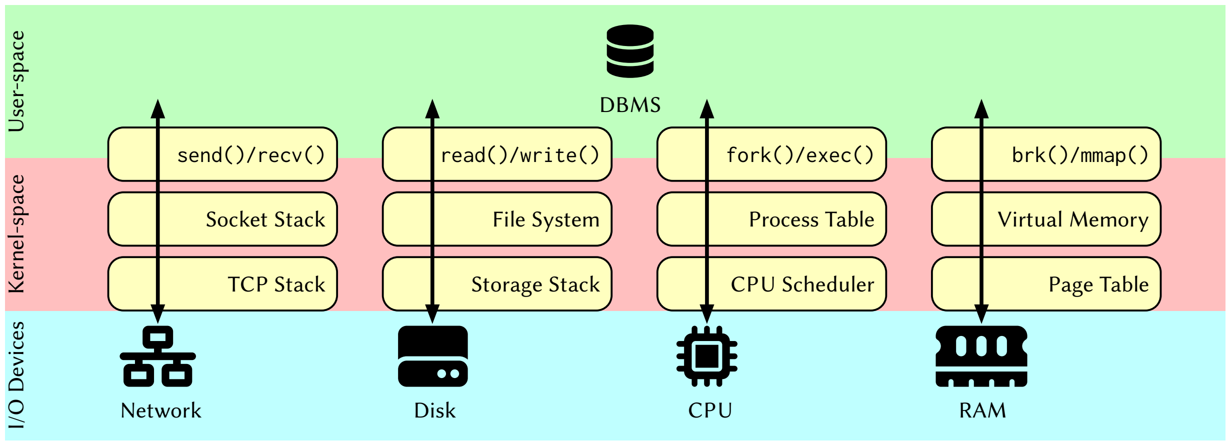


Figure 1.1: User-Space DBMS – DBMS running in user-space, accessing system hardware resources managed by OS services via system calls. For brevity, we omit some layers from each resource stack.

,

OS-level memory management and I/O interfaces make it difficult to reason about durability [90, 164, 185, 186], while thread scheduling and file system abstractions are deemed too slow [123, 142, 192]. With a monolithic OS kernel (e.g., Linux, Unix), services are general-purpose frameworks that applications interface with via system calls. Due to the large number of features they must support, kernel developers do not optimize these system calls for any one use case [28].

Figure 1.1 shows an example of a DBMS running in user-space and relying on OS system calls to access hardware resources. For each operation, the DBMS invokes a system call, enters a privileged execution mode, executes the kernel code, and then returns to user-space to continue its execution. For example, the DBMS uses the `send()` system call to send data over the network. The OS shepherds the data through multiple layers of its networking stack (i.e., socket mapping, TCP, IP, Ethernet) before issuing a write to the network adapter. Similarly, the DBMS writes storage devices with the `write()` system call, which then passes data through its file system, virtual file system, and block translation layers before sending data to the disk. The DBMS acquires CPU resources by creating processes with `fork()`, which the OS manages by scheduling fairly. Lastly, the OS abstracts system memory through the heap abstraction. DBMSs can acquire memory by extending their heap space with `brk()` or mapping anonymous pages with `mmap()`.

These systems calls and their associated layers of frameworks introduce overheads in multiple ways. For example, when saturating the Linux kernel's network stack with traffic, over 50% of the CPU cycles can be attributed to data movement (i.e., `memcpy()`) between kernel-space and user-space [83]. Linux's storage stack contains buffering layers that make it difficult for DBMSs to ensure durability while remaining performant. Due to the overhead of these software layers, a naïve blocking I/O implementation using system calls would require 86 CPU cores at 3 GHz to saturate a 3 GB/s solid-state drive (SSD) [124]. New system calls for asynchronous I/O and using direct I/O to bypass kernel buffers reduce the overheads, but these approaches

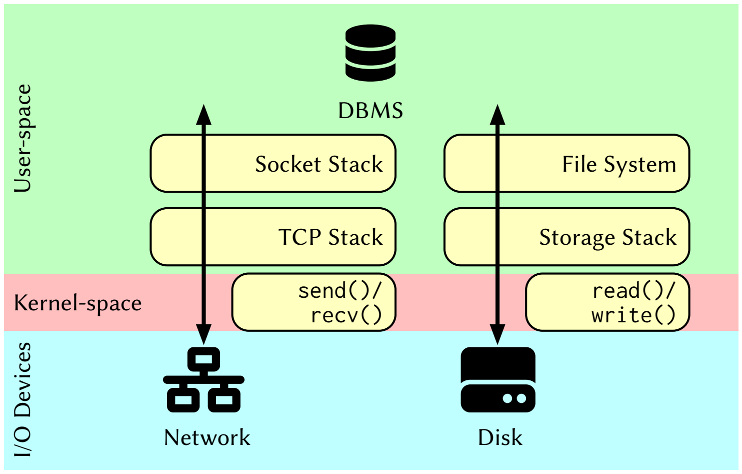


Figure 1.2: Kernel-Bypass DBMS – DBMS running in user-space, accessing system resources (e.g., network, disk) via kernel-bypass to send and receive data.

push additional complexity into the DBMS. Lastly, the virtual memory abstraction struggles to scale in modern multi-core CPUs, with a microbenchmark of memory allocations spending 79% of its CPU time in translation lookaside buffer (TLB) shootdowns (i.e., cache coherency operations) [144].

DBMSs employ domain-specific knowledge to manage system resources in user-space to improve performance. For example, coroutines are a cooperative multithreading technique that elides the preemptive multitasking interface of the OS. This design reduces the overhead associated with CPU scheduling, freeing resources for the DBMS to process queries more efficiently [126]. DBMSs manage the memory for database pages via buffer pools with custom eviction algorithms to maximize inter-query parallelism concurrency and minimize cache misses [101, 107]. These algorithms adapt to DBMS operations to prevent degenerate cases like sequential flooding, and research continues on new designs that reflect modern storage devices like SSDs and non-volatile memory [64, 143].

In addition to managing disk pages in memory with buffer pools, DBMSs apply complex user-space techniques to other memory allocations. For example, PostgreSQL defines memory contexts built on top of glibc's `malloc()` to provide memory accounting and limits across system components [198]. DBMS researchers have also demonstrated the benefits of applying custom memory allocators (e.g., jemalloc[26], TCMalloc [55]) to DBMS workloads [100]. These user-space methods improve DBMS performance, particularly for resources whose OS abstractions cannot be avoided (e.g., processes with CPU, virtual memory with RAM).

1.2 Kernel-Bypass Methods

The OS does not allow DBMSs to completely control CPU and memory allocation. However, for supported devices, the OS allows a DBMS to manage I/O resources (e.g., disk, network) with an approach called *kernel-bypass*.

Figure 1.2 shows an example of a DBMS running in user-space, using kernel-bypass to send

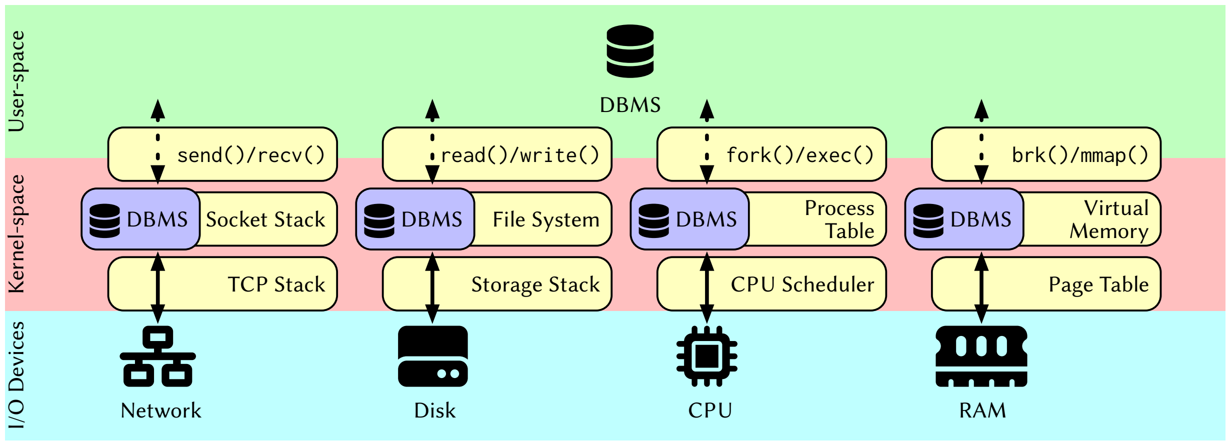


Figure 1.3: User-Bypass DBMS – DBMS running in user-space, accessing system hardware resources managed by OS services via system calls. However, user-bypass augments the kernel processing stacks with DBMS-specific logic.

and receive data between I/O devices. In this scenario, a DBMS communicates directly with the device driver to send and receive data—not using system calls and bypassing the kernel's stack. Thus, the application manages device and communication protocols and their associated state machines. For network sockets, this includes protocols like TCP and UDP, while for storage devices this includes file systems and block translation.

Despite this added user-space logic, kernel-bypass aims to improve system performance by hoisting data to user-space as quickly as possible. Once in user-space, developers can reduce their network processing stack to include only the relevant subset for their application. For example, if the target application only uses the TCP protocol to communicate, researchers have shown more efficient TCP stacks that omit logic for processing other protocols (e.g., UDP) [130].

However, achieving good performance from kernel-bypass frameworks is challenging for several reasons. First, they require a CPU dedicated to polling a ring buffer from user-space, potentially wasting CPU cycles [214]. Second, network drivers that enable kernel-bypass frequently break their application programming interface (API), requiring applications to either bundle an old driver or release updates at the same cadence as hardware vendors [161]. Lastly, kernel-bypass puts devices into an exclusive mode, making them incompatible with standard deployment and debugging tools [200].

The complexity of developing, debugging, and deploying kernel-bypass techniques in production has limited its adoption in the database community. To our knowledge, the only DBMS making significant use of kernel-bypass is Yellowbrick, which wrote its own network and storage device drivers [120]. Although ScyllaDB promotes kernel-bypass support in their Seastar framework for building high-performance data processing applications, they do not enable kernel-bypass in their production deployments [119].

1.3 User-Bypass Methods

Kernel-bypass is an all-or-nothing approach whereby the DBMS wrests complete control of a resource from the OS. As an alternative between complete reliance on the OS and kernel-bypass, we present a technique called *user-bypass*, whereby developers push DBMS logic into the kernel's stack to avoid copying data between user-space and kernel-space. Figure 1.3 shows a DBMS running in user-space that has extended the behavior of the OS stacks with its logic. User-bypass maintains the automatic resource management and compatibility of user-space system calls while providing many customization and performance benefits that kernel-bypass affords. With user-bypass, DBMSs can perform valuable work in the kernel stack without moving data to user-space, scheduling user-space threads, or calling more system calls.

The research community has proposed several methods to extend OS behavior to better suit user-space applications (i.e., OS extensibility). First introduced in 1994, SPIN [72, 73] is an extensible microkernel that interfaces with applications through a custom programming language. Through this interface, applications define their custom resource management semantics. That same year, researchers at MIT introduced the Exokernel [103, 104, 112, 132] architecture as a distinct approach to OS extensibility. With Exokernel, untrusted applications implement abstractions like virtual memory entirely in user-space.

However, these OS extensibility efforts lacked a universal API like POSIX, which made it challenging to develop cross-platform applications. Developing a DBMS for one of these early extensible OSes created vendor lock-in, imposing the compatibility limitations of a unikernel without all of the associated performance benefits.

OS developers have provided few avenues to extend kernel behavior in commonly deployed operating systems (e.g., Linux, Windows) for system reliability and safety reasons. Until recently, if a developer wanted to embed DBMS logic into the kernel, they would have to either load a kernel module or modify and recompile it. Researchers explored the challenges of safely implementing such approaches [182]. On modern systems, changing the OS requires disabling Secure Boot unless a developer can have their software signed by Microsoft—even when booting Linux [50, 51]. Secure Boot remains a valid mitigation for firmware-based "bootkit" attacks, leaving enterprises reluctant to turn it off [6].

The technology that enables user-bypass and addresses the shortcomings of past OS extensibility efforts is *extended Berkeley Packet Filter* (eBPF). This modern Linux subsystem enables developers to write safe, event-driven programs running in kernel-space [190]. However, these eBPF programs have restrictions on their execution time, kernel API, and memory safety. For this reason, it is not possible to implement every DBMS feature (e.g., SQL parsing) with user-bypass.

For developers to apply user-bypass to a DBMS, they must consider several questions. First, what DBMS operations most benefit from executing in kernel-space instead of user-space? Second, can the desired DBMS logic be expressed within eBPF's constraints? Third, what software architecture is necessary to support a user-bypass design where some DBMS logic resides in kernel-space and the rest in user-space? This dissertation explores these questions and the ramifications of their answers.

1.4 Summary of Contributions

Thesis Statement: *Embedding database management system logic in the operating system via modern kernel extension mechanisms to bypass user-space for frequently performed operations is a safe and maintainable way to improve system performance and lower deployment costs.*

This dissertation explores the different ways that DBMSs can adopt user-bypass and its benefits. Our work revisits the idea of OS extensibility for DBMSs with modern techniques, and applies user-bypass to applications that are feasible in production environments.

This dissertation makes the following contributions:

User-Bypass Observability for Self-Driving DBMSs (Chapter 3) We present a framework to collect training data for self-driving DBMSs efficiently. Collecting DBMS state and resource consumption in an online setting incurs high overhead, but user-bypass reduces the round-trips between kernel-space and user-space [80].

User-Bypass DBMS Proxy (Chapter 4) We present a DBMS proxy that uses user-bypass for the most frequently performed operations (i.e., queries and their results). This design outperforms other proxy designs, and enables low-cost deployment of a DBMS proxy [81].

User-Bypass DBMS Architecture (Chapter 5) We present a DBMS that runs entirely in kernel-space for user-bypass applications. We explore the design decisions, limitations, and example applications enabled by a user-bypass DBMS.

Chapter 6 discusses related work, and Chapter 7 overviews possible future research directions. We conclude with our final remarks in Chapter 8.

Chapter 2

Background

This chapter presents an overview of high-performance techniques for DBMSs to manage resources with the OS. We first define our user-bypass method in the context of Linux networking and existing kernel-bypass techniques. We then discuss the kernel feature that enables user-bypass and its limitations.

2.1 Kernel-Bypass

We first provide an overview of the relevant OS services for I/O to understand the differences and trade-offs between kernel-bypass and user-bypass. We provide our analysis in the context of the OS networking stack, but the storage stack contains similar software layers (e.g., block layer, virtual file system). A complete overview of Linux's networking stack is beyond the scope of this thesis; we will focus on the portion relevant to DBMSs.

Linux contains multiple layers for processing network traffic. These layers handle transport (e.g., TCP), network (e.g., IP), and link layer (e.g., Ethernet) protocols. The stack exposes a socket interface [52] for applications to copy data between user-space and kernel-space, along with traffic control [54] interfaces to configure queuing disciplines and network filters.

Using *kernel-bypass* methods, user-space applications prioritizing performance over simplicity can elide these software layers. In this scenario, a user-space application receives bytes directly from the device driver—bypassing the kernel's network stack. Thus, the application manages communication protocols and their associated state machines. The most common kernel-bypass pattern for network applications is to use Intel's Data Plane Development Kit (DPDK) [16] software library with user-space TCP implementations like mTCP [130]. Intel provides the similarly named Storage Performance Development Kit (SPDK) [60] for storage devices. Other kernel-bypass libraries (e.g., F-Stack [21] and ScyllaDB's Seastar [48]) attempt to simplify development by bundling DPDK with bespoke TCP logic.

In 2018, Linux added native kernel-bypass support with AF_XDP [127], removing the dependency on kernel modules such as DPDK. AF_XDP introduces a new socket type for user-space applications that can receive data from low levels in the kernel networking stack before the kernel allocates socket buffers or applies TCP/IP logic. This form of kernel-bypass has not yet received widespread adoption. Still, it presents a more standardized interface for future kernel-bypass efforts compared to vendor-specific kernel modules like DPDK.

Historically, kernel-bypass was the preferred way to implement high-performance Linux networking applications [21, 130, 200]. There are several reasons for this view: (1) the Linux networking stack was perceived as slow and inefficient, (2) applications performed encryption in user-space using a software library (e.g., GnuTLS, LibreSSL, OpenSSL), and (3) a lack of programmability in the networking stack. The interfaces to program the network stack were limited to filtering and routing decisions based on L2/L3 rules (e.g., `iptables`, `nftables`, `tc`). This lack of extensibility in the kernel prevented L7 DPI necessary for a DBMS proxy. Thus, kernel-bypass remained viable for network applications with complex user-space logic.

To our knowledge, the only database vendor making significant use of kernel-bypass is Yellowbrick, which built a reliable transport protocol on top of UDP using DPDK [120]. Yellowbrick's kernel bypass efforts extend to a custom memory manager to group allocations by lifetime and a task scheduler optimized to keep query data in CPU's L3 cache. Their efforts with DPDK for node-to-node improve their query execution performance by 20% [91]. Although ScyllaDB promotes DPDK compatibility in their Seastar framework for low-latency and high-performance networking [49], they do not deploy DPDK in production [119]. In general, deploying DPDK applications into production is difficult due to API/ABI instability, which forces developers to choose between features and fixes of new releases and code stability of LTS releases [161].

Kernel-bypass aims to improve performance by transferring buffers between devices and user-space applications instead of the kernel processing them. *User-bypass* is the opposite approach: the developer pushes application logic into the kernel's network stack as low as possible to avoid copying data between user-space and kernel-space, while benefiting from the kernel handling L1-L4 networking. Until recently, if a developer wanted to embed application logic into the kernel, they would have to either (1) load a kernel module or (2) modify and recompile it. Such approaches are difficult to implement and sacrifice system reliability and safety. However, updates to Linux make user-bypass a viable alternative to kernel-bypass. The efficiency of the Linux networking stack is also fast enough for most DBMS deployments. For example, recent papers showed that a single CPU core can process 42 Gbps [83], and dedicated servers can process 670 Gbps of data [63]. Next, kernel TLS (kTLS) allows developers to move encryption into kernel-space and hardware [183]. But the key reason that user-bypass is now possible is the increased programmability via eBPF, which is composable with kTLS [76].

2.2 eBPF

To understand user-bypass, we now provide an overview of how the technique embeds DBMS logic in the OS kernel. *Extended Berkeley Packet Filter* (eBPF) is the key technology that enables this functionality. This modern OS subsystem enables developers to write safe, event-driven programs running in kernel-space [190]. Application developers and cloud vendors have adopted eBPF due to its safety guarantees and features [7, 9, 12]. For example, Meta loads over 40 eBPF programs on every server, with hundreds more loaded on demand [191].

McCanne and Jacobson introduced the original Berkeley Packet Filter (BPF) library [158] in 1993. BPF enabled the execution of user-written code in the OS kernel without the complexity or safety concerns of writing kernel modules. Its developers designed BPF for high-performance network packet handling using a virtual machine (VM) with a limited instruction set. These BPF programs did not support function calls or memory allocation. They were a customizable way to express whether to allow or drop a packet to the networking stack. With only 22 instructions and minimal capabilities, developers wrote BPF programs as bytecode rather than compiling from a higher-level language.

Two decades later, the Linux kernel team introduced the "extended" BPF (eBPF) variant in 2013, which expanded its semantics and capabilities [190]. eBPF provides a more robust VM than the original BPF VM with three key features: (1) kernel-embedded data structures (referred to as eBPF maps), (2) direct access to kernel functions (via eBPF helpers), and (3) enhanced tracing features. Open-source developers also maintain a clean room implementation of eBPF for Microsoft Windows [12], but it does not support all of the features of Linux. Due to the deprecation of classic BPF, and the increasing capabilities of eBPF, we refer to these technologies as eBPF for the rest of this dissertation.

eBPF programs still use a limited instruction set, though their capabilities have increased to include function calls, loops, and the ability to tail call from one eBPF program to others. Developers typically write eBPF programs in higher-level languages like C or Rust that compile to eBPF bytecode via LLVM or GCC. Upon loading the eBPF program, modern kernels just-in-time (JIT) compile the bytecode to native machine code rather than interpreting it in a VM. eBPF program capabilities vary based on their type and attachment point, but they attach to predetermined functions in the OS stack for network processing.

Developers load eBPF programs into kernel-space and then associate them with events (e.g., functions, static tracepoints) to trigger their execution. When a running thread hits the attachment point, the execution of the eBPF program starts in privileged mode. Depending on their behavior and attachment location, developers use eBPF programs for software debugging, profiling, or modifying data flow (e.g., network buffers) through kernel-space.

2.2.1 eBPF Maps

The execution state of eBPF programs is ephemeral, meaning that its decision-making is limited to the data available during a single handler invocation. However, with *eBPF maps*, the program can maintain state across events, enabling user-bypass to support more complex application behavior. These data structures reside in kernel-space and are the primary mechanism for creating stateful user-bypass programs.

eBPF maps expose a key-value interface, though their underlying behavior, supported data types, and storage structures vary. Some maps support custom key types (e.g., hash table), and others only expect numeric indexes for their keys (e.g., array). Linux supports concurrent access to eBPF maps through internal synchronization methods, detailed in Section 5.1.2. For performance reasons, the hash table and array variants of eBPF maps provide per-CPU implementations that trade off eliminating synchronization overheads for development complexity.

Some eBPF maps provide more features than just key-value lookup, enabling more robust eBPF applications. For example, there are stacks (i.e., last-in, first-out) and queues (i.e., first-in, first-out) with peek and pop capabilities and hash tables that apply least-recently-used (LRU) semantics. Recent Linux kernel releases (i.e., v5.16 in 2022) support bloom filters, which DBMSs use to accelerate query processing [217] and optimize index storage [95].

2.2.2 eBPF Verifier

Because the CPU is in privileged mode when eBPF programs run, the kernel requires them to pass a verification step before it loads them. The *eBPF verifier* enforces kernel API compliance, memory access safety, execution bounds, and instruction count. The verifier generates a control flow graph for all possible branches of the eBPF program and enforces limits like 512 B maximum stack size and 1 M total instructions. Although these restrictions limit the complexity of application logic with user-bypass, DBMS network protocols for the most common message types (i.e., queries, results) are expressible in eBPF. Despite the verifier's guarantees, developers should consider security and safety practices with eBPF deployment due to its close interaction with kernel functions, especially in multi-tenant environments.

eBPF's verifier terminates after processing 1 M instructions and rejects loading the program if verification is incomplete. Although this creates a *de facto* limit of 1 M instructions for an eBPF program, the limit is much lower in practice due to how the verifier processes programs' conditional logic. The verifier explores all branch and loop states, exponentially increasing verifier work with nested branches or loops. For example, if a loop examines eight elements of an array, the verifier simulates the program state through all loop iterations. Any eBPF code inside the loop will count eight times toward the verifier's 1 M instruction limit. A new `bpf_loop()` helper function expresses loops as a callback function with a compile-time iteration count to reduce loops' verifier burden [10], but it is only available on newer Linux kernel releases (i.e., v5.17 in 2022).

To overcome this limit, eBPF supports *tail calls* between eBPF programs. Unlike calling a function, tail calling jumps to the new program location, overwrites the call stack, and does not return to the caller upon completion. For example, one eBPF program could parse a socket buffer and then tail call into one of many eBPF programs depending on the buffer's contents. If developers decompose their application logic into separate eBPF programs and chain them with tail calls, they can overcome the verifier's instruction limits. eBPF developers must rely on eBPF maps to carry state between tail calls. However, the verifier restricts developers to a maximum of 31 tail calls in a sequence, imposing a total verification complexity limit of 32 M eBPF instructions across 32 programs.

2.2.3 eBPF Applications

The need for eBPF grew out of high-performance network packet processing, but its applications now extend to container coordination, performance engineering, security enforcement, and more. Cilium [7] is a container networking interface (CNI) encompassing all of these aspects, allowing developers to customize their Kubernetes container environment in kernel-space. Meta drives a significant amount of Linux kernel enhancement for eBPF, and their Katran [9] load balancer manages traffic in all their data centers.

Observability has been a significant driver for eBPF adoption due to its flexibility and native integration with the Linux kernel [118]. Off-the-shelf tools like BCC [25] bundle eBPF programs to measure resources like I/O activity, CPU time, and file system behavior. For application-specific tracing, bpftrace [14] provides a high-level scripting language that generates eBPF programs at runtime, allowing developers to analyze system performance without downtime. Combining these approaches offers developers new ways to debug, test, and optimize system software [167].

Due to its safe integration with the Linux kernel, eBPF presents new ways for DBMSs to tune OS-specific knobs. In 2023, Oracle introduced bpftune [153] to automatically adjust network stack parameters with the eventual goal of extending its behavior to other knobs. Researchers recently explored dynamically adjusting OS mutex policies at runtime using eBPF [171, 172]. We discuss other research works in detail in Chapter 6.

Chapter 3

User-Bypass Observability for Self-Driving DBMSs

Self-driving DBMSs seek to automate the arduous tuning and optimization tasks for databases [38, 173, 174]. 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 [93], knob tuning [202]), 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* [152]. 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 [99, 155, 202]. To avoid copying the database, another method uses hand-written "runners" that execute queries in an offline environment [152]. 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 chapter presents the **TScout** (TScout) 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 eBPF [118]), and application-level counters. TScout combines these metrics with descriptors of DBMS behavior during query execution and internal maintenance tasks to create training data for machine learning models. TScout uses code generation to create a kernel-level program that collects metrics tailored to the DBMS.

We integrated TScout with the **NoisePage** DBMS [36] 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 TScout 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 TScout reduces the error for NoisePage by 98% over existing approaches.

This chapter is organized as follows. We first discuss in Section 3.1 training data collection challenges. Section 3.2 presents the TScout framework, followed by how to collect metrics in Section 3.3 and engineering issues in Section 3.4. Lastly, we present our evaluation in Section 3.5.

3.1 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.

3.1.1 Self-Driving DBMS

A self-driving DBMS's architecture is comprised of three components: (1) forecasting system, (2) behavior models, and (3) planning system [174]. The *forecasting system* is how the DBMS observes and predicts the application's future workload [151]. The DBMS then uses these forecasts with its *behavior models* to predict its runtime behavior relative to the target objective (e.g., latency, throughput) [152]. 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 [155] 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 [152].

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.

3.1.2 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 [155]. 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 [106].

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.

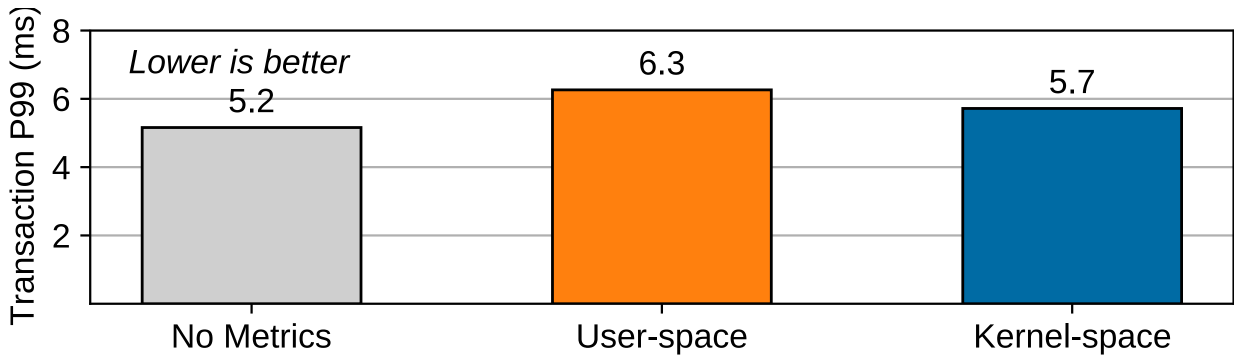


Figure 3.1: User-space vs. Kernel-space Metrics Collection – Transaction latency of TPC-C with (1) DBMS metrics collection disabled, (2) metrics collected in user-space, and (3) metrics collected in kernel-space using eBPF.

3.1.3 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` [155]. In addition to `EXPLAIN`'s limitations that we describe in Section 3.1.2, `EXPLAIN ANALYZE` does not return query results on the client, and the instrumentation imposes runtime overhead to query execution [42]. 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 (**eBPF**) library [190]. eBPF 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 eBPF validator strictly limits a program's length, permissible operations, and storage allocation. But eBPF programs can execute OS-level tasks efficiently due to their privileged execution mode. Developers compile their programs to eBPF bytecode and then load them into the kernel. During this loading step, the eBPF 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 eBPF 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 eBPF code. Unlike some syscalls to retrieve metrics, the OS does not preempt eBPF programs, which yields more predictable performance for the DBMS.

When a eBPF program runs, it can inspect kernel data structures, explore the DBMS's address space, accumulate metrics into eBPF maps, and communicate with the DBMS in user-space. Upon completion, the thread resumes the DBMS's original execution path. eBPF 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 eBPF 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 eBPF. We run the workload with a single client to reduce the effects of contention on any shared data structures. The results in Figure 3.1 show how metrics collection using eBPF 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 eBPF reduces the tail latency of queries compared to the user-space approach. The eBPF approach generates the same data as user-space syscalls, but with fewer execution mode switches, resulting in better performance.

3.1.4 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 [99, 155, 202]. Although creating database snapshots has become easier in recent years, it still requires a new DBMS instance for the clone, which

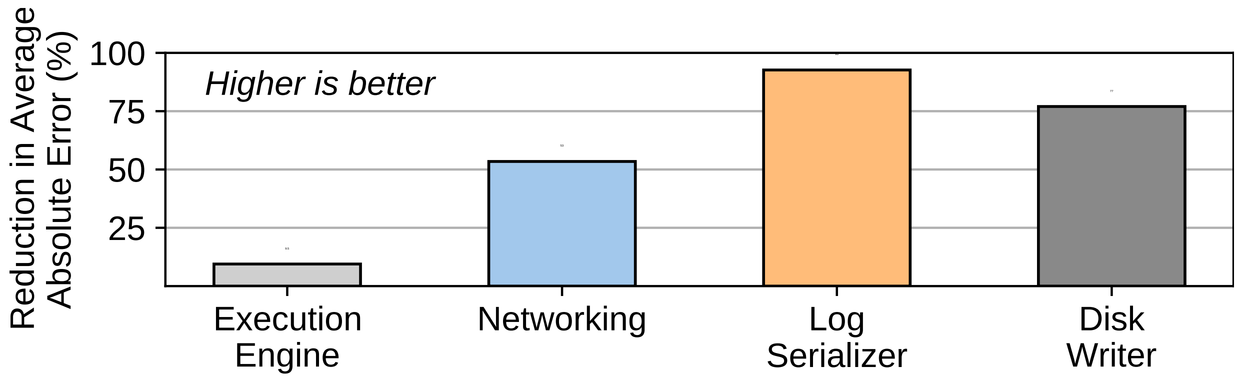


Figure 3.2: Offline vs. Online Training Data – Accuracy measurements of behavior models trained with offline and online data when predicting the execution time of TPC-C queries.

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* [152]) 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 [152]. 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 Figure 3.2 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.

3.2 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 (TScout) training data collection framework. TScout facilitates the recording and processing of OU-granular training data from a multithreaded 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. TScout collects output metrics in kernel-space using eBPF with minimal performance impact without sacrificing measurement accuracy. When it is not feasible to use kernel-space methods, TScout also supports user-space metric collection and makes it easy to combine metrics collected from different approaches.

The architecture overview in Figure 3.3 shows that TScout'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. TScout then extracts these markers and codegens a custom program for interfacing with the DBMS and collecting training data in the **Runtime Phase**.

We designed TScout with two key properties to achieve non-intrusive instrumentation for training data collection. The first is that TScout does not produce back pressure on the DBMS. Although there is a small, unavoidable overhead to collecting data, TScout does not add blocking synchronization mechanisms on critical paths. The second property is that it supports fine-grained, dynamic collection. TScout 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 TScout framework in more detail. We first describe how developers integrate TScout 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 TScout collects hardware resource data in Section 3.3.

3.2.1 Setup Phase

TScout assumes that the target DBMS uses an internal approach to autonomous planning [152]. Thus, before deploying the DBMS with TScout, a developer must first modify the system's source code to indicate when, where, and how TScout 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 TScout to monitor. TScout's Compiler uses this information to generate the Collector's eBPF code (Section 3.2.2).

Markers: TScout provides its markers as statically-defined tracepoint macros [20, 27, 66]. 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 TScout to instrument the OU when the program starts.

eBPF supports other invocation methods (e.g., uprobes [29]) 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 [168] further obfuscate system behavior, making instrumentation less reliable.

TScout 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 TScout to enable metrics collection for that thread. The framework then disables collection when that same thread encounters a `END` marker. TScout maintains an internal state machine to handle situations when markers are in an unexpected order (Section 3.4.1).

The third marker type (`FEATURES`) is how the DBMS records the input features and any user-level metrics for each OU's behavior model (Section 3.1.1). In most cases, the features of an OU are known before execution (i.e., an OU's inputs describe the amount of work 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, TScout treats input feature generation as a separate event after OU execution.

Figure 3.3 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.

TScout 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, TScout needs to toggle collection on and off at runtime on a per query basis. TScout 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 TScout (e.g., total memory used by an OU over multiple allocations). TScout 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 Section 3.4.3.

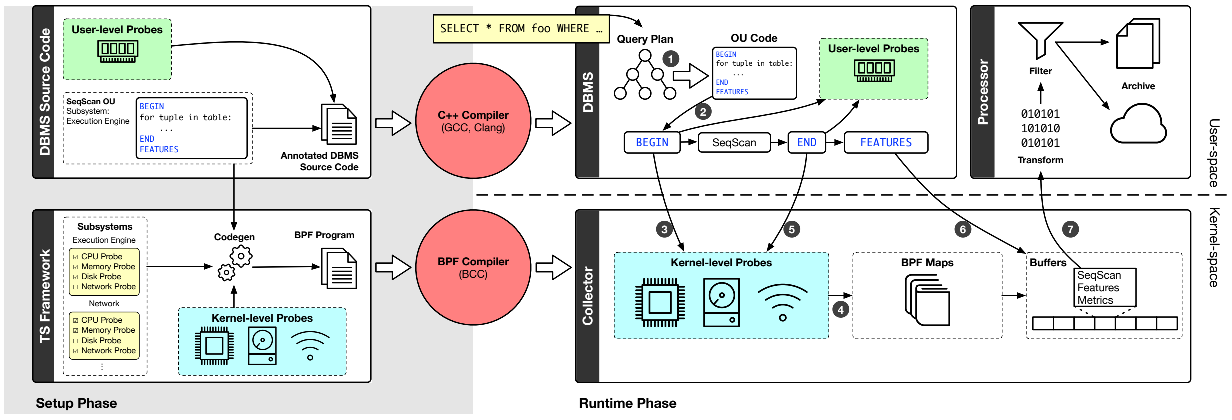


Figure 3.3: Framework Overview – TScout's architecture is split into two phases. In the Setup Phase, a developer annotates the DBMS's source code. Then, TScout uses code generation to create a customized Collector for the Runtime Phase. During the Runtime Phase, the Collector retrieves metrics and the DBMS's OU input features to create training data. Finally, these data are shipped to the Processor for archiving with other training data.

Codegen: After the developer adds markers to the DBMS's source code, TScout then extracts their embedded metadata from the DBMS to determine which metrics it needs to collect per subsystem. For the example shown in Figure 3.3, 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 TScout's built-in probes whereas the memory metrics will come from a developer provided probe. TScout then generates the source code for a eBPF program to create the Collector component. As we describe in Section 3.2.2, 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 eBPF compiler (BCC).

3.2.2 Runtime Phase

The administrator deploys the DBMS together with TScout on the same server. Using the example in Figure 3.3, 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 TScout supports multithreaded 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 TScout to enable metrics collection using its probes.

We next describe how TScout coordinates the retrieval of metrics from its probes with its Collector component. We then discuss how TScout 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 TScout's Codegen component, the Collector also includes data structures to store intermediate metrics.

❹ The Collector uses a eBPF 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 eBPF 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 eBPF 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: TScout'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 Figure 3.3, the framework contains a probe to retrieve the number of CPU cache misses during the sequential scan OU.

TScout 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 (Section 3.3). TScout'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 Figure 3.3 is an example of a user-level probe because the developer writes the code to perform metrics retrieval from inside the DBMS. TScout 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, TScout 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 [117] 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 TScout will overwrite samples if it is full.

3.3 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. TScout 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 I/O 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 TScout is either (1) user-level or (2) kernel-level.

TScout provides kernel-level probes written in eBPF for measuring CPU, network, and disk activity. Section 3.2.2 describes how TScout 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, TScout 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 TScout's Collector.

In addition to user-level and kernel-level metrics, hardware devices can expose counters (e.g., CPU PMU, disk SMART stats). TScout 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 TScout's probes extract metrics per hardware category.

3.3.1 CPU Probe

TScout'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. TScout use the `perf_event` functions of eBPF to retrieve this information. These functions are a stable API provided by the Linux kernel that supports multiple architectures [206]. The framework could also access `perf_event` in user-space; we measure the trade-offs between different access methods in Section 3.5.2.

At the `BEGIN` marker, TScout's CPU probe reads `perf_event` counters, normalizes their values, and stores them in the Collector's eBPF 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. TScout 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 eBPF map, computes their difference, and stores the final metrics back in the eBPF map.

3.3.2 Memory Probe

TScout's memory probe is the only user-level probe in the framework. The developer instruments their OUs to collect their memory consumption for TScout. 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 eBPF. Lastly, the abstraction layers of memory allocators and virtual memory make it difficult to assign ownership to an OU from eBPF. 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, eBPF probes would force every allocation to trap into kernel-space, which would be unbearably slow. TScout's query sampling reduces overhead, but it prolongs the time to generate sufficient training data. TScout'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, TScout 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. TScout 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 [156] and queries like `EXPLAIN ANALYZE` [15].

3.3.3 Network Probe

TScout 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 TScout'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 TScout, a DBMS needs additional code to respond to the socket results and accumulate metrics. With TScout, 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 TScout's eBPF 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, TScout's network probe limits the transition overhead to just the mode-switch and extracts socket statistics by reading kernel data structures (`tcp_sock`).

3.3.4 Disk Probe

TScout'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 I/O. Like for network activity, this probe could track this information in user-space based on the results of I/O syscalls, but TScout 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.

TScout'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, TScout'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` I/O accounting struct. One can extend the probe to support tracking I/O for multiple files by using their descriptors as marker parameters.

3.4 Engineering

We now present additional details and considerations of TScout's implementation that we did not cover in Sections 3.2 and 3.3.

3.4.1 eBPF Development

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

Most of these limitations come from eBPF'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. eBPF does not allow dynamic allocation other than in eBPF maps, and it restricts pointers to a safe API. Because TScout reads kernel data structures to extract runtime metrics, TScout validates kernel memory accesses before dereferencing. The verifier also restricts the total length of the program to 1 M instructions. This is not a problem for TScout as its compiled eBPF programs only contain 100s of instructions.

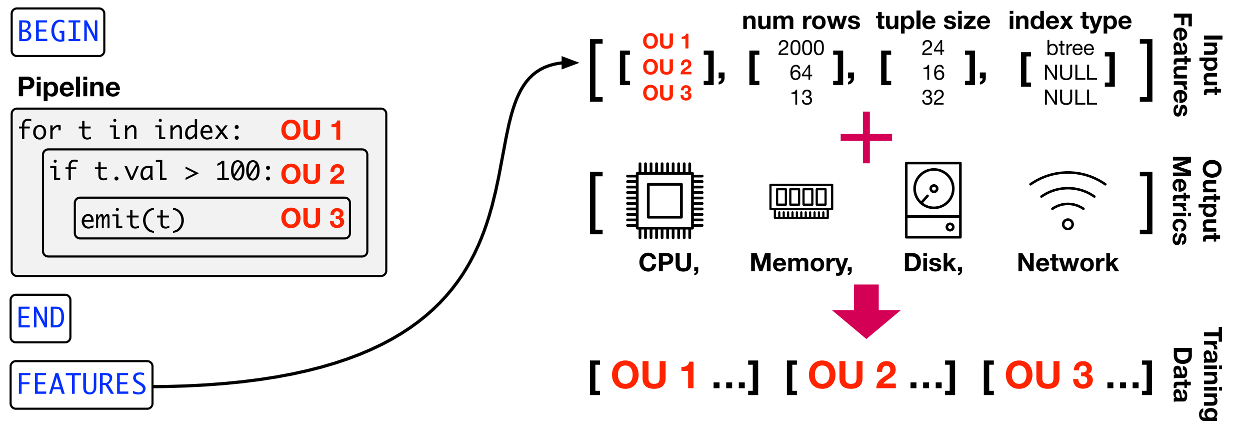


Figure 3.4: Query Engine Integration – TScout's vectorized input features supports collecting training data from a JIT compilation execution engine with a pipeline containing multiple OUs.

eBPF tooling is limited, and messages from the eBPF compiler and verifier do not always make errors obvious. TScout simplifies eBPF development by using its Codegen component to generate high-level C code that uses the BCC library to generate eBPF code. As described in Section 3.2.1, DBMS developers select the hardware resources to monitor for an OU, and TScout automatically composes probe code.

TScout benefits from eBPF constraints by enforcing a strict state machine. If TScout 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 TScout deems incorrect (e.g., out-of-order markers) and (2) exceptional control flow due to client request to the DBMS.

3.4.2 Query Engine Integration

TScout'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 [168]. The problem, however, is that tracepoints in this dynamic code are not known when TScout's Codegen component creates its eBPF 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.

To overcome this problem, TScout supports recording the input features for multiple OUs that the DBMS executes together. In the example shown in Figure 3.4, 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 TScout.

Recursive Operators: Another problem is when an operator invokes itself, causing TScout to encounter the same `BEGIN` marker twice before a `END` marker. This can occur in a Volcano-style [117] 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 TScout's Collector to maintain intermediate results using a stack eBPF 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.

3.4.3 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. TScout 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.

3.4.4 Dynamic Feature Selection

A key challenge in a training data collection framework like TScout 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 [202]. 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 TScout supports dynamic selection of internal features without restarting in two ways: (1) TScout's Collector runs in the same address space as the DBMS, so it has access to all information available at deployment. (2) TScout can dynamically unload eBPF programs, modify them, and reload them. With this capability, developers can change the internal features to collect, restart TScout, and generate new models from the training data. We defer the problem of using TScout to automatically identify which features to include in a DBMS's OU models as future work.

3.5 Evaluation

To evaluate the efficacy of the TScout framework, we integrated it in the NoisePage DBMS [36]. NoisePage is a PostgreSQL-compatible DBMS that uses HyPer-style MVCC [169] over Apache Arrow in-memory columnar data [149]. The original version of NoisePage uses OU-level behavior models that it trains via offline runners with user-level probes [152]. We modified NoisePage's source code to introduce TScout's markers and probes for its OUs.

We deployed NoisePage on a server with 2×20-core Intel Xeon Gold 5218R CPUs, 192 GB DRAM, and Samsung PM983 SSD. For the experiment in Section 3.5.6 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 eBPF.

In each experiment, we deploy NoisePage and TScout's eBPF-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 (Section 3.4.2).

When evaluating the performance of OU models trained with data from TScout, 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.

3.5.1 Workloads

We use the following workloads in our evaluation. All queries execute over JDBC using the BenchBase framework [5, 96].

YCSB: The Yahoo! Cloud Serving Benchmark [88] 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 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 [65, 82]. 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 [209]. 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 [196]. 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 [87]. We use four threads to execute TPC-H queries and 16 threads to execute TPC-C transactions.

3.5.2 Runtime Overhead

We first measure to what extent collecting training data with TScout reduces the DBMS's performance. We modified NoisePage to use three collection methods with TScout: (1) kernel-level probes with continuous `perf` counters (user-bypass), (2) user-level probes with dynamic `perf` counters (user-space *toggle*), and (3) user-level probes with continuous `perf` counters (user-space *continuous*). For the first and last methods, TScout enables its probes even if the framework never retrieves the data (e.g., TScout 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 eBPF's access API.

This experiment enables data collection for all DBMS subsystems to ensure the maximum impact. We sweep the TScout'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 TScout generates. We execute the workloads described in Section 3.5.1 with 20 client threads. We use user-bypass 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 Figure 3.5 show that the DBMS's performance drops as the sampling rate increases for each method. This reduction is because the DBMS, OS, and TScout spend more time collecting and processing training data as the rate increases. In the user-space *toggle* configuration when the sampling rate reaches 100%, the DBMS's throughput drops by almost 50%. The user-space *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-space *continuous* only requires a single syscall to retrieve `perf` counters, it incurs at most a 15% reduction compared to the other methods. user-bypass requires

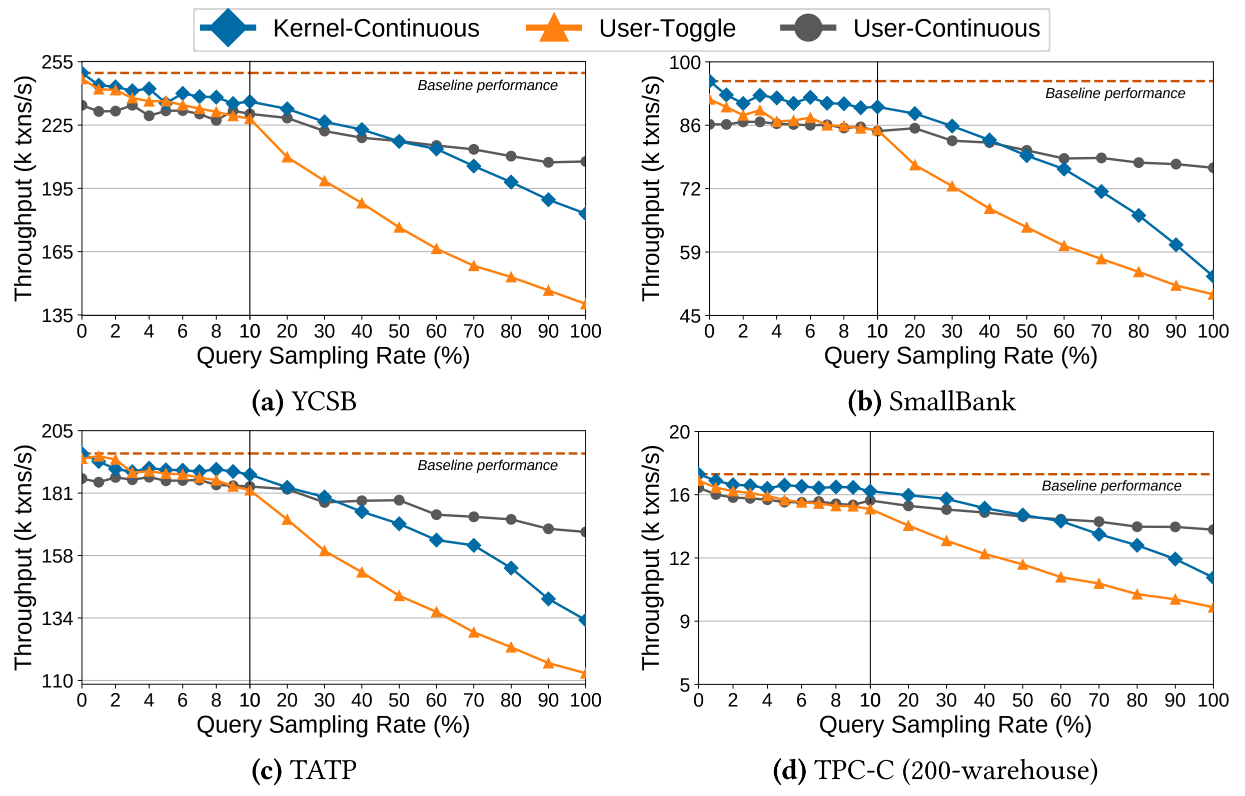


Figure 3.5: Runtime Overhead (Transaction Throughput) – Impact of query sampling on OLTP transaction throughput, comparing user-space and kernel-space approaches to system metrics collection.

multiple syscalls to read `perf` counters, which slow the system down despite the syscalls occurring under a single switch in execution mode. user-space *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-space *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 TScout 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.

Figure 3.6 shows TScout's data generation throughput increases as the sampling rate increases but only up to a point. For user-space *toggle* and user-space *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, TScout generates data using user-bypass at a 3× higher generation rate than the other two methods. user-bypass 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 user-bypass's advantage to its in-kernel data structures with efficient RCU synchronization [160].

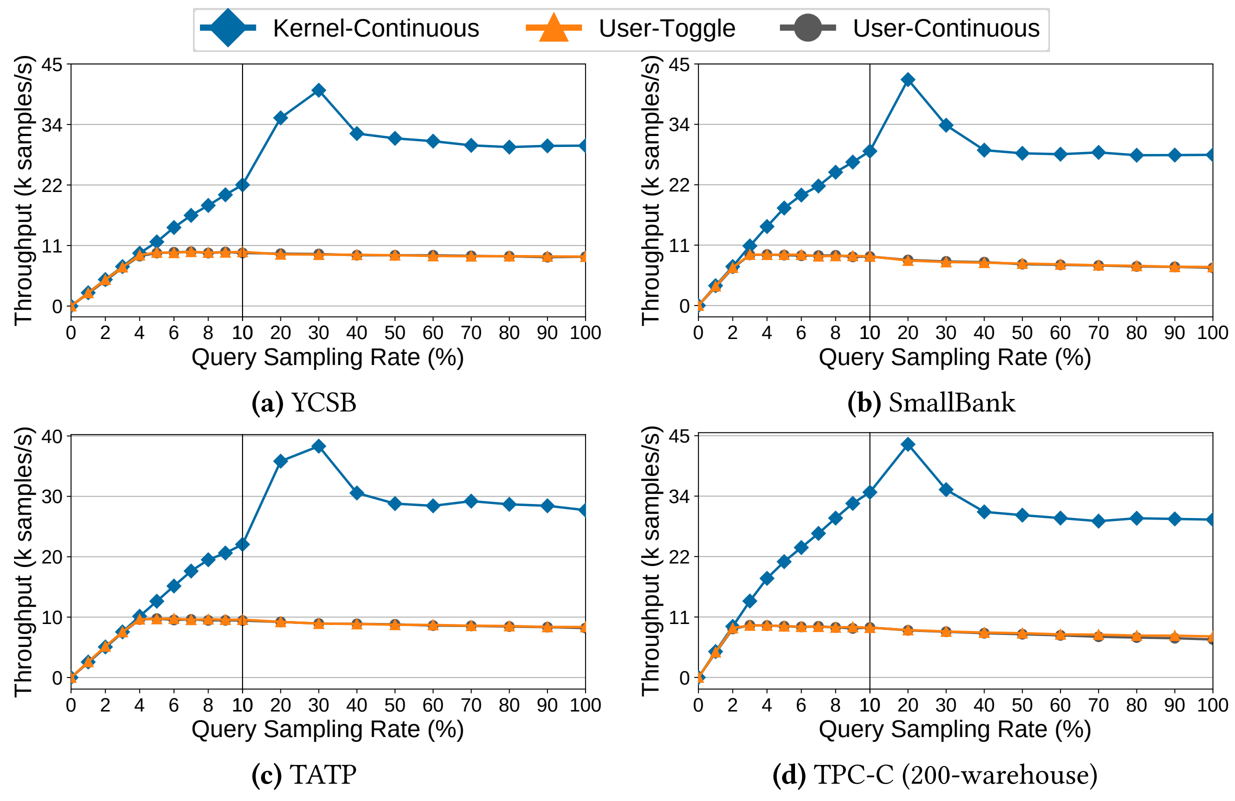


Figure 3.6: Runtime Overhead (Training Data Generation) – Impact of query sampling on OLTP training data generation, comparing user-space and kernel-space approaches to system metrics collection.

The results in Figures 3.5 and 3.6 show that the ideal configuration for TScout is to use user-bypass with a sampling rate of 10%. This setup yields the maximum throughput for both the application's workload and training data generation.

3.5.3 Adjustable Sampling

The previous experiment evaluated TScout's overhead when collecting training data for all the DBMS's subsystems simultaneously. As described in Section 3.4.3, TScout also has per-subsystem sampling that allows it to collect training data only for the OUs whose models require refinement. Such flexibility further reduces TScout'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 TScout's subsystem sampling rates at runtime. TScout starts with a 0% sampling rate. After 60 sec, TScout 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, TScout adjusts the execution engine and networking sampling rates to 0% to simulate a scenario where TScout has generated enough data. TScout maintains its 10% sampling rate for the other two DBMS subsystems.

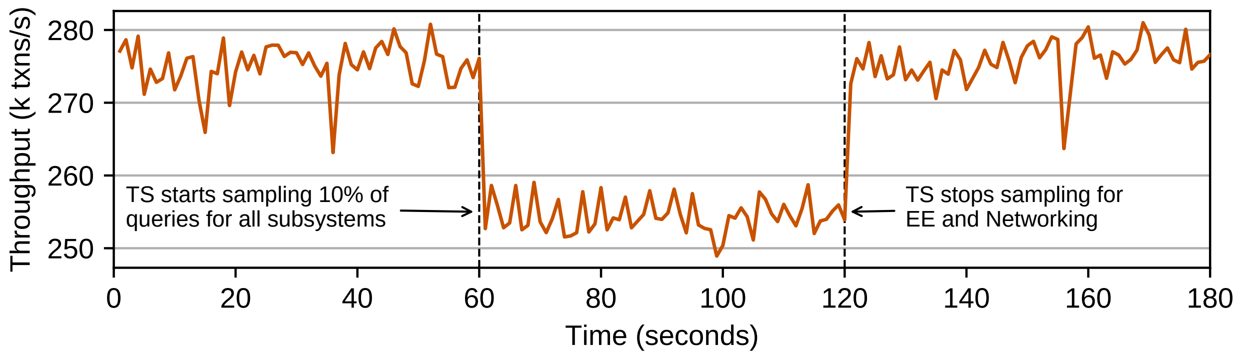


Figure 3.7: Adjustable Sampling – Impact of training data sampling on YCSB transaction throughput.

As shown in Figure 3.7, the DBMS's throughput drops by ~7% when TScout starts training data collection for all subsystems. But the DBMS's returns to its original throughput when TScout disables collection for the system's execution engine and networking. Although TScout keeps collecting data for the DBMS's WAL-related subsystems, the workload is read-only, and TScout does not generate much training data and imposes minimal overhead to the DBMS's throughput.

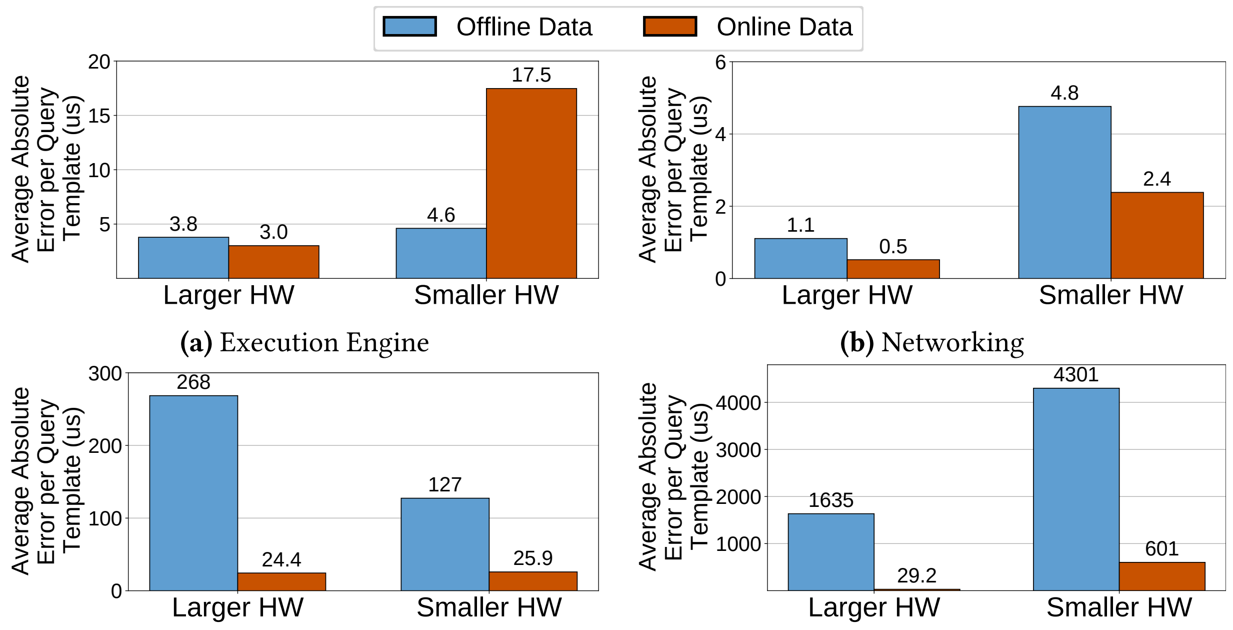
3.5.4 Adapting to Environment Changes

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 TScout's collection for 1 min while executing TPC-C (20 warehouses, one client). During this time, TScout 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 Figure 3.8 show that online data helps in nearly every scenario across all subsystems. The largest improvement is NoisePage's disk writer (Figure 3.8d), 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



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

Figure 3.8: Adapting to Environment Changes – Comparing prediction error of baseline OU models trained with offline runner data against models trained with online TPC-C data.

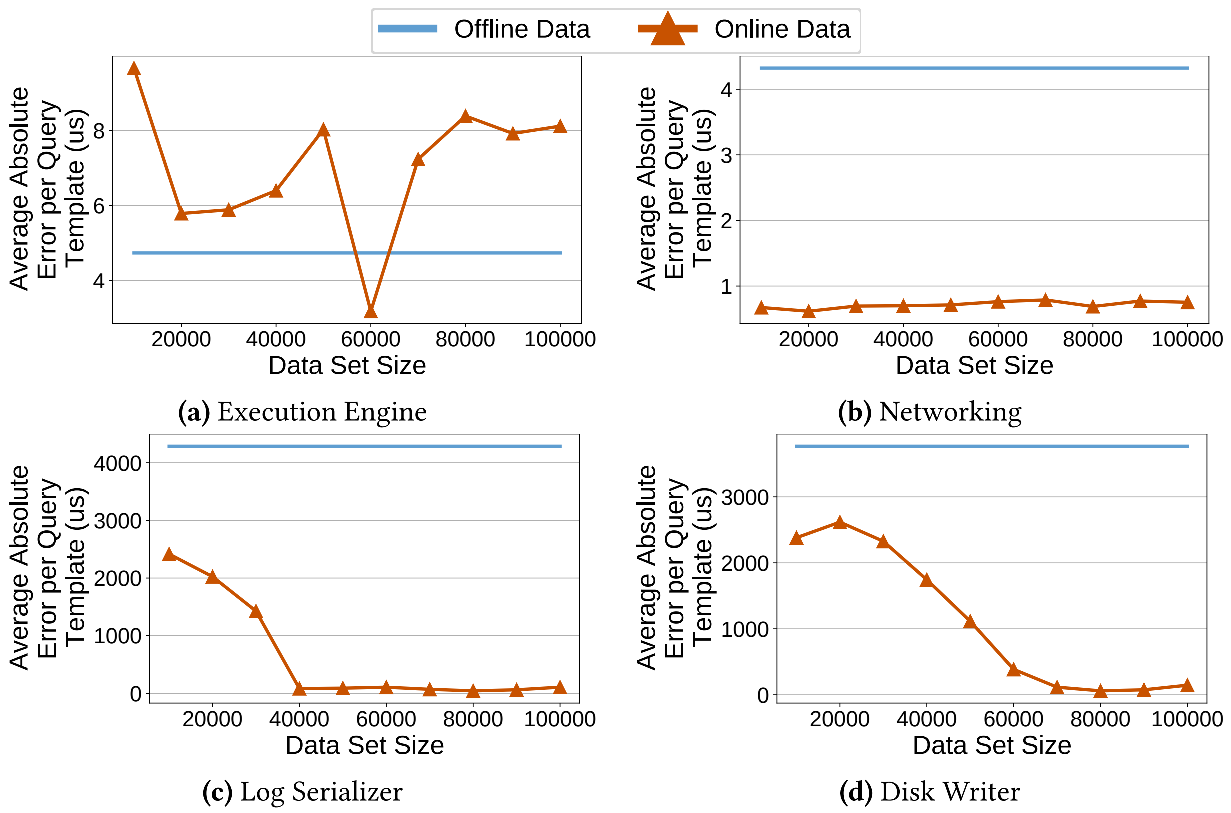
on smaller hardware in Figure 3.8a. We see a similar but a less pronounced reduction when running this same experiment on other machines that are slightly less powerful than our 2×20 machine. We believe that this diminution is due to a shortage of features describing the CPU for the models to generalize 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.

3.5.5 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 Section 3.5.2, TScout 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 TScout'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 Figure 3.9 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



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

Figure 3.9: 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.

the baseline accuracy of NoisePage's offline models. Figure 3.9c 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 I/O. 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 Figure 3.9d. This subsystem is workload dependent like the log serializer, but also shows how online data benefits scenarios where the I/O device changes. The networking models shown in Figure 3.9b 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 Figure 3.9a 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 [152], 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 Figure 3.10 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 Figure 3.9. As the number of clients increases, the offline models are less

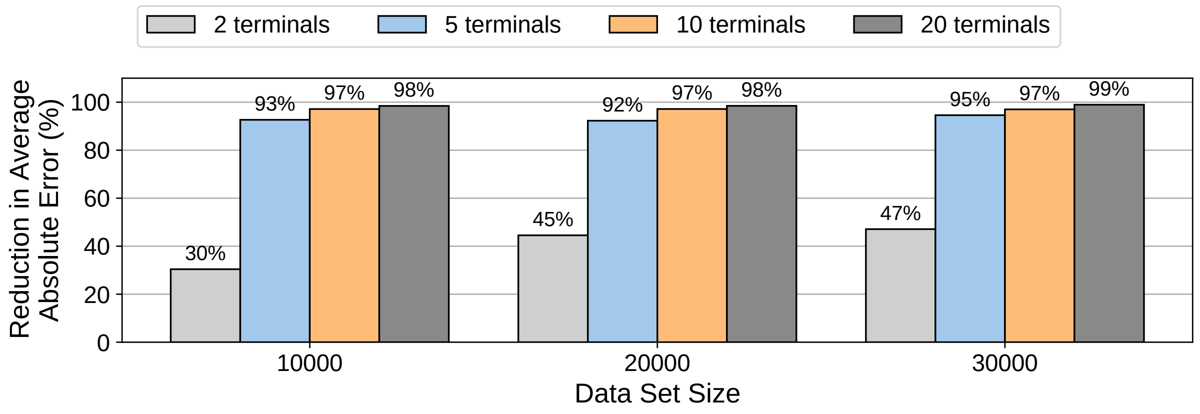


Figure 3.10: Model Convergence (TPC-C) – OU model accuracy improvement for the execution engine with increasing number of workers. Higher bars show a greater reduction in error.

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.

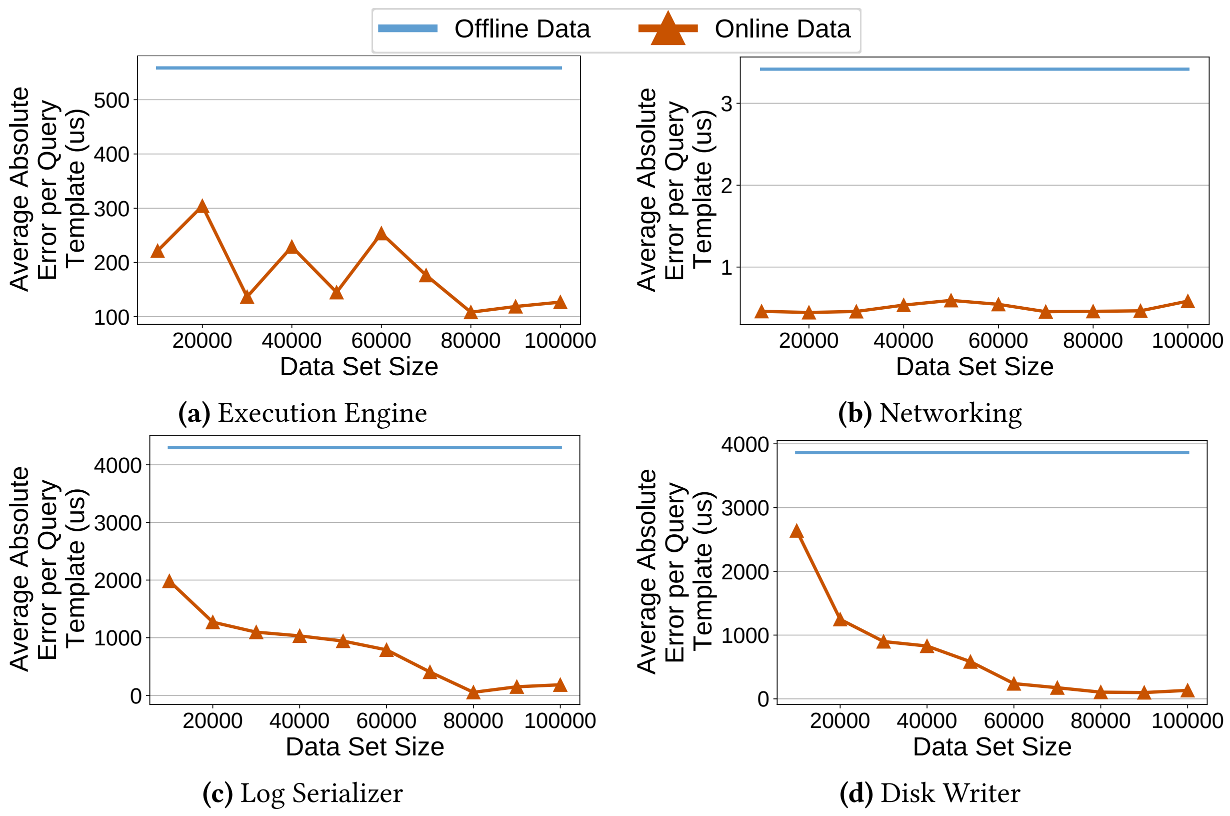
Lastly, we measure model convergence with an HTAP workload. We run CH-benCHmark for 15 minutes with one warehouse and 20 clients. We configure BenchBase to use 16 clients to execute TPC-C and four clients to execute CH-benCHmark queries. We use the same 5-fold cross-validation for evaluation.

The trends for the results in Figure 3.11 are similar to our measurements in Figure 3.9. 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. Figure 3.11a shows strict improvement with 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.

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

3.5.6 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



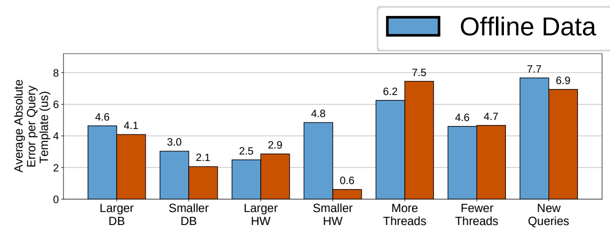
(a) Execution Engine **(b)** Networking **(c)** Log Serializer **(d)** Disk Writer

Figure 3.11: 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.

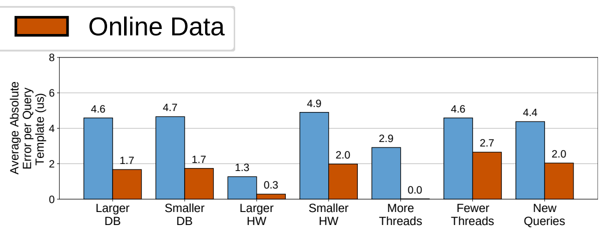
counterparts. For example, we are interested if the online models generated in Section 3.5.4 overfit to their new environment.

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.

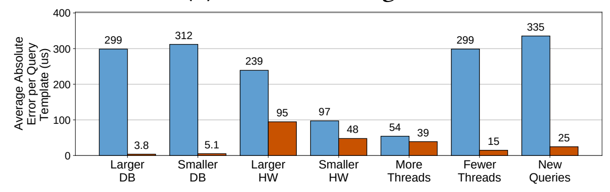
Database Size: We vary the size of the database by changing the number of TPC-C warehouses with a single client. The first configuration starts with online data from TPC-C with one warehouse and then evaluates TPC-C with 20 warehouses (Larger-DB). The second configuration starts with online data from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



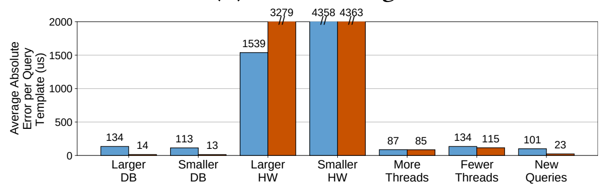
(a) Execution Engine



(b) Networking



(c) Log Serializer



(d) Disk Writer

Figure 3.12: 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.

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.

Figure 3.12 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 Figure 3.12d. 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 [213].

As in Section 3.5.4, Figure 3.12c 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 Figure 3.12a, while the latter's accuracy drops by 113% in Figure 3.12d. 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 [213]).

3.6 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 TScout, 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). TScout automatically receives and processes events at these markers using a kernel-space eBPF program. In our evaluation using NoisePage, we showed that TScout produces better training data that results in more accurate models than previous methods with minimal performance overhead.

Chapter 4

User-Bypass DBMS Proxy

Modern cloud applications often connect to database management systems (DBMSs) over a network in a manner that is not optimal for performance: either maintaining large numbers of persistent, mostly idle connections or rapidly churning connections. Both scenarios cause the DBMS to squander resources on state information for each connection or perform onerous and redundant network operations. One standard solution to overcome this problem is to use a *proxy* that acts as a middlebox [84] between the front-end client and back-end DBMS. Examples include AWS RDS Proxy [47], PgBouncer [39], Pgpool-II [40], MaxScale [30], and ProxySQL [45]. Although their features vary, these proxies all implement a target DBMS's network protocol so that clients can connect to them without needing to modify application code.

Using a DBMS proxy as the authoritative connection manager for the back-end DBMS provides multiple benefits. The proxy maintains a persistent connection pool to the back-end DBMS to multiplex client requests and responses — minimizing the number of connections talking to the DBMS. This optimization reduces the overhead of tasks that scale linearly with the number of clients like transaction commit. Furthermore, because the proxy's pooled connections only authenticate with the DBMS once, it prevents the DBMS from repeatedly performing connection setup procedures and frees system resources for query execution.

But the need to support DBMS network protocols imposes constraints on proxies' implementations. As Application Layer (L7) [94] middleboxes, these proxies follow the same design: they are user-space applications that read byte streams from client sockets, extract connection state, match the client to a back-end socket, and forward messages to the correct destination. This design incurs significant overhead from the repeated operating system (OS) system calls to read and write to sockets. Some high-performance systems have previously used kernel-bypass to elide the OS networking stack [48]. Unfortunately, this approach increases engineering and deployment complexity. To our knowledge, no DBMS proxies employ this technique; they are inefficient user-space applications that do not scale effectively for the most demanding applications.

Given this, we present an alternative system architecture called **user-bypass**. User-bypass relies on eBPF programs that push down DBMS-specific connection logic into the OS to create fast paths in the Linux networking stack. User-bypass elides expensive system calls and buffer copying without sacrificing functionality or safety. We are not the first to embed L7 logic into the kernel using eBPF [113, 140, 184]. However, to our knowledge, we are the first to articulate the idea of "user-bypass" instead of "kernel-bypass" and the first to deploy this design strategy to DBMS proxies. To demonstrate the effectiveness of this approach, we implemented a PostgreSQL-compatible proxy called **Tigger** using user-bypass. It offers efficient connection pooling and workload mirroring within kernel-space. Tigger supports the target DBMS's network protocol and is a drop-in replacement for other proxies.

We compare Tigger's performance against other open-source proxies for online transaction processing (OLTP) workloads. Our results show that Tigger's user-bypass architecture is the most responsive and reduces transaction latency by up to 29%. Tigger also minimizes CPU utilization — offering the best performance by being 42% more efficient than the next proxy. When comparing Tigger's workload mirroring capabilities, it provides an average of 92% lower transaction latency while using 88% less CPU.

Our work makes the following contributions: (1) a discussion of DBMS proxies in the context of modern cloud applications, (2) the user-bypass method for pushing DBMS logic into kernel-space, and (3) the design of the Tigger DBMS proxy using user-bypass.

This paper is organized as follows. Section 4.1 describes network connection challenges, how DBMS proxies alleviate those issues, and the shortcomings of existing designs. Section 4.2 introduces user-bypass and the technologies that enable it. Section 4.3 presents the Tigger architecture, followed by a discussion of its implementation in Section 4.4. We then present our experimental evaluation in Section 4.5. We provide concluding remarks in Section 3.6.

4.1 Background

We begin with motivating the need for DBMS proxies. We present the challenges of scaling the number of persistent DBMS clients and their overhead, and then describe how proxies use pooling to improve DBMS efficiency. Lastly, we discuss existing proxies implementations and detail their scalability limits in cloud environments.

4.1.1 Connection Scaling

Modern cloud application frameworks perform horizontal scaling in response to changes in load, creating more instances that must communicate with the DBMS [2, 56]. Thus, a short burst of activity can quickly generate thousands of connections. Some programming frameworks and libraries use client-side pooling to maintain long-lived connections (e.g., Phoenix [159], HikariCP [24], pgxpool [41]). This scenario amplifies the scale of connections to the DBMS, as new instances in an autoscaling group can result in thousands of connections for their pools that may sit idle.

For multiple reasons, supporting many persistent clients is challenging for DBMSs. The first is that each new connection incurs some fixed overhead in the DBMS even before that connection issues a query request. To support multiple connections in parallel, the DBMS will assign each connection a *worker* that is either a lightweight thread (e.g., MySQL, Oracle, MSSQL) or a heavyweight process (e.g., PostgreSQL, DB2, TimescaleDB). Each worker requires a fixed amount of memory, CPU, and effort from the OS to manage those resources. For example, PostgreSQL has a memory overhead on the order of megabytes per connection [108, 176]. This cost

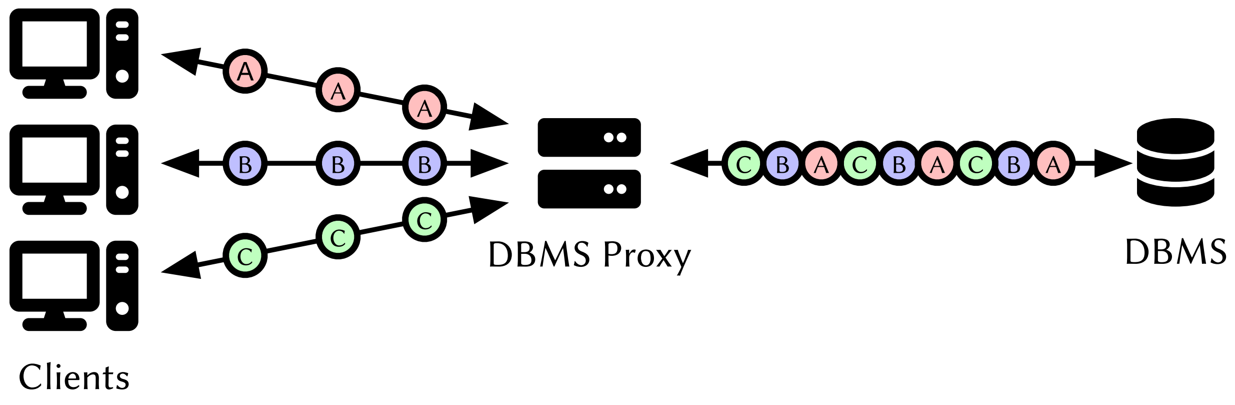


Figure 4.1: Connection Pooling Example – A DBMS proxy reduces the number of connections to back-end DBMSs by multiplexing connections from many clients to a smaller number of pooled connections.

is for a new, unused connection before the DBMS populates auxiliary data structures, such as result caches or prepared statement digests.

Second, since a DBMS provides ACID guarantees for transactions, its concurrency control scheme requires it to perform additional work for each connected client. For example, with optimistic concurrency control, the DBMS performs a validation step when a transaction commits to check whether that transaction conflicts with the read/write sets of other clients connected to the DBMS [211]. PostgreSQL allocates memory for each possible connection, so the conventional wisdom is to set its `max_connections` knob as small as possible to keep concurrency control operations fast; otherwise, it limits the scalability of the DBMS [109].

One approach to handle many connections is to scale the DBMS horizontally by adding more nodes. Then, an L3/L4 proxy (e.g., HAProxy [22], nginx [35]) performs load balancing over TCP sessions between nodes. However, these additional nodes add complexity and cost to a deployment. Combined with client-side pooling, the DBMS could waste resources supporting mostly idle connections. A better approach to address these scaling challenges is to deploy a DBMS proxy that performs *connection pooling* [150]. Instead of connecting to the DBMS, applications connect to the proxy using the same authentication methods. The proxy then manages all connections between front-end clients and one or more back-end DBMSs. With *transaction pooling*, the proxy multiplexes client connections over shared back-end connections, as shown in Figure 4.1. The proxy temporarily allocates a server connection until a transaction commits or rolls back. This approach improves performance when many persistent connections do not continuously submit transactions to the DBMS.

To demonstrate how transaction pooling can improve performance, we run the YCSB workload [88] with 10,000 connections in two configurations. The clients connect directly to the DBMS in the first setup and through Tigger DBMS proxy in the second. We discuss our experimental setup in Section 4.5.

Figure 4.2 shows the latency distributions from the two scenarios. YCSB's p99 and mean latencies are much higher than the median when many clients connect directly to the DBMS. In contrast, using the proxy slightly increases the minimum latency while significantly reducing

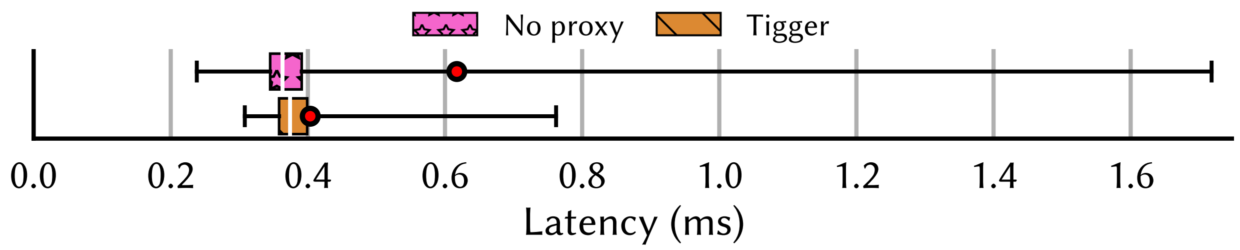


Figure 4.2: Connection Pooling for Many Clients – YCSB transaction latencies showing the effect of Tigger's transaction pooling. The red circle shows sample mean, and the upper whisker shows p99.

the p99 and mean latencies. We explore more workloads in this configuration in Section 4.5.2.

4.1.2 Connection Establishment

In addition to large numbers of persistent connections, short-lived ephemeral clients also pose challenges for DBMSs. Ideally, applications should use client-side connection pooling, but improperly tuned web frameworks result in short client *sessions* (i.e., the time between connect and disconnect) that last only for the duration of a transaction. Popular frameworks, such as Laravel [61] and Django [62], do not use persistent connections by default. The application opens a new connection to the DBMS for each request and discards the connection upon completion. Furthermore, some cloud software designs cannot exploit client-side connection pooling. For example, microservices that rely on stateless serverless functions (e.g., AWS Lambda) create a new connection on each invocation.

Short sessions are problematic in DBMSs because the connection setup and teardown work steal CPU cycles that the DBMS could use for executing queries. This overhead is due to how DBMSs handle parallel connections: the system creates a new worker dedicated to each connection. Creating a new worker incurs overhead for (1) task creation, (2) socket allocation, (3) TCP handshakes, (4) Transport Layer Security (TLS) handshakes, (5) client authentication, and (6) querying DBMS settings and catalogs. These unavoidable steps take milliseconds and waste DBMS resources when performed thousands of times per second. Then when the connection closes, the DBMS performs additional bookkeeping and cleanup.

Once again, DBMS proxies help with this problem. Since they are optimized for connection management, their logic for opening and closing connections is more efficient than DBMS workers that have to balance many tasks (e.g., query planning/execution, logging, garbage collection). Moving connection setup and teardown execution to a different machine frees the DBMS's CPU to focus on these other tasks. The extra network hop introduced by the DBMS proxy offsets costly system calls like `fork()` and the subsequent page faults accompanying a new process.

To highlight these issues, we use PostgreSQL and YCSB in two configurations representative of serverless applications where each transaction opens a new connection. In the first setup they connect directly to the DBMS, and in the second they connect through the Tigger DBMS proxy.

Figure 4.3 shows the latency improvement at 2,000 TPS when serverless clients connect to

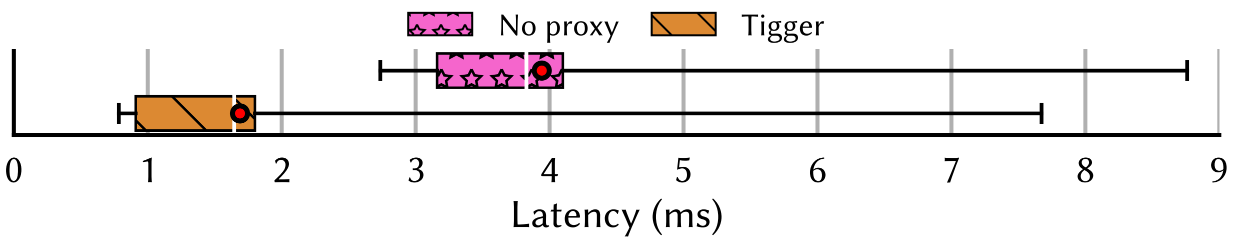


Figure 4.3: Connection Pooling for Serverless Clients – YCSB transaction latencies showing the effect of Tigger's persistent connections. The red circle shows sample mean, and the upper whisker shows p99.

Tigger instead of directly to the DBMS. PostgreSQL uses processes for multitasking, so every transaction performs the `fork()` and `wait()` system calls. As a baseline not shown in the figure, the mean latency for YCSB with persistent servers in this environment is 0.2 ms. Creating and discarding a connection in PostgreSQL adds over 3.5 ms of latency.

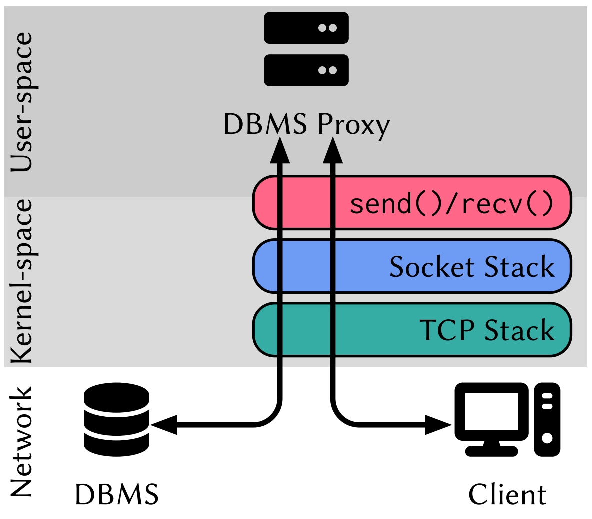
DBMS proxies can provide partial benefits of a persistent connection pool even when it is not an option. Despite introducing extra network hops, Tigger reduces the latency overhead of short-lived connections — dropping from an average of 3.9 ms to 1.7 ms. For short transactions like in YCSB, a DBMS proxy halves the latency in serverless environments. These results demonstrate how connection establishment affects performance.

4.1.3 User-Space Proxy Design

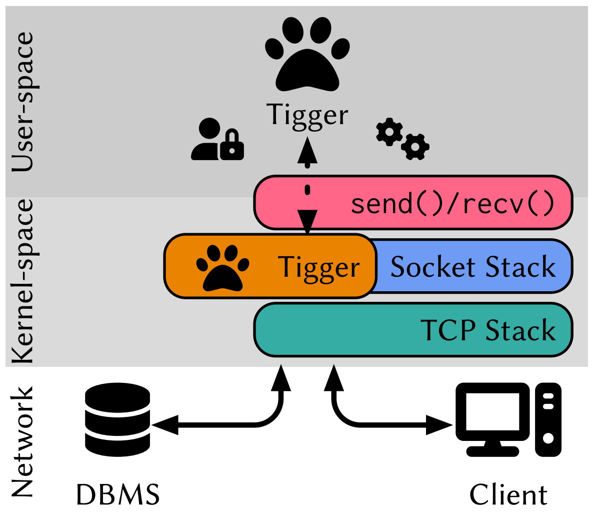
DBMS proxies contain logic specific to a back-end DBMS's network protocol, thus making them L7 network applications. This behavior contrasts with L3/L4 proxies (e.g., HAProxy [22], nginx [35]) that transparently perform load balancing at lower layer protocols. Such L7 logic requires processing network data in user-space (i.e., above the OS networking stack) or applying deep packet inspection (DPI) in lower levels of the network stack.

To the best of our knowledge, all existing DBMS proxies follow the same design in Figure 4.4a: they are event-driven user-space applications that, after the authentication steps, (1) read client messages from a network socket, (2) inspect the stream of bytes, (3) match the client to a back-end server, and (4) send the data on the matched socket. Query results follow a similar logic but with the sender and receiver reversed. Our inspection of open-source proxies supports this belief, though implementation details vary from proxy to proxy. PgBouncer is written in C and uses the libevent library for notifications but is entirely single-threaded. Odyssey [37] is also in C, directly calls `epoll` for event management, but uses a bespoke coroutine library written in assembly and asynchronous I/O to enable parallelism. ProxySQL is in C++, relies on the `poll` system call, and follows a more typical multithreaded design.

These proxy implementations coordinate `send()` and `recv()` system calls with DBMS protocol-specific control logic. This design limits their scalability as network bandwidths increase. Research shows that copying buffers during system calls accounts for ~50% of the kernel's network stack CPU cycles [83]. PostgreSQL developers concluded that a saturated PgBouncer process spends most of its time copying data in and out of buffers between user-space and kernel-space [85]. As we will show in Section 4.5, PgBouncer's single-threaded design limits



(a) User-space proxy



(b) User-bypass proxy

Figure 4.4: User-Space and User-Bypass DBMS Proxies – User-space proxies rely on system calls to redirect queries and results between clients and DBMSs. Tigger employs user-bypass as a fast path in kernel-space, only passing authentication and user settings message to user-space.

its throughput. Although Odyssey offers more parallelism, its CPU demands scale with its capabilities. Ideally, a DBMS proxy should do most of its work without copying socket buffers to user-space via system calls.

4.2 User-Bypass

We next define our user-bypass method in the context of Linux networking and existing kernel-bypass techniques. We also discuss the kernel feature that enables user-bypass and its limitations.

4.2.1 Kernel-Bypass vs. User-Bypass

A complete overview of Linux's networking stack is beyond the scope of this chapter; we will focus on the portion relevant to DBMS proxies. Linux contains multiple layers for processing network traffic. These layers handle transport (e.g., TCP), network (e.g., IP), and link layer (e.g., Ethernet) protocols. The stack exposes a socket interface [52] for applications to copy data between user-space and kernel-space, along with traffic control [54] interfaces to configure queuing disciplines and network filters.

User-space applications that prioritize performance over simplicity can elide these software layers using *kernel-bypass* methods. In this scenario, a user-space application receives bytes directly from the device driver — bypassing the kernel's network stack. Thus, the application manages communication protocols and their associated state machines. The most common kernel-bypass pattern for network applications is to use Intel's Data Plane Development Kit

(DPDK) [16] software library with user-space TCP implementations like mTCP [130]. Other libraries (e.g., F-Stack [21] and ScyllaDB's Seastar [48]) attempt to simplify development by bundling DPDK with bespoke TCP logic. In 2018, Linux added native kernel-bypass support with AF_XDP [127], removing the dependency on out-of-tree kernel modules such as DPDK. We discuss our (frustrating) experiences using kernel-bypass in DBMS proxies in Section 7.1.

Historically, kernel-bypass was the preferred way to implement high-performance Linux networking applications. There are several reasons for this view: (1) the Linux networking stack was perceived as slow and inefficient, (2) applications performed encryption in user-space using a software library (e.g., GnuTLS, LibreSSL, OpenSSL), and (3) a lack of programmability in the networking stack. The interfaces to program the network stack were limited to filtering and routing decisions based on L2/L3 rules (e.g., `iptables`, `nftables`, `tc`). This lack of extensibility in the kernel prevented L7 DPI necessary for a DBMS proxy. Thus, kernel-bypass remained viable for network applications with complex user-space logic.

Kernel-bypass aims to improve performance by transferring buffers between devices and user-space applications instead of the kernel processing them. *User-bypass* is the opposite approach: the developer pushes application logic into the kernel's network stack as low as possible to avoid copying data between user-space and kernel-space, while benefiting from the kernel handling L1-L4 networking. Until recently, if a developer wanted to embed application logic into the kernel, they would have to either (1) load a kernel module or (2) modify and recompile it. Such approaches are difficult and sacrifice system reliability and safety. However, updates to Linux make user-bypass a viable alternative to kernel-bypass. The efficiency of the Linux networking stack has also improved: a single CPU core can process 42 Gbps [83], and dedicated servers can process 670 Gbps of data [63]. Next, kernel TLS (kTLS) allows developers to move encryption into kernel-space and hardware [183]. But the key reason that user-bypass is now possible is the increased programmability via eBPF, which is composable with kTLS [76].

4.2.2 eBPF

To understand user-bypass, we now provide an overview of how the technique embeds DBMS logic in the Linux kernel. The primary technology that enables this functionality is *extended Berkeley Packet Filter* (eBPF). This modern Linux subsystem enables developers to write safe, event-driven programs running in kernel-space [190]. Application developers and cloud vendors have rapidly adopted eBPF due to its safety and features [7, 9, 12]. For example, Meta loads over 40 eBPF programs on every server, with hundreds more loaded on demand [191].

eBPF programs run a limited instruction set in a kernel-embedded virtual machine. Developers typically write eBPF programs in higher-level languages like C or Rust that compile to eBPF bytecode via LLVM. Upon loading the eBPF program, modern kernels compile the bytecode to native machine code. eBPF program capabilities vary based on their type and attachment point, but they attach to predetermined functions in the OS stack for network processing.

Developers load eBPF programs into kernel-space and then associate them with events (e.g., functions, static tracepoints) to trigger their execution. When a running thread hits the attachment point, it starts the execution of the eBPF program in privileged mode. Depending on their behavior and attachment location, developers use eBPF programs for software debugging, profiling, or modifying data flow (e.g., network buffers) through kernel-space. Tigger attaches eBPF programs with DBMS protocol logic in the socket and Traffic Control (TC) layers of the Linux networking stack.

The execution state of eBPF programs is ephemeral, meaning that its decision-making is limited to the data available during a single invocation of the handler. However, with *eBPF maps*, the program can maintain state across events, enabling user-bypass to support more complex application behavior.

These data structures reside in kernel-space and are the primary mechanism for creating stateful user-bypass programs. Tigger relies on three eBPF map types to coordinate execution across multiple eBPF programs: the (1) stack and (2) array map types are a persistent stack and array, respectively, while the (3) sockmap type is a unique map type that attaches an eBPF program to socket activity. Developers associate a sockmap with a single eBPF program and then add or remove socket file descriptors to or from the map. When activity occurs on a registered socket (i.e., updates to an ingress or egress buffer), the kernel executes the associated eBPF program.

Because the CPU is in privileged mode when eBPF programs run, the kernel requires them to pass a verification step before it loads them. The eBPF verifier enforces kernel API compliance, memory access safety, execution bounds, and instruction count. The verifier generates a control flow graph for all possible branches of the eBPF program and checks limits like 512 B stack size and 1 M instructions. Although these restrictions limit the complexity of application logic with user-bypass, DBMS network protocols for the most common message types (i.e., queries, results) are expressible in eBPF. Despite the verifier's guarantees, developers should consider security and safety practices with eBPF deployment due to its close interaction with kernel functions, especially in multi-tenant environments.

4.3 Tigger DBMS Proxy

We present Tigger as a solution that relies on user-bypass to overcome the challenges with existing DBMS proxies described in Section 4.1.3. As shown in Figure 4.4b, Tigger employs user-bypass, resulting in both user-space and kernel-space components to implement the DBMS's network protocol.

In this section, we first show how to apply user-bypass to DBMS proxies with our implementation for Tigger. We then detail Tigger's support for the PostgreSQL network protocol.

4.3.1 User-Bypass Proxy Design

Tigger is a modified version of PgBouncer [39] that employs user-bypass to replace core components with eBPF-enhanced implementations. We chose PgBouncer as the foundation for Tigger because it is the most widely deployed DBMS proxy for PostgreSQL. However, the user-bypass design in this section could be applied to a different proxy (e.g., MaxScale for MySQL) to support other DBMSs. User-bypass results in a hybrid software design with both user-space and kernel-space logic, providing a fast path for the most common proxy tasks. Tigger's user-space component retains PgBouncer's single-threaded design but achieves parallelism through user-bypass because eBPF components run on kernel threads.

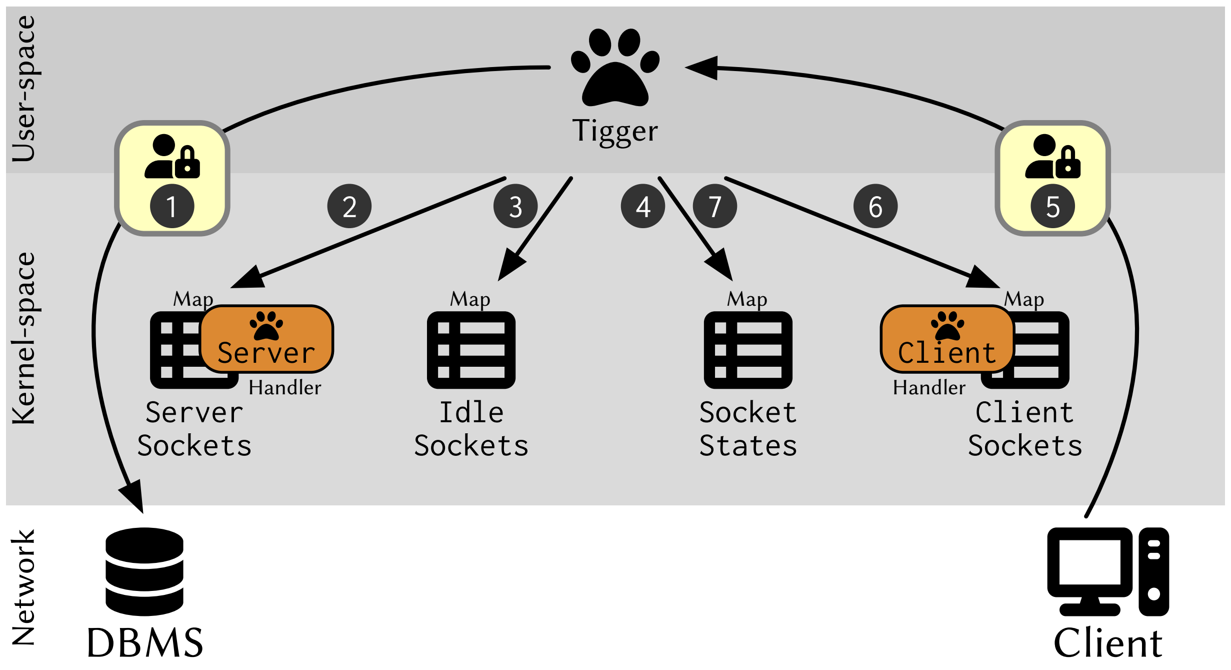


Figure 4.5: Tigger's User-Bypass Architecture – Tigger's hybrid design constrains user-space and kernel-space (i.e., eBPF programs and maps) components. We describe the steps in Section 4.3.1

.

The user-space portion of Tigger is responsible for connection establishment, client authentication, and settings management. This component synchronizes the connection state with Tigger's kernel-space logic by reading and writing to eBPF maps. Tigger retains these parts as the user-space component for several reasons: (1) DBMS authentication methods (e.g., SCRAM-SHA-256, GSSAPI) are too complex to satisfy the eBPF verifier's limitations described in Section 4.2.2, (2) these events are infrequently executed and not part of the hot path of dispatching queries, and (3) it simplifies user-bypass engineering by reusing an existing application. These user-space authentication components follow the same semantics as in PgBouncer, enabling administrators to define how front-end users map to back-end connection pools. These settings ensure clients cannot submit queries to the DBMS with improper credentials.

Tigger maintains two independent connection pools: (1) a user-bypass pool (2) a user-space pool. The first is dedicated to user-bypass links between clients and back-ends. In the typical case, Tigger links a client to a back-end DBMS, redirects the session's queries and responses, and unlinks the two endpoints without executing any user-space code. In the rare event that all user-bypass sockets are in use, Tigger can pass a client's session up to the fallback connection pool managed in user-space at the cost of being slower. Exceptional protocol operations like cancel requests also pass to user-space and route through this pool.

When Tigger starts, it loads its eBPF maps and programs into kernel-space. Tigger's connection pooling relies on two eBPF *handler* programs: one to process buffers for back-end DBMS sockets (`Server`) and another for front-end client sockets (`Client`). Each handler attaches to its sockmap to trigger execution, and the other two maps (`IdleSocketsMap` and

SocketStatesMap) synchronize the kernel-space state with the user-space components. Depending on the proxy's configuration, Tigger may install more eBPF handlers to provide additional features (see Section 4.4.2).

Most of Tigger's handlers operate at the socket layer (i.e., above the TCP stack) attached to sockmap events. The handlers run on kernel worker threads responsible for software interrupts, which is how Linux processes network events in kernel-space. Although eBPF programs can hook into lower levels of the networking stack (e.g., XDP, TC) it is not ideal to push DBMS protocol logic lower than the socket layer: every layer bypassed in kernel must be re-implemented by the proxy. For example, hooking into the network stack below the TCP layer requires the proxy to implement the complex TCP state machine and its associated messages (e.g., ACKs and retries). Furthermore, eBPF programs must be attached at the socket layer to perform DPI on encrypted content using kTLS [76].

At the sockmap layer, Tigger benefits from the OS handling protocols below the L7 layer. The OS arranges ingress bytes in their correct sequence order and reliably sends egress bytes. Socket buffers appear as they would in user-space (i.e., ready for L7 logic), with the OS processing headers related to TCP/IP and Ethernet. Therefore, Tigger's eBPF handlers' logic is similar to user-space DBMS proxy logic. After compilation, Tigger's `Client` handler contains 267 eBPF instructions, but due to its loops and branches the verifier evaluates 217,732 instructions. Although `Client` is lower than the verifier's 1 M instruction limit, more branches or loop iterations exponentially increases the verifier's work.

Figure 4.5 shows Tigger's hybrid design and steps through the user-space and eBPF features that enable user-bypass. After Tigger loads its handlers (i.e., `Client` and `Server`) and maps into kernel-space, ❶ its user-space component opens and authenticates connections to the back-end DBMS for pooling. Next, ❷ Tigger adds the server sockets to `ServerSocketsMap`. This step ensures that the `Server` handler runs whenever a DBMS socket buffer is ready for processing. With the back-end socket ready to accept queries, ❸ Tigger adds it to the stack map of idle sockets. ❹ Tigger resets the state metadata associated with the socket. Upon a new connection request, ❺ the client authenticates with Tigger's user-space component. ❻ Tigger adds the client's socket to `ClientSocketsMap`. The `Client` handler will now execute on buffer activity from a front-end connection. Lastly, ❼ Tigger resets the metadata associated with the client socket stores in `SocketStatesMap`. To apply user-bypass to a different system, developers reproduce this logic in the user-space components of another DBMS proxy.

4.3.2 DBMS Protocol Logic

When a socket buffer arrives, Tigger's `Server` and `Client` handlers perform DPI to apply DBMS protocol logic — extracting the state of client sessions to implement features like connection pooling. For PostgreSQL messages, processing a socket buffer involves inspecting each message header to determine its type, length, and, if necessary, the body. A PostgreSQL message is not guaranteed to fit within a single buffer, so `SocketStatesMap` maintains metadata to help the handlers process messages that span buffers. First, it contains enough space to store a partial header in case the header spans multiple socket buffers. Second, it has an offset into the following buffer to find the next message.

Figure 4.6 shows an example of how Tigger processes multiple PostgreSQL messages in

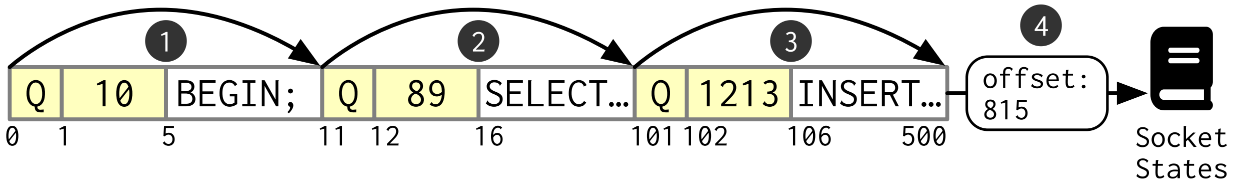


Figure 4.6: Applying DBMS Protocol Logic – Tigger performs DPI on DBMS messages to determine types and lengths. If a message spans multiple buffers, Tigger stores the position to start reading the next buffer in the `SocketStatesMap` eBPF map. We describe the steps in Section 4.3.2.

a socket buffer containing 500 bytes. ❶ The `Client` handler reads the first message header and length and computes the location of the next header. ❷ The `Client` handler repeats the process and arrives at the third PostgreSQL message. ❸ The header indicates that the `INSERT` statement is `1213` bytes long, but the socket only has `398` of its `500` bytes remaining. In this scenario, ❹ `Client` stores the offset (i.e., `815`) to look for the following PostgreSQL message header in `SocketStatesMap`. When the next client socket buffer arrives, the handler starts processing at that offset rather than inspecting every byte.

Both the `Client` and `Server` handlers process the PostgreSQL protocol similarly but apply different control logic. `Client` looks for message headers related to the session (e.g., authentication, disconnect). As described in Section 4.3.1, client messages are typically query requests, and `Client` redirects those to a back-end DBMS. If the buffer contains session messages, `Client` passes it to Tigger's user-space component. Similarly, `Server` looks for control messages that denote transaction status (i.e., active vs. idle), which requires reading the message body and the headers. Based on this information, `Server` uses eBPF maps to coordinate transaction pooling among multiple front-end and back-end connections.

4.4 Tigger Features

We now describe how Tigger implements DBMS-specific logic to support the two most important features of proxies: pooling and replication. Although the message protocol will differ for other DBMSs, the way Tigger achieves user-bypass for these features would be the same. Specifics related to the PostgreSQL protocol could be adapted to other DBMS protocols.

4.4.1 Connection Pooling

As introduced in Section 4.1.1, connection pooling is when a proxy shares a single server connection across one or more client collections. Tigger supports two common forms of pooling: session and transaction pooling. These settings determine how long a front-end client holds a pooled back-end connection. Session pooling allocates a pooled connection for the duration of a client's session, so it is impossible to multiplex connections to reduce the peak number of connections to the DBMS. This mode requires less work from the DBMS proxy than transaction pooling since it does not maintain transaction state. Instead, the proxies only need to

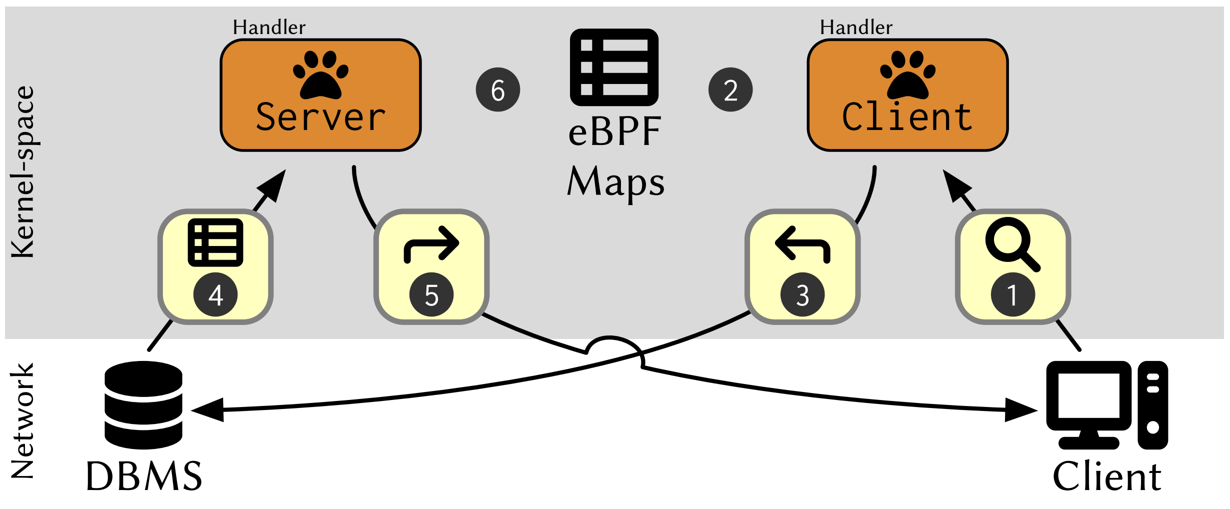


Figure 4.7: Connection Pooling with User-Bypass – Tigger performs pooling with two handlers and the maps shown in Figure 4.5. We describe the steps in Section 4.4.1.

check for messages that terminate the client's session. In contrast, transaction pooling releases connections back to the pool at the end of a transaction, reducing the number of connections to the back-end DBMS as shown in Figure 4.1. Tigger's `Client` handler does not link clients and servers at authentication time, instead waiting until a query arrives. This approach minimizes how long a client holds a connection and makes it available to other client requests as soon as possible.

Figure 4.7 details Tigger's steps when handling a client request over a pooled connection. When an authenticated client submits a query, ❶ Tigger's `Client` handler executes when the buffer arrives at the socket layer of the proxy. ❷ `Client` first checks `SocketStatesMap` to see if this socket is already linked to a back-end DBMS socket. If not, Tigger acquires the first socket available from `IdleSocketsMap`. If there are no available user-bypass sockets, the session passes to user-space to be handled by Tigger's slow path. After matching with a user-bypass socket, `Client` processes the buffer (as described in Section 4.3.2) to determine if it is a session, and if it is `Client` passes it to user-space.

In the typical case the socket buffer contains a query, so `Client` redirects the buffer to the linked user-bypass socket and updates the metadata in `SocketStatesMap`. ❹ The back-end DBMS executes the query and sends the results back to Tigger. The `Server` handler runs on buffer arrival, finding the linked front-end socket for the back-end. ❺ `Server` processes the buffer, stores any intermediary state in `SocketStatesMap`, and redirects the buffer to the linked DBMS socket. ❻ occurs depending on the proxy's pooling mode: at transaction completion for transaction pooling or client disconnect for session pooling. During this step, `Server` unlinks the client from the DBMS and inserts the back-end socket into `IdleSocketsMap`.

4.4.2 Workload Mirroring

DBAs also deploy DBMS proxies to provide different forms of replication. Applications use such replication for load balancing, high availability, or test environments. One important type of

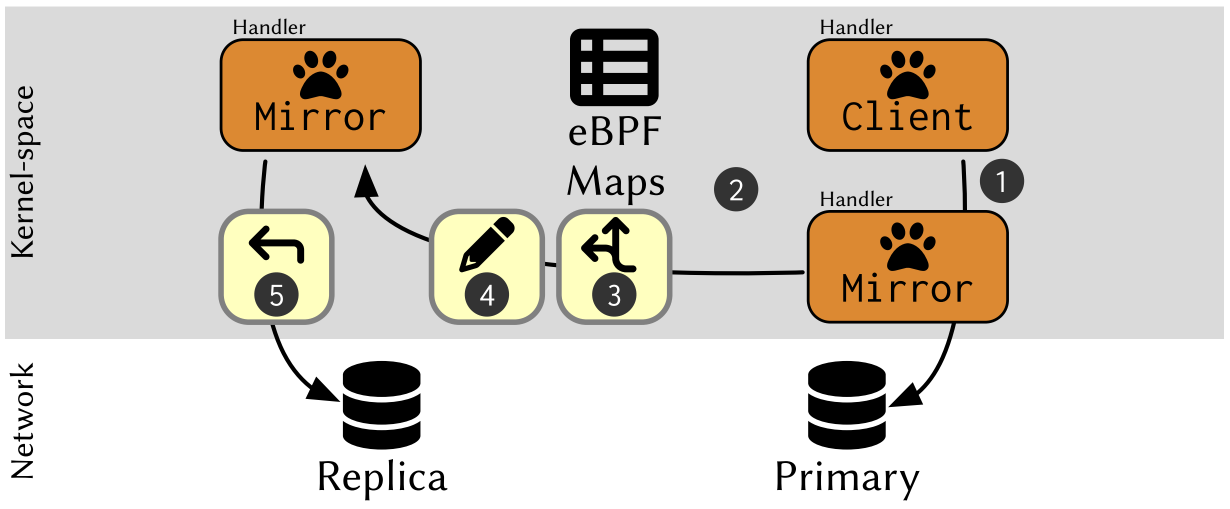


Figure 4.8: Workload Mirroring with User-Bypass – Tigger performs workload mirroring with additional eBPF programs and maps, including one attached at the TC layer. We describe the steps in Section 4.4.2.

replication is *workload mirroring*, where the proxy sends the same queries to multiple DBMSs but treats one as the authoritative primary node [201]. With mirroring, there are no consensus protocols, result set validation, or awareness between DBMS nodes of their arrangement. It is helpful for prewarming replicas before adding them to the pool of active instances, as well as facilitating the testing of DBMS versions using live traffic during upgrades [137].

Tigger supports workload mirroring between multiple back-end DBMSs. Tigger still performs connection pooling with the primary DBMS, as described in Section 4.4.1. Replication requires an additional handler (`Mirror`) to send one inbound message to multiple back-ends. Tigger cannot perform workload mirroring in the `Client` handler due to a limitation in the eBPF verifier: the API to clone socket buffers is unavailable at the socket layer. eBPF programs can only access the clone API at the TC interface.

As shown in Figure 4.8, Tigger's `Mirror` handler consists of multiple eBPF programs. The first program clones the necessary buffers and attaches as a TC classifier for egress traffic — executing after the socket, TCP, IP, and Ethernet stacks. The second eBPF program attaches at the sockmap layer like the `Client` and `Server` handlers and sends cloned buffers to their replicas. These two programs cooperate in mirroring outbound traffic to replica DBMSs.

When workload mirroring is enabled, Tigger's user-space component adds entries in a eBPF map (`MirrorSocketsMap`) that associates primary DBMS sockets to their replicas' sockets. The user-space component also creates a separate pool of connections for replicas but does not place them into the `IdleSocketsMap` to avoid linking them directly to clients.

Step ❶ in Figure 4.8 picks up after ❸ in Figure 4.7 (see Section 4.4.1). At this point, `Client` bypassed user-space and redirected a query from the client to the primary DBMS. Any new messages arriving at `Client` now respond with "not ready" until all backends are ready. As the outbound message leaves the proxy for the primary, ❷ `Mirror` checks the destination port in `MirrorSocketsMap`, and then ❸ clones the buffer to be sent to a replica.

However, the replica's buffer contains Ethernet, IP, and TCP headers for the primary's ses-

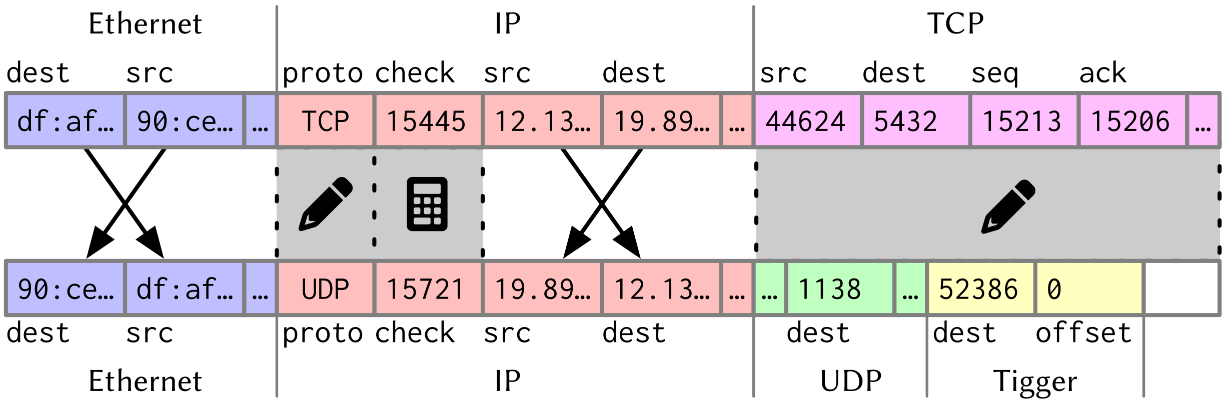


Figure 4.9: Workload Mirroring Header Manipulation – `Mirror` clones buffers at the TC layer, then manipulates protocol headers to send them back to the socket layer via UDP.

sion. At the TC layer, manually changing the cloned buffer's headers is not possible: permuting the headers requires Tigger to maintain its own TCP state machines for replicas. Not only is this too complex to implement in eBPF, it is also impossible to use the OS's network stack for any further communication on that socket due to mismatches with OS's TCP logic.

To overcome this problem, `Mirror` uses a eBPF program at the sockmap layer to redirect the cloned buffer to the replica, similar to `Client` and `Server`. This program allows the OS to manage all communication with replicas, but the cloned buffer still resides at the TC layer. To get it back to the `Mirror` handler at the sockmap layer, ❹ `Mirror`'s TC program manually changes the cloned buffer's message type from egress TCP to ingress UDP, as shown in Figure 4.9.

To change all the necessary headers, `Mirror` first swaps the Ethernet header's addresses, which does not change the header's checksum. Similarly, `Mirror` switches the IP header's addresses and changes the protocol to UDP. The latter operation requires updating the IP header checksum, but the change from TCP to UDP is a compile-time constant, so this is a fast operation. The final header update completely overwrites the old egress TCP header with an ingress UDP header. `Mirror`'s TC component then writes two pieces of metadata in the buffer after the UDP header. Since TCP headers are larger than UDP, Tigger uses these unused bytes to store (1) the replica's socket for this buffer and (2) the offset to application data. TCP headers are variable length, so Tigger must explicitly track where the DBMS message begins.

When the cloned buffer arrives at the socket layer, ❺ `Mirror` extracts the stored replica socket and trims the excess bytes from the original TCP header. Keeping the destination socket in the buffer is an optimization that enables `Mirror`'s sockmap program to redirect buffers without retrieving additional map data. Lastly, `Mirror` redirects the buffer to the replica DBMS, and updates `SocketStatesMap` so that `Client` can synchronize with all the responses.

4.5 Evaluation

We evaluate Tigger's using PostgreSQL (v14.5) DBMS, and configure the `shared_buffers` knob so that working sets fit in memory. We compare Tigger against three other open-source PostgreSQL-compatible proxies:

Table 4.1: AWS EC2 Instance Details – Hardware configurations and pricing for the c6i family of AWS EC2 instances.

Instance	vCPUs	RAM (GB)	Cost ($/hr)
c6i.large	2	4	0.085
c6i.xlarge	4	8	0.17
c6i.2xlarge	8	16	0.34
c6i.4xlarge	16	32	0.68
c6i.8xlarge	32	64	1.36
c6i.12xlarge	48	96	2.04

- **PgBouncer (v1.17):** With decades of development, PgBouncer is the most popular proxy for connection pooling with PostgreSQL. Due to its popularity and maturity, we use PgBouncer as the reference implementation for user-space DBMS proxies.
- **Odyssey (v1.3):** Yandex developed Odyssey as a modern replacement for PgBouncer. It uses multiple workers and coroutines to support parallelism across connections. We use Odyssey as an example of a high-performance user-space DBMS proxy.
- **Pgpool-II (v4.3.3):** This proxy predates PostgreSQL's native replication features, and was commonly deployed to provide high availability for PostgreSQL clusters. Pgpool-II does not support transaction pooling, so we omit it from most of the evaluation. Instead, we only compare Pgpool-II's workload mirroring against Tigger's since PgBouncer and Odyssey do not support that feature.

We do not compare against RDS Proxy because it is a fully managed service. This design makes it impossible to control RDS Proxy's instance size or investigate its performance characteristics.

Unless otherwise specified, all experiments run as follows. We evaluate Tigger using AWS EC2 c6i instances running Ubuntu Linux 22.04 LTS in the same availability zone [17].

Table 4.1 summarizes the resources and pricing for these instances. We use separate instances for each system component: (1) 12xlarge for PostgreSQL, (2) 12xlarge for application servers, and (3) and xlarge for DBMS proxies. We use a smaller instance for the proxies to reflect how users provision proxies in real-world deployments. To better compare user-bypass, we reduce the proxy server's number of receive queues to one to eliminate kernel parallelism for network processing [3].

Each experiment runs at least five times. Latency evaluations use a fixed submission rate of 2000 transactions per second (TPS). This setup ensures all systems perform the same work to compare latency and CPU efficiency. Throughput evaluations run with an unlimited submission rate.

In all box plots, the lower whisker shows the minimum data point, and the upper whisker shows the 99th percentile (p99) data point. We plot p99 instead of the maximum for two reasons: (1) p99 is a standard metric when evaluating the latency of software systems, and (2) write-heavy workloads generate latency outliers due to DBMS resource conflicts (e.g., locks, `fsync()`) that do not reflect the performance of DBMS proxies.

4.5.1 Workloads

In these experiments, we only consider OLTP workloads; OLAP workloads (e.g., TPC-H) are bottlenecked by query execution at the DBMS (e.g., scans, aggregations), which makes them unsuitable for demonstrating the benefits of Tigger's user-bypass design. In a serverless OLAP scenario, connection establishment overhead will not be the bottleneck compared to query execution. Customers deploy DBMS proxies for OLAP for reasons other than performance (e.g., security) outside this evaluation's scope.

All queries execute over JDBC using the BenchBase framework [5, 96]. We turn off automatically prepared statements in the JDBC driver to avoid naming contamination when the back-end connections are shared across client connections. This problem occurs when drivers prepare different statements under the same name, so DBAs turn off this feature when deploying a DBMS proxy.

- **No-op:** The workload contains a single transaction that executes an empty query string (i.e., "; "). The DBMS returns an empty query result. This workload is ideal for measuring DBMS proxy overhead because it minimizes the work that the client and DBMS execute.
- **SmallBank:** This workload models a banking application where transactions perform short read and update operations [65, 82]. The database contains three tables which we scale to ~3.4 GB.
- **TATP:** This benchmark simulates a cellphone caller location system [209]. It has nine transaction types, and we scale the database to ~1.9 GB stored across four tables.
- **TPC-C:** This order-processing application contains nine tables and five transaction types [196]. For our experiments, we use a 20-warehouse database (~2.1 GB).
- **Twitter:** This workload models the troubled social media website where users post messages and follow others. We scale the database to ~4.4 GB.
- **YCSB:** The Yahoo! Cloud Serving Benchmark models cloud service workloads [88]. We run read-only transactions to reduce bottlenecks from EBS writes. The database contains a single table, 1 KB tuples, and a total database size of ~2.3 GB.

4.5.2 Connection Pooling for Many Clients

We begin our evaluation by exploring the scenario from Section 4.1.1 in more detail. The workload represents a typical cloud-native application. We use 25 application servers, each with 400 clients, for 10,000 database connections. The DBMS utilization is low, with each of the 25 application servers submitting 80 TPS distributed across their connections, for a total of 2,000 TPS arriving at the back-end DBMS.

Table 4.2 shows summary statistics of transaction latencies for the six workloads. To more easily quantify overheads and benefits, Figure 4.10 shows the percent change for each DBMS proxy compared to the scenario with no proxy. Due to its user-bypass design, Tigger offers the lowest latencies of any DBMS proxy in every workload. Compared to running without a proxy, Tigger lowers the mean and p99 latencies in every workload except No-op and TPC-C.

No-op does not benefit from using a DBMS proxy. Even at a high connection count, PostgreSQL performs minimal work for No-op, so reducing the number of connections with a DBMS proxy provides no benefit. However, as a microbenchmark for proxies' protocol efficiency, we

Table 4.2: Connection Pooling for Many Clients – Transaction latencies (ms) of DBMS proxies compared to connecting directly to PostgreSQL.

	No-op			SmallBank			TATP			TPC-C			Twitter			YCSB		
	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99
No proxy	0.18	0.15	0.32	2.10	1.84	4.14	0.90	0.48	2.60	13.92	5.74	224.84	0.72	0.47	2.07	0.62	0.36	1.72
PgBouncer	0.34	0.32	0.52	2.16	2.16	3.58	0.84	0.63	2.16	19.55	7.44	339.95	0.66	0.62	1.48	0.50	0.47	0.90
Odyssey	0.35	0.28	0.48	2.19	2.12	3.38	0.88	0.59	2.13	52.24	7.76	1259.11	0.59	0.48	1.29	0.60	0.46	1.08
Tigger	0.24	0.22	0.39	1.96	1.99	3.20	0.71	0.50	1.96	16.08	7.15	258.00	0.52	0.48	1.29	0.40	0.37	0.76

Table 4.3: Connection Pooling for Many TPC-C Clients – Transaction latencies (ms) of DBMS proxies compared to connecting directly to PostgreSQL.

	Delivery			NewOrder			OrderStatus			Payment			StockLevel		
	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99
No proxy	8.68	8.02	26.42	9.57	7.55	42.99	1.85	1.22	7.31	21.05	3.25	562.54	3.60	3.08	6.36
PgBouncer	14.04	10.84	143.69	14.15	9.82	157.60	4.71	1.42	128.15	28.41	3.69	756.52	5.42	2.22	128.59
Odyssey	36.47	11.57	824.07	39.52	10.28	887.05	26.97	1.44	824.11	71.46	3.79	1578.33	29.43	2.25	881.82
Tigger	10.42	10.15	24.04	10.85	9.36	31.28	1.46	1.26	3.77	24.77	3.54	691.03	2.27	2.06	4.53

see that Tigger offers up to 31% lower latency than the other proxies. The size of the ingress query and egress results are essentially zero, so we are seeing the benefits of user-bypass eliding system calls and user-space thread scheduling.

The diversity of TPC-C's transactions makes it difficult to summarize its performance, as in Table 4.2. For example, Delivery includes more queries and thus round-trips to the DBMS. OrderStatus and StockLevel are short and read-only. NewOrder and Payment experience more contention, especially when the number of clients exceeds the number of warehouses. These characteristics and non-uniform transaction profile submission rates result in a multimodal latency distribution. For that reason, we provide per-transaction summary statistics.

Table 4.3 shows that while Tigger reduces the p99 latency in four out of five TPC-C transactions, Payment's higher submission rate and latency hide this in Table 4.2. Tigger's user-bypass design offers the best performance of the three proxies again, though we cannot explain Odyssey's poor performance. TPC-C's performance subtleties demonstrate the trade-offs in deploying a DBMS proxy. For example, deploying Tigger may be a good decision if reduced tail latencies are worth the slight increase in average latency.

4.5.3 Connection Pooling for Serverless Clients

We next revisit the YCSB experiment from Section 4.1.2 that evaluates the latency impact of short-lived connections — this time with more proxies than Tigger. We aim to measure the proxies' connection creation overhead and assess the benefits of Tigger's user-bypass design. Because BenchBase opens a connection at the start of a transaction and closes the connection at the end, we configure the proxies to use session pooling for this experiment. Thus the results

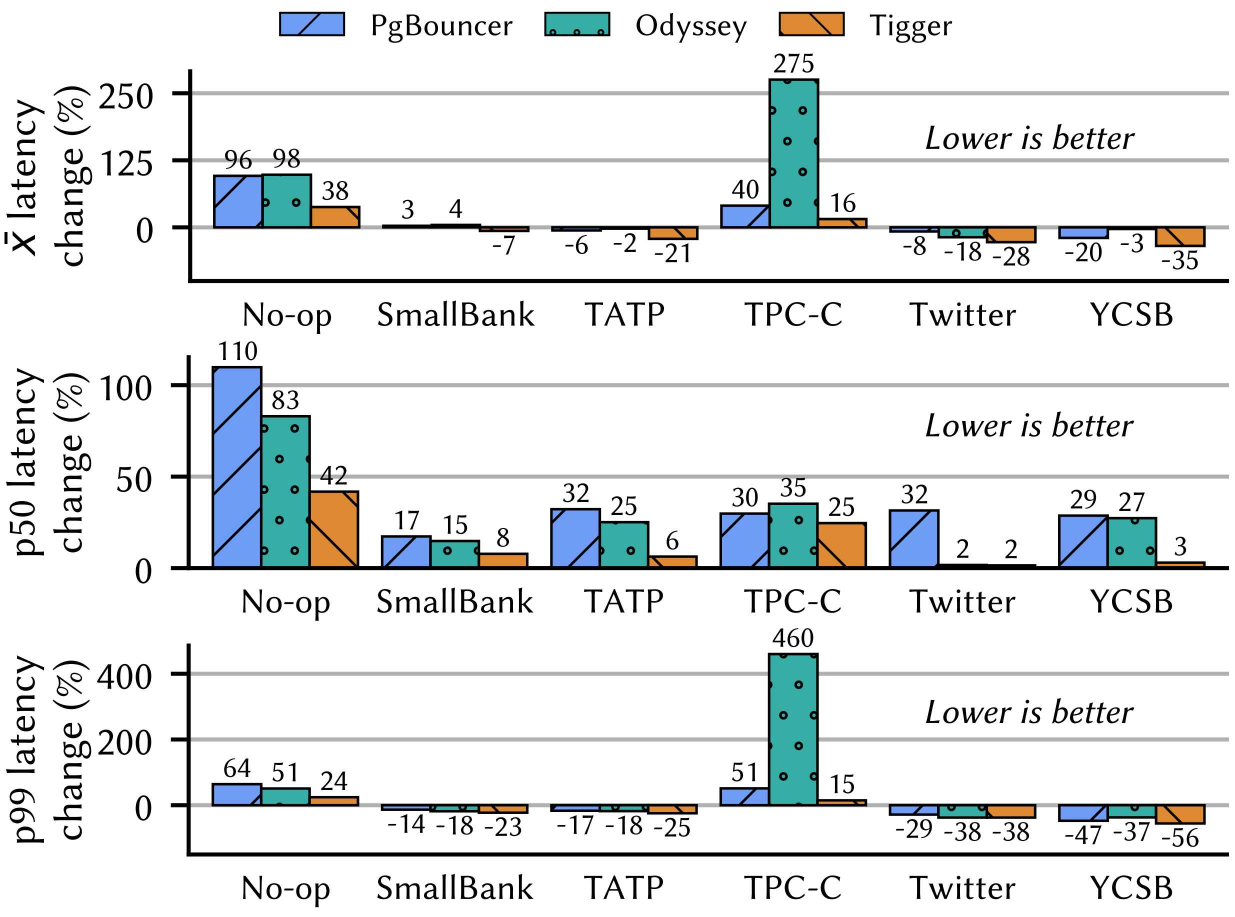


Figure 4.10: Connection Pooling for Many Clients – Percent change of Table 4.2 comparing different DBMS proxies against connecting directly to PostgreSQL.

from Figures 4.3 and 4.11 are not directly comparable to other transaction pooling experiments. The effect is the same (i.e., connections are released to the pool at the end of the transaction), but this configuration requires the proxies to perform less work to track transaction status.

Figure 4.11 shows the same data as Figure 4.3 but with latency measurements for Odyssey and PgBouncer. As discussed in Section 4.1.2, DBMS proxies provide performance improvements in serverless environments by maintaining a connection pool. Figure 4.11 shows how allocating a connection from a persistent pool is about 2 ms faster than going to the DBMS for a new connection.

Tigger outperforms the other two proxies, offering the lowest minimum, mean, and median latencies. Tail latency is slightly higher for Tigger because this scenario is ill-suited to Tigger's user-bypass design. As described in Section 4.3.1, Tigger passes client authentication to its user-space components, which ultimately do more work than PgBouncer's because Tigger has to maintain metadata in eBPF maps. Tigger handling subsequent queries with user-bypass makes up for the authentication overhead to provide the best performance on average for serverless clients.

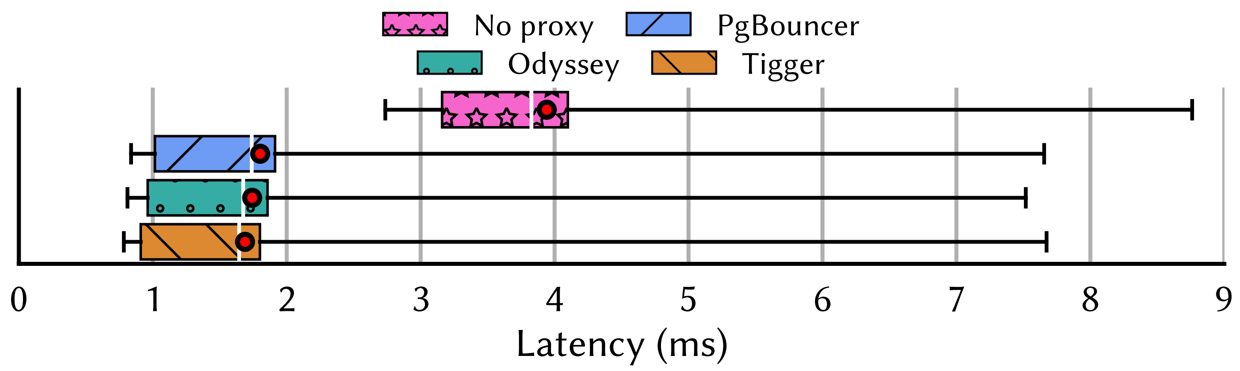


Figure 4.11: Connection Pooling for Serverless Clients – YCSB transaction latencies showing the effect of proxies' persistent connections. The red circle shows sample mean, and the upper whisker shows p99.

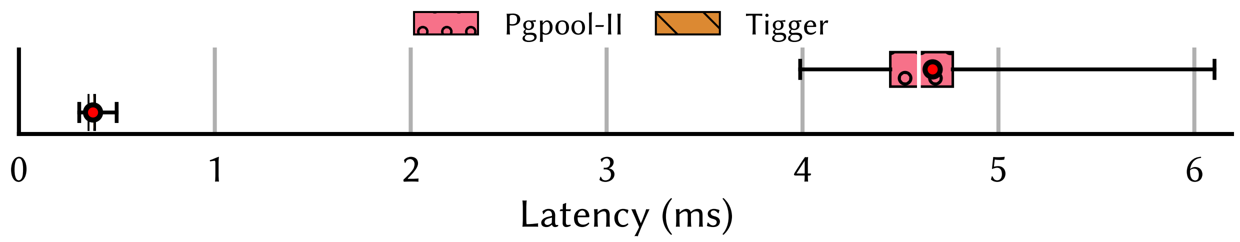


Figure 4.12: Workload Mirroring – Transaction latencies through DBMS proxies configured for workload mirroring. The red circle shows sample mean, and the upper whisker shows p99.

4.5.4 Workload Mirroring

We now evaluate Tigger's workload mirroring as described in Section 4.4.2. For this experiment, we deploy PostgreSQL with two nodes: (1) primary and (2) replica. We configure Pgpool-II [40] to use its "native replication" clustering mode that routes queries to connection pools for both the primary and the replica DBMSs. The proxy returns results to the client from the primary, and there is no validation that both servers produced the same results. The proxy only ensures that both servers are ready for the following query before allowing the client to proceed. This scenario mimics simultaneously sending an actual workload to a production and staging DBMS. We configure Tigger's mirroring logic to behave like Pgpool-II. Pgpool-II does not support transaction pooling, so we use session pooling and lower the number of client connections to 20.

Figure 4.12 shows the mean latencies of YCSB in the mirroring scenario described above. On average, Tigger produces transaction latencies that are 92% lower than those with Pgpool-II. Mirroring is expensive to implement in user-space because for each replica, a user-space proxy must copy the queries and results from the kernel, which is the most costly part of the Linux networking stack. In contrast, Tigger's user-bypass design benefits from eliding system calls and zero-copy socket buffer duplication in the Linux kernel to enable mirroring. Comparing Tigger's performance from another YCSB experiment in Table 4.2 without mirroring and much more connections, we see that Tigger maintains <1 ms transaction latency in both scenarios.

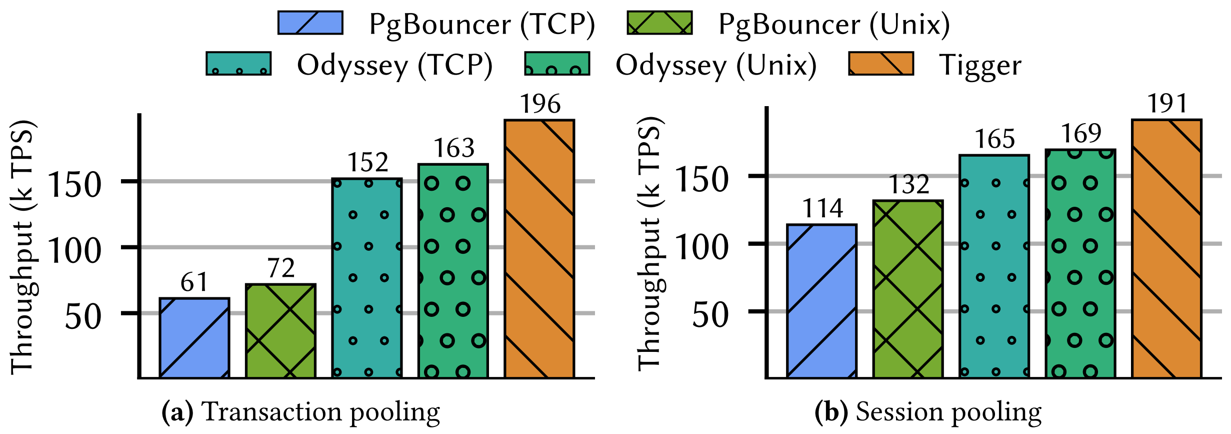


Figure 4.13: Protocol Efficiency – No-op throughput to compare transaction against session pooling, and TCP against Unix sockets.

4.5.5 Protocol Efficiency

We next evaluate the proxies' efficiency at applying DBMS protocol logic and explore the overheads associated with TCP and Unix sockets. We configure the proxies for both session and transaction pooling, as detailed in Section 4.4.1. We reduce the number of terminals to 20 because we are using session pooling and do not want the number of back-end DBMS connections to get too large. This experiment evaluates how efficiently proxies apply knowledge of the DBMS's network protocol to find transaction boundaries in messages and maintain state about connection status.

To avoid unnecessary overhead from the DBMS and exercise proxies' logic as much as possible, we run the No-op workload. We also run the proxies on the same server as the DBMS to reduce network hops. This configuration also allows us to evaluate Odyssey and PgBouncer with TCP and Unix socket connections to the DBMS. Unix sockets are more efficient than TCP networking, so we enable this optimization to quantify their benefit.

The results in Figure 4.13a show the throughput of the proxies in transaction pooling mode. As a baseline, No-op throughput when BenchBase connects to PostgreSQL without a proxy is 214k TPS. PgBouncer performs the worst, with 71% and 67% reductions in No-op throughput over TCP and Unix sockets, respectively. Odyssey's throughput drops 29% and 24% over TCP and Unix sockets. Unix sockets perform better for both proxies because it is a more efficient form of interprocess communication than going through the entire TCP stack. In comparing these results to Figure 4.13b, we see that Tigger experiences no throughput degradation when performing extra logic for transaction pooling. The other proxies do not perform as poorly with the simpler task of session pooling, but Tigger offers consistently better performance and pushes the bottleneck to communication overhead elsewhere.

These results also show the benefit of Tigger's user-bypass, as it outperforms the other proxies, reducing the throughput by only 8% compared to the baseline performance. No-op's messages are small (i.e., fewer than 64 bytes), so the primary benefit of user-bypass is reducing system call overhead. Unix sockets between the DBMS and proxy yield little benefit over TCP sockets.

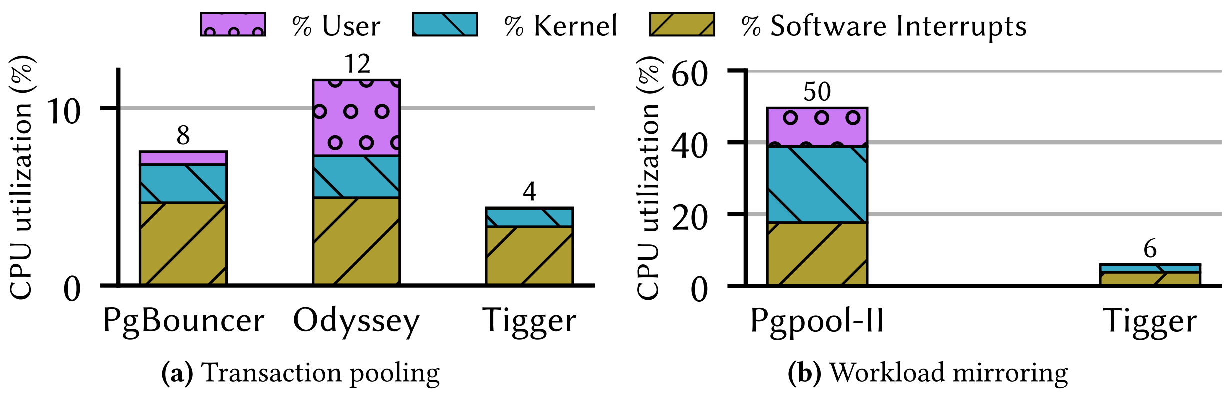


Figure 4.14: CPU Efficiency – Breakdown of CPU utilization for YCSB in Table 4.2 and Figure 4.12. % Software Interrupts is distinct from % Kernel to emphasize Tigger's user-bypass execution.

4.5.6 CPU Efficiency

We next evaluate the CPU efficiency of Tigger's user-bypass approach compared to other DBMS proxies using YCSB. We measure the proxy server's CPU utilization using `sysstat` [53], which attributes time spent in user-space, kernel-space, and software interrupts. We are interested in the latter because Tigger's eBPF programs attached at the sockmap layer run on kernel threads queued as software interrupts (see Section 4.3.1).

Figure 4.14a shows the CPU utilization for the YCSB experiment in Table 4.2. PgBouncer again serves as the baseline. Although Odyssey provides lower latencies than PgBouncer, it increases the CPU overhead by 54%. Odyssey's complex user-space logic for coroutines incurs high overhead to perform the same amount of work as PgBouncer. Lastly, Tigger requires the fewest CPU cycles while providing the best performance of the three proxies. As expected, Tigger's user-bypass design shows almost no CPU time spent in user-space, and reduces the time spent in the kernel by not repeatedly copying socket buffers in system calls.

Figure 4.14b shows the CPU utilization for the YCSB experiment in Figure 4.12. Tigger uses 88% less CPU than Pgpool-II to perform workload mirroring. Pgpool-II must copy buffers and duplicate system calls to perform mirroring, which incurs significant overhead. Tigger benefits from zero-copy socket buffer cloning and redirects these buffers to mirrors without system calls.

4.5.7 Large Data Transfer

Although large data transfers through DBMS proxies are not as common as transactional workloads, we evaluate the overhead of performing such a task. In this scenario, we run the `pg_dump` utility that creates a backup of a PostgreSQL database over the network. `pg_dump` connects to the DBMS like any client, communicates over the PostgreSQL protocol, and extracts database contents data using the `COPY` SQL command. We run `pg_dump` on a dedicated application server, create a PostgreSQL database with 20 tables, and insert 50m single-column `INTEGER` tuples into each table. The overall database size is 34 GB.

Figure 4.15 shows the elapsed time for `pg_dump` to complete a backup in different proxy

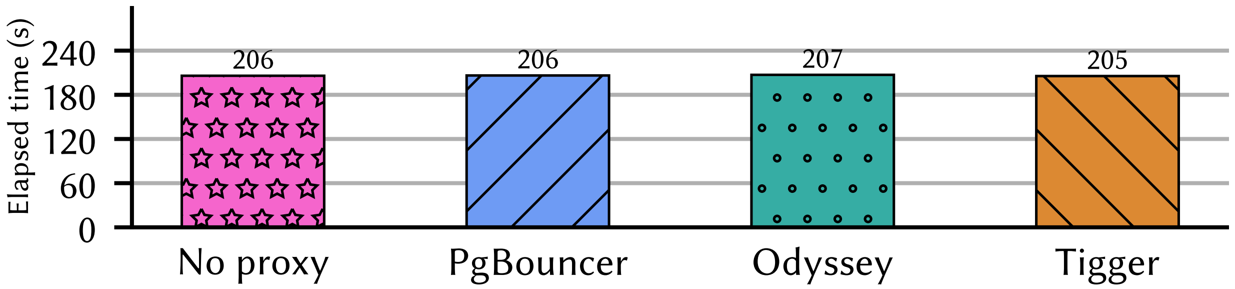


Figure 4.15: Large Data Transfer – Elapsed time for pg_dump to back up a 34 GB database through different DBMS proxies.

scenarios. All three proxies complete the copy without overhead (i.e., pg_dump's efficiency is the bottleneck). This experiment demonstrates a workload where the bottleneck exists at the client or the DBMS, rather than the proxy. In this scenario, a DBMS proxy adds no measurable overhead.

4.5.8 Proxy Instance Size

To complete our evaluation, we now measure the maximum throughput of the proxies with varying EC2 instance sizes. While customers typically deploy proxies on weak servers, we aim to investigate their performance benefits when allocated more resources. For Odyssey and Tigger, this offers opportunities to express parallelism, while PgBouncer only supports a single worker. We run No-op, TPC-C, and YCSB with unlimited submission rates.

Figure 4.16 shows the throughput of the different DBMS proxies on varying instance sizes. For every instance type, Tigger yields the highest throughput. All three proxies scale roughly linearly as the resources of each instance increase. For Odyssey, this makes sense because it uses multiple user-space worker threads, so Odyssey expresses greater parallelism. PgBouncer performs better as instance types change despite being single-threaded. We attribute this to the noisy neighbor problem: the more resources requested from a multi-tenant system, the less competition for unprovisioned hardware (i.e., CPU caches).

With No-op and the smallest instance type (i.e., large, which the other experiments use for the proxy), Tigger more than doubles the throughput of PgBouncer and Odyssey. In this scenario, all three proxies are bottlenecked by CPU utilization. Tigger's higher performance demonstrates the benefit of user-bypass, which does not waste CPU cycles shuffling network buffers between user-space and kernel-space. In all three workloads, Odyssey requires an 8xlarge instance at 8× the cost to match Tigger's baseline performance on a large instance. This experiment demonstrates the benefit of user-bypass to efficiently use CPU resources in resource-constrained scenarios and the impact that user-bypass has on cloud cost.

4.6 Conclusion

DBMS proxies manage connections to address scalability and connection life cycle issues introduced by modern cloud applications. These programs apply DBMS-specific protocol logic to

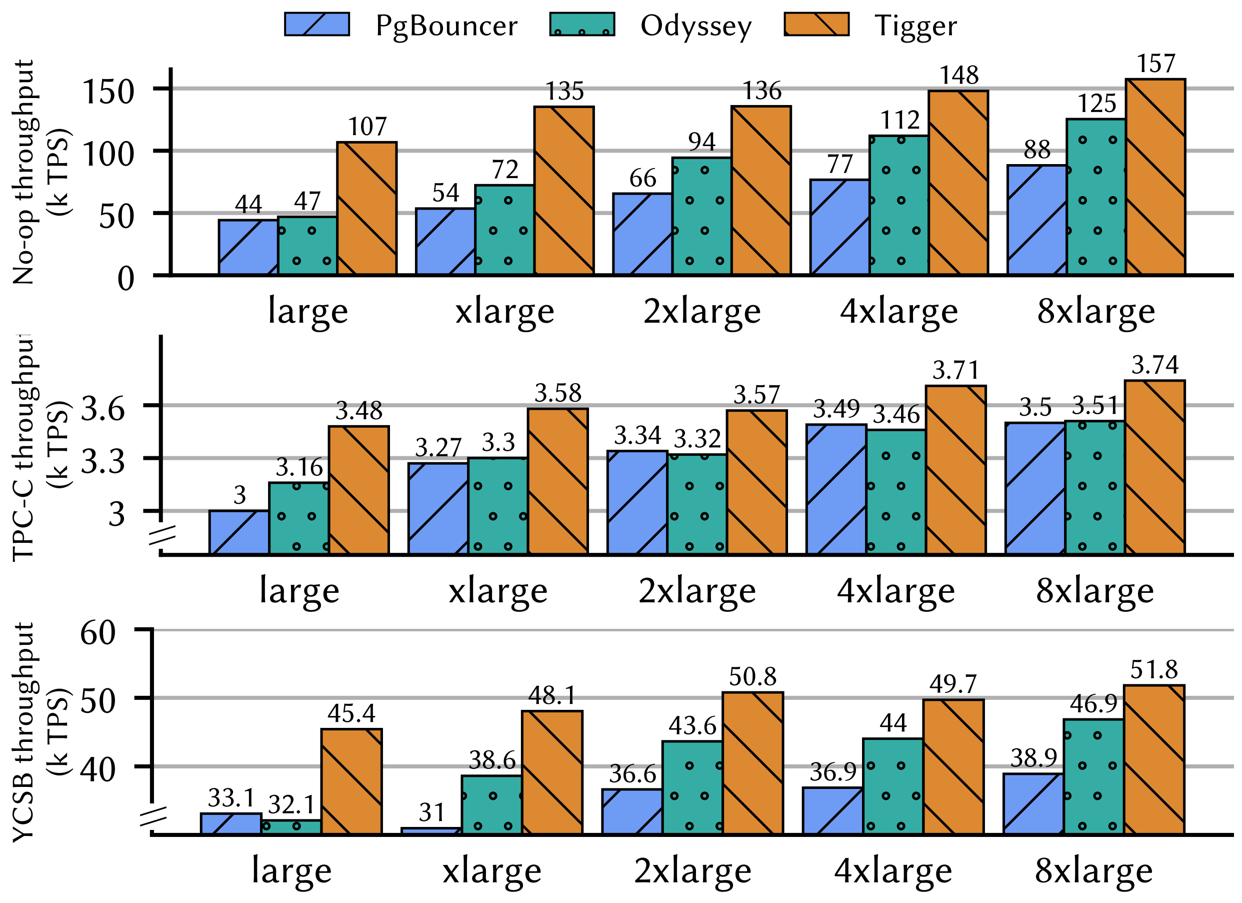


Figure 4.16: Proxy Instance Size – Transaction throughput while varying proxy server instance size.

multiplex client connections. But these proxies incur inefficiencies because they are user-space applications that rely on system calls to copy buffers to and from kernel-space. We introduce a user-bypass technique to overcome these shortcomings by pushing application logic into the Linux kernel using eBPF. We show how to apply user-bypass with our PostgreSQL-compatible proxy Tigger. When compared against other DBMS proxies, Tigger offers the best performance and lowest operating cost. Our evaluation shows the value of user-bypass to reduce data movement between kernel-space and user-space and avoid costly system calls.

Chapter 5

User-Bypass DBMS Architecture

As more developers adopt eBPF and apply user-bypass techniques, the need for more robust data management solutions grows. eBPF programs store their data in eBPF maps, which do not offer ACID properties. For example, eBPF offers safe concurrent accesses for multithreaded applications based on Linux's read-copy-update (RCU) semantics, but only for a single access (i.e., read, update). This limitation forces developers to implement their own synchronization mechanisms to avoid race conditions when accessing eBPF maps multiple times.

Simmilarly, eBPF programs do not support inter-process communication (IPC) and must coordinate concurrent execution through eBPF maps. Building this bespoke logic is onerous and error-prone, making it difficult for developers to reason about the serializability of execution. Lastly, eBPF maps reside in kernel-space memory, providing no interface to make their contents durable on disk.

As a workaround, eBPF developers may choose to store their application state in a user-space DBMS . However, eBPF programs cannot communicate synchronously with user-space applications. This restriction makes it impossible for event-driven, time-limited eBPF programs to retrieve data within their execution budget.

To solve the abovementioned problems, we present **BPF-DB**, an in-memory, kernel-embedded DBMS that offers ACID-compliant transactions for eBPF applications. We implement BPF-DB in eBPF rather than modifying the OS kernel. With this approach, developers can develop user-bypass applications using BPF-DB without sacrificing their system's security or stability. We discuss our deviations from traditional DBMS design to suit eBPF's restrictive execution environment.

We build two sample applications using BPF-DB: (1) a transactional key-value store (KVS), and (2) a voter application that exercises serializable multi-statement transactions. We compare BPF-DB's KVS against Redis-compatible in-memory DBMSs, and its voter application against VoltDB. The former scenario matches state-of-the-art designs, offering over 2 M requests per second for networked clients. Compared to VoltDB, BPF-DB achieves 43% higher throughput.

This chapter is organized as follows. We first discuss embedded DBMSs and expand on the limited data management options in eBPF's runtime in Section 5.1. Then, we introduce BPF-DB as a solution to those limitations in Section 5.2. Next, in Section 5.3, we detail the implementation challenges and design decisions of BPF-DB in the context of traditional DBMS solutions. We demonstrate two applications built on BPF-DB in Section 5.4 and then compare them against state-of-the-art implementations in Section 5.5. We provide concluding remarks in Section 5.6.

5.1 Background

We begin by discussing embedded DBMSs in contrast to standalone DBMSs. Then, we discuss state of eBPF storage and concurrency semantics.

5.1.1 Embedded DBMSs

DBMS developers typically distribute standalone DBMSs that run in dedicated user-space processes. Client applications send queries and retrieve results from the DBMS via inter-process communication (IPC), for example, over sockets with a DBMS-specific protocol. In contrast, *embedded DBMSs* compile the DBMS into the client application, either by including the DBMS source code or linking the DBMS as a library. Clients interact with an embedded DBMS through an application programming interface (API) and access database contents in the context of their own source code, thread execution, and process address space. Developers may use an embedded DBMS if their application is the only database client or the deployment environment does not offer robust multi-tasking or IPC (e.g., embedded systems).

Embedded DBMSs vary in their capabilities and complexity. SQL systems like SQLite [110] and DuckDB [175] offer ACID transactions and robust analytics performance, respectively. Other embedded DBMSs like Berkeley DB [170] and RocksDB [98] present a key-value interface and enable client applications to run transactions over database contents. Embedded DBMSs simplify the non-trivial work of query processing and data integrity so application developers can focus on the core functionalities of their products.

5.1.2 eBPF Storage and Concurrency

As introduced in Section 2.2, eBPF maps store data in kernel-resident memory between program invocations. These maps use a key-value interface and are safe for concurrent access due to the Linux kernel's read-copy-update (RCU) interface [160]. However, each access to a map is a single atomic operation, and eBPF execution does not provide any interface to atomically modify multiple eBPF map entries at once (e.g., transactions or grouped writes). These abstractions are sufficient if eBPF programs attach to points without concurrent execution (e.g., single-threaded application). However, eBPF programs can attach to multithreaded applications or launch from kernel tasks that run on CPU interrupts. In this parallel execution scenario, developers must consider race conditions when interleaving eBPF map operations.

Although its eBPF maps rely on RCU internally, the Linux kernel provides limited access to its RCU mechanism. With RCU, developers can mark explicit synchronization points in their programs to implement features similar to epoch-based garbage collection. A recent kernel patch added an RCU-based read latch for reading kernel data protected by RCU semantics [11]. However, its applications are limited, and the feature is only available on new Linux kernels.

Without RCU primitives, ordering and synchronization from eBPF programs are challenging to implement without an explicit coordination mechanism. eBPF maps do not expose any way to roll back operations, so guaranteeing execution with “exactly once” semantics is non-obvious. As a reduced instruction set, eBPF supports a subset of atomic instructions (e.g., ADD, AND, OR, XOR, XCHG, CMPXCHG) [8]. The kernel extends atomic primitives by providing a spin latch eBPF helper, but the verifier restricts the scenarios when developers can use it. For example, the verifier enforces that the eBPF program does not call functions while holding a latch, and programs cannot hold more than one latch at a time to prevent deadlocks. With these restrictions, developers cannot use a spin latch to provide mutual exclusion to access multiple eBPF map entries simultaneously because these operations require function calls. In practice, eBPF maps provide no ACID guarantees, and it is non-trivial and laborious to reimplement this behavior for every application in eBPF.

5.2 BPF-DB Embedded DBMS

We now introduce a solution to the challenges outlined in Section 5.1. Specifically, we ease eBPF’s concurrency and data management challenges with an embedded DBMS for eBPF programs. **BPF-DB** is an in-memory DBMS that runs in kernel-space to provide serializable ACID transactions and a key-value interface for eBPF applications. Through BPF-DB’s API, developers define eBPF program sequences akin to stored procedures. These procedures interleave custom eBPF code with BPF-DB’s operators using *continuation-passing style* (CPS) [194] logic. With this form of control flow, developers define continuations when calling into procedures. When the called procedure completes, rather than returning to its caller as in direct style, it calls the appropriate continuation. Developers can construct procedures that tail call through multiple eBPF programs and BPF-DB operators through continuations. We describe this design in further detail in Section 5.2.2

Figure 5.1 shows the design of BPF-DB: eBPF programs (shown in blue) running in kernel-space access and modify database contents through ACID-compliant operators. BPF-DB stores database contents in eBPF maps that client applications only access through BPF-DB’s operators. However, they can define and access other eBPF maps to store application data that does not require ACID semantics.

BPF-DB’s primary eBPF map is its Index, which stores keys and their associated metadata (e.g., locks, version information). BPF-DB accesses associated values in eBPF maps for different value sizes from this key metadata. BPF-DB uses these value maps to address the lack of dynamic memory allocation in eBPF programs. For example, values up to 8 B are stored in one eBPF map, and values up to 16 B in another. Because BPF-DB stores its data in kernel memory independent from its eBPF programs, developers can load and unload client applications and BPF-DB operators without losing database contents. This design is similar to how Scuba achieves system restarts without reloading data by storing its database contents in shared system memory [114]. We discuss BPF-DB’s storage architecture in greater detail in Section 5.3.1.

To deploy BPF-DB, eBPF developers implement their application logic in standalone programs. Each time their application accesses BPF-DB, developers define continuations that instruct BPF-DB what code to execute after completing its operator. Developers then compile

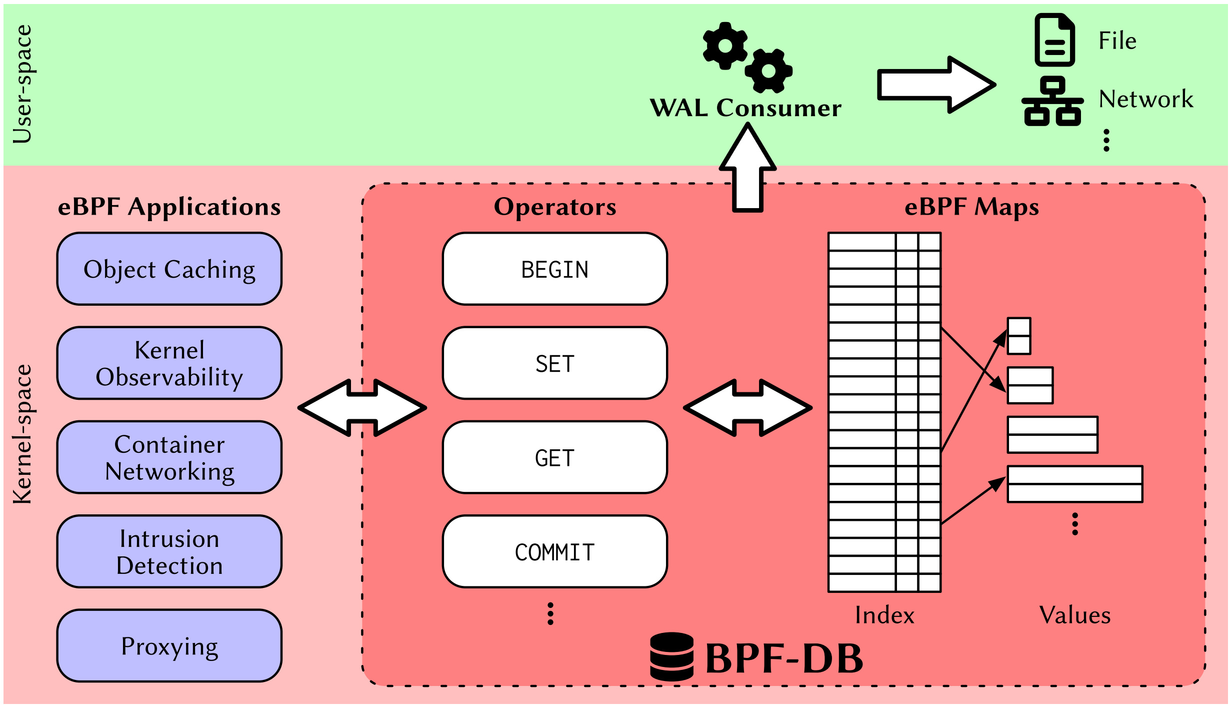


Figure 5.1: BPF-DB Embedded DBMS – eBPF programs (shown in blue) running in kernel-space invoke BPF-DB's API to access and modify database contents through serializable, ACID-compliant operators. BPF-DB optionally logs writes to a buffer accessible from user-space which a consumer can then persist locally or replicate.

their client application and BPF-DB and load the programs into the kernel with the `BPF()` system call. Once verified and loaded into the kernel, BPF-DB instantiates its eBPF maps and applications can start accessing database contents. One benefit of compiling BPF-DB operators with client applications is that compilers can optimize the generated eBPF code for continuation definitions and current knob settings (e.g., WAL enabled, maximum number of versions, maximum locks per transaction). The result is specialized operators akin to pre-compiled stored procedures. For example, turning logging on or off is a compile-time constant for BPF-DB operators. If WAL is turned off at compilation time, an optimizing compiler like Clang or GCC will omit all WAL logic during dead code elimination. However, this design choice does not prevent developers from enabling BPF-DB's WAL during runtime. By later recompiling BPF-DB's operators with logging enabled, the compiler generates new eBPF programs that can be atomically swapped with existing programs using the `BPF()` system call.

5.2.1 ACID Semantics

BPF-DB provides ACID (i.e., atomicity, consistency, isolation, durability) guarantees to eBPF programs accessing its database contents concurrently. Transactions enable developers to group multiple database operations (e.g., GET, SET) together and appear as though they occur as a single logical event. BPF-DB also provides roll-back semantics to eBPF programs. When combined

Table 5.1: BPF-DB Operators and Tail Call Continuations – Application developers define eBPF programs for each of BPF-DB's possible tail call exits.

Operation	Tail Call Continuations
BEGIN	(1) OK, (2) Error
BEGIN (read-only)	(1) OK
GET	(1) Key found, (2) Key not found, (3) Error
GET (read-only)	(1) Key found, (2) Key not found
GET (auto-commit)	(1) Key found, (2) Key not found
SET	(1) OK, (2) Error
SET (auto-commit)	(1) OK, (2) Error
COMMIT	(1) OK, (2) Error
COMMIT (read-only)	(1) OK
ROLL BACK	(1) OK

with transaction atomicity, BPF-DB ensures that either all operations in a transaction complete or none of the operations complete.

BPF-DB makes it easier for eBPF developers to consider race conditions through consistency and isolation. For example, BPF-DB's key-value API maintains primary key semantics so developers can uniquely address records and use BPF-DB as their primary data store. For eBPF programs accessing BPF-DB from multiple threads, their transactions proceed as if running in isolation as long as their operations do not violate serializability. Developers receive feedback when their transactions cannot commit and can react depending on the semantics of their application (e.g., retry). BPF-DB's design provides eBPF programs with "exactly once" semantics that they previously lacked. Lastly, BPF-DB can optionally record modifying transactions to a write-ahead log (WAL), ensuring the durability of all writes to the DBMS. eBPF maps are not persistent and do not provide an interface to save their contents to disk. BPF-DB addresses this shortcoming by exporting a WAL of its operations that a user-space application can store on disk or replicate over the network.

5.2.2 Application Programming Interface

BPF-DB presents a key-value interface to its client application. Table 5.1 enumerates BPF-DB's operators and each operator's possible tail call continuations. Application developers must define all continuations for operators they use. Otherwise, BPF-DB's code will fail to compile.

Depending on the operator they are using, developers define continuations from a set of four possible states. GET operators require continuations for whether a key exists or not. The OK state signifies that the called operator succeeded, and is ready to tail call its continuation. In contast, the Error state denotes that the operation cannot proceed and that the transaction is now in a failed state and cannot commit. Error states can be originate from several sources. First, BPF-DB employs locking to provide ACID semantics, and lock conflicts for a GET or SET operator cause an Error state. Second, if WAL is enabled and BPF-DB is unable to write entry due to the buffer being full, it will generate an Error state. Lastly, BPF-DB checks the return

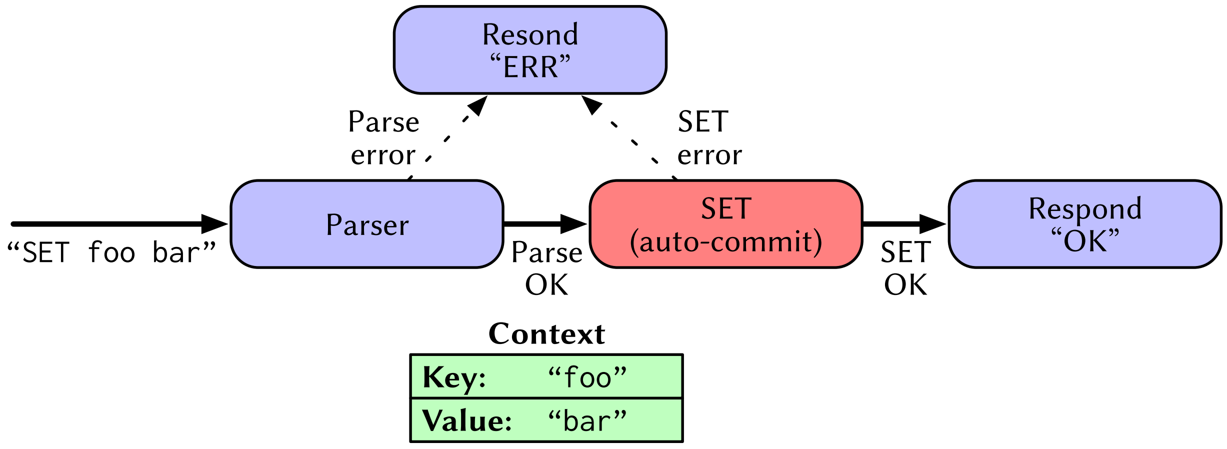


Figure 5.2: Continuations for SET – BPF-DB's SET operator (shown in red) with application logic (shown in blue). Conditional tail calls define the continuations for each eBPF program, represented by a box.

code eBPF helper function calls provided by the kernel. If any of those functions fail due to an internal constraint (e.g., out of memory), then BPF-DB reports an Error state.

BPF-DB provides read-only transactions for performance optimization and developer convenience. Because BPF-DB maintains multiple versions of database records, non-modifying transactions can proceed without blocking writers. Read-only transactions do not contain any Error states because they do not acquire locks or flush log records. Read-only transactions also do not perform writes on database contents using eBPF helpers. This design improves BPF-DB's performance and reduces the number of continuations for application developers to define.

Figure 5.2 shows a control-flow graph of BPF-DB's CPS design and integration with custom application logic. Each box represents an eBPF program, with application logic in blue and BPF-DB logic in red. In this example, a parser written in eBPF receives an input query string to set the key "foo" to the value "bar". If the string is a valid query, the parser populates a Context object that BPF-DB stores in an eBPF map with both the key and the value and tail calls into the SET operator. The BPF-DB SET operator runs to completion and either tail calls to the error handler because it could not set the value or tail calls to the success handler. This example uses strings for keys and values, but BPF-DB does not expose a notion of types in its API. Keys and values are byte sequences with corresponding length attributes, and applications use their own data encoding scheme. This design makes it easy for application developers to store binary or string-encoded data in BPF-DB, as demonstrated in Section 5.4.

BPF-DB's CPS architecture provides multiple benefits. First, BPF-DB's operators are self-contained eBPF programs accessed through tail calls, so their eBPF instructions do not count against an application's instruction limit. Second, we define BPF-DB's operators (along with their continuations) separately so developers compile and load BPF-DB into the Linux kernel independently. If an eBPF developer is writing their project in a language other than C (e.g., Rust, Go) they can continue to use their original language independently of BPF-DB. Lastly, the separation of BPF-DB and application logic not only simplifies development, but also provides runtime benefits. The Linux kernel atomically loads and unloads eBPF programs so developers can modify application and BPF-DB logic independently without downtime. We discuss some possibilities of this benefit in Section 7.2.

BPF-DB exposes a low-level DBMS API, similar to RocksDB. For this reason, developers are responsible for producing valid BPF-DB procedures. For example, custom procedures must not BEGIN a modifying transaction and then call BPF-DB's read-only COMMIT operator. We believe this is a suitable design choice for multiple reasons. First, as discussed in Section 7.2, eBPF is amenable to code generation from high-level scripting languages. A future domain-specific language and corresponding compiler could enforce BPF-DB constraints at code generation time, similar to checks within eBPF's verifier. Even without code generation, building a static analysis tool to enforce BPF-DB semantics could help developer productivity. Second, if an application provides interactive transactions to clients, whereby clients could invoke behavior that would violate BPF-DB's semantics, the application code could enforce this runtime validation logic. For example, the parser in Figure 5.2 could handle the case of an interactive client calling an invalid command sequence with its continuations rather than calling into BPF-DB. Third, eBPF developers are used to disciplined software design due to the eBPF verifier's strict requirements, and BPF-DB's requirements are no more onerous than existing eBPF limitations.

5.3 BPF-DB Design

In this section, we detail BPF-DB's design considerations in the context of traditional in-memory DBMS design. First, we discuss BPF-DB's storage manager, which uses eBPF maps to store variable-sized key-value pairs. Next, we present BPF-DB's concurrency control implementation for serializable transactions. Then, we show the challenges of ensuring durability for contents stored in kernel-space and describe BPF-DB's write-ahead logging solution. Lastly, we detail design and implementation considerations for BPF-DB's operators.

5.3.1 Storage Management

DBMSs rely on storage managers to store, organize, and retrieve database entries. This component builds upon complex data structures that ensure correctness even under concurrent operations. For example, a disk-based storage manager will store its databases in fixed-size pages that it organizes into pools with specialized eviction policies (e.g., LRU, clock). The storage manager is also responsible for maintaining supplemental data structures for indexes like B+ trees or hash tables. Some storage managers are key-value-based and rely entirely on tree-based data structures like the log-structured merge (LSM) tree. BPF-DB relies on an in-memory storage manager for variable-length key-value pairs, providing a design that generalizes to more complex data types for its host applications.

BPF-DB stores its database in kernel-resident data structures to access and manage the contents for other eBPF applications. As detailed in Chapter 2, the only memory that eBPF programs can use to retain data in between program executions is eBPF maps. In this execution environment, there is no API to access the heap (i.e., dynamic memory allocation) like `malloc()`. Thus, BPF-DB uses eBPF maps to store all of its database contents.

eBPF programs must define their eBPF maps and corresponding key and value sizes before

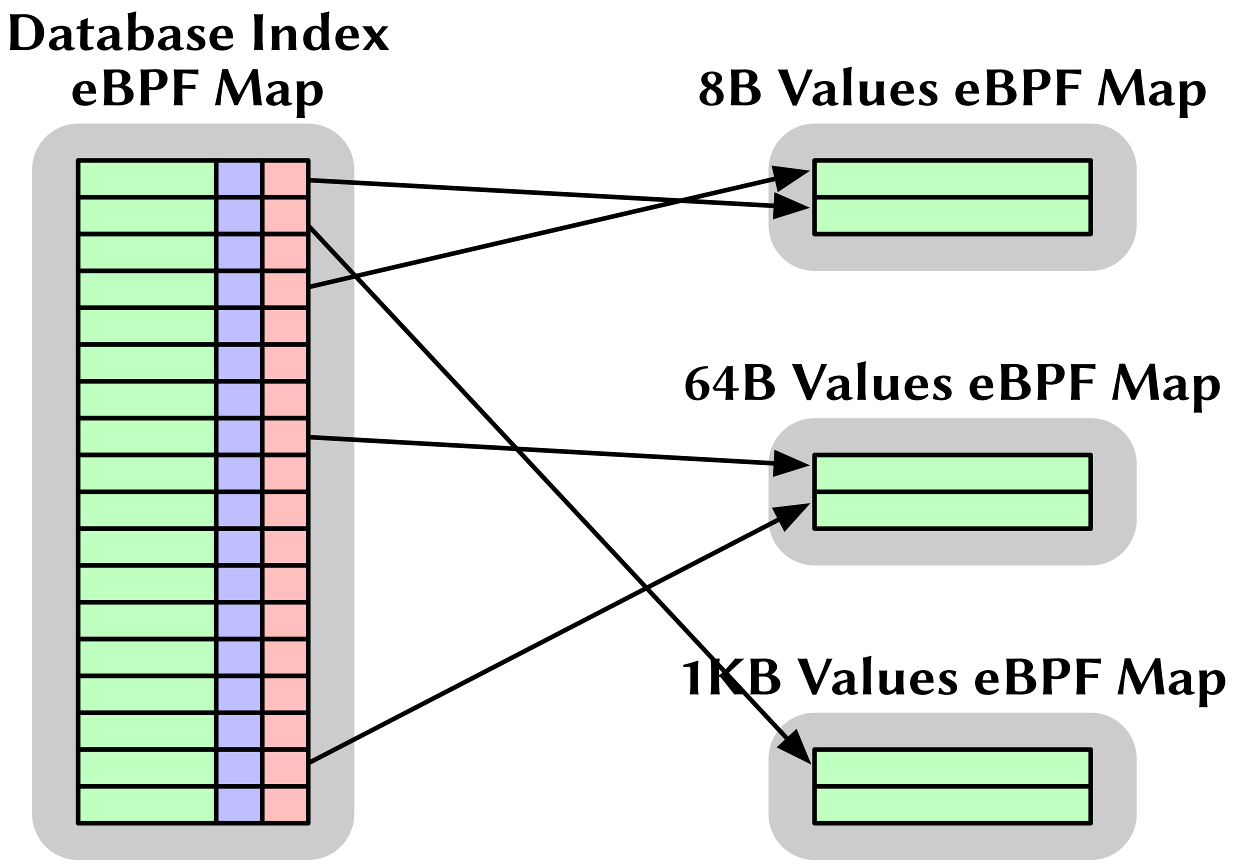


Figure 5.3: BPF-DB's Index and Value Maps – BPF-DB uses a top-level index where locks are stored inline with the key and version metadata. Values are stored in separate eBPF maps based on their size class. Values maps for other size classes are not shown.

being loaded into the kernel. This requirement exists because the verifier must enforce the rule that eBPF programs do not access keys or values out of bounds. A naïve implementation could use a single eBPF hash map for the entire database, where the keys and values are restricted to one size. However, this design would trade simplicity for memory fragmentation. Every key-value pair would use the maximum number of bytes regardless of the actual size of the entry. BPF-DB works around this lack of dynamism in eBPF value size by instantiating eBPF maps for different size classes. Then, BPF-DB stores values in the eBPF map that best suits their sizes, similar to user-space memory allocators that maintain free lists for different object sizes [74, 75, 121, 208].

Figure 5.3 shows the design of BPF-DB's hierarchical eBPF maps that store its database contents. The first level is the database Index, which contains keys and metadata (e.g., locks, timestamps) for every database entry. For brevity, Figure 5.3 only shows a subset of the value maps for different size classes. BPF-DB first performs a lookup in the database index to access a database entry, . For a GET operator, the absence of an index entry is enough information to call to the continuation for the key not existing in the database. If an index entry does exist, then the GET operator must perform a second eBPF map lookup into the correct value map.

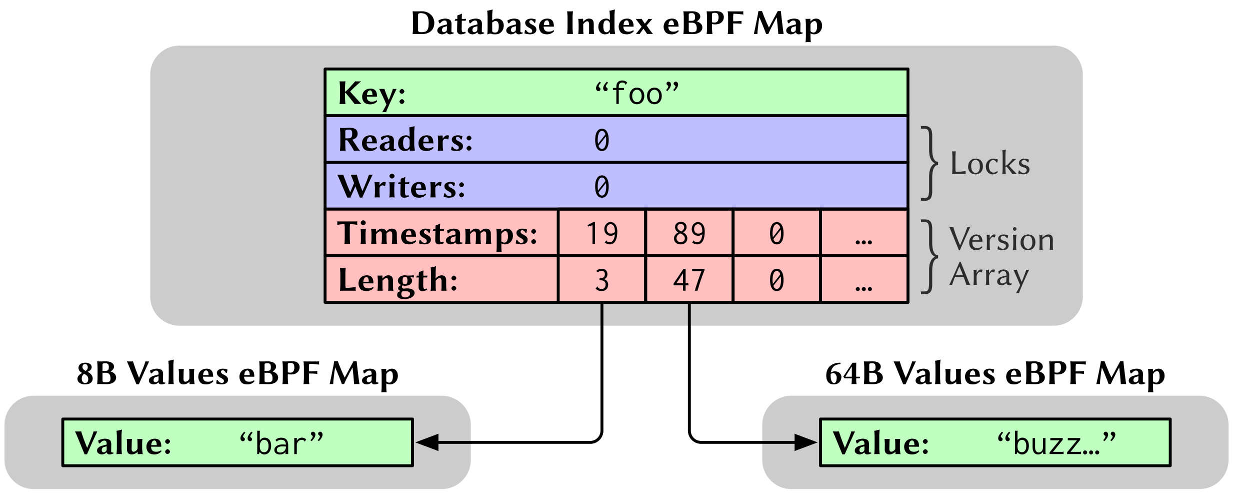


Figure 5.4: Database Key with Multiple Values – BPF-DB maps a single key entry to multiple values based on their commit timestamp. These values can belong to different size classes.

BPF-DB employs multi-versioned concurrency control (MVCC) to provide isolation between concurrent transactions. Figure 5.4 details the contents of a single entry in the database index, showing a key "foo" with two corresponding versions: (1) a "bar" value in one size class and (2) a "buzz..." value in another size class. Rather than using a centralized lock table, BPF-DB stores locks inline with the key and version metadata, a common design for in-memory DBMSs [116]. We elaborate on BPF-DB's locking and concurrency control in Section 5.3.2.

Each database index entry also contains MVCC information for its key. Unlike many DBMSs, the number of versions for each entry is not unbounded. There are several reasons for this design choice. (1) The eBPF verifier requires compile-time bounds on all loops, so traversing a version chain requires an *a priori* limit on the number of elements that a eBPF program will inspect. (2) The lack of dynamic memory allocation precludes a typical linked list implementation for the version chain. (3) BPF-DB assumes short transactions; thus, it will not need to maintain many versions of an entry for extended periods.

Typical DBMSs that employ MVCC order their version chains based on timestamps, either from newest to oldest or oldest to newest [211]. The former optimizes for recent transactions to scan fewer entries of the chain at the expense of increased contention and possible index maintenance at the head of the chain. The latter simplifies index maintenance during updates, deferring their updates until GC prunes old versions.

BPF-DB implements neither of these version chain orderings—instead it maintains an unordered array of 8 B version timestamps. When a GET or SET operator needs to determine the correct version of a record to access based on its timestamp, it scans the whole array. As described above, version chains must be bounded to satisfy the eBPF verifier, and BPF-DB assumes transactions are short-lived and thus does not need to maintain many old versions. For these reasons, we assume that a version array fits within a CPU's 64 B L1 cache line, corresponding to a maximum number of either versions. Scanning a data structure that fits within a cache line is fast, and the added logic of maintaining an ordered version array is not worth the verifier complexity penalty.

Updating a database entry creates a new version; eventually, the old version will no longer be visible to any new transactions. For this reason, BPF-DB must perform garbage collection (GC) to prune old versions from the system. BPF-DB employs a cooperative GC approach where SET operators, if the version array is full, then SET also removes any obsolete versions. We use cooperative GC to exploit the fact that the SET operator must acquire the write lock on an entry to update it so we can avoid added contention from an external GC worker trying to acquire write locks. This design has the trade-off of potentially allowing stale versions to persist in the database until a key's version array fills and triggers cooperative GC. We explore changing version array sizes and disabling MVCC in Sections 5.5.8 and 5.5.9, respectively. If cooperative GC retains too many stale versions, a future optimization for BPF-DB would be to create a background kernel worker (written in eBPF) that uses the `bpf_timer()` helper to perform background GC at a fixed interval. This approach would come at the expense of increased per-entry lock and eBPF map latch contention. This design is evocative of Deuteronomy [145], which applies soft and hard limits to the total number of versions in the system to trigger GC. However, BPF-DB's limits are more fine-grained on a per-key basis.

5.3.2 Transaction Management

Multi-statement transactions are a hallmark feature of DBMSs. The transaction manager applies policies to enforce a concurrency control protocol. These protocols maintain the ACID properties of the DBMS, ensuring (1) all operations in a transaction appear to occur atomically, (2) the database maintains a consistent view to users, (3) transactions run in isolation from each other, and (4) and changes to the database are durable before a transaction completes.

Transaction managers apply concurrency control methods like two-phase locking (2PL), optimistic concurrency control (OCC), or timestamp ordering (TO) to allow for inter-query parallelism [211]. Each method has trade-offs, and multi-versioned systems incur additional choices like version chain ordering (e.g., newest-to-oldest, oldest-to-newest), garbage collection schemes, and version contents.

eBPF's execution environment limits traditional concurrency control implementations. For example, each operator is a self-contained eBPF program and thus must encapsulate all concurrency control logic for that operator. Also, BPF-DB requires a concurrency control protocol that is simple enough to satisfy the eBPF verifier. OCC introduces non-determinism, whereby commit logic must reevaluate database operations against others to detect conflict. As described in Section 5.1.2, eBPF's inability to nest spin latches or call eBPF helper functions while holding a latch restricts concurrency control protocols with complex mutual exclusion logic. Lastly, eBPF programs are event-driven and cannot yield their thread or suspend execution. This requirement precludes concurrency control methods that require an external scheduler whereby an operator would enqueue a request and await its scheduled result [177, 197].

For these reasons, BPF-DB does not have a centralized transaction manager. BPF-DB applies a multi-versioned Strict Two-Phase Locking (Strict-2PL) for serializable transactions [71]. With this approach, operators acquire read and write locks that reside inline with the records [116] rather than in a centralized lock table. The "Strict" implementation of 2PL requires transactions to hold all their locks until they either commit or roll back. BPF-DB's timestamp-based multi-versioning presents performance benefits. For example, we adopt a multiversion mixed method [71] (ROMV [207]) whereby read-only transactions do not acquire any read locks but still maintain serializable properties.

Locking-based concurrency control introduces the potential for deadlocks, and DBMSs typically implement either deadlock detection or deadlock prevention. Deadlock detection generates dependency graphs and then applies a cycle detection algorithm—logic that is too complex for the eBPF verifier. Instead, BPF-DB employs a policy for lock acquisition that prevents deadlocks. BPF-DB allocates timestamps for multi-versioning but does not use these timestamps to implement a wound-wait or wait-die deadlock prevention policy. These policies require coordination between workers to implement lock stealing and coordinate restarts, and eBPF does not provide primitives for this behavior. Instead, BPF-DB applies a no-wait policy for lock acquisition that provides the fastest transaction performance for in-memory DBMSs [211].

5.3.3 Logging and Checkpointing

In-memory DBMSs must write their database contents to persistent storage to ensure durability. These systems rely on two methods to guarantee that they can restore their database contents after a restart: logging and checkpointing. Logging and checkpointing components are the least amenable to applying user-bypass. This conflict stems from eBPF's inability to write to disk. I/O operations like writes can block for an indeterminate amount of time, which is not allowed during the execution of eBPF programs. We will discuss the challenges and opportunities of applying these techniques in BPF-DB.

Write-ahead logging (WAL) requires the DBMS to maintain a ledger of all database modifications. Upon restart, the DBMS replays the log to reconstruct the database contents. The standard method to perform these tasks is ARIES [165], though modern systems introduce new optimizations [125]. Most DBMSs expose a knob that determines whether transactions commit synchronously or asynchronously. With asynchronous durability, the DBMS submits a transaction's changes to the log writer but can acknowledge transaction completion to the client without guaranteeing that the changes are persistent on disk. Synchronous durability ensures the OS has written a transaction's contents via the `write()` system call. It may also guarantee that the system performs the `fsync()` system call to ensure OS write buffers are flushed.

eBPF's inability to initiate I/O (i.e., disk or network writes) necessitates a user-space component to persist BPF-DB's database contents. Thus, BPF-DB takes a hybrid user-space and kernel-space approach to logging. If WAL is enabled, BPF-DB sends all write operations to a ring buffer accessible from user-space. If the ring buffer is full and BPF-DB cannot complete its write operation, it reports an error and tail calls the error continuation. The application must roll back the transaction because not all its writes are in the log.

As shown in Figure 5.1, a WAL consumer process in user-space consumes the contents of the ring buffer. eBPF's ring buffer offers the ability to notify user-space consumers via `epoll()` when to consume the buffer. BPF-DB offers a tunable knob whereby it will only notify waiters after writing a specified amount of data. This approach reduces the amount of signal handling overhead between kernel-space and user-space, and can be adjusted based on the desired wakeup frequency of any user-space consumers.

After wakeup by notification or a timeout, the user-space component consumes data from the buffer, providing asynchronous durability for BPF-DB writes. BPF-DB writes timestamps out with its logical log records, and provides an eBPF map entry for WAL consumers to update the last persisted timestamp. With this approach, client applications implement their durability semantics and can optionally check if their writes are durable before proceeding. The fixed-sized ring buffer and the timestamp for most recently flushed writes present two possible mechanisms for applications built on BPF-DB to apply back pressure to a workload. We evaluate the performance and trade-offs of this logging approach in Section 5.5.4.

Checkpointing is the process of writing all the contents of the database to disk, thus preserving a consistent snapshot of the database at a single point in time. Upon restart, the DBMS reads the checkpoint file's contents to load the database contents into memory. The most straightforward approach to checkpointing is to block all transactions that are not read-only while flushing the database contents to disk. Once the checkpoint is complete, all transactions can resume. This approach limits the transaction performance of the system if there are no predictable periods of quiescence. DBMSs commonly support fuzzy checkpointing [178], whereby all transactions continue to run during the checkpointing process. However, the DBMS performs extra bookkeeping to guarantee that the checkpointing component sees a consistent snapshot of the database contents.

Similar to the limitations related to I/O initiation for logging, checkpointing also requires a user-space component, which we defer for future work. However, we still discuss some of the challenges associated with checkpointing below. We discuss future work related to checkpointing in greater detail in Section 7.2.

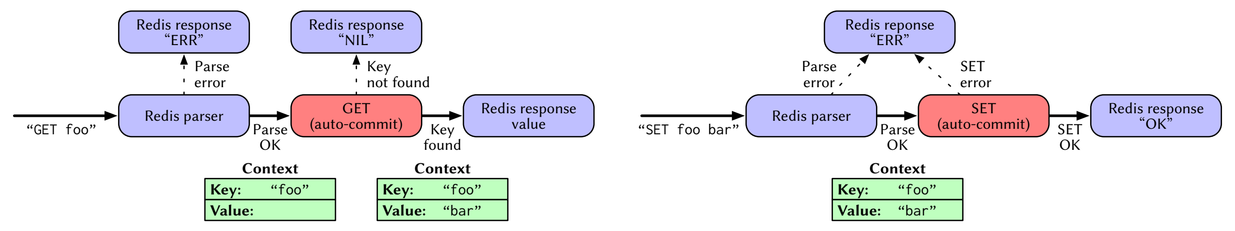
Techniques that rely on virtual memory's copy-on-write to create a snapshot of database contents by calling `fork()` on the DBMS's process will not work with a user-bypass system because the database is stored in kernel memory [135]. Instead, a user-space process must read the contents of BPF-DB's eBPF maps using the `BPF()` system call. This approach requires the DBMS to stop accepting queries because no DBMS locking mechanisms are available from user-space.

5.4 Applications

Section 5.2.2 describes BPF-DB as an embedded DBMS that enables developers to build eBPF applications around its operators. We now detail the two sample applications we use to demonstrate and evaluate BPF-DB: (1) a transactional key-value store and (2) a multi-statement transaction workload.

5.4.1 Transactional Key-Value Store

The first system we use to evaluate BPF-DB is a transactional key-value store (KVS) that supports a modified Redis network protocol. We implement an eBPF program that parses a subset of Redis commands and types with continuations to call into corresponding BPF-DB operators. This design allows us to reuse Redis's command-line interface and benchmarking tools. The KVS's parser supports basic `SET` and `GET` commands. We also extended the protocol with custom commands for beginning and ending transactions because the Redis protocol's `MULTI`/`EXEC` commands do not provide serializable ACID semantics. Redis is a string-based protocol, so the



(a) GET **(b)** SET

Figure 5.5: Transactional Key-Value Store Control-Flow Graph – The transactional KVS composes BPF-DB's GET and SET commands (shown in red) with application logic (shown in blue). Conditional tail calls define the continuations for each eBPF program.

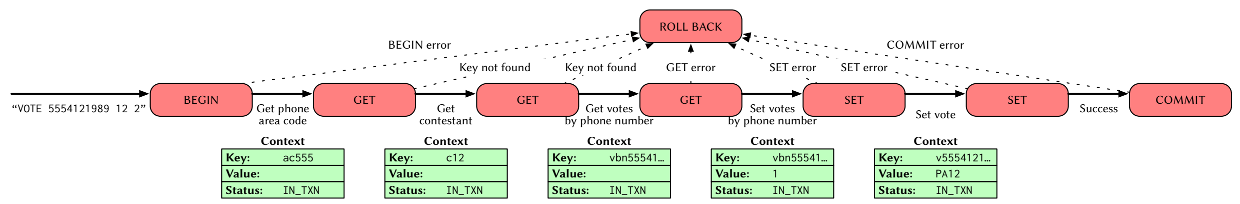


Figure 5.6: Voter Control-Flow Graph – The VOTE procedure from the Voter benchmark includes three GET and two SET commands with application logic to validate between each operator.

transactional KVS stores all keys and values in their native Redis encoding. We also define eBPF programs that serialize and send Redis protocol responses over the network to the client as BPF-DB continuations.

Figure 5.5 shows an example of the control-flow graphs to support GET and SET. In Figure 5.5a, the query "GET foo" is input to the first piece of application code, the Redis parser. If the input string parses as a GET command, the application populates BPF-DB's context object with the key and tail calls into BPF-DB's GET operator. GET has two possible outcomes: (1) the provided key is not found, in which case BPF-DB tail calls to an eBPF program that responds with Redis's NIL object, or (2) the key is found, and the continuation eBPF program serializes the value response to the client. Figure 5.5b shows similar behavior but with continuations that reflect SET's possible continuations.

5.4.2 Voter

The second system exercises BPF-DB's serializable transactions with multiple GET and SET operators per transaction. We implement the Voter benchmark [193], which models a call-in voting system. We only run the VOTE stored procedure and drop any views and procedures related to results tallying. Parameters include the number of contestants and the maximum number of votes per phone number, which we set to 12 and two, respectively. The schema consists of (1) CONTESTANTS table that maps contestant identifiers to contestant names, (2) VOTES table that records the phone number, state, and contestant identifier of each vote, and (3) AREA_CODE_STATE that maps a set of phone area codes to their states. We briefly describe the VOTE stored procedure below:

- The workload generator provides three arguments. (1) The ten-digit phone number's area code is selected uniformly from a known set, with the seven remaining digits generated from a uniform distribution. (2) The contestant identifier is uniformly chosen from the set of valid contestants, but approximately 1% of VOTE operations contain an invalid contestant number to exercise the stored procedure's validation logic. (3) The maximum number of votes is defined at the start of the workload and remains constant for all invocations of the VOTE procedure.
- The DBMS looks up the phone area code to determine its state.
- The DBMS looks up the contestant identifier to verify its validity.
- The DBMS looks up the phone number's vote count to see if the phone number is over its vote limit.
- The DBMS records a new vote if the input arguments satisfy the constaints.

Figure 5.6 shows how these steps map to BPF-DB operations. The input to the system is the name of the stored procedure, the phone number, the contestant identifier, and the vote limit. VOTE's logic is simple enough that we do not need separate eBPF programs between each operator. Instead, it can embed the data validation logic (e.g., check if the phone number is over its vote limit) in the continuation definitions before tail calling the next operator.

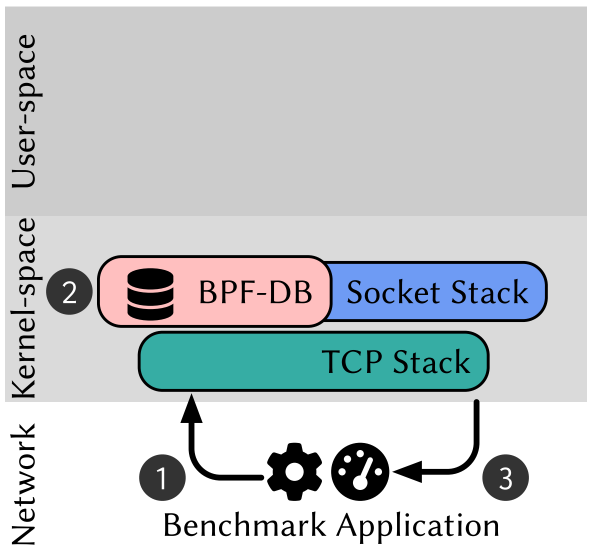
Because of its KVS interface, BPF-DB does not expose explicit schemas or tables. Instead, BPF-DB's Voter application prepends keys with distinct prefixes so entries in different tables map to distinct key spaces. To mimic the view for the number of votes by each phone number, BPF-DB's implementation defines a separate keyspace and explicitly maintains an entry for each phone number. This approach requires an extra SET for BPF-DB compared to the VOTE procedure's SQL definition.

5.5 Evaluation

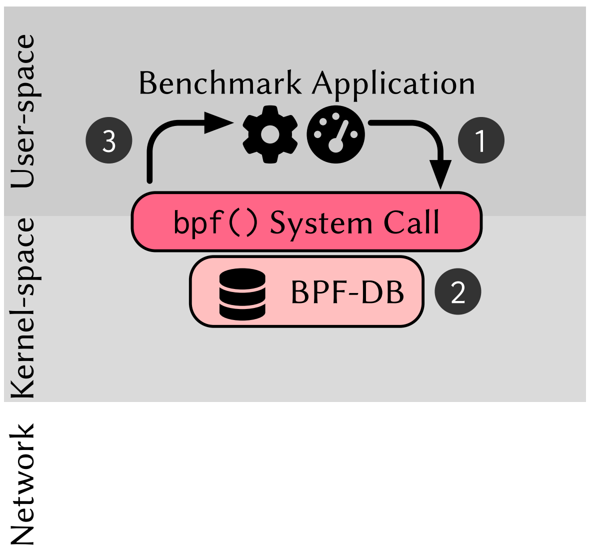
In this section, we evaluate BPF-DB's design and performance characteristics. First, we describe our experimental setup, including environmental parameters and workloads. Then, we provide an in-depth discussion of each experiment and its results.

5.5.1 Experimental Setup

eBPF programs are typically attached to other pieces of software (e.g., kernel stack, user-space programs) and execute when a running process traverses their attachment point. For network applications, developers have several choices of where to attach eBPF programs depending on their performance and feature requirements. For our transactional KVS, we attach its initial eBPF program (i.e., Redis parser) in the socket layer of the network stack, as shown in Figure 5.7a. When a buffer arrives on specified sockets, the Redis parser and subsequent BPF-DB continuations will run. This approach introduces network stack overheads but allows us to compare BPF-DB's performance against other Redis-compatible systems.



(a) Network Application



(b) Local Application

Figure 5.7: BPF-DB Benchmarking Strategies – We benchmark BPF-DB in two ways: (1) as BPF-DB applications attached at the socket layer of the kernel's network stack as in Figure 5.7a, driven by an external tools over the network, and (2) loaded into kernel-space not attached to any events as in Figure 5.7b. Instead, we drive BPF-DB with the `bpf()` system call from user-space threads.

To eliminate external bottlenecks and more directly measure BPF-DB's performance, we also run workloads where a user-space thread invokes BPF-DB operations, as shown in Figure 5.7b. These threads use the `bpf()` system call and its `BPF_PROG_TEST_RUN` argument to run BPF-DB in a controlled fashion. In this scenario, we no longer evaluate BPF-DB in the network layer, so we do not need the Redis parser before BPF-DB's `GET` or `SET` operators. We do not need continuations to serialize and send the result back to a network client. Compared to Figure 5.7a, there are no software interrupts for the kernel to schedule, no extra buffers to copy, and no network protocols or NIC driver overheads. This scenario targets BPF-DB's operators directly.

We evaluate BPF-DB on servers that contain 2×20-core Intel Xeon Gold 5218R CPUs, 192 GB DRAM, a Samsung PM983 SSD, and a dual-port 10GbE network adapter. All servers run Ubuntu Linux 22.04 LTS (Linux kernel v5.15) and connect to a Cisco Nexus 3064 10GbE network switch. Unless otherwise specified, we use separate servers for DBMSs and workload generators. We pin user-space programs to a single CPU socket to avoid NUMA effects. Similarly, we configure the NIC to use 20 receive queues, each with affinity set to a dedicated CPU core on the local NUMA node. Lastly, we optimize the network stack by disabling interrupt rate limiting, expanding network buffer sizes, and turning off TCP timestamps and selective acknowledgements.

Workloads: We compare BPF-DB's transactional KVS against three other open-source Redis-compatible proxies:

- **Redis (v7.2.4):** Redis is one of the most widely deployed in-memory key-value stores. Due to its popularity and maturity, we use Redis as the baseline implementation for user-space key-value stores.
- **KeyDB (v6.3.4):** KeyDB is a multithreaded fork of Redis. Developers added user-space worker threads and coarse-grained latches to maintain consistency across workers. This implementation enables us to evaluate the Redis architecture with added parallelism. We configured KeyDB to use four worker threads, as their documentation does not recommend increasing beyond this number due to latch contention.
- **Dragonfly (v1.14.4):** This multithreaded KVS is a new in-memory implementation of the Redis protocol. Dragonfly employs key-space partitioning, lightweight user-space threads (i.e., fibers), asynchronous I/O, and a locking scheme inspired by VLL [177] to maintain consistency across workers. Dragonfly serves as the state-of-the-art implementation for our networked KVS comparison. We configure Dragonfly to use 20 worker threads to match the number of physical CPUs on its NUMA node.

In these networked KVS scenarios, we use memtier_benchmark [31] for workload generation and modify its key generation to support skewed Zipf distributions ($\theta = 0.99$). We load 10M keys for each experiment and use 16 B and 1 KB value sizes to reflect typical in-memory cache workloads and YCSB [88], respectively. memtier_benchmark submits the workload in a closed loop across 1,000 simultaneous clients on 80 worker threads. Each configuration runs five times. All Redis workloads run with logging and checkpointing disabled because each DBMS provides different durability semantics. For this reason, we evaluate BPF-DB's WAL characteristics in isolation in Section 5.5.4.

In addition to Redis workloads, we evaluate BPF-DB's serializable multi-statement transaction performance. We compare against **VoltDB** [58] (now Volt Active Data), a high-performance, ACID-compliant in-memory DBMS that supports serializable transactions. VoltDB is based on the H-Store [133] academic project, and applies performance optimizations like horizontal partitioning (i.e., sharding) across CPU threads and pre-compiled Java stored procedures. This design is a valuable comparison to BPF-DB, which supports chaining multiple operations together through tail calls to run all of a transaction's statements in a single operation. We configure VoltDB to use 20 partitions to maximize its performance when pinned to 20 dedicated CPU cores.

5.5.2 KVS Throughput (Networked)

We begin our evaluation by investigating the transactional KVS as a networked in-memory DBMS, a typical deployment scenario for Redis-compatible systems. This experiment compares BPF-DB against other Redis-compatible in-memory systems for mixed `GET` and `SET` workloads. We attach the networked KVS described in Section 5.4.1 at the kernel's socket layer. When a socket buffer arrives at the network layer, the networked KVS (1) parses the message for a valid `GET` or `SET` message, (2) parses the key and (optional) value, (3) tail calls into the corresponding BPF-DB operation, and lastly (4) tail calls into continuations that construct and send the response to the network client.

In addition to measuring with both 16 B and 1 KB values as described in Section 5.5.1,

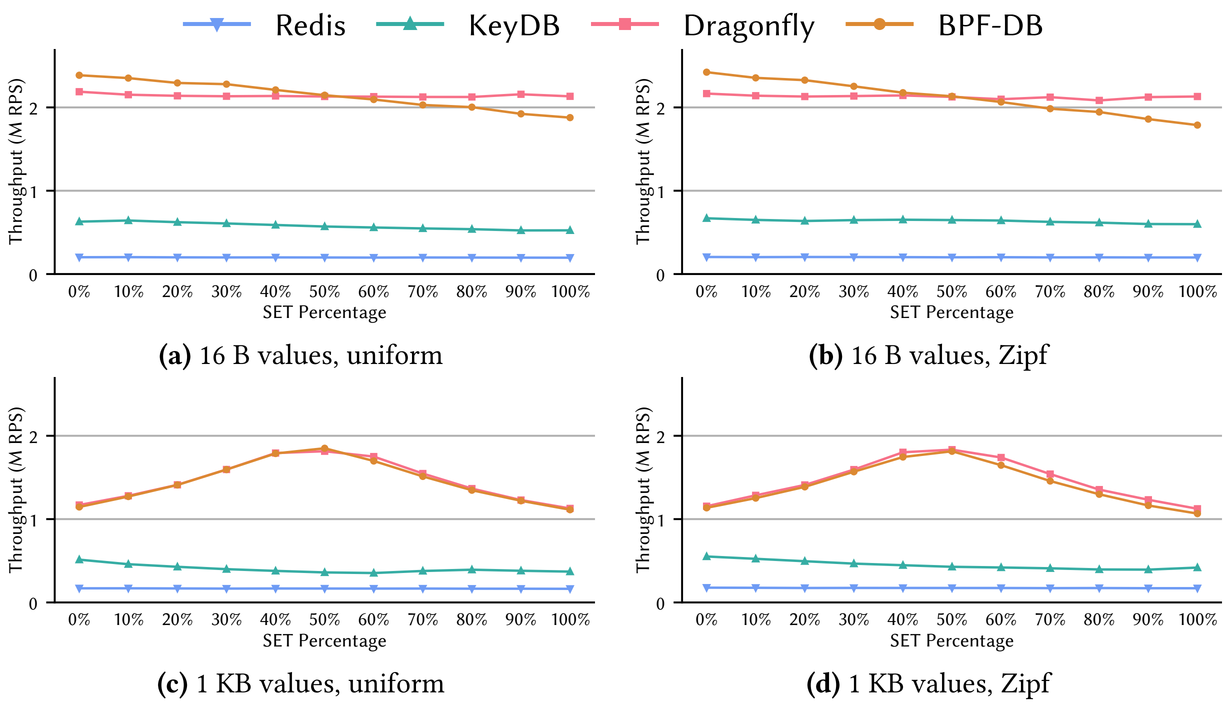
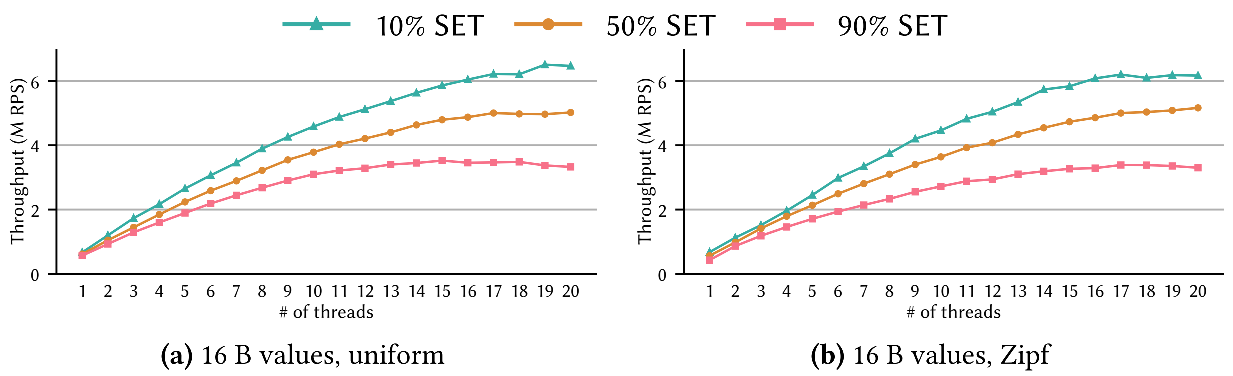


Figure 5.8: KVS Throughput (Networked) – Total requests per second over 10GbE network (Redis protocol) while varying the ratio of SET commands from read-only to write-only. Scenarios differ in value size (16 B or 1 KB) and key distribution (uniform or Zipf).

we also want to evaluate how different ratios of reads and writes affect DBMS performance. Increased writes can reduce the scalability of a DBMS due to memory allocations, data structure synchronization (e.g., latches), and concurrency control protocols (e.g., 2PL, OCC). For this reason, we scale the ratio of SET operations from 0% (i.e., read-only) to 100% (i.e., write-only) in 10% increments and measure the maximum throughput as reported by memtier_benchmark.

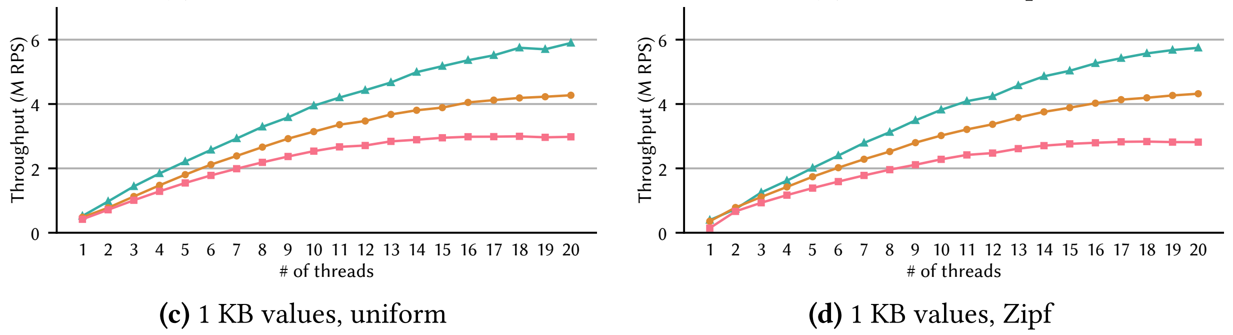
Figure 5.8a shows the throughput for 16 B values with uniform random key generation. As a single-threaded user-space application, Redis offers the lowest performance with 202 K requests per second (RPS) for the read-only scenario, dropping 3% to 196 K RPS as SETs increase. KeyDB's thread-based parallelism enables it to outperform Redis, albeit with a speedup that is not linear to its four worker threads. KeyDB starts at 629 K RPS for the read-only scenario, dropping 17% to 524 K RPS in the write-only scenario. In general, Redis and KeyDB exhibit similar trends in the other scenarios (i.e., value size and key distribution) of Figure 5.8, and due to their non-competitive throughput (<1 M RPS), we omit further discussion of their performance.

In Figure 5.8a, for the read-only scenario, Dragonfly processes 2.19 M RPS while BPF-DB achieves 2.39 M RPS, a 9% increase in throughput. However, as writes increase, the performance lead flips: KeyDB yields 2.13 M RPS and BPF-DB performs 1.88 M RPS in the write-only scenario. In this scenario, Dragonfly's user-space data structures are more efficient than BPF-DB's kernel-space eBPF maps, which require internal latches and RCU synchronization. As writes increase, the overheads of these synchronization methods hurt BPF-DB's throughput. Figure 5.8b exhibits similar trends, demonstrating the performance of BPF-DB's strict-2PL concurrency



(a) 16 B values, uniform

(b) 16 B values, Zipf



(c) 1 KB values, uniform

(d) 1 KB values, Zipf

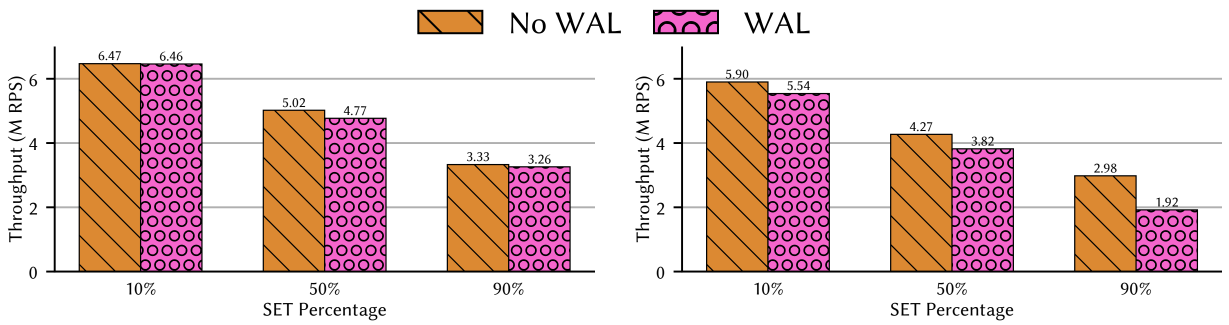
Figure 5.9: KVS CPU Scalability (Local) – Total requests per second running BPF-DB locally while varying the number of CPU cores dedicated to BPF-DB. We measure three ratios of SET commands, and scenarios differ in value size (16 B or 1 KB) and key distribution (uniform or Zipf).

control and bounded version vectors, even under high contention scenarios like skewed key distributions.

In Figure 5.8a, Dragonfly and BPF-DB with 1 KB values exhibit different behavior compared to 16 B values. These results demonstrate that Dragonfly and BPF-DB can saturate the network bandwidth limits of the test environment. At read-only and write-only extremes, the network limits DBMS throughput to 10 Gbps, while more diverse workloads benefit from bidirectional bandwidth. Throughput peaks when GET and SET each make up half of the requests. As a backing store for a networked KVS, BPF-DB remains competitive with state-of-the-art implenentations like Dragonfly and offers fully serializable transactions.

5.5.3 KVS CPU Scalability (Local)

Next, we remove the network bottleneck that limits throughput in Section 5.5.2 to find BPF-DB's maximum throughput. In this scenario, we use the `bpf()` system call and its `BPF_PROG_TEST_RUN` argument to trigger BPF-DB GET and SET operators from user-space threads in a controlled manner. We scale the number of CPU threads running in user-space to understand better BPF-DB's scalability under increasing resource (e.g., CPU, memory, lock) contention. Once again, we load 10 M keys before test execution and run the workload with different ratios of SET operations, value sizes, and key distributions. We will use this configuration for the remaining KVS experiments.



(a) 16 B values, uniform

(b) 1 KB values, uniform

Figure 5.10: Write-Ahead Logging – Total requests per second running BPF-DB locally with WAL disabled and enabled. We measure three ratios of SET commands, and scenarios differ in value size (16 B or 1 KB) with a uniform key distribution to reduce contention and maximize throughput.

Figure 5.9 shows the throughputs achieved by BPF-DB driven by local CPU threads using the `bpf()` system call. The 90% SET scenario plateaus earlier than the less write-intensive workloads, demonstrating the increased write overhead with eBPF maps' RCU synchronization. In the 16 B values and uniform key distribution scenario, 10% SET levels off at 6.5 M RPS, continuing to scale almost to the number of CPU cores (20). 90% SET reaches maximum performance near 3.5 M RPS much earlier, where synchronization overheads limit CPU core utilization to 15 of the total 20 cores.

Changing the key distribution from uniform to Zipf does not introduce a significant performance drop. The added lock contention and garbage collection work are not the bottleneck; instead, they emphasize again that the underlying eBPF maps limit performance.

Increasing the value size to 1 KB reduces throughput in all test scenarios due to the increased time spent copying values to and from BPF-DB. In the 1 KB values and uniform key distribution scenario, 10% SET plateaus at 5.9 M RPS, 9% lower than the test with smaller value sizes. 90% SET levels off at 3 M RPS, this time a 14% drop compared to smaller value sizes. Larger value sizes introduce longer critical sections to eBPF map synchronization mechanisms; thus, the write-intensive workload suffers from a more significant performance drop. This analysis mirrors Linux's recommendations for kernel patches that rely on RCU, which states that RCU is not a scalable synchronization technique when writes constitute more than 10% of the workload [46]. However, eBPF does not expose an alternative synchronization method for eBPF maps, so BPF-DB must use the provided interface.

5.5.4 Write-Ahead Logging Throughput (Local)

We now quantify the performance impact of BPF-DB's WAL architecture, as described in Section 5.3.3. In this scenario, we want to maximize BPF-DB's requests per second to exercise its WAL code path and reduce overheads in other system areas. For this reason, we only generate keys from a uniform distribution. Zipf key distribution accesses a small number of keys, generating lock contention and many versions for cooperative GC to clean up.

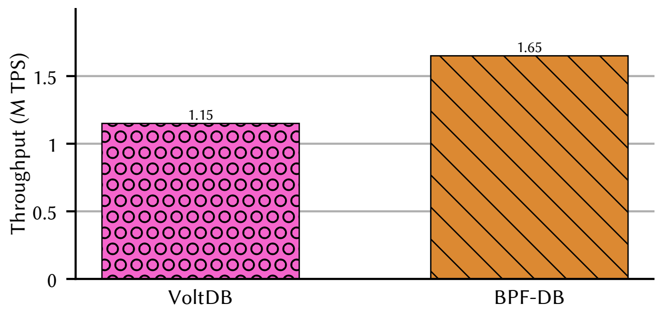


Figure 5.11: Voter Throughput (Networked) – Total transactions per second over 10GbE network compared to VoltDB.

We run BPF-DB's local KVS with 20 worker threads with multiple `SET` ratios and then repeat the experiment with BPF-DB's WAL enabled, as described in Section 5.3.3. To not be limited by disk throughput, the user-space log consumer immediately removes data from the shared ring buffer and does not write log records to disk. Subsequently, sending this data to another device (e.g., replicated over the network or written to a local disk) would introduce bottlenecks that do not measure the overhead of BPF-DB's user-space WAL design. Knobs related to the WAL are as follows: 512 MB ring buffer, BPF-DB notifies any processes waiting on the ring buffer's file descriptor after writing 2 MB, and the user-space consumer has a timeout of 1 ms if it does not receive a notification to consume the buffer.

Figure 5.10 shows BPF-DB's throughput with WAL enabled and disabled. The 90% `SET` scenario with both 16 B and 1 KB values induces the most writes and, thus, is the most taxing on the WAL architecture. With 16 B values, the write-heavy scenario drops from 3.33 M RPS to 3.26 M RPS, showing that WAL incurs a 2% overhead. None of those operations incur a roll-back due to a full ring buffer, so we attribute this overhead exclusively to the additional instructions for copying and submitting write operations into the ring buffer.

The 90% `SET` 1 KB scenario suffers a 35% drop in performance with WAL enabled, going from 2.98 M RPS to 1.92 M RPS. The larger value sizes require BPF-DB to spend more CPU cycles copying data into the ring buffer and induce more work for the user-space consumer. This workload averages almost 1.4 M roll-backs per second due to a full ring buffer when BPF-DB attempts to submit data. This experiment demonstrates the limitations of BPF-DB's user-space WAL design, where the user-space consumer cannot exceed approximately 2 GB/s. We could enhance BPF-DB for write-heavy scenarios by applying parallelism techniques similar to those in Silo [199]: multiple ring buffers could have multiple user-space consumers if the host can write more than 2 GB/s to storage. We defer this optimization for future work.

5.5.5 Voter Throughput (Networked)

We now evaluate BPF-DB's multi-statement transaction performance compared to VoltDB using the Voter benchmark described in Section 5.5. For VoltDB, we use its asynchronous workload generator to maximize throughput over the network. However, we modify the driver to use multiple threads because one instance cannot saturate our VoltDB server. For BPF-DB, we define a stored procedure that contains `BEGIN`, three `GET`s, two `SET`s, and either `COMMIT` or `ROLL`

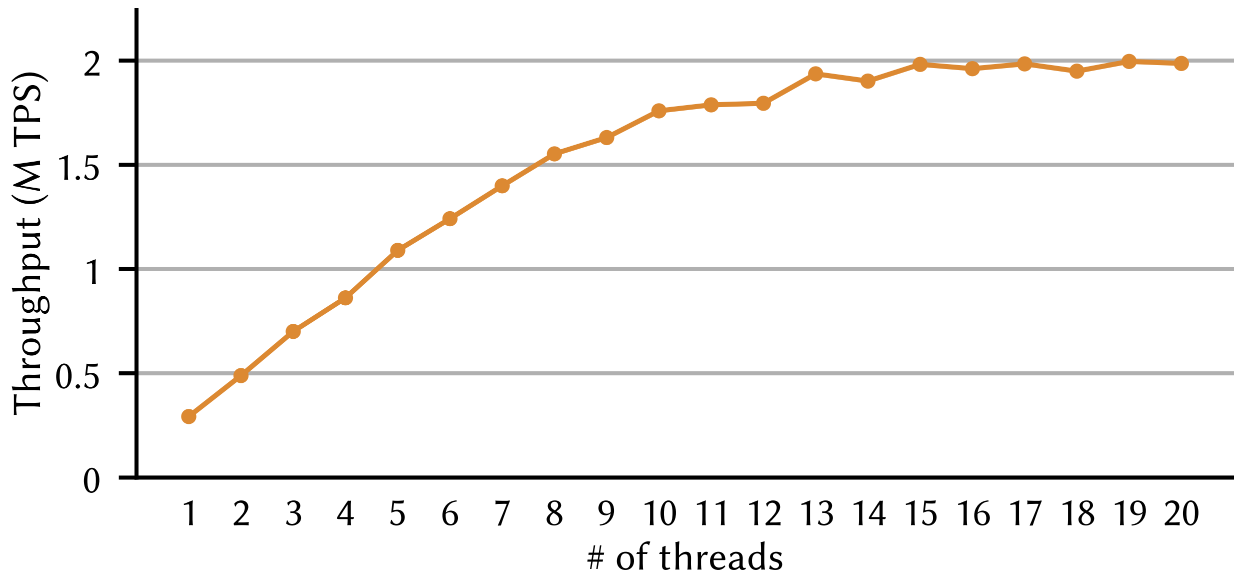


Figure 5.12: Voter CPU Scalability (Local) – Total transactions per second running BPF-DB locally while varying the number of CPU cores dedicated to BPF-DB.

BACK stitched together through tail call continuations. We implement the logic in eBPF between each operator to maintain the VOTE procedure's constraints. As in Section 5.5.2, we attach the stored procedure in the kernel's socket layer and prepend BPF-DB's operations with a parser to extract the arguments. We modify memtier_benchmark to generate arguments that match Voter's specifications, and trigger the stored procedure with a new VOT Redis command. This design allows us to reuse memtier_benchmark's network protocol, thread-based parallelism, optional rate-limiting, and summary statistics to evaluate this workload.

Figure 5.11 shows the Voter benchmark's maximum throughput, showing that BPF-DB achieves 43% higher throughput than VoltDB. The KVS throughput in Figure 5.8 achieves 2 M RPS for 16 B values, allowing us to quantify the overhead of running multiple BPF-DB operations in a transaction. In this experiment, BPF-DB processes 1.65 M transactions per second (TPS). We attribute this performance drop, compared to single statement operations, primarily to the extra eBPF map operations of reading and updating multiple values. Despite its partitioned and lock-free architecture, when we profile VoltDB at maximum throughput we find that it spends significant time in futex and synchronization calls introduced by the Java Virtual Machine and user-space thread scheduling. We discuss BPF-DB's Voter performance in greater detail in the following section.

5.5.6 Voter CPU Scalability (Local)

Like Section 5.5.3, we remove network overheads from the workload and instead drive BPF-DB's VOTE stored procedure with user-space threads using the `bpf()` system call. We will measure BPF-DB's maximum throughput for serializable transactions with multiple statements. This scenario removes all overheads associated with the network layer and targets BPF-DB's performance directly. We also want to show BPF-DB's scaling characteristics for this workload,

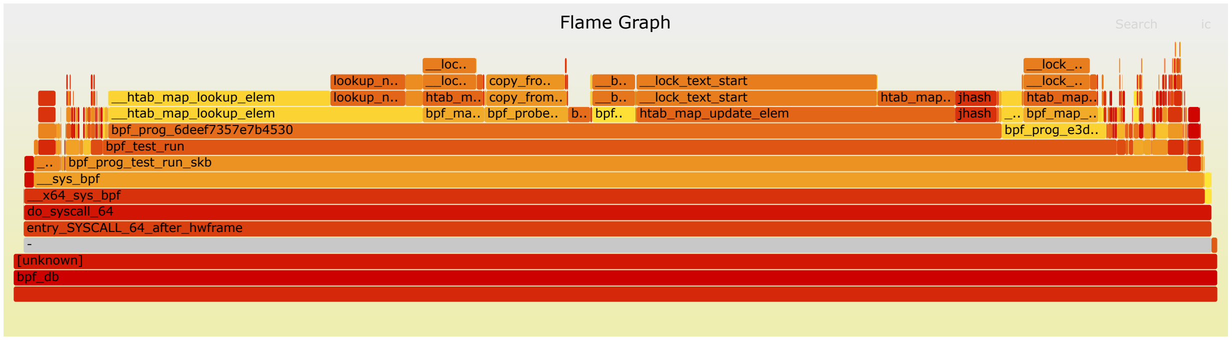


Figure 5.13: Voter Flame Graph – Aggregation of CPU call stacks for BPF-DB running the Voter workload.

so we sweep the number of CPU threads running in user-space that invoke the VOTE stored procedure.

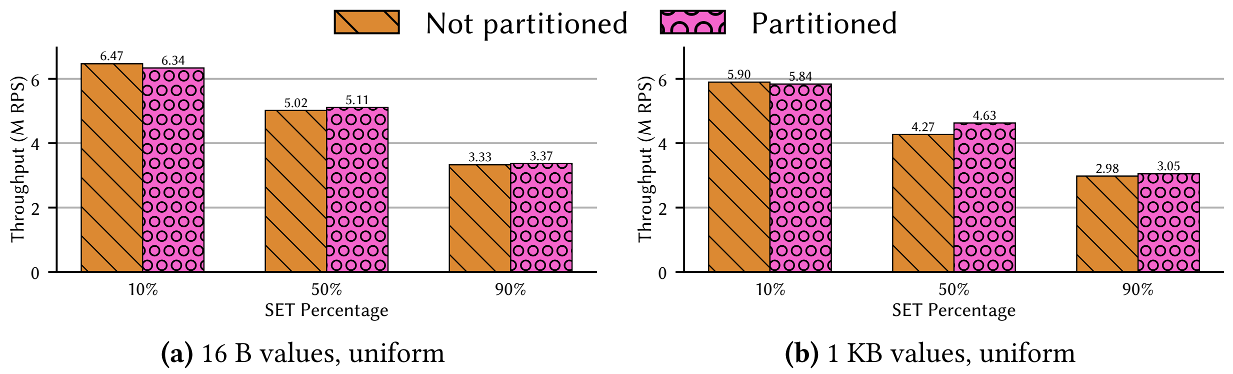
Figure 5.12 shows BPF-DB's maximum TPS for the Voter workload while increasing CPU threads. Like the 90% SET workload in Figure 5.9, Voter is write-intensive and plateaus when using around 15 of the 20 available CPU cores. However, the maximum throughput is lower than that experiment due to the extra BPF-DB operations, eBPF map lookups, and commit logic. We find that BPF-DB can maintain around 2 M TPS on a single 20-core CPU socket for multi-statement serializable transactions.

To further understand BPF-DB's performance characteristics, we use BCC's `profile` tool [4] to sample call stacks while BPF-DB runs the Voter workload. We then render a Flame Graph [19] to aggregate and analyze the resulting stack traces, as shown in Figure 5.13. BPF-DB spends most of its CPU time executing code not contained within its own eBPF programs. Instead, kernel-provided eBPF helper functions limit BPF-DB's performance. For example, 78.5% of the stack trace samples are in eBPF map operations (e.g., lookup, update), 4.6% are in the kernel's hash function (i.e., `jhash`), 4.8% are in the kernel's spin_lock implementation, and 9.5% are in helpers that copy data between kernel-owned memory. We leave optimizing BPF-DB further to reduce its reliance on kernel-provided helpers as future work.

5.5.7 Partitioned Storage

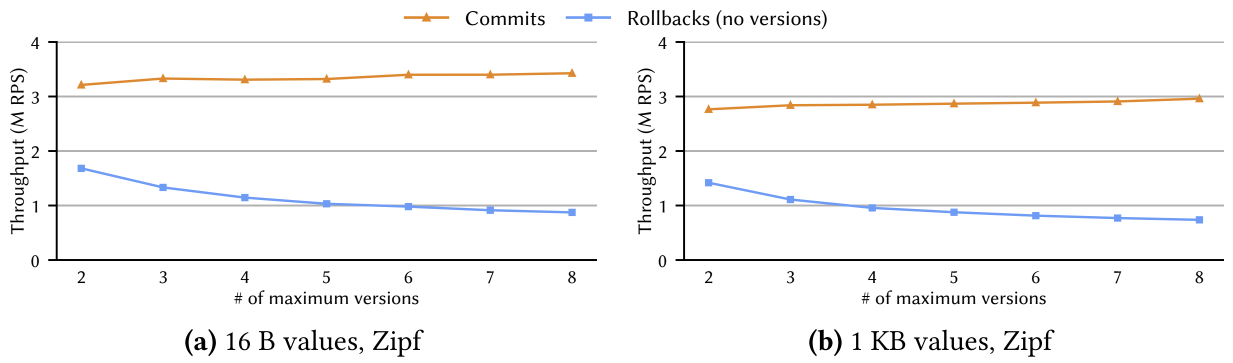
VoltDB applies horizontal partitioning to elide lock acquisition and reduce contention within its storage manager. Based on this design, we evaluate BPF-DB's performance when partitioning its key space across four shards. We replace the top-level index eBPF map with four indexes and similarly split the eBPF maps for value storage similarly. Before performing eBPF map lookups, we apply a fast hash function to determine the partition for a given key. We use the XXH64 [59] function due to its good performance and logic that passes the eBPF verifier. Faster hash functions either perform logic that is too complex to satisfy the verifier or rely on SIMD instructions (e.g., AVX2) that are unsupported by eBPF. We run BPF-DB locally using the KVS workload and generate keys from a uniform distribution to maximize throughput.

Figure 5.14 shows BPF-DB's throughout with varying SET ratios. Performance is slightly lower in the read-heavy workload (i.e.,10% SET) with 16 B and 1 KB values. Once SET commands



(a) 16 B values, uniform **(b)** 1 KB values, uniform

Figure 5.14: Partitioned Storage – Total requests per second running BPF-DB locally with and without key space partitioning. We measure three ratios of SET commands, and scenarios differ in value size (16 B or 1 KB) with a uniform key distribution to reduce contention and maximize throughput.



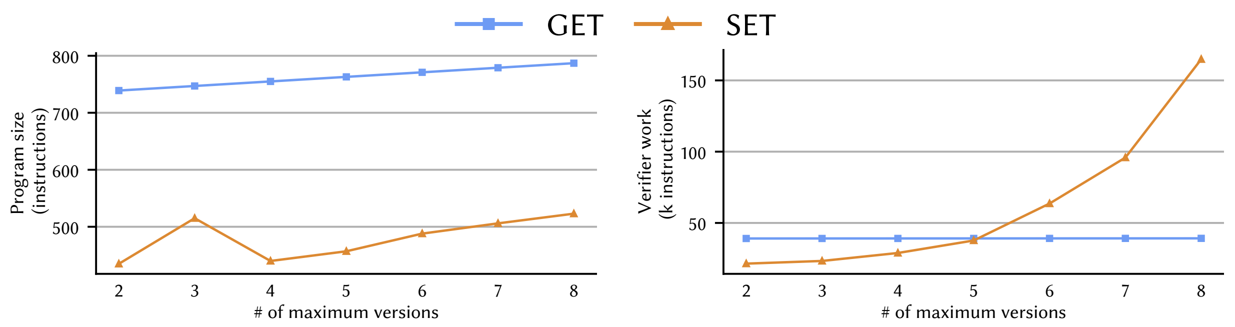
(a) 16 B values, Zipf **(b)** 1 KB values, Zipf

Figure 5.15: Scaling Maximum Versions – Total requests per second running BPF-DB locally while varying the maximum number of versions per database entry. Scenarios differ in value size (16 B or 1 KB), and we maximize version generation by using the 90% SET workload with skewed (i.e., Zipf) key distribution.

exceed 50%, partitioned storage yields a slight performance benefit. The increase in throughput never exceeds 8%, though, as seen in the scenario with 10% SET and 1 KB values. In practice, we find that key space partitioning does not yield a significant speedup because BPF-DB does not spend most of its execution time in code areas that benefit from the reduced map contention.

5.5.8 Scaling MVCC Versions

As detailed in Section 5.3.1, due to the compile-time verifier constraints, BPF-DB supports MVCC with a maximum number of versions for each database value. If one of BPF-DB's SET operations attempts to write to a value that has reached its maximum number of versions, it must attempt cooperative garbage collection or roll back, as described in Section 5.3.1. In this experiment, we measure the performance impact of changing this compile-time constant while running a workload that generates a large number of versions (90% SET) by using a skewed key



(a) eBPF program size **(b)** Verifier instructions

Figure 5.16: Maximum Versions and the Verifier – program size and verifier work for BPF-DB's GET and SET operators while varying the maximum number of versions per database entry.

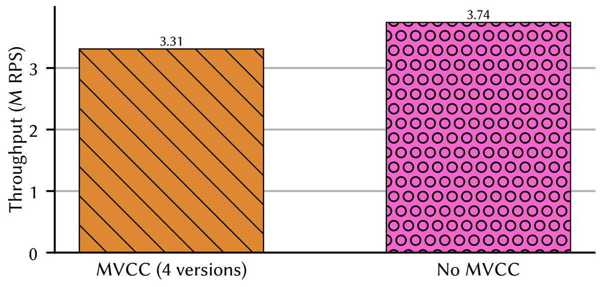
distribution (i.e., Zipf).

Figure 5.15 shows the performance impact on BPF-DB of changing the maximum number of versions per database entry. For 16 B values, BPF-DB with two versions per record supports 3.2 M RPS and 1.7 M roll backs per second from no available versions, while eight versions per key yields 3.4 M RPS and 0.9 M roll backs per second. While this change nearly halves the number of roll backs per second, there is only a 6.6% increase in overall throughput. At maximum throughput of over 3 M RPS on a small number of keys, increasing the number of keys does little to change how quickly BPF-DB exhausts its available versions. The results for 1 KB values are similar, demonstrating that maximum throughput is not significantly affected by changing the maximum number of versions.

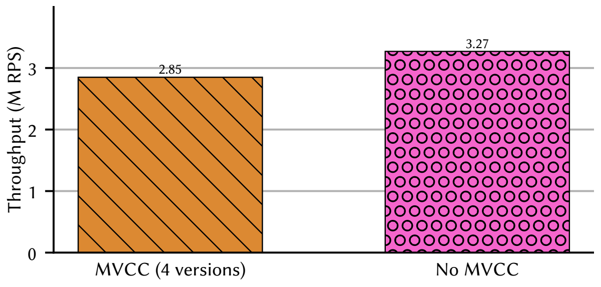
In addition to altering runtime behavior for BPF-DB, changing the number of maximum versions also affects both program size and the amount of work for the eBPF verifier. Recall that we limit the maximum number of versions per database entry because the verifier requires compile-time constants for all loops, and must explore all iterations and branches during verification. Consequently, changing this value affects the compiled code and the verifier's work.

Figure 5.16 shows the output of using `bpftool` [13] to get eBPF program size and the number of instructions processed by the verifier. We scale the maximum number of versions as in Figure 5.15. In Figure 5.16a GET's number of eBPF instructions scales linearly with the maximum number of versions. This behavior is due to the loop that compares timestamps to the entire version array to find the correct value to read. The optimizing compiler unrolls the loop to improve performance, thus increasing instruction count. In contrast, SET's number of instructions oscillates before scaling linearly. When the compiler generates eBPF code for SET with maximum number of versions set to three, it optimizes the C code differently and results in increased instruction count. This oscillation shows the difficulty in anticipating how compilers like Clang and GCC generate optimized eBPF bytecode.

GET contains more instructions than SET despite its less complex logic (e.g., GET does not perform GC). GET's high instruction count is due to the instructions that copy values out of the value map, which we implement due to the lack of `memcpy()`. SET does not require these instructions because the logic to copy a value into a map is handled by the eBPF helper that



(a) 16 B values, Zipf



(b) 1 KB values, Zipf

Figure 5.17: Single Version Storage – Total requests per second running BPF-DB locally with MVCC and without. Scenarios differ in value size (16 B or 1 KB), and we maximize version generation by using the `90% SET` workload with skewed (i.e., Zipf) key distribution.

inserts the value into the map. Looking up a value in a eBPF map only returns a pointer to the value, which GET in turn copies at its instruction cost into the BPF-DB Context object.

Figure 5.16b shows the number of instructions that the verifier processes when loading GET and SET operators with different numbers of maximum versions. GET scales linearly because the number of states that the verifier inspects increases as the limit on the loop that scans the version array increases. However, SET's verifier complexity increases exponentially as the maximum number of versions increases. SET contains more complex logic inside of the loops that scan the version array, particularly to perform cooperative GC. Though not shown in the figure, increasing the maximum number of versions past eight causes the verifier to reject the SET program. The results of this experiment demonstrate the non-obvious lack of correlation between eBPF program size and the amount of work the verifier performs to load a program.

5.5.9 Single Version Storage

We next evaluate a BPF-DB design that eliminates MVCC entirely. For single-statement operations like the KVS workload, maintaining multiple versions adds unnecessary overhead because a transaction does not need a consistent snapshot of multiple database entries. In this workload, `SET` and `GET` operations only access a single entry, and we can rely on Strict-2PL concurrency control to maintain correctness. Removing MVCC allows BPF-DB to elide version chain traversal, timestamp allocation, and cooperative garbage collection. In exchange, timestamp-based optimizations for read-only transactions become impossible, requiring these operations to acquire a read lock. To compensate, we augment BPF-DB's lock acquisition logic to retry up to 4096 times, which is within the bounds of the verifier. We evaluate this design choice with the same high contention workload as Section 5.5.8: `90% SET` with Zipf key distribution.

Figure 5.17 shows BPF-DB's maximum throughput with MVCC and without. The reduced complexity of BPF-DB's SET and GET operators enables higher overall throughput, despite GET now requiring a lock acquisition. This single-statement workload shows notable speedup from this change, increasing throughput by 13% and 15% for 16 B values and 1 KB values respectively. In exchange, multi-statement transactions with conflicting access sets would incur more roll backs. Developers may decide to make this trade-off if transactions are unlikely to conflict

Table 5.2: Maximum Versions and Latency (16 B) – Average operator latency (in microseconds) running BPF-DB locally with MVCC and without. This scenario uses 16 B values, and we maximize version generation by using the 90% SET workload with skewed (i.e., Zipf) key distribution.

Max versions	Min	p25	p50	p75	p99
1	0.49	0.96	1.12	1.39	4.42
2	0.52	0.91	1.03	1.20	12.80
3	0.41	0.85	1.01	1.23	16.84
4	0.40	0.85	1.00	1.30	13.98
5	0.40	0.84	0.98	1.24	11.36
6	0.40	0.82	0.96	1.19	12.96
7	0.40	0.80	0.94	1.16	19.41
8	0.40	0.80	0.94	1.14	15.61

Table 5.3: Maximum Versions and Latency (1 KB) – Average operator latency (in microseconds) running BPF-DB locally with MVCC and without. This scenario uses 1 KB values, and we maximize version generation by using the 90% SET workload with skewed (i.e., Zipf) key distribution.

Max versions	Min	p25	p50	p75	p99
1	0.68	1.46	1.7	2.0	5.32
2	0.65	1.32	1.55	1.83	46.21
3	0.53	1.25	1.51	1.86	45.04
4	0.52	1.22	1.49	1.87	44.03
5	0.52	1.18	1.44	1.83	39.56
6	0.52	1.16	1.41	1.78	40.89
7	0.52	1.14	1.39	1.75	37.32
8	0.52	1.13	1.38	1.72	37.25

because their access sets are either small or unlikely to overlap.

We conclude our analysis of MVCC versions by combining the approaches of Sections 5.5.8 and 5.5.9 to measure the impact on GET and SET operator latency. We fix the number of requests at 100 K RPS and measure operator latency in a high contention scenario that generates many versions (90% SET, Zipf keys). Tables 5.2 and 5.3 show the results of measuring operator latency while changing the maximum number of MVCC versions. Removing MVCC entirely (i.e., one maximum version) increases p25, median, and p75 latency in exchange for a reduction in p99 latency. Note that this scenario removes all BPF-DB logic related to garbage collection as well, which contributes to the large reduction in tail latency. Developers may wish to specialize BPF-DB's MVCC design to their application and workload, and we discuss this possibility further in Section 7.2.

5.6 Conclusion

We presented BPF-DB, an embedded DBMS that offers serializable, ACID-compliant transactions in kernel-space for user-bypass applications. BPF-DB decomposes its operators into individual eBPF programs that developers can use to build custom procedures. These procedures interleave custom application logic with BPF-DB operators, held together in sequences through tail calls. We demonstrated applications built on BPF-DB and compared them against state-of-the-art counterparts. Our results showed that embedding a DBMS in kernel-space is not only feasible but scalable. Future eBPF applications can use BPF-DB for robust, safe, and high-performance DBMS semantics.

Chapter 6

Related Work

6.1 Observability and Training Data Collection

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 [57] and Amazon's RDS Performance Insights [1], 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 TScout uses for query tracing, debugging, and performance analysis [43]. One could use these probes to build markers that communicate with TScout'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 [33]. MongoDB has support for creating USDT probes, but this functionality appears unused [32].

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 built-ins (e.g., `clock_gettime`) or language-level high-resolution clocks (e.g., `std::chrono`). Some DBMSs such as PostgreSQL [44] and MySQL [34] capture hardware metrics using tools like `rusage`. As we describe in Section 3.3, 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 [67, 128]. 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 [146] is an adaptive eBPF-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 eBPF 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 eBPF), or eBPF. For example, Slack's Observability Data Management System [134] analyzes metrics to diagnose and resolve disruptions across multiple services. Seer [111], MicroRCA [210], and LOUD [157] 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 [195] identifies a salient metrics and associated dependencies, but it analyzes offline. Pythia [69] is a proposed always-on automated instrumentation framework.

In comparison, TScout is an adaptive *internal* approach to dynamic metrics collection. Furthermore, we designed TScout 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 [202] and CDBTune [147] 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 [155] extracts execution times and execution plans from PostgreSQL's `EXPLAIN ANALYZE` to train models that predict the execution time of queries. ModelBot2 [152] exercises offline runners to train OUs-level models similar to TScout. DBSeer [166] extracts SQL query logs to cluster queries based on their runtime behavior. DeepSketch [136] executes training queries to obtain cardinalities. GPredictor [215] extracts query plans and execution performance from the DBMS to train a graph-embedding performance model. UDO [205] 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 [148]. Other learned system proposals like SageDB are not prescriptive in generating training data in online settings [138].

6.2 Network Functions and System Calls

We now discuss existing research on eBPF observability, programmable networks, and techniques similar to user-bypass for pushing DBMS-specific logic into kernel-space.

Network Observability Proxies: Envoy is an L3/L4 proxy with some L7 capabilities for DBMS observability [18]. Endpoint support is limited to HTTP/2 and gRPC, along with "sniffing filters" for DBMSs like MongoDB and PostgreSQL. For PostgreSQL, Envoy can accumulate counters for queries (i.e., `INSERT`, `UPDATE`) and errors in the query stream but does not perform connection pooling or workload replication. Because it is not a PostgreSQL endpoint, it cannot process encrypted traffic.

Network Function Virtualization (NFV): Custom behavior in the network stack and middleboxes has increased in recent years, particularly with the rise of SmartNICs among cloud vendors [105]. Programming languages like P4 [78] and Domino [187] allow more expressibility in the network layer, but developers apply them to flow routing using packet headers rather than L7 logic.

Recent works increasingly rely on the benefits of eBPF to push logic into kernel-space and even to hardware layers [102, 163, 180, 203, 212]. These eBPF efforts target L3/L4 applications like flow routing, intrusion detection, and denial-of-service mitigation and rely on a class of eBPF programs called XDP which are more restrictive than Tigger's sockmap programs. BMC also uses XDP to apply user-bypass to memcached, enabling kernel threads to handle the key-value store's simple operations over UDP [113].

TCP Splicing: Network researchers explored kernel-space connection pooling at the transport layer (e.g., TCP) in kernel-space with a technique called *TCP splicing* [154]. The goal was similar to user-bypass: avoid copying buffers to user-space that will ultimately be forwarded to another socket. Prior work evaluated these techniques on web server and firewall workloads but required invasive changes to the OS to implement. For example, IBM AIX developers made changes directly to the kernel source code [179], a Linux implementation required an out-of-tree kernel module [86], and other approaches used custom designs like Scout OS and Exokernel [112, 189]. The Linux effort was unique in applying L7 logic to provide URL-aware forwarding [86]. This logic could be duplicated using user-bypass to offer a safe implementation without risking kernel panics.

One outcome of these efforts is the `splice()` system call in Linux that promises zero-copy forwarding between file descriptors without duplicating the data in user-space. However, this approach still incurs system call overhead and requires waking up an user-space application to coordinate forwarding. Lastly, it is impossible to peek at the data in flight, so DPI to apply DBMS logic requires copying the data to user-space.

6.3 OS Extensibility and Unikernels

OS Extensibility: For decades, database researchers discussed the shortcomings of using OS services to design DBMSs [192]. In response, the research community proposed methods to extend OS behavior to better suit user-space applications [73, 181, 182]. However, these approaches lacked a standard API and strong OS support like eBPF to extend behavior.

More recently, researchers presented the ExtOS Linux prototype designed to reduce data movement in the software stack [70]. For example, they proposed adding programmability to the `read()` system call to push database filters into the OS via a kernel module. Their initial results showed promising speedup, and they proposed extensions to eBPF which was not yet suitable for their task in 2019.

Unikernels: These applications represent kernel-bypass pushed to an extreme, where applications are compiled with a library OS to yield an application-specific machine [104, 129, 139, 204]. Unikernels remain an active and promising research area, but the industry has been slow to adopt this approach. Designing unikernel applications requires similar expertise and has the same challenges as kernel-bypass methods. While more limited in capabilities, eBPF is easily deployed in cloud native environments and companies are rapidly adopting it, as described in Section 4.2.2. Recent work argues that cloud virtualization and their hypervisor abstractions provide new opportunities for DBMSs designed as unikernels [141].

eXpress Resubmission Path (XRP): Recent work applies user-bypass to the Linux kernel's storage stack [214]. For example, B+ tree lookups read multiple disk pages before finding the destination leaf node. The DBMS copies the entire page from kernel-space to user-space via system calls, only to find a pointer to the correct child node and repeat the process. Similar to the network stack, repeatedly accessing the storage stack imposes overhead from system calls and memory copies.

XRP pushes DBMS logic into the NVMe driver via eBPF. With XRP, the DBMS performs B+ tree node traversal in kernel-space by resubmitting multiple NVMe operations. XRP's use of user-bypass reduced the amount of data copied to user-space and the number of repeated system calls. They demonstrated the latency and throughput benefits of XRP against kernel-bypass using DPDK and asynchronous I/O.

Kernel-Bypass: As discussed in Section 4.2.1, kernel-bypass is a technique to bring packets directly to user-space by eliding the Linux network stack. We created a version of PgBouncer that uses kernel-bypass via F-Stack [21] to evaluate the benefits and challenges of kernel-bypass, especially compared to user-bypass. But integrating F-Stack dependencies and creating a productive development environment proved challenging. For example, DPDK requires exclusive control of a dedicated NIC, making debugging difficult with standard networking tools. Despite communicating with F-Stack engineers, we did not achieve acceptable performance with DPDK-enhanced PgBouncer; it is 10–100× slower than unmodified PgBouncer. Our engineering problems match the recent effort to create a DPDK version of Open vSwitch [200], which concludes by recommending AF_XDP instead.

To our knowledge, the only database vendor making significant use of kernel-bypass is Yellowbrick, which does not use DPDK but instead wrote their own network and storage device drivers [120]. Although ScyllaDB promotes DPDK compatibility in their Seastar framework, they do not deploy DPDK in production [119]. In general, deploying DPDK applications into production is difficult due to API/ABI instability, which forces developers to choose between features and fixes of new releases and code stability of LTS releases [161]. eBPF experienced similar API growing pains over the last ten years but is proving to be a less burdensome development environment than kernel-bypass.

Kernel-embedded DBMSs: McObject sells eXtremeDB as a user-space embedded DBMS that also offers a kernel mode [115]. In kernel mode, developers like their user-space applications with eXtremeDB's API library. This library communicates with a kernel module that contains a proxy, the DBMS, and runtimes for network and storage drivers. As discussed in Section 1.3, unsigned kernel modules require users to disable Secure Boot—a mitigation technique against firmware-based attacks. BPF-DB does not require a kernel module and runs via eBPF at the expense of a more restricted kernel API.

More recently, researchers introduced DINT, a distributed, key-value transaction service that runs in kernel-space via eBPF [216]. DINT applies user-bypass similarly to Tigger, whereby common operations run in kernel-space while less common operations run in user-space. For DINT, this approach stores small value sizes in fixed-sized eBPF maps, while larger value sizes that require `malloc()` reside in user-space. Similar to BPF-DB, DINT use a hybrid user-space and kernel-space logging and recovery approach with user-space components responsible for log replay. DINT, like BMC, uses XDP eBPF programs and attachment points, which limits its application to UDP protocols. In their evaluation, DINT achieves 2.6× higher throughput than a kernel-bypass baseline.

Chapter 7

Future Work

7.1 User-Bypass DBMS Proxy

We show a technique called user-bypass in the context of DBMS proxies. We believe this opens a number of opportunities for both DBMS proxies and aspects of DBMS design that could benefit from application logic in kernel-space.

Proxy Features: Tigger implements the most common features needed in DBMS proxies to demonstrate the benefits of user-bypass, but there are more capabilities to evaluate. For example, Vitess [188] supports SQL-aware logic to perform query rewriting and sharding for MySQL. Also, techniques that modify the query stream, like transaction reordering and in-network computation [97, 131], may be feasible in a DBMS proxy using user-bypass.

Pgpool-II and ProxySQL provide a query result cache to return result sets for frequently submitted queries. These caches rely on simple time-to-live (TTL) policies with string-matching of queries rather than SQL-aware eviction methods. For example, a cached result will not automatically be evicted if a query changes its value in the DBMS. For this reason, these caches are error-prone but could be used to reduce the burden of queries automatically submitted by application frameworks like "`SELECT 1`". More sophisticated solutions (e.g., Heimdall [23]) support transparent query caching with automatic invalidation. These techniques could benefit from user-bypass if their approaches satisfy the eBPF verifier. For example, a full SQL parser is outside the verifier's scope, and eBPF maps may not be large enough for a robust query result cache.

Asynchronous I/O: Linux provides an asynchronous I/O interface called *io_uring*. In late 2021, a Linux developer posted an experimental patch for io_uring that offers support for network sockets [89]. The new kernel feature reads and writes via shared buffers with user-space, which reduces the number of data copies. A DBMS proxy would still need to copy data between these buffers, and all coordination would still be performed via user-space application waiting on `epoll()`. Nonetheless, we remain interested in evaluating io_uring's impact on DBMS proxy design.

Hardware Acceleration: The improved capabilities of SmartNICs and FPGAs and their proliferation in public cloud settings create new opportunities for accelerating applications via hardware [105]. For example, Mellanox and Netflix added support for offloading kTLS to NICs [183]. This development may enable performing L7 logic like DBMS proxies at lower levels of the network stack. Hardware TLS is not broadly available, but future devices may enable user-bypass techniques at the hardware layer.

7.2 User-Bypass DBMS Architecture

We demonstrate a design for an embedded DBMS that runs in kernel-space to enable rich user-bypass applications. We believe there are more opportunities to continue this work in varying directions.

Code Generation: Currently, applications developers wanting to use BPF-DB must write their code in C++ and require significant eBPF knowledge. However, high-level scripting languages that compile down to eBPF (e.g., bpftrace [14]) lower the barrier to entry. Prior work demonstrated the ability to code-generate eBPF programs from templates, which were then specialized to DBMS source code [80]. A domain-specific language (DSL) for eBPF programs that includes BPF-DB semantics would make it easier for developers to write stored procedures like `VOTE` shown in Section 5.4. Generating these stored procedures would also enable BPF-DB specialization at code generation time. For example, BPF-DB could elide eBPF maps for value sizes that would not be used or tailor the number of MVCC versions.

Fuzzy Checkpointing: As described in Section 5.3.3, BPF-DB supports write-ahead logging for durability. BPF-DB's continuation-passing style presents new opportunities to implement different checkpointing techniques. For example, the Linux kernel can atomically swap eBPF programs at runtime. One way to implement blocking checkpointing would be to swap out BPF-DB's SET operators with eBPF programs that just call the error continuation while checkpointing is in progress. Once the checkpoint is complete, the SET operators could be swapped to their original version and applications could resume writing to the database. More sophisticated fuzzy checkpointing could build on this technique by temporarily swapping in SET operators that write to an alternate set of eBPF maps while a checkpoint is in progress. GET operators would also need to be swapped to be able to look in the right eBPF map for their values. After the checkpoint is complete, the original operators could be swapped back.

Kernel Patches: While this work demonstrates the feasbility and benefits of an embedded DBMS as eBPF programs, it also explores the limitations imposed by the verifier. If BPF-DB were implemented as a new eBPF map type in the Linux kernel (e.g., `BPF_MAP_TYPE_TXN_HASH`) with corresponding eBPF helper functions, then some of this limitations would be relaxed. For example, the ability to hold multiple latches would enable metadata (e.g., locks, versions, timestamps) that are currently packed into fixed-size eBPF map values to be located in different memory locations while still being updated atomically. Similarly, transactions could call the kernel's RCU semantics directly which would reduce the current synchronization overheads.

Chapter 8

Concluding Remarks

In this dissertation, we presented user-bypass as a technique to improve the performance of frequent DBMS operations. This design exists in the content of current systems' "all or nothing" approach to resource management and the OS. Current DBMSs either acquire and release resources through OS system calls or wrest control of system resources through kernel-bypass. We showed how user-bypass takes an alternative approach, whereby DBMSs can extend OS behavior to suit their performance requirements and capabilities.

We detailed how user-bypass builds upon a Linux technology called eBPF that enables OS extensibility in a safe and performant manner. Chapter 3 demonstrated how eBPF programs attached to a DBMS running in user-space can provide behavior akin to a new system call. These new kernel operations reduced the number of round-trips for the DBMS to kernel-space, increasing query performance while collecting system metrics. Then, in Chapter 4, we used user-bypass to build a high-performance DBMS proxy that performs connection pooling and workload mirroring in kernel-space. This design showed the benefits of embedding DBMS-specific logic into the networking stack, resulting in a proxy that required fewer resources than state-of-the-art contemporaries. Lastly, in Chapter 5, we designed an embedded DBMS that runs serializable, ACID-compliant transactions in kernel-space. We explored the challenges of building a DBMS for this environment and compared our approach with traditional DBMS design. We implemented sample applications around this embedded DBMS and demonstrated the promise of this design for future applications.

Taken as a whole, the work presented in this dissertation demonstrates the performance and robustness benefits of applying user-bypass to design DBMS software in concert with OS extensibility.

Bibliography

[1] Amazon RDS Performance Insights. https://aws.amazon.com/rds/performance-insights/, . 6.1
[2] App Scaling - AWS Application Auto Scaling - AWS. https://aws.amazon.com/autoscaling/, . 4.1.1
[3] ENA Linux Driver Best Practices and Performance Optimization Guide. https://github.com/amzn/amzn-drivers/blob/master/kernel/linux/ena/ENA_Linux_Best_Practices.rst, . 4.5
[4] bcc/tools/profile.py at master - iovisor/bcc - GitHub. https://github.com/iovisor/bcc/blob/master/tools/profile.py. 5.5.6
[5] GitHub - cmu-db/benchbase: Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase. 3.5.1, 4.5.1
[6] Moonbounce bootkit shows significantly enhanced elusiveness and persistence | Kaspersky. https://usa.kaspersky.com/about/press-releases/2022_kaspersky-uncovers-third-known-firmware-bootkit. 1.3
[7] GitHub - cilium/cilium: eBPF-based Networking, Security, and Observability. https://github.com/cilium/cilium, . 2.2, 2.2.3, 4.2.2
[8] eBPF Instruction Set — The Linux Kernel documentation. https://www.kernel.org/doc/html/v5.17/bpf/instruction-set.html, . 5.1.2
[9] GitHub - facebookincubator/katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran, . 2.2, 2.2.3, 4.2.2
[10] Add bpf_loop helper. https://lwn.net/Articles/877170/, . 2.2.2
[11] bpf: Add bpf_rcu_read_lock() support. https://lwn.net/Articles/915273/, . 5.1.2
[12] GitHub - microsoft/ebpf-for-windows: eBPF implementation that runs on top of Windows. https://github.com/microsoft/ebpf-for-windows, . 2.2, 4.2.2
[13] Ubuntu Manpage: bpftool-prog - tool for inspection and simple manipulation of eBPF progs. https://manpages.ubuntu.com/manpages/focal/en/man8/bpftool-prog.8.html, . 5.5.8
[14] GitHub - bpftrace/bpftrace: High-level tracing language for Linux eBPF. https://github.com/bpftrace/bpftrace, . 2.2.3, 7.2
[15] EXPLAIN ANALYZE | CockroachDB Docs. https://www.cockroachlabs.com/docs/stable/explain-analyze.html. 3.3.2
[16] Intel Data Plane Development Kit (DPDK). https://www.dpdk.org. 2.1, 4.2.1
[17] Amazon EC2 C6i Instances - Amazon Web Services. https://aws.amazon.com/ec2/

instance-types/c6i/. 4.5

[18] Postgres — envoy 1.24.0-dev-fbcf42 documentation. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_protocols/postgres. 6.2

[19] GitHub - brendangregg/FlameGraph: Stack trace visualizer. https://github.com/brendangregg/FlameGraph. 5.5.6

[20] folly/tracing: Utility for User-level Statically Defined Tracing. https://github.com/facebook/folly/tree/master/folly/tracing. 3.2.1

[21] F-Stack | High Performance Network Framework Based On DPDK. http://www.f-stack.org. 2.1, 4.2.1, 6.3

[22] Github - haproxy/haproxy: Haproxy load balancer's development branch (mirror of git.haproxy.org). https://github.com/haproxy/haproxy/. 4.1.1, 4.1.3

[23] Home - Heimdall Data. https://www.heimdalldata.com. 7.1

[24] GitHub - brettwooldridge/HikariCP: A solid, high-performance, JDBC connection pool at last. https://github.com/brettwooldridge/HikariCP. 4.1.1

[25] GitHub - iovisor/bcc: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc. 2.2.3

[26] GitHub - jemalloc/jemalloc. https://github.com/jemalloc/jemalloc. 1.1

[27] sthima/libstapsdt: Create Systemtap's USDT probes at runtime. https://github.com/sthima/libstapsdt. 3.2.1

[28] Re: [PATCH 09/13] aio: add support for async openat() [LWN.net]. https://lwn.net/Articles/671657/, . 1.1

[29] kernel/git/torvalds/linux.git - Linux kernel source tree. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=654443e20dfc0617231f28a07c96a979ee1a0239, . 3.2.1

[30] Github - mariadb-corporation/maxscale: An intelligent database proxy. https://github.com/mariadb-corporation/MaxScale. 4

[31] RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark. 5.5.1

[32] mongodb/mongo: usdt.h. https://github.com/mongodb/mongo/blob/master/src/mongo/platform/usdt.h. 6.1

[33] 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, . 6.1

[34] mysql/mysql-server: sql_profile.cc. https://github.com/mysql/mysql-server/blob/8.0/sql/sql_profile.cc, . 6.1

[35] nginx. https://nginx.org/en/. 4.1.1, 4.1.3

[36] NoisePage – Database Management System Project. https://noise.page. 3, 3.5

[37] Github - yandex/odyssey: Scalable postgresql connection pooler. https://github.com/yandex/odyssey. 4.1.3

[38] Self-driving database | autonomous database oracle 19c. https://oracle.com/database/autonomous-database/. 3

[39] PgBouncer - lightweight connection pooler for PostgreSQL. https://www.pgbouncer.org. 4, 4.3.1

[40] pgpool Wiki. https://www.pgpool.net/. 4, 4.5.4

[41] pgxpool package - github.com/jackc/pgx/v4/pgxpool - Go Packages. https://pkg.go.dev/github.com/jackc/pgx/v4/pgxpool. 4.1.1

[42] PostgreSQL: Documentation: 14: EXPLAIN. https://www.postgresql.org/docs/current/sql-explain.html, . 3.1.3

[43] postgres/postgres: probes.d. https://github.com/postgres/postgres/blob/master/src/backend/utils/probes.d, . 6.1

[44] postgres/postgres: getrusage.c. https://github.com/postgres/postgres/blob/master/src/port/getrusage.c, . 6.1

[45] Proxysql - a high performance open source mysql proxy. https://proxysql.com. 4

[46] Review Checklist for RCU Patches — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/RCU/checklist.html. 5.5.3

[47] Amazon RDS Proxy | Highly Available Database Proxy | Amazon Web Services. https://aws.amazon.com/rds/proxy/. 4

[48] Seastar. https://seastar.io, . 2.1, 4, 4.2.1

[49] Networking - seastar. https://seastar.io/networking/, . 2.1

[50] SecureBoot - Debian Wiki. https://wiki.debian.org/SecureBoot, . 1.3

[51] mjg59 | Responsible stewardship of the UEFI secure boot ecosystem. https://mjg59.dreamwidth.org/60248.html, . 1.3

[52] socket(2) - Linux manual page. https://www.man7.org/linux/man-pages/man2/socket.2.html. 2.1, 4.2.1

[53] GitHub - sysstat/sysstat: Performance monitoring tools for Linux. https://github.com/sysstat/sysstat. 4.5.6

[54] tc(8) - Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html. 2.1, 4.2.1

[55] GitHub - google/tcmalloc. https://github.com/google/tcmalloc. 1.1

[56] Infrastructure – Vercel. https://vercel.com/features/infrastructure. 4.1.1

[57] Database Performance Monitor (DPM) | SolarWinds. https://vividcortex.com/. 6.1

[58] GitHub - VoltDB/voltdb: Volt Active Data. https://github.com/VoltDB/voltdb. 5.5.1

[59] GitHub - Cyan4973/xxHash: Extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash. 5.5.7

[60] Introduction to the Storage Performance Development Kit (SPDK). https://www.intel.com/content/www/us/en/developer/articles/tool/introduction-to-the-storage-performance-development-kit-spdk.html, 2015. 2.1

[61] PHP: Connections and Connection management - Manual. https://www.php.net/manual/en/pdo.connections.php, November 2021. 4.1.2

[62] Databases | Django documentation | Django. https://docs.djangoproject.com/en/4.1/ref/databases/, August 2022. 4.1.2

[63] David Ahern. Can the linux networking stack be used with very high speed applications? https://lpc.events/event/16/contributions/1345/, September 2022. 2.1, 4.2.1

[64] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. The evolution of leanstore. In

BTW, volume P-331 of *LNI*, pages 259–281. Gesellschaft für Informatik e.V., 2023. 1.1

[65] Mohammad Alomari, Michael J. Cahill, Alan D. Fekete, and Uwe Röhm. The cost of serializability on platforms that use snapshot isolation. In *ICDE*, pages 576–585. IEEE Computer Society, 2008. 3.5.1, 4.5.1

[66] Chris Andrews. chrisa/libusdt: Create DTrace probes at runtime. `https://github.com/chrisa/libusdt`. 3.2.1

[67] Kiam Heong Ang, G. Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13(4):559–576, 2005. 6.1

[68] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. Amazon redshift reinvented. In *SIGMOD Conference*, pages 2205–2217. ACM, 2022. 1

[69] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *SoCC*, pages 165–170, 2019. 6.1

[70] Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. Extos: Data-centric extensible OS. In *APSys*, pages 31–39. ACM, 2019. 6.3

[71] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*, chapter 5, pages 160–161. Addison-Wesley, 1987. 5.3.2

[72] Brian N. Bershad, Craig Chambers, Susan J. Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN - an extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop*, pages 68–71. ACM, 1994. 1.3

[73] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, pages 267–284. ACM, 1995. 1.3, 6.3

[74] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98. USENIX Association, 1994. 5.3.1

[75] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources. In *USENIX Annual Technical Conference, General Track*, pages 15–33. USENIX, 2001. 5.3.1

[76] Daniel Borkmann and John Fastabend. Combining ktls and bpf for introspection and policy enforcement. November 2018. 2.1, 4.2.1, 4.3.1

[77] Matt Bornstein, Jennifer Li, , and Martin Casado. Emerging Architectures for Modern Data Infrastructure. `https://a16z.com/emerging-architectures-for-modern-data-infrastructure/`, October 2020. 1

[78] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95,

2014. 6.2

[79] Randal E. Bryant and David R. O'Hallaron. *Computer systems - a programmers perspective*. Pearson Education, 2003. 1

[80] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. Tastes great! less filling! high performance and accurate training data collection for self-driving database management systems. In *SIGMOD Conference*, pages 617–630. ACM, 2022. 1.4, 7.2

[81] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A database proxy that bounces with user-bypass. *Proc. VLDB Endow.*, 16(11):3335–3348, 2023. 1.4

[82] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD Conference*, pages 729–738. ACM, 2008. 3.5.1, 4.5.1

[83] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *SIGCOMM*, pages 65–77. ACM, 2021. 1.1, 2.1, 4.1.3, 4.2.1

[84] Brian E. Carpenter and Scott W. Brim. Middleboxes: Taxonomy and issues. *RFC*, 3234: 1–27, 2002. 4

[85] Elizabeth Christensen. Postgres at scale: Running multiple pgbouncers. https://www.crunchydata.com/blog/postgres-at-scale-running-multiple-pgbouncers, November 2022. 4.1.3

[86] Ariel Cohen, Sampath Rangarajan, and J. Hamilton Slye. On the performance of TCP splicing for url-aware redirection. In *USENIX Symposium on Internet Technologies and Systems*. USENIX, 1999. 6.2

[87] 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. The mixed workload ch-benchmark. In *DBTest*, 2011. 3.5.1

[88] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154. ACM, 2010. 3.5.1, 4.1.1, 4.5.1, 5.5.1

[89] Jonathan Corbet. Zero-copy Network Transmission with io_uring. https://lwn.net/Articles/879724/, 2021. 7.1

[90] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are you sure you want to use MMAP in your database management system? In *CIDR*. www.cidrdb.org, 2022. 1.1

[91] Mark A Cusack, John Adamson, Mark Brinicombe, Neil A. Carson, Thomas Kejser, Jim Peterson, Arvind Vasudev, Kurt Westerfeld, and Robert Wipfel. Yellowbrick: An elastic data warehouse on kubernetes. In *CIDR*. www.cidrdb.org, 2024. 2.1

[92] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *SIGMOD Conference*, pages 215–226. ACM, 2016. 1

[93] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. Automatically indexing millions of databases in microsoft azure sql database. In *SIGMOD '19*, pages 666–679, 2019. 3

[94] J.D. Day and H. Zimmermann. The osi reference model. *Proceedings of the IEEE*, 71(12): 1334–1340, 1983. doi: 10.1109/PROC.1983.12775. 4

[95] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *SIGMOD Conference*, pages 79–94. ACM, 2017. 2.2.1

[96] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, 2013. 3.5.1, 4.5.1

[97] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proc. VLDB Endow.*, 12(2):169–182, 2018. 7.1

[98] Siying Dong, Shiva Shankar P., Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. Disaggregating rocksdb: A production experience. *Proc. ACM Manag. Data*, 1(2):192:1–192:24, 2023. 5.1.1

[99] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009. 3, 3.1.4

[100] Dominik Durner, Viktor Leis, and Thomas Neumann. Experimental study of memory allocation for high-performance query processing. In *ADMS@VLDB*, pages 1–9, 2019. 1.1

[101] Wolfgang Effelsberg and Theo Härder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984. 1.1

[102] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. Partition-aware packet steering using XDP and ebpf for improving application-level parallelism. In *ENCP@CoNEXT*, pages 27–33. ACM, 2019. 6.2

[103] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The exokernel approach to operating system extensibility (panel statement). In *OSDI*, page 198. USENIX Association, 1994. 1.3

[104] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, pages 251–266. ACM, 1995. 1.3, 6.3

[105] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In

NSDI, pages 51–66. USENIX Association, 2018. 6.2, 7.1

[106] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, April 2007. 3.1.2

[107] Michael J. Franklin. Concurrency control and recovery. In *The Computer Science and Engineering Handbook*, pages 1058–1077. CRC Press, 1997. 1.1

[108] Andres Freund. Measuring the Memory Overhead of a Postgres Connection. https://blog.anarazel.de/2020/10/07/measuring-the-memory-overhead-of-a-postgres-connection/, October 2020. 4.1.1

[109] Andres Freund. Analyzing the Limits of Connection Scalability in Postgres. https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/analyzing-the-limits-of-connection-scalability-in-postgres/ba-p/1757266, October 2020. 4.1.1

[110] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. Sqlite: Past, present, and future. *Proc. VLDB Endow.*, 15(12):3535–3547, 2022. 5.1.1

[111] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. ASPLOS '19, pages 19–33, 2019. 6.1

[112] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Héctor M. Briceño, Russell Hunt, and Thomas Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.*, 20(1):49–83, 2002. 1.3, 6.2

[113] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: accelerating memcached using safe in-kernel caching and pre-stack processing. In *NSDI*, pages 487–501. USENIX Association, 2021. 4, 6.2

[114] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet L. Wiener. Fast database restarts at facebook. In *SIGMOD Conference*, pages 541–549. ACM, 2014. 5.2

[115] Andrei Gorine and Alexander Krivolapov. A kernel mode database system for high performance applications. Technical report, McObject, 2020. 6.3

[116] Vibby Gottemukkala and Tobin J. Lehman. Locking and latching in a memory-resident database system. In *VLDB*, pages 533–544. Morgan Kaufmann, 1992. 5.3.1, 5.3.2

[117] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994. 3.2.2, 3.4.2

[118] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 1st edition, 2019. 2.2.3, 3

[119] CMU Database Group. ScyllaDB: No-Compromise Performance (Avi Kivity). https://youtu.be/0S6i9BmuF8U?t=2586, 2020. 1.2, 2.1, 6.3

[120] CMU Database Group. Yellowbrick: An Elastic Data Warehouse on Kubernetes (Mark Cusack). https://youtu.be/uHMcVDNkHi4, 2022. 1.2, 2.1, 6.3

[121] Dirk Grunwald and Benjamin G. Zorn. Customalloc: Efficient synthesized memory allo-

cators. *Softw. Pract. Exp.*, 23(8):851–869, 1993. 5.3.1

[122] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD Conference*, pages 1917–1923. ACM, 2015. 1

[123] Gabriel Haas and Viktor Leis. What modern nvme storage can do, and how to exploit it: High-performance I/O for high-performance storage engines. *Proc. VLDB Endow.*, 16(9): 2090–2102, 2023. 1.1

[124] Gabriel Haas, Michael Haubenschild, and Viktor Leis. Exploiting directly-attached nvme arrays in DBMS. In *CIDR*. www.cidrdb.org, 2020. 1.1

[125] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. Rethinking logging, checkpoints, and recovery for high-performance storage engines. In *SIGMOD Conference*, pages 877–892. ACM, 2020. 5.3.3

[126] Yongjun He, Jiacheng Lu, and Tianzheng Wang. Corobase: Coroutine-oriented main-memory database engine. *Proc. VLDB Endow.*, 14(3):431–444, 2020. 1.1

[127] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *CoNEXT*, pages 54–66. ACM, 2018. 2.1, 4.2.1

[128] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *OSDI '18*, pages 1–16, October 2018. 6.1

[129] Takayuki Imada. Mirageos unikernel with network acceleration for iot cloud environments. In *ICCBDC*, pages 1–5. ACM, 2018. 6.3

[130] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *NSDI*, pages 489–502. USENIX Association, 2014. 1.2, 2.1, 4.2.1

[131] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-network support for transaction triaging. *Proc. VLDB Endow.*, 14(9):1626–1639, 2021. 7.1

[132] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP*, pages 52–65. ACM, 1997. 1.3

[133] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008. 5.5.1

[134] Suman Karumuri, Franco Solleza, Stan Zdonik, and Nesime Tatbul. Towards observability data management at scale. *SIGMOD Rec.*, 49(4):18–23, 2020. 6.1

[135] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE Computer Society, 2011. 5.3.3

[136] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. Estimating cardinalities with deep sketches. In *SIGMOD*, pages 1937–1940, 2019. 6.1

[137] Lev Kokotov. Scaling PostgresML to 1 Million Requests per Second. https://postgresml.org/blog/scaling-postgresml-to-one-million-requests-per-second, 2022. 4.4.2

[138] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019. 6.1

[139] Simon Kuenzer, Vlad-Andrei Badoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Raducanu, Cristian Banu, Laurent Mathy, Razvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: fast, specialized unikernels the easy way. In *EuroSys*, pages 376–394. ACM, 2021. 6.3

[140] Kahina Lazri, Antoine Blin, Julien Sopena, and Gilles Muller. Toward an in-kernel high performance key-value store implementation. In *SRDS*, page 268. IEEE, 2019. 4

[141] Viktor Leis and Christian Dietrich. Cloud-native database systems and unikernels: Reimagining os abstractions for modern hardware. *Proc. VLDB Endow.*, 17, 2024. 6.3

[142] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD Conference*, pages 743–754. ACM, 2014. 1.1

[143] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *ICDE*, pages 185–196. IEEE Computer Society, 2018. 1.1

[144] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. In *SIGMOD Conference*. ACM, 2023. 1.1

[145] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. Multi-version range concurrency control in deuteronomy. *Proc. VLDB Endow.*, 8(13): 2146–2157, 2015. 5.3.1

[146] Joshua Levin and Theophilus A. Benson. Viperprobe: Rethinking microservice observability with ebpf. In *CloudNet*, pages 1–8, 2020. 6.1

[147] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. volume 12, pages 2118–2130. VLDB Endowment, 2019. 6.1

[148] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. opengauss: An autonomous database system. *Proc. VLDB Endow.*, 14 (12):3028–3041, 2021. 6.1

[149] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wes McKinney, and Andrew Pavlo. Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats. volume 14, pages 534–546. VLDB Endowment, 2021. 3.5

[150] Tim Liang. The growing pains of database architecture. https://www.figma.com/blog/how-figma-scaled-to-multiple-databases/, 2022. 4.1.1

[151] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *SIGMOD*, pages 631–645, 2018. 3.1.1

[152] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. Mb2: Decomposed behavior modeling for self-driving database management systems. In *SIGMOD*, pages 1248–1261, 2021. 3, 3.1.1, 3.1.4, 3.1.4, 3.2.1, 3.5, 3.5.5, 6.1

[153] Alan Maguire. Introducing bpftune for lightweight, always-on auto-tuning of system behaviour. https://blogs.oracle.com/linux/post/introducing-bpftune, June 2023. 2.2.3

[154] David A. Maltz and Pravin Bhagwat. TCP splice for application layer proxy performance. *J. High Speed Networks*, 8(3):225–240, 1999. 6.2

[155] Ryan Marcus and Olga Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, 2019. 3, 3.1.1, 3.1.2, 3.1.3, 3.1.4, 6.1

[156] MariaDB. MariaDB Enterprise Documentation. https://mariadb.com/docs/reference/mdb/system-variables/join_buffer_size/. 3.3.2

[157] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 262–273. IEEE, 2018. 6.1

[158] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, January 1993. 2.2

[159] Chris McCord. Rise of the Phoenix - Building an Elixir Web Framework. July 2014. 4.1.1

[160] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. RCU usage in the linux kernel: Eighteen years later. *ACM SIGOPS Oper. Syst. Rev.*, 54(1):47–63, 2020. 3.5.2, 5.1.2

[161] John McNamara, Ian Stokes, Luca Boccassi, and Kevin Traynor. API/ABI Stability and LTS: Current state and Future. https://www.dpdk.org/event/dpdk-userspace-dublin-2017/, 2017. 1.2, 2.1, 6.3

[162] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1):330–339, 2010. 1

[163] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *HPSR*, pages 1–8. IEEE, 2018. 6.2

[164] Domas Mituzas. Mysql is bazillion times faster than memsql. https://dom.as/2012/06/26/memsql-rage/, June 2012. 1.1

[165] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992. 5.3.3

[166] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and

resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*, pages 301–312. ACM, 2013. 6.1

[167] Maciej Mościcki and Piotr Rżysko. Unlocking Kafka's Potential: Tackling Tail Latency with eBPF. https://blog.allegro.tech/2024/03/kafka-performance-analysis.html, March 2024. 2.2.3

[168] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011. 3.2.1, 3.4.2

[169] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, pages 677–689, 2015. 3.5

[170] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. USENIX, 1999. 5.1.1

[171] Sujin Park, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Contextual concurrency control. In *HotOS*, pages 167–174. ACM, 2021. 2.2.3

[172] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-informed kernel synchronization primitives. In *OSDI*, pages 667–682. USENIX Association, 2022. 2.2.3

[173] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, et al. Self-driving database management systems. In *CIDR*, 2017. 3

[174] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. External vs. internal: An essay on machine learning agents for autonomous database management systems. *IEEE Data Engineering Bulletin*, pages 32–46, June 2019. 3, 3.1.1

[175] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *SIGMOD Conference*, pages 1981–1984. ACM, 2019. 5.1.1

[176] Yaser Raja. Resources consumed by idle PostgreSQL connections | AWS Database Blog. https://aws.amazon.com/blogs/database/resources-consumed-by-idle-postgresql-connections/, January 2021. 4.1.1

[177] Kun Ren, Alexander Thomson, and Daniel J. Abadi. VLL: a lock manager redesign for main memory database systems. *VLDB J.*, 24(5):681–705, 2015. 5.3.2, 5.5.1

[178] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *SIGMOD Conference*, pages 1539–1551. ACM, 2016. 5.3.3

[179] Marcel-Catalin Rosu and Daniela Rosu. An evaluation of TCP splice benefits in web proxy servers. In *WWW*, pages 13–24. ACM, 2002. 6.2

[180] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance implications of packet filtering with linux ebpf. In *ITC (1)*, pages 209–217. IEEE, 2018. 6.2

[181] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. RUMA has it: Rewired user-space memory access is possible! *Proc. VLDB Endow.*, 9(10):768–779, 2016. 6.3

[182] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with

disaster: Surviving misbehaved kernel extensions. In *OSDI*, pages 213–227. ACM, 1996. 1.3, 6.3

[183] Slava Shwartsman and Drew Gallatin. Kernel tls and tls hardware offload. https://papers.freebsd.org/2019/eurobsdcon/shwartsman_gallatin-kernel_tls_harware_offload/, September 2019. 2.1, 4.2.1, 7.1

[184] Giulio Sidoretti, Sebastiano Miano, Stefano Salsano, Gianni Antichi, and Aurojit Panda. Application Layer Processing Offload in the Kernel. https://2023.eurosys.org/docs/posters/eurosys23posters-final39.pdf, 2023. Poster presented at EuroSys 2023. 4

[185] Emin Gün Sirer. Broken by Design: MongoDB Fault Tolerance. https://hackingdistributed.com/2013/01/29/mongo-ft/, January 2013. 1.1

[186] Emin Gün Sirer. When PR says No but Engineering says Yes. https://hackingdistributed.com/2013/02/07/10gen-response/, February 2013. 1.1

[187] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM*, pages 15–28. ACM, 2016. 6.2

[188] Sugu Sougoumarane and Mike Solomon. Vitess: Scaling MySQL at YouTube using go. San Diego, CA, 2012. USENIX Association. 7.1

[189] Oliver Spatscheck, Jørgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Trans. Netw.*, 8(2):146–157, 2000. 6.2

[190] Alexei Starovoitov. LKML: Alexei Starovoitov [PATCH net-next] extended BPF. https://lkml.org/lkml/2013/9/30/627, 2013. 1.3, 2.2, 3.1.3, 4.2.2

[191] Alexei Starovoitov. BPF at Facebook. https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/, 2019. 2.2, 4.2.2

[192] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981. 1.1, 6.3

[193] Michael Stonebraker and Ariel Weisberg. The voltdb main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013. 5.4.2

[194] Gerald J. Sussman and Guy L. Steele Jr. Scheme: A interpreter for extended lambda calculus. *High. Order Symb. Comput.*, 11(4):405–439, 1998. 5.2

[195] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 14–27, 2017. 6.1

[196] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/spec/tpcc_current.pdf, June 2007. 3.5.1, 4.5.1

[197] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12. ACM, 2012. 5.3.2

[198] Chris Travers. An introduction to Memory Contexts. https://www.pgcon.org/2019/schedule/attachments/514_introduction-memory-contexts.pdf, May 2019. 1.1

[199] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy

transactions in multicore in-memory databases. In *SOSP*, pages 18–32. ACM, 2013. 5.5.4

[200] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. revisiting the open vswitch dataplane ten years later. In *SIGCOMM*, pages 245–257. ACM, 2021. 1.2, 2.1, 6.3

[201] Marco Tusa. What About ProxySQL and Mirroring? https://www.percona.com/blog/proxysql-and-mirroring-what-about-it/, jun 2017. 4.4.2

[202] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017. 3, 3.1.4, 3.4.4, 6.1

[203] Marcos Augusto M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson Rubens da Silva Santos, Eduardo P. M. Câmara Júnior, and Luiz Filipe M. Vieira. Fast packet processing with ebpf and XDP: concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1):16:1–16:36, 2021. 6.2

[204] Thiemo Voigt and Bengt Ahlgren. Scheduling TCP in the nemesis operating system. In *Protocols for High-Speed Networks*, volume 158 of *IFIP Conference Proceedings*, pages 63–80. Kluwer, 1999. 6.3

[205] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. Demonstrating udo: A unified approach for optimizing transaction code, physical design, and system parameters via reinforcement learning. In *SIGMOD*, pages 2794–2797, 2021. 6.1

[206] Vince Weaver. The Unofficial Linux Perf Events Web-Page. http://web.eece.maine.edu/~vweaver/projects/perf_events/, March 2013. 3.3.1

[207] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*, chapter 5, pages 211–213. Morgan Kaufmann, 2002. 5.3.2

[208] Charles B. Weinstock and William A. Wulf. An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–148, 1988. 5.3.1

[209] Antoni Wolski. TATP Benchmark. http://tatpbenchmark.sourceforge.net. 3.5.1, 4.5.1

[210] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. Microrca: Root cause localization of performanceissues in microservices. In *IEEE/IFIP Network Operations and Management Symposium*, NOMS '20, pages 1–9. IEEE, 2020. 6.1

[211] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, 2017. 4.1.1, 5.3.1, 5.3.2

[212] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. Leveraging ebpf for programmable network functions with ipv6 segment routing. In *CoNEXT*, pages 67–72. ACM, 2018. 6.2

[213] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 415–432, 2019. 3.5.6

[214] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy,

Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: in-kernel storage functions with ebpf. In *OSDI*, pages 375–393. USENIX Association, 2022. 1.2, 6.3

[215] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. Query performance prediction for concurrent queries using graph embedding. *Proc. VLDB Endow.*, 13(9):1416–1428, 2020. 6.1

[216] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: fast in-kernel distributed transactions with ebpf. In *NSDI*, pages 401–417. USENIX Association, 2024. 6.3

[217] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. Looking ahead makes query plans robust. *Proc. VLDB Endow.*, 10(8):889–900, 2017. 2.2.1