OLTP Through the Looking Glass 16 Years Later: Communication is the New Bottleneck

Xinjing Zhou
MIT CSAIL
USA
xinjing@mit.edu

Viktor Leis
Technische Universität München
Germany
leis@in.tum.de

Xiangyao Yu
University of Wisconsin-Madison
USA
yxy@cs.wisc.edu

Michael Stonebraker
MIT CSAIL
USA
stonebraker@csail.mit.edu

Abstract

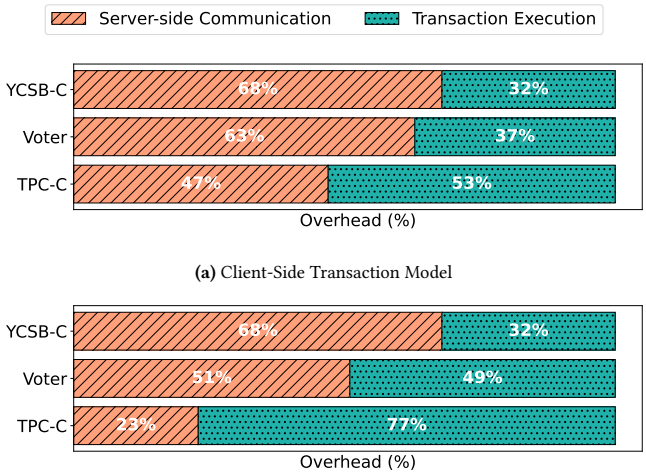
OLTP systems have significantly evolved since the 2008 study that identified CPU-bound tasks, such as buffer pool and concurrency control, as primary bottlenecks. In this paper, we revisit these assumptions and demonstrate that communication overhead is now the dominant factor affecting OLTP performance. Using comprehensive whole-stack benchmarks on modern hardware, we analyze both stored procedure and client-side transaction models, revealing that messaging costs play a critical role in end-to-end performance. Additionally, we explore user-defined code isolation, an often-overlooked aspect that poses security risks, especially in multi-tenant database systems. We find that increased isolation can lead to higher communication costs, highlighting the need for new strategies to mitigate these overheads.

1 INTRODUCTION

In a 2008 paper, some of us explored OLTP performance on conventional hardware and software [12]. Notably, we ran TPC-C on the Shore DBMS, a conventional DBMS with disk-based storage, a buffer pool, concurrency control based on locking, and standard Aries-style write-ahead logging. As such, it was typical of the commercial DBMSs of the time. We found that the overwhelming majority of CPU time was spent on services such as buffer pool management, concurrency control, crash recovery, and latching. Hence, to go a lot faster, one has to deal with these sources of overhead.

Since then, there have been several high-performance OLTP engines, including H-Store, VoltDB, Hyper, LeanStore, MemSQL, Silo, and HANA. These systems exploit various innovative solutions to the issues noted above to provide tremendous performance improvements over traditional systems in benchmarks. However, most academic OLTP benchmarks (including [12]) have two major problems.

Ignored Network/Communication. Previous work [12] only analyzed back-end components of a database system and ignored the messaging cost of communicating with a front-end application or within the database system. Even when running a stored procedure (SP) transaction model, this cost cannot be ignored. Moreover, although stored procedures are standard practice in research systems, many real-world applications prefer running transaction logic on the client side for better debuggability, security, and ease of version control [19]. This makes messaging costs much more expensive than in the SP model.



(a) Client-Side Transaction Model

(b) Stored-Procedure Transaction Model

Figure 1: Server-side CPU Overhead Breakdown. In the client-side transaction model the DBMS only processes SQL queries, while in the stored-procedure model it additionally runs the transaction logic.

Ignored User-Defined Code Isolation. When systems use an SP transaction model, most assume it is permissible to run stored procedures in the same address space as the DBMS. This means there is no *isolation* between an SP and the DBMS kernel or between different SPs. This allows errant or malicious SPs to interfere with each other and the DBMS. Worse yet, malicious SPs can obtain unauthorized data, which presents a security flaw. In some use cases, this risk is acceptable due to careful testing; in others, non-isolation is unacceptable. As cloud database systems move toward a multi-tenant architecture, such risks will become less acceptable.

In response, this paper presents "Looking Glass 2.0". We perform whole-stack benchmarks measuring throughput and response time on modern hardware, comparing the SP model with client-side transaction logic under different levels of isolation. To avoid the performance bottlenecks of traditional DBMSs, we primarily focus on high-performance OLTP engine VoltDB in this study. PostgreSQL results are also included for some minor experiments for reference.

Figure 1 depicts one of the major findings of this paper. When isolation is not required, the CPU cycles spent in messaging are "the high pole in the tent", even when using a stored procedure model. With a traditional transaction model, CPU overhead in communication is higher. While the stored-procedure model helps reduce

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

Database System	Programming Language for Stored Procedures and UDFs	Isolation Mechanisms			
		None	Language Isolation	OS-Level Isolation	Virtualization
VoltDB	Java		✓		
Oracle	PL/SQL, Java, JavaScript		✓		
SQL Server	T-SQL, C#		✓		
PostgreSQL	PL/pgSQL, C, Python, Rust, JavaScript	✓	✓		
MySQL	PL/SQL, JavaScript	✓	✓		
IBM DB2	PL/SQL, Java, C++	✓	✓		
MariaDB	PL/SQL, C/C++ (UDFs)	✓	✓		
SAP HANA	PL/SQL, R		✓		
Teradata	PL/SQL		✓		
MongoDB	JavaScript		✓		
Snowflake	Java, JavaScript, Scala, Python		✓	✓	
Redshift	PL/pgSQL, Python, AWS Lambda		✓	✓	✓

Table 1: Programming languages and isolation mechanisms for user-defined code in existing database systems

such costs by reducing the number of round trips between transaction logic and the database engine, it does not help with simpler transactions such as those in YCSB-C and Voter workloads. Hence, any improvements to DBMS performance impact a minority of the path length, and the most profitable source of OLTP improvement is in the arena. When isolation is required, communication cost is dramatically the highest cost item, overshadowing the CPU cycles spent in transaction execution.

Hence, this paper explores possible improvements to TCP/IP and Linux kernel. These include kernel bypass with user-space networking for messages between the client and the DBMS. We also report unpleasant experiences when applying kernel-bypass to DBMS, suggesting room for improvement on the OS side. To isolate SPs from the DBMS we classified five possible levels of isolation and explored a spectrum of approaches from no isolation to virtual machine isolation. Higher degrees of isolation resulted in additional messages and higher costs.

The contributions of this paper can be summarized as follows:

- We performed a systematic whole-stack performance breakdown of a modern OLTP engine.
- We carefully dissected the space of possible isolation mechanisms for user-defined code in OLTP systems and experimentally explored the trade-off between the degree of isolation and overall cost.
- We envision several directions for future research based on the experimental results.

2 BACKGROUND AND METHODOLOGY

We first examine isolation mechanisms before explaining our benchmarking methodology.

2.1 Isolation Mechanisms for User Code

Client-Server Isolation. The baseline option is to run user-defined code in a client process separate from the DBMS. This is also commonly denoted as *interactive transactions* and is supported by virtually all DBMSs. The extreme case of isolation is to place the user and server processes on different servers connected through the network. Reducing communication overhead requires co-locating the user code and DBMS on the same server, as we discuss next.

No Isolation. Many systems, including PostgreSQL, DB2, MariaDB, and Teradata, allow user-defined logic in *unsafe* languages such as C by loading user code into the DBMS as a shared library. Given that malicious or erroneous user-defined code can easily access the content of DBMS memory or bring down the entire system, this approach provides effectively *no isolation*. In exchange for giving up isolation, this maximizes performance as user-defined code can communicate with the DBMS kernel through shared memory.

Language Isolation. The next level of isolation is provided by *safe* stored procedure (SP) languages running within the DBMS process. Such languages include domain-specific languages (e.g., PL/SQL, T-SQL), virtual-machine-based languages (e.g., Python, Java, JavaScript), and memory-safe languages (e.g., Rust). By disallowing unsafe memory accesses, safe SP languages provide some level of isolation. However, the isolation fully relies on the correctness of compiler and language runtime and is therefore somewhat limited. For example, Oracle [18] and VoltDB run Java-based stored procedures in the same address space as the DBMS kernel. User-defined code can break out of the virtual machine by using unsafe language features, VM exploits [5], or bugs in the compiler.

OS-Level Isolation. To provide better isolation guarantees than language-based isolation, one can run an SP in a different OS process than the DBMS but on the same server. However, important operating system resources such as the file system and network devices are still shared. One way to provide stronger isolation is using *containers*, which virtualize all OS resources, and Linux' *SECCOMP* mechanism, which limits the system calls that can be made by a process. Snowflake, for example, runs user-defined code in a separate container with access to a dedicated file system without the ability to make network connections. In this approach, user-defined code uses OS-provided inter-process communication (IPC) mechanisms such as sockets to interact with the DBMS kernel process. This provides much better isolation than language-based approaches.

Virtualization. OS-level isolation obviously relies on the correctness of the OS. Attacks against the OS kernel can compromise the memory of all running processes. Therefore, a stronger isolation

approach is to run user-defined code on a separate OS kernel in a virtual machine on the same host. This confines malicious code execution to the virtual machine. This model requires inter-VM communication between the DBMS and user-defined code.

To summarize, the five mechanisms discussed provide trade-offs between the proximity of the user code to the DBMS and the degree of isolation and communication overhead. We summarize the isolation mechanisms for user-defined logic of existing DBMSs in Table 1. Note that the table is non-exhaustive.

2.2 VoltDB Primer

To understand our testing approach, we first review VoltDB's architecture. As shown in Figure 2, VoltDB adopts a shared-nothing architecture, partitioning data across multiple cores. Each core has its own partition serviced by a single worker thread. Networking threads handle client connections and interact with the TCP/IP stack in the Linux Kernel. Each worker runs stored procedures (SPs) from a task queue in a single-threaded manner within the same process as the DBMS kernel. The Java-based query engine executes SPs using an in-memory C++ storage engine, and SPs are written in Java with embedded SQL. VoltDB distinguishes between single-partition and multi-partition transactions. Single-partition transactions, which access data within one partition, are optimized to run to completion without locks or latches. As a result, VoltDB maximizes CPU efficiency in its OLTP engine when executing these transactions. Conversely, multi-partition transactions are serialized through a global coordinator, allowing only one at a time, and use standard two-phase commit. This paper focuses on single-partition transactions, as they provide state-of-the-art performance, which we strive to improve on. VoltDB separates the logic of transaction processing from network I/O activities. A group of dedicated network threads are created for interfacing with the Linux Kernel with BSD socket APIs and event polling. The network threads parse requests and dispatch stored procedure invocations to dedicate OLTP workers through message passing. The network threads also write responses back to the sockets after OLTP workers finish transaction execution.

2.3 Modeling Network and Isolation Models

Interactive Transactions. Since VoltDB requires a stored procedure model, we simulate interactive transactions by breaking any stored procedure into multiple stored procedures, one per SQL statement. This closely simulates a classic environment with the client running on the same machine as the DBMS. As an optimization, we batch together SQL queries that have no dependencies. These batches are then run as individual stored procedures.

User-Space Networking. Like most DBMSs, VoltDB relies on the TCP/IP stack of the host operating system. We extended VoltDB with support for user-space networking using DPDK and FStack [1], a user-space TCP/IP stack. In order to interface with F-stack, we had to rewrite major parts of the VoltDB networking layer because F-stack requires using a set of new polling/read/write APIs and only supports a single-threaded TCP/IP stack. Therefore, F-stack-enabled VoltDB uses only one network thread for performing network I/O. The network thread spin-polls the NIC for available packets.

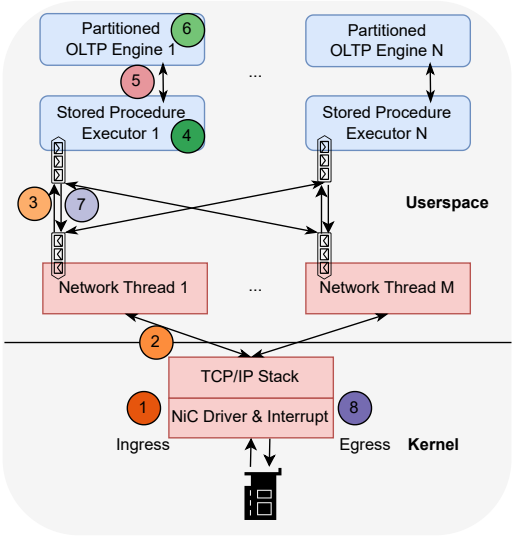


Figure 2: Architecture of VoltDB

Language Isolation. Java SPs for VoltDB are compiled into individual classes and loaded into the same JVM on which VoltDB runs. Therefore, VoltDB provides language isolation by default.

OS-Level (Process) Isolation. To provide process isolation, we modified the VoltDB SP runtime to pair every VoltDB worker with a separate Linux process in which SPs for that worker are loaded. To communicate between this process and the corresponding worker, we compared three IPC mechanisms: TCP/IP, shared memory with busy-polling for notification, and shared memory with a Unix domain socket for notification. When exchanging messages between the SP process and the OLTP engine, SQL queries and result sets are serialized and deserialized. Although we could have run each SP in its own process, we chose not to, as the performance would be essentially identical to the architecture we tested.

Containerization. In addition to process isolation, we explored running each SP process in a Docker container and the same set of communication mechanisms used for process isolation.

Virtualization. The last and strongest isolation approach we explored is virtualization. We ran each SP process on top of a Linux kernel in a virtual machine (KVM). For this setup, the DBMS kernel communicates with each SP process through the TCP/IP stack in the host Linux kernel as well as the guest Linux kernel. When a network packet is sent from the DBMS kernel, it needs to first traverse the TCP/IP stack of the host Linux kernel, and then it is routed through the kernel network bridge to arrive at a virtualized NIC maintained in the hypervisor in the kernel. Lastly, the packet crosses the virtualization boundary to be handled by the receiving side of the guest kernel's virtual NIC and TCP/IP stack.

2.4 Profiling Methodology

We use manual instrumentation to profile VoltDB and the Linux perf tool for the kernel. Specifically, for kernel-space profiling, we leveraged the `perf-trace` [14] tool to place trace points inside the Linux kernel, which replace traced instructions with breakpoint instructions. When a trace point is hit, the Linux kernel traps into a breakpoint handler that records timing information and then resumes execution. For VoltDB profiling, we found `perf-trace` [10]

Workload - TX	# DB Interactions	Ratio
YCSB C - Get	1	100%
Voter - Vote	2	100%
TPC-C - New Order	8-18	45%
TPC-C - Payment	5	43%
TPC-C - Delivery	4	4%
TPC-C - Stock Level	2	4%
TPC-C - Order Status	3	4%

Table 2: Workloads. A DB interaction is the largest batch of SQL queries that can be sent to the DBMS in one round-trip to be executed independently. We break the logic of each transaction into the shortest sequence of interactions with the database systems where an interaction depends on the result sets of previous interactions.

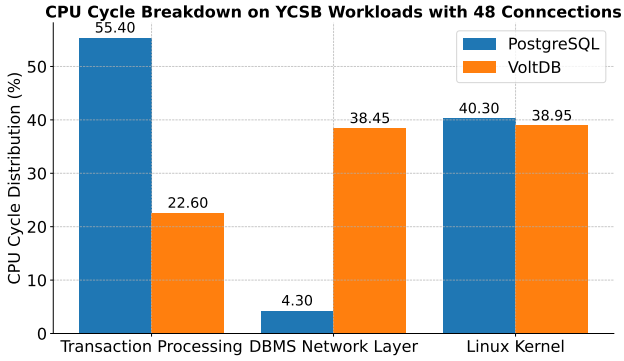


Figure 3: CPU Cycle Distribution for PostgreSQL and VoltDB. Cycles are classified into Transaction Processing, DBMS Networking Layer, and Linux Kernel.

too expensive as each activated trace point results in a context switch to the kernel. Instead, we manually instrumented the VoltDB code to record a trace entry in a memory buffer. At the end of the benchmark, VoltDB dumps the buffer to a file in the same format as the output by `perf-trace`.

3 EXPERIMENTAL EVALUATION

3.1 Environment

Hardware. We ran all experiments on Google Cloud using two compute instances, each with 16 vCPU (Intel Haswell, 2.3GHz) and 64 GB memory. The two instances are connected by a network with a measured 16-Gigabit bandwidth in one direction and 60us median round-trip latency (measured using sockperf).

Software. The machines are running Linux version 6.8.0-1007-gcp. VoltDB is compiled with OpenJDK version 1.8.0_402 and gcc 11.4.0. We configure the client machine to run 512 concurrent connections to the VoltDB server. Each client iteratively runs the benchmark script. We evaluate system performance using three workloads. As shown in Section 3.1, these workloads cover a wide range of transaction complexity and round-trips to the database system.

YCSB. The Yahoo! Cloud Serving Benchmark is a key-value store benchmark. We initialize the YCSB table with 10M 128-byte key-value pairs and perform random single-tuple index lookups.

Voter. This benchmark simulates a phone-based election process and contains a collection of single record queries and updates. 10,000 voters are given a fixed number of votes for 6 candidates. Each voter selects any candidate iteratively until their votes are exhausted.

TPC-C. The TPC-C benchmark is the industry-standard benchmark suite for OLTP databases. Each transaction contains complex interactions with the database. We configured the system with 256 warehouses and partitioned the database by warehouse. We modified the benchmark so that every transaction targets only a single warehouse to avoid multi-partition transactions on VoltDB.

3.2 Where do the CPU Cycles Go?

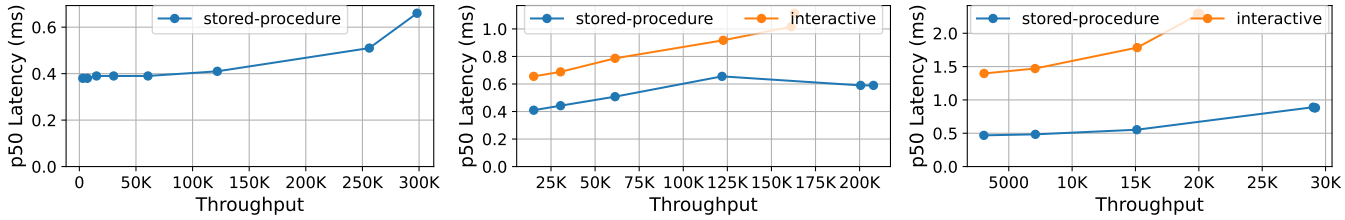
We begin by showing CPU cycle distribution on the server. We run simple YCSB-C workloads with 48 connections. The results are shown in Figure 3. VoltDB spends less than a quarter of the total cycles on transaction processing. This is partly because the VoltDB eliminated buffer pool/ locking/latching that were bottlenecks pointed out by the original looking glass paper [12]. Instead, the CPU bottlenecks are now shifted to the DBMS networking layer and the Linux Kernel. Surprisingly, the DBMS networking layer in VoltDB contributes almost 40% of the CPU cycles. Our profiling found that 8% of the cycles are spent on `epoll_wait/epoll_ctl/read/write` functions in the glibc library. The rest of the cycles are mostly spent on message passing and scheduling among the network threads and OLTP workers. For reference, we also included PostgreSQL (v13) results. We configure PostgreSQL buffer pool so that all data fits in memory. This eliminates storage I/O overhead and allows us to focus on communication within the system. As expected, transaction processing dominates with 55% of cycles due to the bottlenecks of a traditional DBMS [12]. The DBMS network layer consumes only 4.3% in PostgreSQL compared to 39% in VoltDB. This is because transaction processing worker directly read/write to/from a TCP socket, avoiding most of the DBMS internal communication overhead. Interestingly, there are still nearly 40% cycles spent in Linux Kernel on networking for PostgreSQL, similar to VoltDB.

Phase	Description
1 Network Receive	Time from receiving data on the NIC to readying it in a socket.
2 Socket Read	Time to read data from the socket.
3 Request Queuing	Time to assign the request to a OLTP worker.
4 Procedure Execution	Time to run the transaction logic.
5 Isolation Overhead	Time for communication related to isolation.
6 Query Execution	Time to run SQL queries.
7 Response Queuing	Time to queue the response for transmission.
8 Network Send	Time to write the response through the network stack.

Table 3: Phases of processing a transaction in chronological order.

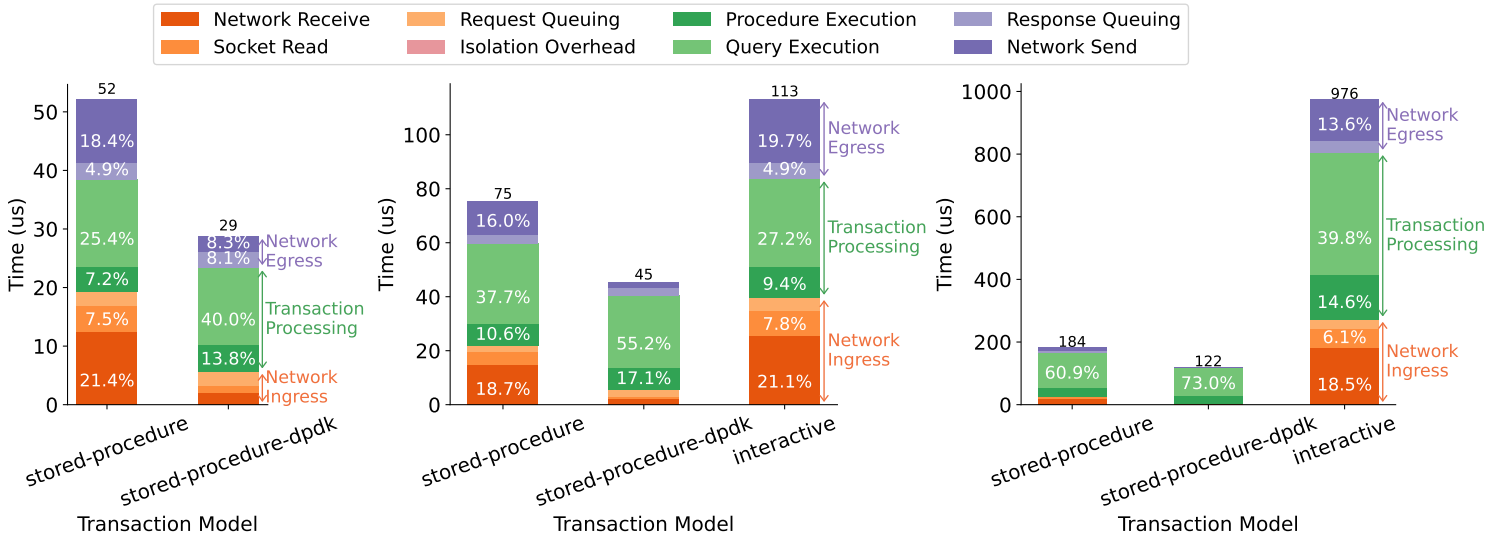
3.3 Results for No Isolation

We first present detailed results for the no-isolation case, comparing interactive transactions with SPs. Specifically, in Figure 4 we show the median latency for the three benchmarks as we ramp up the throughput on our system. Note that there is a single curve for



(a) YCSB **(b)** Voter **(c)** TPC-C

Figure 4: Median Latency vs. Load for Interactive Transactions and Stored-procedure Transactions



(a) YCSB **(b)** Voter **(c)** TPC-C

Figure 5: Server-side Transaction Latency Breakdown (Excluding Queuing).

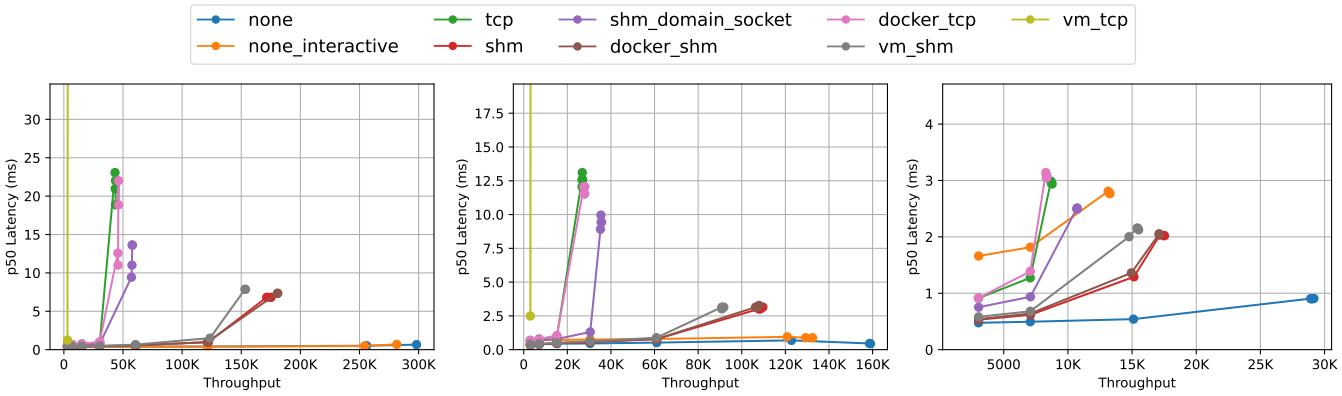
YCSB as the two implementations are effectively identical. Also, note that the logic for both cases is the same; hence, the only difference between the two implementations is the number of messages. Obviously, throughput increases as the load is applied until our system becomes CPU-bound and saturates. Equally obvious is the increase in latency as load is applied because of queuing delays.

With the Voter workload, shown in Figure 4b, SP transactions have up to 72% lower latency and 23% higher achievable throughput. With more complex workloads such as TPC-C where there are more network round-trips, shown in Figure 4c, we observe that interactive transactions incur 3.5× higher latency at similar throughput points. The SP model can achieve a maximum throughput that is 2.1× higher than the interactive case. These results show the desirability of using an SP model.

Server-side Transaction Latency Breakdown (excluding queuing). To understand the performance of the SP model, we categorize server-side latency of a transaction into the 8 phases highlighted using colored numbers in Table 3.

Networking Dominates CPU Time. Figure 5 presents a detailed breakdown of server-side latency for interactive (bars labeled as interactive) and SP transactions (bars labeled as stored-procedure) for our three benchmarks. Each bar represents the average CPU time spent in each of the eight categories measured in microseconds (µs). The first conclusion to draw from Figure 5 is the high pole in the tent is usually the server CPU time spent in networking. Except for TPC-C, networking dominates DBMS processing. Remarkably, it is more expensive to send work to the DBMS and receive answers than it is to construct those answers. Of course, if the transaction stored procedures get beefy enough, as exemplified by TPC-C, the system remains DBMS-bound. The second point is that stored procedures are a good idea. They lower the number of messages and increase the percentage of work spent on DBMS processing. The third point to make is that there is no magic bullet for improving network CPU time; the cost is spread over the whole stack. As a result, one needs to explore kernel bypass to get better performance, a topic to which we now turn.

Kernel Bypass. To leverage kernel bypass, we replace the Linux networking stack used by VoltDB with a user-space TCP/IP stack (F-stack) using the DPDK library. As shown in Figure 5 (bars labeled `stored-procedure-dpdk`), with kernel-bypass and the SP model, the CPU time spent on transaction processing increases from 33%/48%/70% to 53.8%/72%/91% for YCSB/Voter/TPC-C workloads. These improvements mainly come from the elimination of system calls, reduced copying, interrupt processing, and a simpler networking stack. Note that the overhead of internal DBMS communication for queuing requests and responses remains.



(a) YCSB **(b)** Voter **(c)** TPC-C

Figure 6: Median Latency vs. Load Across Various Isolation Mechanisms

End-to-end Latency Breakdown. We next break down transaction end-to-end latency. To minimize queuing delays distorting the results, we offer the maximum load before the end-to-end latency spikes. The results are shown in the following table, with the first three rows using the SP model and the last three rows using the client-side transaction model:

Workload	Total	Cli.↔Srv.	Srv. Net.	Xact Exec.
YCSB-C	410μs	312μs	64μs	34μs
Voter	425μs	343μs	40μs	42μs
TPC-C	483μs	312μs	42μs	129μs
YCSB-C	410μs	312μs	64μs	34μs
Voter	688μs	572μs	67μs	49μs
TPC-C	1,471μs	710μs	310μs	451μs

The results show that the time spent on client-to-server networking (the 3rd column) is the dominant component. This component increases further when using the client-side transaction model, as there are more round-trips. It also shows the superiority of the SP model. When running TPC-C using client-side model, the time for transaction execution increases compared to SP model due to repeated SQL parsing, planning, serialization, and deserialization.

3.4 Results for Isolation

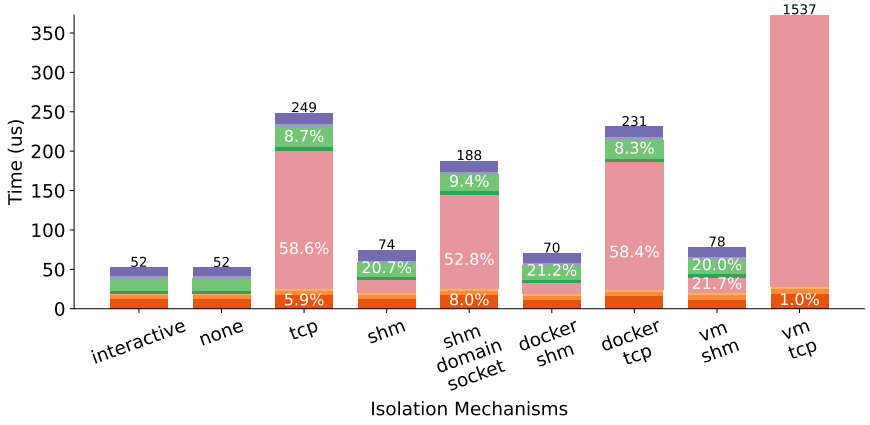
Latency vs. Throughput. We next explore the end-to-end latency-throughput trade-offs for different isolation mechanisms and examine server-side overhead in detail. The results for different isolation mechanisms are shown in Figure 6. Generally, all configurations experience latency increases as more load is applied to them and systems get more saturated. Not surprisingly, the configuration with no isolation (labeled `none`) has the best latency-throughput trade-off curve, as there is no IPC overhead between the OLTP engine and the stored procedure. It achieves up to 65% higher throughput compared to the next-best-performing ones when the highest load is present. The next best-performing configurations are the ones with shared memory and polling (`shm, docker_shm, vm_shm`). At low load, the latency is very close to the baseline with no isolation. With more complex transactions and higher load, shared-memory adds a couple of milliseconds of latency and tops out at dramatically less throughput. This is mostly due to serialization and de-serialization overhead. Configurations with TCP and domain sockets have much worse latency and achieve significantly lower throughput compared to shared-memory and no-isolation. Domain sockets perform slightly better than TCP because domain sockets in Linux are more efficient than the general TCP/IP stack. We do not observe a significant difference between the procedure process running in a container and running in the host, suggesting the Linux kernel uses a similar code path for the IPC mechanisms. We also observe similar trends for 99-percentile latency for all the configurations.

To summarize, isolation imposes dramatic throughput and latency overheads. The best-performing mechanism involves process isolation with shared-memory and polling. Containerization does not impose significant overhead on top of process isolation.

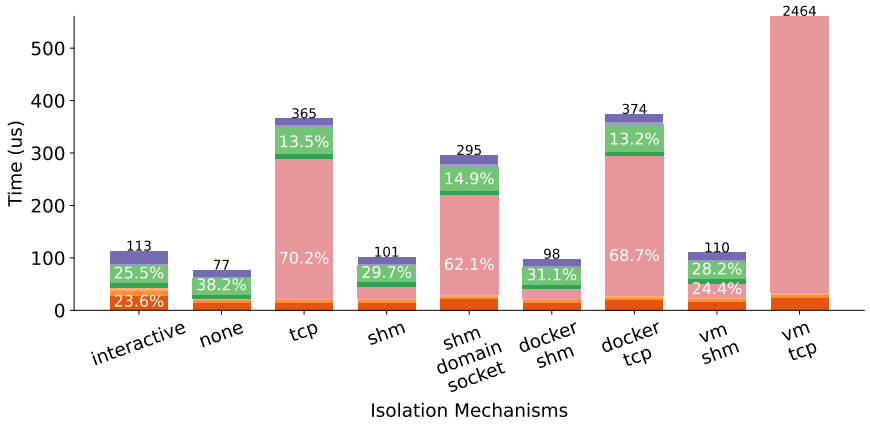
Server-side Transaction Latency Breakdown. As before, we perform the same server-side transaction latency breakdown. The results are shown in Figure 7, which provides a detailed analysis of the three benchmarks: YCSB, Voter, and TPC-C using different isolation mechanisms.

One should note that all mechanisms that provide some degree of isolation are primarily dominated by the CPU path length in the communication stack. Specifically, using process isolation and TCP, `Isolation Overhead` dominates server-side overhead, taking up 58%/70%/63% of the CPU time on YCSB/Voter/TPC-C workloads. Each DB interaction requires a traversal through the local TCP/IP stack between the procedure process and the OLTP engine. Unix domain sockets help reduce the overall overhead by about 32% compared to TCP, as the kernel path is shorter. However, it is still significantly higher than no-isolation. The next best-performing configuration is process isolation with shared memory and polling for communication that still has 34%/31%/66% higher overall overhead compared to no isolation. Note that with shared memory and polling, we do not observe a significant difference between running the procedure process in the host and container. This is because shared memory allows for exchanging messages directly through physical memory, bypassing the container boundary completely.

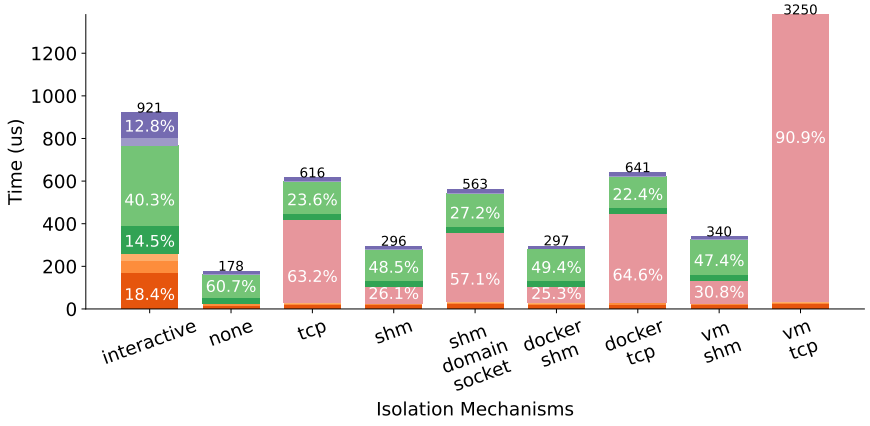
Note that, when running the procedure process in a virtual machine, `Isolation Overhead` drastically increases. To isolate why virtualization introduces such high overhead, we ran a micro-benchmark for TCP using `sockperf`, obtaining the following round trip times (RTT):



(a) YCSB



(b) Voter



(c) TPC-C

Figure 7: Server-side Transaction Latency Breakdown (Excluding Queuing).

Config	RTT
VM ↔ VM	60us
VM Guest ↔ Host	176us
TCP via Loopback	21us

Surprisingly, the RTT between two Google Cloud instances is lower than that of the RTT between a guest VM and host on the same instance. From the table, we can see that the overhead introduced by the guest VM is about 8.5× compared to communicating through the loopback interface on the same host. This is consistent with the difference between `vm tcp` and `tcp` shown in Figure 7a. We

attribute this high cost to QEMU/KVM and nested virtualization, as each packet needs to traverse two TCP/IP stacks and the VM boundary in one direction.

4 DISCUSSIONS AND RESEARCH DIRECTIONS

In this section, we summarize the lessons we learned from the experiments and lay down a few research directions that we think are worth exploring.

4.1 Lessons Learned

Lesson 1. Both the interactive transaction model and stored-procedure model are bottlenecked on messaging when transactions are simple (YCSB/Voter), accounting for 52%-67% of the runtime overhead. The stored procedure model helps relieve these bottlenecks when transactions are more complex at the expense of security. Kernel-bypass and user-space network stacks can help reduce such overhead down to 28%-46%.

Lesson 2. On modern networks, the stored-procedure model still has substantial latency and throughput advantages over the interactive transaction model. These increase with the number of interactions with the DBMS. The two models are on par when a transaction contains only one round trip with the DBMS.

Lesson 3. Stronger isolation mechanisms for stored-procedure introduce substantial overhead. For example, process isolation with shared memory and polling offers a good balance but still incurs 31%-66% higher overhead than no-isolation. TCP/IP and domain socket communication for process isolation further increases that overhead. Lastly, virtualization imposes an order of magnitude higher overhead on communication.

4.2 Research Direction #1: Beyond VoltDB and Main-Memory

Beyond VoltDB. This paper focuses primarily on VoltDB and single-partition transactions. However, we believe many conclusions also apply to other modern memory-optimized engines with shared-memory concurrency control, such as LeanStore [15], where the majority of CPU cycles are spent on useful work. An intriguing next step would be to explore distributed transactions and replication with proper concurrency control. In such scenarios, we anticipate that communication overhead would be even more pronounced, as distributed transactions and replication require additional network messages.

Beyond Main-Memory. While in-memory DBMSs have gained significant attention over the last decade, disk-based DBMSs continue to dominate the commercial market due to their cost-efficiency [17]. Network stack overhead is also critical for disk-based DBMSs, as shown in Figure 3, where networking accounts for 45% of CPU cycles in PostgreSQL when data fits in memory. Disk-based DBMSs have also seen renewed innovation driven by the trend of cloud disaggregation, where systems separate compute from storage[4, 21]. In such disaggregated OLTP systems, persisting log records and handling buffer pool misses involve network I/O operations with storage servers. A systematic analysis of the overhead in these systems could uncover valuable insights and future research directions.

DBMS Internal Networking. A notable finding in this study is that VoltDB's internal networking layer consumes a substantial portion of CPU cycles, as shown in Figure 3. In contrast, the internal networking overhead in PostgreSQL is significantly smaller. We also summarized how various DBMSs handle network I/O in Table 4, based on whether dedicated threads are used for network operations. The results reveal no clear consensus on network I/O strategies among DBMSs. A promising research direction would be to systematically investigate the trade-offs between these approaches.

Unified TX & Network Thread	Dedicated Network Threads
PostgreSQL	Redis
MySQL	VoltDB
SQL Server	Oracle
AWS Aurora	Cassandra
TiDB	ScyllaDB
	OceanBase

Table 4: How DBMSs Perform Network I/O.

4.3 Research Direction #2: Better Bypass Mechanisms

Better Kernel Bypass Mechanisms. While kernel bypass mechanisms like DPDK reduce networking overhead, they require exclusive NIC access and busy polling, which is inflexible and energy-inefficient. Our experience integrating DPDK into VoltDB was challenging and painful, as DPDK operates as a layer-2 stack without support for routing and transport protocols, on which most DBMSs depend. As a result, DBMS developers must either build their own transport and routing protocols or use third-party implementations, which are also known to have unstable APIs across versions [20]. Integrating DPDK requires significant engineering effort to rewrite the DBMS networking layer and adds maintenance burdens for environments lacking DPDK support. Additionally, standard Linux networking tools are unavailable in the DPDK ecosystem, complicating debugging, development, and monitoring [20]. Similar challenges with DPDK integration in DBMSs have been reported [7]. The only DBMS using DPDK in production we know of is Yellowbrick [9]. Such usability issues is one of the reasons kernel bypass has seen limited adoption in data systems. Therefore, further research is needed to design a user-friendly, efficient kernel bypass approach with full transport support for DBMSs.

Bypass for Client-side Network Stack. An often-overlooked but critical area for improvement is the client network stack supporting applications that connect to DBMSs. Most clients, such as web servers or API servers, rely on kernel-provided network stacks, which are inefficient. In terms of end-to-end latency, the client network stack likely adds as much overhead as the DBMS server stack. From a cost perspective, optimizing client stacks for DBMS communication could reduce the number of application instances required, lowering the total cost of ownership. However, kernel-bypass frameworks like DPDK are typically used on back-end servers, where administrators can control the hardware stack and dedicate a NIC exclusively to the application. Such stringent constraints are impractical in environments where servers host multiple applications for efficiency. This underscores the need for more usable and resource-efficient kernel bypass mechanisms that

can be deployed in diverse and constrained environments, not just controlled back-end servers.

Better User Bypass Mechanisms. Another promising direction is exploring approaches that push database logic into the kernel space. Recent works have demonstrated that the eBPF infrastructure enables data-intensive operations, such as database proxying [7] and caching [11], to be integrated into the OS networking stack, avoiding costly data transfers between user space and kernel space. Building on this foundation, an intriguing avenue for research is pushing more, if not all, stateful database components—such as query execution and buffer management—into the Linux kernel [6]. However, a significant research challenge lies in eBPF's constraints, as it only supports user programs that are verifiably safe. For example, loops must be bounded, and programs are restricted to a predefined set of kernel data structures for stateful operations. Additionally, the Linux kernel prohibits floating-point operations, which are critical for many database applications. These restrictions necessitate addressing portability and compatibility challenges. Despite these limitations, this remains a promising research direction.

4.4 Research Direction #3: Better Isolation Mechanisms

Tighter DB-OS Co-Design. Existing mechanisms provided by general OSs are often insufficient for database system needs. For example, the isolation mechanisms explored in this study cannot simultaneously meet the demands of safety, low communication overhead, and high efficiency. More research on the intersection of the DB and the OS is needed to design mechanisms that can strike a better balance in this area. In particular, process-level isolation provides the most practical and widely used safety guarantees based on virtual memory and privilege separation hardware. It is, therefore, worth exploring better IPC mechanisms tailored to this approach. Although bypass mechanisms help relieve some of the overhead, they come with their own set of challenges. Most of these challenges are a result of security concerns. For example, interrupt processing, virtual memory, and privilege separation are key to implementing efficient IPC and process isolation. However, they are only programmable in the kernel space. The database system that runs in user space needs to cross expensive security boundaries and traverse complex kernel paths to access these hardware constructs. Direct and safe access to these hardware constructs would enhance the efficiency and flexibility of the database system. DB-OS co-design approaches [16, 22] that utilize virtualization offer a potentially fruitful research direction.

WebAssembly Sanboxing for Isolation. WebAssembly [3] (Wasm) is an emerging binary instruction format for language virtual machines that shows significant promise for sandboxing untrusted code. A compelling research direction is the integration of WebAssembly into DBMSs to sandbox user-defined code. However, it remains unclear whether WebAssembly can meet the flexibility and performance demands of user-defined code in a DBMS context. Thus, a systematic study of this integration would be valuable.

4.5 Research Direction #4: Better Transaction Models

While stored-procedure transactions are known to reduce communication overhead and end-to-end latency for complex transactions, many applications favor client-side transactions due to their simplicity, decoupling, language flexibility, ease of integration with external services, and manageability. Additionally, the client-side transaction model inherently provides the strongest isolation guarantees. Research is needed to combine the advantages of both models. For instance, stored-procedure synthesis [2, 8, 13] with program analysis techniques converts client-side transaction logic into stored procedures, offering a promising approach to bridging the gap between these two paradigms. However, this approach still faces many challenges, such as incompatibility between source programming languages and stored procedure languages (e.g., Java vs. PL/SQL), and often requires manual intervention and complex setup.

5 CONCLUSION

This work provides a detailed analysis of OLTP performance in a modern database engine on contemporary hardware. Our findings reveal that bottlenecks have shifted toward communication, and the increasing demand for enhanced security in user-defined logic significantly amplifies these challenges. Consequently, the database community should prioritize improving networking and communication around the DBMS, rather than focusing solely on the DBMS core engine.

ACKNOWLEDGMENTS

We thank Prabhakar Kafle and Darren Lim for contributing to the experiments. We thank the anonymous CIDR reviewers for many insightful suggestions on improving our manuscript. This research is supported by Google.

References

[1] [n.d.]. F-Stack. https://github.com/F-Stack/f-stack
[2] [n.d.]. Stored Procedures: The Good, The Bad, and The Elegant. https://www.dbos.dev/blog/stored-procedures-good-bad-elegant
[3] [n.d.]. Web Assembly. https://webassembly.org/
[4] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.
[5] David Buchanan. [n.d.]. A library to assist writing memory-unsafe code in "pure" python, without any imports. https://github.com/DavidBuchanan314/unsafe-python.
[6] Matthew Butrovich. 2024. *On Embedding Database Management System Logic in Operating Systems via Restricted Programming Environments.* Ph.D. Dissertation. Carnegie Mellon University.
[7] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. 2023. Tigger: A database proxy that bounces with user-bypass. *PVLDB* 16, 11 (2023), 3335–3348.
[8] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C Myers. 2012. Automatic partitioning of database applications. *arXiv preprint arXiv:1208.0271* (2012).
[9] Mark A Cusack, John Adamson, Mark Brinicombe, Neil A Carson, Thomas Kejser, Jim Peterson, Arvind Vasudev, Kurt Westerfeld, and Robert Wipfel. 2024. Yellowbrick: An Elastic Data Warehouse on Kubernetes.. In *CIDR*.
[10] Srikar Dronamraju. [n.d.]. Uprobe-tracer: Uprobe-based Event Tracing. https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt.
[11] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. {BMC}: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *NSDI*. 487–501.

[12] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *SIGMOD*.
[13] Gansen Hu, Zhaoguo Wang, Chuzhe Tang, Jiahuan Shen, Zhiyuan Dong, Sheng Yao, and Haibo Chen. 2024. WeBridge: Synthesizing Stored Procedures for Large-Scale Real-World Web Applications. *PACMMOD* 2, 1 (2024), 1–29.
[14] Masami Hiramatsu Jim Keniston, Prasanna S Panchamukhi. [n.d.]. Kernel Probes (Kprobes). https://docs.kernel.org/trace/kprobes.html.
[15] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4536–4545.
[16] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *PVLDB* 17, 8 (2024), 2115–2122.
[17] David Lomet. 2018. Cost/performance in modern data stores: How data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*. 1–10.
[18] Oracle. [n.d.]. Main Components of Oracle JVM. https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdev/Oracle-JVM-components.html.
[19] Andrew Pavlo. 2017. What are we doing with our lives? Nobody cares about our concurrency control research. In *SIGMOD*.
[20] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 245–257.
[21] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.
[22] Xinjing Zhou, Viktor Leis, Jinming Hu, Xiangyao Yu, and Michael Stonebraker. 2025. Practical DB-OS Co-Design with Privileged Kernel Bypass. *To appear at SIGMOD 2025* (2025).