ScaleRDB: A Scalable Relational Database for Manycores through Multi-Instance Shared-Memory Architecture

Changjong Kim
Seoul National University of Science and Technology
Seoul, South Korea
changjong5238@seoultech.ac.kr

Yongseok Son
Chung-Ang University
Seoul, South Korea
sysganda@cau.ac.kr

Sunggon Kim
Seoul National University of Science and Technology
Seoul, South Korea
sunggonkim@seoultech.ac.kr

Abstract

This paper presents ScaleRDB, a scalable relational database (RDB) designed to improve scalability on manycore systems. Our key idea is to leverage an independent yet cooperative architecture, enabling highly concurrent transaction execution while maintaining consistency via a lightweight mechanism. Specifically, ScaleRDB first introduces a transaction coordinator that distributes transactions across cores, allowing each core to process its transactions independently. Second, ScaleRDB devises a two-level asynchronous checkpoint mechanism, where local checkpoints are decoupled from the global checkpoint to achieve database consistency with minimal synchronization overhead. Finally, ScaleRDB adopts a localized I/O approach, enabling each core to handle its own logging and checkpoint I/O operations independently, thereby minimizing I/O stalls. We implement ScaleRDB with three techniques based on MySQL and evaluate its performance on a 128-core machine. Our evaluation results show that ScaleRDB improves transaction throughput (TPS) by up to 28.06×, 1.88×, 2.02×, and 1.52× compared with MySQL, PostgreSQL, *WAR* and *LRU-C*, respectively.

1 Introduction

Relational databases are most widely utilized, serving as the core software components for applications requiring structured data management and transactional integrity. A key distinguishing feature of relational databases is the ability to enforce data integrity through schemas, support complex queries with SQL, and guarantee strict ACID transactions [31, 75, 88]. These capabilities distinguish them from alternative databases, including key-value stores [32, 57, 89, 98], document databases [6, 9, 28, 76, 91], and wide-column stores [16, 59, 94, 106], which often sacrifice transactional guarantees, consistency, or query expressiveness for scalability or flexibility. These features make relational databases essential.

Despite these advantages, traditional relational databases (e.g., MySQL [77] and PostgreSQL [86]) do not fully leverage the parallel execution capabilities of modern hardware resources (e.g., manycore processors) [1, 23, 71, 117, 122]. Manycore systems, equipped with tens or even hundreds of cores, provide substantial hardware parallelism. However, relational databases remain unable to utilize this parallelism effectively due to global contention in concurrency control, shared data structure access, and synchronization [5, 19, 61, 99, 115, 121].

Figure 1 presents throughput under TPC-C for MySQL [77] with adjusted configuration (Section 3), PostgreSQL [86], and two SOTA mechanisms, *WAR* [5] and *LRU-C* [61], as thread count increases. MySQL throughput increases 4.89× from 1 to 8 threads but falls sharply by 3.32× at 32 threads, remaining low to 128. PostgreSQL

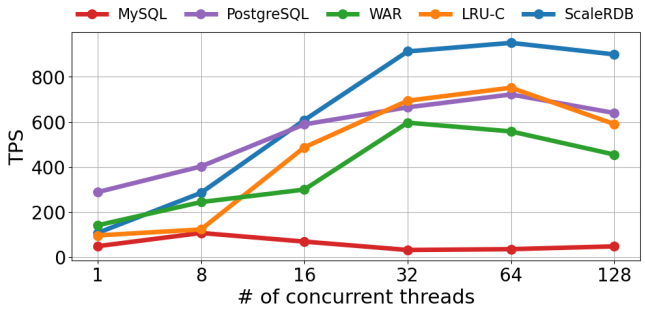


Figure 1: Throughput under TPC-C with MySQL, PostgreSQL, SOTA (*WAR* and *LRU-C*), and ScaleRDB (Proposed).

throughput rises 1.79× to 32 threads but declines by 1.25× at 128. Both *WAR* and *LRU-C* demonstrate gains of 3.93× and 7.16× up to 32 threads but plateau with additional threads. These trends underscore contention and synchronization overhead growing faster than concurrency gains [48, 108], highlighting two key scalability challenges in current database designs.

Inefficient Utilization of Cores: Efficiently utilizing multicores in relational databases presents significant challenges due to the inherent complexity of managing concurrency and resource allocation across multiple cores. Prior research [10, 95, 97] has identified that complex tasks, such as query analysis, execution, logging, and transaction handling, resist straightforward parallelization. Maintaining ACID guarantees further complicates the scheduling and ordering of concurrent threads. Recent studies [38, 93, 120] have advanced scalable methods for focused functions such as the join operation. These specialized solutions underscore the ongoing challenge of achieving general efficiency across diverse operations of relational databases in manycore systems.

Locking in Shared Data Structures: Database systems allow concurrent thread access to shared data, protecting consistency via locks such as spinlocks or mutexes on tables, rows, indexes, and logs. However, as demonstrated by prior studies [13, 21, 56, 126], locking introduces performance degradation. Although fine-grained locking can enhance scalability, recent research tends to optimize individual structures (e.g., transactions, tables, logs), underscoring the difficulty of efficient locking in complex shared data frameworks.

Many prior works, detailed in Table 1, improve database performance on manycore systems by advancing transaction management, concurrency control in shared data, and core utilization. These primarily target specialized systems such as graph databases [21, 38, 126], key-value stores [13, 56], and applications

Table 1: Categories and comparison with previous studies (Scope: Deployment Scope, TD: Transaction Distribution, SS: Shared Data Structure, IU: Idle Core Utilization).

Study	Target Database	Scope	TD	SS	IU
Shanbhag *et al.* [95]	Relational DB	Single	✓		✓
Sirin *et al.* [97]	HTAP DB	Single			✓
Fuchs *et al.* [38]	GraphDB	Single		✓	✓
Rui *et al.* [93]	Relational DB	Single	✓		✓
Zhang *et al.* [120]	OpenMLDB	Single			✓
Cai *et al.* [13]	Key-value DB	Single	✓	✓	
Cheng *et al.* [21]	GraphDB	Single		✓	
Kim *et al.* [56]	Key-value DB	Single		✓	✓
Zhi *et al.* [126]	GraphDB	Single		✓	
Lee *et al.* [61]	Storage engine in RDB	Single		✓	✓
An *et al.* [5]	Storage engine in RDB	Single		✓	✓
Kallman *et al.* [50]	Relational DB	Distributed	✓		
Michael *et al.* [100]	Relational DB	Distributed	✓		
ScaleRDB	Relational DB	Single	✓	✓	✓

in machine learning and IoT [114, 120]. Our work targets relational monolithic databases on single manycore servers, where achieving robust end-to-end scalability remains a significant challenge. While distributed relational databases such as VoltDB [100] and H-Store [50] scale efficiently across clusters and can be adapted to manycore systems to improve scalability, their network-centric designs and coordination protocols introduce overhead that limits performance within a single manycore node. Recent studies on monolithic relational databases [5, 61] focus on optimizing specific components, such as storage engine I/O by partitioning the LRU list and buffering dirty pages to reduce contention. However, these optimizations address isolated components and do not resolve key scalability bottlenecks beyond those specific layers, which are critical for achieving robust, end-to-end scalability. In contrast, our research addresses robust end-to-end scalability in manycore monolithic relational databases by enabling parallel, independent management across full system layers, overcoming low core utilization through multiple instance deployment within a single server.

In this paper, we propose ScaleRDB, a novel database system for manycore systems. Our key idea is to leverage an independent yet cooperative design, realizing shared-nothing and high concurrency and parallelism. To achieve this, ScaleRDB introduces three lightweight mechanisms. First, its *Tx coordinator* distributes transactions for parallel processing by multiple instances. The coordinator in the master uses a hash-based scheduler to assign each transaction to multiple slaves based on the data pages involved, enabling execution without cross-instance conflicts. Second, a two-level asynchronous checkpoint mechanism maintains consistency with minimal synchronization overhead. Each slave independently persistent state of a transaction via Local Checkpoints (LCPs), while the master asynchronously creates a Global Checkpoint (GCP) by aggregating these LCPs, separating global consistency from local transaction processing. Finally, a localized I/O approach with per-instance logging enables parallel disk access to minimize I/O stalls. Each slave writes to its own Write-Ahead Log (WAL) on shared storage, removing the I/O serialization bottleneck of monolithic systems that use a single log file.

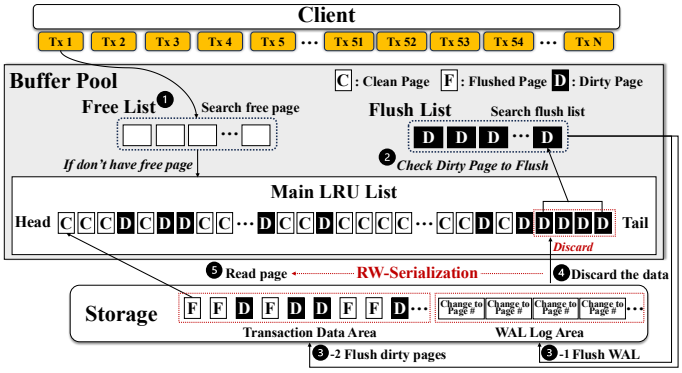


Figure 2: Buffer Management and I/O Operation.

Our evaluation using TPC-C and YCSB benchmarks shows that ScaleRDB significantly outperforms MySQL and PostgreSQL, improving transaction throughput by up to 28.02× and 1.88×, and operation throughput by up to 64.97× and 1.90×, respectively, while reducing the 99th-percentile latency by up to 99.68% and 85.04%. Furthermore, compared to state-of-the-art database optimizations, *WAR* [5] and *LRU-C* [61], ScaleRDB improves operation throughput by up to 5.43× and 2.53×, respectively. To the best of our knowledge, this is the first work that implements a shared-nothing on shared-memory distributed architecture in a single manycore system. It achieves end-to-end scalability by addressing the limitations of monolithic relational databases, eliminating global contention and I/O serialization while enabling parallel transaction processing through lightweight coordination and transaction distribution. ScaleRDB is publicly available at https://github.com/ScaleRDB/ScaleRDB.

2 Background

2.1 Bottlenecks in Database I/O Path

Since the essence of a database lies in storing and fetching data, the efficient utilization of memory tiers such as DRAM and storage is critical. To achieve this, database systems employ buffer caches to reduce the overhead of disk access by keeping recently used data in limited, high-speed memory. The buffer cache is typically managed by a Least Recently Used (LRU) list. Because every transaction references this list, it requires lock-based synchronization, which creates a major bottleneck under concurrent access from multiple threads.

Figure 2 shows the buffer management architecture in MySQL. As depicted, the buffer cache maintains pages in the free list, flush list, and main LRU list. Each list is protected by a global mutex, and under high concurrency, these mutexes can cause contention, serialization, and reduced throughput [4, 5, 37, 49, 61, 62, 64, 112, 118]. Consequently, this design introduces three primary bottlenecks: (1) serialized access to the centralized LRU list, (2) contention on the flush list, and (3) stalls caused by Write-Ahead Logging (WAL) dependencies.

Centralized LRU List Access: As shown in the figure, searching for a clean page requires a full iteration of the LRU list (❷). This

operation acquires the global mutex protecting the centralized LRU list to prevent concurrent access. Additionally, moving or removing a page also requires acquiring the same mutex. Consequently, all threads contend for this lock, creating a serialized execution path that limits scalability under high concurrency.

Prior studies have addressed this issue by exploring fine-grained or partitioned LRU lists and lock-free data structures to reduce contention [3–5, 61, 62, 112]. However, these methods present drawbacks: they increase maintenance overhead, do not fully resolve the complexity of synchronizing a large shared structure, and still require coordination for global operations such as page promotion and eviction.

Flush List Contention: As shown in the figure, when no clean page is available (❷), the database selects a dirty page from the tail of the LRU list and initiates a flush operation via the flush list. The associated WAL records are first written to the WAL log area to guarantee durability (❸-1), followed by flushing the dirty page to the transaction data area (❸-2). While the LRU list manages candidates for page replacement, the flush list separately tracks dirty pages pending writeback to storage. Although disk I/O for flushing individual pages can occur in parallel, the flush list is guarded by a global mutex. This lock serializes all modifications on the flush list, creating a bottleneck that impairs overall parallelism. The resulting serialization delays allocation of clean pages and stalls foreground transactions under write-heavy workloads (❹).

To alleviate this bottleneck, prior works have proposed various solutions spanning the system stack, including I/O path parallelization within the buffer manager [3, 5, 61], novel hardware I/O commands to reduce flush latency, and offloading flush operations to remote memory tiers to remove slow I/O from critical paths [4, 62, 83, 92, 101, 123]. Nevertheless, these approaches often require specialized hardware support or introduce considerable architectural complexity.

WAL-Dependent Eviction Delays: As shown in the figure, before a dirty page can be evicted, the corresponding Write-Ahead Log (WAL) records up to the LSN of the page must be flushed to persistent storage to ensure durability (❸-1). Page eviction depends on WAL persistence, so any delay in flushing the WAL, such as *fsync* contention on the WAL file, stalls eviction until all necessary log records are safely persisted (❹). Once the WAL file is persisted, the dirty page is written to the operating system page cache using *pwrite*, and the kernel subsequently flushes this data to disk asynchronously.

To address this flush overhead, some research eliminates the need to persist log buffers directly in non-volatile memory [4, 33, 53, 58, 62]. Other approaches reduce synchronous I/O and improve multicore scalability by employing lock-free logging mechanisms that mitigate contention in centralized log buffers [45, 56, 79, 80, 109, 113]. However, these solutions often require specialized hardware support and focus narrowly on optimizing the logging component, leaving other database subsystems unoptimized and limiting holistic performance gains.

2.2 Database for Manycore System

Due to hardware limitations, server systems have evolved from single-core to multicore and now to manycore architectures [15, 39, 72, 73, 85, 105]. Manycore systems contain a large number of distributed CPUs across multiple physical sockets, enabling extreme parallelism [10, 24, 35, 82, 122]. This architectural complexity demands new software designs that can efficiently exploit parallelism while managing inter-core communication and synchronization overhead. Existing software, however, faces substantial challenges in parallelization and synchronization, limiting scalability on modern manycore hardware. For example, conventional relational databases rely on coarse-grained locking to protect shared data and critical sections. This blocks concurrent access to shared data structures by multiple processes running on different CPUs. As a result, it causes severe lock contention, which has become a major bottleneck in database performance on manycore systems [13, 18, 38, 56].

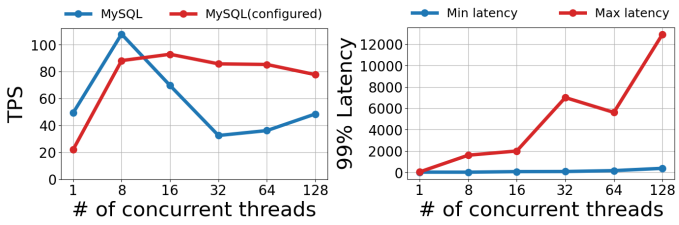
Prior work has primarily targeted optimizing the internals of a database system on manycore hardware within a single node. Techniques such as fine-grained locking, lock-free data structures [3, 5, 13, 38, 56, 61, 62, 112], and NUMA-aware optimizations [43, 64, 67, 68, 107] aim to reduce lock contention and minimize global synchronization. While these methods improve scalability, shared resources still necessitate centralized coordination, which limits scalability. Other efforts have explored scaling databases across multiple nodes [8, 20, 50, 60, 69, 74, 100, 116, 124]. By distributing transactions across nodes, they reduce contention within a single server and improve aggregate throughput. However, distributed architectures introduce added complexity in data partitioning, transaction coordination, and ensuring consistency under high concurrency.

Table 2. Adjusted MySQL Configuration Parameters.

Category	Parameter	Default	Modified
Concurrency	innodb_thread_concurrency	0	32
	innodb_buffer_pool_instances	8	16
	innodb_read_io_threads	4	8
	innodb_write_io_threads	4	8
	innodb_lru_scan_depth	1024	4096
Flush efficiency	innodb_flush_neighbors	1	0
	innodb_io_capacity	200	2000
	innodb_io_capacity_max	2000	6000
	innodb_max_dirty_pages_pct	30	75
	innodb_max_dirty_pages_pct_lwm	0	10

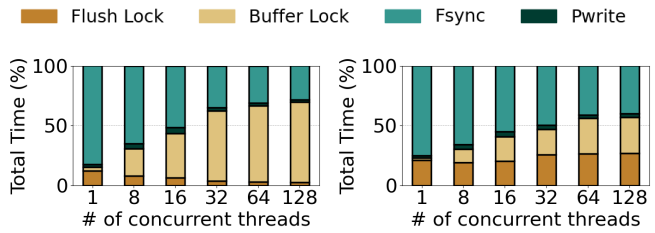
3 Motivational Evaluation

To identify the performance bottlenecks discussed in Sections 2.1 and 2.2, we conduct a motivational evaluation using MySQL running the TPC-C benchmark on a manycore system, with the experimental setup described in Section 6. Since previous works have demonstrated that MySQL with its default configuration exhibits extremely low performance, we modified several InnoDB parameters for a fair evaluation. We use this modified configuration throughout the paper, following similar practices as the studies for *WAR* [5] and *LRU-C* [61]. The modified configurations include parameters for thread concurrency and flush efficiency, as detailed in Table 2.



(a) TPS. (b) Latency-MySQL(Configured).

Figure 3: Limited scalability of MySQL.



(a) Buf 5%. (b) Buf 50%.

Figure 4: Lock contention and I/O operation breakdown under different buffer pool sizes in MySQL.

3.1 Limited Scalability

Figure 3 shows the throughput per second (TPS) and 99% tail latency as the number of threads increases. As shown in Figure 3a, the modified configuration maintains higher TPS with increasing concurrent threads. While the default configuration performs better at 8 threads due to lower contention, from 16 threads onward the modified configuration consistently outperforms it, with improvements ranging from 1.33× to 2.63×. In both configurations, performance peaks at a low thread count and declines as thread count rises. Limited scalability is also evident in latency, as seen in Figure 3b. Although TPS drops less sharply for the modified configuration, 99% latency rises steeply with thread count, reaching a maximum of 12,924.72 ms at 128 threads compared to 51.21 ms with a single thread. These results highlight the complex synchronization in database operations and confirm that the current database is not optimized to scale on manycore systems.

3.2 Lock Contention and I/O Stall

Figure 4 presents the runtime breakdown for different buffer pool sizes, set to 5% and 50% of the total database size, as the number of concurrent threads increases. We analyze the relative time spent in four key components: *Flush lock*, *Buffer Lock*, *Fsync*, and *Pwrite*.

Lock Contention: For the 5% buffer pool (Figure 4a), overhead from the *Buffer Lock* increases sharply as the number of threads grows, rising from 3.16% of total runtime with 1 thread to 63.85% at 64 threads. This increase results from frequent page replacements caused by the limited buffer pool size, which creates contention on the global LRU mutex. The contention intensifies when the victim page is dirty, as it must be added to the flush list while holding the global LRU mutex, thereby extending the lock duration. In contrast, with the 50% buffer pool (Figure 4b), *Buffer Lock* overhead remains low, while *Flush lock* overhead increases with more threads. A larger buffer pool holds more clean pages, reducing replacements. However, dirty pages accumulate before flushing, resulting in higher contention on the global flush mutex. The *Flush lock* overhead grows from 13.10% to 32.76% at 64 threads, indicating that global flush mutex contention becomes higher than that of the global LRU mutex.

I/O Stall: The two key I/O operations described in Section 2.1, *Fsync* (for WAL persistence) and *Pwrite* (for dirty page writes), exhibit distinct behaviors. The time spent in *Fsync* increases sharply with the number of threads, as synchronous flushes to storage become a bottleneck that blocks more threads under higher concurrency. At 128 threads, *Fsync* accounts for 41.04% of total runtime in the 5% buffer pool case and 52.16% in the 50% buffer pool case, making it the dominant bottleneck. In contrast, *Pwrite* writes data to the OS page cache and is handled asynchronously by kernel threads, resulting in stable overhead regardless of thread count or buffer pool size.

In summary, a small buffer pool size causes frequent evictions that increase contention on the global LRU mutex, while a large buffer pool size accumulates more dirty pages, leading to higher contention on the global flush mutex and increased I/O synchronization overhead.

4 ScaleRDB Design

We propose ScaleRDB, a scalable relational database for manycore systems that adopts a single-master, multi-slave architecture with a shared-nothing design on a shared-memory single node to overcome lock contention and I/O stalls. It overcomes lock contention by simplifying ordering and synchronization via a single master and leverages transaction distribution and parallel processing across database instances to reduce I/O stalls.

ScaleRDB is designed using the following three key strategies:

- **Strategy #1:** ScaleRDB *adopts a shared-nothing architecture on shared memory and handles transactions using a single-master, multiple slave instances within a single manycore server. Each database instance is pinned to dedicated CPU cores and memory regions to minimize cross-instance interference.*
- **Strategy #2:** ScaleRDB *addresses lock contention by employing a single master to simplify ordering and synchronization while partitioning transactions for independent processing by slave instances. During data modification, instances acquire row-level locks scoped to their own data, preventing global lock contention.*
- **Strategy #3:** ScaleRDB *maintains separate WALs for distributed transaction data within each instance, alleviating I/O stalls through parallel I/O. To ensure consistency, the master routinely verifies the persistent state of all distributed instances and exclusively creates the global metadata.*

Together, these strategies address key scalability bottlenecks in manycore systems. Unlike approaches that optimize individual components, such as *WAR* [5] for logging or *LRU-C* [61] for eviction,

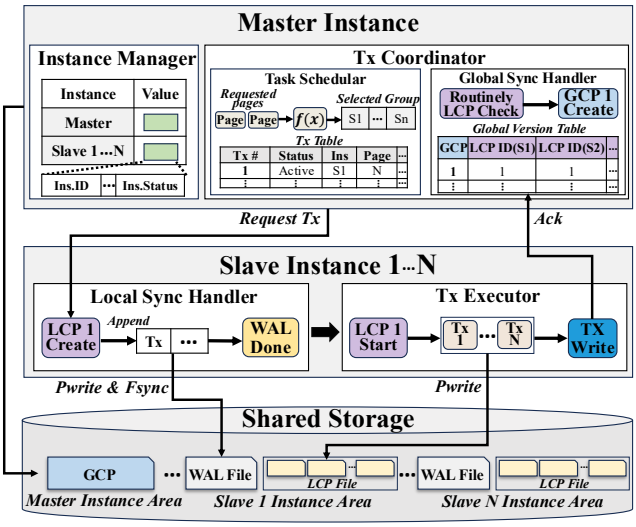


Figure 5: Overall Architecture of ScaleRDB.

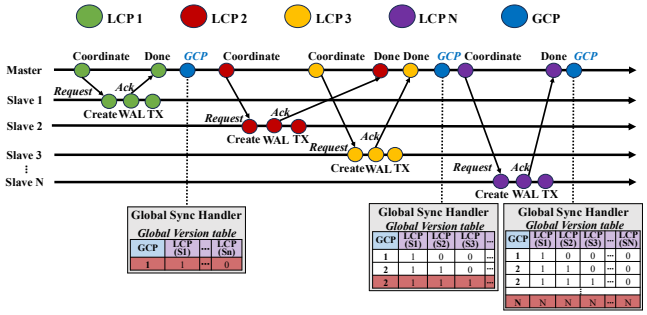


Figure 6: Timeline of Local and Global checkpoints (S#: Slave ID, GCP: Global Checkpoint, LCP: Local Checkpoint, WAL: WAL Flush, TX: Transaction Data Write).

ScaleRDB integrates architectural, concurrency, and I/O optimizations into a unified design. In addition, it avoids deployment complexities faced by fully distributed databases such as *VoltDB* [100], which encounter challenges in data partitioning, transaction coordination, and network overhead, making them ill-suited for single manycore nodes. By leveraging a single-node, shared-memory architecture, ScaleRDB balances scalability with operational simplicity, making it well-suited for modern manycore hardware.

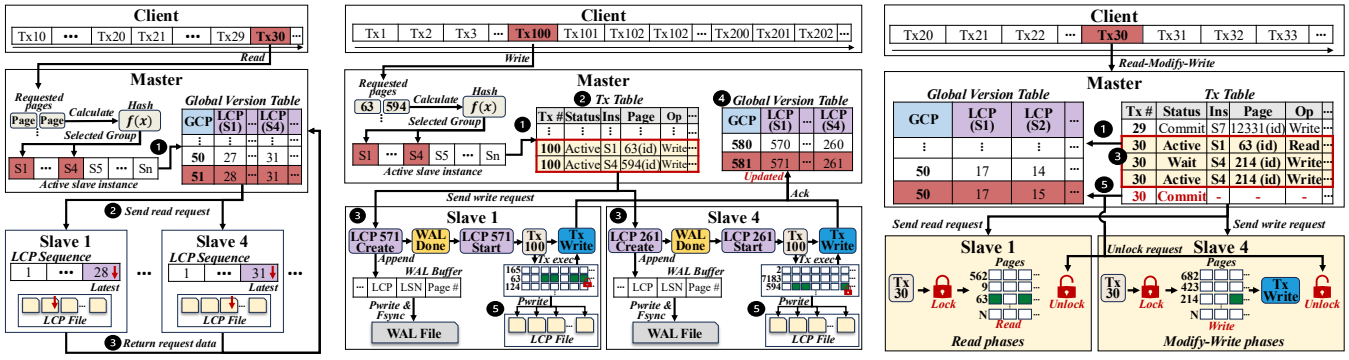
4.1 Overall Architecture

Figure 5 shows the architecture of ScaleRDB. As shown in the figure, it consists of a single master and multiple slave instances.
Master Instance: The master exclusively handles transaction distribution, coordination, ordering, and global database synchronization. ScaleRDB chooses single coordination over multiprocessing coordination in traditional RDBs or multiple coordination strategies used in distributed databases. Multiprocessing coordination is inherently complex, involving multiple processes contending for shared resources such as flush and buffer lists, and scales only within limited single components. Likewise, multiple coordination relying on heavyweight distributed consensus faces scalability ceilings. In contrast, ScaleRDB employs a lightweight single-master coordinator coupled with a two-level asynchronous checkpointing scheme on shared storage. Despite using single-master coordination, ScaleRDB sacrifices the simplicity of a fully shared global state to achieve massive parallelism through architectural partitioning. This practical trade-off balances scalability and complexity by avoiding challenges inherent in both multiprocessing coordination and heavyweight distributed consensus. Decoupling coordination overhead from I/O stalls is critical to maintaining high throughput, as it frees transaction processing from being blocked by synchronous I/O operations.

The master comprises three key components: *Instance Manager*, *Tx Coordinator*, and *Global Sync Handler*. First, *Instance Manager* maintains metadata for both the master and slave instances, including identifiers (Instance ID, Data File ID, Log File ID) and instance status (e.g., active or passive). Second, *Tx Coordinator*, composed of *Task Scheduler* and *Global Sync Handler*, schedules transaction processing across active instances and manages global synchronization. *Task Scheduler* maps transactions to instances using a hash function (*f(x)*) on requested pages, identifies the responsible slaves, and records detailed entries for each transaction in `Tx Table`. This entry contains the transaction ID, execution status, and participating active instance IDs with their associated locks (e.g., Page-Row ID) to enforce concurrency control. The master sends processing requests to all involved slaves and waits for acknowledgments before finalizing the transaction by updating its status to committed in `Tx Table`. Third, *Global Sync Handler* regularly monitors instance states and creates global checkpoints (GCPs) representing the persistent database state. A GCP aggregates metadata from Local Checkpoints (LCPs) of all slaves, which track pages processed by transactions. After the GCP is created, `Global Version Table` is updated to map each GCP ID to the latest LCP ID across instances.
Slave Instance: Each slave processes transactions assigned by the master via WAL and LCP writes. Because transactions are pre-coordinated by the master, slaves execute them independently and in parallel. Each slave consists of *Local Sync Handler* and *Tx Executor* coordinating transaction processing with the master. *Local Sync Handler* receives assigned transactions, creates Local Checkpoints (LCPs), appends log records to WAL buffer, and flushes WAL to ensure durability. *Tx Executor* performs parallel writes to in-memory buffer pages. After both WAL flush and in-memory writes complete, the slave sends an acknowledgment to the master confirming LCP finalization. Dirty pages are then asynchronously flushed to storage.

4.2 Scalability with Two-Level Asynchronous Checkpointing

In ScaleRDB, transaction consistency and scalability are ensured through a two-level checkpointing mechanism. Each slave creates a local checkpoint (LCP) to persist its part of a transaction, while the



(a) read. (b) write. (c) read-modify-write.

Figure 7: Procedure of Concurrency Control with Different Operations: read, write, read-modify-write.

master periodically creates a global checkpoint (GCP) that aggregates the most recent LCPs from all instances to capture the entire database state.

Decentralized LCP Management: As shown in Figure 6, the master initiates a transaction by selecting a single slave to perform the required WAL and transaction data writes. This avoids broadcasting to all slaves, which would be inefficient on shared storage due to multiple writes of the same data adding overhead without improving reliability. The designated slave then processes the transaction in two phases: *Local Sync Handler* first handles the WAL flush, followed by *Tx Executor* applying the requested transaction data per local checkpoint (i.e., LCP 1 - Slave 1, LCP N - Slave N). Once the local checkpoint is completed, the slave sends an acknowledgment (Ack) to the master, finalizing the local checkpoint process.

`ScaleRDB` introduces three main contributions through the local checkpoint (LCP) mechanism. 1) It writes LCPs only once via a designated slave instead of broadcasting to all instances, reducing redundant I/O overhead on shared storage and eliminating unnecessary coordination while preserving crash consistency. 2) It enables pipelined LCP processing for increased parallelism by distributing independent LCPs to designated slaves, allowing the master to coordinate subsequent transaction requests without waiting for prior completions and letting slaves process their LCPs concurrently without synchronization. 3) It writes LCPs to shared storage, which allows universal access for all instances and lays a foundation for robust crash recovery, enabling seamless master failover by promoting another slave and reassignment of failed slave workloads to other instances.

Global Persistence with GCP: In `ScaleRDB`, the master periodically (e.g., every 2000 milliseconds) creates a global checkpoint (GCP) to ensure consistency across local checkpoints. This process extends the persistent state of all slaves, represented by their local checkpoints (LCPs), into a unified global database snapshot (the GCP) by tracking the latest state of each instance and persisting their combined metadata. As shown in Figure 6, the master first creates an initial global checkpoint (GCP 1) using the existing local checkpoint data (LCP 1). Subsequent global checkpoints (e.g., GCP 2) aggregate completed local checkpoints (LCP 1, LCP 2, LCP 3, etc.) to maintain a consistent state across the system. Once all transactions are complete, a final global checkpoint (GCP N) consolidates information from all persistent local checkpoints (LCP 1 through LCP N).

`ScaleRDB` introduces two main contributions through the global checkpoint (GCP) mechanism. 1) In`ScaleRDB`, it increases GCP efficiency by having a single slave write each LCP instead of broadcasting to all slaves, which eliminates redundant data and reduces the number of checkpoints the GCP must manage. 2) It enables non-blocking slave progress by allowing the master to create the GCP independently. The master reads LCPs directly from shared storage rather than requesting them from slaves.

4.3 Transaction Processing with Concurrency Control

Figure 7 shows the procedures for each transaction operation type. For read transactions, `ScaleRDB` employs a lock-free mechanism over global snapshots (GCPs), allowing non-blocking retrieval of up-to-date data. For write and read-modify-write transactions, it enforces exclusive row-level locks on all target rows until commit or abort to ensure atomicity and isolation.

Read Transaction: As shown in Figure 7a, *Task Scheduler* in the master first hashes ($f(x)$) the requested pages to locate the relevant slaves (e.g., S1 and S4). The master then references the `Global Version Table` to find the most recent data version among the identified slaves (e.g., S1 - LCP 28, S4 - LCP 31) (❶). This process ensures that read operations use a globally consistent snapshot, because the latest GCP in the table points to the correct LCPs. In addition, to maintain this consistency without blocking, the system uses a lock-free update mechanism for the table. When the master updates the `Global Version Table`, it forks the table and atomically swaps in the new version. This method prevents partial updates and allows only selected slaves to respond to read requests without contention.

To perform an actual read operation, the master sends the read request to the slaves that have the required LCPs for the current GCP (e.g., GCP 51 requires S1 - LCP 28 and S4 - LCP 31) (❷). Upon receiving a request from the master, each selected slave refers to

its local LCP sequence and then reads the target data from the corresponding LCP file location. Then, the requested data is sent to the master, which combines the partial results into a globally consistent snapshot and returns it to the client (❸).

Write Transaction: As shown in Figure 7b, for a write transaction (e.g., Tx 100), *Task Scheduler* in the master first hashes (*f(x)*) the target pages to identify the owner slaves (e.g., S1 and S4) (❶). Then, it records the transaction entry, containing the identified instances and pages, into `Tx Table` to track its progress (❷). The master then sends a request to these designated slaves to process the transaction.

To perform an actual write operation, the master sends the write request to the slaves. If a write transaction spans multiple slaves (i.e., Tx 100 on S1 and S4), the master instructs each relevant slave to create a new LCP (e.g., S1 - LCP 571, S4 - LCP 261). Then, each designated slave processes its portion of the transaction in parallel, which includes flushing its WAL and updating the in-memory pages under a row-level lock (e.g., Slave 1 – LCP 571 for Page 63, Slave 4 – LCP 261 for Page 594) (❸). After receiving acknowledgments from all participating slaves, the master updates `Global Version Table` to finalize the new GCP (e.g., GCP 581) and commits the transaction (❹). Concurrently, updated dirty pages are persisted to each slave LCP File asynchronously (❺). LCP files are managed in an append-only manner, with a background garbage collection process that periodically removes outdated pages.

Read-Modify-Write Transaction: As shown in Figure 7c, when a client requests a read-modify-write transaction (e.g., Tx 30), the master records it in `Tx Table`: transaction ID, target pages, operation type, and assigned instances. It references `Global Version Table` to find the slave holding the source data (e.g., Slave 1 with Page 63) and sends the read request (❶).

The execution proceeds in two phases. In the read phase, the designated slave (e.g., Slave 1) acquires a row-level lock on the source page, reads the data, and returns it to the master (❷). Upon receiving the data, the master updates `Tx Table` to mark the transaction status as waiting for the modify-write phase. The slave (e.g., Slave 1) holds a row-level lock on the source page to prevent large table-level serialization (❸). To ensure consistency, this lock is not released until the transaction is committed for consistency. In the modify-write phase, the master sends the write request with the read source data to the target slave (e.g., Slave 4). The slave then locks its target page (e.g., Page 214), writes the data, and sends an acknowledgment (❹). After receiving the acknowledgment for the write, the master commits the transaction, updates `Global Version Table` with the new LCP (i.e., LCP 15), and instructs both slaves (e.g., Slave 1, Slave 4) to release their row-level lock of the committed page (❺). This design, where each instance manages its own locks, avoids the bottleneck of a centralized lock manager found in monolithic systems.

4.4 `ScaleRDB` implementation

We implemented `ScaleRDB` based on the NDB engine of MySQL NDB Cluster 7.4.10, a widely used distributed database for managing distributed nodes and storage [78]. Modifications were primarily concentrated in the Distributed Data Handler (Dbdih) module, responsible for deploying database instances and managing global and local checkpoints. Although the code changes were minimal (fewer than 300 lines), they fundamentally transformed the central coordination model of `ScaleRDB` into a shared-nothing on shared-memory design. This was achieved by reorganizing the responsibilities of the Dbdih module from centralized control to distributed instance orchestration and redesigning checkpoint handling to support independent transaction processing under a lightweight global coordinator. The three primary modifications are as follows: 1) The transaction distribution algorithm is modified to make the master a central coordinator. A new data structure allows the master to assign transactions to specific slaves, avoiding inefficient broadcasts. 2) Transaction processing is updated to support concurrent, independent local checkpoints (LCPs). Each slave now generates and flushes its own LCP, enabling parallelism and reducing latency. 3) The communication protocol is changed to a direct request-acknowledgment model. The master now communicates only with assigned slaves and waits for their specific responses, simplifying recovery.

5 Limitation and Future work

Uniform Resource Allocation: Although `ScaleRDB` achieves higher CPU utilization than existing databases, its performance scalability is limited as CPU utilization plateaus beyond a certain point. This limitation primarily arises from: (1) reduced per-instance memory allocation with an increasing number of instances, which deteriorates performance in memory-intensive tasks; and (2) the presence of workload skew introduced by heavy transactions that disrupt balanced resource distribution. To overcome these issues, adopting workload-aware dynamic pooling of memory and CPU resources appears essential [25, 81]. The master node can continuously monitor transaction characteristics and instance states, enabling dynamic reallocation of resources to maintain balance [81]. Furthermore, transaction distribution strategies that consider both workload skew and historical local checkpoint progress (LCP) of slaves may further alleviate resource imbalance [34].

Coordination with Single-Master: In `ScaleRDB`, the master node coordinates all transactions to ensure concurrency control and data consistency. This centralized coordination eliminates the bottleneck of lock contention, as illustrated in Figure 4, by delegating control to a single process. However, this architecture introduces a single point of failure: if the master fails, the entire database system is compromised. To mitigate this risk, adopting a decentralized coordination model coupled with novel concurrency control mechanisms is essential. For instance, leveraging timestamp-based consistency models [47, 52, 54, 60, 90, 102] allows for eventual consistency with relaxed synchronization constraints, balancing performance and correctness. Furthermore, exploring multi-master architectures [30, 66] can provide resilience and improved resource utilization. In the context of a manycore system, where coordination domains are comparatively small, deploying multiple masters—potentially one per NUMA node—may enhance coordination efficiency and memory locality.

6 Evaluation

6.1 Experimental setup

Table 3. List of various workloads.

Workload	R/W Ratio(%)	Complexity	Skewness
TPC-C	92/8	High	Realistic
YCSB-Workload A	50/50, 20/80, 80/20	Low	Realistic, Extreme
YCSB-Workload B	95/5	Low	Realistic, Extreme
YCSB-Workload F	50/50 (RMW)	High	Realistic, Extreme

Table 4. List of evaluated database systems.

Database	Type	Architecture	Operation	Version
MySQL [77]	Relation	Monolithic	Disk-based	5.6
WAR [5] (MySQL)	Relation	Monolithic	Disk-based	5.7
LRU-C [61] (MySQL)	Relation	Monolithic	Disk-based	5.6
PostgreSQL [86]	Relation	Monolithic	Disk-based	16.2
MongoDB [76]	NoSQL	Non-monolithic	Disk-based	7.0
VoltDB [100]	Relation	Non-monolithic	In-memory	14.4
ScaleRDB	Relation	Monolithic	Disk-based	Prototype

The evaluation runs on a server with AMD EPYC 7713 (64 cores, 128 threads, 2.0GHz), 45 GB DDR4 memory, a 500 GB Samsung 870 Pro SATA SSD, and Ubuntu 22.04.1 LTS (Linux 6.5.0-25-generic).

Benchmarks: We employ TPC-C [104] (25 GB database, 250 warehouses) for OLTP and YCSB [26] workloads A, B, and F. Skew is evaluated from realistic to extreme: realistic skew via TPC-C (New-Order and Payment dominant) and YCSB using Zipfian from *ZipfianGenerator* ($\theta \approx 0.99$). Extreme skew via YCSB hotspot by tuning the hot data fraction and the hot access fraction. A summary appears in Table 3.

Evaluated Databases: For comparison, we use MySQL 5.6.28 [77] with tuned settings for a fair baseline (In Section 3) and PostgreSQL [86], a monolithic RDBMS employing MVCC and group commit for high concurrency. For state-of-the-art, we evaluate *WAR* [5] and *LRU-C* [61], which target lock contention and I/O stalls in buffer management: *LRU-C* reduces contention by splitting the global buffer list mutex, while *WAR* mitigates I/O stalls via a temporary buffer that writes dirty pages (*pwrite*) asynchronously to decouple reads from slow flushes (*fsync*). These monolithic schemes form strong baselines against ScaleRDB, as all address lock contention and I/O stalls, but at the component level. We also include non-monolithic systems: *MongoDB* [76], a sharded NoSQL store, and *VoltDB* [100], an in-memory OLTP engine. Section 6.7 analyzes these systems, and Table 4 summarizes all databases.

6.2 Throughput and Latency of Monolithic Database

6.2.1 Throughput.

Figure 8 shows the throughput of MySQL, PostgreSQL, SOTA (*WAR*, *LRU-C*), and ScaleRDB with TPC-C and YCSB. We configured ScaleRDB with its optimal setup of 12 instances (1 Master and 11 Slaves), and 64 concurrent threads were distributed among these instances and pinned to dedicated cores.

Comparison with MySQL and PostgreSQL: Figure 8 shows that MySQL TPS decreases with more concurrent threads due to contention, while ScaleRDB improves TPS by 3.57×–28.06×, maintaining scalability. PostgreSQL outperforms MySQL at all thread

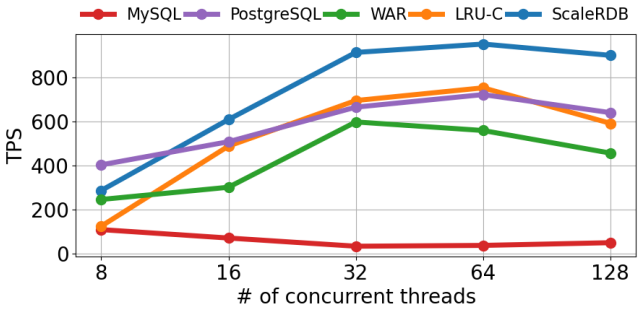


Figure 8: TPS using TPC-C.

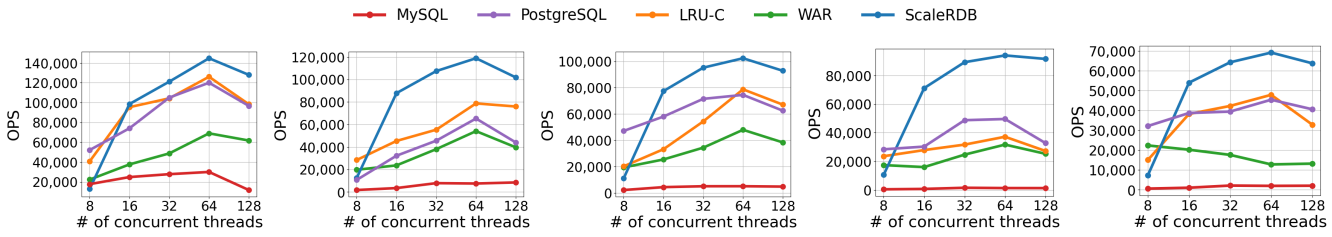
counts (e.g., 721 TPS vs. 36.17 TPS at 64 threads) thanks to better concurrency control, yet ScaleRDB still surpasses PostgreSQL by 1.32× (64 threads) and 1.40× (32 threads). This pattern also appears in YCSB (Figure 9), where ScaleRDB achieves OPS improvements of up to 110.91× and consistently outperforms PostgreSQL with 1.20×–1.88× gains across workloads.

These results highlight that MySQL suffers from centralized lock contention and synchronous WAL flushes, whereas PostgreSQL alleviates these bottlenecks through MVCC-based concurrency control and more efficient WAL handling, which explains its consistently higher throughput than MySQL. However, it still faces global contention in shared components (e.g., buffer manager, WAL writer, and transaction ID management via the ProcArray), limiting its scalability under high concurrency. The benefits of ScaleRDB vary by workload. The gains are greatest in write-intensive workloads (RW:20/80 and RMW) because the asynchronous flush and distributed logging design avoids the severe I/O stalls that bottleneck both MySQL and PostgreSQL. In read-heavy workloads (RW:95/05), the improvements are smaller but still significant, as the system eliminates global contention in shared structures and enables slaves to process reads independently, thereby improving concurrency.

Comparison with *WAR*, and *LRU-C*: As shown in Figure 8, ScaleRDB achieves 950.43 TPS at 64 threads, outperforming *WAR* and *LRU-C* by 1.70× and 1.20×, respectively. At 128 threads, it maintains 899.22 TPS, while *WAR* and *LRU-C* drop to 455.31 and 590.49 TPS, resulting in 1.98× and 1.52× improvements. In YCSB (Figure 9), ScaleRDB surpasses *WAR* and *LRU-C* by 2.97× and 2.53× in write-heavy workloads (RW:20/80), 2.14× and 1.30× in balanced workloads (RW:50/50), 5.43× and 1.44× in read-modify-write (RMW), and 2.10× and 1.15× in read-heavy workloads (RW:95/05).

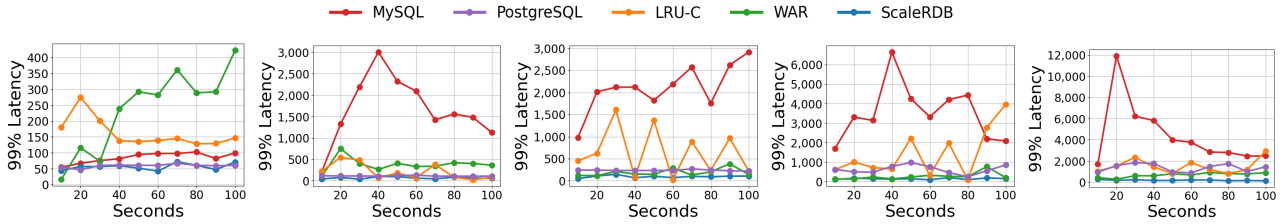
These results show the structural efficiency of ScaleRDB. This efficiency is enabled by distributing transactions across multiple database instances, which allows parallel execution without global contention. In contrast, *LRU-C* enhances I/O parallelism by prioritizing clean pages and dividing locks to mitigate contention. However, this design reduces hit ratio and still relies on a shared buffer, which leads to new bottleneck points (e.g., flush bursts and mutex serialization). Similarly, *WAR* alleviates read stalls by reversing the read-write order but still suffers from limited scalability because slow flush operations eventually accumulate, increasing latency and reducing throughput under a high number of threads.

6.2.2 Latency.



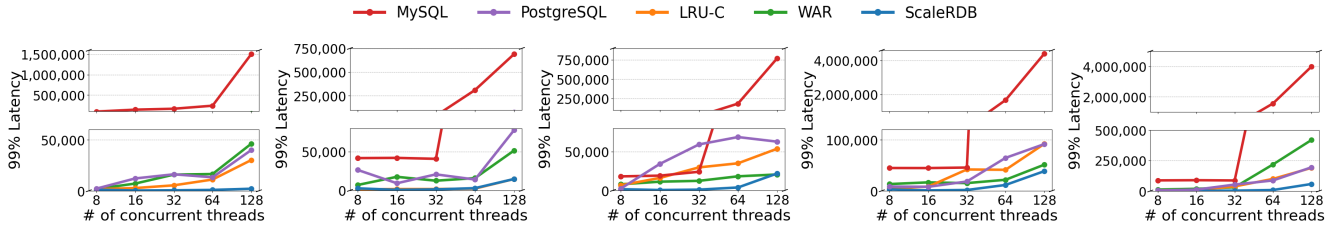
(a) R/W:95/5%. (b) R/W:80/20%. (c) R/W:50/50%. (d) R/W:20/80%. (e) R-M/W:50/50%.

Figure 9: OPS using YCSB.



(a) Thread: 8. (b) Thread: 16. (c) Thread: 32. (d) Thread: 64. (e) Thread: 128.

Figure 10: 99% Latency (ms) using TPC-C.



(a) R/W:95/5%. (b) R/W:80/20%. (c) R/W:50/50%. (d) R/W:20/80%. (e) R-M/W:50/50%.

Figure 11: 99% Latency (µs) using YCSB.

Figure 10 shows the 99% latency of MySQL, PostgreSQL, SOTA (*WAR*, *LRU-C*), and ScaleRDB with TPC-C and YCSB. We configured ScaleRDB to its optimal instance setup: 12 instances (1 Master, 11 Slaves).

Compared to MySQL and PostgreSQL: Figure 10 shows MySQL latency sharply rises with concurrency, reaching 3,522 ms at 64 threads and 4,398 ms at 128 threads. PostgreSQL lowers this to 690 ms and 631 ms, thanks to MVCC and efficient WAL processing. ScaleRDB further reduces latency to 135 ms (96.16% lower than MySQL, 80.40% lower than PostgreSQL) at 64 threads and 180 ms at 128 threads, maintaining low latency. In YCSB, it shows similar trends: ScaleRDB reduces latency by over 99.87% vs. MySQL and over 94.90% vs. PostgreSQL in write- and read-heavy workloads and maintains more than 85.04% latency reduction across workloads even at 64 threads.

MySQL shows sharp latency growth as threads increase due to serialization on centralized resources and synchronous WAL flush–induced I/O stalls. PostgreSQL lowers latency via MVCC and group commit, but still spikes under high contention because WAL writing and buffer eviction serialize in the background, causing periodic I/O stalls. In contrast, ScaleRDB contains tail latency by distributing requests across independent slaves and overlapping background flush with parallel transactions, yielding a non-blocking execution model that avoids such spikes.

Compared to *WAR*, and *LRU-C*: Figure 10 shows ScaleRDB at 135.08 ms with 64 threads versus *WAR* at 260.48 ms and *LRU-C* at 1,423.93 ms; at 128 threads, ScaleRDB holds 179.92 ms while *WAR* and *LRU-C* rise to 677.06 ms and 1,512.46 ms, indicating robust latency under high contention. In *WAR*, the temporary dirty page buffer that prevents read stalls saturates under heavy writes and stalls foreground threads until flushing completes; in *LRU-C*, frequent clean to dirty transitions cause pointer update overhead and many small fragmented flushes, both elevating latency. Even at 8 threads, *WAR* and *LRU-C* (288.27 ms, 161.48 ms) exceed ScaleRDB (55.91 ms) and MySQL (85.10 ms) because their added mechanisms

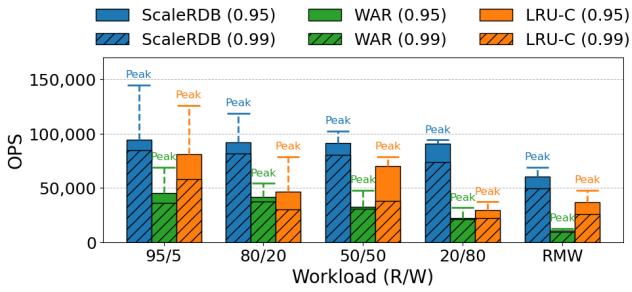
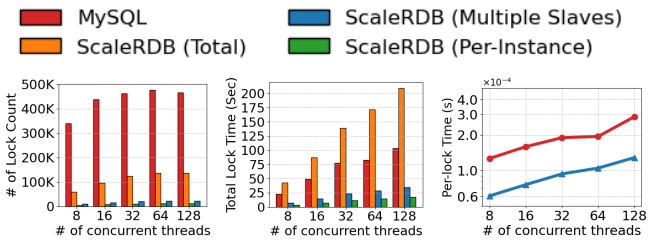
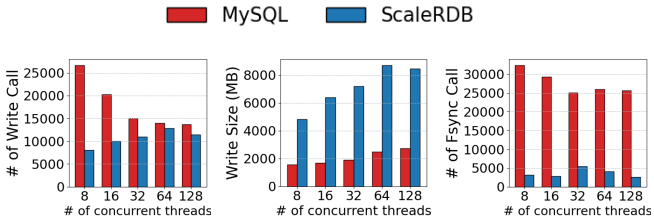


Figure 12: OPS across different read/write ratios and data skew (Legend: Hotpot data fraction, Peak: OPS with a non-skewed workload).



(a) Total lock count. **(b) Total lock time.** **(c) Per-lock time (log-scale).**

Figure 13: Lock Overhead of MySQL and ScaleRDB.



(a) Total write count. **(b) Total write size.** **(c) Total fsync count.**

Figure 14: I/O Stall of MySQL and ScaleRDB.

help mainly under contention. ScaleRDB uses single master coordination without complex shared structures, keeping low thread latency close to MySQL. In YCSB at 128 threads, ScaleRDB achieves its largest latency gains in read heavy 95/5, remains competitive in 80/20, 50/50, and RMW, and also improves 20/80, consistently outperforming *WAR* and *LRU-C* with fewer spikes.

6.3 Performance Under Data Skew

Figure 12 evaluates skewed YCSB via hotspot distributions, varying (*hotspotdatafraction*, *hotspotopnfraction*) as (0.05, 0.95) and (0.01, 0.99). The Zipfian default ("Peak") is the baseline in Section 6.2.1.

Read-Intensive: Under 95/5 with extreme skew (0.99), ScaleRDB reaches 84,615 OPS, 2.35× over *WAR* (35,965) and 1.46× over *LRU-C* (58,085); at 80/20, it sustains 81,412 OPS, 2.18× and 2.70× over *WAR* and *LRU-C*. ScaleRDB routes reads via the master *task scheduler* and a global version table, so only relevant slaves serve the single correct snapshot, avoiding global contention, whereas monolithic systems contend on shared buffer metadata.

Write-Intensive: At 20/80, ScaleRDB delivers 73,728 OPS, 3.55× and 3.33× over *WAR* and *LRU-C*, because monolithic designs serialize commits on a centralized WAL and contend on the shared buffer; ScaleRDB dispatches long I/O asynchronously across slaves, each flushing its own WAL.

Balanced-RW: At 50/50 with skew 0.99, ScaleRDB achieves 80,182 OPS, exceeding *WAR* (30,965) and *LRU-C* (38,230) by 2.59× and 2.10×; in the RMW case, ScaleRDB sustains 49,501 OPS (5.04× over *WAR*'s 9,812 and 1.89× over *LRU-C*'s 26,185), retaining over 71% of its Peak (69,077 to 49,501), while *LRU-C* drops to 55% (47,791 to 26,185). By localizing locks within slaves and eliminating global serialization, ScaleRDB outperforms component-level SOTA under hotspots.

6.4 Lock Contention and I/O Stall

Figures 13 and 14 analyze MySQL and ScaleRDB under TPC-C (64 threads, fixed 100s) to show how ScaleRDB mitigates the Section 2 bottlenecks, with MySQL using a 100% buffer pool and ScaleRDB using a 12-instance setup.

Lock Contention: As shown in Figure 13a, ScaleRDB issues far fewer locks (5.77× fewer at 8 threads, 3.43× fewer at 128 threads) and holds them for less time (103.06s vs 34.85s total at 128 threads), with shorter per-lock time (2.01× at 8 threads, 1.55× at 128 threads). MySQL contends on globally shared structures (e.g., Flush list, LRU list) guarded by a single mutex; ScaleRDB uses `Tx Table` to route to target instances, where slaves execute independently with per-instance WAL flush and row-level locks, confining contention to the touched rows and avoiding global bottlenecks.

These results show the structural efficiency of ScaleRDB. MySQL struggles under high concurrency because all threads need to compete when accessing globally shared structures, such as the Flush list and LRU buffer list, which is protected by a single mutex. In contrast, ScaleRDB avoids this bottleneck by using `Tx Table` to locate target instances. Each slave then executes transactions independently, managing its own WAL flushes and row-level locks. Although locking is still required, it is confined to the target row of the page within each instance, preventing global contention.

I/O Stall: As shown in Figure 14a, ScaleRDB issues fewer writes than MySQL at 128 threads (11,479 vs 13,674), and each call carries more data, reducing stall frequency. For example, at 64 threads the total write size is 8462.29 MB for ScaleRDB versus 2738.87 MB for MySQL (3.09×), reflecting higher throughput with fewer calls (Figure 14b). For durability, MySQL performs 25,662 fsync calls at 128 threads while ScaleRDB requires 2,585, nearly 10× fewer (Figure 14c). These results indicate that ScaleRDB sustains higher throughput by avoiding frequent small writes and fsync calls that trigger I/O stalls under high concurrency.

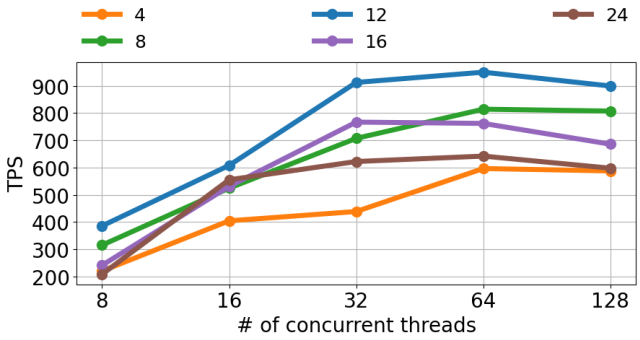
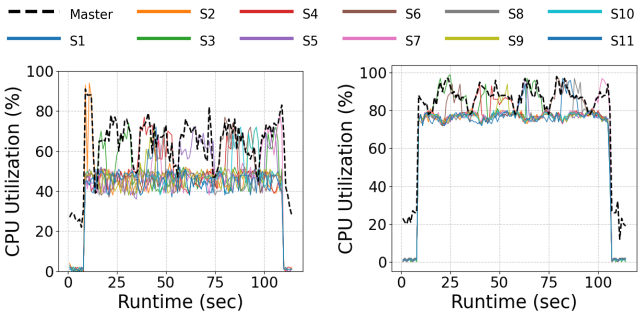


Figure 15: TPS of ScaleRDB with varying number of instances (1 Master and all slaves).



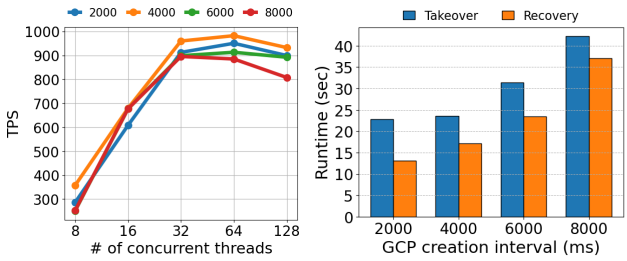
(a) Threads: 16. **(b) Threads: 64.**

Figure 16: CPU utilization of 12 Instances (1 Master and 11 Slaves) running TPC-C.

6.5 Scalability

Threads and Instance Scalability: Figure 15 reports ScaleRDB throughput on TPC-C, peaking at 950.02 TPS with 12 instances (a 1.59× gain over 4 instances) and exhibiting diminishing returns thereafter. TPS also scales with threads up to between 32 and 64 but falls at 128 for all instance counts, reflecting intra-instance contention when multiple threads share an instance. Increasing instances improves performance until 12, beyond which shared memory limits reduce per-instance capacity and amplify master coordination overhead, yielding flat or reduced TPS.

CPU Utilization: Figure 16 reports CPU usage for ScaleRDB (master and 11 slaves) on TPC-C with 16 and 64 threads. At 16 threads (Figure 16a), slaves average 43.18% and the master 62.4%, indicating underutilization; at 64 threads (Figure 16b), slaves rise to 67.22% as the master effectively distributes work, but the master averages 78.61% with peaks near 98%, revealing a coordination bottleneck. This pattern shows the strengths and limits of ScaleRDB design: slaves parallelize well, boosting throughput and CPU use, but a single coordinator caps scalability once its CPU saturates. However, we also think that moving to multi-master or multiprocessing would add substantial complexity per Section 3 without guaranteeing relief; ScaleRDB therefore opts for a simple single-master to remove lock contention on a single manycore node. Future work will explore partitioning the master role by key ranges or distributing coordination with consensus algorithms.



(a) GCP Interval. **(b) Takeover and Recovery.**

Figure 17: Overhead of global checkpoints.

6.6 Overhead of Global Checkpoints

Figure 17 shows the trade-off between throughput and recovery cost as the global checkpoint interval increases from 2,000 ms to 8,000 ms. Experiments were run after the system generated over 3,000 GCPs to ensure stable performance without warm-up effects.

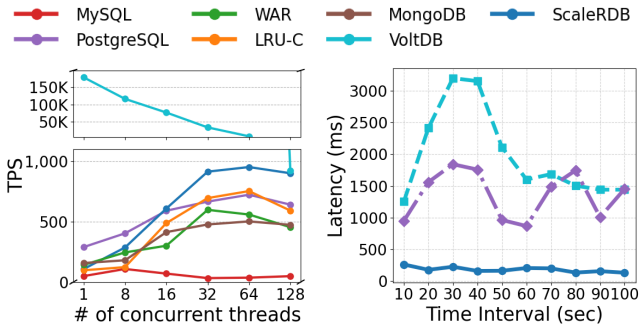
Impact of GCP Interval: As shown in Figure 17a, extending the GCP interval from 2000 ms to 4000 ms slightly improves throughput, reaching a peak of 982.18 TPS at 64 threads, a 1.03× increase over 950.43 TPS at 2000 ms due to reduced checkpoint frequency. However, increasing the interval further to 8000 ms degrades performance to 884.82 TPS, a 10% drop from the peak. This decline results from the accumulation of more dirty pages and metadata, causing heavier disk flushes and longer serialization, which increase contention with transaction processing.

Impact of Recovery: Figure 17b shows that longer GCP intervals significantly increase takeover and recovery times. Takeover, the process of switching from a failed master to an active slave instance, includes slave promotion, coordination, and data recovery, and its runtime increases 1.59× (from 22.78s at 2000 ms to 36.21s at 8000 ms). Recovery restores a single logical checkpoint (LCP) failure by replaying all transactions since the last checkpoint, causing runtime to grow 2.84× (from 13.08s to 37.14s) due to increased volume of WAL logs and LCPs, which raises both I/O (log scanning, page loading) and CPU (log parsing, redo) overhead. While takeover time increases as the interval grows, it rises more slowly than recovery time. At 8000 ms, recovery dominates the total takeover runtime, making coordination overhead negligible.

6.7 Monolithic vs. Multi-Instance Performance

Figure 18 compares throughput (TPS) and average latency at 128 threads between monolithic and non-monolithic databases, highlighting how centralized resource sharing and decentralized data partitioning affect scalability on manycore systems.

Monolithic: Figure 18a shows that monolithic systems have limited scalability due to contention for centralized resources. For example, MySQL performance collapses beyond 8 threads because of lock contention in a shared-memory model. SOTA optimizations



(a) TPS. **(b) Latency by time interval (128 threads).**

Figure 18: TPS and Latency (Avg) of various databases.

(*WAR* and *LRU-C*) improve this by addressing lock contention and I/O stalls, but still eventually decline, indicating inherent limits in centralized designs. PostgreSQL scales better than MySQL, peaking at 799.75 TPS, but contention on the buffer cache and background writer limits further growth.

Non-Monolithic: In contrast, non-monolithic systems reduce contention through distributed transaction processing, yielding superior scalability. *MongoDB* scales horizontally via sharding and document-level locking, reaching 501.15 TPS with multiple instances on a single node before hitting its limits.

VoltDB, an in-memory database, uses a single-threaded partition model to eliminate intra-partition locking, reaching 178,203 TPS on a single thread—two orders of magnitude faster by avoiding disk I/O. However, throughput collapses under multi-threading, falling to 923 TPS at 128 threads due to costly coordination for multi-partition transactions and poor scaling on synchronization-heavy queries; latency also degrades (Figure 18b), exceeding 3200 ms at 128 threads, 15.7× higher than `ScaleRDB` (203 ms) and 1.7× higher than PostgreSQL (1842 ms). In contrast, `ScaleRDB` is a disk-based, shared-nothing system on shared memory that partitions transactions across independent instances with single-master coordination, thereby avoiding global lock contention; although its peak is 950.43 TPS, it sustains superior single-node scalability with disk flushes, in contrast to *VoltDB*'s in-memory-only operation.

7 Related Works

Database for Modern Hardware: Prior work boosts parallelism either by exploiting fast storage (e.g., NVDIMM, NVMe) for workload-specific I/O paths [4, 18, 42, 62, 63, 65, 116] or by making key-value systems NUMA-aware and splitting global key spaces for concurrency [13, 43, 64, 67, 68, 70, 107]. Unlike component-level optimizations, `ScaleRDB` targets the full relational stack, harnesses idle cores, and, with a shared-nothing-on-shared-memory design, distributes workloads across a single manycore system to deliver scalable performance while preserving full RDB features.

Database for Distributed Architecture: Previous studies on distributed databases center on shared-nothing designs that scale by partitioning data and computation across independent nodes. They combine sharding, distributed query processing, and replicated high availability for scalability [8, 20, 29, 50, 69, 74, 100, 110]. Other work optimizes specific operations within this setting, including skew-resistant joins, recursive evaluation, and online schema migration [22, 27, 110, 119]. More recent studies have introduced next-generation and hybrid architectures to overcome the limitations of the traditional model, adopting deterministic execution models to reduce transaction overhead, leveraging shared-storage designs for cloud elasticity, and employing hybrid architectures to accelerate skewed workloads [14, 60, 103, 111, 124]. `ScaleRDB` departs from scale-out: it targets single-server bottlenecks by restructuring a monolithic RDBMS into shared-nothing on shared memory, eliminating global contention (e.g., centralized locking, I/O serialization) and exploiting manycore parallelism without network overhead.

Synchronization Overhead in Database: Prior work targets I/O paths, consistent WAL logging, fine-grained locking, parallel reads/writes, and reducing duplicated persistence across WAL and MySQL binlog [2, 5, 37, 46, 61]. Other work optimizes transaction-processing synchronization, index recovery, checkpoints, and locking [11, 38, 40, 41, 96]. `ScaleRDB` shares the objective but differs in approach. It achieves parallelism by restructuring the entire database into a shared-nothing-on-shared-memory architecture with multiple instances, and it eliminates the root cause of global contention by partitioning shared resources rather than merely optimizing the locks that protect them.

Application/Operation-Specific Optimizations: Prior work proposes parallelization for specific systems or tasks: graph databases exploit distinct access and traversal patterns to utilize idle cores and improve cache behavior [12, 17, 21, 87, 126]. WAL-centric studies reduce consistency overhead through refined logging paths and concurrency control [45, 51, 55, 56, 79, 80, 84, 109, 113, 125]. More recent work accelerates specific queries (e.g., joins) and targets application domains such as ML databases [7, 36, 44, 95, 120]. `ScaleRDB` shares the performance goal but differs in scope: it is application-agnostic, adapts across workloads without specialized access patterns, and, as shown by TPC-C and YCSB, sustains scalability and flexibility without per-application tuning.

8 Conclusion

In this paper, we observe that the existing relational database system shows limited scalability, especially when adapted to manycore systems where a large number of cores exist. To address this issue, we design and implement `ScaleRDB`, a database for manycore systems based on a shared-nothing on shared-memory distributed architecture. By operating multiple database instances in a single manycore machine, we optimized transaction processing to be performed independently per instance, fully utilizing resources and enhancing transaction processing parallelism. Our evaluations using OLTP real-time data processing workloads showed that `ScaleRDB` achieved higher transaction throughput by 28.06×, lower latency, and high CPU utilization compared to the existing database. Moreover, it achieves a performance increase of up to 5.43× compared to state-of-the-art database systems.

References

[1] Mohammed A Noaman Al-Hayanni, Ashur Rafiev, Fei Xia, Rishad Shafik, Alexander Romanovsky, and Alex Yakovlev. 2020. PARMA: Parallelization-aware run-time management for energy-efficient many-core systems. *IEEE*

Trans. Comput. 69, 10 (2020), 1507–1518.

[2] Adnan Alhomssi and Viktor Leis. 2023. Scalable and robust snapshot isolation for high-performance storage engines. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1426–1438.

[3] Mijin An, Soojun Im, Dawoon Jung, and Sang-Won Lee. 2022. Your read is our priority in flash storage. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1911–1923.

[4] Mijin An, Jonghyeok Park, Tianzheng Wang, Beomseok Nam, and Sang-Won Lee. 2023. NV-SQL: Boosting OLTP Performance with Non-Volatile DIMMs. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1453–1465.

[5] Mijin An, In-Yeong Song, Yong-Ho Song, and Sang-Won Lee. 2022. Avoiding read stalls on flash storage. In *Proceedings of the 2022 International Conference on Management of Data*. 1404–1417.

[6] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: the definitive guide: time to relax.* " O'Reilly Media, Inc.".

[7] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. Autosteer: Learned query optimization for any sql database. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3515–3527.

[8] Jason Arnold, Boris Glavic, and Ioan Raicu. 2019. A high-performance distributed relational database system for scalable OLAP processing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 738–748.

[9] Azure. 2025. *CosmosDB.* https://azure.microsoft.com/en-us/products/cosmos-db Accessed: 2025-09-29.

[10] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. 2022. The full story of 1000 cores: An examination of concurrency control on real (ly) large multi-socket hardware. *The VLDB Journal* 31, 6 (2022), 1185–1213.

[11] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 1–8.

[12] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. 2020. A1: A distributed in-memory graph database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 329–344.

[13] Miao Cai, Junru Shen, Yifan Yuan, Zhihao Qu, and Baoliu Ye. [n. d.]. BonsaiKV: Towards Fast, Scalable, and Persistent Key-Value Stores with Tiered, Heterogeneous Memory System. ([n. d.]).

[14] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, et al. 2022. Polardb-x: An elastic distributed relational database for cloud-native applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2859–2872.

[15] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. 2020. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys (CSUR)* 53, 4 (2020), 1–38.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[17] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-tran: A high performance distributed graph database with a decentralized architecture. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2545–2558.

[18] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 17–32. https://www.usenix.org/conference/fast21/presentation/chen-hao

[19] Kuan-Hsun Chen, Jian-Jia Chen, Florian Kriebel, Semeen Rehman, Muhammad Shafique, and Jörg Henkel. 2016. Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity. *IEEE Trans. Comput.* 65, 11 (2016), 3441–3455.

[20] Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2024. TDSQL: tencent distributed database system. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3869–3882.

[21] Audrey Cheng, Jack Waudby, Hugo Firth, Natacha Crooks, and Ion Stoica. [n. d.]. Mammoths Are Slow: The Overlooked Transactions of Graph Data. ([n. d.]).

[22] Long Cheng, Spyros Kotoulas, Tomas E Ward, and Georgios Theodoropoulos. 2014. Robust and skew-resistant parallel joins in shared-nothing systems. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. 1399–1408.

[23] Lin Cheng, Peitian Pan, Zhongyuan Zhao, Krithik Ranjan, Jack Weber, Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Preslav Ivanov, et al. 2021. A tensor processing framework for CPU-manycore heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2021), 1620–1635.

[24] Younghyun Cho, Jiyeon Park, Florian Negele, Changyeon Jo, Thomas R Gross, and Bernhard Egger. 2022. Dopia: online parallelism management for integrated CPU/GPU architectures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 32–45.

[25] Yannis Chronis. 2025. Databases in the Era of Memory-Centric Computing. In *CIDR Conference*. https://vldb.org/cidrdb/papers/2025/p6-chronis.pdf Accessed: 2025-09-28.

[26] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[27] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[28] couchbase. 2025. *couchbase.* https://www.couchbase.com/ Accessed: 2025-09-29.

[29] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed postgresql for data-intensive applications. In *Proceedings of the 2021 International Conference on Management of Data*. 2490–2502.

[30] Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, et al. 2023. Taurus MM: bringing multi-master to the cloud. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3488–3500.

[31] Till Döhmen, Radu Geacu, Madelon Hulsebos, and Sebastian Schelter. 2024. Schemapile: A large collection of relational database schemas. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–25.

[32] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[33] Rémi Dulong, Rafael Pires, Andreia Correia, Valerio Schiavoni, Pedro Ramalhete, Pascal Felber, and Gaël Thomas. 2021. NVCache: A plug-and-play NVMM-based I/O booster for legacy systems. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 186–198.

[34] Andrew Pavlo et al. 2012. Skew-aware automatic database partitioning in shared-nothing systems. In *ICDE '12 Proceedings of the 28th IEEE International Conference on Data Engineering*. 61–72. doi:10.1109/ICDE.2012.19

[35] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing* 2 (2020), 382–400.

[36] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1891–1904.

[37] Michael Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-optimized multi-version concurrency control for disk-based database systems. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2797–2810.

[38] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1173–1186.

[39] Manjari Gupta, Lava Bhargava, and S Indu. 2021. Mapping techniques in multicore processors: current and future trends. *The Journal of Supercomputing* 77, 8 (2021), 9308–9363.

[40] Michael Haubenschild and Viktor Leis. 2023. Lock-Free Buffer Managers Do Not Require Delayed Memory Reclamation. In *Proceedings of the 1st Workshop on Simplicity in Management of Data*. 1–3.

[41] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking logging, checkpoints, and recovery for high-performance storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 877–892.

[42] Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. 2023. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1712–1725.

[43] Yang Hong, Yang Zheng, Fan Yang, Bin-Yu Zang, Hai-Bing Guan, and Hai-Bo Chen. 2019. Scaling out numa-aware applications with rdma-based distributed shared memory. *Journal of Computer Science and Technology* 34, 1 (2019), 94–112.

[44] Xue-Xuan Hu, Jian-Qing Xi, and De-You Tang. 2020. Optimization for multi-join queries on the GPU. *IEEE Access* 8 (2020), 118380–118395.

[45] Kecheng Huang, Zhaoyan Shen, Zhiping Jia, Zili Shao, and Feng Chen. 2022. Removing {Double-Logging} with passive data persistence in {LSM-tree} based relational databases. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 101–116.

[46] Kecheng Huang, Zhaoyan Shen, Zhiping Jia, Zili Shao, and Feng Chen. 2022. Removing Double-Logging with Passive Data Persistence in LSM-tree based Relational Databases. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 101–116. https://www.usenix.org/conference/fast22/presentation/huang

[47] Joseph Idziorek, Alex Keyes, Colin Lazier, Somu Perianayagam, Prithvi Ramanathan, James Christopher Sorenson III, Doug Terry, and Akshat Vig. 2023. Distributed Transactions at Scale in Amazon {DynamoDB}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 705–717.

[48] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. 2009. Improving OLTP scalability using speculative lock inheritance. *Proceedings of the VLDB Endowment* 2, 1 (2009), 479–489.

[49] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 24–35.

[50] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[51] Dong Hyun Kang, Woonhak Kang, and Young Ik Eom. 2018. S-WAL: Fast and efficient write-ahead logging for mobile devices. *IEEE Transactions on Consumer Electronics* 64, 3 (2018), 319–327.

[52] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. {SpanFS}: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 249–261.

[53] Netanel Katzburg, Amit Golander, and Shlomo Weiss. 2018. NVDIMM-N persistent memory and its impact on two relational databases. In *2018 IEEE International Conference on the Science of Electrical Engineering in Israel (ICSEE)*. IEEE, 1–5.

[54] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. 2022. ScaleXFS: Getting scalability of XFS back on the ring. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 329–344. https://www.usenix.org/conference/fast22/presentation/kim-dohyun

[55] Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-lived transactions made less harmful. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 495–510.

[56] Jongbin Kim, Hyeongwon Jang, Seohui Son, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. 2019. Border-collie: a wait-free, read-optimal algorithm for database logging on multicore hardware. In *Proceedings of the 2019 International Conference on Management of Data*. 723–740.

[57] Rusty Klophaus. 2010. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*. 1–1.

[58] Dimitrios Koutsoukos, Raghav Bhartia, Ana Klimovic, and Gustavo Alonso. 2021. How to use persistent memory in your database. *arXiv preprint arXiv:2112.00425* (2021).

[59] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[60] Jennifer Lam, Jeffrey Helt, Wyatt Lloyd, and Haonan Lu. 2024. Accelerating Skewed Workloads With Performance Multipliers in the {TurboDB} Distributed Database. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1213–1228.

[61] Bohyun Lee, Mijin An, and Sang-Won Lee. 2023. LRU-C: Parallelizing Database I/Os for Flash SSDs. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2364–2376.

[62] Bohyun Lee, Seongjae Moon, Jonghyeok Park, and Sang-Won Lee. 2025. Boosting OLTP Performance with Per-Page Logging on NVDIMM. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–28.

[63] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4536–4545.

[64] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 185–196.

[65] Cheng Li, Hao Chen, Chaoyi Ruan, Xiaosong Ma, and Yinlong Xu. 2021. Leveraging NVMe SSDs for building a fast, cost-effective, LSM-tree-based KV store. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–29.

[66] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3786–3798.

[67] Yuguo Li, Shaoheng Tan, Zhiwen Wang, and Dingding Li. 2022. A NUMA-aware Key-Value Store for Hybrid Memory Architecture. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–6.

[68] Gang Liu, Leying Chen, and Shimin Chen. 2023. Zen+: a robust NUMA-aware OLTP engine optimized for non-volatile main memory. *The VLDB Journal* 32, 1 (2023), 123–148.

[69] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the design and scalability of distributed shared-data databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 663–676.

[70] Ziyi Lu, Qiang Cao, Hong Jiang, Shucheng Wang, and Yuanyuan Dong. 2022. p2kvs: a portable 2-dimensional parallelizing framework to improve scalability of key-value stores on ssds. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 575–591.

[71] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. 2011. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the 38th annual international symposium on Computer architecture*. 449–460.

[72] John L Manferdelli, Naga K Govindaraju, and Chris Crall. 2008. Challenges and opportunities in many-core computing. *Proc. IEEE* 96, 5 (2008), 808–815.

[73] GR Markall, A Slemmer, DA Ham, PHJ Kelly, CD Cantwell, and SJ3001216 Sherwin. 2013. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids* 71, 1 (2013), 80–97.

[74] Umar Farooq Minhas, Rui Liu, Ashraf Aboulnaga, Kenneth Salem, Jonathan Ng, and Sean Robertson. 2012. Elastic Scale-Out for Partition-Based Database Systems. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. 281–288. doi:10.1109/ICDEW.2012.52

[75] Mohamed A Mohamed, Obay G Altrafi, and Mohammed O Ismail. 2014. Relational vs. nosql databases: A survey. *International Journal of Computer and Information Technology* 3, 03 (2014), 598–601.

[76] MongoDB. 2024. *MongoDB*. https://www.mongodb.com/ Accessed: 2024-10-15.

[77] MySQL. 2024. *MySQL*. https://www.mysql.com/ Accessed: 2024-10-15.

[78] MySQL. 2024. *MySQL NDB Cluster*. https://www.mysql.com/products/cluster/ Accessed: 2024-10-15.

[79] Lam-Duy Nguyen, Adnan Alhomssi, Tobias Ziegler, and Viktor Leis. 2025. Moving on From Group Commit: Autonomous Commit Enables High Throughput and Low Latency on NVMe SSDs. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–24.

[80] Hyeonseok Oh, Hyeongwon Jang, Jaeeun Kim, Jongbin Kim, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. 2020. DEMETER: hardware-assisted database checkpointing. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 394–403.

[81] Oracle. 2025. Introduction to Dynamic Resource Pools. https://docs.oracle.com/cd/E36784_01/html/E37631/gbtkx.html. Accessed: 2025-09-28.

[82] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but mighty: designing and realizing scalable latency tolerance for manycore SoCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 817–830.

[83] Jiwoong Park, Cheolgi Min, and HeonYoung Yeom. 2017. A new file system I/O mode for efficient user-level caching. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 649–658.

[84] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. 2017. SQL statement logging for making SQLite truly lite. *Proceedings of the VLDB Endowment* 11, 4 (2017), 513–525.

[85] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. 2022. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture* 129 (2022), 102561.

[86] PostgreSQL. 2025. *PostgreSQL*. https://www.postgresql.org/ Accessed: 2025-09-26.

[87] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: a locality-centric high-performance streaming graph engine. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 33–49.

[88] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. 2009. Keyword search in databases: the power of RDBMS. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 681–694.

[89] Redis. 2024. *Redis*. https://redis.io/ Accessed: 2024-10-15.

[90] Yujie Ren, Changwoo Min, and Sudarsun Kannan. 2020. {CrossFS}: A cross-layered {Direct-Access} file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 137–154.

[91] Daniel Ritter, Luigi Dell'Aquila, Andrii Lomakin, and Emanuele Tagliaferri. 2021. OrientDB: A NoSQL, Open Source MMDMS.. In *BICOD*. 10–19.

[92] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. 2023. Persistent memory disaggregation for cloud-native relational databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 498–512.

[93] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient join algorithms for large database tables in a multi-GPU environment. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 14. NIH Public Access, 708.

[94] ScyllaDB. 2025. *ScyllaDB*. https://www.scylladb.com/ Accessed: 2025-09-29.

[95] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[96] Ge Shi, Ziyi Yan, and Tianzheng Wang. 2023. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.

[97] Utku Sirin, Sandhya Dwarkadas, and Anastasia Ailamaki. 2021. Performance characterization of HTAP workloads. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1829–1834.

[98] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 729–730.

[99] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. 2018. High-Performance Transaction Processing in Journaling File Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 227–240. https://www.usenix.org/conference/fast18/presentation/son

[100] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[101] Chao Sun, Asuka Arakawa, and Ken Takeuchi. 2014. SEA-SSD: A storage engine assisted SSD with application-coupled simulation platform. *IEEE Transactions on Circuits and Systems I: Regular Papers* 62, 1 (2014), 120–129.

[102] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. 2020. An analysis of concurrency control protocols for in-memory databases with ccbench. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3531–3544.

[103] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[104] Transaction Processing Performance Council (TPC). 2024. *TPC-C*. https://www.tpc.org/tpcc/ Accessed: 2024-10-15.

[105] András Vajda and András Vajda. 2011. Multi-core and many-core processor architectures. *Programming Many-Core Chips* (2011), 9–43.

[106] Mehul Nalin Vora. 2011. Hadoop-HBase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 1. IEEE, 601–605.

[107] Mehul Wagle, Daniel Booss, Ivan Schreter, and Daniel Egenolf. 2015. NUMA-aware memory management with in-memory databases. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 45–60.

[108] Donghui Wang, Peng Cai, Weining Qian, and Aoying Zhou. 2021. Discriminative admission control for shared-everything database under mixed OLTP workloads. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 780–791.

[109] Hao Wang, Jiaxin Ou, Ming Zhao, Sheng Qiu, Yizheng Jiao, Yi Wang, Qizhong Mao, Zhengyu Yang, Yang Liu, Jianshun Zhang, et al. 2024. LavaStore: ByteDance's Purpose-Built, High-Performance, Cost-Effective Local Storage Engine for Cloud Services. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3799–3812.

[110] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1542–1553.

[111] Ruihong Wang, Jianguo Wang, Stratos Idreos, M Tamer Özsu, and Walid G Aref. 2022. The case for distributed shared-memory databases with RDMA-enabled memory disaggregation. *arXiv preprint arXiv:2207.03027* (2022).

[112] Xiang Wu, Delin Cai, and Shujie Guan. 2019. A multiple LRU list buffer management algorithm. In *IOP Conference Series: Materials Science and Engineering*, Vol. 569. IOP Publishing, 052002.

[113] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: lightweight parallel logging for in-memory database management systems. (2020).

[114] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2022. Time series data encoding for efficient storage: A comparative analysis in apache iotdb. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2148–2160.

[115] Yao Xiao, Yuankun Xue, Shahin Nazarian, and Paul Bogdan. 2017. A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 217–224.

[116] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. 2023. OceanBase Paetica: A Hybrid Shared-Nothing/Shared-Everything Database for Supporting Single Machine and Distributed Cluster. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3728–3740.

[117] Lujia Yin, Yiming Zhang, Zhaoning Zhang, Yuxing Peng, and Peng Zhao. 2021. Parax: Boosting deep learning for big data analytics on many-core cpus. *Proceedings of the VLDB Endowment* 14, 6 (2021), 864–877.

[118] Yigui Yuan, Peiquan Jin, and Xiaoliang Wang. 2025. twCache: Thread-Wise Cache Management with High Concurrency Performance. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 1677–1689.

[119] Zhilin Zeng, Hui Li, Xiyue Gao, Hui Zhang, Huiquan Zhang, and Jiangtao Cui. 2024. SLSM: an efficient strategy for lazy schema migration on shared-nothing databases. In *International Conference on Database Systems for Advanced Applications*. Springer, 357–367.

[120] Hao Zhang, Xianzhi Zeng, Shuhao Zhang, Xinyi Liu, Mian Lu, and Zhao Zheng. 2023. Scalable Online Interval Join on Modern Multicore Processors in Open-MLDB. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3031–3042.

[121] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. 2020. Scalable Parallel Flash Firmware for Many-core Architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 121–136. https://www.usenix.org/conference/fast20/presentation/zhang-jie

[122] Peng Zhang, Jianbin Fang, Canqun Yang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Optimizing streaming parallelism on heterogeneous many-core architectures. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1878–1896.

[123] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, et al. 2021. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1900–1912.

[124] Yingqiang Zhang, Xinjun Yang, Hao Chen, Feifei Li, Jiawei Xu, Jie Zhou, Xudong Wu, and Qiang Zhang. 2024. Towards a Shared-Storage-Based Serverless Database Achieving Seamless Scale-Up and Read Scale-Out. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5119–5131.

[125] Bolong Zheng, Yongyong Gao, Jingyi Wan, Lingsen Yan, Long Hu, Bo Liu, Yunjun Gao, Xiaofang Zhou, and Christian S Jensen. 2023. DecLog: Decentralized Logging in Non-Volatile Memory for Time Series Database Systems. *Proceedings of the VLDB Endowment* 17, 1 (2023), 1–14.

[126] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. [n. d.]. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. ([n. d.]).