PhoebeDB: A Disk-Based RDBMS Kernel for High-Performance and Cost-Effective OLTP

Boge Liu
Data Principles (Beijing) Technology
boge.liu@enmotech.com

Chunling Wang
Data Principles (Beijing) Technology
chunling.wang@enmotech.com

Xiaoshuang Chen
Data Principles (Beijing) Technology
xiaoshuang.chen@enmotech.com

Yu Hao
Data Principles (Beijing) Technology
yu.hao@enmotech.com

Zhengyi Yang*
University of New South Wales
zhengyi.yang@unsw.edu.au

Yi Jin
Data Principles (Beijing) Technology
yi.jin@enmotech.com

Yixing Yang
Data Principles (Beijing) Technology
yixing.yang@enmotech.com

Wenke Yang
Data Principles (Beijing) Technology
wenke.yang@enmotech.com

Wanchuan Zhang
Data Principles (Beijing) Technology
wanchuan.zhang@enmotech.com

Wenjie Zhang
University of New South Wales
wenjie.zhang@unsw.edu.au

ABSTRACT

Relational databases have long been fundamental to data management. This paper presents PhoebeDB, an enterprise- and commercial-oriented RDBMS kernel that integrates recent research with practical innovations to deliver high-performance, cost-efficient OLTP solutions. It features: 1) an in-memory data-centric storage design optimized for parallel access and data temperature-based buffer management, 2) a co-routine pool-based runtime with a smart scheduler that maximizes CPU utilization for high-concurrency workloads, and 3) optimized transaction management with in-memory UNDO logs, hybrid concurrency control, parallel Write-Ahead Logging with Remote Flush Avoidance, and enhanced snapshot isolation. Experiments show that PhoebeDB achieves nearly 13.7 million tpmC and 30 million tpm on the TPC-C benchmark using a single machine, delivering a 27× improvement over PostgreSQL.

1 INTRODUCTION

Relational Database Management Systems (RDBMS) have been a cornerstone of data management since their inception in the 1970s, supporting a wide range of applications from financial transactions to customer relationship management. These systems structure data into tables with rows and columns, enabling efficient storage, retrieval, and manipulation.

Motivation. Early studies [15, 38, 40] have demonstrated that conventional RDBMS architectures struggle to fully exploit modern hardware capabilities (e.g., multi-core processors, high-capacity DRAM, and high-speed PCIe SSDs) due to legacy designs that were originally optimized for older hardware configurations (e.g., limited-core CPUs, small-capacity DRAM, and slower magnetic disks). As a result, substantial efforts have been devoted to optimizing RDBMS, leading to groundbreaking advancements in its architectures over the past decade [10, 13–15, 17, 18, 20, 21, 23, 25, 28, 30, 33, 35, 36, 41]. Despite these advancements, many remain siloed, lacking integration into a cohesive system. A holistic, full-stack design is essential to meet evolving enterprise demands, unifying these innovations to fully leverage modern hardware and enhance performance in today's dynamic database landscape.

Background. Modern businesses require high performance and cost-effectiveness, necessitating designs that maximize modern hardware utilization and adapt to evolving operational needs.

- *Hardware Trends:* Modern hardware evaluation, including multi-core CPUs, large-capacity DRAM, and high-speed SSDs, have significantly enhanced computational capabilities. These advancements create substantial opportunities for optimizing RDBMS performance [15, 21, 23, 33, 41].
- *OLTP Requirements:* Commercial relational databases are essential to enterprise IT, powering critical applications in banking, finance, and e-commerce. High-performance Online Transaction Processing (OLTP) is often primary focus in these sectors, alongside the need for seamless integration with existing ecosystems.
- *Cost Effectiveness:* Increasing cost pressures drive enterprises to seek RDBMS solutions that minimize overhead. An effective RDBMS must maximize the performance-to-cost ratio by leveraging modern hardware, streamlining operations, and reducing operational expenses to meet economic demands.

Aims. We focus on the following three key goals in this work.

- *High-Performance OLTP:* Optimize OLTP workloads by harnessing modern hardware, including multi-core CPUs, large DRAM capacities, and high-speed SSDs, combined with a re-engineered full technical stack and design innovations.
- PostgreSQL *Compatibility:* Ensure full compatibility with PostgreSQL, providing a familiar interface that simplifies migration to PhoebeDB while preserving existing workflows and toolchains.
- *Commodity Hardware Oriented:* Deliver robust performance on disk-based storage and commodity servers, eliminating reliance on specialized hardware to reduce infrastructure costs.

Contributions. In this paper, we present PhoebeDB, an enterprise- and commercial-oriented RDBMS kernel designed to deliver high-performance and cost-effective OLTP. By combining recent advancements in database research [14, 15, 21, 30, 41] and modern hardware capabilities, PhoebeDB aims to address real-world business requirements while maintaining compatibility with PostgreSQL. PhoebeDB leverages innovative designs to optimize OLTP workloads, achieving 13.7 million tpmC (transactions per minute Type-C) and 30 million tpm (transactions per

*Zhengyi Yang is the corresponding author.

minute) on the TPC-C benchmark using a single machine, delivering a 27× performance improvement over PostgreSQL. The main contributions are summarized as follows.

- *In-Memory Data-Centric Storage Design:* PhoebeDB adopts an in-memory data-centric storage optimized for parallel access, utilizing data temperature-based buffer policies to organize data into hot, cold, and frozen storage layers, ensuring optimal access patterns and efficient data retrieval.
- *Co-Routine Pool-based Parallel Execution:* To maximize CPU utilization, PhoebeDB implements a co-routine pool-based runtime with a pull-based scheduler that minimizes synchronization overhead. This approach ensures efficient parallel execution and scalability for high-concurrency workloads.
- *Transaction Management and Concurrent Control:* PhoebeDB optimizes transactions for massive-core using in-place updates with in-memory UNDO logs and a hybrid concurrency control mechanism, while maintaining PostgreSQL compatibility. A parallel Write-Ahead Logging (WAL) mechanism with Remote Flush Avoidance (RFA) [30] achieves high-throughput I/O on NVMe SSDs, while snapshot acquisition under snapshot isolation is reduced to $O(1)$.

2 RELATED WORK

PhoebeDB builds on recent advancements in database research, drawing inspiration from systems such as LeanStore [21], Umbra [41], and MosaicDB [14].

Shifted Optimization Focus. Traditional DBMS architectures were primarily designed to mitigate I/O latencies. However, modern hardware has shifted the focus to in-memory data-centric processing and maximizing computational efficiency [12, 15, 23].

In-Memory Data-Centric Architecture. Anti-Caching [15] introduces optimization of computations on main-memory-resident data to enhance performance, along with the concept of data temperature to manage DRAM by evicting cold data to SSD. This paradigm aligns with modern hardware trends [1], spurring research into optimizing CPU-DRAM interactions.

Reducing Contention in Massive-Core Computation. Minimizing globally shared data structures is key in massive-core environments. LeanStore [21] employs a B-Tree with pointer swizzling to eliminate the need for a global hash table, reducing contention.

Maximizing Computational Resource Utilization. Corobase [13] and MosaicDB [14] show that co-routines facilitate lightweight context switching and efficient state management, outperforming traditional threading models in RDBMS workloads.

Improving CPU Cache Efficiency. Efficient CPU cache utilization reduces latency between cache and main memory. MosaicDB [14] uses co-routines [9] with cache prefetching to avoid CPU stalls, while ART [44] and OLC [22] enhance cache coherence.

Reducing Instructions in Critical Code Paths. [39] discusses the instruction level cost of transactions, showing inefficiencies in transaction routines. This motivates systems that reduce instruction costs and improve throughput [30].

Disk-Based Systems with In-Memory Performance. LeanStore [1], Umbra [41], and CedarDB [6] leverage NVMe SSD technology with PCIe channels and internal parallelism to deliver near in-memory performance on disk-based systems, narrowing the performance gap between disk and memory.

3 POSITIONING AND DESIGN PHILOSOPHY

Designed for industry-grade performance, PhoebeDB delivers a high-performance, cost-effective solution for business needs.

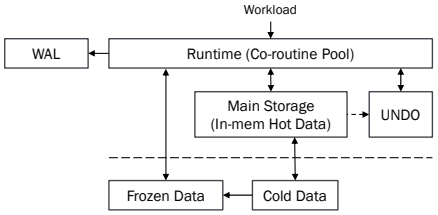


Figure 1: PhoebeDB **Implemented Components**

Business Value-Centric Design. PhoebeDB adheres to two fundamental principles in its business value-centric design.

- *Alignment with Industry Standards:* PhoebeDB is developed to align with widely adopted industry standards, conventions, and ecosystems. By using PostgreSQL, one of the most popular open-source databases, as a reference, PhoebeDB ensures full compatibility with PostgreSQL and its plugins, enabling seamless integration into existing customer environments and workflows.
- *Optimization for Standardized Hardware:* PhoebeDB is optimized to operate on cost-effective, general-purpose hardware and software stacks. This design ensures broad deployability, allowing customers to maximize their return on investment by leveraging existing infrastructure and minimizing hardware costs.

Approaches to Cost-Effectiveness. Cost-effectiveness is a key for businesses. PhoebeDB adheres to the following principles.

- *Centralized over Distributed Solutions:* Distributed systems add complexity and network overhead that can affect cost-effectiveness [27]. For most businesses, excluding the largest, single-server systems are often sufficient and more efficient. Therefore, PhoebeDB prioritizes optimizing single-server performance.
- *Performance Optimization:* Improving DBMS performance is the most effective way to expand capacity while maintaining budget constraints. By optimizing resource utilization, PhoebeDB reduces infrastructure costs and lowers energy consumption.
- *Prioritizing OLTP Workload:* PhoebeDB employs resource isolation and scheduling to prioritize OLTP while allowing less critical workloads, such as batch or analytical queries, to use idle resources. This maximizes resource utilization without compromising OLTP performance, ensuring high-quality service (QoS).
- *Leverage a Mature Ecosystem:* PhoebeDB builds upon verified efforts while introducing re-innovations tailored to practical use cases. It maintains compatibility with PostgreSQL and its plugins, reducing deployment costs and maintenance risks.
- *Future HTAP Potential:* While PhoebeDB is currently optimized for OLTP workloads, its design includes forward-looking features to support native Hybrid Transactional and Analytical Processing (HTAP) in the future [8, 24, 26].

4 SYSTEM ARCHITECTURE

This section discusses the core architecture of PhoebeDB. The major components are illustrated in Figure 1.

Evolving Hardware. The evolution of hardware has significantly transformed the configuration of modern computers. We summarize three major advancements as follows.

- *Massive Cores:* Modern servers now can feature hundreds of cores, enabling high parallelism and allowing centralized systems to efficiently handle OLTP workloads.
- *Large Memory Size:* Affordable high-capacity DRAM allows the majority of OLTP data to reside in memory, transforming operations into in-memory computations and boosting performance.
- *Fast NVMe SSD:* PCIe-based SSDs deliver high throughput, eliminating disk latency [23]. SSD arrays further scale throughput, making I/O latency no longer a major RDBMS bottleneck [15].

In-memory Data Centric Processing. Leveraging recent hardware advancements, PhoebeDB adopts the *in-memory data-centric processing* paradigm from [15] to optimize performance. Unlike traditional relational DBMS architectures designed to mitigate I/O latency, PhoebeDB integrates modern research innovations to deliver high-performance solutions tailored for real-world applications. Its architecture includes the following key features.

- *Parallel Access Optimized Storage:* Targeting future HTAP support, PhoebeDB adopts the PAX format [2] for storing base table data. Similar to Umbra [41], PhoebeDB leverages secondary indexes to enhance general-purpose data access flexibility, avoiding the integration of primary keys into base tables as seen in [21].
- *Data Temperature-based Buffer Policies:* PhoebeDB employs the concept of *data temperature* to dynamically organize data for optimal access. By categorizing data pages as hot, cold, or frozen based on access frequency, PhoebeDB optimizes in-memory access for hot and cold data while using an in-memory MVCC implementation [29]. Batch update mechanisms are re-innovated to ensure efficiency and practicality in real-world applications.
- *Co-routine Pool-based Runtime with Smart Scheduler:* PhoebeDB employs lightweight co-routines as execution units, managed by a smart scheduler to maximize CPU utilization and minimize resource wastage from synchronization or idle waiting. . By minimizing the use of spin-lock-based synchronization, PhoebeDB ensures that CPUs are fully dedicated to de-facto computations.
- *Massive-core Optimized Transaction Management:* PhoebeDB implements transaction isolation levels identical to PostgreSQL, ensuring compatibility with the PostgreSQL ecosystem. An in-place update strategy combined with an in-memory UNDO log enables high-performance OLTP transactional operations.
- *Massive-core Optimized Pessimistic Concurrency Control:* To maintain compatibility with PostgreSQL, PhoebeDB adopts a pessimistic concurrency control model. It introduces a new mechanism to eliminate reliance on global lock data structures, enhancing scalability and efficiency in multi-core environments.
- *Parallel WAL with RFA:* PhoebeDB integrates a parallel Write-Ahead Logging (WAL) flushing technique, utilizing Remote Flush Avoidance (RFA) to maximize parallelism. This design, combined with high-throughput NVMe SSDs, meets the demanding I/O requirements of modern workloads.
- *Efficient Snapshot Isolation:* PhoebeDB maintains compatibility with PostgreSQL by supporting the same snapshot isolation levels (*read committed and repeatable read*). However, it enhances efficiency by replacing traditional transaction scanning with a simplified snapshot mechanism based on a single timestamp, reducing snapshot acquisition to $O(1)$ complexity.

Transaction Management. PhoebeDB employs a transaction management system for high performance and reliability.

- *In-Memory History Version Storage:* Retains historical data versions in memory to enhance transaction processing performance.
- *Efficient Visibility Checks:* Commit timestamps recorded for every UNDO log to enable fast and accurate visibility checks.
- *Decentralized UNDO Log:* Groups UNDO logs by the same transaction to minimize write contention and garbage collection.
- *Optimized Garbage Collection:* Detailed UNDO log information accelerates table and index cleanup, enabling fast log recycling.
- *Efficient Concurrency Support:* The design ensures efficient concurrent transaction processing by eliminating critical bottlenecks.

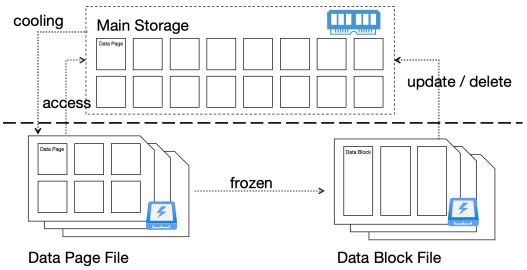


Figure 2: PhoebeDB**'s 3-Layer Storage**

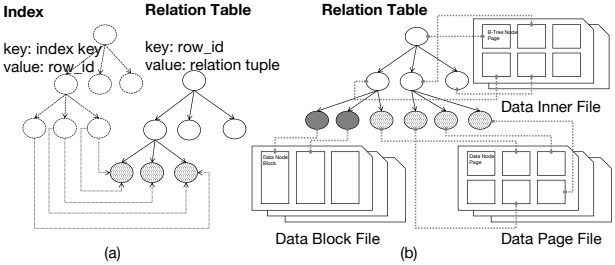


Figure 3: PhoebeDB **Storage File Structure**

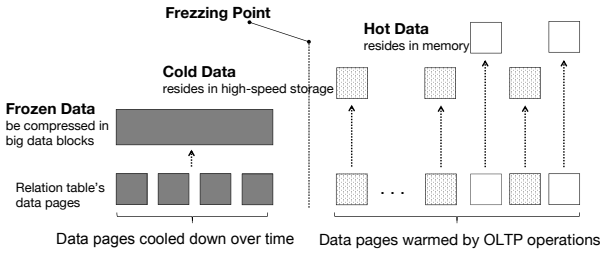


Figure 4: PhoebeDB **Page Temperature**

5 DATA STORAGE

5.1 In-Memory Data-Centric Storage

PhoebeDB organizes data into relations, storing each relation's data together. Designed for future HTAP support, PhoebeDB distributes data across three storage layers (Figure 2) based on access frequency: **Main Storage** (in memory), **Data Page File** (on disk), and **Data Block File** (on disk), representing *hot*, *cold*, and *frozen* data, respectively. Data moves among layers for optimized performance and resource utilization.

PhoebeDB organizes data using a *B-Tree*, where each B-Tree represents a relation. It Leverages swizzle pointer technology [21] for efficient memory management while avoiding contention from global hash tables, ensuring unified management of hot, cold, and frozen data. To address inefficiencies in B-Tree page management, particularly insertions such as node splitting, and to support tables without primary indexes, PhoebeDB utilizes an internally maintained, monotonically increasing ***row_id*** as the key. Each tuple is stored as the value in the Data Page or Data Block Files (Figure 3(b)). User-defined indexes are implemented as secondary indexes, storing (key, row_id) pairs in the Index File. For data consistency and efficient access under high concurrency, PhoebeDB employs a Multi-Version Concurrency Control (MVCC) mechanism (discussed later).

5.2 Temperature-based Exchange

PhoebeDB categorizes data pages as *hot*, *cold*, or *frozen*, based on access frequency. This *temperature*-based approach ensures optimal storage for OLTP and OLAP workloads.

Hot and cold pages, which store most OLTP-accessed data, use the PAX (Partition Attributes Across [2]) format. Hot pages reside in memory (Main Storage) for rapid access, while cold

pages are stored on disk. Both hot and cold pages support in-place updates, with historical versions maintained in a separate transaction version buffer (UNDO log). Frozen pages primarily serve OLAP workloads, utilizing a compressed data block format to optimize analytical query performance. Out-of-place updates prevent the need for full decompression and recompression, reducing write amplification. The data exchange process consists of the following three cases (Figure 4).

(1) *Hot to Cold Data Exchange:* Hot and cold data pages transition based on OLTP access operations and buffer replacements in the main storage. Frequently accessed cold pages become hot, while less accessed hot pages move to cold storage.
(2) *Freezing Hot or Cold Pages:* Since most data is time-sensitive and only a small portion remains frequently accessed, PhoebeDB freezes data for extended periods, ensuring operations like table scans do not warm any data. PhoebeDB tracks OLTP access counts and the last OLTP access time for each hot page, using access frequency over time to manage data movement . PhoebeDB also tracks a $max\_frozen\_row\_id$, classifying data before it as frozen and data after it as unfrozen (hot or cold). Consecutive node pages with OLTP access counts below a predefined threshold are grouped into frozen data blocks, and $max\_frozen\_row\_id$ is increasing accordingly. Several consecutive leaf node pages are compressed into a frozen data block while preserving the row_id order. In rare cases, hot data may also transition between frozen and warmed states due to read, delete, and insert operations.
(3) *Warming Frozen Pages:* Delete & update operations mark frozen data as deleted, inserting updated versions into hot data pages. Frequently accessed frozen pages, identified by exceeding a predefined row_id read threshold, are marked as deleted and reinserted into hot storage, requiring updates to related table indexes.

5.3 In-Memory Main Storage

PhoebeDB stores hot data in memory. To reduce contention and overhead from global hash maps in traditional database systems [39], it employs Pointer Swizzling [21] for efficient data storage and retrieval. Additionally, a B-Tree structure organizes table data, enabling efficient page lookup and improving performance under high-concurrency workloads.

B-Tree Structure. Traditionally, a global hash map is used to quickly determine whether a page resides in main storage. However, locking the hash map leads to high concurrency contention. To mitigate the issue, we eliminate the global hash map and instead use a B-tree to accelerate page lookup. The B-Tree structure (Figure 3(a)) in PhoebeDB organizes table and index data. Each tuple is assigned a unique row_id as the primary key, enabling rapid data retrieval and manipulation with minimal overhead. In the *table B-Tree*, tuples are stored in a compressed format within leaf nodes to optimize space and access speed, supporting efficient sequential and random access for both transactional and analytical workloads. The *index B-Tree* complements this by enabling efficient point and range queries, mapping user-defined keys to row_id values that link to tuples in the table B-Tree. This dual B-Tree design ensures flexibility, accuracy, and efficiency.

Pointer Swizzling. PhoebeDB employs pointer swizzling [16, 21] to manage seamless transitions between three states: *Hot*, *Cooling*, and *Cold*. In the *Hot* state, the swizzle pointer directly references the buffer frame in memory, eliminating indirection through a mapping table, reducing latency, and enhancing performance. When memory reaches its capacity, pages enter the *Cooling* state, where they remain in memory with a marked cooling bit on the swizzle pointer, signaling readiness for eviction while allowing fast access if needed. Pages no longer required in memory transition to the *Cold* state, with their IDs stored to indicate relocation to disk. Upon access, cold pages are reloaded into memory, and their pointers are swizzled back to the *Hot* state, restoring direct references. This dynamic state management optimizes memory usage and reduces access overhead.

Remark. PhoebeDB integrates the B-Tree structure with pointer swizzling to achieve notable advantages. This combination eliminates reliance on a global hash map, simplifying system architecture and reducing overhead. By dynamically managing page states, pointer swizzling optimizes memory usage and ensures sustained high performance under heavy workloads. The B-Tree structure further enhances query processing and data manipulation, offering a robust organizational framework. Together, these innovations position PhoebeDB as a high-performance database system capable of efficiently handling large-scale data operations.

6 TRANSACTION MANAGEMENT

6.1 PostgreSQL-Compatible Transaction

PhoebeDB maintains compatibility with PostgreSQL by adopting the same snapshot isolation levels (*read committed* and *repeatable read*). However, it improves efficiency by representing snapshots with a single timestamp, reducing snapshot acquisition to constant time ($O(1)$), unlike PostgreSQL 's transaction-scanning approach. PhoebeDB uses a **62-bit global logical clock**, implemented as a globally incrementing atomic integer, to assign transaction IDs (XID) and manage snapshot and commit timestamps (cts). At transaction start, a **64-bit** XID is generated, with the most significant bit set to 1, 62 bits for the start timestamp, and 1 bit reserved for future use. Upon commit, the transaction assigns a cts and records it in the UNDO logs. Visibility checks compare snapshot timestamps with commit timestamps, ensuring efficient snapshot management. PhoebeDB also adopts PostgreSQL 's lock semantics, including tuple and transaction ID locks. Future plans include developing PhoebeDB as a PostgreSQL plugin, delegating unsupported SQL queries to PostgreSQL while progressively expanding support for PL/pgSQL.

6.2 MVCC with In-memory UNDO

In this section, we introduce the multi-version concurrency control (MVCC) design with in-memory UNDO in PhoebeDB.

Before-Image Delta for UNDO Log. PhoebeDB adopts before-image delta for UNDO logging [3, 34, 46], where only the differences between the old and new data are recorded. The UNDO logs generated by a single transaction are stored together, and those for the same tuple are linked from the newest to the oldest, forming a version chain. In Figure 5, the curved arrows illustrate how version chains are formed by connecting UNDO logs from various transactions. Each UNDO log includes two timestamps: the start timestamp (sts) and the end timestamp (ets). The sts indicates when the before image was committed, while the ets represents the timestamp when the UNDO log itself was committed. When a transaction generates a new UNDO log, it inserts at the head of the version chain and writes its XID to the ets field. The sts is set to the ets of the previous UNDO log, or to 0 if the previous UNDO log has been reclaimed. Upon commit, the ets is updated to the transaction's commit timestamp (cts).

Example 6.1. In Figure 5, XID 7 updates the value of rid1 from b to a. The old value b is modified by XID 4. In the previous UNDO log ets = 6, we have XID 4 commits at timestamp 6. Thus, XID 7 sets sts as 6. As XID 7 has not committed yet, the ets field remains as its XID.

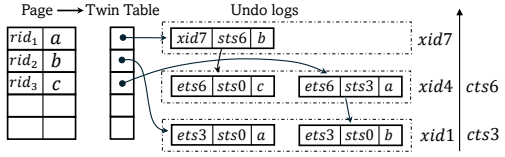


Figure 5: PhoebeDB **MVCC**

Algorithm 1: Retrieve Visible Version

Input : Transaction ID: xid and snapshot: $snapshot$
Output : The visible version of required $tuple$

```
1  tuple ← read from table;
2  if current page has no twin table then return tuple;
3  header ← read from twin table;
4  if header = null ∨ header is invalid ∨header.ets ≤ snapshot
     ∨header.ets = xid then return tuple;
5  cur ← header;
6  while cur ≠ null ∧ cur is valid do
7      assemble before-image delta read from cur into tuple;
8      if cur.sts ≤ snapshot then break;
9      cur ← cur.next;
10 return tuple;
```

Remark. The necessity of having a sts alongside the ets may seem redundant, as sts typically matches ets of the previous UNDO log. However, sts plays a crucial role in preventing the current UNDO log from inadvertently becoming an endpoint in the version chain. Without sts, reclaiming the previous UNDO log would require verifying that no active transaction depends on it, which would significantly complicate garbage collection. By explicitly defining sts, PhoebeDB simplifies the management of the version chain, ensuring efficient garbage collection and consistent performance.

Updating cts **During Commit.** In PhoebeDB, UNDO logs generated by the same transaction are stored together, allowing the transaction to maintain a single pointer to the beginning of its UNDO log during execution. At the commit phase, the ets of all UNDO logs can be updated to the transaction's cts in a single scan, ensuring efficient log updates.

Linking Tuples to UNDO Logs Using a Twin Table. A naive approach to link a tuple to its UNDO log is to append a pointer to the version chain header at the end of each tuple. However, not every tuple has UNDO logs, particularly in TP-heavy environments where most transactions focus on a small set of hot data. Adding pointers to all tuples wastes disk and memory space and increases recovery costs since pointers must be validated during recovery. To address this issue, Phoebe employs a page-level mapping table called the twin table, where each tuple has a corresponding entry storing a pointer to its version chain (or null if none). When a tuple is modified, the data page is loaded into memory (if needed), and a twin table is created if it doesn't already exist. The transaction copies the delta to a new UNDO log, writes its XID in the ets field, and updates the twin table entry to point to the UNDO log. While this approach may appear to increase memory usage, the impact is minimal. Most TP workloads concentrate on a small subset of hot data, meaning the twin table's memory footprint remains small, ensuring efficient memory usage in PhoebeDB.

Retrieve Visible Version. The process of retrieving visible versions using UNDO logs is detailed in Algorithm 1, ensuring efficient and consistent access to historical data. Visibility during a tuple read is determined as follows.

- *No twin table*: If the current page has no twin table, the tuple is immediately considered visible (Line 1-2).
- *Invalid pointer or reclaimed UNDO log*: If the pointer in the corresponding entry is null or the pointed UNDO log has been reclaimed (marked as invalid), the original tuple remains visible (Line 3-4).
- *Examine the version chain header*: The transaction checks the ets in the version chain header: If ets is not XID, the tuple is visible if ets is less than or equal to the snapshot. If ets is XID, the tuple is visible only if ets equals the current transaction's XID. (Line 3-4)
- *Traverse the version chain*: If neither condition is met, the transaction traverses the version chain. It assembles deltas from UNDO logs until it finds the first UNDO log where the sts (start timestamp) is less than or equal to the snapshot. At this point, the reconstructed version of the tuple becomes visible (Line 5-9).

Example 6.2. In Figure 5, XID 3 ($snapshot = 5$) reads three tuples. For $rid1$, as the ets in the version chain header is XID7 ≠ XID3, value 'a' is invisible. The sts of the version chain header is 6 which is larger than 5, the value 'b' is also invisible. As a result, $rid1$ is read as 'c'. For $rid2$, since the ets in the version chain header is 3 which is less than 5, $rid2$ is read as 'b'. For $rid3$, the ets in the version chain header is 6 larger than 5, the value 'c' is invisible. The sts of the version chain header is 3 less than 5, value 'a' is visible, hence rid3 is read as 'a'.

The algorithm and example above demonstrate how historical versions are retrieved for read-only operations. For write operations at the Read Committed isolation level, the process first checks the sts field in the version chain header: 1) If sts is a timestamp, the tuple is updated. 2) If sts is a transaction ID, the operation waits on the concurrent transaction's XID lock before proceeding. At the Repeatable Read isolation level: 1) If the concurrent transaction aborts, the operation proceeds. 2) If it commits, the operation aborts.

7 PARALLEL EXECUTION & CONCURRENCY CONTROL

7.1 Co-routine Pool for High Concurrency

PhoebeDB employs a co-routine driven framework by executing transactions as co-routines. Co-routines enable lightweight user-level context switching [13, 14], allowing transactions to yield during waits (e.g., for locks or async reads). This efficiency allows a single thread to manage multiple transactions concurrently, maximizing resource utilization and concurrency.

Co-routine Driven Framework. In PhoebeDB, a worker thread[1] manages a fixed number of task slots. A task slot executes one co-routine task at a time without switching until completion. The configured number of worker threads and the task slots determine transaction concurrency. As shown in Figure 6, a worker thread handles multiple co-routines simultaneously but actively executes only one task at a time, avoiding contention for resources such as buffer frame allocation and latch acquisition within the thread. However, contention across worker threads still presents.

Reducing Contention between Workers. In PhoebeDB, each worker thread operates independently, with components such as buffer management, MVCC, and garbage collection partitioned by the worker thread. For instance, a worker thread manages its own buffer pool partition and handles page swaps locally. Similarly, the UNDO logs are managed and garbage is collected by the same worker thread that generates them. As shown in

[1]The number of worker threads can be configured to match the CPU core count to maximize CPU utilization.

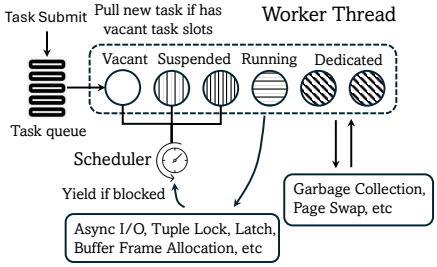


Figure 6: PhoebeDB **Worker Thread**

Figure 6, a worker thread has dedicated task slots for page swaps and garbage collection, reducing cross-thread contention.

Pull-based Scheduler. Scheduling in PhoebeDB is challenging due to the flexible yielding nature of transactions, which can pause execution for various reasons. Transactions that yield due to latch spins or asynchronous reads should be resumed promptly, whereas those yielding due to tuple locks may involve longer waits until other transactions signal readiness. Additionally, unlike threads, co-routines cannot be interrupted by other co-routines, which requires voluntary yielding to adjust execution priorities, creating potential delays. To address the challenges, *pull-based* execution model is employed. Transactions are submitted to a global task queue, and worker threads pull new tasks when task slots are vacant. Yields are categorized by urgency, with worker threads prioritizing high-urgency cases (e.g., mutex spins, async reads) by pausing new task acceptance and resolving current tasks. Low-urgency cases (e.g., tuple locks), do not block task pulling. Page swaps are triggered when buffer frames drop below a threshold, while garbage collection occurs after a certain number of transactions.

7.2 In-memory Parallel Optimization

Hybrid Lock Strategies. PhoebeDB employs a hybrid locking strategy that combines pessimistic and optimistic locks, boosting system concurrency while minimizing transaction abort rates.

Optimistic Locks. An *optimistic lock* [7] is a lightweight synchronization mechanism to enhance concurrency, particularly in read-heavy workloads. Read operations proceed without locking and its correctness is verified by checking a version counter (increment on every write) at the end. If it remains unchanged, the read completes successfully; otherwise, it retries. This method reduces contention and avoids blocking during write operations.

Optimistic Lock Coupling. *Lock coupling* [4] is a common technique for safe, concurrent access to data structures such as B-trees. A thread locks the current node until it locks the next, and then releases the previous lock, minimizing lock contention by limiting the number of locks held simultaneously. *Optimistic Lock Coupling* (OLC) [22] extends lock coupling by integrating optimistic locks. OLC interleaves version validation during traversal, allowing optimistic reads and avoiding bottlenecks caused by coarse-grained locks, thereby boosting concurrent performance.

Hybrid Lock Strategies. While OLC improves system throughput, it introduces additional retry logic and increased validation complexity, which can lead to higher transaction abort rates under high concurrency. PhoebeDB adopts a hybrid lock strategy with three locking modes: *optimistic*, *shared*, and *exclusive*. Optimistic locks are used during B-tree traversal to maximize read concurrency, while shared and exclusive locks are applied for tuple read/write operations on B-tree leaf nodes. This hybrid strategy combines the advantages of optimistic locking for read-heavy scenarios with reduced abort rates for write-intensive operations.

Decentralized Lock Management. In traditional RDBMS systems like MySQL [32] and PostgreSQL [37], object locks are stored in a global hash table, creating contention hotspots. We addresses this with decentralized lock storage and management.

Transaction ID Lock. A *Transaction ID Lock* assigns a lock to each transaction ID. At the start of a transaction, it acquires an exclusive lock on its own ID, held until the transaction commits or aborts. If transaction A attempts to modify a tuple being modified by transaction B, it must acquire a shared lock on B's transaction ID. This ensures transaction A waits for B to complete before proceeding. In PhoebeDB, transaction ID locks manage dependencies among concurrent transactions, allowing tuple locks to be released immediately after operations. This ensures each active transaction holds at most one tuple lock at a time, preventing lock proliferation and enabling more efficient lock management.

Remark. Transaction ID locks have the following properties: (1) A transaction A can acquire only one shared lock on transaction B's ID at a time, remaining in a sleeping state until B completes and wakes it up. (2) No transaction other than A itself can hold an exclusive lock on A's ID. Hence, all waiting locks on A's ID are shared locks, which are released simultaneously once A finishes.

Lock Storage and Management. The following describes how different types of locks are managed in PhoebeDB.

- *Table Lock*: Each B-Tree in PhoebeDB corresponds to a single relation or table (Section 5), offering a natural distribution for table lock storage. Table lock information is stored in a dedicated memory block, referenced by a pointer in the B-Tree root node.
- *Transaction ID Lock*: Each transaction maintains a list of shared locks waiting on its transaction ID. Upon starting, a transaction acquires an exclusive lock on its ID, releasing it upon completion and notifying any waiting shared locks to proceed.
- *Tuple Lock*: Each active transaction holds at most one tuple lock at a time. Instead of being stored within the transaction itself, tuple locks are managed in co-routine task slots (Section 7.1). This allows a tuple lock to be applied once and reused across subsequent transactions. Tuple lock metadata, such as the number of granted locks per tuple, is stored in the twin table (Section 6.2).

7.3 Garbage Collection

Garbage collection (GC) [5, 19] is crucial for memory management, as UNDO logs are in-memory for efficient commit timestamp updates (Section 6.2) and faster access to historical versions [46], which can lead to high memory consumption.

GC Watermarks. PhoebeDB uses two watermarks for GC: the minimum active XID and the max frozen XID. The minimum active XID is the smallest XID among active transactions, while the max frozen XID is the highest XID for which all transactions with XID less than or equal to it are globally visible. The minimum active XID is determined by scanning active transactions, and a transaction is globally visible if its commit timestamp is earlier than any active transaction's snapshot. These watermarks guide GC for UNDO logs, twin tables, and deleted tuples.

GC for UNDO Logs. In PhoebeDB, a transaction's snapshot is always taken at or after its start timestamp. Therefore, UNDO logs from transactions with commit timestamps earlier than the minimum active XID 's start timestamp are eligible for reclamation. Since the commit timestamps of transactions within the same task slot are strictly ordered, UNDO logs can be reclaimed in a queue-like manner. The GC task sequentially scans these logs until it encounters one whose commit timestamp is not earlier than the minimum active XID 's start timestamp.

Remark. One significant advantage of PhoebeDB 's MVCC design (Section 6.2) is that the GC procedure avoids the need to re-acquire an exclusive latch to update pointers in the twin table. As UNDO logs are reclaimed in a queue-like manner, tracking the address of the first unreclaimed UNDO log for each task slot becomes straightforward. This enables efficient validation of the pointer in the twin table by simply comparing it to the address of the first unreclaimed UNDO log.

GC for Twin Tables. In PhoebeDB, the twin table records the largest XID of transactions that have modified it, and it can only be reclaimed if this XID is no larger than the max frozen XID, which is computed as a by-product during the UNDO log GC. Since transactions executed within the same task slot have strictly increased XID, PhoebeDB tracks the XID of the most recently reclaimed UNDO log for each task slot. The max frozen XID is then determined as the minimum value among all the recorded XID from each task slot.

GC for Deleted Tuples. Deleted tuples are physically removed once they become globally visible, a process carried out during UNDO log GC. Whenever an UNDO log containing a deletion operation is reclaimed, the corresponding tuple is removed from both the table and its associated index.

8 PARALLEL FLUSH WRITE AHEAD LOG

PhoebeDB employs Write-Ahead Logging (WAL) to ensure data integrity, and follows the "Non-Force, Steal" principle [31], that is, when a transaction is committed, it is not necessary to flush all changes of the transaction to disk, and changes from uncommitted transactions are allowed to be flushed to disk.

Traditional WAL Flushing. WAL flushing often becomes a bottleneck due to the sequential nature of flushing logs to persistent storage. While parallel WAL writes to in-memory buffers are standard, the actual flush process typically remains serialized, creating a bottleneck. Distributed Logging [45] introduces a solution where parallel transactions write local log files independently. During recovery, logs are ordered by a Global Sequence Number (GSN) to maintain transaction execution order. However, this requires transactions to wait for all prior WALs with lower GSNs to flush before committing, introducing delays.

Remote Flush Avoidance. Remote Flush Avoidance (RFA) [30] improves logging by allowing transactions to commit without waiting for unrelated transaction logs, provided they modify different data pages. This reduces commit latency by removing unnecessary dependencies between unrelated transactions.

Phoebe's Parallel WAL Design. PhoebeDB extends the parallel log writing solution from Leanstore [21] with tuple-level RFA, decoupling transaction commit dependencies from log flushing. This design enables efficient log flushing, allowing transactions to commit independently without unnecessary delays, thereby enhancing system throughput and scalability.

- *Global and Local Sequence Numbers:* Each log entry records a GSN (Global Sequence Number) and an LSN (Log Sequence Number). The GSN is a globally monotonically increasing but not unique, incremented for cross-page modifications. The LSN is strictly monotonically increasing within each WAL writer.
- *Task-Slot Specific WAL Writers:* PhoebeDB maintains a separate WAL writer for each task slot. Transactions on different task slots, modifying separate data pages, only need to wait for their respective WAL writer to flush logs to their WAL files.
- *Decoupled Dependencies:* When transactions are committed, they do not need to wait for unrelated logs with the same GSN to flush, significantly reducing contention and improving parallelism.

9 EVALUATION

Implementation and Hardware. PhoebeDB is implemented in C++ and compiled using GCC 11. All experiments were conducted on a server equipped with two Intel(R) Xeon(R) Gold 5320 CPUs (2021 release, 2.2 GHz base frequency, 52 physical cores, 104 virtual cores), 512 GB of main memory, and two Samsung PM9A3 Enterprise NVMe SSDs (2021 release), running CentOS 9.

Metrics. We evaluate throughput using the TPC-C benchmark [42], implemented as server-side user-defined functions (UDFs). We use the latest PostgreSQL 17.0 and run the TPROC-C workload [43], by HammerDB [11]. HammerDB generates a TPC-C-like workload by invoking server-side UDFs, allowing for a fair comparison. All experiments are conducted over a 30-minute duration with PhoebeDB configured to allocate 32 task slots per worker thread and the transaction isolation level set to *read committed*. Workload affinity is enabled by default, where each worker thread is bound to a CPU core. WAL sync is enabled, and a buffer size of 1GB (Main Storage) per warehouse is reserved.

Exp 1: tpmC Throughput. We evaluate PhoebeDB's transaction throughput by varying the number of worker threads and running the TPC-C benchmark with 1, 10, 25, 50, and 100 warehouses and worker threads. Figure 7(a) shows the average tpmC over the test period. Notably, PhoebeDB achieves average throughput values of 349k, 3362k, 6903k, 11578k, and 13690k tpmC at the respective scales, with stable throughput throughout the test. Despite additional storage overhead designed for future evolution into an HTAP kernel, PhoebeDB achieves a peak throughput of approximately 13.7 million tpmC.

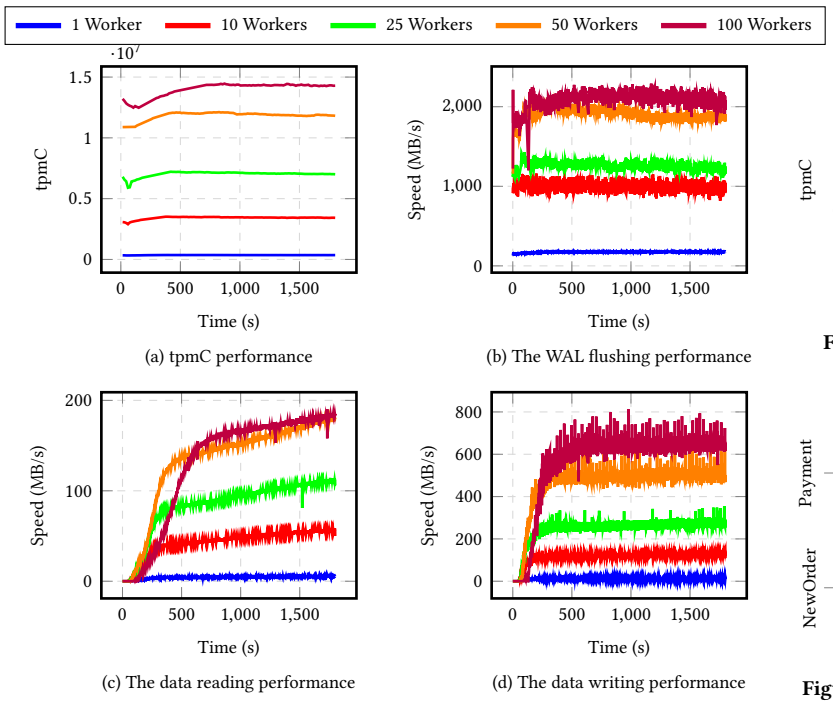
Exp 2: Scalability. Figure 8 shows average throughput as worker count increases. Performance scales nearly linearly up to 52 workers (physical CPU cores). Beyond this, per-worker performance degrades slightly, but total throughput continues to rise.

Exp 3: WAL Flushing Performance. In this experiment, we separate the WAL and data storage onto two different physical disks to evaluate WAL flushing performance, as shown in Figure 7(b). Leveraging *io_uring*, PhoebeDB effectively utilizes NVMe SSDs, achieving an average WAL flushing throughput of up to 1800 MB/s (130K IOPS for NVMe SSD Random Write). The throughput remains stable throughout the entire runtime, demonstrating the system's efficiency in handling high I/O workloads.

Exp 4: Disk I/O Throughput. As a 1GB buffer is reserved per warehouse, resulting in Main Storage sizes of 1GB, 10GB, 25GB, 50GB, and 100GB, respectively. Post-test, observed TPC-C data sizes are 12GB, 119GB, 248GB, 423GB, and 480GB, showing most data resides on disk. As seen in Figure 7(c) and (d), data exchange between Main Storage and disk begins 2 minutes in, causing tpmC fluctuations. After 7 minutes, data writing throughput and tpmC stabilize, while data reading throughput gradually increases.

Exp 5: Varying Buffer Size. Figure 10 shows PhoebeDB's performance with 100 warehouses and worker threads, varying the buffer size from 4GB to 100GB. Larger buffers improve tpm performance by reducing data page exchanges between memory and disk. However, the performance improvement slows down when the buffer size exceeds 25GB, as the buffer becomes sufficient to hold all hot pages, leading to diminishing returns.

Exp 6: Co-routine vs Thread Model. Figure 11 compares PhoebeDB's performance in high concurrency using the co-routine and thread models. We ran TPC-C with 100 warehouses and 100 worker threads under the co-routine model, assigning 32 task slots per thread. For the thread model, we configured 3200 worker threads with 1 task slot per thread, maintaining the same level of concurrency. Since the number of worker threads differs from the



(a) tpmC performance (b) The WAL flushing performance

(c) The data reading performance (d) The data writing performance

Figure 7: Performance Comparison Varying Numbers of Workers

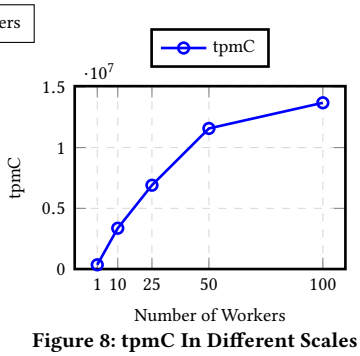


Figure 8: tpmC In Different Scales

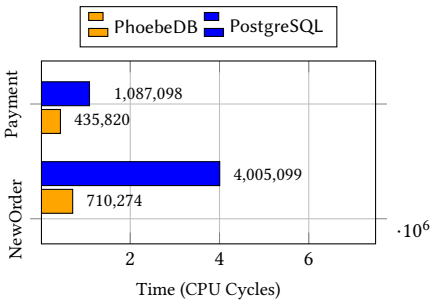


Figure 9: Transaction Processing Time

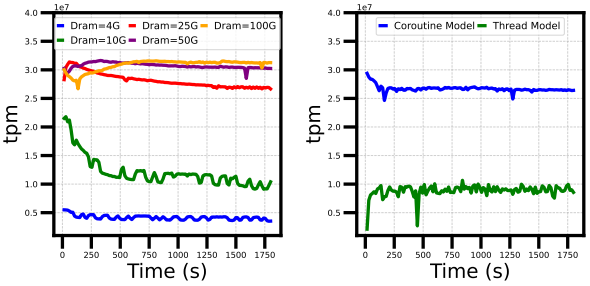


Figure 10: Performance Under Different Buffer Size

Figure 11: Performance of Co-routine vs. Thread Model

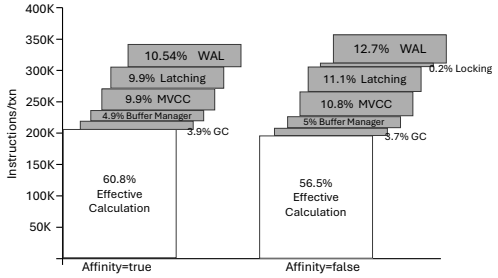


Figure 12: Breakdown of Instruction Count per Transaction for PhoebeDB **from TPC-C.**

number of warehouses, we set *affinity* to false in this experiment. As shown in Figure 11, PhoebeDB achieves significantly higher tpm in the co-routine model due to lightweight user-level context switching, enabling efficient transaction management and better resource utilization.

Exp 7: Breakdown of Instruction Count per Transaction. We also evaluate the cost distribution across various components in PhoebeDB by analyzing the instruction count per transaction from TPC-C. As shown in Figure 12, we observe: (1) When *affinity* is set to true, there is no contention, resulting in no visible locking cost. The costs associated with other components, such as WAL, MVCC, latching, buffer manager and GC, remain low. Notably, effective computation accounts for 60.8% of the total instruction count, highlighting efficient resource utilization. (2) When *affinity* is disabled, representing a scenario with contention, the instruction count per transaction increases. This is reflected in the appearance of locking costs and a higher WAL overhead, as threads contend for shared resources. Despite this, effective computation still accounts for 56.5% of the total instructions. These results demonstrate the efficacy of our proposed lock management techniques and WAL design.

Exp 8: Transactions vs PostgreSQL**.** Under identical settings, PhoebeDB achieves a total throughput of 30 million tpm, significantly outperforming PostgreSQL's 1.1 million tpm. This highlights PhoebeDB's clear advantage for OLTP workloads, delivering 27x higher throughput than PostgreSQL. In terms of CPU cycles, as illustrated in Figure 9, PhoebeDB reduces the cycles required for two critical TPC-C workload queries: the Payment transaction and the NewOrder transaction. Specifically, PhoebeDB achieves a 2.5x reduction for the Payment transaction and a 5.6x reduction for the NewOrder transaction.

Exp 9: Performance Comparisons with Commercial RDBMS. We also compare PhoebeDB with a widely used commercial RDBMS, referred to as O-DB, on the same physical machine but running CentOS 7.9. O-DB is configured with five NVMe SSDs for storage and a 260 GB buffer pool. Using HammerDB 4.4, it achieves a peak TPROC-C throughput of 3.2 million tpm. We observed that, due to I/O bandwidth limitations, O-DB utilizes only approximately 77% of CPU resources in this environment.

10 CONCLUSION

PhoebeDB introduces a disk-based RDBMS kernel optimized for high-performance and cost-efficient OLTP workloads. By leveraging modern hardware and integrating innovative features, PhoebeDB achieves about 13.7 million tpmC on the TPC-C benchmark. Future work will focus on: 1) Develop SQL interface to establish PhoebeDB as a standalone server. 2) Implement a primary-standby high-availability solution. 3) Evolve storage and execution to support native HTAP.

REFERENCES

[1] Alhomssi Adnan, Haubenschild Michael, and Leis Viktor. 2023. The Evolution of LeanStore. *20th Conference on Database Systems for Business, Technology and Web, BTW 2023* (2023), 259–281.

[2] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal* 11 (2002), 198–215.

[3] 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.

[4] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of operations on B-trees. *Acta informatica* 9 (1977), 1–21.

[5] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable garbage collection for in-memory MVCC systems. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 128–141. https://doi.org/10.14778/3364324.3364328

[6] CedarDB [n.d.]. CedarDB. https://cedardb.com/.

[7] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, Vol. 1. 181–190.

[8] Zhang Chao, Li Guoliang, Zhang Jintao, Zhang Xinning, and Feng Jianghua. 2024. HTAP Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 36 (2024), 6410–6429.

[9] Coroutines [n.d.]. Coroutines (C++20). https://en.cppreference.com/w/cpp/language/coroutines.

[10] Immanuel Haffner and Jens Dittrich. 2023. A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries.. In *EDBT*. 1–13.

[11] HammerDB [n.d.]. HammerDB. https://www.hammerdb.com/.

[12] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 981–992. https://doi.org/10.1145/1376616.1376713

[13] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *Proceedings of the VLDB Endowment* 14, 3 (2020), 431–444.

[14] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proceedings of the VLDB Endowment* 17, 3 (2023), 577–590.

[15] Debrabant Justin, Pavlo Andrew, Tu Stephen, Stonebraker Michael, and Zdonik Stan. 2013. Anti-Caching: A New Approach to Database Management System Architecture. In *2013 Proceedings of the VLDB Endowment*. ACM, 1942–1953.

[16] Alfons Kemper and Donald Kossmann. 1995. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *The VLDB Journal* 4 (1995), 519–566.

[17] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[18] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 197–208.

[19] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1307–1318. https://doi.org/10.1145/2882903.2903734

[20] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 743–754.

[21] 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.

[22] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[23] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 447–461. https://doi.org/10.1145/3341301.3359628

[24] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Haozhou Wang, Gang Guo, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Ashwin Agrawal, Alexandra Wang, Wen Lin, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. arXiv:cs.DB/2103.11080 https://arxiv.org/abs/2103.11080

[25] Miao Ma, Zhengyi Yang, Kongzhang Hao, Liuyi Chen, Chunling Wang, and Yi Jin. 2024. An Empirical Analysis of Just-in-Time Compilation in Modern Databases. In *Databases Theory and Applications*. Springer Nature Switzerland, Cham, 227–240.

[26] Pezzini Massimo, Feinberg Donald, Rayner Nigel, and Edjlali Roxane. 2014. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. (2014). https://www.gartner.com/en/documents/2657815

[27] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! but at what cost?. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*. USENIX Association, USA, 14.

[28] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment* 11, 1 (2017), 1–13.

[29] Freitag Michael, Kemper Alfons, and Neumann Thomas. 2022. Memory-Optmized Multi-Version Concurrency Control for Disk-Based Database Systems. *Proceedings of VLDB Endowment, PVLDB* 15 (2022), 2797–2810.

[30] Haubenschild Michael, Sauer Caetano, Neumann Thomas, and Leis Viktor. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. *Proceedings of the 2020 ACM SIGMOD* (2020), 877–892.

[31] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[32] MySQL [n.d.]. MySQL. https://www.mysql.com.

[33] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[34] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[35] Haoran Ning, Bocheng Han, Zhengyi Yang, Kongzhang Hao, Miao Ma, Chunling Wang, Boge Liu, Xiaoshuang Chen, Yu Hao, Yi Jin, Wanchuan Zhang, and Chengwei Zhang. 2024. Exploring Simple Architecture of Just-in-Time Compilation in Databases. In *Web and Big Data*. Springer Nature Singapore, Singapore, 504–514.

[36] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[37] PostgreSQL [n.d.]. PostgreSQL. https://www.postgresql.org/.

[38] Mohammad Sadoghi and Spyros Blanas. 2019. *Transaction processing on modern hardware*. Morgan & Claypool Publishers.

[39] Harizopoulos Stavros, Abadi Daniel J., Madden Samuel, and Stonebraker Michael. 2008. OLTP through the looking glass, and what we found there. *Proceedings of the 2008 ACM SIGMOD* (2008), 981–992.

[40] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 1150–1160.

[41] Neumann Thomas and Freitag Michael. 2020. Umbra: A Disk-Based System with In-Memory Performance. *10th Annual Conference on Innovative Data Systems Research, CIDR* (2020).

[42] TPC-C [n.d.]. TPC-C Benchmark. https://www.tpc.org/tpcc/.

[43] TPROC-C [n.d.]. HammerDB TPROC-C Workload. https://www.hammerdb.com/docs/ch03s03.html.

[44] Leis Viktor, Kemper Alfons, and Neumann Thomas. 2013. The Adaptive Radix Tree:ARTful Indexing for Main-Memory Databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE.

[45] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876.

[46] Wu Y, Arulraj J, Lin J, et al. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.