Chimera: Mitigating Ownership Transfers in Multi-Primary Shared-Storage Cloud-Native Databases

Chunyue Huang
Renmin University of China
Beijing, China
huangcy@ruc.edu.cn

Shuang Liu
Renmin University of China
Beijing, China
shuang.liu@ruc.edu.cn

Xinyi Zhang
Renmin University of China
Beijing, China
xinyizhang.info@ruc.edu.cn

Wenhao Li
Renmin University of China
Beijing, China
ruclwh@ruc.edu.cn

Wei Lu*
Renmin University of China
Beijing, China
lu-wei@ruc.edu.cn

Xiaoyong Du
Renmin University of China
Beijing, China
duyong@ruc.edu.cn

ABSTRACT

Cloud-native database systems with multi-primary shared-storage architecture have emerged due to their superior performance over primary-secondary architecture on write-intensive workload scenarios. However, these systems face performance degradation as the proportion of shared data increases, adversely affecting their Cost-Performance Ratio (CPR). In this paper, we identify frequent page ownership transfers between primaries as a key factor contributing to these performance bottlenecks. To address this challenge, we propose `Chimera`, a multi-primary database system that employs a two-phase transaction scheduling mechanism, combined with a delay-fetch ownership transfer strategy to effectively reduce the overhead of ownership transfers. Extensive experiments on SmallBank and TPC-C benchmarks demonstrate that `Chimera` outperforms existing schedule methods for multi-primary systems, achieving performance gains of 1.86× ~19.03× on throughput.

PVLDB Reference Format:
Chunyue Huang, Shuang Liu, Xinyi Zhang, Wenhao Li, Wei Lu, and Xiaoyong Du. Chimera: Mitigating Ownership Transfers in Multi-Primary Shared-Storage Cloud-Native Databases. PVLDB, 18(10): 3368-3381, 2025.
doi:10.14778/3748191.3748201

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/HuangDunD/Chimera.

1 INTRODUCTION

Recently, multi-primary architectures for cloud databases have attracted significant research interest. Compared to traditional primary-secondary architecture [1, 26, 50], they offer improved write scalability and fault tolerance, making them well-suited for write-intensive cloud-native applications. Existing multi-primary architectures can generally be categorized into two types: shared-nothing and shared-storage. In the shared-nothing architectures [10, 20, 47, 55], data is partitioned across nodes (also called primaries), with each partition managed exclusively by a single primary. Transactions involving only one partition are treated as local transactions and handled directly by the corresponding primary. However, it faces challenges with cross-partition transaction processing, necessitating the two-phase commit (2PC) [6] for consistency, which introduces potential performance bottlenecks.

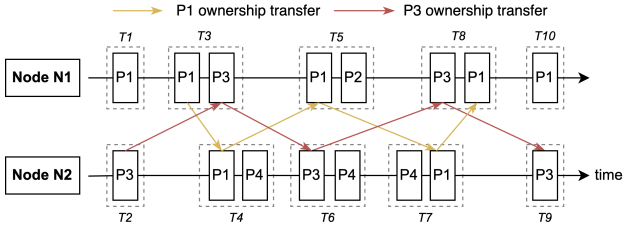

Figure 1: A motivating example. Poor transaction scheduling leads to frequent page ownership transfers.

In the shared-storage architecture [14, 27, 54], data is stored in the storage layer, and primaries can process any transaction without restrictions. When a primary N handles a read ($R(x)$) or write ($W(x)$) operation on data item x for a transaction T, the process involves these steps: ❶ $OT(x.p)$: Transfer ownership of the page $x.p$, where x resides, to N if it doesn't already have it. ❷ $LT(x.p)$: Acquire the *latch* on page $x.p$, ❸ $R(x.p)$: Read page $x.p$, ❹ $LK(x)$: Acquire a shared (for read) or exclusive (for write) *lock* on x, ❺$R(x)/W(x)$: Perform $R(x)$ or $W(x)$ on x, and ❻$UL(x.p)$: Release the *latch* on page $x.p$. As page $x.p$ is not bound to any specific primary, the remote operation $OT(x.p)$ is frequently invoked.

Page ownership transfers are costly in shared-storage architectures primarily because they require coordination through the *Global Page Lock Manager* (GPLM). When a primary N requests ownership of a page $x.p$ via $OT(x.p)$, the GPLM instructs the current owner to release it and grants ownership to N. This process, along with synchronizing page data, involves four network round-trip times (RTTs) (detailed in Section 2). To illustrate, we conduct an experimental study over the SmallBank benchmark [3] using the multi-primary architecture implemented in [27]. The results, reported in Table 1, show that when the ownership transfer rate is

* Wei Lu is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748201

Table 1: Effect of ownership transfer on performance.

The Percentage of Ownership Transfers	5%	15%	25%	35%	45%
Throughtput	1.0x	0.37x	0.24x	0.18x	0.15x

45%, the throughput is 85% lower compared to when the transfer rate is 5%. Experimental results from several commercial systems have similarly demonstrated a significant decline in performance as the proportion of shared data increases [14, 27, 54]. This is fundamentally attributed to the frequent transfer of ownership.

Existing methods to reduce page ownership transfers primarily depend on data partitioning, which assigns co-accessed pages to the same primary(i.e., the pages in this partition have an affinity with the primary). Transactions are then routed to the primary that holds most of the pages accessed according to the partitioning strategy. However, challenges arise in shared-storage designs where cross-node page accesses can disrupt consistency between actual page owners and their affinity-assigned nodes. Specifically, the existing "first-come, first-served" (FCFS) transaction scheduling strategy could lead to frequent and substantial transfers of page ownership among different nodes, a phenomenon we refer to as the "ping-pong" effect. Take Figure 1 as an example. The pages accessed by each transaction are marked with slashed rectangles. P_1 and P_2 are assigned to primary N_1 while P_3 and P_4 are assigned to primary N_2, and transactions are routed to different primaries based on the assignment. However, the problem of frequent transfers of ownership has not been mitigated due to poor scheduling of transactions, resulting in eight ownership transfers. In contrast, a different scheduling strategy (discussed later in Figure 5) results in only two ownership transfers. **Therefore, we argue that a judicious transaction scheduling strategy is crucial for reducing the number of ownership transfers, yet remains overlooked in existing literature on multi-primary shared-storage databases.**

To further reduce the number of ownership transfers and address the gaps in existing research, we formally define the transaction scheduling problem in multi-primary databases. Let $N.\mathcal{T}$ be the collection of transactions executed on primary N. $\forall T \in N.\mathcal{T}$, let $pot(T)$ be the number of pages not owned by node N during the time N executes T. Given a set of primaries $\mathcal{N}$, the number of page ownership transfers is quantified as $\Sigma_{N\in\mathcal{N}}(\Sigma_{T\in N.\mathcal{T}} pot(T))$. Intuitively, a primary N should retain ownership of its current page P as long as possible to process as many transactions routed to N accessing P before the ownership of P is transferred to other nodes. However, while the intuition reduces the number of page ownership transfers for P, it may make transactions routed to other nodes accessing P to wait for the ownership. Given this dilemma, we formalize that an ideal transactions scheduling strategy should maximize database throughput by minimizing the total number of ownership transfers, $\Sigma_{N\in\mathcal{N}}(\Sigma_{T\in N.\mathcal{T}} pot(T))$ while ensuring that the latency of executing each transaction remains reasonable.

Solving the transaction scheduling problem is non-trivial for several reasons. First, predicting future transactions is inherently difficult, making it hard to determine how long a primary should keep ownership of its pages to avoid the "ping-pong" effect while preventing blocking on other nodes. For example, it is hard to decide whether a primary N should release ownership of P when other nodes need it, as future transactions routed to N may access P. Second, acquiring page ownership has global effects, causing transactions on the other primaries to wait. Even worse, in a multi-primary database, each primary schedules transactions independently, without visibility into the transactions scheduled on other nodes. This lack of coordination makes it more difficult to achieve global optimization of page ownership transfers.

Given the challenges above, we propose `Chimera`, a shared-storage cloud-native database system that adopts a generic multi-primary design while specifically focusing on solving the transaction scheduling problem without assuming knowledge of future transactions or having visibility into transactions on other nodes. To achieve this, `Chimera` employs a **two-phase transaction scheduling** (short for 2PS) mechanism, which divides transaction execution into two alternating phases: the partitioned phase and the global phase. For clarity, transactions that access pages affiliated with a single primary are referred to as *intra-node* transactions, while transactions accessing pages across multiple primaries are referred to as *inter-node* transactions. In the partitioned phase, each primary retains ownership of pages within its affinity partition, and only *intra-node* transactions are scheduled, maximizing the number of transactions performed on pages already residing in each primary without ownership transfers. The *inter-node* transactions are deferred to the global phase for execution. A well-designed switching mechanism ensures that the latency of transactions stays close to *group commit* [15] latency and remains imperceptible to users.

Additionally, we introduce a **delay-fetch ownership transfer** (short for **delay-fetch**) mechanism in the global phase where primaries compete for ownership of pages necessary for processing *inter-node* transactions to avoid the "ping-pong" effect. Specifically, for any operation of node N accessing a page P owned by other nodes, the system delays acquiring ownership of P until either a sufficient number of operations targeting P have accumulated, or a predefined timeout is reached. After N acquires ownership of the delayed page P, all operations targeting this page during the delay period can be scheduled within a single ownership cycle, thereby reducing ownership transfers significantly. To achieve fine-grained scheduling, we designed a proactive transaction switching algorithm, ensuring that transaction operations are fully executed without unnecessary blocking. We leverage coroutines to implement the suspend-and-resume mechanism of transaction for this algorithm. The primary contributions are summarized below:

- We identify the frequent page ownership transfer as a primary factor limiting the performance of multi-primary database systems in cloud-native shared-storage architectures.
- In order to reduce the frequent page ownership transfer cost, we introduce `Chimera`, which employs a novel two-phase transaction scheduling mechanism, eliminating ownership transfers for *intra-node* transactions processed in the partitioned phase.
- We propose a delay-fetch ownership transfer mechanism that groups the operations accessing the same pages, further reducing the ownership transfers in the global phase.
- We present a theoretical analysis demonstrating that `Chimera` effectively reduces the ownership transfer cost. Experimental results showcase its superior performance and scalability compared to existing scheduling algorithms.

2 BACKGROUND

In this section, we shall describe how primaries correctly read and write pages in a multi-primary cloud-native database.

2.1 Hierarchical Page Locking Mechanism

In the single-primary database architecture, *latch* (a local short-term page lock) is commonly used to prevent other threads from reading/writing a page while one thread is writing to it. Before a thread operates on a page, the thread competes for the latch of this page to ensure that no conflicting operations are performed. When the system scales to a multi-primary architecture, latches alone cannot resolve conflicts caused by concurrent operations from multiple primaries, as a latch is maintained in a primary's local memory and is not visible to other primaries. Therefore, the global page lock is introduced to coordinate the concurrent accesses to one page by multiple primaries.

Most multi-primary database systems employ a hierarchical page-lock mechanism to avoid conflicts among multiple primaries accessing pages concurrently [14, 54]. This model comprises a *Global Page Lock Manager* (GPLM), a global component responsible for managing page ownership across primaries, and a *Local Page Lock Manager* (LPLM), maintained within each primary to manage latches among local threads. When a transaction within a primary needs to read or write a page, the node must hold the ownership of the page, and then it can compete for the latch. Only then can the read/write operation be performed.

Figure 2 illustrates an example of two primaries N_1 and N_2, concurrently accessing pages in a multi-primary architecture. Each primary maintains its own LPLM and buffer pool. The LPLM keeps track of the latch and ownership status of pages within the *latch* and *global lock* fields, respectively. The buffer pool caches pages like most centralized databases [28, 35]. The GPLM is deployed on a meta server to track the ownership of each page across nodes. For example, N_1 and N_2 hold the shared ownership of P_1 in Figure 2. Suppose a transaction on N_2 is attempting to write to P_{101}. It first checks with its LPLM and detects that the global lock is 0, indicating that N_2 does not hold ownership of P_{101}. Consequently, it must request the exclusive ownership on P_{101} from GPLM (①) via a network message. When GPLM receives the request, it tracks that P_{101} is now owned by N_1 (indicated by its *Lock* field), and when N_1 releases the ownership, the GPLM grants ownership of P_{101} to N_2 (④) and informs N_2 of the successful ownership granting, accompanied by the page validity information of P_{101} from GVT (⑤), which will be described in detail in the next subsection.

There are two strategies for releasing the ownership of a page. A straightforward strategy is to release ownership as soon as the primary no longer needs the page, which we refer to as ***Eager-Release*** in this paper. For example, when a transaction releases its latch on P_{101}(②), it checks the reference of P_{101} indicating the number of pending read/write operations on P_{101} within the node. If the reference count reaches zero, N_1 releases the ownership of P_{101} back to GPLM(③-a) instantly. Another strategy is called ***Lazy-Release***, where the ownership is not released voluntarily by the primary holding the ownership of the page. If another primary requests it, GPLM notifies the current holding node to release the ownership. For example, assume that before N_2 requests ownership

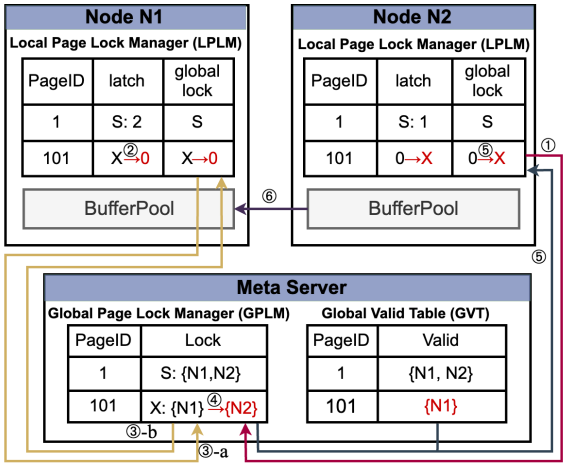

Figure 2: Concurrent page access in multiple primaries.

of P_{101}, N_1 releases the latch on P_{101} and the reference count of P_{101} drops to zero. At this point, the ownership of P_{101} on N_1 becomes available for release, but it is still temporarily retained by N_1. If N_1 reads/writes P_{101} again, N_1 can compete for the latch directly without acquiring ownership from GPLM. When N_2 requests ownership of P_{101} from GPLM, the GPLM receives the request and notifies N_1 to release the ownership on P_{101}(③-b). Since *Lazy-Release* significantly reduces the number of remote ownership requests, it is widely adopted by most systems. Therefore, the discussions in this paper are based on this strategy.

2.2 Distributed Cache Coherence

When a transaction obtains page ownership and writes to the page, the corresponding pages in the buffer pools of other nodes become stale. Distributed cache coherence protocols are used to ensure that all nodes can access the latest data. Cache coherence is typically maintained through two main approaches: **snooping-based** and **directory-based** protocols. Snooping-based cache coherence protocols [2, 38, 43] rely on extensive broadcasting and are well-suited for buses in multi-core processors. However, in multi-primary systems where multiple nodes are connected via a network, increasing nodes significantly raises network overhead due to broadcasting.

In multi-primary cloud-native databases, most systems use the directory-based cache coherence protocol [27, 54], which keeps track of the state of cache entities at a designated location known as the meta server. The nodes access the directory, a structure we refer to as the *Global Valid Table* (GVT), to determine whether the cached entity is invalid. As shown in Figure 2, recall that when N_1 writes P_{101} and subsequently releases the ownership of the page, the cached page of P_{101} in the buffer pools of N_2 becomes stale. Therefore, N_1 clears the set of currently valid nodes in the GVT and assigns its node id in the *Valid* field of P_{101} when releasing the ownership. This makes nodes that access this page later aware that the latest page is located on N_1. When GPLM grants N_2 the exclusive ownership of P_{101}, GPLM sends the set of valid nodes with the latest of P_{101} to N_2 (⑤). If N_2 is not included in this set, N_2 synchronizes the latest page with any valid node (⑥). If the valid set is empty,

it means that no primary caches the latest version in its buffer pool, prompting a synchronization request to be sent to the storage layer. The process by which a primary requests ownership and is subsequently granted ownership, along with the synchronization of the page, is referred to as an **ownership transfer**.

3 ARCHITECTURE OVERVIEW

Chimera is a multi-primary, cloud-native OLTP database that adopts the compute-storage disaggregated architecture. As depicted in Figure 3, Chimera consists of a shared storage layer that stores data pages and logs, a meta server maintaining the GPLM and GVT, multiple primaries for transaction execution, a transaction router for forwarding transactions to primaries, and a coordinator responsible for scheduling the phase of all nodes and ensuring fault tolerance. The components of Chimera are interconnected via TCP/IP-based Local-Area-Network (LAN) within a data center. In the following, we describe each component of Chimera in detail.

Transaction Router. The transaction router is responsible for receiving transactions from clients on the fly and routing them to a primary for processing. To enable efficient transaction routing, Chimera establishes affinity between data and primaries by logically partitioning the database using range partitioning. Each page only stores tuples that belong to the same partition. The number of partitions matches the number of primaries with the partitioning strategy defined by users or adaptive workload-aware methods [12, 39, 40, 62]. The router maintains *partition meta*, where each partition Par_i is uniquely affiliated with a primary N_i. Upon receiving a transaction, the router identifies its accessed partitions. If the transaction accesses only one partition, it is classified as an ***intra-node*** transaction and forwarded to the primary affiliated with this partition. Conversely, if the transaction spans multiple partitions, it is classified as an ***inter-node*** transaction and forwarded to one of the primaries associated with the accessed partitions.

Chimera assumes that at least one key in the read/write set is known, rather than requiring the entire read/write set to be pre-acknowledged [17, 18, 45, 48, 63]. This design is particularly friendly to transactions with internal read-write dependencies. The transaction router utilizes the exposed keys to determine the partitions accessed by a transaction. While this may misclassify *inter-node* transactions as *intra-node* transactions, the correctness of the system remains unaffected, as explained in Section 4. The transaction router can scale out without performance bottlenecks.

Coordinator. Chimera proposes a two-phase transaction scheduling mechanism, which divides the system execution into partitioned phases and global phases, executing in interleaving. The coordinator is responsible for synchronizing phase states across all primaries and adjusting phase durations in Chimera, as detailed in Section 4. Additionally, it monitors the heartbeat of each primary. In the event of a primary failure, the coordinator detects the issue and informs the remaining active nodes to recover and ensure consistency. Further details are provided in Section 7. The coordinator can be deployed on one or multiple physical machines and maintains high availability through Paxos or Raft protocols.

Primary nodes. A primary in Chimera can execute any transaction and access all database pages. The transaction execution process in a primary follows the two-phase transaction scheduling

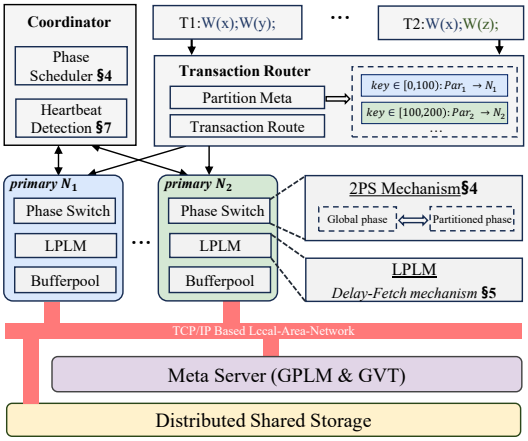

Figure 3: The architecture of Chimera.

mechanism, which transits between the partitioned phases and global phases interleavingly (detailed in Section 4). The primary executes transactions using the MV2PL [5, 16] protocol, a variant of the MVCC [53] protocol, and each tuple in pages maintains the row lock. When a transaction has taken ownership and latch of a page, and is going to read or write a tuple, it acquires the row lock on this tuple. If another transaction already holds the row lock, the system follows a No-Wait deadlock prevention policy [4], immediately aborting the transaction. The row lock is held until the transaction is committed. Additionally, each primary maintains a local page lock manager (LPLM), which manages the latches of the node. Chimera introduces a delay-fetch ownership transfer mechanism to reduce the ownership transfer frequency during the global phase, which is detailed in Section 5.

Meta Server. To manage page ownership and ensure cache coherence, Chimera employs a meta server and includes the global page lock manager (GPLM) and the global valid table (GVT), as described in Section 2. The meta server can be distributed by partitioning the metadata. As metadata updates are performed alongside ownership transfers during transaction execution, rather than at the granularity of entire transactions, the updates do not involve cross-partition operations. This design enables the meta server to scale out without introducing coordination overhead.

Distributed Shared Storage. The shared storage pool comprises nodes equipped with high-capacity SSDs, providing data access interfaces for primaries. Chimera ensures high availability by maintaining multiple replicas and employing Paxos [25] or Raft [37] protocols for data replication. Following the *Log is the database* principle [50], primaries persist write-ahead logs (WALs) to the storage layer before committing transactions. Storage nodes asynchronously replay these logs to update pages, minimizing network I/O by avoiding direct page flushing.

Similar to many high-performance transactional databases [33, 44, 60], Chimera is primarily designed for stored procedure-based transactions, though it also supports long-running and interactive transactions, as explained in Section 4.4. Chimera defaults to *serializable* isolation and also supports *read-committed* isolation. The brief proof of serializability is provided in Section 7.1.

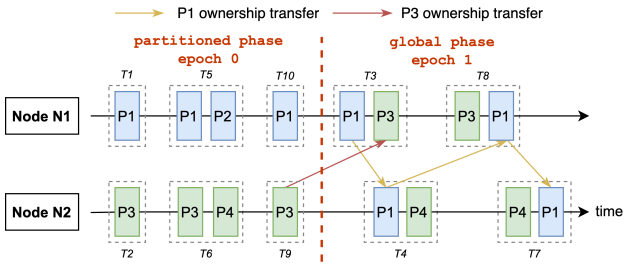

Figure 4: Chimera divides the system into the partitioned phase and the global phase

4 TWO-PHASE TRANSACTION SCHEDULING MECHANISM

This section explains how the two-phase transaction scheduling (2PS) mechanism in Chimera helps reduce ownership transfers between *intra-node* and *inter-node* transactions. Chimera divides the system into two alternating phases: the partitioned phase and the global phase, with the system starting in the partitioned phase. Algorithm 1 describes the detailed steps of our 2PS algorithm. Each primary in Chimera includes a phase-switching thread to manage transitions between two phases and multiple worker threads responsible for executing transactions. The two phases last for dynamically determined durations, with the partitioned phase running for t_p ms and the global phase for t_g ms, as detailed in Section 4.3. Upon startup, the primary initializes its resources and the phase-switching thread notifies the coordinator of its readiness to enter the partitioned phase (line 4). Once all primaries report readiness, the coordinator responds to the sysEpoch function and signals them to commence the partitioned phase.

Algorithm 1: 2PS Mechanism

```
1  Function PhaseSwitchThread():
2      epoch_id ← 0, node.phase ← phase :: init;
3      while node.is_running do
4          epoch_id ← sysEpoch (epoch_id) ;  // sys all nodes
5          invalid_page_set ← requestPagesInPartition ();
6          node.phase ← phase :: partitioned;
7          sleep(t_p) ;            // run intro-node txns for t_p ms
8          node.phase ← phase :: switching;
9          waitAllWorkersStop ();
10         epoch_id ← sysEpoch (epoch_id);
11         node.phase ← phase :: global;
12         sleep(t_g) ;            // run inter-node txns for t_g ms
13         node.phase ← phase :: switching;
14         waitAllWorkersStop ();
15         releasePagesOutPartition ();
16     end
17 return
18 Function WorkerThread():
19     while node.is_running do
20         if node.phase = phase :: switching then
21             stopThisThread ();
22         else if node.phase = phase :: partitioned then
23             runIntraNodeTxns () ;  // exec a intra-node txn
24         else if node.phase = phase :: global then
25             runInterNodeTxns () ;  // exec a inter-node txn
26         end
27     end
28 return
```

4.1 Partitioned Phase

Receiving the coordinator's message indicates that each primary is allowed to enter the partitioned phase. Before execution, each primary requests from the GPLM to acquire ownership of all pages indicated by its affinity partition yet it does not currently own. At the same time, it requests from the GVT the set of pages that have already been invalidated within its partition (line 5). After that, the node transitions its state to *partitioned* (line 6) and can read and write to pages within its affinity partition freely. Worker threads detect this state change and begin executing transactions (line 23).

During the partitioned phase, the primary processes only *intra-node* transactions, similar to shared-nothing systems handling non-distributed transactions. If a transaction, incorrectly identified by the router, accesses a page outside its affinity partition, it is aborted and deferred for re-execution in the global phase. As shown in the Figure 4, pages P_1 and P_2 (blue) are affiliated with N_1, while P_3 and P_4 (green) are affiliated with node N_2. During the partitioned phase, N_1 executes transactions that only access P_1 and P_2 (e.g., T_1, T_5, T_{10}), whereas other transactions (e.g., T_3, T_8) are deferred to the global phase for execution.

Since the logical partitions handled by each primary are non-overlapping, there are no page conflicts between nodes during the partitioned phase. This allows primaries to execute transactions locally without requiring page ownership transfers or encountering newly generated page invalidations. During this phase, when an *intra-node* transaction accesses a page, it does not need to check the global lock field in the LPLM. Instead, it only verifies whether the page is in the invalidated set. If it is, the primary synchronizes the page to the latest version, removes it from the invalidated set, and then competes for the latch to perform read or write operations using the MV2PL protocol. Notably, the invalidated set allows pages to be synchronized only when necessary, which reduces unnecessary data page transfer overhead.

After running the partitioned phase for t_p milliseconds, the phase-switching thread transitions the phase state to *switching* (line 8) and waits for all worker threads to complete their current transactions (line 9). Once the worker threads finish their ongoing transactions, they pause and wait for the system to transition to the next phase (line 21).

4.2 Global Phase

Once the coordinator receives readiness messages from all primaries to enter the global phase, it notifies them to start the global phase (line 11). Any transactions, including those initially marked as *inter-node* transactions by the router or identified as a *inter-node* transaction during execution in the partitioned phase, are handled

during the global phase. Unlike the partitioned phase, primaries in this phase can access any page across all logical partitions. At the start of the global phase, primaries only own the pages within their affinity partition. As shown in Figure 4, both N_1 and N_2 can process transactions that operate on any partitions during the global phase. When T_4 needs to access P_1, it checks the global lock status of P_1 in the LPLM on N_2. Since T_3 on N_1 has just written P_1 and holds ownership, N_2 must request GPLM to transfer exclusive ownership of P_1 from N_1 and synchronize the page. After that, T_4 writes tuples on P_1. Subsequently, both T_8 and T_7 follow the same process to write P_1. When the global phase runs for t_g milliseconds and all worker threads have stopped, the primary releases ownership of all locally held pages outside its partition and updates the GVT table if it was operated during the ownership period (line 15).

The 2PS mechanism eliminates the ownership transfers for *intra-node* transactions, as the ownership of pages accessed by these transactions does not transfer during the partitioned phase. As shown in Figure 1, eight ownership transfers occur during the execution of ten transactions. However, in Figure 4, *intra-node* and *inter-node* transactions do not interfere with each other, resulting in only four ownership transfers.

4.3 Dynamic Phase Durations

In `Chimera`, the durations of the partitioned phase (t_p ms) and the global phase (t_g ms) are dynamically determined, while the total iteration time remains fixed. As shown in Equation 1, the partitioned phase and the global phase each constitute an epoch, and their combined durations form the iteration time, denoted as e. The value of e is adjustable and can be configured by users. Its default value is aligned with the *group commit* interval set by the system, such as the *binlog_group_commit_sync_delay* parameter in MySQL. *Group commit* is a strategy commonly adopted by most databases [15], where a group of transactions in a specified interval is delayed and committed in batch, amortizing the cost of flushing Write-Ahead Log (WAL). This ensures the blocking time for most transactions remains below or near the *group commit* latency, preventing a noticeable increase in perceived latency.

$$t_p + t_g = e \tag{1}$$

$$\frac{\alpha_p t_p}{C} - \frac{\alpha_g t_g}{1 - C} = 0 \tag{2}$$

The coordinator in `Chimera` periodically collects each primary's throughput in the partitioned phase, α_p, and in the global phase, α_g, as well as the ratio of *intra-node* transactions, C. These metrics allow the coordinator to calculate and communicate the appropriate durations for both phases to all primaries according to Equation 2. The principal goal is to align the ratio of transactions executed in the two phases with the actual workload distribution. This minimizes the risk of certain transactions being deferred across multiple epochs before execution. Phase switching in the system is a strongly synchronized operation among primaries. However, since each phase typically lasts hundreds of milliseconds, the switching overhead has minimal impact on system performance. Note that transactions are assumed to arrive randomly and uniformly, with any transaction being executed before the end of the subsequent phase. Thus, the latency of transactions between partitioned and global phases is symmetric. To reduce logging overhead, `Chimera` adopts the *group-commit* approach, where at the end of each partitioned and global phase, all transactions from that epoch are committed across all partitions. Therefore, the expected transaction latency is $\frac{1}{2}e$.

4.4 Support for Long-Running Transactions

`Chimera` supports both long-running stored procedures and interactive transactions that may span multiple phases. When the system enters the *phase::switching* state, denoted as t_s, worker threads are granted additional time, t_Δ, to complete ongoing transactions. The duration is approximately the execution time of a short transaction, ensuring that most short transactions can finish within $t_s + t_\Delta$. After this moment, any remaining transactions are then suspended and resume only when the corresponding phase begins in the next cycle. Transactions in the next phase that access tuples held by suspended transactions will abort, avoiding system blocking. This behavior is inherent under the default `No-Wait` deadlock prevention strategy. When extended to `Wait-Die`, each primary can broadcast its list of suspended transactions during the phase switch, enabling each primary to recognize transactions suspended on other nodes.

We implement the suspend-and-resume mechanism using coroutines. Each thread contains two coroutines: $Coro_0$ handles *intra-node* transactions during the partitioned phase, while another coroutine, $Coro_1$, processes transactions in the global phase. Phase checks are inserted at the end of page operations. If the current time exceeds $t_s + t_\Delta$, the worker thread halts the execution and signals its readiness to transition to the next phase. In the subsequent phase, the worker thread switches to another coroutine to execute transactions. The previously suspended coroutine resumes processing after the next transition. Notably, in the next section, we use multiple coroutines, $Coro_{1..n}$, to process transactions during the global phase. Similarly, the system transitions to the next phase and switches to $Coro_0$ only when all coroutines, $Coro_{1..n}$, are suspended.

5 DELAY-FETCH OWNERSHIP TRANSFER MECHANISM

Although the 2PS mechanism eliminates the transfer of ownership for *intra-node* transactions, frequent transfers remain noticeable during the global phase. As illustrated in Figure 4, transactions T_3, T_4, T_8, and T_7 alternately access P_1 between two primaries, resulting in three ownership transfers.

To address this issue, `Chimera` incorporates a delay-fetch ownership transfer mechanism, which delays the timing of fetching a page to further reduce the frequency of ownership transfers. Specifically, when a primary encounters a page ownership miss while processing transactions, `Chimera` does not instantly request the ownership from the GPLM. Instead, the request is added to a *request set*, which is maintained by each primary and shared among its worker threads. The deferred transaction sends its ownership request to the GPLM when the number of accumulated requests reaches a certain threshold or a timeout occurs. This approach leverages the delay window to group more transactions operating on the same page, ensuring that these operations can be completed within a single ownership cycle.

The delay-fetch mechanism operates during transaction execution, making it challenging to yield the CPU and efficiently handle

Algorithm 2: Delay-Fetch Ownership Transfer Mechanism

```
1  Function LockPage(lock_mode):
2      enter, lock_success ← false;
3      while ¬enter do
4          if is_notified_release then Yield();
5          else
6              enter ← true;
7              ref ← ref + 1;
8          end
9      end
10     while ¬lock_success do
11         if global_lock covers lock_mode then
12             lock_success ← LockLocal();
13             if lock_success then ref ← ref - 1;
14         else
15             if ¬in_request_set then
16                 ts_ddl ← t.now() + timeout;
17                 in_request_set ← true;
18             else if ref > ref_thold or t.now() > ts_ddl then
19                 LockRemote();
20             else Yield();
21         end
22     end
23 return
```

other transactions during the delay period. Traditional techniques rely on an excessive number of worker threads (e.g., 256 worker threads on a 12-core CPU) and the operating system controlled schedule, potentially leading to excessive and random thread context switching and significantly degrading system performance. To address this, we designed a proactive transaction-switching algorithm to ensure efficient system operation during the delay period.

Algorithm 2 describes how a transaction acquires the page lock when accessing a page under the delay-fetch mechanism. When accessing a page, the transaction calls `LockPage()` (line 1) with the desired lock mode (e.g., shared or exclusive). First, it initializes two key variables: *enter*, indicating whether the operation can proceed, and *lock_success*, tracking lock acquisition status (line 2). If the page has not yet been notified to release ownership, the operation proceeds to the locking process and increases the page counter, ref (lines 5–7). Next, the algorithm checks whether the current node has held the corresponding ownership (line 11), if so, it attempts to grant the latch locally (line 12). Upon success, the page counter ref of the corresponding page is decremented (line 13). Otherwise, it needs to acquire global page ownership from the GPLM. If the page is not yet in the request set, the transaction adds it to the set and calculates the timeout threshold, ts_ddl (lines 15-17) for this request. If the page is already in the request set, the transaction checks whether ref has reached the threshold, ref_{thold}, or the current time has exceeded ts_ddl. If either condition is met, a remote lock request is issued (lines 18-19). Otherwise, we proactively perform a context switch (i.e. `Yield()`) to process another transaction (line 20).

To prevent starvation, if other nodes request ownership of this page ($is\_notified\_release = True$), further locking attempts from

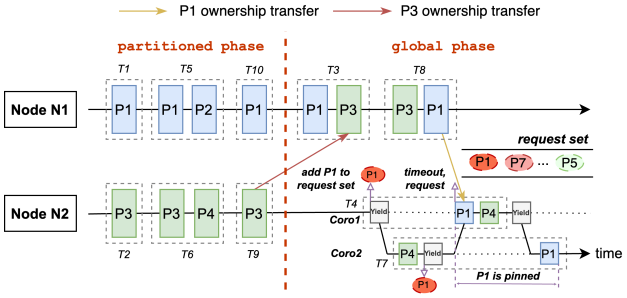

Figure 5: `Chimera` utilizes the delay-fetch mechanism to reduce ownership transfers further.

this node on the page are not permitted during the current ownership cycle. The algorithm switches to the next transaction to process (line 4). This step is necessary because other active transactions will not resume execution unless we proactively switch. A potential deadlock scenario occurs when a suspended transaction has acquired page ownership but has not resumed. If another node subsequently requests the page, the request is blocked since ownership is only released when ref reaches zero.

Implementation. To implement this proactive transaction switching algorithm, we leverage coroutines to suspend and resume transactions. Specifically, we allocate multiple coroutines, $Coro_{1..n}$, for each worker thread, with each coroutine bound to a transaction execution function. Within a worker thread, these coroutines are organized in a circular linked list. The collaboration of these coroutines is managed by a coroutine scheduler. When a transaction switch is triggered, the current coroutine yields to the next coroutine in the list, ensuring that all transactions have an equal opportunity to be scheduled. Notably, the `Yield()` process in the algorithm aligns perfectly with the coroutine switching interface (i.e. `yield(fun())`), making the implementation surprisingly seamless.

Example 5.1. Figure 5 provides a concrete example of the delay-fetch mechanism. During the global phase, $Coro_1$ on N_2 begins executing T_4, which first attempts to write to P_1. However, as N_2 does not currently hold ownership of P_1, $Coro_1$ performs a context switch (i.e., yield) and adds P_1 to the local request set instead of immediately issuing a remote request. The coroutine scheduler then switches to $Coro_2$ to process T_7. Since N_2 holds ownership of P_4, T_7 locks the target tuple and updates it. Later, T_7 also attempts to write to P_1, but since P_1 is already in the request set and has not timed out, the scheduler switches back to $Coro_1$. When $Coro_1$ resumes, it finds the ownership request for P_1 has timed out and issues a request to the GPLM. Once ownership of P_1 is acquired, T_4 attempts to write P_1 and P_4. If the target tuple is locked by another transaction, T_4 follows the `No-Wait` policy and aborts to ensure serializability. After T_4 finishes, the scheduler switches back to $Coro_2$ to process its P_1 operation in T_7. Ownership of P_1 is retained until both transactions complete their operations on it.

□

Conditional Delay. The delay-fetch mechanism has two potential problems. First, the delay-fetch mechanism does not benefit all pages uniformly. For instance, delaying access to "cold" pages

may fail to group subsequent transactions targeting the same page. Second, setting an excessively high threshold for the expected reference count or timeout can cause all coroutines in a worker thread to enter a delayed state, leading to CPU idling. To address these challenges, `Chimera` employs a conditional delay strategy. Specifically, each primary periodically collects page access frequency statistics and identifies the top $\mathcal{H}$ most frequently accessed pages to apply the delay-fetch mechanism. The reference threshold is set to $\mathcal{K}$, and the timeout value is determined empirically. Assuming a primary has $\mathcal{W}$ worker threads, each equipped with $\mathcal{Q}$ coroutines.

$$\mathcal{H} \cdot \mathcal{K} - \mathcal{W} \cdot (\mathcal{Q} - 1) \leq 0 \tag{3}$$

Equation 3 provides the relationship governing this process. The maximum number of transactions managed by the coroutines is $\mathcal{W} \cdot \mathcal{Q}$, while the maximum number of transactions triggering the delay-fetch mechanism is $\mathcal{H} \cdot \mathcal{K}$. This equation ensures that, under most conditions, at least $\mathcal{W}$ transactions will not enter the delay state, allowing the system to receive and process transactions consistently. Meanwhile, the conditional delay mechanism selectively applies the delay-fetch strategy to hot pages, ensuring that delaying ownership acquisition for these pages is generally beneficial.

6 ANALYSIS OF OWNERSHIP TRANSFERS REDUCTION

In this section, we analyze the reduction in ownership transfers achieved by the 2PS and delay-fetch mechanism. Consider a system with $\mathcal{N}$ primaries ($N_{1...n}$). For a given page P_i, let $\mathcal{M}$ denote the total number of operations performed on P_i within one iteration. The proportion of *intra-node* transactions is C, and for *inter-node* transactions, each primary accesses P_i with equal probability.

Assume P_i has an affinity with node N_1. Operations by an *intra-node* transaction on P_i at N_1 are denoted as $N_1^{\mathcal{P}}$, while those performed by an *inter-node* transaction at any node N_i are denoted as $N_i^{\mathcal{G}}$. The sequence of operations on P_i can be represented as: $[N_1^{\mathcal{P}}, N_2^{\mathcal{G}}, N_1^{\mathcal{G}}, N_3^{\mathcal{G}}, \dots]$. Two operations are considered an ownership transfer if they are adjacent but executed on different nodes.

As a result, $(C + \frac{1-C}{\mathcal{N}})\mathcal{M}$ represent the number of operations on P_i performed by N_1, which we abbreviate as O_1, and $\frac{(1-C)(\mathcal{N}-1)}{\mathcal{N}}\mathcal{M}$ denote the total operations by $N_{2...n}$, denoted as $O_{2...n}$. We evaluate the worst-case scenario for different scheduling strategies.

FCFS Scheduling.

- **Case 1:** $O_1 \geq O_{2...n}$**.** In this arrangement, the worst schedule occurs when the operations of $N_{2...n}^{\mathcal{G}}$ alternate with those of $N_1^{\mathcal{P}}$ and $N_1^{\mathcal{G}}$. The sequence of operations follows the pattern $[N_1^{\mathcal{P}}, N_1^{\mathcal{P}}, N_2^{\mathcal{G}}, N_1^{\mathcal{G}}, N_3^{\mathcal{G}}, N_1^{\mathcal{P}}, \dots]$. Therefore, the number of ownership transfers is approximately:

$$\mathcal{T}_{FCFS} = \frac{2(1-C)(\mathcal{N}-1)}{\mathcal{N}} \cdot \mathcal{M} \tag{4}$$

- **Case 2:** $O_1 < O_{2...n}$**.** In this case, since the probabilities of each primary accessing P_i in *inter-node* transactions are equal, the number of nodes satisfies $\mathcal{N} \geq 3$. Under worst-case scheduling, the operation sequence can alternate sequentially among $N_{2...n}$, like $[N_2^{\mathcal{G}}, N_3^{\mathcal{G}}, \dots, N_n^{\mathcal{G}}, N_2^{\mathcal{G}}, \dots]$. And then $N_1^{\mathcal{P}}$ and $N_1^{\mathcal{G}}$ operations are interspersed within this sequence. This implies that each consecutive operation triggers an ownership transfer. Hence, the total number of ownership transfers is given by:

$$\mathcal{T}_{FCFS} = (C + \frac{1-C}{\mathcal{N}})\mathcal{M} + (\frac{(1-C)(\mathcal{N}-1)}{\mathcal{N}})\mathcal{M} = \mathcal{M} \tag{5}$$

Add 2PS Scheduling. With the introduction of the 2PS mechanism, ownership transfers for $N_1^{\mathcal{P}}$ are eliminated entirely. To simplify the analysis, we disregard minor ownership operations that may occur during phase transitions. For $N_1^{\mathcal{G}}, \dots, N_N^{\mathcal{G}}$, the worst-case scenario resembles Case 2 under the FCFS scheduling strategy, where ownership rotates sequentially among all primaries. Consequently, the total number of ownership transfers can be expressed as:

$$\mathcal{T}_{+2PS} = (1-C)\mathcal{M} \tag{6}$$

Add Delay-Fetch Scheduling. When the delay-fetch mechanism is further introduced, for operations $N_1^{\mathcal{G}}, \dots, N_N^{\mathcal{G}}$, we assume that the coroutine mechanism captures up to $\mathcal{K}$ operations accessing the same page. Thus, within a single ownership cycle, the minimum number of operations that can be executed is $\mathcal{K}$. Consequently, the number of ownership transfers can be expressed as:

$$\mathcal{T}_{+2PS+delay\text{-}fetch} = \frac{(1-C)\mathcal{M}}{\mathcal{K}}, (K \geq 1) \tag{7}$$

Based on the above formula, we derive the following inequality:

$$\mathcal{T}_{FCFS} \geq \mathcal{T}_{+2PS} \geq \mathcal{T}_{+2PS+delay\text{-}fetch} \tag{8}$$

This theoretically demonstrates that the 2PS and delay-fetch mechanism can separately reduce the number of ownership transfers.

7 DISCUSSION

7.1 Correctness

Physical consistency. In `Chimera`, primaries must hold page ownership and synchronize pages to the latest before read/write operations, ensuring the physical consistency of serializing page modifications among primaries. At the end of global phases, pages that do not belong to the affinity partition of the primary are released and GVT is updated, with the coordinator ensuring strong synchronization of phase transitions. During the partitioned phase, primaries exclusively own pages and retrieve the set of invalid pages from GVT within their partitions, restricting access by other nodes. In the global phase, primaries access the GPLM and GVT to acquire ownership and valid information for pages they do not own and update them upon releasing ownership. Since all operations on any page are completed within a single phase, `Chimera` ensures physical consistency even for long-running transactions.

Serializability Both the partitioned and global phases employ the MV2PL concurrency control algorithm which guarantees serializability. The serial order of transactions corresponds to the order in which row locks are acquired. In the global phase, the ownership acquisition is delayed, but transactions do not acquire row locks until they have obtained page ownership, thereby it does not affect the serializable scheduling of transactions.

7.2 Fault Tolerance

Leveraging the multi-primary architecture, Chimera can continue processing transactions on the remaining live primaries even if some nodes fail. The coordinator tracks the status of all primaries within the cluster. Upon detecting a node failure, the system redistributes the pages of the logical partition owned by the failed node to active nodes and updates the router accordingly.

For page ownership held by a failed node, the GPLM periodically monitors and proactively releases ownership, enabling other primaries to re-acquire it as needed. Chimera employs the WAL logging technique, utilizing the Logical Log Sequence Number (LLSN) [54], to establish a partial order for logs related to the same page across different primaries. Redo/Undo logs are forced to flush at two critical points. First, logs are flushed to disk before transactions are committed. If a primary caching the latest page fails, other primaries can replay committed transactions using the redo log. Second, when a primary releases page ownership or evicts a page from the buffer pool, any relevant redo/undo logs must be flushed to disk. This process ensures that, before any primary modifies a page, all previous logs of changes made by other nodes to that page have been safely persisted. In the event of a node failure, the surviving primaries can use the LLSN to undo uncommitted transactions from the failed node, thus maintaining database consistency.

8 EVALUATION

8.1 Experimental Setup

8.1.1 Setup. We evaluate our system on nine machines in an Enhanced Data Rate (EDR) cluster. Eight nodes serve as primaries, each equipped with 16-core 2.30 GHz virtual Intel(R) Xeon(R) Gold 5218 CPUs and 40GB of DRAM. An additional node, equipped with 32 cores of the same CPU model and 80GB of DRAM, is used to deploy the GPLM, GVT, and storage services. The network between nodes delivers over 10 Gbits/s throughput as measured using `iperf`. The nodes communicate via bRPC[7]. Boost.coroutine2 from Boost 1.63.0 is used to manage coroutines. In our experiment, we deploy 12 worker threads on each node.

8.1.2 Workloads. We evaluate our algorithms using two widely-used OLTP benchmarks, Smallbank [3] and TPC-C [49].

Smallbank. SmallBank is a popular benchmark for evaluating OLTP systems. It simulates a banking environment with operations typical of financial transactions and consists of five primary transactions. In our evaluation, we set the number of accounts to 300,000, some of which are designated as hotspots. The proportion of hotspot accounts is adjustable, while their access frequency is fixed at 80%. Real-world workloads often exhibit spatial locality in data access patterns [13, 42]. To simulate this characteristic, we prioritize placing hot data items on the same page by default. Under this layout, the range of hotspot items and hotspot pages is roughly equal. Hotspot pages are evenly distributed across primaries.

TPC-C. The TPC-C workload is a well-established benchmark designed to evaluate the performance of OLTP systems. It consists of nine tables and five kinds of transactions, simulating a warehouse-centric order processing application. In our experiments, we follow the standard transaction mix as defined by the TPC-C benchmark, with the number of warehouses set to 48. We adjust the default ratio of remote accesses to vary the proportion of *intra-node* transactions. Following the settings of existing work [19, 47, 54], we assume that our evaluation is performed under load balancing across primaries; otherwise, dynamic repartitioning techniques [12, 39, 46, 62] can be used to adjust the affinity between data and primaries.

8.1.3 Multi-Primary systems. To enable fair comparison, we implemented the following multi-primary systems in C++ within a unified code framework. We process transactions using the *read-committed* isolation level across all methods.

***Eager-Release*:** This strategy is the most straightforward page ownership acquisition algorithm in shared-storage multi-primary systems. The primary requires ownership from GPLM before operations and reverts it back to GPLM once no longer needed.

***Lazy-Release*:** This strategy is employed by PolarDB-MP and Taurus MM. Instead of immediately releasing ownership when the primary is not actively operating on the pages, ownership is retained until another primary requests it. Both the *Eager-Release* and *Lazy-Release* strategies employ the FCFS scheduling algorithm.

Two-Phase Commit (2PC): 2PC is an atomic commit protocol for shared-nothing architectures. To ensure a fair comparison, we decouple the system into compute and storage layers. Each primary exclusively accesses a data partition. In distributed transactions, the coordinator forwards remote read/write operations to the relevant primary and commits the transactions using 2PC.

Leap: Leap [29] follows a design philosophy similar to *Lazy-Release*, acquiring ownership only when a miss occurs. However, unlike *Lazy-Release*, Leap acquires ownership at the tuple level rather than pages. Additionally, it maintains only exclusive ownership globally, without shared ownership.

Chimera: This is the system proposed in this paper, as described in Sections 3 and 4. In Chimera, the iteration duration is configured to be 100 ms, and each worker thread has 16 coroutines. Further details will be provided in Section 8.4.

8.2 Performance Comparison

8.2.1 Throughput of each approach. We first compare Chimera with standard approaches using SmallBank and TPC-C workloads, varying the percentage of *intra-node* transactions. We report the throughput of committed transactions in Figures 6(a)-(d). For SmallBank, as the proportion of *intra-node* transactions increases, the performance of all methods except *Eager-Release* improves, with Chimera consistently leading. At the 10% *intra-node* transaction ratio, Chimera achieves 2.16x throughput of *Lazy-Release* under a 1% hotspot range. However, for 10% and 100% hotspot ranges, the improvement is less pronounced, as only 10% *intra-node* transactions limit the reduction in ownership transfers. In contrast, for highly concentrated hotspots, the delay-fetch mechanism captures more same-page accesses, resulting in greater performance gains.

Once the *intra-node* transaction ratio exceeds 30%, Chimera outperforms all other systems across hotspot ranges, achieving 1.33-3.38x higher throughput than *Lazy-Release*. The peak gain appears at the 1% hotspot ratio and 50% *intra-node* transactions, where 2PS and delay-fetch jointly contribute. Notably, Chimera performs even better than under dispersed hotspots, as delay-fetch becomes more effective in concentrated access. Leap achieves 1.48–1.80x *Lazy-Release*'s performance at the 1% hotspot range but remains similar

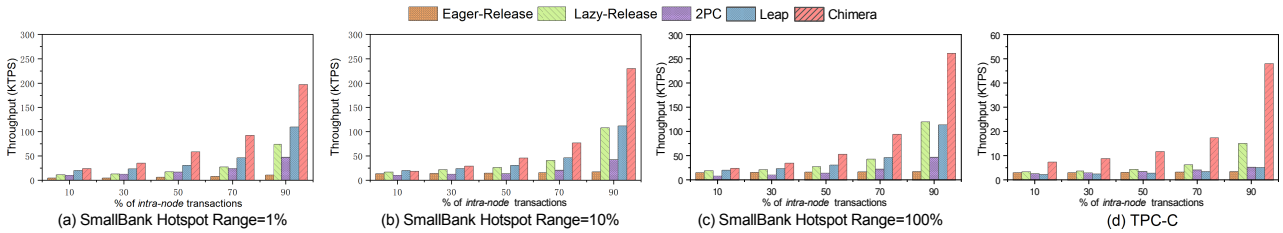

Figure 6: Performance comparison of each approach on SmallBank and TPC-C

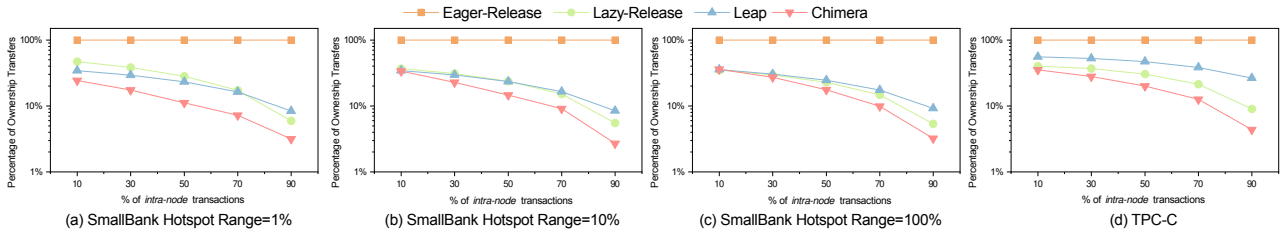

Figure 7: Percentage of ownership transfers of each approach on SmallBank and TPC-C

otherwise, benefiting from tuple-level ownership granularity that reduces contention. *Eager-Release* suffers from high network overhead and limited scalability due to immediate ownership release. While 2PC scales better, its two-phase communication incurs higher overhead than *Lazy-Release*. Overall, `Chimera` delivers performance improvements ranging from 1.86x to 19.03x over *Eager-Release* and 2PC. Under TPC-C, the New Order transactions dominate and frequently update the *District* table, leading to a high frequency of ownership transfers. Since Leap enforces exclusive ownership for all accesses, even for reads, it underperforms *Lazy-Release*. Across various *intra-node* transaction ratios, `Chimera` demonstrates significantly better performance than the baseline systems, achieving up to 3.17x the throughput of *Lazy-Release*, 8.93x that of 2PC, 9.24x that of Leap, and 13.31x that of *Eager-Release*.

8.2.2 Percentage of ownership transfers of each approach. Since 2PC does not involve ownership transfers, we compare the percentage of ownership transfers among *Eager-Release*, *Lazy-Release*, Leap, and `Chimera`, as shown in Figure 7(a)-(d). *Eager-Release* requests ownership from the GPLM and releases it immediately, resulting in an almost 100% transfer ratio. Under SmallBank, `Chimera` consistently exhibits a lower transfer ratio than *Lazy-Release* and Leap. Specifically, at 10% *intra-node* transaction ratio and 1% hotspot range, *Lazy-Release* reaches its highest transfer ratio of 47.2%, while `Chimera` remains at 24.2%. As the *intra-node* transaction ratio increases, all methods exhibit a declining trend. At 90% *intra-node* transactions, both Leap and *Lazy-Release* reduce their transfer ratios below 10%, while `Chimera` maintains a 39.8%–50.8% lower transfer ratio than *Lazy-Release* and 62.2%–68.1% lower than Leap. For TPC-C, `Chimera` achieves a 9.18%–83.6% reduction in ownership transfers compared to *Lazy-Release* and Leap.

8.2.3 Latency of each approach. We analyze the latency of each approach, as shown in Table 2, which reports the 50th and 90th percentile latencies under a workload with 50% *intra-node* transactions.

Table 2: Latency (ms) of each approach: P50 / P90

	Eager-Release	*Lazy-Release*	2PC	Leap	Chimera
SmallBank	55.6/96.1	55.4/95.4	56.6/91.7	54.3/94.4	17.2/123.2
TPC-C	84.5/130.8	76.4/118.9	82.8/123.72	90.6/145.9	30.8/152.7

All methods use the group commit mechanism. For the SmallBank workload, *Eager-Release*, *Lazy-Release*, 2PC, and Leap exhibit uniform transaction processing rates, resulting in a 50th percentile latency of approximately 50 ms and a 90th percentile latency of around 90 ms. In contrast, `Chimera` benefits from the conflict-free process of partitioned phases, enabling it to achieve lower 50th latency. However, due to the possibility of deferred execution, its 90th-percentile latency can exceed that of other methods. Under TPC-C, where contention and ownership transfers are more frequent, overall latency increases. While `Chimera` 's 90th percentile latency is slightly higher than other methods, typically within tens of milliseconds, this remains acceptable in cloud environments.

8.2.4 Throughput of each approach with long-running transactions. We evaluate the impact of mixing a certain proportion of long-running transactions into short-running transactions on system performance. The length of a long transaction is set to be 10 times that of a short transaction. We vary the proportion of long transactions and report performance in Figure 8(a). When the proportion of long transactions is 0%, `Chimera` achieves 1.53-3.97x the performance of other systems. As the proportion increases, `Chimera` 's performance gradually decreases. However, even at 25%, it still maintains 1.01-3.56x the throughput of other systems. Since long transactions are typically rare in real-world scenarios, especially in OLTP systems, `Chimera` retains an advantage over other systems.

8.2.5 Throughput of each approach with random hotspot distribution. We evaluate throughput under different hotspot distributions in SmallBank, where hotspot accounts are randomly spread across

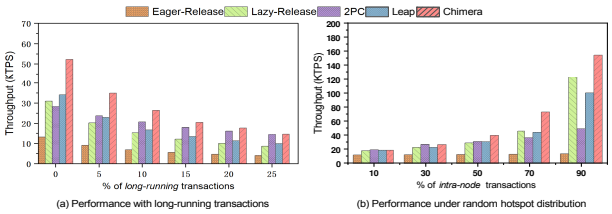

Figure 8: Performance comparison with long-running transactions and under random hotspot distribution

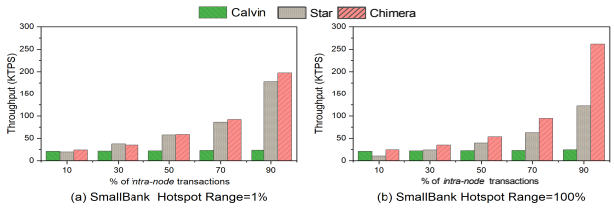

Figure 9: Compare with batch/epoch-based database

pages. In our implementation, each page stores 56 tuples, so with a 1% hotspot range, about half of the pages are hot, but most of them contain only 1~2 hotspot accounts. The results are presented in Figure 8(b). When the *intra-node* transaction ratio is low, `Chimera` shows little advantage. As the ratio increases, it achieves only up to 1.26x *Lazy-Release* 's throughput, less than its gains under the default hotspot distribution. This is because the dispersed hotspot pages hinder the delay-fetch mechanism from capturing multiple accesses to the same page. Moreover, increased contention on individual data items within the same page raises the likelihood of transaction aborts, further limiting `Chimera` 's benefit.

8.3 Batch/Epoch-Based Database Comparison

We implemented Calvin[48] and Star[33] in our prototype system for comparative evaluation. Unlike `Chimera` and other standard methods, Calvin requires collecting a batch of transactions and determining their execution order before processing. In our implementation, each node employs a sequencer thread for transaction ordering and a scheduler thread for execution coordination, while the remaining CPU cores serve as worker threads. Star follows an asymmetric replication shared-nothing architecture, where a "super node" maintains a full replica. Like `Chimera`, Star employs an epoch-based two-phase execution model, but only a single primary exclusively processes all distributed transactions in the second phase. To ensure a fair comparison, we decouple Star into storage and compute layers, allocating equal memory resources to each primary node and maintaining the full replica at the storage layer.

We report the results on SmallBank workloads under hotspot ranges of 10% and 100%, with varying percentages of *intra-node* transactions, in Figure 9(a) and 9(b). Calvin's performance remains unaffected as the *intra-node* ratio increases due to its fixed transactional messaging pattern. It requires only a single round of read-write set synchronization when sub-transactions across nodes have dependencies. However, its throughput is constrained compared to `Chimera`, as it relies on a single scheduler, whereas `Chimera` allows worker threads to compete for locks in parallel, achieving 1.15–10.75x Calvin's throughput. When the hotspot range is 100%, Star achieves only 0.42–0.73x of `Chimera`'s throughput due to the limited computational power, memory, and network I/O capacity of a single primary node. In contrast, `Chimera` leverages a logical shared memory pool across multiple primaries for higher efficiency. At the 1% hotspot range, Star improves performance by avoiding ownership transfers and caching most accessed pages locally. Nonetheless, `Chimera` maintains a slight performance advantage due to the effectiveness of delay-fetch mechanisms.

8.4 Analysis of `Chimera`

8.4.1 Sensitivity Analysis. We analyze the impact of iteration duration and the number of coroutines on throughput. As shown in Figure 10, throughput increases rapidly as the iteration duration grows from 10 ms, and then the growth rate slows significantly around 100 ms. Based on this observation, we set 100 ms as the iteration duration for high performance and low latency. As shown in Figure 11, throughput increases rapidly as the number of coroutines rises from 1, eventually leveling off after reaching 20, with a peak improvement of 2.28x compared to a single coroutine.

8.4.2 Ablation Study. This experiment examines the effects of the 2PS and delay-fetch mechanisms on system performance. We implemented three variants: `Chimera(PS)`, `Chimera(DF)`, and `Chimera`. `Chimera(PS)` extends *Lazy-Release* with the 2PS mechanism, while `Chimera(DF)` incorporates the delay-fetch mechanism. The full `Chimera` algorithm combines both mechanisms. As shown in Figure 12, `Chimera(DF)` greatly improves throughput at lower *intra-node* ratios, achieving 1.47–1.64x over *Lazy-Release*. This improvement stems from most transactions being *inter-node*, which causes high contention for page access. `Chimera(DF)` reduces ownership transfers by effectively grouping transactions that access the same pages. As the *intra-node* transaction ratio increases, `Chimera(DF)`'s benefits decline, while `Chimera(PS)` shows greater performance improvements, achieving 1.97–2.68x the throughput of *Lazy-Release* due to the increased elimination of ownership transfers for *intra-node* transactions. Combining both mechanisms, `Chimera` reaches 2.16–3.17x the throughput of *Lazy-Release* strategy, 1.35-1.98x that of `Chimera(PS)`, and 1.31-2.68x that of `Chimera(DF)`.

8.4.3 Performance impact of misclassifying transactions. Figure 13 shows `Chimera`'s throughput under known and partially unknown read/write sets varying *intra-node* ratios. The results show that performance degradation remains within 10% across all *intra-node* transaction ratios. This is because the throughput of partitioned phases significantly exceeds that of global phases. At low *intra-node* transaction ratios, the cost of speculative execution and rollback misclassified *inter-node* transactions is negligible compared to the execution overhead in the global phase. At high *intra-node* ratios, fewer transactions are misclassified. Thus, transactions with unknown read/write sets have minimal impact on overall performance.

8.5 Scalability Experiment

In this experiment, we study the scalability of `Chimera`. We report the result in Figure 14. We set the *intra-node* transaction ratio to 50%. As the number of primary nodes increases, `Chimera` at 8 nodes

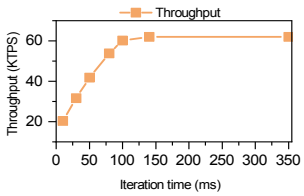

Figure 10: Overhead of phase transitions

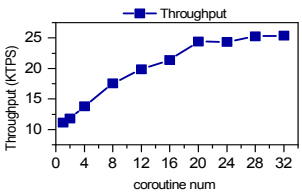

Figure 11: Throughput varying coroutine number

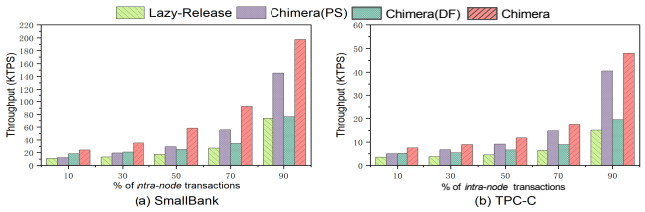

Figure 12: Performance of ablation experiments

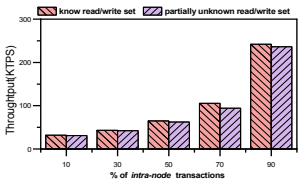

Figure 13: Impact of unknown read/write sets

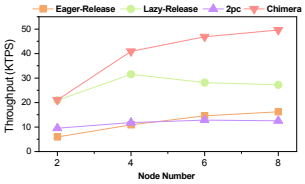

Figure 14: Scalability experiment on SmallBank

has 2.34x throughput greater than Chimera at 2 nodes. For *Lazy-Release*, performance peaks at four nodes but then declines as the number of nodes grows, primarily due to the increased frequency of page ownership transfers, which degrades performance. In contrast, *Eager-Release* begins with relatively low performance and scales nearly linearly. However, as the node number increases, a single GPLM may become a bottleneck due to the need to request locks from the GPLM for every operation. For 2PC, performance plateaus between 6 and 8 nodes, likely due to the storage service bottleneck caused by the high overhead of remote log writes.

9 RELATED WORK

This section reviews related work on transaction processing of multi-primary database systems.

Shared-Nothing database systems. Shared-nothing systems provide linear scalability but suffer performance drops in distributed transactions due to 2PC and replica synchronization [10, 20, 23, 47, 55]. Consequently, some works [24, 34, 58, 60, 61] combine 2PC with replica synchronization to minimize network round-trips and reduce the overhead of distributed transactions. Additionally, some approaches, such as Schism [12] and Sword [39], reduce distributed transactions by analyzing historical workloads to optimize data partitioning and migration. These methods are orthogonal to Chimera and can be integrated to enhance page-primary affinity, increasing the ratio of *intra-node* transactions. Star [33] employs the asymmetric replication, relying on a single node with the full replica for processing all cross-partition transactions. While it avoids 2PC, its performance is constrained by the limited memory, network I/O, and lack of parallel processing of a single instance in the cloud. In contrast, Chimera maintains parallelism, offers higher fault tolerance, and better adapts to workloads with temporal locality [8, 29].

Shared-Storage database systems. Unlike shared-nothing multi-primary database systems, another class of multi-primary databases adopts a shared-storage architecture that allows each primary to access all data. Traditional systems [9, 21], are based on non-cloud environments and rely on specialized hardware, leading to a high total cost of ownership (TCO). In contrast, recent work[14, 27, 50, 54] leverages the elasticity of cloud resources to reduce database construction costs. These systems employ either pessimistic or optimistic strategies to detect and resolve concurrent update conflicts at the page level, with most relying on RDMA two-sided verbs to expedite ownership acquisition, page synchronization, or log replay. However, RDMA technology [22, 51, 56, 57, 59, 60, 64] is costly and generally limited to a single data center, which poses a significant challenge to high availability and cloud service providers' network conditions. Tell [31] and LEAP [29] maintain data ownership at the tuple level, but still suffer from performance degradation due to frequent migrations, a challenge that Chimera effectively mitigates.

Full replica database systems. Another category of multi-primary systems deploys multiple full replicas across nodes. Some studies [30, 32, 36, 48] adopt deterministic execution, ensuring that each replica processes transactions in the same pre-determined order to achieve replica consistency. Other works [11, 52, 63] leverage *Conflict-Free Replicated Data Types* (CRDTs) [41] to merge conflicts without coordination, supporting low-latency reads and writes in geo-distributed environments. However, these methods often relax consistency levels, and the CRDT merge rules are hard-coded, limiting their applicability to specific scenarios where strong transactional consistency is required.

10 CONCLUSION

In this paper, we identify frequent page ownership transfers as a key bottleneck limiting the performance of multi-primary databases. To address this challenge, we propose Chimera, a multi-primary, shared-storage, cloud-native database designed to optimize frequent ownership transfers and enhance system throughput. Chimera adopts a two-phase transaction scheduling mechanism that alternates between partitioned and global phases. In the partitioned phase, *intra-node* transactions are aggregated to reduce ownership transfers. In the global phase, Chimera employs a delay-fetch ownership transfer mechanism to maximize operations on a page within a single ownership cycle. These strategies together alleviate transfer overhead in multi-primary systems. Evaluation on SmallBank and TPC-C benchmarks shows that Chimera outperforms other approaches in most scenarios.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62441230, 62472429, 62461146205. We thank the reviewers for their constructive suggestions and Zhanhao Zhao, Hongyao Zhao, and Huicong Xu for their discussions. We also thank Public Computing Cloud (Renmin University of China).

REFERENCES

[1] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1756. https://doi.org/10.1145/3299869.3314047

[2] Russell R. Atkinson and Edward M. McCreight. 1987. The dragon processor. *SIGPLAN Not.* 22, 10 (oct 1987), 65–69. https://doi.org/10.1145/36205.36185

[3] Smallbank benchmark. 2021. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[4] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221. https://doi.org/10.1145/356842.356846

[5] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483. https://doi.org/10.1145/319996.319998

[6] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., USA.

[7] Apache bRPC. 2024. https://brpc.apache.org/.

[8] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD bufferpool extensions for database systems. *Proceedings of the VLDB Endowment* (Sep 2010), 1435–1446. https://doi.org/10.14778/1920841.1921017

[9] Sashikanth Chandrasekaran and Roger Bamford. 2003. Shared cache-the future of parallel databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE Computer Society, 840–840. https://doi.org/10.1109/ICDE.2003.1260883

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (aug 2013), 22 pages. https://doi.org/10.1145/2491245

[11] Redis CRDT. 2022. https://redis.io/blog/diving-into-crdts/.

[12] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[13] Shaul Dar, Michael J. Franklin, Björn Pór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 330–341.

[14] Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. 2023. Taurus MM: Bringing Multi-Master to the Cloud. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3488–3500. https://doi.org/10.14778/3611540.3611542

[15] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. *SIGMOD Rec.* 14, 2 (June 1984), 1–8. https://doi.org/10.1145/971697.602261

[16] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (nov 1976), 624–633. https://doi.org/10.1145/360363.360369

[17] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[18] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High performance transactions via early write visibility. *Proc. VLDB Endow.* 10, 5 (jan 2017), 613–624. https://doi.org/10.14778/3055540.3055553

[19] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 553–564. https://doi.org/10.14778/3055540.3055548

[20] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[21] J. W. Josten, C. Mohan, I. Narang, and J. Z. Teng. 1997. DB2's use of the coupling facility for data sharing. *IBM Systems Journal* 36, 2 (Jan 1997), 327–351. https://doi.org/10.1147/sj.362.0327

[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[23] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[24] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[25] Leslie Lamport. 2019. *Time, clocks, and the ordering of events in a distributed system.* Association for Computing Machinery, New York, NY, USA, 179–196. https://doi.org/10.1145/3335772.3335934

[26] Feifei Li. 2019. Cloud-native database systems at Alibaba: opportunities and challenges. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2263–2272. https://doi.org/10.14778/3352063.3352141

[27] 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. *Proc. VLDB Endow.* 17, 12 (2024). https://doi.org/10.14778/3685800.3685806

[28] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: an autonomous database system. *Proc. VLDB Endow.* 14, 12 (July 2021), 3028–3042. https://doi.org/10.14778/3476311.3476380

[29] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1659–1674. https://doi.org/10.1145/2882903.2882923

[30] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don't Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1156–1168. https://doi.org/10.1145/3448016.3452827

[31] 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* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 663–676. https://doi.org/10.1145/2723372.2751519

[32] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proc. VLDB Endow.* 13, 12 (July 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808

[33] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: scaling transactions through asymmetric replication. *Proc. VLDB Endow.* 12, 11 (July 2019), 1316–1329. https://doi.org/10.14778/3342263.3342270

[34] Sujaya Maiyya, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2019. Unifying consensus and atomic commitment for effective cloud data management. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 611–623. https://doi.org/10.14778/3303753.3303765

[35] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts.* Vol. 192. Addison-Wesley New York. https://doi.org/10.5555/359414

[36] Cuong D. T. Nguyen, Johann K. Miller, and Daniel J. Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proc. ACM Manag. Data* 1, 2, Article 148 (June 2023), 27 pages. https://doi.org/10.1145/3589293

[37] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[38] Mark S. Papamarcos and Janak H. Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News* 12, 3 (jan 1984), 348–354. https://doi.org/10.1145/773453.808204

[39] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 430–441. https://doi.org/10.1145/2452376.2452427

[40] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.* 10, 4 (nov 2016), 445–456. https://doi.org/10.14778/3025111.3025125

[41] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer

Berlin Heidelberg, Berlin, Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[42] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 510–521.

[43] Hanan Shukur, Subhi Zeebaree, Rizgar Zebari, Omar Ahmed, Lailan Haji, and Dildar Abdulqader. 2020. Cache coherence protocols in distributed systems. *Journal of Applied Science and Technology Trends* 1, 2 (2020), 92–97. https://doi.org/10.38094/jastt1329

[44] 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* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1150–1160. http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf

[45] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27. http://sites.computer.org/debull/A13june/VoltDB1.pdf

[46] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 245–256. https://doi.org/10.14778/2735508.2735514

[47] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[48] 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* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[49] TPC-C. 2024. http://www.tpc.org/tpcc/.

[50] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052. https://doi.org/10.1145/3035918.3056101

[51] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[52] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2021. Anna: A KVS for Any Scale. *IEEE Trans. on Knowl. and Data Eng.* 33, 2 (Feb. 2021), 344–358. https://doi.org/10.1109/TKDE.2019.2898401

[53] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.* 10, 7 (mar 2017), 781–792. https://doi.org/10.14778/3067421.3067427

[54] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/3626246.3653377

[55] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397. https://doi.org/10.14778/3554821.3554830

[56] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (feb 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[57] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (feb 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[58] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4, Article 12 (Dec. 2018), 37 pages. https://doi.org/10.1145/3269981

[59] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 51–68. https://www.usenix.org/conference/fast22/presentation/zhang-ming

[60] Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. 2023. Efficient Distributed Transaction Processing in Heterogeneous Networks. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1372–1385. https://doi.org/10.14778/3583140.3583153

[61] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. 2022. Starry: multi-master transaction processing on semi-leader architecture. *Proc. VLDB Endow.* 16, 1 (Sept. 2022), 77–89. https://doi.org/10.14778/3561261.3561268

[62] Qiushi Zheng, Zhanhao Zhao, Wei Lu, Chang Yao, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. Lion: Minimizing Distributed Transactions Through Adaptive Replica Provision. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2012–2025. https://doi.org/doi:10.1109/ICDE60146.2024.00161

[63] Weixing Zhou, Qi Peng, Zijie Zhang, Yanfeng Zhang, Yang Ren, Sihao Li, Guo Fu, Yulong Cui, Qiang Li, Caiyi Wu, Shangjun Han, Shengyi Wang, Guoliang Li, and Ge Yu. 2023. GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. *Proc. ACM Manag. Data* 1, 1, Article 62 (may 2023), 27 pages. https://doi.org/10.1145/3588916

[64] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. *Proc. ACM Manag. Data* 1, 2, Article 131 (June 2023), 26 pages. https://doi.org/10.1145/3589276