Tigon: A Distributed Database for a CXL Pod

Yibo Huang, Haowei Chen, Newton Ni, Yan Sun†, Vijay Chidambaram, Dixin Tang, Emmett Witchel
The University of Texas at Austin †*University of Illinois Urbana-Champaign*

Abstract

Building efficient distributed transactional databases remains a challenging problem despite decades of research. Existing distributed databases synchronize cross-host concurrent data accesses over a network, which requires numerous message exchanges and introduces performance overhead.

We describe Tigon, the first distributed in-memory database that synchronizes concurrent cross-host data accesses using atomic operations on CXL memory. Using CXL memory is more efficient than network-based approaches, but Tigon's design must address CXL's higher latency and lower bandwidth relative to local DRAM, as well as CXL's limited hardware support for cross-host cache coherence. For TPC-C and a variant of YCSB, Tigon achieves up to 2.5× higher throughput compared with two optimized shared-nothing databases that use CXL memory as a transport and up to 18.5× higher throughput compared with an RDMA-based distributed database.

1 Introduction

Despite decades of research, efficiently scaling a transactional database to multiple hosts remains a challenging task due to the complexities of data synchronization and concurrency control across hosts [17, 33, 48, 49, 56, 61, 66, 77, 85]. Existing distributed databases synchronize cross-host concurrent data accesses over a network, which introduces high and variable overhead due to cross-host message exchanges and high network latencies. For example, traditional shared-nothing databases partition data across hosts and let each host process all read/write operations to its partition [17, 33, 56, 66]. The performance of this architecture drops significantly when the database needs to process a large number of *multi-partition transactions*, that is, transactions that need to read or write data in multiple partitions and will be executed by multiple hosts. Multi-partition transactions are costly due to numerous message exchanges between hosts and the two-phase commit (2PC) protocol necessary to complete them [23, 48, 49, 61, 77, 85].

A recent line of research leverages RDMA to accelerate distributed transaction processing [4, 13, 20, 21, 32, 62, 71–73]. Many papers adopt an RDMA-based shared and disaggregated memory architecture to avoid multi-partition transactions [19, 44, 75, 78–80, 87]. However, the latency for memory access through RDMA networks is one to two orders of magnitude higher than local DRAM [26] and synchronizing concurrent read/write operations to database tuples through RDMA networks is still expensive, because round trips on RDMA-based networks have latencies in the microseconds range.

This paper proposes a new direction for building distributed transactional databases: *instead of synchronizing cross-host concurrent data accesses over a network, we propose synchronizing these accesses directly through memory, leveraging the capabilities of emerging CXL technology*. CXL memory is a memory module that can be physically connected and shared by a small number of hosts (e.g., 8–16 [6, 45]). The CXL protocol allows CPUs to directly access CXL memory using normal load and store instructions, and it provides much lower latency compared with RDMA-based shared memory. With CXL version 3.0 (and in 3.2 [2], which is the current version as of this writing) the specification allows CXL memory to be shared across hosts with hardware cache coherence, which provides new opportunities for cross-host data sharing and data synchronization. We call a collection of machines that share CXL memory a *CXL pod* [11, 29, 74, 86].

While CXL memory has advantages over RDMA and other networking technologies, it must be used with care. CXL memory has two major limitations: 1) it has higher latency (214-394 ns vs. 111-117 ns) and lower bandwidth (18-52 GB/s vs. 218-246 GB/s for a read-only workload) compared with local DRAM, as measured on hardware prototypes in a recent study [47]. and 2) only a small part of the physical CXL memory space, ranging from dozens to hundreds of MBs, will be kept hardware cache-coherent as reported in recent work from AMD [30]. Higher latency and lower bandwidth means a database cannot simply place all its data in CXL memory if it wants to achieve peak performance. Limited hardware cache-coherent memory means that database synchronization structures must be reorganized to minimize the use of this memory.

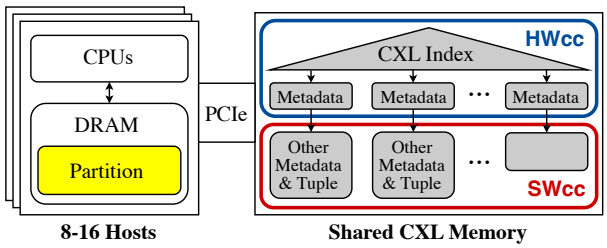


Figure 1: Tigon on a CXL pod.

In this paper, we introduce Tigon[1], the first distributed in-memory database that leverages CXL memory for efficiently synchronizing cross-host concurrent data accesses. Tigon benefits from the performance of CXL memory while avoiding CXL's weaknesses, based on a key insight about general transactional workloads: *while the size of a database could be large, the size of the set of tuples that will be concurrently read or written by running transactions from different hosts is small*. This is because each transaction typically accesses a small number of tuples (e.g., from a few to dozens) and the number of concurrently running transactions is typically equal to the number of available CPU cores for an in-memory database. For example, in TPC-C [7], a transaction accesses an average of 39 tuples, with a total data size of approximately 7 KB. Assuming 1,000 cores, each core executing one transaction, the system has at most 39K active tuples, which amount to 7 MB of data. We call such a set of tuples the *Cross-host Active Tuples* or CAT for short. We design Tigon to efficiently maintain the CAT in CXL memory, thereby increasing performance by converting multiple message exchanges into data structure operations. Because the data structure needs to contain only data that is currently shared, Tigon's performance is not limited by CXL's inferior latency and bandwidth relative to local DRAM.

Tigon initially partitions data across hosts as shown in Figure 1 and dynamically maintains the CAT in CXL memory. Cross-host concurrent accesses to the CAT can be efficiently synchronized using atomic operations, metadata (e.g., database latches), and hardware cache coherence, allowing Tigon to adopt a single-host concurrency control protocol to ensure transaction semantics without using 2PC. If one host needs to access a tuple stored in a remote host, Tigon moves the tuple to shared CXL memory. The data shared in CXL memory will be moved back to its original host based on a system policy that considers access patterns to the shared data. In addition, each host can retain the data accessed exclusively by that host in local DRAM, leveraging its low latency and high bandwidth.

Tigon addresses two broad challenges for maintaining the CAT in CXL memory. First, maintaining the CAT in the limited hardware cache-coherent region may introduce frequent data movement between local DRAM and CXL memory, increasing CXL memory bandwidth usage and the latency for processing a transaction. Tigon addresses this challenge with an efficient software cache coherence protocol that enables it to use CXL memory that is not kept coherent by hardware. Tigon uses the capacity of this large software cache-coherent region for database data, which reduces the need for data movement. Tigon also reduces data movement with common sense optimizations like only copying updated tuples from CXL memory back into local DRAM.

Indexes and certain metadata (e.g., database latches) require intensive synchronization across hosts, which is efficiently provided by atomic instructions operating on hardware cache-coherent memory. A key design principle for Tigon is to separate data by synchronization requirements: synchronization-heavy metadata (e.g., indexes and latches) is stored in the hardware cache-coherent (HWcc) region, while other data (such as tuples) is stored in the software cache-coherent (SWcc) region, as illustrated in Figure 1. Databases already have synchronization primitives to ensure the integrity of tuples under concurrent accesses, so we co-design Tigon's software cache coherence protocol with the database's mechanism for protecting the integrity of tuples to minimize synchronization overhead.

The second challenge Tigon addresses is to efficiently access data and provide transaction semantics (i.e., ACID properties) while data is moving between local DRAM and CXL memory. To do so, Tigon utilizes database latches and indexes to provide efficient access to the CAT under data movement and enhances concurrency control (i.e., two-phase locking (2PL) and next-key locking) and logging protocols to ensure transaction semantics without using 2PC. First, for a tuple moved to CXL memory, the owner host can cache a shortcut pointer to quickly find this tuple without searching the CXL index in CXL memory, reducing latency and CXL memory bandwidth usage. The challenge is to ensure its correctness under concurrent data movement. Second, we enhance 2PL to work together with a scalable logging protocol [82] such that Tigon, running on multiple hosts, does not need to use 2PC when committing transactions. Finally, we design an enhanced next-key locking protocol that maintains minimal additional metadata for each tuple to avoid the phantom problem [24]. We also design a lightweight data movement policy that keeps tuples that will most likely be exclusively accessed by a single host in local DRAM.

The novelty of Tigon lies in its use of inter-host CXL memory to accelerate transaction processing, while avoiding the performance pitfalls of using CXL memory naïvely. Tigon uses inter-host CXL memory to accelerate message passing, but it also reduces the need to pass messages by using data structures in inter-host CXL memory that are synchronized via atomic operations. The result is greater throughput for transaction processing, especially transactions that access data from multiple partitions. Experiments on TPC-C and a variant of YCSB demonstrate that Tigon achieves up to 2.5× higher throughput compared with two optimized shared-nothing databases using CXL memory as a transport and up to 18.5× higher throughput compared with an RDMA-based distributed database.

While prior research has explored improving the efficiency

[1] A Tigon is a hybrid of a male tiger and a female lion.

of databases using CXL memory [8, 10, 39, 60], none considers leveraging limited hardware cache-coherent memory for building an efficient distributed database. This work is also different from prior research on memory tiering and pooling [22, 45, 54, 59, 65], which makes CXL memory available to applications transparently, placing the burden of memory management and data migration to system software (or software and hardware [83]). Tigon, in contrast, is built to explicitly move data to and from CXL memory shared among multiple hosts and explicitly allocates CXL memory and places data structures in either the hardware or software cache-coherent regions.

This paper makes the following contributions:

- We introduce the first distributed in-memory database that synchronizes cross-host concurrent data accesses via atomic operations on CXL memory.
- We address the hardware limitations of CXL memory by maintaining the CAT in CXL memory and using a software cache coherence protocol to reduce the cost of maintaining the CAT (§3.2).
- We design new methods for efficiently accessing the CAT under data movement and enhance concurrency control and logging protocols to ensure transaction semantics without using 2PC (§3.4 and §3.5).
- We implement Tigon (https://github.com/ut-data sys/tigon) and perform extensive experiments that show its performance advantages over shared-nothing databases and an RDMA-based distributed database (§4).

2 CXL Pod Background and Challenges

Compute Express Link (CXL) is a high-performance, open-standard interconnect for communications between CPUs, devices, and memory based on PCIe 5.0 and 6.0. CXL memory is a memory module connected to one or multiple hosts and has different characteristics depending on the CXL specification [2] version and the type of device. Type-3 CXL devices are memory expanders (that use the CXL.mem protocol), with the version 1.1 CXL specification allowing a single host to access PCIe-connected memory in a cache-coherent manner. CXL 3.0 and the most recent CXL 3.2 specify cacheline-granularity memory sharing and cache coherence across multiple hosts.

A recent study [47] shows that CXL memory has significantly higher latency (214-394 ns vs. 111-117 ns) and lower bandwidth (18-52 GB/s vs. 218-246 GB/s for a read-only workload) compared with local DRAM. Exact bandwidth depends on the specific hardware prototype and the memory access pattern. Another study [64] reports 11.7 GB/s sequential read bandwidth, but falls as low as ∼5 GB/s for other access patterns.

CXL pod. A CXL pod [11, 29, 74, 86] includes a small number of machines (e.g., 16) that connect directly to a shared CXL memory module via multiple ports (called a multi-headed device [2] (MHD)). The advantage of an MHD is that it allows multiple hosts to connect to a single memory device without a switch, which adds significant latency. This scale of configuration provides rich computing resources for transactional databases and has small latency and bandwidth penalties compared with a single host [45].

A CXL pod is an intermediate organization between a shared-memory multi-processor and a distributed system. The pod has hosts with local, hardware cache-coherent memory. Each host also connects to a single, shared CXL memory module. The inter-host CXL memory supports limited (and expensive) hardware cache coherence. There is a long debate in computer architecture as to the limits of scalability of SMP machines [36, 53], but the pod provides a new set of tradeoffs. We built Tigon to navigate this novel and complex tradeoff space.

CXL memory devices that can be used in a CXL pod exist, notably SK Hynix's Niagara 2.0 [6] and a Microsoft prototype [11]. The former is limited to 8 hosts, and the latter to 2. Neither supports hardware cache coherence, but their latency and bandwidth are broadly similar to our evaluation testbed (§4). Tigon is designed for CXL devices with some hardware cache-coherent memory. Designing a database to be efficient without hardware cache-coherent memory is worthwhile future work.

Comparison to NUMA and RDMA-based architectures. NUMA and CXL memory share some architectural properties, but detailed studies show that NUMA behavior often differs significantly from CXL [31, 64]. In our experience building Tigon, we found that a NUMA platform served as an informative proxy for CXL, but much of our system tuning was specialized to CXL (e.g., CXL bandwidth is much lower than NUMA bandwidth on our testbeds). RDMA-based distributed databases are more scalable in terms of the number of nodes in the system because they use explicit addressing and switches, but they have a higher latency, in the microseconds range.

Hardware limitations of CXL memory. Three properties of CXL memory make it challenging to use for a database: (1) it has higher latency than local DRAM; (2) it has lower bandwidth than local DRAM; (3) it has limited hardware cache-coherent memory capacity. The high latency and low bandwidth means any system that uses CXL memory must lean heavily on the performance of local DRAM. Data can be copied into and out of CXL memory, but too much data movement will overwhelm its limited bandwidth.

Why inter-host hardware cache coherence is limited. The CXL standard, starting from 3.0, provides support for inter-host hardware cache coherence through back-invalidations, that is, the CXL device includes a snoop filter (similar to processor snoop filters) for managing coherence. However, providing cache coherence for the entire CXL memory address space will be too expensive to be practical due to the size of the snoop filter required and the metadata overhead, as shown by AMD [30]. There is no theoretical reason for the limitation, it is purely practical. There is a fixed area budget for the CXL

controller that is consumed by ports and control logic [11]. Snoop filters for the entire CXL memory address space would be large because they must track all cacheable data for every host. As an extreme example, Intel's Granite Rapids 6980P processor would require tags for 16×504 MB, or 7.9 GB of cacheable data. Snoop filters that are cost effective will cover a much smaller region. In order to control costs and complexity, vendors will limit the size of the hardware cache-coherent region of CXL memory devices. Because we do not have exact figures, our study varies the amount of available hardware cache-coherent memory to evaluate its impact (§4).

Failure model. We assume a fail-stop model, and achieve durability using logging. Logs are written to local SSD devices. A failure of any component causes a failure of (and subsequent recovery for) the entire system.

3 Tigon Design and Implementation

Tigon is a distributed in-memory database designed to run transactions efficiently on a CXL pod. Unlike traditional distributed databases, Tigon exploits unique features of the CXL pod to lower the overhead of cross-host coordination and significantly increase transaction throughput.

3.1 Tigon Overview

Architecture. Tigon adopts the Pasha architecture [29], where data is initially partitioned across the hosts of the CXL pod and dynamically moved between local DRAM and CXL memory. Each host is the *owner* of a disjoint partition of the data. It stores its partition in local DRAM and accesses this partition with low latency and high bandwidth. When data is accessed by its owner host, no cross-host coordination is required. When a host wants to access data it does not own, it requests the owner to move the data to shared CXL memory such that Tigon can maintain the *cross-host active tuples* (CAT), the set of tuples that are concurrently read or written by active transactions running on different hosts, in CXL memory. Shared data requires synchronization across hosts; hosts synchronize using latches and locks in the hardware cache-coherent part of CXL memory. Hosts communicate with each other by sending messages using CXL memory as a transport, similar to HydraRPC [50].

Software cache coherence (§3.2)**.** Tigon introduces a novel software cache coherence protocol for CXL memory because most of the physical CXL memory space will lack hardware cache coherence support. The chief insight is that hardware support for cache coherence is *only required* for key metadata, such as latches and locks, that are frequently accessed by multiple hosts; by carefully limiting the size of such metadata and strategically placing it in the hardware cache-coherent region, Tigon can provide software cache coherence using the database's existing concurrency control techniques. Software cache coherence allows Tigon to use much more CXL memory than the limited hardware cache-coherent region.

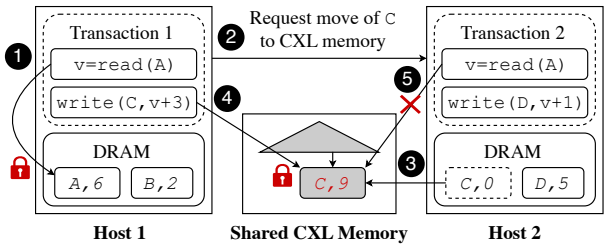


Figure 2: An example transaction workflow in Tigon.

Handling concurrency (§3.4)**.** Tigon needs to manage concurrency: both among threads within a host, and between hosts. Tigon utilizes database latches and indexes to efficiently access the CAT even when data is dynamically moving between local DRAM and CXL memory. Tigon adapts concurrency control protocols, two-phase locking (2PL) and next-key locking, to ensure serializability while maintaining the CAT.

Avoiding two-phase commit (2PC) (§3.3 and §3.5)**.** 2PC adds considerable overhead to transactions as it requires two rounds of message exchanges. Avoiding 2PC requires a single host to execute and log all database modifications involved in a transaction, enabling a host to commit its transaction locally. Tigon avoids 2PC based on two insights. First, by maintaining the CAT in CXL memory, a single host can complete all modifications to database tuples required by a transaction (e.g., by taking locks on CXL-resident structures). Second, although transaction execution in Tigon may involve modifying indexes of other hosts, indexes can be *reconstructed* from the database tuples during recovery, avoiding the need to log index modifications. As a result, only tuple changes need to be logged at the host executing the transaction to ensure atomicity and durability. Combining these two insights, a single host executes all transaction operations (excluding data movement and modifications to indexes of other hosts) and logs tuple modifications. We adapt a logging protocol [82], which was designed for scalability, to avoid 2PC.

Example transaction workflow. Figure 2 shows an example transaction workflow in Tigon. The database stores key-value pairs (e.g., tuple (A, 6) has key A and value 6). The example transaction (transaction 1) reads tuple *A* and writes tuple *C*, and is executed by a worker thread on Host 1 (its transaction worker).

1. The transaction worker acquires the read lock for tuple (A, 6) using 2PL. It then reads tuple (A, 6).
2. To write to tuple (C, 0), which is stored in Host 2, the transaction worker sends a message to Host 2 to request a move of (C, 0) to CXL memory.
3. A worker thread on Host 2 moves (C, 0) to CXL memory.
4. The transaction worker acquires the write lock on tuple (C, 0) and updates it to (C, 9).

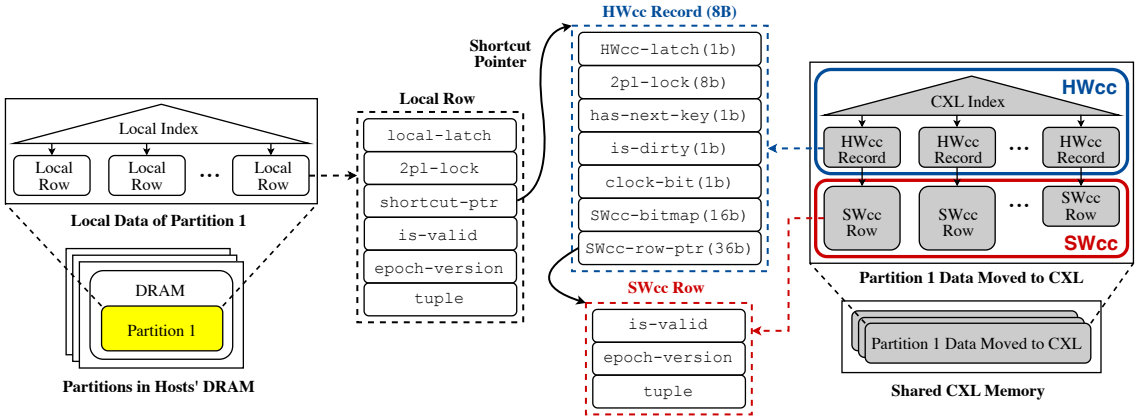


Figure 3: Data organization in Tigon.

5. The transaction worker of transaction 2 needs to access the tuple for key C at the same time; it finds (C, 9) in CXL memory is locked and aborts transaction 2 to prevent deadlocks (using the NO_WAIT policy).

After the transaction worker finishes transaction 1, it commits the transaction without any further messages to Host 2 or using 2PC, since it executes all operations of transaction 1 itself and has all the log records of the changes transaction 1 has made to the database. In this example, the CAT is the single tuple with key C.

3.2 Data Organization & SW Cache Coherence

We now describe how Tigon organizes data and implements software cache coherence. Figure 3 shows an overview of the data organization in Tigon.

Partitions and local row. Tigon organizes data into tables and partitions them across hosts. Tigon stores the partitioning function in a shard map and replicates it across hosts (not shown in the figure). Each host uses the shard map to find the host that owns a certain tuple, as do many partition-based distributed databases [1, 18, 33, 77]. Partitions are initially stored in each host's local DRAM. Data is stored in the form of *local row*. Each local row contains the following:

- Latch for mutual exclusion (`local-latch`)
- Locking metadata for 2PL (`2pl-lock`)
- Pointer to the tuple in CXL memory (`shortcut-ptr`)
- Flag indicating whether the tuple is valid (`is-valid`)
- Field used for logging (`epoch-version`)
- Database tuple (`tuple`)

Data and metadata in CXL memory. Tigon maintains the CAT in CXL memory. To minimize hardware cache-coherent (HWcc) memory usage while still enabling efficient cross-host synchronization on tuples moved to CXL memory, Tigon stores the metadata needed for intensive cross-host synchronization in the HWcc region and everything else (i.e., databse tuples and other metadata) in the CXL memory region that is not hardware cache-coherent (non-HWcc). If a local row is moved to CXL memory, Tigon divides it into an *HWcc record* stored in the HWcc region and an *SWcc row* stored in the non-HWcc region. We develop a software cache coherence protocol to enable cacheable reads of the SWcc rows.

HWcc record and SWcc row. Each SWcc row contains the `is-valid`, `epoch-version`, and `tuple` fields, similar to a local row. Each HWcc record occupies 8 bytes and contains the following fields:

- 1-bit exclusive latch for mutual exclusion (`HWcc-latch`)
- 8-bit locking metadata for 2PL (`2pl-lock`)
- 1-bit flag supporting next-key locking (`has-next-key`)
- 1-bit flag indicating whether the tuple has been modified since being moved to CXL memory (`is-dirty`)
- 1-bit supporting the CLOCK policy (`clock-bit`)
- 16-bit bitmap for software cache coherence (`SWcc-bitmap`)
- 36-bit pointer to SWcc row (`SWcc-row-ptr`)

The `SWcc-row-ptr` stores an offset relative to a base memory pointer. It can represent up to 64 GB of memory space, and is large enough for storing the data that needs to be frequently accessed across hosts in a transactional database. If the offset is defined at the granularity of cachelines, the addressable tuple storage expands to 2 TB (i.e., $2^{36} \times 2^{6}$ bytes).

The `is-dirty` bit allows the owner host to read clean tuples from local DRAM instead of CXL memory, significantly reducing CXL memory bandwidth usage.

Indexes and shortcut pointer. Tigon maintains two types of indexes: local index and CXL index. Each host maintains a local index in DRAM, which maps a primary key to a local row, for its own partition of each table. Local indexes are used by each host to find the tuples that it owns. Tigon maintains a primary index in the HWcc region (the *CXL index*) for tuples from the same table and partition that are resident in CXL memory. If a tuple is moved to CXL memory, the local row includes a pointer, `shortcut-ptr`, that points to the corresponding HWcc record, which in turn has the `SWcc-row-ptr` that points to the SWcc row. Otherwise, the `shortcut-`

ptr is NULL. Tigon pays the price of storing the shortcut-ptr in a local row to efficiently access a tuple moved to CXL memory without searching the CXL index. As tuples are moved between local DRAM and CXL memory, the owner host updates the CXL index while each non-owner host uses the CXL index to find tuples that it does not own.

Example. We use the example in Figure 2 to explain the data organization. Initially, the table is partitioned across the two hosts. Tuple (A, 6) belongs to the partition owned by Host 1 and is stored in the local DRAM of Host 1. So the shortcut-ptr of the local row is NULL. When transaction 1 from Host 1 needs to access tuple (C, 0), Host 2 moves this tuple to CXL memory by creating an HWcc record and an SWcc row, setting the SWcc-row-ptr of the HWcc record to point to the SWcc row, and inserting the HWcc record to the CXL index. It will also set the shortcut-ptr of the local row to point to the HWcc record. Host 1 can now get tuple (C, 9) using the CXL index.

Data movement policy. Tigon moves tuples to CXL memory upon request. If a host's HWcc memory usage exceeds a predefined threshold, it will move a tuple it owns from CXL memory back to local DRAM. Ideally, a host prioritizes moving tuples least likely to be accessed by non-owner hosts in the future back to local DRAM. This makes the Least Recently Used (LRU) policy a natural fit. Under LRU, the tuple least recently accessed by a non-owner host is moved back to local DRAM. However, maintaining the metadata required for LRU is expensive; LRU requires maintaining a linked list across tuples in CXL memory and updating this list for every access from a non-owner host.

To reduce overhead and minimize the HWcc memory usage, Tigon adopts the CLOCK policy [16]. Each tuple in CXL memory is associated with a bit (the clock-bit, as shown in Figure 3) stored in the HWcc record. The clock-bit is set whenever a non-owner host accesses the tuple. To move a tuple back, the owner host maintains a circular cursor to scan the tuples in CXL memory. If the clock-bit for a tuple is set, the host unsets it and moves the cursor forward. Otherwise, the host moves the tuple back to local DRAM.

Software cache coherence. We co-design a novel software cache coherence protocol that works with the database's internal synchronization. Tying coherence to synchronization has the advantage that the coherence granularity is large (i.e., a tuple, which is larger than a cacheline) and the maintenance of coherence metadata is less expensive because it is combined with synchronization tasks that are already part of the database's operation. The disadvantage is that the mechanism requires source code changes and is specific to the database, so it cannot be directly reused by other applications.

A database thread needs to take a latch before accessing a tuple to ensure only one thread can read or modify this tuple. Therefore, Tigon reuses this latch, HWcc-latch, to support software cache coherence by embedding the metadata, SWcc-bitmap, within the HWcc record.

The SWcc-bitmap uses a bit for each host (Tigon supports up to 16 hosts) to track the hosts that can do cacheable loads to SWcc rows, because cacheable loads have much lower latency than non-temporal loads. When a host reads an SWcc row, it looks at its bit in the SWcc-bitmap to determine how to access the data in the row. If the host's bit is set, it uses cacheable loads. Otherwise, its cached copies might be out of date. So it flushes the relevant cachelines, and reads the SWcc row from CXL memory into the CPU cache, and sets its bit in the SWcc-bitmap. So long as no other host updates the SWcc row (i.e., so long as the bit remains set), the host can use cacheable loads to read the SWcc row. A host that updates an SWcc row will unset the bits for all other hosts.

3.3 Transaction Execution

Transactions in Tigon use epoch-based group commit similar to SiloR [82]. Tigon breaks transactions into epochs and ensures that transactions in smaller epoch numbers are serially ordered and committed before those in larger ones.

Workers. Each host in Tigon includes multiple threads for executing transactions. A transaction is initially assigned to a thread, called a *transaction worker*, for executing and committing this transaction. The transaction worker may ask another host to do work (e.g., move data into CXL memory) to complete the transaction it is currently executing. We call the host the transaction worker runs on the *local host* and the partition owned by the local host the *local partition*. Similarly, we call the other hosts and the other partitions *remote hosts* and *remote partitions*, respectively, with respect to the transaction worker.

Read, write, and range query. Algorithm 1 shows the process of reading or writing a tuple in Tigon. If the tuple is in the local partition, the transaction worker gets the local row through the local index and further gets the HWcc record if the tuple resides in CXL memory (Lines 1-11 in Algorithm 1). To read or write a tuple (i.e., the ReadOrWrite function in Algorithm 1), the transaction worker checks the 2pl-lock to take a read or write lock or abort the transaction based on the 2PL protocol. For example, if the write lock bit of the 2pl-lock is already set, the transaction will be aborted. Otherwise, the transaction worker proceeds to perform read or write.

To read or write a tuple owned by a remote host (Lines 13-22 in Algorithm 1), the transaction worker will ask a remote host to move the tuple to CXL memory if it cannot find it in the CXL index. Note that we use a while loop to retry the data move in request because it is possible that the remote host moves the tuple back to local DRAM before the worker accesses this tuple. In practice, the worker will almost never retry due to the data movement policy of Tigon.

Processing range queries is similar to processing read operations. If a range query is on the local partition, the transaction worker searches the local index to get the local rows within the range and optionally follows the shortcut-ptr to access the tuples moved to CXL memory. Otherwise, the transaction

Algorithm 1: Read or write a tuple in Tigon.

```
Input: k – key of the tuple to be read or written
if k belongs to local partition then
    localRow ← Get(k, local index);
    Acquire local-latch for localRow;
    if shortcut-ptr of localRow is NULL then
        ReadOrWrite(localRow);
    else
        hwccRecord ← shortcut-ptr of localRow;
        Acquire HWcc-latch for hwccRecord;
        ReadOrWrite(hwccRecord);
        Release HWcc-latch for hwccRecord;
    Release local-latch for local row;
else
    hwccRecord ← Get(k, CXL index);
    while hwccRecord is NULL do
        Ask the owner host to move k to CXL memory;
        if the host replies k does not exist then
            return NOT_EXIST;
        else
            hwccRecord ← Get(k, CXL index);
    Acquire HWcc-latch for hwccRecord;
    ReadOrWrite(hwccRecord);
    Release HWcc-latch for hwccRecord;
```

worker will search the corresponding CXL index. If the CXL index does not include all tuples for the queried range, the worker asks the corresponding remote host to move all tuples in that range to CXL memory and searches the CXL index again.

Insert and delete. Inserting a tuple to local partition requires creating a local row for the tuple and inserting it into the local index. The challenge is inserting a tuple to a remote partition. To address this challenge, our key idea is to let the remote host prepare the index entry and metadata for the insert and let the transaction worker perform the actual insert operation. Specifically, the remote host is requested to create a local row with the `is-valid` flag set to False, insert the local row to its local index, and move the local row to CXL memory (the value of the `is-valid` flag is copied from the local row to the SWcc row). The `is-valid` flag prevents other transaction workers from reading or writing this tuple. Afterward, the transaction worker gets the HWcc record from the CXL index, acquires the write lock, completes the insert by copying the tuple to SWcc row and setting the `is-valid` flag to True.

The algorithm for deleting a tuple is similar to the one for reading or writing a tuple. If the tuple belongs to the local partition, the transaction worker acquires the write lock and sets the `is-valid` flag to False to delete the tuple. To delete a tuple from a remote partition, the transaction worker asks the remote host to move the tuple to CXL memory, gets the HWcc record from the CXL index, and sets the `is-valid` flag to False. Deleted tuples are reclaimed using epoch-based reclamation (details in 3.6).

3.4 Concurrency Control

Tigon manages concurrency at two levels: 1) maintaining and accessing the CAT under concurrent accesses across hosts; 2) providing serializability for concurrent transactions.

3.4.1 Maintaining and accessing the CAT

Tigon has the following synchronization goals for maintaining the CAT. (1) Even during data movement, HWcc records can be safely inserted and deleted; (2) a worker thread on the owner host can safely use the `shortcut-ptr`, even when tuples are moving between CXL memory and local DRAM; (3) workers from different hosts can concurrently and efficiently get HWcc records by caching and reusing the pointers to HWcc records within a transaction.

Tigon achieves these goals using (1) the `local-latch`, which can only be accessed by a thread on the owner host; (2) the CXL index, which can be accessed by any host; and (3) the `HWcc-latch`, which can be accessed by any host.

Data movement. In Tigon, only threads on the owner host can insert and delete HWcc records for data movement. If a non-owner thread wants to insert or delete a record, it sends a message to the owner host. The owner host will make inserted HWcc records visible to non-owner hosts by putting their entry in the CXL index, which it only does after all of its metadata structures are valid. Deleted HWcc records are removed from the CXL index by the owner host. HWcc records are moved from CXL memory to local DRAM by the owner host following a data movement policy.

Shortcut pointer. Owner hosts in Tigon can use the `shortcut-ptr` of the local row to find and update a tuple in CXL memory without searching the CXL index. Owner hosts can also move their own tuple into or out of CXL memory by inserting or deleting an HWcc record to/from the CXL index. The key challenge is to ensure that the value of the `shortcut-ptr` always reflects the latest state of an HWcc record, even when a worker thread accesses it while another thread wants to move it. The `shortcut-ptr` must either point to the correct HWcc record or it must be `NULL` if the tuple is not resident in CXL memory. Tigon solves this challenge by reusing the database latch for mutual exclusion. Each operation (i.e., data movement and tuple get) needs to take the `local-latch` of the local row before executing, ensuring that only one worker from the owner host can proceed at a time. Given that a non-owner host cannot move a tuple directly, the tuple get operation can then safely follow the `shortcut-ptr` to get the HWcc record. For data movement operations, they will then insert the newly created HWcc record into the CXL index, update the `shortcut-ptr` to point to the HWcc record, and release the `local-latch`.

Non-owner get. Non-owner hosts get an HWcc record through the CXL index and cache the pointer to the HWcc record for subsequent accesses within a transaction. When a tuple is being moved from CXL memory back to local DRAM, the non-owner host must be able to safely dereference the pointer to the HWcc record. At the start of a CXL memory to local DRAM move, Tigon marks the SWcc row as invalid by setting the `is-valid` flag to 0, allowing the non-owner host to know that if they read this entry, they have followed a stale pointer to an invalid entry. Tigon eventually reclaims invalid entries using epoch-based reclamation (details in §3.6), when it knows there are no more possibly stale pointers that point to these invalid entries.

3.4.2 Serializing transactions

Two-phase locking. We adapt 2PL to Tigon, as described in §3.3. 2PL requires a transaction to acquire either a read or write lock before accessing a tuple, depending on the type of the access. Lock acquisitions and releases are broken into two consecutive phases: the transaction only acquires or upgrades locks in the first phase and only releases or downgrades locks after all locks are acquired. Tigon uses Strong Strict 2PL (SS2PL), which acquires locks during transaction execution and releases them only upon completion [12, 24]. Tigon prevents deadlocks by aborting a transaction when a lock request is denied (NO_WAIT policy), following prior research [76] showing that this approach yields better scalability than deadlock detection for in-memory databases with a similar setup to Tigon.

As the first work to leverage CXL memory for building efficient distributed transactional databases, we choose to support conventional concurrency control protocols, 2PL and next-key locking, rather than optimistic concurrency control (OCC) and multi-version concurrency control (MVCC), even though OCC and MVCC may yield better performance for read-heavy workloads as shown in prior research [38, 68, 77, 79]. Supporting these protocols in Tigon requires addressing non-trivial technical challenges and goes beyond the scope of this paper.

Phantoms and next-key locking. Phantom problems [24] occur because 2PL locks only individual keys but does not lock the gaps between them. For instance, when a transaction scans a range, it locks only the tuples encountered during the scan, but allows other transactions to modify the range's membership (e.g., through inserts), violating serializability.

The classic solution to addressing phantom problems is *next-key locking* [55]. Records are accessed via an ordered index (such as a B$^+$-tree) and for inserts, deletes, scans, and point queries on non-existing keys, locks are acquired for both the key or key range and the *next* key in order.

We adapt next-key locking for Tigon. Supporting next-key locking for a local index is straightforward, but a challenge arises when a transaction worker needs to access data in a remote partition. In this case, the worker relies on the CXL index to perform next-key locking. Unfortunately, the CXL index may not include the next key as it only includes a subset of tuples from the local index.

To address this limitation, we enhance the CXL index with additional next-key information. Each tuple in the CXL index is augmented with the `has-next-key` flag indicating whether its next key in the CXL index is also the next key in the local index. With this mechanism, the transaction worker can safely determine whether it can directly lock the next key in the CXL index or needs to request a remote host to move the tuple for the next key to CXL memory.

The `has-next-key` information may be updated whenever an insert or delete operation is performed on the local index or a tuple is moved into or out of CXL memory. Additionally, the `has-next-key` information can be used to determine whether the CXL index has missing keys with respect to the local index for a given range, which can be used to support range queries on the CXL index.

3.5 Logging and Recovery

As mentioned in §3.1, to avoid 2PC, Tigon maintains the CAT to ensure each host can perform and log all modifications to tuples involved in a transaction. Our logging protocol is adapted from SiloR [82], a fast and scalable logging protocol for in-memory databases. Since SiloR is designed for an optimistic concurrency control protocol, we adapt its key ideas to design a logging protocol for 2PL in Tigon.

Key ideas. Tigon shares high-level ideas with SiloR [82], including epoch-based group commit and parallel value logging. Specifically, Tigon guarantees that transactions with a smaller epoch number are serially ordered and committed before those with a larger epoch number. Transactions of each epoch are committed by flushing the buffered log records to local storage. To support parallel logging, each worker thread independently generates a log record for each write at the commit phase, including the value of the tuple being modified and the epoch number and version number associated with the tuple. The log records are sent to dedicated logger threads for persistence. To support recovery, the protocol guarantees that the log record with the highest epoch and the highest version number within that epoch represents the most recent value for the tuple.

Logging. Tigon associates each transaction with an epoch number, and each tuple with an epoch number and a version number (i.e., `epoch-version` in Figure 3). The epoch number of a tuple indicates the epoch of the most recent transaction that modified it, while the version number specifies the most recent version within that epoch.

Tigon maintains a global epoch number, E, that advances periodically and is shared across all hosts via the HWcc region of CXL memory. Since E advances slowly (e.g., every 10 ms in our prototype), synchronizations on E will not be a bottleneck as SiloR shows [82].

During transaction commit, the transaction worker sets the epoch number of the transaction by reading the current

E from CXL memory. When applying writes to the tuple, the transaction worker updates the epoch number and version number as follows. If the transaction's epoch number equals to the tuple's, the version number is incremented. Otherwise, the tuple's epoch number is updated to the transaction's epoch number and the version number is reset to 0, indicating the start of a new epoch.

Log records. A log record is generated for each write, insert, and delete. It includes the identifier of the tuple (i.e., table ID and primary key), the `epoch-version`, and the corresponding value of the tuple or the delete operation. Each transaction worker stores the log records in a memory buffer and passes it to the logger thread when it fills or at an epoch boundary. Each logger thread reads log buffers and flushes them to local storage. In additional to the global epoch number, Tigon maintains a local epoch number, e_l, for each logger thread, in the HWcc region. Each logger thread increments e_l as it flushes the log buffers, indicating that all log records with an epoch number less than e_l are persisted to local storage. Transactions in an epoch e are regarded as committed if e is less than e_l of all logger threads.

Recovery. With this logging protocol, Tigon can adopt the same parallel recovery algorithm as SiloR. That is, Tigon can move all log records of committed epochs to CXL memory and then divide them across worker threads such that each thread applies assigned log records in parallel. For a log record that corresponds to a database tuple, it is applied to the tuple if it has a larger `epoch-version` than the last log record applied to this tuple. Otherwise, this log record is skipped.

3.6 Implementation

We implement Tigon in C++ by building on the Lotus [85] codebase, which has ∼18,000 LoC. We added ∼5,000 LoC. Tigon provides read, write, delete, insert, and range query APIs for developers to implement transactions or parameterized transactions. We implement local index and CXL index using an existing B$^+$-tree implementation [84], which adopts the optimistic crabbing index concurrency control protocol [42]. We extend the B$^+$-tree to support next-key locking. Tigon adopts offset pointers [14] to make CXL-resident data structures position-independent. CXL memory is exposed as a CPU-less NUMA node by Linux. We modify the mimalloc [41] memory allocator to use the CXL memory region.

Each host in Tigon includes multiple worker threads for processing transactions and data movement requests. If a transaction needs to access data of another host, its worker thread will send a data movement request using CXL memory as a transport. A dedicated input thread pulls these requests and dispatches them round robin to the local message queues of worker threads. The worker thread will process data movement requests from other hosts when it waits for the result of a data movement request it makes, or after a transaction is finished. If a transaction aborts due to a conflict, Tigon retries it until success.

We implement Tigon's CXL-based transport layer as lock-free multi-producer single-consumer (MPSC) ringbuffers in CXL memory. The metadata of the ringbuffer (e.g., head and tail) is stored in the HWcc region while the buffer entries are stored in the non-HWcc region. Each input thread is assigned a ringbuffer for receiving messages from other hosts.

Tigon uses epoch-based reclamation (EBR) [25] to ensure memory safety. To adapt EBR to the CXL pod, Tigon stores a local epoch number for each worker thread (in addition to the global epoch number and the per-logger local epoch numbers in §3.5) in the HWcc region and maintains per-worker retired objects list in the local DRAM of each host. Worker threads check the global epoch number periodically to determine the current epoch and update their local epoch number. Worker threads reclaim the memory objects retired in a given epoch only after all workers have exited that epoch.

4 Evaluation

We seek to answer the following questions:

- What are the end-to-end performance benefits of Tigon compared with the state-of-the-art? (§4.2)
- How does varying the hardware cache-coherent (HWcc) memory budget impact Tigon's performance? (§4.3)
- What is the performance benefit of software cache coherence compared with using HWcc memory only? (§4.4)
- How does logging impact the throughput and latency of Tigon? (§4.5)
- How effective are our optimizations? (§4.6)

4.1 Experimental Setup

CXL pod emulation. We emulate a CXL pod on a machine that includes an Intel Xeon Platinum 8568Y+ CPU, 512 GB local DRAM, and a 128 GB CXL 1.1 memory device. Both the local DRAM and CXL memory use DDR5 4800 DRAM. The CXL memory module has a single memory channel and is connected to the CPU via a PCIe 5.0 x8 link. This CXL memory device has a 1.6× higher latency than the local DRAM (259 ns vs. 159 ns), and its bandwidth is 13% of the bandwidth of the local DRAM (31.8 GB/s vs. 238.3 GB/s) using Intel's Memory Latency Checker [3] under a 3:1 read/write ratio. The machine is equipped with a Samsung Datacenter SSD [5] for persistent storage and a 100 Gbps Mellanox MT42822 BlueField-2 ConnectX-6 NIC.

We run 8 virtual machines (VMs) on this machine, each with 5 vCPUs and 10 GB local DRAM. We use SR-IOV to create the virtualized network for our VMs, using TCP over Ethernet. We configure the VMs to share the CXL memory device to emulate a CXL pod that includes 8 machines sharing CXL memory. There are currently no physical CXL devices that support inter-host cache coherence. For our setup, the inter-VM cache coherence is implemented by the physical machine's cache coherence, which will be faster than real inter-host hardware cache

coherence (when it becomes available). We analyze the potential impact of higher-latency inter-host cache coherence in §4.3.

We limit the HWcc region of the emulated CXL pod to 200 MB for all experiments. That is, our CXL memory allocator is configured to allocate no more than 200 MB of HWcc memory, despite the emulated CXL pod offering 128 GB. We study the impact of varying HWcc memory budgets in §4.3 and adjust the size of the HWcc region for that experiment only.

Baselines. We compare Tigon with three distributed databases, Sundial [77], DS2PL [85], and Motor [79]. We do not compare with Lotus [85] because it does not support all five transactions in TPC-C. We choose Sundial and DS2PL as representatives of traditional partition-based shared-nothing databases that support general-purpose transactional workloads, just as Tigon supports them. DS2PL uses two-phase locking (2PL) for reads and writes, similar to Tigon. Sundial includes an advanced concurrency control protocol that uses OCC for reads and 2PL for writes. Both Sundial and DS2PL are vulnerable to the phantom problem [24] that Tigon correctly addresses. Motor is a state-of-the-art distributed database that uses RDMA-based shared disaggregated memory and achieves persistence via replication.

Improved baselines. For a fairer comparison with Tigon, we enhance our chosen baselines in a number of ways. We implement next-key locking in DS2PL to avoid the phantom problem. We enhance Sundial and DS2PL to support durability using our logging protocol (§3.5). Sundial and DS2PL support only two of the five transactions in TPC-C, so we added implementations of insert, delete, and scan so they could execute all five.

Sundial and DS2PL originally use the network as their transport layer as they adopt the shared-nothing architecture, whereas Tigon uses a queue in CXL memory. To demonstrate that Tigon's superior performance is not simply due to its faster transport layer, we upgrade Sundial and DS2PL to also pass messages over CXL memory (*i.e.,* Sundial-CXL and DS2PL-CXL). Tigon uses 16 MB of CXL memory that is not hardware cache-coherent (non-HWcc) for its message buffers and we give the baselines 512 MB because they pass larger messages.

Using a memory queue for message passing allows us to make another improvement: we repurpose an I/O thread as a worker thread (for executing transactions). The IO thread was necessary because it wrote to a network device and could block in the OS, while our memory queue is non-blocking. We name these improved baselines Sundial+ and DS2PL+. We later show that these improved baselines outperform the original Sundial and DS2PL, and only compare Tigon with these improved baselines (§4.2).

Configurations and metrics. We run Tigon, Sundial, and DS2PL in the VMs of the emulated CXL pod. For each test, we run each system for 30 seconds to warm up and report the performance for the following 30 seconds. We mainly present throughput since Tigon prioritizes throughput over latency using group commit, a common practice widely adopted in transactional databases [9, 33, 69, 75, 77, 85]. We set each epoch to be 10 ms for the group commit protocol.

We run Motor on four machines, each including two Intel Xeon Platinum 8380 CPUs, 256 GB local DRAM, and a 25 Gbps Mellanox ConnectX-6 NIC. We use one machine as the compute server and three machines as memory servers, as the paper specifies [79]. Motor replicates its data in the three memory servers to achieve durability. We configure Motor to use 40 CPU cores to match the configuration of Tigon and the other two baselines.

Benchmarks. We use the full TPC-C [7] and a variant of YCSB [15] as our benchmarks. We configure TPC-C to use 24 warehouses (2.2 GB total), where each warehouse is a partition, and store 3 warehouses in each host (one warehouse per worker thread). We evaluate the standard mix for TPC-C's five stored procedures: 45% NewOrder, 43% Payment, 4% OrderStatus, 4% Delivery, and 4% StockLevel. We configure YCSB to generate a table including 2.4M tuples, each with one 4-byte key column and ten 100-byte value columns (2.2 GB total) [15]. The table is range-partitioned into 8 partitions across hosts. Each transaction performs 10 read or write operations with a configurable read/write ratio. The key accessed by each operation is generated using the Zipfian distribution with skewness factors of 0.7 and 0.99. We use 0.7 and 0.99 to simulate medium-contention and high-contention workloads, respectively, as done by prior research [79].

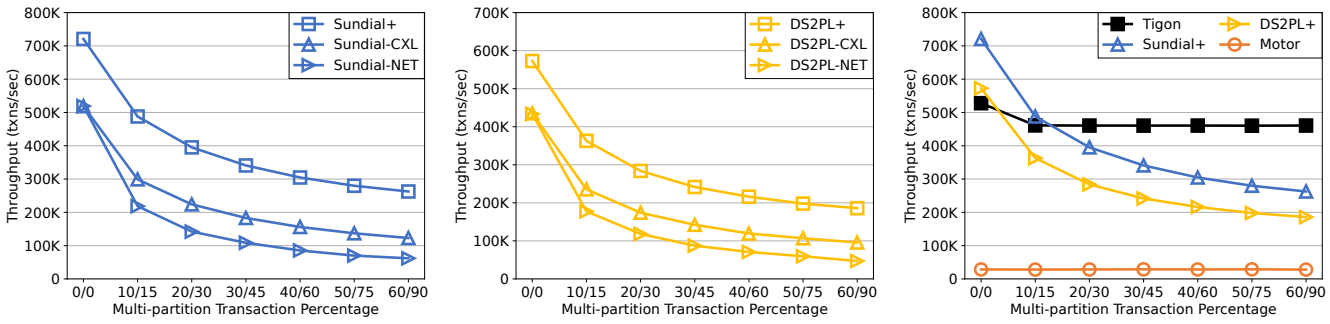
4.2 End-to-end Performance

We evaluate Tigon's end-to-end performance using TPC-C and YCSB.

TPC-C setup. The performance of the compared systems changes significantly based on how many transactions access data from multiple partitions. In the default configuration of TPC-C, 10% of NewOrder and 15% of Payment transactions access multiple warehouses (i.e., multi-partition transactions). Therefore, we vary the two percentages proportionally, up to 60% remote NewOrder and 90% remote Payment transactions (and we refer to this configuration as 60/90).

Performance of improved baselines. We measure the performance benefits of the improved baselines using TPC-C with varying percentages of multi-partition transactions. Figure 4a and 4b show that replacing the network with CXL memory increases the transaction throughput for the workloads that involve multi-partition transactions since these transactions need to exchange network messages during transaction execution and 2PC. For the 60/90 configuration, performance improves by 2.0× for both Sundial and DS2PL. Repurposing the I/O thread as a transaction thread brings the total improvement to 39% and 32% when there are no multi-partition transactions (0/0), and 4.2× and 3.9× at 60/90. The rest of our evaluation compares only against the optimized baselines.

TPC-C performance. Figure 4c shows TPC-C throughput under varying percentages of multi-partition transactions.



(a) Improved Sundial **(b)** Improved DS2PL **(c)** Tigon comparing with improved baselines

Figure 4: TPC-C performance comparison, varying percentages of multi-partition transactions. We optimize (a) Sundial and (b) DS2PL by replacing the original network transport layer (*-NET) with CXL (*-CXL), and again by replacing an I/O thread with a worker thread (Sundial+ and DS2PL+). (c) Then, we compare Tigon with Sundial+, DS2PL+, and Motor.

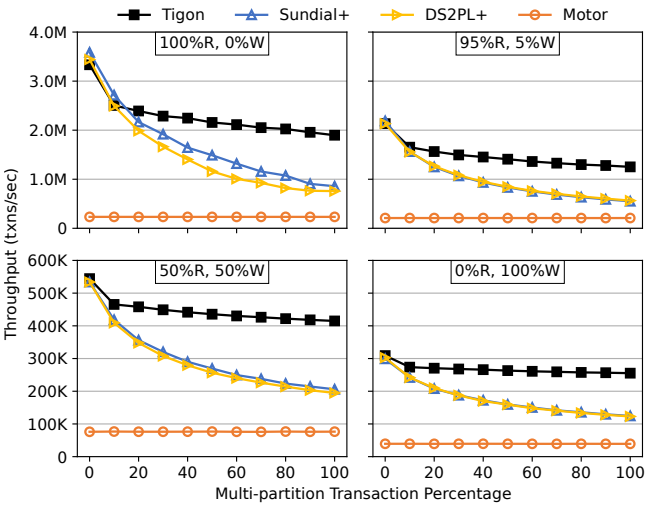


Figure 5: YCSB throughput, varying both read/write ratios and percentages of multi-partition transactions.

When there are no multi-partition transactions, Sundial+ and DS2PL+ are 37% and 8.5% faster than Tigon, respectively. The performance benefit of Sundial+ is due to its optimistic concurrency control protocol and the fact it does not avoid phantoms. Avoiding phantoms using next-key locking (§3.4.2) costs Tigon 10%-12% of its performance across all tested percentages of multi-partition transactions; for TPC-C's default configuration of 10/15, a variant of Tigon that allows phantoms (not shown in the figure) outperforms Sundial+ by 5.8%.

As the percentage of multi-partition transactions increases to 60/90, Tigon outperforms Sundial+ by 75% and DS2PL+ by 2.5×. The throughput of Sundial+ and DS2PL+ drops significantly due to the overhead of inter-host communication during transaction execution and 2PC. For instance, in the 60/90 case, Sundial+ and DS2PL+ send 3.3 and 4.1 messages per transaction, respectively, whereas Tigon moves the data that will be accessed across hosts to CXL memory (during the warmup phase) and does not send messages during transaction execution. More specifically, Tigon moves the `CUSTOMER` and `STOCK` tables to CXL memory, with `CUSTOMER` containing 720K tuples and `STOCK` containing 2.4M tuples. The two tables consume 176 MB of HWcc memory (for metadata) and 1.6 GB CXL memory in total. No data is moved from CXL memory to local DRAM during the experiment. For TPC-C, supporting software cache coherence allows Tigon to use 8.2× more CXL memory for sharing data.

Compared with Motor, Tigon achieves 15.9×-18.5× higher throughput mainly because Motor shares and replicates data in RDMA-based memory servers and suffers from the high latency of one-sided RDMA operations. We observe that the performance of Motor, approximately 30K/s, is constrained by the limited network bandwidth (25 Gbps) of the machines available to us. The original paper [79] reports that Motor achieves a maximum TPC-C throughput of approximately 100K/s with 24 warehouses and 40 CPU cores. Tigon uses the same configuration and its throughput ranges from 460K/s to 528K/s, as shown in Figure 4c.

YCSB setup. We run YCSB with a skewness factor of 0.7 under varying percentages of multi-partition transactions and different read/write ratios (i.e., 100R/0W, 95R/5W, 50R/50W, and 0R/100W). To generate a multi-partition transaction, we randomly choose a remote partition and have half (5) of the operations access that remote partition.

YCSB performance. Figure 5 shows the results. When there are no multi-partition transactions, the evaluated systems (except Motor) perform similarly (within 3.3%) for workloads with writes. For the read-only workload, Tigon outperforms both Sundial+ and DS2PL+ once the percentage of multi-partition transactions reaches 20% or higher. Comparing Tigon with Sundial+ at 100% multi-partition transactions, Tigon is 2.0×–2.3× faster. Tigon moves all 2.4M tuples to CXL memory as they are accessed across hosts. These tuples consume 112 MB of HWcc memory and 2.6 GB CXL memory in total. No Data is moved from CXL memory back to local DRAM, as HWcc memory usage remains well below the 200 MB limit.

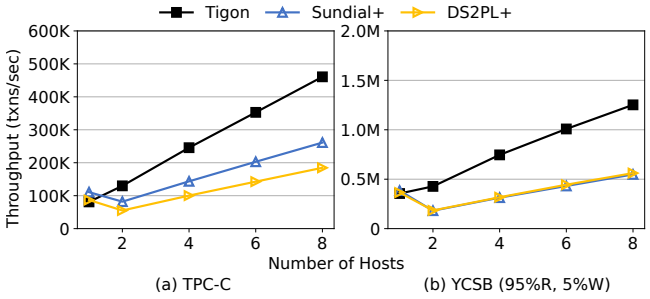


Figure 6: Throughput of TPC-C (60% remote NewOrder and 90% remote Payment) and YCSB (95%R, 5%W, 100% multi-partition transactions) on 1, 2, 4, 6, and 8 hosts.

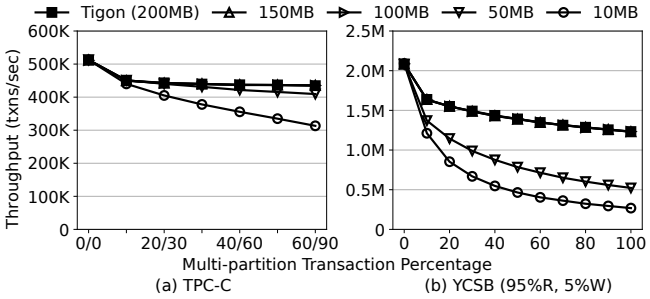


Figure 7: Throughput of TPC-C and YCSB (95%R, 5%W), varying both hardware cache-coherent memory budgets and percentages of multi-partition transactions.

Motor's performance is independent of the percentage of multi-partition transactions, but it is 5.4×–14.3× slower than Tigon. Results with a skewness factor of 0.99 show a similar trend. For example, with 100% multi-partition transactions, Tigon outperforms Sundial+ and DS2PL+ by 2.7× and 3.5× for the 50R/50W workload, respectively. We use a skewness factor of 0.7 for all remaining tests.

Scalability. We evaluate the scalability of Tigon, Sundial+, and DS2PL+ by running TPC-C with 60/90 multi-partition transactions and YCSB (95% reads, 5% writes) with 100% multi-partition transactions on configurations with 1, 2, 4, 6, and 8 hosts. As shown in Figure 6, Tigon scales efficiently, achieving a 5.7× throughput improvement for TPC-C and a 3.5× improvement for YCSB as the number of hosts increases from 1 to 8. In comparison, Sundial+ and DS2PL+ exhibit significantly lower scalability, with throughput gains of only 2.4× and 2.1× for TPC-C, and 1.4× and 1.5× for YCSB, respectively. The slope of Tigon's TPC-C scaling graph from 2 to 8 processors is 55.1Ktx/s per processor, while Sundial+ is 29.9Ktx/s per processor and DS2PL+ is 21.6Ktx/s per processor. For YCSB the slopes are 137.5Ktx/s per processor for Tigon, and 61.1Ktx/s per processor for Sundial+ and 63.0Ktx/s per processor for DS2PL+.

We are unable to empirically determine the scalability limit of Tigon due to hardware constraints that prevent large-scale experiments. Performance could plateau for many reasons: limited scalability of atomic instructions over CXL memory, especially under contention, limited scalability of hardware cache coherence, or the fixed size of the HWcc region could all become bottlenecks. Identifying and analyzing these limits is an important direction for future research.

4.3 Impact of Limited HWcc Memory

We measure TPC-C performance while varying the HWcc memory budget from 10 MB to 200 MB. Figure 7 shows the results. It is encouraging that even with as little as 50 MB of HWcc memory, Tigon performs only 5.8% slower than the configuration with unlimited HWcc memory.

With only 10 MB of HWcc memory, performance suffers because transactions wait for data to be moved into and out of CXL memory. When running TPC-C with a 60/90 mix of multi-partition transactions, 10 MB of HWcc memory requires Tigon to move 16K tuples in and out of CXL memory every second. Bandwidth is not a factor limiting performance; Tigon consumes only 367 MB/s of CXL bandwidth (approximately 1.1% of device bandwidth) in this configuration.

YCSB with a 95/5 read/write mix requires 100 MB to achieve full performance because there is more multi-partition sharing in YCSB than TPC-C—YCSB has only a single table and every data tuple has a probability of being remotely accessed. At 100% multi-partition transactions and with 10 MB HWcc memory, Tigon moves 110K tuples every second and consumes 2.9 GB/s of CXL bandwidth (approximately 9.1% of device bandwidth).

Inter-host cache coherence. We emulate the latency of inter-host cache coherence (i.e., back invalidations [2]) using inter-core cache coherence on our hardware prototype. This underestimates the cost of inter-host cache coherence for our evaluated systems. We cannot directly measure the number of back-invalidates our workloads would generate, because there are no hardware counters that report it and accurate modeling would require knowing the cache state. However, we can measure inter-host invalidates for our software cache coherence protocol (§3.2) for 60/90 TPC-C. We measure 12.0 million tuple accesses that require inter-host invalidations (which is 14.5% of total accesses, due to locality). If we conservatively estimate back-invalidations to be 4× slower than local invalidations, they would reduce performance by 41.4% (computed using the CXL miss latency times four). Tigon would be 2.8%, 45%, and 9.6× faster than Sundial+, DS2PL+, and Motor, respectively.

4.4 Software Cache Coherence

We evaluate the performance impact of software cache coherence by running TPC-C and YCSB (95% reads, 5% writes) in four configurations: only using HWcc memory (NoSWcc), using HWcc memory and only using non-temporal loads/stores to access CXL memory in the software cache-coherent (SWcc) region (NonTemporal), using HWcc memory and only allowing a

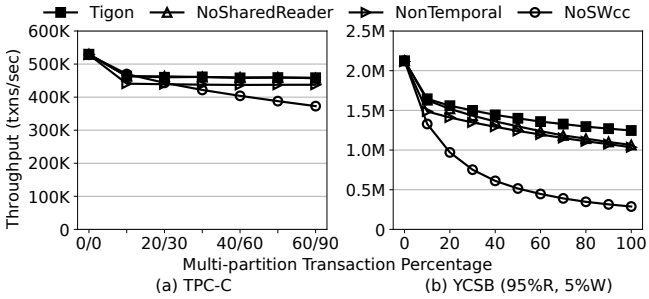


Figure 8: Throughput of TPC-C and YCSB (95%R, 5%W) with different software cache coherence protocols and varying percentages of multi-partition transactions.

	1ms	10 ms	20 ms	30 ms	40 ms	50 ms
TPC-C						
Tput (Ktx/s)	508	525	532	535	537	540
p50 (ms)	17.7	22.6	26.0	31.6	37.3	43.3
p99 (ms)	345.0	54.9	57.0	64.7	75.2	86.8
YCSB (50% read and 50% write)						
Tput (Ktx/s)	516	534	538	541	542	545
p50 (ms)	12.4	24.6	30.8	36.8	41.5	45.0
p99 (ms)	373.1	62.3	72.3	82.6	89.6	95.4

Table 1: Throughput, in Ktx/s (1,000 × transactions/second) and latency (in ms) for different epoch durations, running TPC-C and YCSB (50% reads, 50% writes) without multi-partition transactions.

single host to read data in the SWcc region (NoSharedReader), and Tigon with all optimizations for software cache coherence. NoSWcc avoids cache coherence problems by only using HWcc memory, but it suffers from the limited size of the HWcc region. NonTemporal provides cache coherence by avoiding the cache; accesses to the SWcc region use non-temporal loads and stores which bypass the CPU cache. NoSharedReader only allows reads of SWcc data for a single host at a time.

TPC-C. Figure 8 shows the benefits of Tigon's software cache coherence. When there are no multi-partition transactions, no data is moved to CXL memory and all variants perform identically. Across all tested percentages of multi-partition transactions, the NonTemporal configuration shows 4.5%-5.1% less performance than Tigon. The small performance win is because Tigon already caches tuples in local DRAM, which obviates the performance advantage of cacheable access to CXL memory. NoSWcc performs well at low multi-partition transaction percentages because it uses the HWcc region to store data; this data can be cached, leading to higher performance. However, as the number of multi-partition transactions increases, the performance of NoSWcc drops because it requires frequent data movement. At 60/90, NoSWcc is 19% slower than Tigon. The loss of performance comes from the latency of moving 9K tuples in and out of CXL memory every second. The workload consumes 641 MB/s of CXL bandwdith, well within the device's capabilities. TPC-C does not have read-shared data, so NoSharedReader has almost identical performance with Tigon.

YCSB. YCSB has significant read-sharing and shared data. NoSWcc is 19% slower than Tigon for 10% multi-partition transactions, but it is 4.3× slower (and the lowest performing option) at 100% multi-partition transactions. At 100% multi-partition transactions NoSWcc's performance is limited by the latency of data movement to the HWcc region—115K tuples/s (but only 3.0 GB/s bandwidth use; ∼9.4% of CXL bandwidth). Across all tested percentages of multi-partition transactions, Tigon is 11%-20% faster than NonTemporal because data is repeatedly accessed and Tigon can cache the tuples, which is much faster than non-temporal reads. But the advantage is small because Tigon caches tuples in local DRAM. At 100% multi-parition transactions, NoSharedReader is 15% slower than Tigon. NoSharedReader's performance suffers because it manipulates the `SWcc-bitmap`, and the read-sharing of data in YCSB means that NoSharedReader incurs 2.6× more cacheline flushes than Tigon.

4.5 Logging and Latency

Tigon uses epoch-based group commit (based on SiloR [82]) to ensure that it can maintain high throughput at reasonable latency. Committing logs at end of each transaction would provide low latency, but the resulting small, synchronous writes would severely under-utilize storage bandwidth. The use of virtualization further drops the storage bandwidth realized inside the VMs (from 600 MB/s to 380 MB/s, tested using `dd` with 4MB block size). To better utilize storage performance, Tigon collects 4 MB of logs before flushing them to SSD.

Table 1 shows the throughput and latency for a variety of logging epochs, running TPC-C and YCSB (50% reads, 50% writes) with no multi-partition transactions. For a 10 ms epoch, when compared with a 50 ms epoch, throughput for TPC-C drops only 2.8%, while its p50 latency drops by 48%. With a 10 ms epoch, Tigon has only 6.0% lower throughput for TPC-C compared with Tigon without logging. YCSB shows a similar result. Since Sundial+ and DS2PL+ adopt the same logging protocol, their latencies are similar to Tigon. Motor does not do group commit, but instead uses three in-memory replicas to achieve durability, so their p50 latencies are in the hundreds of microseconds. Tigon, instead, prioritizes throughput over latency using group commit.

4.6 Optimizations

Using CLOCK. Tigon moves data into CXL memory on demand, and uses the CLOCK algorithm to determine which data to move back to local DRAM (§3.2). Tigon uses CLOCK to minimize its use of HWcc memory and reduce the overhead of maintaining the replacement metadata; we compare its performance to the more accurate LRU policy to quantify the accuracy sacrificed for metadata efficiency.

Implementing LRU requires more CXL memory and more synchronization (than CLOCK) to maintain the LRU list. If we place the LRU list pointers in HWcc memory, YCSB will require 149 MB of HWcc memory to move all its tuples into CXL memory, which is 33% more than CLOCK (112 MB). If we limit the HWcc memory budget to 100 MB, LRU is 2.4× slower than CLOCK for YCSB with a 95/5 read/write mix and 100% multi-partition transactions because HWcc memory is oversubscribed. If we set the HWcc memory budget to unlimited and run the same experiment, LRU is 17% slower than CLOCK because of lock contention while manipulating the LRU list. CLOCK is therefore preferred due to its metadata efficiency.

Shortcut pointer. Tigon maintains the `shortcut-ptr` in the local row (Figure 3) to allow the owner host to find the tuple in CXL memory without looking through the CXL index. Across varying percentages of multi-partition transactions (excluding 0%), shortcut pointers improve throughput by 16% for TPC-C, and by 8.3%–24% for YCSB with 95% reads and 5% writes.

Tracking tuple modifications. Tigon allows the owner host to read CXL-resident tuples from local DRAM rather than CXL memory when the tuple is clean (i.e., the `is-dirty` flag in Figure 3 is False). This optimization is important for read-heavy workloads; YCSB read-only throughput improves by 60% for 10% multi-partition transactions and by 27% for 100% multi-partition transactions.

5 Related Work

CXL-enabled memory tiering. A lot of recent work on CXL memory management focuses on system software moving memory pages between local and CXL memory on behalf of applications [22, 54, 59, 65, 83]. Much of the work focuses on accurate detection of hot pages and policies for moving pages. Some hot pages only contain a few hot lines [63], making it difficult to recoup the cost of migration and motivating systems where hardware migrates cachelines [83].

CXL-enabled memory pooling and sharing. Recent work explores how CXL-enabled memory pooling and sharing can reduce cost in datacenters and HPC systems [11, 28, 45, 70], benefit serverless computing [28, 57], and accelerate distributed applications [50, 51, 81]. Distributed applications sharing CXL memory are vulnerable to partial failures [81, 86]. CXL-SHM [81] describes a distributed memory management system based on reference counting that tolerates partial failures.

Databases over CXL memory. Many papers consider optimizing databases using CXL memory [8, 10, 39, 60] such as elastically allocating CXL memory [39] or leveraging it for data shuffling [10]. One paper considers mapping SSD to the memory address space through CXL protocols and adopting hardware-based optimizations (e.g., (de)compression) when databases interact with this memory address space [40]. Two papers discuss the research opportunities of building databases over CXL memory [27, 43], but neither considers building a distributed transactional database over CXL memory. Pasha [29] describes a database architecture for CXL pods, but Tigon is the first system designed, implemented, and evaluated for a pod.

Distributed databases. Traditional distributed databases adopt a partition-based shared-nothing architecture, but their performance degrades quickly as the number of multi-partition transactions increases [17, 33, 56, 66]. Many papers optimize this architecture by reducing the number of multi-partition transactions [18, 49, 58, 61], optimizing concurrency control protocols [37, 52, 77, 85], or eliminating or reducing the cost of 2PC [23, 34, 46, 48, 67]. Tigon's performance remains robust because it uses CXL memory to synchronize multi-partition transactions instead of relying on message passing.

Many papers study RDMA-based distributed transactional databases [4, 13, 19–21, 32, 44, 62, 71–73, 75, 78–80, 87]. Some of them optimize partition-based databases [13, 20, 21, 32, 62, 71–73] while more recent papers adopt a shared disaggregated memory architecture to avoid multi-partition transactions [4, 19, 44, 75, 78–80, 87]. Motor [79] is a recent instance of this design and our evaluation shows the limits of its performance due to the high cost of RDMA relative to CXL memory access (§4).

6 Conclusion

This paper introduces Tigon, the first distributed in-memory database that synchronizes cross-host concurrent data accesses via atomic operations on CXL memory. Tigon addresses the inherent limitations of CXL memory, including higher latency, lower bandwidth, and limited hardware support for cache coherence, through three key innovations: a novel software cache coherence protocol for extending the cache-coherent region, efficient concurrency control protocols for maintaining and accessing data in CXL memory, and a scalable logging protocol for avoiding two-phase commit. Tigon is publicly available at `https://github.com/ut-datasys/tigon`.

7 Acknowledgment

We thank Nam Sung Kim at UIUC for technical consultation and for providing an experimental testbed. We thank our shepherd, Phillip Levis, and the anonymous reviewers for their constructive feedback. We thank Zixuan Wang at SJTU for his help with RDMA setup. Our work is supported in part by PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We used the Chameleon testbed [35] supported by the National Science Foundation for development and artifact evaluation. Vijay Chidambaram was partially supported by Toyota during this project.

References

[1] CockroachDB Serverless. https://www.cockroachlabs.com/blog/announcing-cockroachdb-serverless/. (Accessed: May 2025).

[2] Compute Express Link (CXL) Specification, Revision 3.2. https://computeexpresslink.org/wp-content/uploads/2024/11/CXL-Specification_rev3p2_ver1p0_2024October2_evalcopy.pdf. (Accessed May 2025).

[3] Intel® Memory Latency Checker v3.11b. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html. (Accessed May 2025).

[4] Oracle RAC. https://www.oracle.com/technetwork/database/options/clustering/overview/new-generation-oracle-rac-5975370.pdf. (Accessed: May 2025).

[5] Samsung MZQL2960HCJR-00A07. https://semiconductor.samsung.com/us/ssd/datacenter-ssd/pm9a3/mzql2960hcjr-00a07/. (Accessed: May 2025).

[6] SK hynix Presents CXL Memory Solutions Set to Power the AI Era at CXL DevCon 2024. https://news.skhynix.com/sk-hynix-presents-ai-memory-solutions-at-cxl-devcon-2024/. (Accessed May 2025).

[7] TPC Benchmark C. https://www.tpc.org/tpcc/. (Accessed May 2025).

[8] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna T. Malladi, and Yang-Seok Ki. Enabling CXL memory expansion for in-memory database management systems. In Spyros Blanas and Norman May, editors, *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*, pages 8:1–8:5. ACM, 2022.

[9] 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. Socrates: The new SQL server in the cloud. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1743–1756. ACM, 2019.

[10] Alexander Baumstark, Marcus Paradies, Kai-Uwe Sattler, Steffen Kläbe, and Stephan Baumann. So far and yet so near - accelerating distributed joins with CXL. In Carsten Binnig and Nesime Tatbul, editors, *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, pages 7:1–7:9. ACM, 2024.

[11] Daniel S. Berger, Yuhong Zhong, Fiodar Kazhamiaka, Pantea Zardoshti, Shuwei Teng, Mark D. Hill, and Rodrigo Fonseca. Octopus: Scalable Low-Cost CXL Memory Pooling. https://arxiv.org/pdf/2501.09020, 2025. (Accessed May 2025).

[12] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[13] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[14] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 191–203, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] F.J. Corbató and Project MAC (Massachusetts Institute of Technology). *A PAGING EXPERIMENT WITH THE MULTICS SYSTEM*. Project MAC. Massachusetts Institute of Technology, 1968.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. 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. Spanner: Google's globally-distributed database. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[18] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1):48–57, 2010.

[19] Alex Depoutovitch, Chong Chen, Per-Åke Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. Taurus MM: bringing multi-master to the cloud. *Proc. VLDB Endow.*, 16(12):3488–3500, 2023.

[20] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[21] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70. ACM, 2015.

[22] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and general datacenter transactions for on-disk databases. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 343–360. USENIX Association, 2023.

[24] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[25] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[26] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[27] Yunyan Guo and Guoliang Li. A cxl-powered database system: Opportunities and challenges. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5593–5604. IEEE, 2024.

[28] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma, Haifeng Gong, and YongWei Wu. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 421–437, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. Pasha: An efficient, scalable database architecture for cxl pods. In *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org, 2025.

[30] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. Memory sharing with cxl: Hardware and software design approaches, 2024.

[31] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, and Nam Sung Kim. Demystifying a cxl type-2 device: A heterogeneous cooperative computing perspective. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1504–1517, 2024.

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 185–201. USENIX Association, 2016.

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

[34] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 145–161. ACM, 2021.

[35] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[36] Kimberly Keeton. The machine: An architecture for memory-centric computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Ziliang Lai, Hua Fan, Wenchao Zhou, Zhanfeng Ma, Xiang Peng, Feifei Li, and Eric Lo. Knock out 2pc with practicality intact: a high-performance and general distributed transaction protocol. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 2317–2331. IEEE, 2023.

[38] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, 2011.

[39] Donghun Lee, Thomas Willhalm, Minseon Ahn, Suprasad Mutalik Desai, Daniel Booss, Navneet Singh, Daniel Ritter, Jungmin Kim, and Oliver Rebholz. Elastic use of far memory for in-memory database management systems. In Norman May and Nesime Tatbul, editors, *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*, pages 35–43. ACM, 2023.

[40] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. Database kernels: Seamless integration of database systems and fast storage via cxl. In *CIDR*, 2024.

[41] Daan Leijen, Benjamin Zorn, and Leonardo De Moura. *Mimalloc: Free List Sharding in Action*, volume 11893 of *Lecture Notes in Computer Science*, page 244–265. Springer International Publishing, Cham, 2019.

[42] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 3:1–3:8. ACM, 2016.

[43] Alberto Lerner and Gustavo Alonso. CXL and the return of scale-up database engines. *CoRR*, abs/2401.01150, 2024.

[44] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. Gaussdb: A cloud-native multi-primary database with compute-memory-storage disaggregation. *Proc. VLDB Endow.*, 17(12):3786–3798, August 2024.

[45] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancouver, BC Canada, March 2023.

[46] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[47] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1203–1217, New York, NY, USA, 2025. Association for Computing Machinery.

[48] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based commit and replication in distributed OLTP databases. *Proc. VLDB Endow.*, 14(5):743–756, 2021.

[49] Yi Lu, Xiangyao Yu, and Samuel Madden. STAR: scaling transactions through asymmetric replication. *Proc. VLDB Endow.*, 12(11):1316–1329, 2019.

[50] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, and Tao Ma. HydraRPC: RPC in the CXL era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 387–395, Santa Clara, CA, July 2024. USENIX Association.

[51] Suyash Mahar, Ehsan Hajyjasini, Seungjin Lee, Zifeng Zhang, Mingyao Shen, and Steven Swanson. Telepathic datacenters: Fast rpcs using shared cxl memory, 2024.

[52] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, 2014.

[53] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[54] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[55] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 392–405. Morgan Kaufmann, 1990.

[56] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[57] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, and Nicolai Oswald. Āpta: Fault-tolerant object-granular cxl disaggregated memory for accelerating faas. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–215, 2023.

[58] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 61–72. ACM, 2012.

[59] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Mtm: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817, New York, NY, USA, 2024. Association for Computing Machinery.

[60] Niklas Riekenbrauck, Marcel Weisgut, Daniel Lindner, and Tilmann Rabl. A three-tier buffer manager integrating CXL device memory for database systems. In *40th International Conference on Data Engineering, ICDE 2024 - Workshops, Utrecht, Netherlands, May 13-16, 2024*, pages 395–401. IEEE, 2024.

[61] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, 2016.

[62] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[63] Yan Sun, Jongyul Kim, Douglas Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems, November 2024.

[64] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Bijan Tabatabai, James Sorenson, and Michael M. Swift. FBMM: Making memory management extensible with filesystems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 785–798, Santa Clara, CA, July 2024. USENIX Association.

[66] 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. Cockroachdb: The resilient geo-distributed SQL database. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online*

conference [Portland, OR, USA], June 14-19, 2020, pages 1493–1509. ACM, 2020.

[67] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM, 2012.

[68] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[69] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1041–1052. ACM, 2017.

[70] Jacob Wahlgren, Maya Gokhale, and Ivy B. Peng. Evaluating emerging cxl-enabled memory pooling for hpc systems. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, November 2022.

[71] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[72] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 335–347. USENIX Association, 2017.

[73] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104. ACM, 2015.

[74] Emmett Witchel. Challenges and opportunities for systems using CXL memory. In *ASPLOS 2024*, 2024.

[75] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. Polardb-mp: A multi-primary cloud-native database via disaggregated shared memory. In Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan, editors, *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, pages 295–308. ACM, 2024.

[76] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, 2014.

[77] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, 2018.

[78] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transaction can scale. *Proc. VLDB Endow.*, 10(6):685–696, 2017.

[79] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling Multi-Versioning for distributed transactions on disaggregated memory. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 801–819, Santa Clara, CA, July 2024. USENIX Association.

[80] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: fast one-sided rdma-based distributed transactions for disaggregated persistent memory. In Dean Hildebrand and Donald E. Porter, editors, *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 51–68. USENIX Association, 2022.

[81] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 658–674, New York, NY, USA, 2023. Association for Computing Machinery.

[82] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 465–477, USA, 2014. USENIX Association.

[83] Yuhong Zhong, Daniel S. Berger, Carl A. Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 37–56. USENIX Association, 2024.

[84] Xinjing Zhou. BTree. https://github.com/zxjcarrot/2-Tree/tree/master/backend/btreeolc. (Accessed: May 2025).

[85] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. Lotus: Scalable multi-partition transactions on single-threaded partitioned databases. *Proc. VLDB Endow.*, 15(11):2939–2952, 2022.

[86] Zhiting Zhu, Newton Ni, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, and Emmett Witchel. Lupin: Tolerating partial failures in a cxl pod. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, DIMES '24, page 41–50, New York, NY, USA, 2024. Association for Computing Machinery.

[87] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. Is scalable OLTP in the cloud a solved problem? In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023.