Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases

Xinjun Yang
Alibaba Cloud Computing
Sunnyvale, CA, USA

Yingqiang Zhang
Alibaba Cloud Computing
Hangzhou, China

Hao Chen*
Alibaba Cloud Computing
Hangzhou, China

Feifei Li
Alibaba Cloud Computing
Hangzhou, China

Gerry Fan
XConn Technologies
San Jose, CA, USA

Yang Kong
Alibaba Cloud Computing
Hangzhou, China

Bo Wang
Alibaba Cloud Computing
Hangzhou, China

Jing Fang
Alibaba Cloud Computing
Hangzhou, China

Yuhui Wang
Alibaba Cloud Computing
Hangzhou, China

Tao Huang
Alibaba Cloud Computing
Hangzhou, China

Wenpu Hu
Alibaba Cloud Computing
Hangzhou, China

Jim Kao
XConn Technologies
San Jose, CA, USA

Jianping Jiang
XConn Technologies
San Jose, CA, USA

Abstract

Memory disaggregation has become a major trend in cloud-native databases. However, most existing memory disaggregation solutions suffer from read/write amplification, limited bandwidth, inefficient recovery, and challenges in data sharing. Fortunately, the emerging CXL technology introduces new opportunities for memory disaggregation design in cloud-native databases.

To overcome these challenges, we leverage the CXL switch to design *PolarCXLMem*, a CXL-switch-based disaggregated memory system for cloud-native databases. To the best of our knowledge, *PolarCXLMem* is the first CXL-switch-based disaggregated memory system. Building on *PolarCXLMem*, we propose a novel instant recovery scheme, *PolarRecv*, which enables instant recovery and fast buffer pool warm-up after a crash. To further support *PolarCXLMem* in multi-primary databases, we design a new cache coherency protocol that facilitates data sharing between database nodes based on *PolarCXLMem*. Finally, we evaluate *PolarCXLMem* with PolarDB, a widely deployed cloud-native database, under various workloads. This is the first study, to our knowledge, that investigates the performance of CXL-based disaggregated memory in a commercially deployed cloud-native database. Our evaluation shows that *PolarCXLMem* can improve throughput by up to 2.1× in pooling scenarios and 1.55× in sharing scenarios compared to RDMA-based systems.

*Hao Chen is the corresponding author (ch341982@alibaba-inc.com).

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724460

CCS Concepts

• **Information systems** → **DBMS engine architectures**; • **Computer systems organization** → **Cloud computing**.

Keywords

Compute Express Link (CXL), cloud-native databases, memory disaggregation

ACM Reference Format:
Xinjun Yang, Yingqiang Zhang, Hao Chen*, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724460

1 Introduction

Compute Express Link (CXL) has emerged as a promising solution for low-latency, high-bandwidth interconnects in data centers, with features like inherent cache coherency, native load/store support, memory pooling/sharing across multiple hosts and devices [41, 42, 53], making it an attractive option in memory-intensive applications [21, 24, 31].

One area where CXL's capabilities are especially transformative is cloud databases, which have evolved from storage disaggregation towards memory disaggregation to enhance scalability and resource utilization [13, 33, 46, 64]. However, existing RDMA-based solutions face critical challenges such as limited bandwidth, higher latency, limited concurrency [45, 55] and complexity in managing cache coherency [22, 45]. These limitations have driven the need for more efficient interconnect technologies. The emergence of CXL addresses these challenges by offering low-latency, high-bandwidth connections with native cache coherency and memory pooling/sharing support, significantly improving the performance of cloud databases [6, 24, 31]. CXL's ability to simplify memory

pooling/sharing and enable large-scale memory disaggregation unlocks new opportunities for optimizing cloud database architectures, making it a promising solution for overcoming existing bottlenecks in memory management and data processing.

This paper first reviews the use of RDMA-based disaggregated memory in commercially deployed cloud-native databases and identifies the following limitations: (1) Disaggregated memory and local memory are typically organized in a tiered structure, with data transferred between them at the page level (typically 4-16 KB) [46, 64]. Even small data requests trigger the transfer of entire pages, causing significant read/write amplification (up to dozens of times, as shown in our evaluation). Additionally, maintaining a local buffer pool increases memory overhead, and buffer pool misses lead to frequent RDMA read/write operations, significantly reducing performance. (2) During crash recovery, while the database follows its usual recovery logic, it reduces storage I/O costs when the required pages reside in disaggregated memory. However, the overall recovery process still takes considerable time, as the disaggregated memory is not leveraged to optimize the recovery scheme itself and still relies on the ARIES-style recovery policy. (3) Disaggregated memory systems are predominantly based on RDMA networks. While RDMA offers substantial improvements over TCP/IP, it faces challenges under high concurrency. Bottlenecks often arise from RDMA NICs, such as implicit contention on doorbell registers and cache thrashing [45, 55]. (4) When disaggregated memory is used for data sharing in multi-primary databases [33, 58], the database must manage its own cache coherency mechanisms, which adds overhead and limits performance.

To overcome these limitations, we leverage the world's first CXL switch [50] to design a *CXL-switch-based disaggregated memory* architecture, named *PolarCXLMem*. To the best of our knowledge, *PolarCXLMem* is the first CXL-switch-based disaggregated memory. *PolarCXLMem* is specifically optimized for cloud-native databases, addressing the limitations of RDMA-based solutions by providing high performance, cost-effectiveness, instant recovery, and efficient data sharing. In our design, we argue that with CXL, there is no need for a tiered memory structure. Compared with RDMA, CXL offers sufficient speed and supports native memory load/store instructions, allowing it to directly store all buffered pages. This design eliminates the read/write amplification caused by page copying in tiered structures, saving bandwidth resources. Meanwhile, eliminating the local buffer avoids the local memory overhead and saves costs. DRAM, which has limited scalability, constitutes a significant portion of hardware expenses [48]–accounting for 50% of server costs in Azure [1] and 40% of rack costs at Meta [41]. Additionally, this approach simplifies the system design with minimal modifications to the existing architecture, preserving system stability–a critical factor for commercially deployed systems. Our evaluation shows that storing pages in CXL memory achieves performance on par with local memory.

Building on *PolarCXLMem*, we propose a novel instant recovery scheme, *PolarRecv*. In *PolarRecv*, we move the entire buffer pool, including both data and metadata, to CXL memory. With CXL's low latency and support for load/store instructions, the database can directly operate on the buffer pool's data and metadata without requiring changes to the database logic. After a database crash, the buffered data and its metadata remain intact in CXL memory. By designing a tailored recovery policy, we can restore the buffered data to a consistent state from CXL memory, bypassing the heavy conventional recovery process and significantly improving recovery performance.

Furthermore, we adapt *PolarCXLMem* for data sharing in a multi-primary database. Since CXL 3.0 devices, which inherently support cache coherency, are not yet available, we design a new cache coherency protocol. Existing RDMA-based cache coherency schemes in multi-primary databases [58] synchronize data at page-level granularity, resulting in significant read/write amplification and high latency. Additionally, invalidation messages rely on the RDMA network, which is less efficient. In contrast, our proposed CXL-based cache coherency protocol synchronizes data at the cache line (64B) granularity, minimizing synchronization overhead. Invalidation messages utilize the low-latency CXL interface, further reducing latency and improving efficiency.

Finally, we implemented a CXL-based disaggregated memory using the CXL 2.0 switch in a commercially deployed cloud-native database, accommodating both single-node and multi-primary configurations. We evaluated and analyzed our design using various synthetic and real-world workloads.

We summarize our main contributions as follows:

- We conducted a comprehensive review of existing RDMA-based disaggregated memory designs and identified their key limitations. To address these challenges, we designed and implemented a CXL-switch-based disaggregated (shared) memory system for cloud-native databases, supporting both memory pooling and data sharing scenarios.
- We proposed a novel database recovery scheme, *PolarRecv*, based on *PolarCXLMem*, which substantially enhances recovery performance. To the best of our knowledge, this is the first work to utilize CXL memory for improving database recovery efficiency.
- We adapted *PolarCXLMem* for data sharing in a commercially deployed multi-primary database and designed a new data synchronization protocol for multi-primary databases based on *PolarCXLMem*.
- We thoroughly evaluated *PolarCXLMem* and *PolarRecv* in the widely deployed commercial cloud-native databases under various workloads.

This paper is structured as follows. First, we present the background and motivation in Section 2. Then we provide the *PolarCXLMem*'s design and implementation in Section 3. Next, we evaluate it in Section 4 and review the related work in Section 5. Finally, we conclude the paper in Section 6.

2 Background and motivation

2.1 CXL introduction

CXL is an interconnect technology for high-speed, low-latency communication between host processors and peripheral devices. CXL has evolved significantly across its three major versions, each enhancing its capabilities and flexibility. CXL 1.0 [15], introduced in 2019, laid the foundation with support for three key protocols: *CXL.io* for I/O operations, *CXL.cache* for caching semantics, and *CXL.mem* for memory access semantics. These protocols enable low-latency, high-bandwidth communication between host processors and accelerators, focusing on coherent memory access and

device discovery. CXL 2.0 [16], released in 2020, introduced memory pooling to enable multiple hosts to share memory resources and added switching capabilities to improve scalability. CXL 3.0 [17], launched in 2022, further extended these features with innovations such as multi-level switching, dynamic capacity devices, and support for fabric-based topologies, delivering greater memory scalability, improved disaggregation, and flexible memory sharing across hosts.

Hardware availability. CXL remains a nascent technology, with most products still in the prototype stage and only limited samples available. Due to the scarcity of mature CXL hardware, current research on CXL devices often faces significant limitations. Many studies either focus solely on use cases without experimental validation [10, 11, 24, 31], rely on emulation [34, 36, 53], which fails to accurately capture the true characteristics of CXL, or use internal memory expander samples based on CXL 1.0 for evaluation [6, 41, 49]. A few works [39, 62] explored memory sharing with early-stage FPGA-based hardware without a CXL switch support. These prototypes also face limitations, including lower performance and smaller capacities. For example, the FPGA-based sample used in CXL-SHM [62] supports only two nodes and provides a few hundred GBs of memory. To date, most existing works are restricted to CXL 1.0 and have limited focus on memory pooling and sharing. In contrast, our work leverages a state-of-the-art, commercial-grade CXL 2.0 switch, which supports terabyte-scale memory and allows us to explore the full potential of memory disaggregation (*e.g.*, pooling and sharing) in a real-world environment.

Compared with RDMA. RDMA has been widely adopted for memory disaggregation [12, 45, 47]. In particular, in the database domain, RDMA is extensively employed to accelerate distributed transaction processing [14, 20, 27, 28, 54–56, 61], optimize database indexing on remote memory [37, 38, 51, 68], extend local buffers [13, 46, 64], enable memory pooling and sharing [58], and accelerate message transmission [59]. While many of RDMA's features resemble those of CXL Type 3 devices, CXL offers several advantages over RDMA, and existing RDMA-based research cannot be directly adapted to CXL. The key advantages of CXL include:

- (1) Low latency: CXL connects the host to device-attached memory via PCIe, whereas RDMA requires protocol interface conversions between InfiniBand and PCIe. CXL can also translate memory store/load requests from the last-level cache into CXL flits (Flow Control UnIT), while RDMA relies on DMA to read/write memory data, making it slower [22]. *DirectCXL* reports that CXL is 8.3× faster than RDMA for reading 64 bytes of data [22].
- (2) Load/store support: CXL natively supports load/store instructions, allowing direct access to CXL memory. In contrast, with RDMA-based systems, applications must first read data from the remote memory into local memory, process it locally, and then write the updated data back to the remote memory. This additional step adds significant overhead. In a CXL-based system, the CPU can directly access and process data in the CXL memory without this intermediate step, and the CPU cache can further enhance memory access performance.
- (3) Simplicity: RDMA relies on specialized interfaces and drivers for access, introducing programming complexity. Additionally, RDMA requires developers to possess deep expertise to design high-performance systems that effectively leverage its capabilities. In contrast, CXL offers a transparent memory space for applications, significantly simplifying development and usage.
- (4) Cache coherency: CXL 3.0 supports hardware-based cache coherency across memory and devices attached to multiple hosts, unlike RDMA. This feature opens up new opportunities for memory sharing across systems.

2.2 Memory disaggregation in cloud databases

Databases typically manage physical data in storage at the granularity of pages and cache a subset of these pages in a memory buffer pool to improve performance. During transactions, the transaction engine retrieves required pages from the buffer pool and accesses the data through pointers provided by the buffer pool. The buffer pool operates transparently, allowing the transaction engine to function without awareness of its design.

Memory disaggregation has gained significant attention, particularly in the database domain [8, 13, 33, 58, 64]. Since the buffer pool constitutes the primary use of memory, databases often extend its capacity into disaggregated memory to accommodate more data, thereby reducing costly storage I/O operations. Moreover, as the buffer pool operates independently of other database components, extending it into disaggregated memory can be achieved transparently, enhancing design flexibility. This approach is widely adopted in commercial databases [13, 64]. In contrast, some academic efforts [35, 37, 38, 51, 60] explore a co-design of the transaction engine and disaggregated memory. However, these innovations have not yet been widely implemented in commercial products, requiring substantial effort to transition into production environments. Consequently, this work primarily focuses on leveraging disaggregated memory for the database's buffer pool.

In RDMA-based systems, the CPU cannot access remote memory via RDMA as transparently as it accesses local memory, requiring data to be fetched to local DRAM and processed locally. Additionally, RDMA operations are still much slower than local DRAM access, necessitating the use of a local buffer pool (LBP) to maintain performance. The LBP and remote memory are typically organized in a tiered memory structure. While RDMA access is significantly faster than traditional storage I/Os and can improve performance, this design still faces several limitations:

- **(1) Read-write amplification.** In current designs, databases store pages in disaggregated memory and transfer data at the page granularity. This approach leads to inefficiencies: even when only a small portion of data is required, the entire page must be read, and modifying a small part of a page requires writing back the full page. These operations result in significant read/write amplification, consuming substantial RDMA bandwidth—a critical resource for cloud databases. Figure 1 illustrates the impact of LBP size on database throughput and RDMA bandwidth. The evaluation was conducted on PolarDB (a widely deployed database on Alibaba Cloud) with 16 vCPUs with RDMA-based disaggregated memory [13, 64] under Sysbench point-select and read-write workloads. The LBP size was varied from 10% to 100% of the disaggregated memory. When the LBP size is set to 100%, disaggregated memory is not used, and the database operates entirely on local DRAM. This evaluation reveals that with an

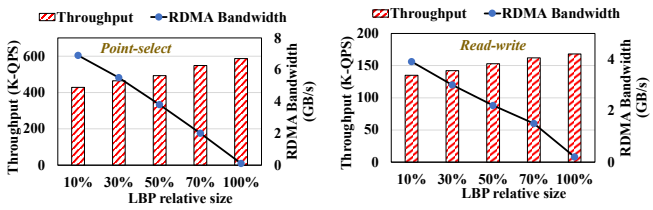

Figure 1: Impact of LBP size in RDMA-based systems.

LBP size of 10%, RDMA bandwidth utilization reaches 6.9 GB/s and 3.9 GB/s under two workloads, consuming up to 57% of a ConnectX-6 NIC's 12 GB/s bandwidth, despite utilizing only 8.3% of CPU resources on a 192 vCPU host. Increasing the LBP size to 50% reduces RDMA bandwidth usage to 3.8 GB/s and 2.2 GB/s, respectively, which, while lower, remain considerable. Additionally, this bandwidth reduction comes with a 50% memory overhead, significantly raising the total cost. These findings highlight the high RDMA bandwidth consumption inherent in such systems and reveal the trade-offs in RDMA-based designs, where increasing the LBP size can alleviate bandwidth pressures but sacrifices cost-efficiency in cloud environments. However, running the database directly on CXL memory eliminates these issues and avoids the LBP overhead.

- **(2) Inefficient recovery.** After a database crash, disaggregated memory retains some data pages. Reading these pages from disaggregated memory can reduce storage I/O, improving recovery performance, as discussed in previous works [33, 64]. However, disaggregated memory is typically only used to accelerate page I/O during recovery. The database must still manage its recovery logic and restore certain in-memory data, such as buffer pool metadata and locally buffered data, which can take considerable time. Storing metadata on disaggregated memory via RDMA is impractical due to RDMA's high latency. However, CXL's low latency opens up new possibilities for redesigning database recovery schemes using disaggregated memory.
- **(3) RDMA bottleneck.** In RDMA-based disaggregated memory architectures, the RDMA driver can become a performance bottleneck. Previous evaluations [45] show that existing IOPS-bound disaggregated applications do not scale well beyond 32 cores, hindering the effective utilization of modern many-core systems. This is due to limitations in RDMA NICs, such as implicit contention on doorbell registers and cache thrashing [45, 55]. CXL, which supports load/store memory semantics via the PCIe interface, can eliminate these bottlenecks.
- **(4) Lack of cache coherency.** The practice of sharing disaggregated memory across multiple database nodes is becoming more prevalent [33, 58]. However, due to the lack of cache coherency between nodes, databases are required to implement their own cache coherency mechanisms. For instance, in PolarDB-MP [58], after a node modifies a page in its local buffer and pushes it to disaggregated memory, it must invalidate copies in other nodes' local buffers to maintain cache coherency. This process incurs additional overhead. Fortunately, the CXL 3.0 protocol natively implements cache coherency, removing this overhead from the application layer and improving overall performance.

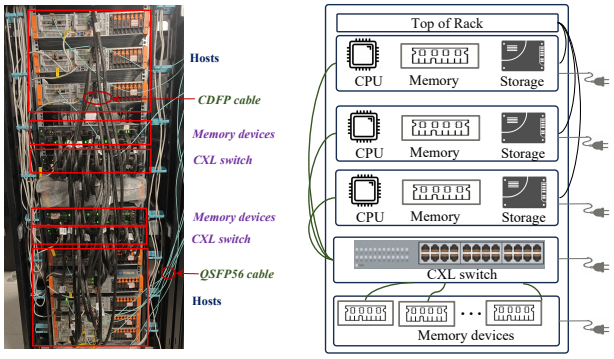

Figure 2: The physical topology of CXL-enabled cluster.

2.3 Characteristics of CXL switch

This paper employs the XConn CXL switch [50] for the evaluation, the world's first CXL 2.0 switch, supporting up to 256 lanes with a total switching capacity of 2TB/s. The left panel of Figure 2 shows our in-house built physical prototype rack, which integrates two CXL switch-enabled clusters, each connected to memory devices and hosts. The right panel illustrates the topology of a single CXL switch system, where both the CXL switch and memory devices operate with independent power supply units. In multi-socket servers, when the CXL switch is connected to one socket, CPUs on other sockets may experience higher latency when accessing CXL memory. Table 1 compares the latency of DRAM and CXL memory access, measured using the Intel MLC tool [3]. Additionally, we juxtapose the latency of directly accessing CXL memory without the switch, used in most existing works. Without the switch, the latency of CXL is comparable to remote DRAM access, aligning with prior claims [9, 34, 36, 53]. However, deploying the CXL switch introduces additional latency. For local NUMA node access, CXL latency is 3.76× that of DRAM; for remote NUMA access, it is 2.82×. Even when comparing remote DRAM access with local CXL access, CXL latency is still 2.38× higher than DRAM. As this paper focuses on memory pooling and sharing based on CXL memory, the use of the CXL switch is essential for these scenarios. All subsequent discussions in this paper are based on the CXL switch. From our evaluation, the additional latency introduced by the CXL switch proves to be negligible in cloud database scenarios.

Table 1: Access latency comparison between DRAM and CXL

	DRAM		CXL w/o switch		CXL w. switch	
	Local	Remote	Local	Remote	Local	Remote
Latency (ns)	146	231	265.2	345.9	549	651

Table 2: Data transfer latency of RDMA *vs* CXL

Size	Write latency (μs)		Read latency (μs)	
	RDMA	CXL	RDMA	CXL
64B	4.48	0.78	4.55	0.75
512B	4.69	0.84	4.79	0.85
1KB	4.77	0.88	4.91	1.07
4KB	5.06	1.02	5.58	1.86
16KB	6.12	1.68	7.13	2.46

Data transfer of RDMA *vs* CXL. Data transfers between local and remote memory in RDMA-based systems are common. Table 2 shows latency comparison between RDMA and CXL for data

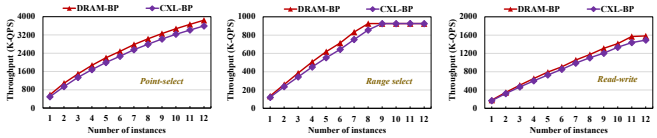

Figure 3: Performance comparison of DRAM-based vs. CXL-based buffer pool in the database.

transfers, including *writes* (local DRAM to remote memory) and *reads* (remote memory to local DRAM). For small data sizes, CXL demonstrates significant advantages over RDMA, reducing latency by 5.74× for writes and 6.07× for reads at 64B. However, as the data size increases to 16KB, CXL's latency rises more substantially than RDMA's. RDMA's latency is less sensitive to data size due to its fixed overhead, *e.g.*, network RTT, protocol handling, and NIC memory access, while CXL latency grows with data size due to limited CPU load/store buffer depth. For instance, increasing the data size from 64B to 16KB only increases RDMA latency by 36.61% and 56.70%, whereas CXL's latency increases by 1.15× for writes and 2.28× for reads. These findings motivate us to consider load/store instructions for direct memory accesses and eliminate the local buffer completely, instead of copying data from the remote memory to the local buffer. Directly accessing data on CXL memory is much faster than with RDMA, making it a feasible option to run databases directly on CXL memory. Moreover, databases often require only a portion of a page during a transaction processing, eliminating the need to transfer entire pages. Additionally, CPU caching further enhances performance when directly accessing CXL memory.

Database performance on CXL. Copying data pages from CXL memory to local DRAM introduces significant latency. Although CXL latency is higher than DRAM, it is much faster than RDMA, and CPU caching mitigates the latency impact. Additionally, database buffer pool operations are more sensitive to bandwidth than latency. These factors motivate our approach to directly running the database on CXL memory. To further prove this, we implement a CXL-based buffer pool (CXL-BP) in PolarDB to enable direct CXL memory access and bypass local DRAM buffering, and compare its performance with DRAM-based buffer pool (DRAM-BP), as shown in Figure 3. In our test, each physical host has 192 vCPUs and supports up to 12 database instances, with each instance configured with 16 vCPUs. We varied the number of instances from 1 to 12 to evaluate whether CXL-BP could fully utilize the 192 vCPUs and achieve throughput comparable to DRAM-BP. We tested three Sysbench workloads: point-select, range-select, and read-write. Detailed configurations are described in Section 4. Figure 3 shows that in the point-select workload, CXL-BP achieves comparable performance to DRAM with no signs of bottlenecks, showing only a 7% throughput difference at the maximum scale of 12 instances. In the range-select workload, CXL demonstrates similar performance (approximately 10% lower) for 1 to 8 instances. Beyond 9 instances, the network becomes the bottleneck due to the volume of query results returned to the client, resulting in similar performance for both CXL-BP and DRAM-BP. For the read-write workload, the throughput difference remains within 7% for up to 11 instances. With more than 11 instances, WAL persistency becomes the system bottleneck, and both DRAM-BP and CXL-BP exhibit similar performance. In summary, using CXL as the buffer pool in databases results in only minor performance drops or none in some scenarios. These results encourage us to rethink disaggregated memory designs with CXL in databases, moving away from the tiered memory structure commonly used in RDMA-based solutions [64].

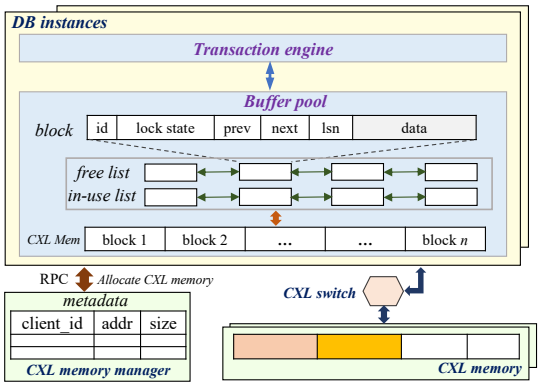

Figure 4: The architecture of *PolarCXLMem*.

3 Design and Implementation

3.1 *PolarCXLMem*

Overview. *PolarCXLMem* shares a similar architecture with RDMA-based disaggregated memory. The key difference is that the database constructs its buffer pool, including both data pages and metadata, directly on CXL memory. Local DRAM is reserved for variables and other transaction engine-related data structures. As shown in Figure 4, all hosts running the database instance, along with the CXL memory devices, are connected to the CXL switch. The database instance can transparently access CXL memory using native load/store instructions, as if it were accessing local DRAM.

CXL Memory allocation. Although the CXL 2.0 switch natively supports memory pooling, the CXL 2.0 driver is not yet fully upstreamed. Therefore, we utilize the *EFI_MEMORY_SP* attribute [2] set by the BIOS. On each host connected to the CXL switch, we configure the CXL-attached memory as a *dax* device in device direct access (*devdax*) mode using *daxctl*, enabling easy access via *mmap*. In this setup, we designed a CXL memory manager to support multi-tenancy, preventing multiple nodes from accessing the same memory region. The manager allocates memory for each node, ensuring that no two nodes access overlapping CXL memory. When a node requests CXL memory, it communicates with the manager via RPC, specifying the required memory size. The manager then returns an offset for the allocated memory, allowing the node to use memory starting from that offset. Since the CXL memory for the buffer pool is only allocated once during database startup, the memory allocation overhead has no impact during runtime.

CXL-based buffer pool. During database startup, the database instance calculates the memory required for the buffer pool and allocates CXL memory from the CXL memory manager. After receiving the offset, the database instance uses *mmap* to map the CXL *dax* device at the specified offset. The allocated CXL memory is logically divided into blocks, with each block storing a database page and its corresponding metadata. These blocks are managed through two linked lists: a free list, representing blocks that are available, and an in-use list, representing blocks already in use for

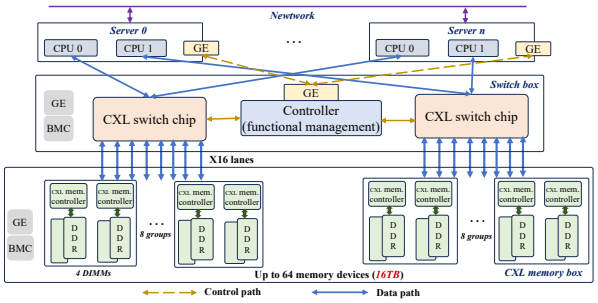

Figure 5: Physical topology of *PolarCXLMem* deployment.

database pages. When the transaction engine requests a page, if the page is in the in-use list, the data pointer is returned directly to the transaction engine. If not, a block from the free list is assigned, an I/O operation is performed to load the page into memory, and the data pointer is returned. The transaction engine can then operate on the data pointer without needing to know whether it points to local DRAM or CXL memory. This design minimally impacts the transaction engine, requiring only a few modifications during memory allocation, thereby preserving system stability and compatibility.

Avoiding Tiered Memory. In *PolarCXLMem*, we eliminate the tiered memory structure commonly used in many RDMA-based disaggregated memory systems [25, 46, 64]. Although CXL memory is slower than local DRAM, it is already fast enough for the buffer pool. In Section 2.3, we investigated the impact of memory speed on overall performance and found that placing the buffer pool directly on CXL memory provides nearly the same performance as using local DRAM. Therefore, a tiered memory structure is unnecessary with CXL-based disaggregated memory, which also simplifies the system design, minimizes modifications to the existing architecture, and enhances system stability. Moreover, tiered memory designs transfer data at the page granularity between memory levels. As a result, accessing small amounts of data can trigger the movement of entire pages, leading to read/write amplification and wasting bandwidth. Additionally, maintaining an extra local DRAM-based buffer pool incurs higher costs.

Physical topology. Figure 5 depicts the physical topology of a *PolarCXLMem* deployment. This setup includes two CXL switches housed in a single switch box. A controller connects to these switches, enabling servers to send control messages via Ethernet for configuration and management. The switches are connected via CXL x16 lanes to memory devices organized in a CXL memory box, which supports up to 16TB of memory, forming a pool of the same capacity. Each CXL switch, along with its connected memory, constitutes an independent memory pool that can be accessed by servers. This example demonstrates a deployment with two CXL-based memory pools, showcasing the scalability and flexibility of the system.

3.2 Instant recovery on *PolarCXLMem*

The CXL switch has an independent power supply unit (PSU). When a host goes down, the data in CXL memory remains intact. We can utilize the CXL-based disaggregated memory to achieve the instant recovery since the main purpose of the database crash recovery is to recover the buffered data pages. Thus, we propose the instant recovery scheme, *PolarRecv*, based on *PolarCXLMem*.

Challenges. Before diving into our design, we first review the process of database recovery after a crash, why RDMA-based disaggregated memory cannot support instant recovery, and the associated challenges. In most databases, each update generates corresponding redo logs. Upon committing a transaction, the database writes these redo logs to persistent storage without flushing the modified page itself. After a crash, the modified pages in the buffer are lost. During recovery, the system scans redo logs from the latest checkpoint onward, reading corresponding pages from storage and applying logs to rebuild pages in the buffer. This process ensures that pages reflect all changes made before the crash, allowing the database to resume serving applications. Meanwhile, the rollback of uncommitted transactions can occur simultaneously with application requests. When using the RDMA-based disaggregated memory, the latest versions of pages are not retained in disaggregated memory, as pages are updated locally in the buffer. Therefore, after a crash, the recovery process must scan all redo logs, retrieve pages from disaggregated memory or storage, and apply these logs [33, 64]. Additionally, since the buffer is not immediately restored, it requires a warm-up period after recovery, meaning applications need additional time to reach pre-crash peak throughput.

In contrast, *PolarCXLMem* allows the database to operate directly on CXL memory. As a result, after a crash, CXL memory retains all the latest updates, enabling us to initialize the buffer pool structure directly from CXL memory. However, because other in-memory data structures, such as thread contexts, are lost during a crash, the database cannot simply restart based on CXL memory alone. There are additional issues that must be addressed to enable instant recovery: (1) The LRU list may be inconsistent after a crash if the crash occurred during the movement of an LRU node. In this case, the system needs to detect this issue and rebuild the LRU list during recovery. (2) If a crash happens during structure modification operations (SMOs) in the B-tree, such as page splitting or merging, the B-tree structure may be inconsistent. The system must restore the B-tree to a consistent state during recovery. (3) When a crash occurs during a page update, the page may be left in a partial state, preventing reliable database operation. Page atomicity must be guaranteed. (4) Since the redo log buffer still uses local DRAM, any logs not flushed to storage at the time of the crash will be lost. This could result in an updated page being present after recovery without corresponding redo logs, which violates the ARIES-style logging scheme. To maintain consistency, it's essential to avoid situations where there are 'too new' pages without associated logs.

***PolarRecv* design.** *PolarRecv* is built on *PolarCXLMem*, where the database directly utilizes disaggregated CXL memory as the buffer pool. To recover the buffer pool structure after a crash, we also store metadata for each page in CXL memory. We organize each page's data and metadata within a structure called a *block*, as shown in Figure 4. The *id* field represents the page ID, and *lock_state* indicates whether the page is currently locked for updating. The *prev* and *next* fields are used for the double-linked list managed by the LRU policy, while *lsn* stores the latest log sequence number corresponding to the page.

In most databases, a page's write or read lock must be acquired before it is updated or read, so we store the lock state in CXL memory. During recovery, we scan each page's lock state. If a page

is write-locked, it may have been in the middle of an update when the crash occurred, potentially leaving it in a partial state. In such cases, we must recover the page from the redo logs rather than using the potentially incomplete page directly.

During a B-tree SMO, the process is protected by a mini-transaction, with the corresponding page locked using a two-phase locking policy. In this case, the locks on related pages involved in the SMO are only released upon the completion of the mini-transaction. Furthermore, redo logs are typically flushed to storage only after the mini-transaction is committed. During recovery, if a page is found to be write-locked, we apply the redo logs to restore it rather than directly using the version in CXL memory. Therefore, if a crash occurs during an SMO, we can still identify the related pages involved in the incomplete SMO's mini-transaction, recover them from the redo logs, and maintain B-tree consistency.

In the recovery process, *PolarCXLMem* first retrieves the maximum LSN from the persistent redo logs. When scanning pages in CXL memory, it checks both the lock state and the LSN field of each page. If a page's LSN is greater than the maximum LSN in the persistent redo logs, *PolarCXLMem* rebuilds the page by applying the redo logs instead of using the version in CXL memory. This approach ensures that we do not use a version of the page that lacks corresponding redo logs.

To ensure LRU consistency, we use a lock state in CXL memory to indicate any modifications. Changes to the LRU structure are protected by a *mutex* lock, with the lock state also stored in CXL memory. When recovering, this lock state is checked to determine if the LRU structure was being modified at the time of the crash. If it was, the LRU list is rebuilt; otherwise, *PolarCXLMem* can directly use the existing LRU list.

3.3 CXL-based data sharing on *PolarCXLMem*

Data sharing has been a popular trend in cloud-native multi-primary databases [19, 33, 58]. Using *PolarCXLMem* for data sharing opens new opportunities to enhance performance compared to RDMA-based solutions [58]. Since CXL 3.0 switches with inherent cache coherency are not yet available, we design a new cache coherency protocol for the CXL 2.0-based disaggregated memory.

Overview. Figure 6 illustrates the design of data sharing in a multi-primary database based on *PolarCXLMem*. Similar to PolarDB-MP [58], we employ a buffer fusion server to manage the metadata of the distributed buffer pool (DBP). The buffer fusion server allocates memory from *PolarCXLMem*, the disaggregated CXL memory. The CXL memory is organized at the page level, with each page storing a single unit of data. Pages are managed using an LRU policy. Initially, all pages are stored in a free list. During runtime, pages are allocated from the free list and moved to the in-use list. A background thread dynamically moves the least recently used pages from the in-use list back to the free list to free space for new pages. On the database side, each buffer pool maintains a set of pages but stores only the CXL memory addresses of the pages rather than the pages themselves. For each page, the system also tracks two additional fields: *invalid* and *removal*. The *invalid* flag ensures cache coherency by indicating whether a page has been modified by another node, requiring the current node to invalidate its CPU cache. The *removal* flag manages the LRU policy on the

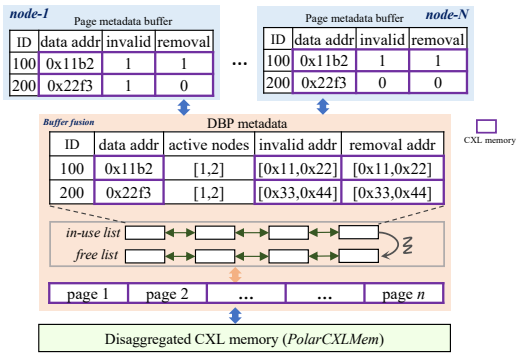

Figure 6: Data sharing based on *PolarCXLMem*.

buffer fusion side. If the buffer fusion server moves a page from the in-use list to the free list, it resets the removal flag across all nodes.

Workflow. On the database side, a hash table is initially allocated to manage the page metadata buffer. The hash table uses the page ID as the key, while the values consist of the corresponding metadata, including *data_addr* (the page's CXL address allocated by the buffer fusion server), *invalid* and *removal*. The *invalid* and *removal* fields are stored in CXL memory and can be updated by the buffer fusion server when the page state changes.

Initially, these metadata records are free and stored in a free list. When the database requests a new page that cannot be found in the local page metadata buffer, it allocates a free metadata record from the free list and sends an RPC to the buffer fusion server, providing its *invalid* and *removal* CXL addresses. If the requested page already exists in the buffer fusion server's in-use list, the server returns its CXL address to the database. Otherwise, the server allocates a page from its free list, updates the page's metadata, adds the requesting node to the active list, and records the corresponding *invalid* and *removal* fields. Once the database node receives the page's CXL memory address from the buffer fusion server, it stores the address in its local page metadata buffer.

If the requested page is found in the local page metadata buffer, the database first checks the *removal* flag. If the flag is set, it indicates that the page's CXL address has been recycled by the buffer fusion server. In this case, the database cannot use the page directly and must request a new address from the buffer fusion server via RPC. Afterward, the *invalid* flag is checked to determine whether the page has been modified by another node. If the invalid flag is set, the database invalidates the CPU cache for this page to ensure consistency. Subsequent accesses to the page will read directly from the CXL memory device to obtain the latest version.

Cache coherency. Since the CXL 2.0 switch lacks inherent cache coherency, a node updating a data page in CXL memory cannot invalidate the same page in another node's CPU cache. Additionally, updates made by a node may remain in its CPU cache and be flushed to CXL memory in the background, preventing immediate visibility of the changes to other nodes. To address this, we implement cache coherency at the database level. In existing cloud-native multi-primary databases [19, 33, 58], concurrent access to data pages is managed through distributed page locks. Before reading from or writing to a page, a node must acquire the page's read or write lock. We integrate our cache coherency scheme with this page locking mechanism. When a node updates a page, it must hold the

write lock for that page, preventing any other node from reading or writing to it until the lock is released. Upon releasing the write lock, the node performs a *clflush* operation on the page to flush the modifications from the CPU cache to CXL memory, ensuring the CXL memory contains the latest version of the page. Additionally, when releasing the page lock, the buffer fusion server sets the *invalid* flag for all nodes where the page is active. This flag update is executed as a memory store operation on CXL memory, which generally completes within a few hundred nanoseconds. If the flag is set, it indicates that the page has been modified by another node. To ensure cache coherency, the node must invalidate its CPU cache for the page to avoid accessing outdated data. Since concurrent nodes cannot update the page without holding the lock, the cache lines of the page should already be clean. Therefore, a *clflush* operation can be performed to invalidate the cache lines of the page. Subsequent accesses will then fetch the data directly from CXL memory, ensuring the node retrieves the latest version.

Background recycle. The CXL memory allocated for the DBP is limited, so a background thread on the buffer fusion server is enabled to recycle the least recently used pages, moving them from the in-use list to the free list. Before recycling a page, the buffer fusion server must ensure it is not in use by any database node by acquiring an exclusive lock on the page. Once a page is recycled, the buffer fusion server sets the *removal* flag for all nodes where the page is active. This flag update is performed through a single memory store operation on CXL memory, which typically takes only a few hundred nanoseconds. When a node detects that a page's removal flag is set, it recognizes that the page has been removed by the buffer fusion server and retrieves the page's new CXL address via RPC. Additionally, on the database side, a background thread periodically scans the page metadata buffer to recycle metadata entries whose removal flag is set.

Benefits. In this CXL-based data sharing solution, database nodes can directly operate on CXL memory without needing to read entire pages into local buffers for processing, as required in existing RDMA-based solutions [58]. Our approach eliminates read/write amplification, thereby significantly saving bandwidth, as thoroughly discussed in previous sections. What's more, when a page is updated, only a *clflush* operation is needed to synchronize the modified cache lines to CXL memory. Unlike RDMA-based solutions, which transfer the entire page to the distributed buffer pool even when only a small part of the page is updated, the CXL-based approach synchronizes only the modified data, avoiding redundant writes and reducing bandwidth usage and synchronization overhead.

4 Evaluation

4.1 Experimental setup

In this section, we evaluate *PolarCXLMem* within the commercial cloud-native database, PolarDB [32], a widely deployed cloud-native OLTP database on Alibaba Cloud. For the data-sharing scenario, we integrate *PolarCXLMem* into the multi-primary version of PolarDB, known as PolarDB-MP [58].

Test platform. For the CXL devices, we use the XConn XC50256, the world's first CXL 2.0 switch, as introduced in Section 2.3. In the experiment setup, the switch connects 8 DDR5 memory modules, totaling 2TB. To enable shared memory access for all hosts connected to the CXL 2.0 switch, we configure the CXL-attached memory on each host as a *dax* device in device direct access (*devdax*) mode using *daxctl*, allowing straightforward memory access through mmap. Each physical machine is equipped with two Intel Xeon Platinum 8575C CPUs (2.8GHz) and runs CentOS-7. For comparison with RDMA-based disaggregated memory, we connected these machines via a 100Gbps Mellanox ConnectX-6 network, which is also the standard configuration for PolarDB deployments on Alibaba Cloud.

Baselines. Since RDMA is widely adopted for disaggregated memory, it serves as the natural baseline for our evaluation. PolarDB already supports RDMA-based disaggregated memory [13, 58], making it a suitable baseline for comparison. Other RDMA-based disaggregated memory systems have a similar architecture and thus exhibit similar performance trends when compared to *PolarCXLMem*. For data sharing scenarios, since we implement *PolarCXLMem* in PolarDB-MP [58], PolarDB-MP with RDMA-based memory sharing serves as the baseline for these evaluations.

Workloads. We evaluate *PolarCXLMem* using three standard OLTP benchmarks, Sysbench [29], TPC-C [18] and TATP [43], and adopt their default configurations. The evaluation focuses on memory-bound scenarios to investigate the impact of disaggregated memory design in cloud-native databases. To ensure this focus, we configure a large disaggregated memory to hold the entire dataset. Storage I/O-bound scenarios are excluded from the discussion, as storage I/O becomes the primary bottleneck in such cases, rendering the design of disaggregated memory less impactful on overall performance.

4.2 *PolarCXLMem* pooling performance

RDMA-based disaggregated memory typically has significant read and write amplification, leading to excessive RDMA bandwidth consumption, as discussed in Section 2.2. In a cloud environment, where each host has limited RDMA bandwidth but often runs multiple database instances, RDMA bandwidth becomes a potential bottleneck for scalability. In this subsection, we test scenarios where multiple database instances on a single host access remote memory, thoroughly comparing the performance of *PolarCXLMem* and RDMA-based disaggregated memory. Each physical host in our test environment is equipped with 192 vCPUs, and each database instance is configured with 16 vCPUs, allowing a total of 12 instances per host. For the RDMA-based setup, the local buffer size is set to 30% of the disaggregated memory size, a commonly used proportion for balancing latency and memory costs, while *PolarCXLMem* operates without a local buffer, reducing memory costs by 30%. To emulate a typical cloud environment that aims to maximize resource utilization, we run diverse workloads on the 12 instances to evaluate if the RDMA-based solution becomes a bottleneck and whether *PolarCXLMem* can fully utilize the 192-vCPU machines.

Point-select. We first run Sysbench's point-select workload with 48 threads to simulate high-concurrency scenarios, varying the number of database instances on the host from 1 to 12. We measure the total throughput across all instances, average latency, and RDMA/CXL bandwidth usage, as illustrated in Figure 7. In RDMA-based systems, querying a single record (a few hundred bytes in this workload) may induce the transfer of an entire page (16KB)

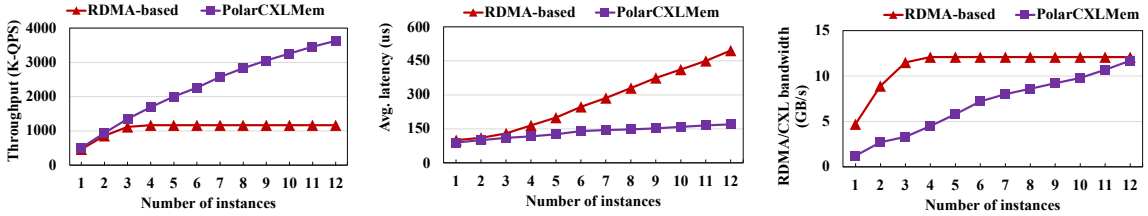

Figure 7: Comparison with RDMA-based disaggregated memory using Sysbench point-select workload.

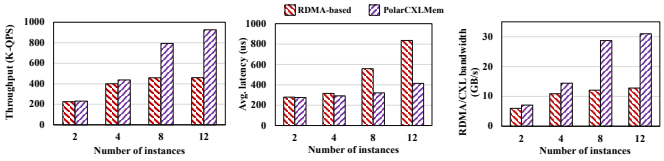

Figure 8: Performance of Sysbench range-select workload.

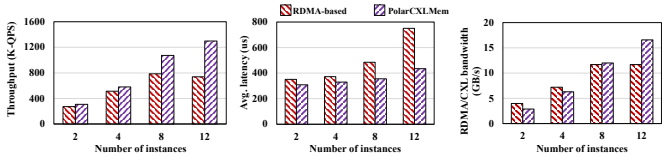

Figure 9: Performance of Sysbench read-write workload.

from the disaggregated memory, as discussed in Section 2.2. This results in significant read amplification, making RDMA bandwidth the primary bottleneck. In contrast, *PolarCXLMem* processes data directly on CXL memory without moving it to local memory, so the CXL bandwidth in *PolarCXLMem* is only used for essential data processing, preventing CXL bandwidth from becoming a bottleneck in the point-select workload.

The left panel of Figure 7 shows that as the number of instances increases from 1 to 12, *PolarCXLMem* 's throughput scales proportionally, reaching 3.6M QPS (queries per second). However, the RDMA-based system's throughput saturates at 3 instances, reaching only 1.1M QPS due to bandwidth saturation. Latency trends in the middle panel of Figure 7 align with throughput observations: after reaching 3 instances, as the number of instances increases, the RDMA-based system experiences a linear rise in latency due to bandwidth saturation, while *PolarCXLMem* maintains stable latency, with only a marginal rise, even as instances scale to 12. This performance disparity stems from high read amplification and bandwidth saturation in the RDMA-based system, which hits its limit at 3 instances, whereas *PolarCXLMem* avoids read amplification and has no CXL bandwidth bottleneck. The right panel of Figure 7 further confirms this, showing that RDMA bandwidth is fully utilized at 3 instances, reaching 11 GB/s, while *PolarCXLMem* 's CXL bandwidth remains lower. In particular, with only 1 instance, neither system is saturated; RDMA bandwidth measures 4.7 GB/s, which is 4× that of *PolarCXLMem*, indicating a fourfold read amplification. Notably, the RDMA-based system also requires additional DRAM bandwidth for data processing.

Range-select. We then run Sysbench's range-select workload with 32 threads per instance. Range-select queries typically retrieve a set of consecutive records, which are often stored adjacently. In RDMA-based systems, when a page is read remotely, all records within that page may be relevant to the current query. As a result, range-select workloads do not incur the same high level of read amplification as point-select workloads. However, range-select queries still require a high bandwidth for data reading, as they retrieve multiple records per query, making bandwidth a potential bottleneck. As seen in Figure 8, the RDMA-based system reaches saturation at 4 instances, with RDMA bandwidth peaking at approximately 11 GB/s. Beyond this point, adding more instances does not increase throughput, and latency rises linearly. In contrast, *PolarCXLMem* benefits from CXL's higher bandwidth, allowing its throughput to continue increasing as more instances are added.

Read-write. Finally, we run Sysbench's read-write workload, which includes point-select, range-select, update, delete, and insert operations, using 48 threads per instance. Operations like updating or deleting an existing record, or inserting a new record, require reading the target page before performing the operation. Thus, even in this mixed read-write scenario, the RDMA-based system encounters bandwidth limitations. As shown in Figure 9, the RDMA-based system saturates at 8 instances. Beyond this point, adding more instances does not improve the total throughput due to RDMA bandwidth constraints. However, *PolarCXLMem* 's throughput continues to increase, benefiting from its lower bandwidth requirements. With a single instance, the RDMA bandwidth is approximately 40% higher than the CXL bandwidth, indicating significant read amplification in the RDMA-based system during this workload. It is important to note that the RDMA bandwidth shown reflects only data transfer bandwidth. Additional DRAM bandwidth used for data processing in the RDMA-based system is not shown, as it is challenging to measure directly. Conversely, *PolarCXLMem* 's CXL bandwidth accounts for all data transfer and processing demands. Despite this, the RDMA-based system's bandwidth consumption still exceeds that of *PolarCXLMem*, underscoring the substantial read/write amplification inherent in RDMA-based designs.

4.3 Recovery performance of *PolarRecv*

In this subsection, we evaluate the recovery performance of *PolarRecv*, which is built on *PolarCXLMem*. For this test, we run three Sysbench workloads, read-only, read-write and write-only, and randomly kill the database process to simulate a crash scenario. Figure 10 presents the recovery performance across different schemes and workloads. The *vanilla* scheme represents the commonly used approach that reads data pages from storage and applies redo logs to recover lost data, while *RDMA-based* scheme refers to the recent solutions that retrieve data from RDMA-based disaggregated memory, as widely adopted in existing RDMA-based systems [58, 64]. To ensure a fair comparison, we adjust the workload pressure to make

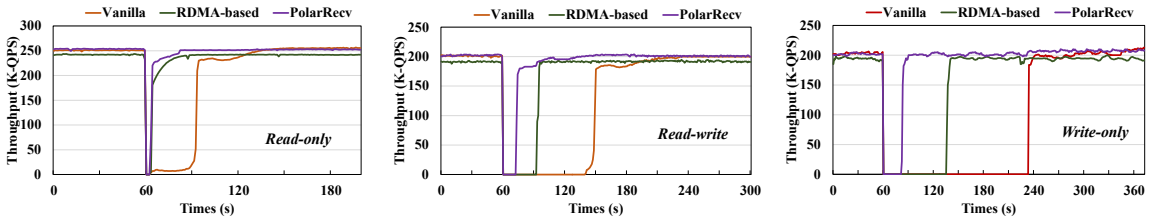

Figure 10: Comparison of recovery performance.

all three systems achieve similar throughput, resulting in equivalent redo log sizes and similar amounts of updated data pages. This setup allows us to compare recovery performance accurately, as recovery time is closely tied to both the size of the redo logs and the volume of lost data pages that need to be restored.

The left panel of Figure 10 depicts the recovery performance under the read-only workloads. At the 60-second mark, the database process was killed and immediately restarted. All three systems recovered within 2 seconds, as no data recovery was required; only in-memory data structures needed reinitialization. However, after startup, *PolarRecv* quickly regained 90% of its pre-crash throughput, whereas RDMA-based and *vanilla* schemes required 10 and 30 seconds, respectively, to warm up to comparable performance levels. *PolarRecv* demonstrated a 5× and 15× speedup in warm-up time compared to the RDMA-based and *vanilla* schemes, respectively. This advantage is attributed to *PolarRecv*'s ability to directly operate on disaggregated CXL memory. Following a crash, it continues operating on CXL memory without losing buffered data. In contrast, the RDMA-based and *vanilla* schemes lose all buffered data during a crash. Consequently, upon recovery, these systems start with empty buffers and must retrieve data from storage or disaggregated memory, resulting in considerable delays.

The middle panel of Figure 10 shows the recovery performance under read-write workloads. The database process was killed and restarted at the 60-second mark. Unlike the read-only workload, the read-write workload involves a certain ratio of writes, resulting in the loss of updated data upon crash. The lost data must be restored during recovery. While *PolarRecv* can directly reuse the buffered data from *PolarCXLMem*, partial pages that were in the middle of updates still need to be recovered using redo logs, which results in a recovery time of 8 seconds, slightly longer than that of the read-only workload. In contrast, RDMA-based and *vanilla* schemes require 33 seconds and 110 seconds, respectively, to complete recovery. These schemes must scan all redo logs, load the corresponding pages into the local buffer, and apply the logs, significantly increasing recovery time. *PolarRecv* demonstrates a 4.13× and 13.75× speedup in recovery time compared to the RDMA-based and *vanilla* schemes, respectively. After recovery, both *vanilla* and RDMA-based schemes quickly achieve their pre-crash throughput. Their warm-up time is shorter than in read-only workloads because some data is already loaded into the buffer pool during the recovery process.

The recovery performance of the write-only workload is shown in the right panel of Figure 10. The trend for the write-only workload is similar to that of the read-write workload but with more significant recovery demands, as the write-only workload involves more writes, leading to greater data loss after a crash and more data must be restored. The RDMA-based and *vanilla* schemes take 73 seconds and 173 seconds, respectively, to complete the recovery, significantly longer than their recovery times for the read-write workload. However, *PolarRecv* requires only 15 seconds to finish the recovery, achieving a 4.87× and 11.53× speedup compared to the RDMA-based and *vanilla* schemes, respectively.

4.4 Data sharing performance

Finally, we evaluate the performance of *PolarCXLMem* in data-sharing scenarios. We integrate *PolarCXLMem* into PolarDB-MP [58], a multi-primary cloud-native database at Alibaba Cloud. In this subsection, we compare the performance of PolarDB-MP using *PolarCXLMem* with the native PolarDB-MP utilizing RDMA for data sharing. The evaluation is conducted on clusters with 8, 12, and 15 nodes, each configured with 16 vCPUs. For *PolarCXLMem*-based PolarDB-MP, all nodes operate directly on shared CXL memory without local buffer. In contrast, for RDMA-based PolarDB-MP, each node maintains a local buffer sized at 30% of each node's accessed dataset size by default. In the 8-node cluster, the total memory overhead of RDMA-based PolarDB-MP is 1.53× that of *PolarCXLMem*-based PolarDB-MP. This overhead increases as the number of nodes grows.

We adapted Sysbench to evaluate performance under varying degrees of data sharing across nodes, following the configuration methods from prior studies [19, 58]. In an N-node cluster setup, tables were divided into N+1 groups. The first N groups were designated as private, with each node exclusively accessing the tables within its assigned group. The final group was shared and accessible by all nodes. The degree of sharing was controlled by a specified percentage X, where X% of queries targeted the shared tables, while the rest were directed to the private tables of each node.

Point-update. We begin our evaluation with Sysbench's point-update workload, where each transaction consists of 10 point-update queries. We vary the percentage of shared data from 0% to 100% and measure the total throughput and average latency, as shown in Figure 11. In the left panel of Figure 11, the primary vertical axis (left) represents the throughput of RDMA-based and *PolarCXLMem*-based PolarDB-MP, displayed as distinct bar groups. The secondary vertical axis (right) corresponds to a line plot that shows the relative improvement of *PolarCXLMem* over RDMA, providing a direct performance comparison

At 0% data sharing, each node accesses only its private data, and the disaggregated memory functions solely as a memory pool without data sharing. In this scenario, *PolarCXLMem* achieves a 33% throughput improvement over RDMA, primarily due to its ability to eliminate read/write amplification and extra overhead, as discussed in Section 4.2. As the percentage of shared data increases, the throughput of both systems decreases due to the data

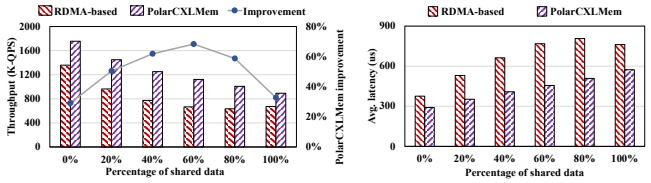

Figure 11: Performance of Sysbench point-update workload.

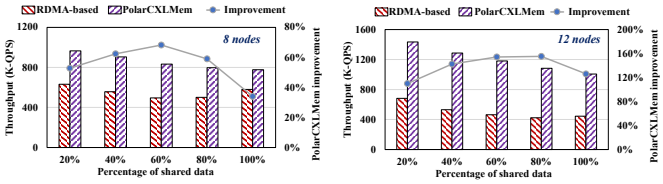

Figure 12: Performance of Sysbench read-write workloads.

contention; however, *PolarCXLMem* 's relative improvement over RDMA grows, benefiting from its efficient data-sharing scheme. In the RDMA-based system, shared data access incurs additional RDMA bandwidth usage, which competes with private data access. RDMA bandwidth becomes a significant bottleneck. Additionally, in RDMA-based PolarDB-MP, releasing a page lock requires flushing the modified page to shared memory. Even a small modification triggers a full-page flush, prolonging the lock release time. This delay prevents other nodes from acquiring the lock promptly, further degrading performance. As the shared data percentage increases from 0% to 40%, *PolarCXLMem* 's advantage grows, reaching a 62% throughput improvement at 40% data sharing. However, as shared data increases beyond 40%, lock contention intensifies. High contention leads to threads transitioning into sleep states, frequent thread context switches, and increased overhead. This contention becomes a new bottleneck as the shared data percentage rises. Consequently, from 60% to 100% shared data, *PolarCXLMem* 's relative improvement gradually declines. Despite heavy lock contention at 100% data sharing, *PolarCXLMem* still achieves a 27% throughput improvement over RDMA. The right panel of Figure 11 illustrates latency, following a trend similar to throughput. As the shared data increases up to 40%, *PolarCXLMem* achieves greater latency reduction. However, beyond 40% shared data, the latency reduction gradually diminishes with further increases in shared data.

Read-write. We then run the Sysbench read-write workload with varying percentage of shared data in setups with 8 and 12 nodes, as shown in Figure 12. Since latency follows a similar trend to throughput, we omit the latency results here to save space. The 0% shared data case, representing pooling scenarios, is excluded here as it was discussed earlier. The read-write workload exhibits a pattern similar to the point-update workload. Between 20% and 60% shared data, as the percentage of shared data increases, *PolarCXLMem* shows growing improvement, leveraging its efficient data synchronization mechanism. *PolarCXLMem* achieves its peak improvement of 68.2% in the 8-node cluster and 154.4% in the 12-node cluster at 60% shared data. The higher improvement in the 12-node setup is attributed to the higher synchronization demands compared to the 8-node setup, where *PolarCXLMem* effectively handles intensive data sharing. However, as data contention becomes severe (e.g., at 100% shared data), page-locking bottlenecks begin to limit system

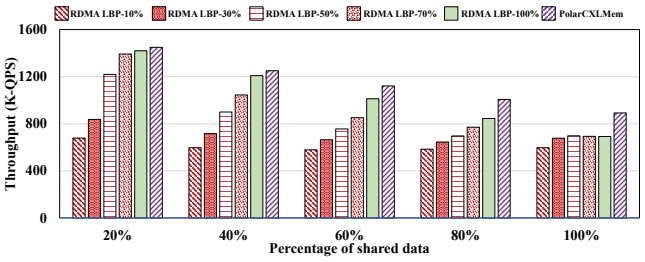

Figure 13: Breakdown analysis of *PolarCXLMem* with Sysbench point-update workload.

throughput, reducing *PolarCXLMem* 's relative advantage. Despite this, *PolarCXLMem* still provides significant throughput improvements of 34.01% and 126.09% in the 8-node and 12-node clusters, respectively, even at 100% shared data.

Breakdown analysis. To further investigate *PolarCXLMem*'s advantages in data sharing, we configured RDMA-based PolarDB-MP with varying local buffer pool (LBP) sizes, ranging from 10% to 100% of each node's accessed dataset. Particularly, LBP-100% represents a setup where the local buffer can hold a node's entire dataset, meaning all communication overhead between database nodes and disaggregated memory is due solely to data synchronization. We conducted tests using the Sysbench point-update workload on an 8-node cluster, with the results shown in Figure 13. At 20% data sharing, where data synchronization is relatively light, RDMA-based systems exhibit greater sensitivity to LBP size. *PolarCXLMem* achieves 2.14× the throughput of RDMA LBP-10%. When the LBP size increases to 70%, RDMA achieves 94% of *PolarCXLMem*'s throughput but at the cost of significantly higher memory overhead, 2.24× that of *PolarCXLMem*. As the percentage of shared data increases, the need for data synchronization grows, making *PolarCXLMem* 's advantages more apparent. Although overall throughput decreases due to data contention, *PolarCXLMem* 's relative improvement continues to rise. Furthermore, as data synchronization demands increase, the impact of LBP size on RDMA performance diminishes. Notably, even with LBP-100% in RDMA, *PolarCXLMem* maintains its performance edge. For instance, at 40% shared data, PolarCXLMem improves throughput by 16.8% to 104.58% compared to RDMA configurations with LBP sizes ranging from 10% to 70%. Especially, at 100% shared data, all RDMA configurations converge to similar performance levels, constrained by lock contention and synchronization overhead. However, *PolarCXLMem* still delivers a 42.22% and 22.48% throughput improvement compared to RDMA LBP-10% and LBP-100%, respectively, demonstrating its superior efficiency and minimal memory overhead in shared data scenarios.

TPC-C and TATP. Finally, we run the TPC-C and TATP workloads on a 15-node cluster, with each node configured with 16 vCPUs. These workloads are inherently well-partitioned, with minimal data sharing. In TPC-C, only about 10% of transactions involve cross-warehouse operations, requiring limited data synchronization. In TATP, there is no data sharing at all. Consequently, the performance improvement of *PolarCXLMem* primarily stems from the same reasons as those in the memory pooling scenarios, as discussed in Section 4.2. For this test, we configured the local buffer pool (LBP) size in RDMA-based PolarDB-MP to 10% and 30% of the

node's accessing dataset, while *PolarCXLMem* continued to operate without an LBP. The disaggregated memory is configured large enough to store the whole dataset since we focus on the performance of disaggregated memory, avoiding the storage I/O impact. This configuration results in significantly higher memory overhead for RDMA-based PolarDB-MP compared to *PolarCXLMem*. Table 3 presents the throughput, latency, and relative total memory overhead (normalized to *PolarCXLMem*) for both workloads.

Table 3: Performance of TPC-C and TATP workloads

		RDMA 10% LBP	RDMA 30% LBP	*PolarCXLMem*
TPC-C	TpmC (M)	1.11	1.65	1.92
	P95. latency (ms)	44.18	29.34	25.32
	Memory overhead	1.1×	1.3×	1×
TATP	QPS (M)	2.35	2.77	3.61
	Avg. latency (ms)	1.27	1.07	0.82
	Memory overhead	1.1×	1.3×	1×

For the TPC-C workload, under the RDMA-10%LBP configuration, the LBP hit ratio is about 93%. However, LBP misses lead to additional overhead from flushing modified pages to or reading required pages from disaggregated memory. This results in significant write/read amplification and excessive RDMA bandwidth consumption, ultimately degrading performance. In this configuration, *PolarCXLMem* achieves a 72.3% improvement in throughput while reducing memory overhead by 10% compared to RDMA-10%LBP. Increasing the LBP size to 30% raises the LBP hit ratio to 97.3%, narrowing *PolarCXLMem* 's relative improvement to 16.36%. However, the 30% LBP configuration incurs a 30% memory overhead compared to *PolarCXLMem*. For the TATP workload, which involves no data sharing, the improvement is less pronounced than in TPC-C. *PolarCXLMem* achieves a 53.6% throughput improvement over RDMA-10%LBP. Even with a 30% LBP, which incurs a 30% memory overhead, *PolarCXLMem* still improves throughput by 30.3%.

To summarize, *PolarCXLMem* demonstrates notable advantages in both performance and cost efficiency. By significantly reducing write/read amplification, it saves bandwidth and boosts throughput. Additionally, *PolarCXLMem* operates without a local buffer pool, leading to substantially lower memory overhead and eliminating RDMA-related overhead compared to RDMA-based systems. These findings position *PolarCXLMem* as a cost-effective solution for cloud-native databases, delivering substantial performance improvements while minimizing memory and bandwidth expenses.

5 Related work

Disaggregated memory. Recently, disaggregated memory has gained popularity both for general-purpose applications [4, 7, 23] and for specific use cases, such as cloud-native databases [25, 61, 64]. Solutions like Infiniswap [23], Remote Region [4] and Leap [7] implement disaggregated memory at the Linux kernel level for general usage but often suffer significant performance degradation when applied directly to cloud-native databases [63]. Consequently, dedicated disaggregated memory systems tailored to cloud-native databases have emerged. Systems such as LegoBase [64], PolarDB Serverless [13] and PilotDB [46] design disaggregated memory to store buffered pages based on RDMA. Although moving some hot pages to remote memory improves performance, these solutions remain limited by read/write amplification and RDMA driver constraints [45, 55]. Other works [35, 37, 38, 51, 52, 60] leverage RDMA-based disaggregated memory for efficient indexing or transaction optimization. However, these designs typically require extensive modifications to existing systems and remain largely experimental, requiring significant efforts to deployment in commercial databases. Unlike these approaches, this paper focuses on using emerging CXL devices to design disaggregated memory that overcomes the limitations of RDMA-based architectures for cloud-native databases. Furthermore, the proposed design minimizes modifications to existing systems and has been implemented in a widely deployed commercial cloud-native database, demonstrating practicality, cost efficiency, and significant performance benefits.

CXL-based systems. Since the emergence of CXL, several works have explored its applications in existing systems. Two studies [24, 31] present CXL use cases for databases but lack detailed evaluations. Tang [49] examines CXL performance and provides a cost-benefit analysis. Systems like Rcmp [53], TPP [41], Pond [34] and DirectCXL [22] design CXL-based memory disaggregation for general-purpose use. Minseon Ahn [5, 6] investigates CXL memory for in-memory databases, while HydraRPC [39] leverages CXL for implementing an RPC system. ReCXL [36] and DeepMemoryDL [9] explore CXL's potential in AI scenarios, and CXL-ANNS [26] utilizes CXL for approximate nearest neighbor search. However, these studies do not focus on cloud-native databases, where CXL-based disaggregated memory can be employed for database buffer pools, instance recovery, and data sharing.

Fast recovery. LegoBase [64], PilotDB [46], PolarDB-MP [58] and GaussDB [33] have explored using disaggregated memory to speed up recovery by reducing storage I/O operations. However, these systems still require scanning all redo logs and completing the full recovery process. PolarDB Serverless [65] proposes seamless migration to quickly start data instances, but it focuses on intentional migration scenarios. Additionally, other works [30, 40, 44] propose checkpoint algorithms for fast recovery in in-memory databases, while PLIN [66] achieves instant recovery using NVM. SiloR [67] and PACMAN [57] enable fast recovery through parallelism. Unlike these approaches, this paper enables instant recovery with disaggregated CXL memory, which can restore buffered data directly from CXL memory without the need for log application.

6 Conclusion

This paper presents *PolarCXLMem*, a CXL switch-based disaggregated memory for cloud-native databases, supporting both memory pooling and sharing. Based on studies of CXL switch characteristics, the design eliminates the tiered local-remote memory structure prevalent in RDMA-based systems, instead placing the entire buffer pool directly on CXL memory. By avoiding the read/write amplification inherent in RDMA-based disaggregated memory, *PolarCXLMem* achieves superior performance and reduced costs. Building on *PolarCXLMem*, we propose *PolarRecv*, an instant recovery scheme that significantly enhances database recovery performance. Additionally, to facilitate *PolarCXLMem* 's deployment in multi-primary databases, we design a novel data synchronization protocol that markedly outperforms existing RDMA-based solutions.

References

[1] 2020. CXL And Gen-Z Iron Out A Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/.

[2] 2022. EFI Special Purpose Memory Support. https://lwn.net/Articles/784971/.

[3] 2024. Intel® Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[4] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. 2018. Remote Regions: a Simple Abstraction for Remote Memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 775–787.

[5] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 1–5.

[6] Minseon Ahn, Thomas Willhalm, Norman May, Donghun Lee, Suprasad Mutalik Desai, Daniel Booss, Jungmin Kim, Navneet Singh, Daniel Ritter, and Oliver Rebholz. 2024. An Examination of CXL Memory Use Cases for In-Memory Database Management Systems using SAP HANA. *Proceedings of the VLDB Endowment* 17 (2024), 3827–3840.

[7] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.

[8] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[9] Moiz Arif, Kevin Assogba, M Mustafa Rafique, and Sudharshan Vazhkudai. 2022. Exploiting CXL-based Memory for Distributed Deep Learning. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[10] David Boles, Daniel Waddington, and David A Roberts. 2023. CXL-Enabled Enhanced Memory Functions. *IEEE Micro* 43, 2 (2023), 58–65.

[11] Anthony M Cabrera, Aaron R Young, and Jeffrey S Vetter. 2022. Design and Analysis of CXL Performance Models for Tightly-Coupled Heterogeneous Computing. In *Proceedings of the 1st International Workshop on Extreme Heterogeneity Solutions*. 1–6.

[12] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1604–1617.

[13] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.

[14] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–17.

[15] CXL Consortium. 2019. Compute Express Link (CXL) 1.0 Specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-1.1-Specification.pdf.

[16] CXL Consortium. 2020. Compute Express Link (CXL) 2.0 Specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf.

[17] CXL Consortium. 2022. Compute Express Link (CXL) 3.1 Specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf.

[18] Transaction Processing Performance Council. 1992. On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

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

[20] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th symposium on operating systems principles*. 54–70.

[21] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling with CXL. *IEEE Micro* 43, 2 (2023), 48–57.

[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DIRECTCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[24] Yunyan Guo and Guoliang Li. 2024. A CXL-Powered Database System: Opportunities and Challenges. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5593–5604.

[25] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[26] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 585–600.

[27] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. 2022. Aurogon: Taming Aborts in All Phases for Distributed In-Memory Transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 21). USENIX Association*.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 185–201.

[29] Alexey Kopytov. 2004. Sysbench: A System Performance Benchmark. *http://sysbench. sourceforge. net/* (2004).

[30] Leon Lee, Siphrey Xie, Yunus Ma, and Shimin Chen. 2022. Index Checkpoints for Instant Recovery in In-Memory Database Systems. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1671–1683.

[31] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. In *Proceedings of the VLDB Endowment*, Vol. 17. 2568–2575.

[32] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

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

[34] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. 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, Volume 2*. 574–587.

[35] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-Oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[36] Haifeng Liu, Long Zheng, Yu Huang, Jingyi Zhou, Chaoqiang Liu, Runze Wang, Xiaofei Liaot, Hai Jinf, and Jingling Xue. 2024. Enabling Efficient Large Recommendation Model Training with Near CXL Memory Processing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 382–395.

[37] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2603–2616.

[38] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 553–571.

[39] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, et al. 2024. HydraRPC RPC in the CXL Era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 387–395.

[40] Arlino Magalhaes, Angelo Brayner, and Jose Maria Monteiro. 2024. MM-DIRECT: Main memory database instant recovery with tuple consistent checkpoint. *The VLDB Journal* 33, 3 (2024), 859–882.

[41] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. 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*. 742–755.

[42] Micron. 2023. Micron's Perspective on Impact of CXL on DRAM Bit Growth Rate. https://tw.micron.com/content/dam/micron/global/public/products/white-paper/cxl-impact-dram-bit-growth-white-paper.pdf.

[43] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikk. 2011. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net/.

[44] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. 2015. Instant Recovery for Main Memory Databases. In *Conference on Innovative Data Systems Research (CIDR)*.

[45] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. 2024. Scaling Up Memory Disaggregated Applications with Smart. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 351–367.

[46] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. 2023. Persistent

Memory Disaggregation for Cloud-Native Relational Databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 498–512.

[47] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[48] Shigeru Shiratake. 2020. Scaling and performance challenges of future DRAM. In *2020 IEEE international memory workshop (IMW)*. IEEE, 1–3.

[49] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, et al. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 818–833.

[50] XConn Technologies. 2023. World's first CXL 2.0 and PCIe Gen5 Switch IC. https://www.xconn-tech.com/product.

[51] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 international conference on management of data*. 1033–1048.

[52] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2835–2849.

[53] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. 2024. Rcmp: Reconstructing RDMA-Based Memory Disaggregation via CXL. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–26.

[54] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. 2021. Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 357–372.

[55] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 233–251.

[56] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory Transaction Processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 87–104.

[57] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 267–281.

[58] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data*. 295–308.

[59] Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Wenchao Zhou. 2023. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3754–3767.

[60] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 801–819.

[61] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 51–68.

[62] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 658–674.

[63] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[64] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1900–1912. https://doi.org/10.14778/3467861.3467877

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

[66] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, et al. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proceedings of the VLDB Endowment* 16, 2 (2022), 243–255.

[67] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 465–477.

[68] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 international conference on management of data*. 741–758.