Persistent Memory Disaggregation for Cloud-Native Relational Databases

Chaoyi Ruan
rcy@mail.ustc.edu.cn
USTC, Alibaba Group
China

Yingqiang Zhang
yingqiang.zyq@alibaba-inc.com
Alibaba Group
China

Chao Bi
bc233333@mail.ustc.edu.cn
USTC, Alibaba Group
China

Xiaosong Ma
xma@hbku.edu.qa
QCRI, HBKU
Qatar

Hao Chen
ch341982@alibaba-inc.com
Alibaba Group
China

Feifei Li
lifeifei@alibaba-inc.com
Alibaba Group
China

Xinjun Yang
xinjun.y@alibaba-inc.com
Alibaba Group
China

Cheng Li
chengli7@ustc.edu.cn
USTC, Anhui Key HPC Lab
China

Ashraf Aboulnaga
aaboulnaga@hbku.edu.qa
QCRI, HBKU
Qatar

Yinlong Xu
ylxu@ustc.edu.cn
USTC, Anhui Key HPC Lab
China

ABSTRACT

The recent emergence of commodity persistent memory (PM) hardware has altered the landscape of the storage hierarchy. It brings multi-fold benefits to database systems, with its large capacity, low latency, byte addressability, and persistence. However, PM has not been incorporated into the popular disaggregated architecture of cloud-native databases.

In this paper, we present PilotDB, a cloud-native relational database designed to fully utilize disaggregated PM resources. PilotDB possesses a new disaggregated DB architecture that allows compute nodes to be computation-heavy yet data-light, as enabled by large buffer pools and fast data persistence offered by remote PMs. We then propose a suite of novel mechanisms to facilitate RDMA-friendly remote PM accesses and minimize operations involving CPUs on the computation-light PM nodes. In particular, PilotDB adopts a novel *compute-node-driven log organization* that reduces network/PM bandwidth consumption and a *log-pull* design that enables fast, optimistic remote PM reads aggressively bypassing the remote PM node CPUs. Evaluation with both standard SQL benchmarks and a real-world production workload demonstrates that PilotDB (1) achieves excellent performance as compared to the best-performing baseline using local, high-end resources, (2) significantly outperforms a state-of-the-art DRAM-disaggregation system and the PM-disaggregation solution adapted from it, (3) enables faster failure recovery and cache buffer warm-up, and (4) offers superior cost-effectiveness.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582055

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Hardware** → **Non-volatile memory**.

KEYWORDS

cloud-native database, persistent memory, memory disaggregation

ACM Reference Format:
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 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582055

1 INTRODUCTION

The past decade has witnessed the emergence and growth of disaggregated cloud-native databases, with successful systems such as Amazon Aurora [65], Alibaba PolarDB [14–16], and Microsoft Socrates [7], spanning their processing across multiple layers of network-connected resource pools (Figure 1). The compute nodes (CNs) host the computation logic, leveraging remote but shared DRAM space as an extension to the local buffer pool for memory capacity, and the replicated storage pool for data persistence and fault tolerance. For users, the rich, elastic, and on-demand configuration of disaggregated cloud-native databases cater to their diverse workload requirements and flexible scaling needs. For service providers,

such services allow the reuse of database software infrastructures as well as the consolidation/sharing of hardware resources.

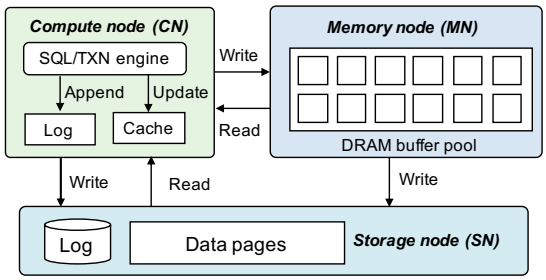

Figure 1: Sample disaggregated cloud-native DB architecture spanning three layers: CPU, memory, and storage

Though such new architecture enables the *independent scaling* of resources, there remain major constraints impeding its adoption. First, DRAM disaggregation faces its limited per-machine density, high (and fluctuating) price [47], and volatility, making it a more costly and less reliable layer for hosting a cloud-native database's working set. Second, writes remain slow, especially with transactions, as changes need to be persisted in time to the storage layer.

In this work, we argue that *Persistent Memory (PM)*, also known as non-volatile memory (NVM), driven by diverse technologies such as 3D XPoint [20], BiCS Flash [21], and PCM [55], emerges as an appealing layer for resource disaggregation. Compared with DRAM, PM offers higher provisioning density (e.g., one DIMM slot can hold 512 GB Optane PM, but only 128 GB DDR4 DRAM). It simultaneously offers persistence, enabling fast writes and recovery. In addition, PM preserves ultra-low-latency remote access via RDMA, an advantage over fast SSDs. Such multi-fold capability makes PM an ideal candidate for disaggregated databases, as we can simultaneously cache hot pages and persist log data on a shared and distributed PM layer. This brings on-demand, cost-effective memory buffer expansion, fast data persistence, and enhanced availability.

However, existing PM disaggregation work has not fully considered database redesign to utilize the versatile PM units, focusing instead on supporting native data structures [45] or simple applications like KV stores [63]. Applying these solutions to cloud-native databases could easily lead to new bottlenecks on the shared, remote PM nodes (PMNs). The first is the tension between the limited PM write bandwidth [26, 32, 75] and the heavy bandwidth consumption of existing solutions. The latter is largely due to writing redundancy/amplification caused by logging and dirty data flushing. Offloading log management to the PM side would reduce the PM bandwidth pressure (by not sending dirty pages but reproducing them by PM-side log application). On the other hand, this comes at the price of heavy CPU involvement on the PM nodes, required to handle offloaded data (especially their updates) and coordinate concurrent data accesses. Finally, complex management logic on the PM side would complicate the critical-path reads and writes. Both these PM-side bottlenecks (write bandwidth and CPU), unfortunately, conflict directly with the main selling point of PM disaggregation for the cloud: having a shared PM node pool supporting many compute nodes running database instances.

To address these challenges, we propose *PilotDB*, a novel PM-disaggregated cloud-native database architecture featuring the following innovations.

First, PilotDB embodies *CDLog* (Compute-node-Driven Logging), a central logging mechanism that efficiently offloads bulk data to the PM layer as a large, fast page buffer, yet with light computation there to support speedy logging and update handling. While retaining page-based data organization of relational databases, it discards the conventional page-based WAL organization and instead adopts fine-grained, physical logging, where data entries directly embed changes at a mini-page granularity as well as concerned remote PM memory addresses. This allows compute nodes only to flush *CDLog* entries to remote PM via one-sided RDMA and enables light-weight, DMA-based log application on the PM nodes, simultaneously reducing PM nodes' CPU and write bandwidth consumption.

Second, PilotDB is designed to be *coordination-free*, even in the presence of concurrent reads/writes to the PM log and buffers, further shaving CPU consumption on the PM nodes. This is enabled by (1) lock-free data structures designed to manage the PM log area, with light-weight conflict check mechanisms and (2) a novel *log-pull* mechanism that allows compute nodes' query processing to perform remote reads optimistically, with logs "read back" from the PM side in the rare occasion of the retrieved PM-cached page found stale, again enabled by our *CDLog* organization.

We implemented a PilotDB prototype atop MySQL [23] and evaluated it using both industry-standard benchmarks and a production workload. The results show that PilotDB achieves up to 98.0% of the throughput of a monolithic configuration (which is given sufficient local DRAM and PM-based storage), even with the vast majority of its data placed on remote, disaggregated PM. With most workloads, PilotDB significantly outperforms LegoBase [74], a state-of-the-art DRAM-disaggregated cloud-native database, and LegoPM, a solution incorporating PM disaggregation. In addition, we made a best-effort attempt to compare PilotDB with Aurora and PolarDB, two mainstream cloud-native database services on the market that adopt storage disaggregation, by allocating Aurora/PolarDB instances with sufficient local DRAM (and careful hardware alignment in other resource dimensions). Results show PilotDB achieves significantly better or comparable performance.

In addition to the above performance results, our multi-tenant tests show that PilotDB has strong service scalability, with a 4-node PM pool serving 32 concurrent DB instances at only a 10.8% performance loss against running each instance exclusively. Moreover, PilotDB brings instant failure recovery, up to 15.27× faster than the baselines, regardless of the crash site. Finally, our cost analysis further confirms the cost-effectiveness of PilotDB. Compared with its closest competitor in cost-effectiveness, the PilotDB configuration is 38.3% lower in hardware ownership cost, uses only 9.1% DRAM across CN and PMN, and 12.5% PMN's CPU core resources, while delivering 91.5% higher throughput per dollar.

To our knowledge, PilotDB is the only database design that leverages *all* major features of PM for disaggregation: capacity, persistence, and RDMA-based low-latency remote accesses. Our research contributions are as follows:

- We advocate a flexible 3-level cloud-native database architecture with aggressively disaggregated resources. It makes CNs

computation-heavy yet data-light, dramatically reducing DRAM provisioning while enabling agile recovery/migration.
- We propose a novel log-centric DB operation mechanism driven by a new log organization (CDLog). It adopts fine-grained and physical log entries designed for fast data persistence and CPU-bypassing PM-side log application, making compute nodes approach stateless services [16].
- We further present a minimal CN-PMN data path design not seen in prior systems, with key functionalities (logging, optimistic read, and log-pull) each done by a single one-sided RDMA operation, without involving PMN-side CPUs.
- Supported by performance evaluation and cost analysis results, this work provides evidence that PM is large, fast, and cheap enough to be used for on-critical-path data caching and persistence, contributing a practical use case for true disaggregated PM offerings in the near future.

2 BACKGROUND

2.1 Disaggregated Cloud-Native Relational Databases

Cloud-native databases refer to systems built on modern cloud infrastructures and offering OLTP processing as a *service*. Compared with existing DB systems simply running on the cloud, they are designed to leverage better cloud benefits such as resource auto-scaling and high availability. Database services' demands in elastic resource allocation and guaranteed latency render them ideal customers for the ongoing resource disaggregation innovations [3, 28, 58, 62], driven partially by the rapid development of low-latency hardware technologies like RDMA. Recently, cloud-native databases [7, 12–16, 25, 46, 57, 65, 74] have been adopting network-connected, flexibly provisioned resource pools, often using software-hardware co-design to reduce network overhead while achieving comparable performance to conventional designs with monolithic resource allocation.

Two-tier architecture (storage disaggregation). Most of the above cloud-native DBs support storage disaggregation with independent compute and storage resource tiers. For example, Alibaba's PolarDB [14, 15] replaces its local storage with a layer of disaggregated storage nodes (SNs). The compute node (CN) side retains most of the database logic, such as transaction execution, concurrency control, Write-ahead Log (WAL) [48] generation, and failure recovery. Only the persisted states (WAL and data pages) are pushed to the remote storage, with optional replication for fault tolerance.

To reduce network pressure, Amazon Aurora [65] and Socrates [7] further offload the critical *page materialization* and *fault tolerance* modules. Their compute nodes write only WAL entries, instead of pages, to the storage nodes. The latter continuously iterates over these log entries in the background and applies the modification chains to the target data pages, adopting a design referred to as "the log is the database."

Three-tier architecture (DRAM disaggregation). More recently, resource disaggregation has expanded naturally to the memory layer [3, 28, 39, 66], consolidating memory resources and decoupling their provisioning from CPU resources. Recently proposed memory-disaggregated cloud-native databases, such as PolarDB-serverless [16] and LegoBase [74], possess separate compute node (CN) and memory node (MN) layers, enabling independent memory resource scaling. This design allows a CN to maintain a very small memory pool to cache the hottest data, relying on the remote memory pool on the MNs as a buffer extension. Meanwhile, the memory-disaggregated database still needs to write WAL entries to the storage node (SN) when a transaction commits, while it flushes data pages from the CN to both the MN and SN.

DRAM buffer and storage limitations. Though promising, the three-tiered disaggregated cloud-native DB architecture has two major problems. First, DRAM has a limited per-machine density, with a single DIMM supporting at most 128 GB currently, leading to high monetary investment for DRAM nodes. Second, DRAM is volatile, therefore slow log persistence to storage (where fast RDMA does not apply) is still required and DRAM-disaggregation solutions suffer a long recovery time, as to be shown later in our performance comparison. Note that in-memory replication helps with the second problem above, at the cost of intensifying the first.

2.2 The Case for Persistent Memory Disaggregation

Persistent memory (PM) emerges as an appealing intermediate media, bridging between the DRAM-based computation layer and SSD/HDD-based storage layer. Multiple technologies, developed by various vendors, emerged from manufacturing PM chips, such as the Intel 3D XPoint [20], phase change memory [55], STT-MRAM [8], etc. In 2019, Intel released the first commodity PM product – the Optane PM [31]. More recently, CXL-based solutions approximate PM by combining DRAM, flash SSDs, and memory-compatible interconnects [22, 61]. Despite their diversity, these PM chips simultaneously provide byte-addressability, RDMA accesses, persistence, and orders-of-magnitude latency advantage over fast SSDs. Cost-wise, the PM sits squarely between DRAM and flash SSDs, thus with its superior capacity density (e.g., up to 512 GB Optane PM per DIMM), bringing a cost incentive as a candidate for resource disaggregation as well.

PM has attracted attention in the database community [9, 30, 34, 40, 64, 77]. However, most of the above systems, including the recent Oracle Exadata [51], treat PM as a locally attached resource and have not considered it a layer of disaggregated resources.

Meanwhile, there is recent work on PM disaggregation [45, 63], adding a group of PM nodes (PMNs) to the system. For instance, AsymNVM [45] leverages PM disaggregation to build persistent data structures with high-performance writes. Its downside is heavy contention on the CPU cycles of remote PM nodes, for ensuring data durability, consistency, and availability. In contrast, pDPM [63] makes the PM layer passively serve data, with the compute nodes directly managing the remote PM resource to eliminate remote CPU consumption.

2.3 PM-Specific Disaggregation Challenges

Previous PM disaggregation proposals [45, 63] focus on supporting native data structures or KV stores. Relational databases, on the other hand, possess much higher software complexity, posing functionality placement challenges not addressed by existing solutions.

On the one hand, an *onload* design, such as pDPM [63], leaves the PMNs' CPU out of the loop of handling logs and pages. Meanwhile, it makes the PM/network bandwidth more bottleneck-prone due to dirty page flushing, log application, and fault tolerance, all driven by the CN in pDPM (as shown by the similar LegoPM behavior in Table 1). On the other hand, an *offload* design, such as AsymNVM [45]), significantly reduces bandwidth consumption, at the cost of heavy remote CPU usage to serve data and ensure data durability, consistency, and availability. This creates a new CPU bottleneck, just when the PM's large capacity and high-density installation enable a desirable disaggregation setup, with each PMN supporting many tenants.

The onload-offload tradeoff is further complicated by RDMA, a crucial mechanism enabling low-latency memory disaggregation. By leaving remote data managed by PMNs, it is challenging for offload solutions to utilize the faster and less CPU-consuming one-sided RDMA. Instead, it requires using many RPC operations, with heavy PMN-side CPU involvement. Latency-wise, our tests find that one-sided RDMA costs under 8 µs end-to-end for 16 KB page remote write, 65.6% shorter than the RDMA-based X-RDMA [44], one of the fastest RPC frameworks to our knowledge.

This RPC reliance, combined with the log management burden, results in a PMN CPU bottleneck demonstrated by the similar PilotDB-RPC baseline in Figure 6.

Finally, both the existing onload and offload solutions focus on a two-tiered architecture, where the disaggregated PM layer hosts the entire dataset.

Our proposed PilotDB can be viewed as a hybrid approach sitting between existing onload and offload approaches. It also advocates a three-tier memory/storage disaggregation architecture, where it couples large, PM-based remote buffers with smaller DRAM local buffers for cost-effectiveness, especially considering the access skewness common in DB workloads.

3 PILOTDB OVERVIEW

3.1 Design Rationale

To our best knowledge, this is the first work exploring PM disaggregation for cloud-native relational DBs, the most popular DBMS model [24], to take full advantage of the multifaceted strengths of PM. We focus on OLTP workloads, which often contain a substantial amount of updates and have stringent latency and recovery time requirements. Thus, OLTP engines favor a large DRAM buffer pool for speedy lookup, plus a fast storage layer for efficient data flushing (WAL in the foreground and dirty pages in the background). This leads to a natural design with a shared PM layer consisting of PMNs, hosting the bulk of hot data and handling WAL writes, shifting these two duties from the DRAM and storage layers, respectively.

However, we face two major challenges in designing such a disaggregated PM layer, as detailed below.

Challenge 1: PM bandwidth contention: PM has considerably lower bandwidth than DRAM. Redirecting the vast majority of DRAM traffic (loading pages, logging WAL entries, and flushing dirty pages) to remote PM, however, will generate high PM bandwidth demands, especially for writes. To alleviate such bandwidth tension, like existing offload approaches [5, 7, 45], PilotDB sends only logs to PMNs, which update PM-buffered pages accordingly for subsequent reads. It however differs from existing solutions as its PMNs enable almost complete CPU-bypass on critical query paths: it uses a single one-sided RDMA operation to perform key tasks such as WAL writing, log application, and ultra-low-latency page read and persistence.

The central idea of PilotDB is to decouple the storage/persistence and computation components of DB logging: logs are shipped to and applied at the PM side, but managed and controlled at the compute side. For this, we propose *CDLog (Compute-node-Driven Log)*, a new logging mechanism *enabling fine-granularity page updates, eliminating the log application overhead on PMNs,* and *maximizing the use of one-sided RDMA*. CDLog re-organizes the PM-side log layout to contain physical page content at a mini-page granularity, with relevant metadata addresses embedded. From the data transfer point of view, this greatly reduces the network and PM bandwidth consumption. From the computation point of view, PM-side log application becomes simple, fast memory copy operations, resulting in light-weight remote log management.

Challenge 2: PM side CPU consumption: The PM's large capacity and high-density installation allow a PMN to support many tenants. However, its CPU resources could quickly become a scalability bottleneck. To this end, we strive to make PM-side compute-light, while still processing log applications and coordinating concurrent CN accesses.

On top of CDLog, we further build a PM-side data plane to virtually eliminate CPU consumption. This includes lock-free data structures, such as ring buffers for PM-side logs, to reduce cross-network coordination for concurrent accesses by CN's log writes and PMN's application. Recognizing the necessity of version check for stale read avoidance, especially with our PM-side fast log application, we devise an optimistic *log-pull* mechanism. It onloads version check and on-demand log application to CNs, so as to simplify the PMN functionality and avoid slowing down queries on the critical path. In more detail, a CN could directly read remote PM pages using one-sided RDMA, only performing on-demand log application locally by pulling back appropriate log entries from a PMN in the rare case of finding a page stale (whose extremely low frequency is given in Table 3).

3.2 PilotDB Architecture Overview

We propose a 3-level architecture for cloud-native relational DBs spanning three layers of nodes, as described below.

Compute node (CN). CNs are provisioned with sufficient CPU cores and limited DRAM for serving user queries. Each CN hosts SQL/TXN engine threads and a moderate-sized *local buffer pool (LBP)* to cache the hottest data in DRAM.

Persistent memory node (PMN). PMNs are a set of PM-equipped servers hosting a layer of shared PM resources and exposing a memory interface to the CNs. A cloud-native DB instance could provide a certain amount of PM space to accelerate its processing. When such space is allocated, the involved PMNs run a light-weight PilotDB PMN daemon supporting two PM-based partitions transparent to DB users: a *remote buffer pool (RBP)*, with sizes typically much larger than their LBP counterparts, and a relatively small *CDLog store*, for fast WAL persistence.

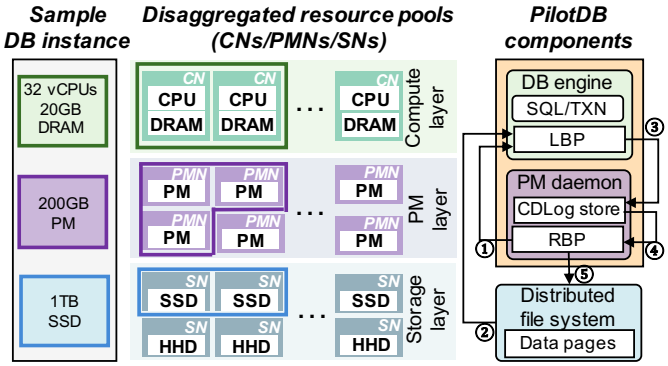

Figure 2: PilotDB architecture overview

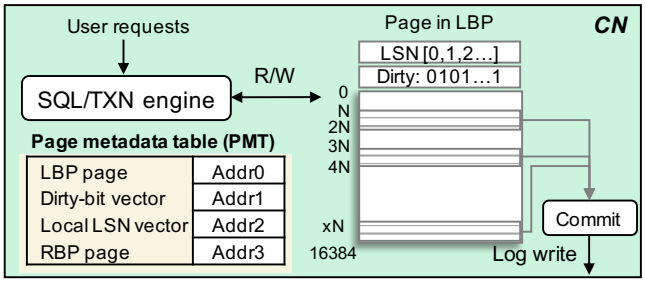

Figure 3: PilotDB CN architecture and page structure

A central registry service allocates PM resources to a DB instance and helps its CN(s) connect to the appropriate PMN group upon database bootstrap. RBP pages are distributed by hashing for load balancing and log stores are replicated for availability, across multiple PMNs. Within a DB's PMN replicas, one is appointed the *primary*, from where PilotDB pulls log data for update application (regardless of whether the target RBP page is on local or remote PM, thanks to one-sided RDMA). As the above distributed system elements leverage mature techniques, this paper's discussion focuses on the PM-disaggregation aspect of DB re-design.

Storage node (SN). The storage layer consists of many SSD/HDDs as long-term and low-cost storage. Each CN is further connected to a group of SNs for distributed file system (DFS) service, with file interfaces and RDMA-based high performance. PilotDB utilizes the storage layer for storing the complete databases and checkpoints, in traditional manners, so the discussion in this paper focuses on CNs and PMNs.

Figure 2 illustrates the PilotDB overall architecture. On its left, we give a sample cloud-native DB instance's resource specifications, which are individually and dynamically allocated from the corresponding disaggregated resource pools shown in the middle (CNs, PMNs, and SNs). Note that each box containing "PM" or "SSD/HDD" is a node (PMN or SN) serving disaggregated resources.

On the right of Figure 2 we give the placement of major DB software components, with PilotDB logic located on the compute and PM layers. Here the arrows outline PilotDB's transaction execution workflow. A transaction first accesses the LBP, which caches the most recently used data pages, at a latency of ~0.91 µs. Upon an LBP miss, the CN contacts the appropriate PMN to fetch the target page from its RBP (①, at ~8 µs). If the RBP also misses, the page is fetched from the shared storage to the CN (②, at ~277 µs), which immediately sends another copy to its RBP (further discussion below). CN collects log entries from writes within the node and flushes them to the CDLog store of its primary PMN (③), for background log application to RBP pages (④). In addition, the CN sends one or two copies of the log data to other PMNs for replication. PMNs evict colder pages when under space pressure, with dirty pages flushed to shared storage (⑤).

We design the PilotDB DRAM-PM page buffers to be *inclusive*: all pages cached in the CN-side DRAM have a copy in the PM layer, for multiple reasons. First, such a design keeps a copy of the hottest CN-side pages in PM, enabling fast CN crash recovery and migration. Second, this allows fast LBP-to-RBP page eviction, by removing the need for page flush upon an RBP hit (which leads to page promotion to LBP). We choose instead to pay the CN-to-PMN page copy overhead upon an RBP miss, which is a less frequent event. Finally, given that PilotDB is designed to enable lighter and more agile CNs with an order-of-magnitude smaller LBP than RBP, plus the cheaper PM storage, the space cost of making the hierarchy inclusive is fairly low.

4 DESIGN

4.1 CN-Driven Log and Buffer Management

PilotDB proposes consolidating the available PM resources on dedicated PMNs (currently requiring high-end processors), which form a separate, shared, and disaggregated PM layer. Space allocation from this layer is individually and dynamically provisioned toward the need of each DB instance.

A key design decision of PilotDB is to split each 16 KB page into configurable, equal-sized *mini-pages*. This follows recent practice [64, 77] for fine-granule data access, which in our case avoids read/write amplification between CNs and PMNs. By default, the PilotDB mini-page is sized at 256 B, the 3D-XPoint physical media access granularity [71]. Our data structure and workflow discussion below mainly focuses on mini-page-level management.

Figure 3 illustrates the CN-side major data structures in DRAM and the associated transaction execution data paths. LBP-related details are omitted, as PilotDB adopts standard LBP management. Note that RDBMSs like MySQL log *logical operators*, requiring page flushing applying changes to full pages, a procedure both CPU- and bandwidth-expensive. PilotDB's CN retains the WAL operations, but avoids such costly dirty page flushing with its *CDLog*, where entries encode *physical page changes* at a finer granularity.

PM-knowledgeable CN. A unique PilotDB design feature is that the CNs have full knowledge and access to the remote PM space, essential to fast, RDMA-based remote PM access. To this end, each CN maintains, as its central page management data structure, a *page metadata table (PMT)*. It is an open-addressing hash table mapping page IDs to the location of metadata structures for each buffered page, as shown in Figure 3. For pages cached locally, in addition to recording its LBP address (the "LBP page" field), the PMT entry also points to two auxiliary data structures: (1) a *dirty-bit vector* tracking which of its mini-pages have been modified and (2) a *local LSN vector* recording the highest log sequence number updating

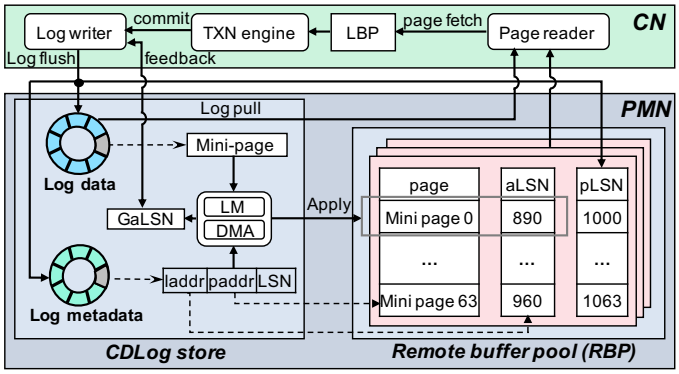

Figure 4: PMN data-path workflow. Note: CDLog store (left side) and RBP (right side) may be on different PMNs.

each mini-page at CN side. Memory space overhead of PMT is quite small: 32 B per 16 KB page, resulting in a 2 GB PMT for a 1 TB DB.

One-sided PM Page Registration. As mentioned earlier, an RBP miss will bring the page directly to the CN's LBP, which will be copied to its RBP. For this, the CN first performs *RBP page registration* on the PMN to get one PM page allocated. To avoid using expensive RPC calls, we instead bind each metadata entry with one remote page address during LBP initialization, as the CN knows the remote PM layout, *i.e.*, filling the `Addr3` field shown in Figure 3 in advance. After copying this page to PM, the CN also persists the "RBP page" field there, for PMT reconstruction under CN crash. Similarly, page de-registration is done upon RBP page eviction, in the background. Though not shown in Figure 3, the CN maintains LRU lists for both the LBP and RBP, dictating RBP cache management.

Fast on-critical-path data persistence. Upon transaction commit, PilotDB persists corresponding changes by writing CDLog entries to the allocated area in the PM-side CDLog store (Figure 4). Each CDLog entry consists of two parts: metadata and data. Multiple PilotDB design decisions accelerate such data persistence.

First, we adopt the mini-page level logging for low PM/network bandwidth consumption. At commit time, PilotDB inspects the dirty-bit vector of touched pages to figure out the list of dirty mini-pages. For each dirty mini-page, it creates a *log metadata entry* – `<laddr, paddr, lsn>`, filled with the address of the mini-page's PM-applied LSN vector (to be introduced below), the PM address of the target mini-page, and the latest LSN updating this page in this log entry. Note that although PMT entries are page-level, given the page address and a per-page vector, mini-page addresses can be easily calculated by the CN using offsets. Alongside the log metadata entry, it also generates a log data entry, containing the new content of that mini-page.

Second, a PilotDB CN writes the two entries to the tail of the log data and metadata ring buffers at the PM side, respectively, via two one-sided RDMA operators. This allows us to bypass the PMN-side CPU and avoids CN-side memory copies (from the conventional log buffer to the RDMA-registered buffer). Also, we adopt RDMA batching as an additional optimization to reduce network overhead.

4.2 Light-Weight, RDMA-Friendly PMN Processing

PMN data structures. Figure 4 illustrates PilotDB's major PM-resident data structures, designed to facilitate high-speed, light-weight, and lock-free PMN processing.

The right side of the figure depicts the RBP, a page server with capacity and persistence advantages over the DRAM-based LBP and a speed advantage over shared storage. The PM also stores two vectors for each RBP-cached page: a PM-applied LSN (aLSN) vector and a PM LSN (pLSN) vector. As the PMN side does not generate new updates, it only applies the CN-side changes to RBP-cached pages, recording its own progress by noting in the aLSN the latest LSN whose changes have been applied to its mini-pages. pLSN, instead, is written by the CN to persist the target LSN each mini-page should be updated to and only used for CN recovery.

The left side of Figure 4 shows the *CDLog store*, with two ring buffers for log entries shipped from the CNs, one holding metadata and the other holding the corresponding mini-page changes. Such a dual-ring-buffer design aligns with the common practice adopted in mainstream DBs (including MySQL) that keeps metadata and actual data changes in separate data structures. The CN needs to write to the two ring buffers with two RDMA operations, which can however be batched.

Light-weight log application. Given the major data structures connecting the CN and PMN spaces, we now present the central techniques that make the PMN-side computation light-weight. Here the main task is to scan the WAL entries and apply updates accordingly. PilotDB enables fast log application with little PMN CPU consumption for two reasons.

First, with CDLog embedding target PM (mini-)page addresses in its log entries, a PMN does not need to parse logical operations in the log or execute encoded mutations, as with conventional redo logging mechanisms [48]. Instead, it performs much cheaper direct memory copies. A *log merger (LM) daemon* running in the background on the PMN applies changes from the log data ring buffer to their destination mini-pages and updates the corresponding aLSN, whose address is given in the log metadata entries.

Second, PilotDB drives the above log application using a DMA engine for memory copy. Here we employ RDMA, which accelerates the case where the CDLog entry and its corresponding page are not on the same PMN, inevitable with our fully disaggregated design. It also works well with local data copying when the two objects co-reside. The result is a virtually CPU-less DB page server. Note that for such write-intensive workloads, the log merger can utilize RDMA doorbell batching [35], which reduces polling overhead by consolidating multiple one-sided RDMA operations' completion entries (to enter the completion queue for polling) into one.

Optimistic remote read. Upon LBP misses that are RBP hits, the CN retrieves the page from a PMN. Simply issuing a one-sided RDMA read primitive is not safe, as the page could be stale, with pending log application. The conventional design adopted by existing "the log is the database" systems [65] waits until the remote PM layer scans the log buffer, parses log entries, and executes the encoded logical mutations against target pages. Such on-demand log application introduces heavy PMN CPU involvement. In particular, despite that stale page retrieval is rendered very rare by PilotDB's

fast log application, *each RBP page* the CN reads has to go through this PM-side freshness check for correctness, which requires RPC and precludes the use of much faster one-sided RDMA primitives.

Again, with PilotDB's CDLog design allowing a CN the full knowledge and control over its PM-side content, it affords a different approach: performing an *optimistic remote read* and conducting the page freshness check locally. Recall those metadata items like the remote page addresses are stored locally within the CN-side PMT. Upon an LBP miss, the CN will simply read the target RBP page using one-sided RDMA. It then checks if the page is stale by comparing its aLSN vector (fetched together with the RDMA page read) against the local one. In most cases, the two vectors match and the page passes the freshness test, concluding the RPB-to-LPB page fetch.

In the less likely case of the page failing the above test, rather than waiting for the remote updates to fetch again, the CN pulls the relevant log data to apply the changes itself locally. Again, this is easy as (1) the CN has all the needed addresses in its PMT for one-sided RDMA and (2) only the latest log per mini-page is needed with our physical log design.

Data integrity and consistency. Our workflow involves RDMA operations accessing disjoint DRAM/PM locations storing components of the same data item, such as the log metadata and log data in separate ring buffers. Therefore we append an 8-byte checksum field to the log metadata entry to protect data integrity and avoid partial writes in the case of crashes. This field will be set by the CN prior to log flush and later used by both the PMN and CN to verify the completeness of RDMA data transfers.

To offer scalable concurrent CN/PMN accesses to the CDLog store, PilotDB strives for a lock-free PMN design to avoid the use of particularly expensive distributed locks.

First, the chance of ring buffer overflow is greatly reduced by the fact that the CN side (log producer) is responsible for all the heavy lifting in transaction processing, while the PMN side (log consumer) only performs light-weight log application, with roughly two orders of magnitude difference in latency. With the CN/PMN going around the ring buffers producing/consuming log items *with strictly incrementing LSNs*, accidental overreads/overwrites can be easily avoided with additional coarse-grained monitoring. For example., PilotDB maintains a per-PMN *Global applied LSN (GaLSN)*, which records the LSN of the last entry that the LM applied and is checked periodically by the CNs. This way, no locks are needed to coordinate the CNs and PMNs in populating and consuming the shared ring buffer contents. The log-pull operations, similarly, are one-sided sneak peeks that do not interfere with the PMN log merging operations.

Page eviction. With our 3-layer architecture, there exist two types of page eviction. One is when the LBP is nearly full, prompting background eviction to the RBP. Here the CN does not need to flush dirty pages to PM, as corresponding page changes are already propagated to PMN by CDLog. The other is when the RBP is nearly full, prompting its background eviction to the storage layer. In order to unify eviction handling and again to reduce the PMN CPU consumption, we maintain one CN-side flush-list to coordinate both LBP and RBP background dirty page flush to the storage.

4.3 Replication and State Recovery

We employ two tiers of replication to offer high data availability and fault tolerance. The first tier replicates CDLog entries across multiple PMNs. The second tier is the standard replication within the underlying SSD-based shared storage. Considering the much higher price of PM compared to SSDs, we decide to replicate only log data among PMNs, while keeping only one copy of each page on one PMN. Since one-sided RDMA write is fast, we make a CN issue a group RDMA write to flush log entries to the ring buffers at all PMN replicas. To replicate between PMNs and shared storage, we periodically create checkpoints and flush them to the underlying shared storage, where all data are fully replicated across storage nodes. Once a dirty page is persisted in the shared storage, we can simply discard its multiple log copies and change its status from "dirty" to "clean". With the above design, state recovery in the presence of failures can be made straightforward. Here, we consider two types of failures.

CN failure. When a CN restarts after it crashes, it first connects back to its PMNs, using connection information managed by the registry service, for instant recovery. As the cached PM pages are still available and reusable, only a small amount of corrupted states left by incomplete transactions need to be properly handled.

More specifically, the CN fetches three necessary metadata items from the PMN, all persisted at fixed locations. These include the RBP-page-fields (written to PMN upon page registration), so that it could reconstruct the CN-side PMT for virtual memory mappings. In addition, it reads the latest LSN to continue its monotonically increasing LSN assignment. To recover from a corrupted state, the CN pulls undo pages from the PMNs, which record uncommitted transactions and pre-images of the affected data pages. These pages are written using the conventional database undo logic, with modifications flushed to PMNs via our CDLog commit. The CN then rolls back these transactions by restoring their pre-images.

Finally, after these steps, the CN returns to its normal state and can start processing queries immediately. Meanwhile, at the PMN side, the state recovery will take place when the log merger consumes CDLog entries from the GaLSN position by applying their changes. Since the LBP is relatively small and a large fraction of pages are still cached remotely, the CN's warm-up phase will be both short and light-weight, again using RDMA operations.

PMN failure. First, we consider the case where a PMN crashes but the physical server on which it runs can be restarted. Due to the persistent nature of PM, the state of the PMN before the crash will be available after a restart. The PMN maps the PM region to the previous virtual memory address to keep the CN-side remote page virtual address information still valid. In this case, when the PMN restarts, it will notify the corresponding CN via re-connection to start a similar corrupted log identifying procedure for the above CN failure handling to undo unfinished transactions. Then the PMN could start accepting CN's requests immediately without a warm-up phase, as all pages are already persisted on that server and can be reused.

Second, we have to handle the worst case where the server on which the primary PMN runs is no longer available and needs to be replaced with another PM-equipped server. In this case, log entries need recovery and the RBP needs warm-up. The new PMN

could start serving CN requests when the former is done from the log replicas. In the background, the PM log merger executes its routine log application from the latest checkpoint to bring pages (fetched from the shared storage) up to date. However, this affects the handling of LBP misses. If the target page was cached on the old PMN prior to its crash, we need to check if it has been updated to the latest version. If not, the CN fetches a clean version of that page from shared storage and performs log-pull to re-generate its most recent version. Simultaneously CN pushes the latest local pages version from the LBP to the RBP.

Compared with the instant recovery from CN failures, it will take a longer time to warm up the PMN's RBP through RBP misses. However, since our design distributes pages across all PMNs, this cost can be diluted when more PMNs are added. This has been confirmed by Ongaro [50], a successor of RAMClouds [53], which scatters backup data across a large number of servers and harnesses them in parallel to reconstruct lost/corrupted data.

In summary, our failure handling follows conventional RDBMS recovery and brings no semantic change from the MySQL design regarding recovery or data consistency. Note that the procedure works regardless of whether any of the CNs co-locate with a PMN. The main difference is performance: with the bulk of dataset residing in RBP, recovery is faster, especially with CN failures. Figure 8 gives related results.

5 EVALUATION

PilotDB is implemented in around 5000 lines of C++ code, reusing DFS-like storage. CN runs a modified MySQL database, with changes mainly within two major areas: (1) the log module, where we replace the original WAL with CDLog (plus new mechanisms such as log-pull) and (2) the buffer pool module, where we retain the MySQL LRU to manage the cached pages while extending the buffer hierarchy to add the RBP. The PMN runs a light-weight daemon whose major tasks include background log application and recovery.

In this section, we present the evaluation results on PilotDB's overall performance and performance breakdown, recovery, elasticity, and cost efficiency.

5.1 Experimental Setup

Test platform. We use a heterogeneous cluster composed of three groups of physical servers: (1) 4 nodes forming the disaggregated PM layer, each equipped with two Intel Xeon Platinum 8260 CPUs, 256 GB DDR4 DRAM, a 4x128 GB 3D XPoint Intel Optane DC Persistent Memory per socket (in App-Direct mode), and 25 Gbps network connection, (2) 8 servers with a weaker network connection (10 Gbps), each with 2 processors and 128 GB DRAM, and (3) 6 servers forming the shared storage layer, providing a 3-way replicated, cloud distributed file system service, leveraging SPDK [18], RDMA, and NVMe SSDs. Note that the compute layer is formed by nodes from both group (1) and group (2), emulating actual heterogeneity seen in cloud systems. When a node in the group (1) is used as a CN, it accesses remote PM only (PM attached to other PMNs). We enable fast RDMA access to PM by configuring it in devdax mode [19].

Baseline setup. We attempted to compare with DBs running on existing disaggregated PM systems, such as Hotpot [59]. Unfortunately, we could not find a working system with compatible Linux kernel.[1] We therefore make our best effort to set up alternative designs, with the following baselines:

- "100D/0 MySQL-ideal", a single-node running MySQL with all resources local, including a 100 GB DRAM buffer and 512 GB Optane PM for WAL persistence and storage.
- "10D/100D LegoBase", a DRAM-disaggregated, cloud-native relational DB [74]. Its CN runs the MySQL kernel with a 10 GB local and a 100 GB remote DRAM buffer for extended cache, persisting WAL and pages to shared storage.
- "10D/56D LegoBase", same as above except that the remote DRAM is configured at 56 GB, to match the hardware cost of 100 GB Optane PM.
- "10D/100P LegoPM", a PM-disaggregation solution by adapting LegoBase, replacing its remote DRAM buffer with a PM buffer. Unlike PilotDB, it simply offloads RBP management and WAL persistence to the PMNs.

System configurations. To evaluate scenarios with large databases and to examine PilotDB's potential in reducing local DRAM requirement, we limit the LBP size to 5% of the total dataset in most cases and a 100 GB RBP (half of the DB size for most of our tests). Note that unless otherwise noted, we deploy a single CN as this work focuses on evaluating alternative resource disaggregation designs for relational DB. We do deploy more CNs in our multi-tenant tests to demonstrate the service capacity of our shared PM layer.

We align the CPU resource allocation for the compute nodes of all systems, giving 16 cores per CN in most cases. Unless stated otherwise, we allocate sufficient CPU cores (26 in our case), per PMN/MN for memory disaggregation, so that the baseline systems are able to demonstrate their performance without saturating the remote CPU resources.

Workloads. We evaluate with two standard OLTP benchmarks (TPC-C [11] and Sysbench [38]), plus a production MySQL workload. For Sysbench, we set up a database consisting of 32 tables, each with 28 M items, creating a total footprint of 200 GB. Requests are issued using *Zipfian-0.99*, following production workload statistics [17]. We adopt Sysbench's standard write-ratio configurations: RO (read-only), WO (write-only), and RW (7:2 R-W ratio). We use the TPC-C for MySQL implementation by Percona [54], which reports throughput and tail latency. We configure it to use 2000 warehouses, creating a 200 GB DB. Finally, we use a production SQL workload from Alibaba Cloud, with a profiled mix of 3:2:5 insert:update:select ratio. The dataset size is also configured to 200 GB, with requests issued synchronously.

5.2 Sysbench Results

Overall performance. Figure 5 gives Sysbench overall performance results, with growing CN-side processing concurrency. As expected, PilotDB's fast remote PM helps most with more write-intensive workloads: with WO, the PilotDB throughput reaches 97.0% of MySQL-ideal, with at least a 1.53× improvement over the other three baselines.

[1] We gave up test deployment of Hotpot after spending substantial effort and submitted an issue summarizing our encountered problems [10].

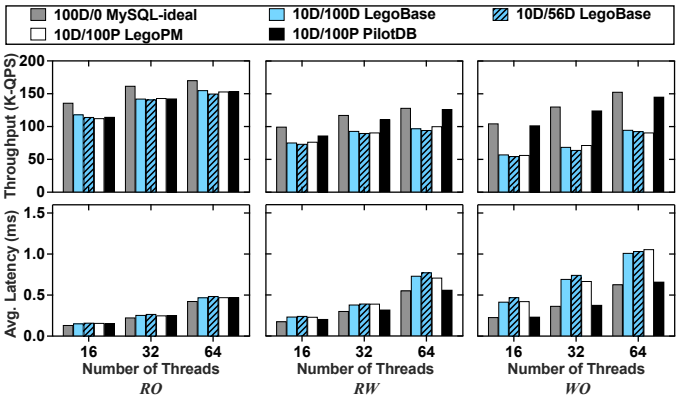

Figure 5: Sysbench overall performance w. 200GB DB fi

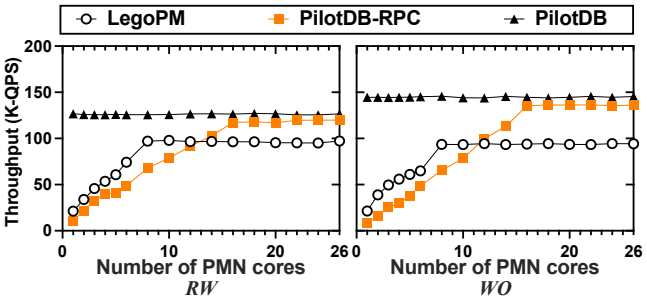

Figure 6: Impact of PMN core allocation w. 200 GB DB

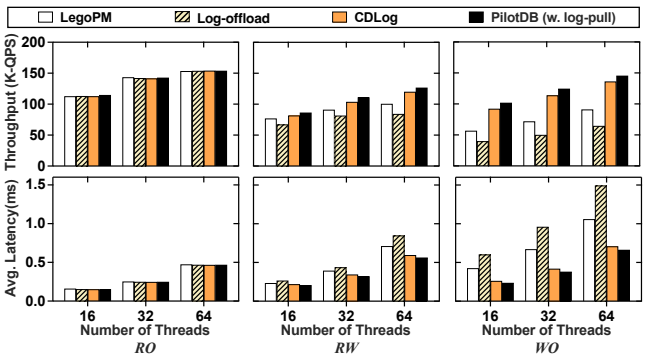

Figure 7: Breakdown analysis w. 200 GB DB

Table 1: LegoPM / PilotDB performance and PM bandwidth usage w. 200 GB DB (64-thread)

	Thpt (K-QPS)	PM Read (GB/s)	PM Write (GB/s)
RO	153.31 / 153.72	0.63 / 0.63	0.20 / 0.20
RW	100.17 / 126.62	1.16 / 1.10	0.76 / 0.45
WO	90.95 / 145.36	1.05 / 1.00	1.55 / 0.69

Meanwhile, even with the more read-heavy test cases, PilotDB still outperforms the other memory disaggregation solutions. In particular, it achieves 90.2% of the MySQL-ideal throughput with RO, where it does not benefit from PM's non-volatile nature. With RW, PilotDB's advantage becomes more evident, especially when request concurrency increases.

Finally, the latency results correspond well with the throughput ones, with PilotDB offering very similar latency to that of MySQL-ideal, despite the latter's luxurious setup, with large DRAM LBP and local PM.

PM bandwidth consumption reduction. Table 1 gives more details from the above test on PM disaggregation, by listing throughput alongside PM bandwidth consumption for the 64-thread runs. It shows that PilotDB, while delivering 26.4% and 59.8% higher throughput over LegoPM, reduces remote PM write bandwidth by 40.8% and 55.5%, for both RW and WO workloads, respectively. As expected, PilotDB does not bring significant PM read bandwidth savings. Note that the read-only workload (RO) still has PM write traffic, due to page registration triggered by RBP misses (Section 4.1).

Remote CPU involvement reduction. Next, we examine the PMN CPU cycle consumption in PM disaggregation. Here PilotDB is tested against two baselines: LegoPM and PilotDB-RPC, a version of PilotDB with one-sided RDMA replaced by our optimized rRPC (performing similarly to X-RDMA [44]).

Figure 6 plots the impact on the Sysbench throughput with its RW and WO workloads, when we vary the number of cores allocated, out of the total 26 available on the PMN. PilotDB-RPC has a slightly lower peak throughput than PilotDB, but requires 16 cores to do so, for both RW and WO. LegoPM reaches its peak performance with fewer cores (around 8), though the peak itself is considerably lower, as shown in Figure 5. Still, it consumes much more PMN CPU resources than PilotDB, as its direct remote page write would still involve RPC for the PMN to read the page with atomicity guarantee.

With its aggressive use of one-sided RDMA, PilotDB offers high throughput with much lower CPU consumption, saturating at *only one core* for both RW and WO. With its log-only PM writes, PilotDB also avoids the aforementioned remote page write problem suffered by LegoPM. Given the large PM capacity per node, such low PMN CPU consumption makes PilotDB powerful, as a PMN could manage an enormous disaggregated PM space without its CPUs becoming a bottleneck.

Please note that we do not consider the PMN cores in setting up MySQL-ideal (Figure 5), so this baseline did use fewer cores overall. To be fair, we replicate the above MySQL-ideal experiments with 1 extra core (sufficient for PilotDB according to results here). This brings an up to 4% throughput boost across three Sysbench workloads, not significant enough to alter the earlier conclusions.

Breakdown analysis. We then check the impact of PilotDB's individual optimizations, such as data placement, log organization, and the *log-pull* mechanism. The first system ("LegoPM") moves WAL persistence from shared storage and hot pages from local DRAM to remote PMs, flushing both log entries and dirty pages via network. The latter three are variants of PilotDB, with optimizations incrementally added. "Log-offload" adapts LegoPM by only flushing log entries, and offloading log application to PMNs. "CDLog" further applies our proposed, CN-driven log organization. Finally, "PilotDB (w. log-pull)" is the complete PilotDB, fully exploiting one-sided RDMA with log-pull enabled. Figure 7 gives the results.

Table 2: Remote PM page read latency

Latency (μs)	Log-offload	CDLog	PilotDB (w. log-pull)
Median	178	23	15
Average	227	25	16
P99	872	48	30

Table 3: RBP stale read ratio w. 200 GB DB (WO, 64-thread)

	PilotDB-RPC	PilotDB
Stale ratio	2.2‰	2.5‰
Thpt (K-QPS)	16.35	144.55

Across workloads, "Log-offload" shows that simply offloading log application degrades performance, as opposed to directly flushing data pages to the PM (LegoPM). With RW and WO, it sees costly on-demand log application on the query-processing critical path, plus slow log application when CPU resources are tight. With PilotDB's new log organization, "CDLog" recovers the loss with these workloads, by its fast, light-weight PM-side log application, outperforming LegoPM by up to 19.7% and 62.8% for RW and WO, respectively. The complete PilotDB, "w. log-pull", outperforms LegoPM slightly for RO, and by 26.4%/80.0% higher for RW/WO.

To take a deeper look, we summarize the remote PM page read latency of the above three PilotDB variants in Table 2. Even with WO, all three PilotDB variants have substantial remote PM reads, since updating a page requires first loading it to LBP. When the requested page is up-to-date, "CDLog" and "PilotDB (w. log-pull)" observe low latency (the "Median" row), since no on-demand log application is needed. Here the full PilotDB is already superior, due to its fast reads with one-sided RDMA. In contrast, "Log-offload" has a median latency significantly higher, due to its resource-heavy conventional log application that contends for the remote CPU cycles with the PM-side page serving.

The average and P99 tail latency, meanwhile, are affected by the worst-case scenario, when the RBP-cached page is found stale. Here "Log-offload" suffers the most, waiting for its slow log application to reach the desired page. CDLog, while also waiting, has log application accelerated by its CN-driven log organization (with fast memory copy), resulting in an average/tail latency reduction from "Log-offload" of around 9/18×. The full PilotDB eliminates such wait altogether (along with the PMN-side version check) and pulls the needed log for CN-side update instead, trimming another 36%/60% from the "CDLog" average/tail latency.

Table 3 shows the benefit of log-pull, listing the ratio of RBP stale reads in a 64-thread WO test. With only 2 PMN cores allocated, PilotDB sees a slightly higher stale ratio than PilotDB-RPC, while delivering nearly 10× throughput. Furthermore, the low stale ratio (2.5‰) confirms the rationale of PilotDB's optimistic read for saving PMN CPU consumption.

Data consistency and recovery test. We check the data consistency of our implementation using highly concurrent workloads and failure injection. We run PilotDB alongside Pstress [42], a database concurrency and recovery testing tool, which generates concurrent workloads to stress PilotDB while logging the initial data set and all queries executed. We terminate PilotDB in the middle of a test, then reboot it and create a checkpoint to compare with the Pstress-generated state. We repeated such a test 50 times and found no inconsistency.

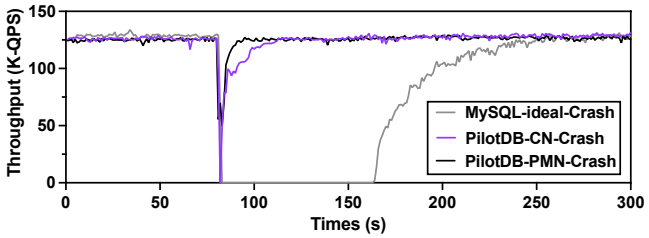

Figure 8: Recovery and warm-up behavior

To examine their recovery performance, we run a 64-thread Sysbench RW with both PilotDB and MySQL-ideal. We inject a crash at 80t second into the test and reboot the crashed instance immediately. For PilotDB, we configure two failure cases, crashing the CN and PMN, denoted as "PilotDB-CN-crash" and "PilotDB-PMN-crash", respectively.

Figure 8 plots throughput, showing both the recovery time (time to resume service) and warm-up time (time to return to pre-crash throughput). As expected, MySQL-ideal performs the worst, taking 81 seconds to recover states and another 87 seconds to populate its empty LBP. This is due to its WAL consisting of many committed transactions not yet applied to storage (local PM in this case), which requires log scan and reconstructing the in-memory checkpoint. In addition, with its large yet volatile buffer pool, all pages are lost and need to be fetched from the PM-based local storage.

In contrast, PilotDB-CN-crash excels with smaller CN footprint, taking nearly no time to recover and only 35 seconds to warm up. One reason is that PilotDB persists logs to the remote PM at the mini-transaction basis. Upon reboot, it just needs to rollback a small number of unfinished transactions via one-sided RDMA. The other important factor is its small LBP size.

Similarly, PilotDB-PMN-crash experiences zero recovery time, here thanks to the three-way log replication. With one PMN down, the entire database service immediately fails over to another alive PMN replica (appointing it the primary) and continues to process client queries. As expected, PilotDB-PMN-crash needs only 11 seconds to recover to its peak throughput, as the RBP pages are not lost after reboot and can directly serve CNs.

Such results reveal that besides significant performance advantage and dramatically reduced CN-side DRAM requirement, PM-disaggregation enables speedy recovery and warm-up, both crucial to performance resilience.

Multi-tenant support. Figure 9 showcases PilotDB's capability to serve multiple DB applications with its disaggregated PM layer, compared with LegoPM. Here each of the 4 PM-equipped Group-1 servers uses one of its CPU sockets for PMN and the other for CN, forming the first CN pool (CNP1). The second pool (CNP2) is formed by 8 Group-2 servers, which are in a different rack, connected to the PMNs via a slower 10 Gbps connection. With both systems, each tenant has a 50 GB DB, running Sysbench RW using 16 threads, with 5 GB DRAM LBP on CNP1 and 30 GB DRAM on CNP2, plus a 50 GB remote PM buffer. Here we adjust LBP provisioning: users

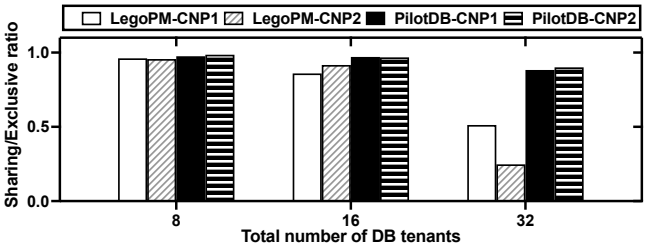

Figure 9: Multi-tenant runs (Sysbench RW, 16 threads)

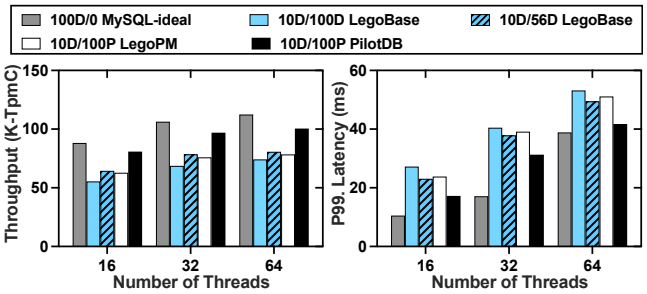

Figure 10: TPC-C performance w. 200 GB DB

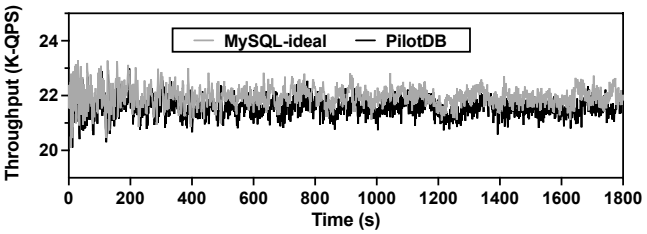

Figure 11: Throughput timeline of production workload

renting CNs further away from the PMs might be encouraged to increase their LBP size due to higher cross-rack latency and lower bandwidth. Both PilotDB and LegoPM turn off log replication in this experiment.

We test two execution modes: (1) "exclusive", where a DB tenant runs individually (with no concurrent workloads), and (2) "sharing", where multiple DB instances, evenly distributed across CNP1 and CNP2, run simultaneously. Figure 9 plots the average per-instance "sharing" throughput within each CN pool, *relative to the respective "exclusive" baseline*, while increasing the total number of tenants.

With 8 or 16 tenants, both LegoPM and PilotDB in sharing mode perform well (achieving around 90% of exclusive throughput) due to sufficient PMN resources. With 32 tenants, LegoPM instances in CNP1 (LegoPM-CNP1) decline to around half of the exclusive performance, seeing intense contention for PMN CPU. LegoPM-CNP2 degrades even more, constrained by the network bottleneck. PilotDB, in contrast, shows a very slight performance drop serving 32 tenants. This implies that with PilotDB's bandwidth- and PM-side CPU-conserving design, a disaggregated PM layer is promising in supporting more concurrent DB instances.

Impacts of DB size and PM ratio. Finally, we also assess the impact of DB size and PM ratio by fixing the memory allocation but expanding the database, producing a decline in PM ratios. With 400 GB DB and 100 GB PM buffer, PilotDB's throughput reaches 88.5%, 92.3%, and 94.5% of "MySQL-ideal" using 100GB local DRAM buffer for RO, RW, and WO workloads, respectively. In contrast to 90.1%, 94.4%, and 97.0% with 200 GB DB and 100 GB PM buffer, the performance is inferior due to the reduced LBP hit ratio. Furthermore, when expanding the DB size to 1 TB, disk I/O becomes a bottleneck. Still, PilotDB's throughput reaches 86.9%, 89.1%, and 91.5% of "MySQL-ideal" with the three workloads. To conclude, with significantly reduced PM ratios, PilotDB achieves slightly worse but still comparable performance as the best-performing baseline, thanks to its fast RDMA-friendly page access path.

5.3 Other Workloads

TPC-C results. Figure 10 shows the TPC-C throughput and P99 latency results, giving similar trends as with Sysbench. With LBP limited at 5% of the 200 GB DB size and a 100 GB PM RBP, PilotDB produces throughput only 10.6% lower than MySQL-ideal in the worst case. Meanwhile, it achieves a throughput 35-46% higher than LegoBase, and 27-29% higher than LegoPM. Its P99 tail latency, on the other hand, is up to 64% higher than MySQL-ideal when fewer threads are used, as its distributed operations bring higher lock contention on the CN. The tail latency difference appears to diminish as thread concurrency grows.

Production workload. Finally we run a trading service workload from the production environment of Alibaba Cloud. It is memory-intensive and contains a significant portion of write transactions. Figure 11 plots the throughput timelines of MySQL-ideal and PilotDB. Despite the inevitable cross-node traffic between the CN and PMNs, PilotDB still nearly matches MySQL-ideal's performance (98.0% on average), without bringing larger throughput variance. This demonstrates that e-commerce applications could indeed run at much lower DRAM configurations when supported by a disaggregated PM layer.

5.4 Cost Comparison

With our performance results, here we assess the cost-effectiveness of PM disaggregation for cloud-native DBs by examining the request processing throughput per $ investment.

We compare the three setups, MySQL-ideal, LegoBase, and PilotDB. Table 4 lists their resource usage for Sysbench RW and TPC-C, both with 100 GB DB size. Aside from the familiar LBP/RBP configurations, we use the minimum remote memory node core counts that achieve each system's peak throughput (with CN core count fixed at 16). For example, MySQL-ideal still needs the "H" CPUs as it uses a local PM for log and data storage, while LegoBase can adopt the "L" CPUs for both its CN and remote memory node (saturating at 8 cores on the latter for Sysbench and 16 for TPC-C). PilotDB, in contrast, requires "L" CPUs for CN and "H" CPUs for PMN, but saturates at only 1 PMN core for both workloads. With the unit prices listed, we derive the per-resource costs, which sum up the total cost and produce the throughput per $.

With both workloads, PilotDB outperforms the other two systems in terms of cost-effectiveness by at least 55.9%. In particular,

Table 4: Cost analysis, with hardware model/price give on the left (detailed data for Sysbench omitted, CDW-G [41] as of 02/2023)

Hardware	Unit Price		Resource	MySQL-ideal (100D-0)		LegoBase (10D-100D)		PilotDB (10D-100P)	
				Usage	Cost	Usage	Cost	Usage	Cost
Intel Platinum 8260 CPU PM-compatible (H)	$6880.0	TPC-C	CN CPU	H-16	$2293.3	L-16	$929.1	L-16	$929.1
			MN CPU	0	$0	L-16	$929.1	H-2	$286.7
Intel Xeon E5-2660 CPU non-PM-compatible (L)	$1626.0		DRAM	100 GB	$685.2	110 GB	$753.7	10 GB	$68.5
			PM	100 GB	$327.3	0 GB	$0	100 GB	$327.3
Samsung DDR4 (128 GB)	$877.0		Total Cost	-	$3305.8	-	$2611.9	-	$1611.6
			Thpt (TpmC)	119.4 K	-	95.2 K	-	112.4 K	-
Intel Optane DCPMM (128 GB)	$419.0		**Thpt/$ (TpmC/$)**	-	**36.1**	-	**36.4**	-	**69.7**
		Sysbench	**Thpt/$ (QPS/$)**	-	**41.8**	-	**59.2**	-	**92.3**

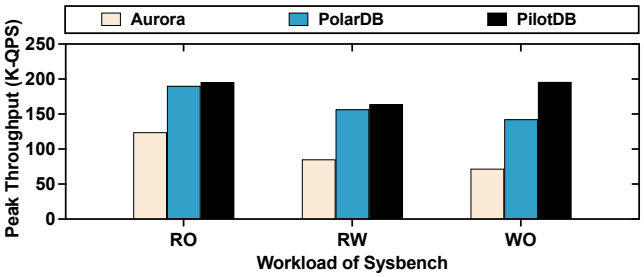

Figure 12: PilotDB vs. public cloud-native DBs (200 GB DB)

with Sysbench, it more than doubles the throughput per $ investment of MySQL-ideal. Compared to DRAM-disaggregation with LegoBase (with the same amount of remote memory, but in DRAM), PilotDB improves both absolute performance and cost-effectiveness.

5.5 Comparison with Disaggregated Cloud-Native DBs

We also compare PilotDB with two popular cloud-native relational DBs on the market: Amazon's Aurora [5] and Alibaba's PolarDB [4], both with storage disaggregation. Given the black-box nature of their hardware configuration, we make our best effort to align testbed resources. When in doubt, we make conservative configurations that are biased *against* PilotDB. CPU remains the most important resource and is straightforward to align in quantity: we allocate 16 cores to each system, and use MySQL's built-in MD5 function to verify that the three setups have similar CPU capabilities. Furthermore, with their resource provisioning policy, such CPU allocation comes with 128 GB DRAM for both Aurora and PolarDB. This way, PilotDB's 10D/100P LBP/RBP configuration has no capacity or speed advantage. We compare the peak throughput of the Sysbench workloads between the three system configurations, using a 200 GB database.

As seen in Figure 12, PilotDB brings a 1.02×, 1.05×, and 1.38× peak throughput improvement over PolarDB and 1.58×, 1.92×, and 2.72× over Aurora in Sysbench RO, RW, and WO workloads, respectively. Overall, PolarDB achieves significantly higher throughput than Aurora (close to the MySQL-ideal baseline above), where the working set of both systems can be cached in its 128 GB local buffer, despite the use of disaggregated storage. We observed from the AWS performance monitor that Aurora has a memory hit ratio of 99% and almost no storage I/O accesses, while its CPU utilization is nearly 100%. We suspect Aurora's lower performance is due to its compute module (*i.e.*, the SQL/TXN engine) design.

Though replacing the bulk of DRAM allocation with disaggregated PM, for read-dominant workloads like RO and RW, PilotDB still matches or slightly outperforms PolarDB, due to its highly-optimized read paths driven by one-sided RDMA and CN-side version checks. Finally, when it comes to write-only (WO), PilotDB shows a considerable advantage over PolarDB, thanks to its fast data persistence.

6 DISCUSSION: APPLICABILITY BEYOND OPTANE

Though our evaluation focuses on the Optane PM, PilotDB's application scope is much wider and is unlikely to be impacted by the discontinuation of the 3D XPoint [29] product. First, the NVM business will continue, as multiple companies are developing products based on different technologies such as STT-MRAM [8], FRAM [36], Nano-RAM [56], and ReRAM [2]. Second, as a promising alternative, the emerging CXL-based storage/memory approximates PM by providing DRAM with flash-based durability [22, 61]. For instance, KIOXIA's XL-FLASH with the CXL.mem interface [22] is a fast SSD with 64-byte random access and persistence. With products like XL-FLASH, PilotDB will retain its relevance as they provide what Optane offers: fine-grained access granularity, ultra-low latency RDMA compatibility, fast persistence, and large capacity. Finally, PilotDB's target problems (such as limited bandwidth and high CPU loads for data durability and availability) likely persist across different NVM technologies, hence its design will remain valid.

Regarding generalization, PilotDB is a PM-disaggregation solution customized for cloud-native relational DBs and demonstrated on MySQL, one of the most used DB implementation. DBs such as PostgreSQL [27] can be easily adapted to use PilotDB techniques in a similar way. It is also possible to encapsulate PilotDB's PMNs into a general-purpose disaggregated PM layer with proper APIs. Finally, PilotDB has the potential to offer cost-effective caching or logging functionalities for various data-intensive applications that assume page-based memory/storage organization [1, 28, 66].

7 RELATED WORK

Below we summarize related work not discussed in Section 2 and note their relationship with PilotDB.

PM for relational DB. Recent advances in commodity PM triggered a large body of initial proposals to incorporate it into relational DBs for performance, sometimes with architecture redesign [6, 9, 30, 34, 37, 40, 46, 51, 52, 64, 67, 68, 77].

A group of the above studies uses PM as an extension to DRAM or a faster SSD, harvesting the PM's byte-addressable performance and capacity benefits. For instance, the SAP HANA in-memory database system [6] partitions and places data on DRAM and PM. FOEDUS [37] leverages a two-tier DRAM-PM approach that executes transactions entirely on DRAM, while persisting states on PM asynchronously. Hymem [64] and Spitfire [77] adopt a three-tier DRAM-PM-SSD hierarchy, focusing primarily on managing data migrations across tiers. Other three-tier systems like Exadata [51] and HiEngine [46] attach PM to either the storage server or the compute node for caching hot data or log entries. Most of these design targets a single machine, while all of them only consider PM as locally attached resource. PilotDB also adopts the cost-effective three-tier DRAM-PM-SSD hierarchy, but designs it for having *fully disaggregated resource layers.*

Another group of work uses PM to improve database reliability. For example, NV-Logging [30] and Distributed-Logging [67] build concurrent logging on PM. SOFORT [52] proposes a copy-on-write architecture for PM storage, which enables direct data modification on PM and almost instantaneous restart after a crash. Zen [40] eliminates logging by reserving fields for data tuples in PM to indicate their persistence status. WBL [9] abandons the traditional write-ahead logging and instead introduces a protocol that first persists data into PM, then records light-weight log entries without the before- and after-images of data. However, these systems are again designed for single-machine execution or suffer high overhead in distributed environments [9, 68]. In contrast, PilotDB is designed specifically for scalable cloud-native services, retaining existing relational DB workflow while offering fast, light-weight PM-side operations through the heavy adoption of one-sided RDMA.

PM in distributed or disaggregated environments. A few related studies combine PM and RDMA to build fast and persistent remote storage or memory systems [43, 45, 60, 63, 70, 76]. Examples include Octopus [43] and Orion [70], both distributed, PM-based file systems. They unfortunately do not meet the demands of cloud-native databases due to the file system abstraction adding excessive performance overhead in accessing remote PM [60]. In addition, Mojim [76] offers a primary-backup replicated storage system with a memory-based abstraction. To replace the bulk SSD/HDD-based storage with such PM-based solutions for cloud-native DBs, however, appears to be beyond the budget of most users.

Hotpot [60] extends Mojim's design to distributed shared persistent memory, unifying data caching and replication into a single layer. However, its design targets symmetric distributed PM systems, where each machine has its own PM device, in contrast to elastically provisioned resources favored by cloud-native databases. Second, they require full replication of data structures in local memory, which is less affordable for cloud-native DBs due to their large data volume. Third, both Mojim and Hotpot offer a transparent distributed PM abstraction to upper-level applications in the Linux kernel. In exchange for such transparency, kernel-based remote memory systems tend to have high overheads, despite the use of RDMA [73, 74].

Recent systems do depart from the symmetric PM architecture and follow the resource disaggregation trend [58] by decoupling compute nodes from the PM layer [45, 63, 72]. Besides the previously discussed AsymNVM [45] and pDPM [63], FORD [72] optimizes distributed transaction processing with the one-sided RDMA and batching techniques [33].

The primary distinctions between PilotDB and the above systems are (1) PilotDB targets a disaggregated three-tier DB design prompted by the overall cost-effectiveness offered by combining multiple features of PM: capacity, performance, and persistence; (2) instead of offering general disaggregated PM, PilotDB is designed for cloud-native databases and incorporates a substantial amount of DB-specific optimizations, to fully unleash the potential of PM for cloud DB services.

Other work. Our CDLog design is inspired by existing log offloading approaches such as "log is the database" [65, 68]. However, the PilotDB PMNs work very differently, designed to possess a CPU-light data-plane. They perform easy and fast log application and replication, both *out of the query critical paths*, with PM-buffered pages retrieved, checked, and updated single-handedly by the compute nodes. Finally, PilotDB has benefited from the design advice by a recent comprehensive and systematic study [69] of remote PM characteristics. For example, we adopted the suggested RDMA batching optimization, which combines the remote persistence and writes operations into one RDMA request to reduce latency.

8 CONCLUSION

Merely five years ago, a SIGMOD '18 paper [64] investigating single-node DRAM-PM-SSD architecture for databases (using emulated PM in experiments) concurred with the conventional wisdom that NVM-DIMMs are "not fast enough to replace main memory and they are not cheap enough to replace disks, and they are not cheap enough to replace flash." [49]. While we also agree that PM is not going to replace main memory or SSD/HDD storage, this study has demonstrated that today's commodity PM hardware is large enough to help reduce memory need (hence facilitating the utilization of fragmented CPU core resources), and fast enough to be used for on-critical-path data persistence. With hardware disaggregation, PM resources can be consolidated into a PM layer that serves concurrent database instances with diverse access patterns and load levels, enhancing the overall cost-effectiveness and elasticity of cloud-native database services.

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their insightful feedback. This work is supported in part by the National Natural Science Foundation of China under Grant No.: 62141216, 62172382 and 61832011, the USTC Research Funds of the Double First-Class Initiative under Grant No.: YD2150002006 and Alibaba Group through Alibaba Innovative Research Program. Cheng Li is the corresponding author.

REFERENCES

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote Regions:

a Simple Abstraction for Remote Memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 775–787.
[2] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.
[3] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.
[4] Alibaba. 2022. PolarDB. https://www.alibabacloud.com/product/polardb. "[accessed-Feb-2023]".
[5] Amazon. 2022. Aurora. https://aws.amazon.com/rds/aurora. "[accessed-Feb-2023]".
[6] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA Adoption of Non-Volatile Memory. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1754–1765. https://doi.org/10.14778/3137765.3137780
[7] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.
[8] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 9, 2 (2013), 1–35.
[9] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind Logging. *Proceedings of the VLDB Endowment* 10, 4 (2016), 337–348.
[10] Anonymous authors. 2021. Issue about Hotpot Running Instructions. https://github.com/WukLab/Hotpot/issues/8. "[accessed-Feb-2023]".
[11] TPC Benchmark. 2022. TPC-C. http://www.tpc.org/tpcc/. "[accessed-Feb-2023]".
[12] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 301–316.
[13] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. 2021. LogStore: A Cloud-Native and Multi-Tenant Log Database. In *SIGMOD*.
[14] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. PolarDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 29–41.
[15] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.
[16] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.
[17] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 239–252. https://www.usenix.org/conference/fast20/presentation/chen-jiqiang
[18] SPDK Community. 2022. Storage Performance Development Kit. https://spdk.io/. "[accessed-Feb-2023]".
[19] Intel Corporation. 2020. Persistent Memory Provisioning Introduction. https://software.intel.com/content/www/us/en/develop/articles/qsg-intro-to-provisioning-pmem.html. "[accessed-Feb-2023]".
[20] Intel Corporation. 2022. 3D XPoint™: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html. "[accessed-Feb-2023]".
[21] KIOXIA Corporation. 2022. What is the 3D Flash Memory BiCS FLASH? https://www.kioxia.com/en-jp/rd/technology/bics-flash.html. "[accessed-Feb-2023]".
[22] Kioxia Corporation. 2022. XL-FLASH Storage Class Memory Solution. https://www.kioxia.com/en-jp/business/news/2022/20220802-1.html. "[accessed-Feb-2023]".
[23] Oracle Corporation. 2022. MySQL. https://www.mysql.com/. "[accessed-Feb-2023]".
[24] DB-Engines. 2022. https://db-engines.com/en/ranking_categories. "[accessed-Feb-2023]".
[25] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus database: How to be fast, available, and frugal in the cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1463–1478.
[26] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.
[27] The PostgreSQL Global Development Group. 2022. PostgreSQL. https://www.postgresql.org/. "[accessed-Feb-2023]".
[28] 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.
[29] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform storage performance with 3D XPoint technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.
[30] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment* 8, 4 (2014), 389–400.
[31] Intel. 2019. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html. "[accessed-Feb-2023]".
[32] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).
[33] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 402–416.
[34] DAVID COHEN JOY ARULRAJ. 2020. Leveraging Persistent Memory in Cloud-native Database Systems. https://pirl.nvsl.io/2020/02/11/leveraging-persistent-memory-in-cloud-native-database-systems/. "[accessed-Feb-2023]".
[35] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 437–450.
[36] Yoshihisa Kato, Yukihiro Kaneko, Hiroyuki Tanaka, Kazuhiro Kaibara, Shinzo Koyama, Kazunori Isogai, Takayoshi Yamada, and Yasuhiro Shimada. 2007. Overview and future challenge of ferroelectric random access memory technologies. *Japanese Journal of Applied Physics* 46, 4S (2007), 2157.
[37] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 691–706.
[38] Alexey Kopytov. 2022. Sysbench. https://github.com/akopytov/sysbench. "[accessed-Feb-2023]".
[39] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, and Ricardo Bianchini. 2022. First-generation Memory Disaggregation for Cloud Platforms. *arXiv preprint arXiv:2203.00241* (2022).
[40] Gang Liu, Leying Chen, and Shimin Chen. 2021. Zen: a High-throughput Log-free OLTP Engine for Non-volatile Main Memory. *Proceedings of the VLDB Endowment* 14, 5 (2021), 835–848.
[41] CDW LLC. 2021. CDW-G. https://www.cdw.com. "[accessed-Feb-2023]".
[42] Percona LLC. 2022. Pstress: database concurrency and crash recovery testing tool. https://www.percona.com/blog/2020/04/15/pstress-database-concurrency-and-crash-recovery-testing-tool/. "[accessed-Feb-2023]".
[43] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 773–785.
[44] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. 2019. X-rdma: Effective rdma middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.
[45] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: an Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 757–773.
[46] Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, and King Lv. 2022. HiEngine: How to Architect a Cloud-Native Memory-Optimized Database Engine. In *Proceedings of the 2022 International Conference on Management of Data*. 2177–2190.
[47] John C. McCallum. 2022. Memory Prices 1957+. https://jcmit.net/memoryprice.htm. "[accessed-Feb-2023]".
[48] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[49] Timothy Prickett Morgan. 2017. How Hardware Drives the Shape of Databases to Come. https://www.nextplatform.com/2017/08/15/hardware-drives-shape-databases-come/. "[accessed-Feb-2023]".
[50] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 29–41.
[51] Oracle. 2020. https://blogs.oracle.com/exadata/post/persistent-memory-in-exadata-x8m. "[accessed-Feb-2023]".
[52] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A Hybrid SCM-SDRAM Storage Engine for Fast Data Recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*. 1–7.
[53] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2010. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.* 43, 4 (jan 2010), 92–105. https://doi.org/10.1145/1713254.1713276
[54] Percona-Lab. 2021. TPCC Repository by Percona-Lab. https://github.com/Percona-Lab/tpcc-mysql. "[accessed-Feb-2023]".
[55] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (2008), 465–479.
[56] Thomas Rueckes. 2011. High density, high reliability carbon nanotube NRAM. In *Flash Memory Summit*.
[57] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *ICDE*.
[58] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.
[59] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610
[60] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing*. 323–337.
[61] Anton Shilov. 2022. Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift. "[accessed-Feb-2023]".
[62] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 255–270.
[63] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: an Exploration of Passive Disaggregated Key-value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 33–48.
[64] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*. 1541–1555.
[65] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.
[66] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 261–280.
[67] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876.
[68] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proceedings of the VLDB Endowment* 11, 4 (2017), 406–419.
[69] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*.
[70] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: a Distributed File System for Non-volatile Main Memory and RDMA-capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 221–234.
[71] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang
[72] 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 21)*. *USENIX Association*.
[73] 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. *Proc. VLDB Endow.* 13, 9 (May 2020), 1568–1581. https://doi.org/10.14778/3397230.3397249
[74] 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.
[75] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10.
[76] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: a Reliable and Highly-available Non-volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.
[77] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-tier Buffer Manager for Volatile and Non-volatile Memory. In *Proceedings of the 2021 International Conference on Management of Data*. 2195–2207.

Received 2022-10-20; accepted 2023-01-19