# Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation

Yingqiang Zhang[‡], Chaoyi Ruan[‡,†], Cheng Li[†], Xinjun Yang[‡], Wei Cao[‡], Feifei Li[‡], Bo Wang[‡], Jing Fang[‡], Yuhui Wang[‡], Jingze Huo[‡,†], Chao Bi[‡,†]
yingqiang.zyq@alibaba-inc.com, rcy@mail.ustc.edu.cn, chengli7@ustc.edu.cn, {xinjun.y, mingsong.cw, lifeifei, xiangluo.wb, hangfeng.fj, yuhui.wyh, jingze.hjz}@alibaba-inc.com, bc233333@mail.ustc.edu.cn
[‡]Alibaba Group and [†]University of Science and Technology of China

## ABSTRACT

It is challenging for cloud-native relational databases to meet the ever-increasing needs of scaling compute and memory resources independently and elastically. The recent emergence of memory disaggregation architecture, relying on high-speed RDMA network, offers opportunities to build cost-effective and elastic cloud-native databases. There exist proposals to let unmodified applications run transparently on disaggregated systems. However, running relational database kernel atop such proposals experiences notable performance degradation and time-consuming failure recovery, offsetting the benefits of disaggregation.

To address these challenges, in this paper, we propose a novel database architecture called LegoBase, which explores the co-design of database kernel and memory disaggregation. It pushes the memory management back to the database layer for bypassing the Linux I/O stack and re-using or designing (remote) memory access optimizations with an understanding of data access patterns. LegoBase further splits the conventional ARIES fault tolerance protocol to independently handle the local and remote memory failures for fast recovery of compute instances. We implemented LegoBase atop MySQL. We compare LegoBase against MySQL running on a standalone machine and the state-of-the-art disaggregation proposal Infiniswap. Our evaluation shows that even with a large fraction of data placed on the remote memory, LegoBase's system performance in terms of throughput (up to 9.41% drop) and P99 latency (up to 11.58% increase) is comparable to the monolithic MySQL setup, and significantly outperforms (1.99×-2.33×, respectively) the deployment of MySQL over Infiniswap. Meanwhile, LegoBase introduces an up to 3.87× and 5.48× speedup of the recovery and warm-up time, respectively, over the monolithic MySQL and MySQL over Infiniswap, when handling failures or planned re-configurations.

**PVLDB Reference Format:**
Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, Chao Bi. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. PVLDB, 14(10): 1900 - 1912, 2021.
doi:10.14778/3467861.3467877

Proceedings of the VLDB Endowment, Vol. 14, No. 10 ISSN 2150-8097.
doi:10.14778/3467861.3467877

*Yingqiang Zhang and Chaoyi Ruan equally contributed to this work.

## 1 INTRODUCTION

With the increasing migration of applications from on-premise data centers to clouds, cloud-native relational databases have become a pivotal technique for cloud vendors. By leveraging modern cloud infrastructures, they provide equivalent or superior performance to traditional databases at a lower cost and higher elasticity. As a consequence, in recent years, major cloud vendors have launched their own cloud-native databases, such as Amazon Aurora[42], Azure Hyperscale [15, 30], Google Spanner [14] and Alibaba PolarDB[9, 10].

However, even the state-of-the-art cloud-native databases still embrace the monolithic server architecture where CPU and memory are tightly coupled. This makes it hard to fulfill the ever-growing and highly elastic resource demands from web-scale applications [36, 38]. For example, analytic queries favor a large amount of memory, which may exceed the capacity of a single machine, and would experience significant performance degradation when the working set does not fit into memory. In addition, the CPU utilization of database instances could be low most of the time [11, 32], but occasionally may reach high levels when handling bursty traffic , e.g., a promotion event. With the monolithic architecture, scaling down/up the memory or CPU resources after provisioning them in a server may cause resource fragmentation or failure [20, 43]. Furthermore, since the memory resource occupied by a single instance cannot be allocated across server boundaries, the unused and available memory resource scattered across the cluster are not easily harvested and utilized, resulting in a waste of resources. For example, one of the Alibaba clusters' memory utilization varies between 5%–60% over 80% of its running time [11].

With the rapid evolution of networking techniques, there is great potential for building a cost-effective and elastic cloud-native database based on memory disaggregation. Most of the recent proposals such as LegoOS [40] and Infiniswap [22] are implemented in the operating system layer, hiding accesses to remote memory resource within interfaces like virtual block device interfaces and virtual memory mapping, and expose them transparently to unmodified memory-intensive applications. In this paper, we pay attention to apply these general-purpose proposals to support databases and understand their performance implications. Interestingly, the integration results in a significant performance loss. This performance gap stems from two factors: (1) every remote page access needs to go through the full Linux I/O stack, which takes nearly an order of magnitude longer than the network latency; and (2) the database-oblivious memory management (e.g. cache eviction) neglects the unique data access patterns of database workloads, resulting in significantly higher cache miss ratio.

Last but not least, another advantage of memory disaggregation is to enable fine-grained failure handling of CPU and memory resources, since they are decoupled and disseminated, and will not fail at the same time. However, the database kernel is still monolithic and cannot use the advantage of disaggregation to improve availability. In particular, the current fault tolerance mechanism prevents from handling failures of local and remote memory independently, and leads to a time-consuming failure recovery.

To address the above problems, we propose LegoBase, a novel cloud-native database architecture to exploit the full potential of memory disaggregation, with the following contributions:

First, we advocate to co-design database kernel with disaggregation. Unlike general purpose solutions that emphasize transparency for database, we instead extend the buffer pool management module in database kernel to be aware of remote memory, which could entirely bypass the cumbersome Linux I/O stack. Furthermore, it allows us to explore optimization opportunities brought by tracking data access patterns for database workloads, and utilize well studied optimizations that already exist in database kernel, such as the LRU mechanism for page management in the buffer pool.

Second, we propose a novel *two-tier ARIES* protocol to handle failures of compute nodes and the remote memory, independently. Since the compute node and the remote memory are unlikely to fail at the same time, we can use the data cached in the remote memory to accelerate the reboot process of the compute node. This protocol evolves the traditional ARIES algorithm by tiering the operations of flushing dirty page and creating checkpoint into two layers. The compute node treats the remote buffer as a persistence layer, and flushes dirty pages and creates *first-tier* checkpoints to it at a high frequency. Furthermore, we offload the duty of ensuring data persistence to the remote memory, which directly flushes dirty pages to underlying storage and create *second-tier* checkpoints.

Finally, we implement a prototype system of LegoBase on the codebase of MySQL. We intensively evaluate LegoBase with TPC-C, TPC-H, Sysbench and a production workload. Our evaluation results show that even with a large fraction of data placed in the remote memory, LegoBase's system performance in terms of throughput and latency is comparable to the monolithic MySQL setup, and significantly outperforms running MySQL over Infiniswap. Meanwhile, the recovery and warm-up time for LegoBase is much faster than both the monolithic MySQL and MySQL over Infiniswap, during failure handling or elastic re-configurations.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Cloud-native Relational Databases

A number of cloud-native database systems have been proposed and designed, for example, Amazon Aurora[42], Microsoft Azure SQL Database [15, 30], Google Spanner [14] and Alibaba PolarDB[9, 10]. They offer similar interfaces as the traditional ones but with significant innovations to leverage the modern cloud infrastructures for better performance, scalability, availability and elasticity [31, 41]. Thus, they have become the building blocks for hosting data and serving queries for a wide range of applications such as e-payment, e-commerce, online education, and gaming.

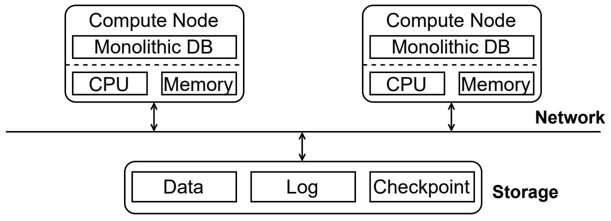

**Figure 1: Architecture of cloud-native relational databases.**

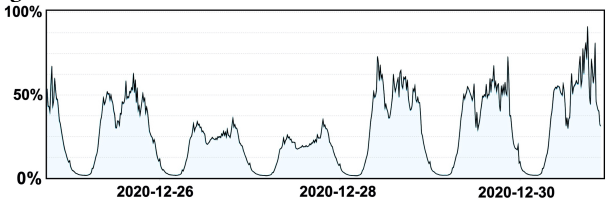

**Figure 2: The varied CPU utilization of a representative Alibaba workload within 7 days.**

Figure 1 illustrates the architecture of a typical cloud-native relational database. It consists of two major system components: (1) a scale-out persistent storage layer hosts the actual data and write-ahead logs for achieving fault tolerance, which is often a distributed file system or object store; and (2) a compute instance running in a container, which still includes most of the software modules of a traditional database kernel such as SQL execution engine, transaction manager, recovery daemon, etc. To offer fast responses, the compute instance caches data loaded from the storage layer in its local memory buffer as much as possible.

### 2.2 Elastic Resource Demands

A large number of applications relying on databases are memory-intensive. Our study and other work [2, 22] identified that they favor a large amount of memory allocation and suffer from significant performance loss when their working sets cannot fit into memory. In addition, their CPU utilizations vary a lot from time to time. Figure 2 shows the 7-day CPU usage ratios of a representative workload from Alibaba cloud with high variation. Most of the time, the CPU utilization is below 50% or even lower. However, it can reach up to 91.27% at peak for short time periods.

Customers who pay for the use of cloud-native databases would expect *an elastic deployment*, which enables to scale up/down memory and CPU resources independently for achieving sustained high throughput and low latency when adapting to workload changes under their monetary constraints. Similarly, cloud vendors also appreciate *flexible resource allocation* to better meet various customers' requirements, improve hardware utilization, and reduce hardware investment. However, the current monolithic server-centric resource allocation and software design pattern are not a good fit for the fast-changing cloud hardware, software, and cost needs.

### 2.3 Resource Disaggregation

With the rapid development of scalable and fast network technologies such as RDMA, resource disaggregation, where hardware resources in traditional server are distributed into network-connected isolated components, is a promising solution to meet the elastic resource allocation demands and has started to gain wide support and

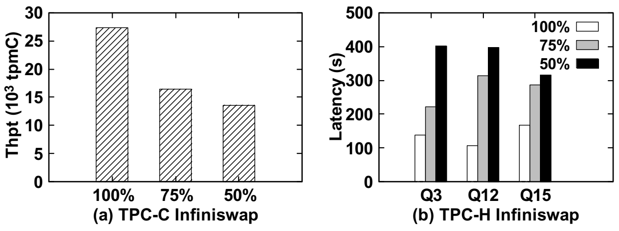

**Figure 3: Performance numbers of TPC-C and TPC-H achieved by running MySQL over Infiniswap with its working set in memory (possibly spanning across local and remote buffer). Here, we have three memory configurations with 100%, 75% and 50% local memory buffer ratios. For TPC-C, we set the warehouse parameter to 200 and use a 20GB dataset, while setting the scale factor to 30 and populating the database with 40GB data for TPC-H.**

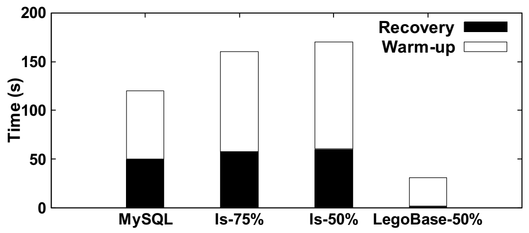

**Figure 4: The recovery performance comparison when rebooting a MySQL instance with TPC-C atop three setups, namely, the monolithic architecture, Infiniswap (Is), and our system LegoBase. The database size is 50GB.**

popularity [3, 19, 40]. Here, we focus on memory disaggregation, which is in high demand for memory intensive database applications and the start-of-the-art cloud-native databases already decouple storage from computation. Recent proposals like Infiniswap[22] and Leap [2] fall into this category and expose a remote memory paging system to unmodified applications, or some proposals like LegoOS [40] design a new OS architecture for hardware resource disaggregation. However, when applying the start-of-the-art memory disaggregated architectures to the database context, we find that they fail to achieve promising benefits in terms of performance gains and elastic deployment.

First, to understand the performance bottlenecks, we run MySQL over Infiniswap with two widely used workloads TPC-C and TPC-H. Figure 3 summarizes the impact of the local buffer size on throughput and query latency of the two workloads. Across all memory configurations, memory space may span over local and remote buffer but the working set of the corresponding workload can always fit into memory. We observe that moving more data from local buffer to remote memory has negative impacts on the overall performance. For instance, with regard to TPC-C, we observe a 39.5% drop in system throughput when allocating 25% remote memory, compared to the monolithic architecture (denoted by "100%"), where all memory space is local. Furthermore, decreasing the local memory ratio from 75% to 50%, the performance becomes worse and is only 49.8% of the best performing monolithic architecture. Similar performance degradation trends are observed for TPC-H. Here, we report the latency comparison of three selective queries. Overall, the query latency significantly increases as the local memory ratio decreases. For instance, the query latencies of the "75%" and "50%" memory allocation are 1.59 - 2.93 × and 1.87 - 3.72 × of that of "100%", respectively.

The performance inefficiencies are mainly contributed by the ignorance of the unique characteristics of a database system with a general and transparent memory disaggregated architecture. First, to avoid introducing significant changes to hosting applications, Infiniswap manifests itself as a Linux kernel module. However, this transparent design imposes extra high overhead due to its slow data path. For example, accessing a remote 4KB page in Infiniswap takes roughly 40 $\mu s$ [2], while the latency of a one-sided RDMA 4KB write/read operation takes only 4-6 $\mu s$. Second, Infiniswap achieves low cache hit ratios and introduces a significant number of RDMA operations, which are still much slower than local memory accesses. The reasons are three-fold: (1) it is not aware of data access patterns inside a database kernel, and may place in the remote buffer temporary or client session related data that occupy a small fraction of memory space but are frequently visited; (2) we find that the OS LRU cache algorithms are less effective to keep hot data locally and find candidates for page eviction, compared to the highly-optimized ones used by databases. However, the general solution sitting in OS cannot use these advanced algorithms; and (3) Virtual memory in OS exposes fixed-sized pages to applications (typically 4KB). However, database data exhibits variable page sizes ranging between 2KB to 16KB. It would cause multiple RDMA operations for a single database page access. Throughout our measurement, the RDMA operation latency is not proportional to the page sizes, e.g., with the 25Gbps RDMA, accessing a 16KB remote page takes 11.2 $\mu s$, only 30% higher than that of an 8KB page.

Apart from the performance issue, though disaggregated resources like CPU and memory are the unit of independent allocation and failure handling by concept, the state-of-the-art solutions fail to provide fast responses to failures or planned hardware operations, which are considered to be more important when running mission-critical and memory-intensive applications. To understand this, we crash a MySQL instance atop Infiniswap with the TPC-C workload and different remote memory ratios when its system performance becomes stable, and measure the fail-over time costs. We measure the recovery time corresponding to the process to restore the inconsistency states right before crashes to the latest checkpoint. We also measure the warm-up time, which corresponds to the time spent in populating all local and remote memory buffers. We also compare the time costs against the monolithic architecture and our proposal LegoBase. In Figure 4, both monolithic architecture and Infiniswap take a long time to recover inconsistent states, e.g., 160 seconds. Similarly, they also experience a much longer warm-up time, e.g., 50-61 seconds. However, Infiniswap introduces a 45.7%-55.7% increase in the warm-up time. This is because there are a significant number of RDMA operations for filling in the remote memory buffer with memory pages that need to be swapped out from the local memory buffer (which is already full). Again, the higher remote memory ratio leads to a longer warm-up time. Finally, unlike these baselines, our novel memory disaggregated solution LegoBase significantly enables fast rebooting. For instance, with 50% remote memory, it introduces a up to 3.87× and 5.48× speedups of the recovery and warm-up time, respectively, compared to both monolithic and Infiniswap.

We identify the above inefficiency stemming from mismatches between the memory disaggregated architecture and the conventional monolithic software design, particularly, the traditional fault tolerance mechanism. Though memory is split across local and memory buffers in the context of memory disaggregation, the conventional designs such as Infiniswap still treat the two buffers as an entirety. Therefore, when a local compute node crashes, the whole memory space should be recovered by replaying the write-ahead logs. This would result in slow recovery considering the remote buffers are much larger than local ones and contain a large number of dirty pages. In addition, this precludes the opportunities to reuse the data already cached in the remote buffers when scaling up/down the compute node.

## 3 DISAGGREGATED DATABASE ARCHITECTURE

### 3.1 Design Rationale

Based on the aforementioned problems, in this paper, we advocate the co-design of database kernel and memory disaggregation to exploit the full potential of flexible memory and CPU resource allocation in the context of cloud-native database, and we are the first to materialize it in practice. We propose LegoBase, a novel cloud-native relational database architecture. It evolves the state-of-the-art designs such as Aurora, HyperScale and PolarDB, where the persistent storage is already decoupled from the rest of database logic, to further remove the binding between CPU and memory resources. LegoBase significantly differs from the existing general memory disaggregation proposals like LegoOS, Infiniswap and Leap by the following design considerations.

The first consideration is to close the performance gap between the monolithic setups and memory disaggregated ones. To do so, we push the memory management and remote memory access back to the database layer rather than the underlying Linux kernel. This provides us with three benefits for achieving better performance compared to the aforementioned database-oblivious solutions: 1) we make the remote memory page access completely bypass the time-consuming kernel data path and avoid the benefits of using RDMA to be offset by the kernel traversing time cost (usually 30 $\mu s$ [2]); (2) we can retain the sophisticated design of the LRU mechanism used in the conventional relational database such as MySQL, which we find much more effective than the similar counterparts used in the Linux kernel; and (3) we could explore database-specific optimizations such as clever metadata caching and dynamic page alignment w.r.t access patterns and data layout for reducing the number of remote communication steps.

Second, to mitigate the problems related to fault tolerance, we re-architecture the fault tolerance mechanism, which enables the local and remote node to handle faults independently as much as possible. In more detail, changes to both local and remote memory need to be protected via dirty page flushing and checkpointing. We treat the remote memory as another persistence layer and first flush dirty pages in the local buffer to remote memory and create checkpoints there. This can be done efficiently at high speed since flushing to remote memory via RDMA is two orders of magnitude faster

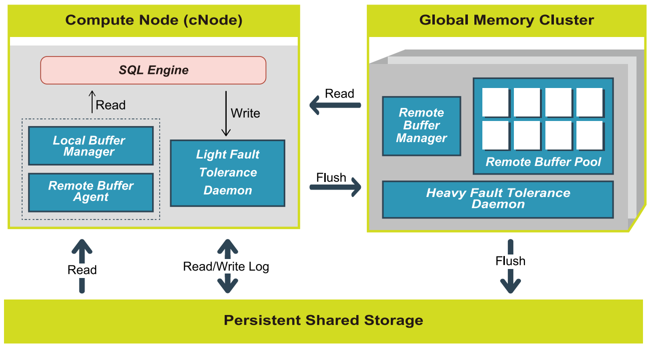

**Figure 5: LegoBase overview. Three components are connected by our highly-optimized RDMA library.**

than flushing to the persistent shared storage. When only the local buffer crashes, its recovery from the checkpoints stored on remote memory can be extremely fast since only quite a few pages that need to be restored between the last checkpoint and the crash point. However, we still need to handle remote memory crashes, which may lead to data loss. To do so, we introduce another level of fault tolerance mechanism to the remote memory side, which flushes dirty pages shipped from the local memory and the generated big checkpoints to persistent storage. When remote memory crashes, its recovery process resorts to the traditional database recovery.

### 3.2 Overall Architecture

Figure 5 gives a high-level overview of LegoBase. This new structure consists of three key components, namely, (1) *Persistent Shared Storage* (*pStorage*), which employs the replication protocols for storing write-ahead logs, checkpoints and database tables, (2) *Compute Node* (*cNode*), which performs SQL queries by consuming data from *pStorage*, and (3) *Global Memory Cluster* (*gmCluster*), which could allocate infinite remote memory to hold the bulk of *cNode*'s working set that cannot fit into its local memory. All these three components are connected by RDMA network for fast data access. Note that we rely on *pStorage* to offer data availability and database crash consistency in the presence of failures.

Within *cNode*, we retain the design of a traditional relational database, where a SQL engine executes SQL queries by interacting with a Local Buffer Manager, which manages a small amount of local memory buffer (e.g., 64MB-1GB) for hosting some hot memory pages. However, different from its original monolithic setup, the bulk of its memory pages are stored in the remote buffer pools allocated and managed by *gmCluster*. Thus, we then introduce a Remote Buffer Agent, which acts as the proxy of *gmCluster*, caches the necessary metadata (such as remote page addresses and identifiers) and performs fast reads and writes to the remote buffer pools managed by *gmCluster*. Additionally, to support fast recovery of *cNode*, here LegoBase augments *cNode* with a Light Fault Tolerance Daemon, which performs exactly the same logic with the following changes. It persists write-ahead logs (WAL) to *pStorage* as usual, but pushes checkpoints and dirty pages at higher frequency to *gmCluster* rather than *pStorage*.

*gmCluster* spans across multiple physical servers and harvests their spare memory to form a number of Remote Buffer Pools, each of which consists of a large number of memory pages and is

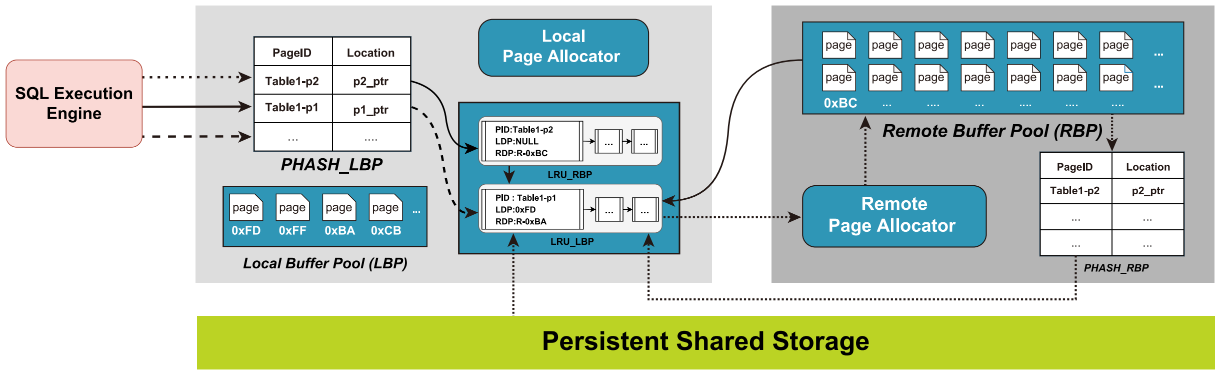

**Figure 6: Local and remote memory management in LegoBase. The dotted arrows represent page accesses from local buffer pool, while the solid arrows correspond to the remote page accesses via the locally cached remote address pointers. And the dashed arrows indicate page accesses from persistent shared storage.**

allocated for one *cNode* upon request. We employ a Remote Buffer Manager to take care of all these pages and serve the read and write requests from *cNode*. Finally, to retain the fault tolerance guarantees, we offload the original checkpointing and dirty page flushing logics from *cNode* to the Heavy Fault Tolerance Daemon residing on the memory node of *gmCluster*, which allows independent recovery of local and remote buffers and easy re-usage of remote buffers. Due to its light management cost, *gmCluster* does not require dedicated servers; instead, it can co-locate with other workloads.

Figure 6 also depicts the different types of LegoBase I/O traffics. When the SQL Engine needs to read a memory page, it first consults the Local Buffer Manager to locate the target page in local memory buffer. If missed, it continues to ask the Remote Buffer Agent to read the target page remotely. When the requested page is neither in local nor remote buffer, *cNode reads it from pStorage*. Upon the completion of a write request, *cNode* writes its WAL log entries to *pStorage* to prevent data loss in presence of failures. *cNode* treats *gmCluster* as another layer of persistence, and its Light Fault Tolerance Daemon periodically flushes cold dirty pages in its local buffer to its remote buffer pool on *gmCluster*, as well as the corresponding checkpoints. On the remote side, the Heavy Daemon also periodically collects dirty pages from the Remote Buffer Pool and flushes them to *pStorage* for generating checkpoints.

另一种是先gm，再到local。直接读固然少了延迟，可是这种缓存策略是否合适呢？

## 4 DATABASE-AWARE MEMORY DISAGGREGATION

### 4.1 Local and Remote Memory Management

In Figure 6, we introduce two layered buffer pools to the compute node for hosting hot data pages in both local and remote memory spaces. When booting a compute node, we first allocate its local buffer pool as a contiguous memory space with a pre-defined size. In addition, *gmCluster* aggregates memory space from multiple physical servers together to form a distributed shared memory. LegoBase's remote shared memory space consists of many *remote buffer pools* that are the unit of address mapping, the unit of recovery, and the unit of registration for RDMA-enabled NICs. When a *cNode* is launching, we also allocate a *remote buffer pool* consisting of the requested number of memory pages for it. It is worth noting that the local buffer pool is small and serves as a cache of the most recently used pages for the remote large buffer pool.

Each page has a metadata frame, which includes a page identifier (e.g., table id in the database namespace plus the offset of the target page in that table) inherited from the upper SQL execution engine, and two address pointers pointing to its address in either local and remote buffer. Free pages' page identifiers are `NULL`, while cached pages' metadata attributes are properly set. We appoint a `Local Page Allocator` for *cNode* and a `Remote Page Allocator` for *gmCluster* with similar logics to manage metadata frames of and allocate free pages when accessing pages that are either not locally cached or only on *pStorage*.

这段不是特别理解，only on pStorage 那不就not locally cached了吗

We employ two LRU lists in *cNode* to organize metadata frames of most recently used memory pages for local and remote buffer pools, respectively. `LRU_LBP` links metadata frames of pages that are cached in both local and remote buffer pools, while `LRU_RBP` manages metadata of pages that have already been evicted from local to remote. This design allows us not only to enjoy the existing database-optimized LRU implementation for managing the local memory buffer, but also to cache necessary metadata of remote memory pages (e.g., their remote memory addresses) for enabling fast RDMA one-sided operations. We make the Remote Buffer Agent manage `LRU_RBP` in *cNode*.

Finally, to quickly locate a page requested by the upper SQL execution engine, *cNode* contains a hash-table `PHASH_LBP`, which maps a page identifier to the metadata frame of the relevant page in two LRU lists. Non-existence results from the hash table lookups indicate the requested pages are not cached and should be loaded from *pStorage*. There is a similar hash table `PHASH_RBP` in *gmCluster*, which is used to accelerate the remote page address lookups when *cNode* reboots and its `LRU_RBP` is not populated yet.

*cNode* includes four RDMA-based operations to communicate with *gmCluster* to co-manage remote memory pages as follows:

- ***Register***: When *cNode* loads a non-cached page from *pStorage*, it must register that page in both remote and local buffer. To do so, *cNode* sends the page id to *gmCluster*'s `Remote Page Allocator`, which then allocates a free remote page and updates `PHASH_RBP`. Next, *gmCluster* returns the page's remote address directly to *cNode* by a one-sided RDMA Write operation. Finally, `Local`

Page Allocator finds a free local page and places its metadata frame with both local and remote addresses set onto LRU_LBP.

- ***DeRegister***: When the remote buffer pool is full, we need to recycle remote memory space. To achieve this, *cNode* collects a few candidate pages from LRU_RBP and sends their page ids to *gmCluster*. Upon arrival, Remote Page Allocator initiates the space recycling task, which will be jointly executed by our page eviction (Section 4.3) and flushing mechanisms (Section 5.1).
- ***Read***: Since the pages' remote addresses are cached in LRU_RBP most of the time, when *cNode* wants to read a remotely cached page, it goes through the fast path, and just needs to use a RDMA Read to directly access *gmCluster*'s memory with no *gmCluster* CPU involvement. However, when the remote address is not cached, remote reads will execute the normal path, where *gmCluster* should be involved for performing the PHASH_RBP lookups.
- ***Flush***: We allow *cNode* to flush dirty pages to its remote buffer pool for either space recycling or fault tolerance (Section 5).

## 4.2 Memory Page Access

**Read paths.** When reading a memory page, the upper layer SQL engine first queries PHASH_LBP to obtain the target page's metadata organized by the two LRU lists. In total, there are three major read paths in LegoBase (Figure 6). First, if the page metadata tells this page is locally cached, then LegoBase reads that page directly from the local buffer pool using its local address pointer. Upon loading a remotely cached page, Local Page Allocator on *cNode* first tries to get a local free page which will be filled with remote content via an RDMA read operation. Next, it needs to finalize this remote loading by removing its metadata frame from LRU_RBP, which is then injected to LRU_LBP with the local address pointers set.

Finally, where the requested page is not found in both buffer pools, we need to read it from *pStorage*. Similar to the above remote page reading, we first obtain a free local page $p_{local}$ from Local Page Allocator and create a connection between the requested page id and $p_{local}$'s metadata frame in PHASH_LBP for future visits. Then, we overwrite this free page by the content loaded from *pStorage*. Next, *cNode* registers this page to the remote buffer pool by making Remote Page Allocator to allocate a free remote page for accommodating the newly read data with PHASH_RBP accordingly updated. In the end, *cNode* properly sets the address pointers of $p_{local}$'s metadata frame, which is then linked into LRU_LBP.

It is worth mentioning that we assume that there are always free pages left for allocation requests along the above paths. With regard to cases where no free pages are available, we have to trigger the page eviction process either locally or remotely (Section 4.3).

read old first, then update。是否可以考虑引入LSM-TREE对B-TREE那样的优化

**Write paths.** The way we handle write requests looks quite similar to reads, since writing a page needs to pull the old content to the local buffer pool via one of the three read paths presented above. Once the requested page is prepared, we overwrite its content and flag it as "dirty". The background flush threads are responsible for writing dirty pages to *gmCluster* for fault tolerance, and we will expand the discussion on dirty page flushing in Section 5.

## 4.3 LRU and Page Eviction

We manage the two LRU lists differently. First, we apply a similar least recently used (LRU) algorithm used in MySQL [34], highly optimized for relational database workloads, to manage LRU_LBP. When the room is needed to add a new page to the local buffer pool, the least recently used page is evicted and a new page is added to the middle of that list. This midpoint insertion strategy partitions LRU_LBP into two parts, namely, the young sub-list (at head) containing most recently accessed pages, while the old sub-list (at tail) including aged pages. The position of the middle point is configurable to adapt to different temporal and spatial localities. Accessing a cached page makes it become "young", moving it to the head of the young sub-list. For local memory reclamation, we just need to evict pages from the tail of the old sub-list.

Unlike LRU_LBP, we use a simpler LRU algorithm to manage LRU_RBP with no midpoint insertion strategy. This is because all pages in that list are retiring from LRU_LBP and we do not insert new pages to it. When no free remote pages are left, for making room for new pages in the remote buffer, *cNode* needs to recycle pages from the tail of LRU_RBP. For dirty pages, we have to flush them back to *pStorage* for data persistence.

There exist two types of LRU eviction, namely, LRU_LBP eviction and LRU_RBP eviction. Concerning the LRU_LBP eviction, when no free pages are available in the local buffer, *cNode* will scan the tail of the old sub-list of LRU_LBP to find the coldest page as an eviction candidate. There are two possible cases. First, if the selected eviction page is unmodified, we will simply move its metadata frame from LRU_LBP to LRU_RBP. Second, if that page is dirty, then we have to flush its content to the remote buffer, followed by returning it to Local Page Allocator for future allocation and moving its metadata frame from LRU_LBP to LRU_RBP.

The LRU_RBP eviction looks slightly different from the LRU_LBP counterpart. When all free pages of the remote buffer are used up, *cNode* will find a remote page eviction candidate $p_{remote}$ from the tail of LRU_RBP and execute a DeRegister operation against *gmCluster* with $p_{remote}$'s page id as the parameter. If this page is unmodified, we simply return it back to Remote Page Allocator and delete its relevant entries from both PHASH_RBP and PHASH_LBP. However, when $p_{remote}$ is dirty, we have to additionally make *gmCluster* flush its content to *pStorage* prior to executing the similar steps as the unmodified pages.

Finally, page eviction can take longer when it involves dirty page flushing. To remove its negative performance impact on the SQL query processing critical path, we appoint a background thread to periodically release cold pages whenever needed. To trigger this procedure, we reserve a fixed number of free pages for cheap allocation. In our practice, we find setting it to 100 is sufficient. When the number of free pages drops below that threshold, the background thread will be woken up to scan both LRU lists and recycle least used pages, possibly flushing dirty pages from local to remote or remote to *pStorage*.

## 4.4 Efficient RDMA-based Communication

We rely on fast RDMA network to close the performance gap between local and remote page accesses. However, unlike the traditional usage of RDMA, we adapt it to take into account the unique I/O characteristics of databases as follows.

**One-sided RDMA operations.** We intensively use one-sided RDMA operations issued by *cNode* to *gmCluster* to achieve high performance with low CPU overhead. However, the difficulty we face here is to notify *gmCluster* when the corresponding operations complete. There are two possible notification techniques such as *memory address polling* and *event driven*, which behave differently and have various system implications. First, the polling method incurs negligible overhead at a high cost of *gmCluster*'s CPU utilization. Therefore, we assign it to handle latency-sensitive LegoBase operations such as *register*, *read* and *write*, which will affect user experience. In contrast, second, the event driven method consumes little CPU resource but incurs high latency. Thus, we use it for the background *evict* operations.

**Variable-sized database pages.** The page sizes in modern database systems range from 2 to 16 KB. However, it is difficult to enable dynamic variable-sized page allocation, since using RDMA requires us to register the remote memory space when bootstrapping *cNode*. Therefore, LegoBase uses fixed-size pages (typically 16KB) as the unit of memory management. This design avoids splitting one database page remote access into a few RDMA operations, which is instead performed by OS-level disaggregated solutions. However, this also introduces two challenges to handle pages smaller than 16KB. First, when accessing a 2KB page, one would read the entire 16KB remote page, resulting in poor network performance and unnecessary bandwidth usage. To address this, we allow each RDMA operation to take a length parameter to indicate its actual data size and ignore the padding data when transferring data from *gmCluster* to *cNode*. Second, using fixed-sized paging for variable-sized pages would lead to memory fragmentation. Here, we adopt the buddy algorithm [28] to reassemble the unused space of allocated pages into 16KB free pages again in the background.

## 5 FAULT TOLERANCE AND STATE RECOVERY

Memory disaggregation offers a great opportunity to consider the local and remote state separately to accelerate the recovery stage. However, the current fault tolerance mechanism such as ARIES[33] still treats the two separated memory spaces as an entirety, and thus fails to exploit such opportunity. To address this fundamental inefficiency, we extend the success of hardware disaggregation to the software context to recover local and remote states independently. In particular, we would re-use the large alive remote buffer when only *cNode* crashes. The challenge here is to help *cNode* obtain a consistent view of states residing in the remote buffer. To do so, we introduce a novel *Two-tier ARIES Fault Tolerance* protocol, where each tier summarizes and protects either local or remote memory space. The first tier involves the interactions between *cNode* and *gmCluster* for fast recovery, while the second tier is between *gmCluster* and *pStorage* for normal recovery when the whole application including local and remote space crashes.

### 5.1 Two-tier ARIES Protocol

Figure 7 illustrates the workflow of our two-tier ARIES mechanism. When receiving modification requests (e.g., *insert*, *update* and *delete*) from a transaction executed by the upper transaction manager, to prevent data loss, we follow the original design to make *cNode* first update the relevant pages in the local buffer and write the corresponding log entries to `Log Buffer` (①-②). Then, *cNode* places metadata frames of the relevant modified pages onto the flush list `FLUSH_LBP` for being flushed in background (③). `FLUSH_LBP` is sorted by *Log Sequence Number* (*LSN*), which is assigned uniquely to different transactions and represents the address of the log entry in `WAL` for the corresponding transaction. Next, *cNode* notifies the upper layer of the completion of the corresponding requests (④). Finally, when transaction commit arrives, the relevant entries in `Log Buffer` will be written to the `WAL` persisted on *pStorage*.

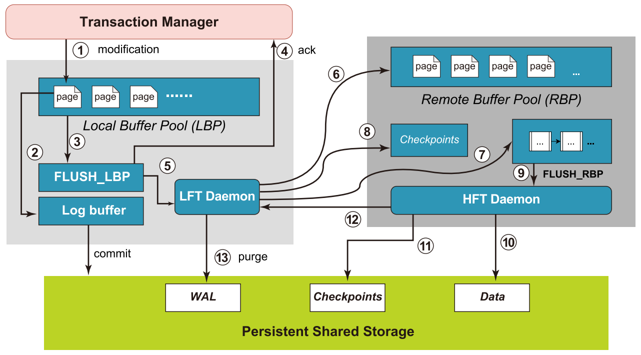

**Figure 7: The two-tier ARIES protocol workflow. ①-④ correspond to the execution of updating queries, while the rest are background flushing, checkpointing and log purging among *cNode*, *gmCluster* and *pStorage*. `LFT` and `HFT` stands for Light and Heavy Fault Tolerance, respectively.**

In background, our `LFT Daemon` collects a batch of dirty pages associated with oldest *LSN* numbers from `FLUSH_LBP` every *T* milliseconds (⑤), and writes their contents to the remote buffer (⑥). Once dirty pages are written to *gmCluster* by *cNode*, we need to create a tier-1 *checkpoint* to summarize all these successful changes and associate it with the largest *LSN* among those pages (⑧).

For dirty pages shipped from *cNode*, *gmCluster* continues to place their metadata entries onto the remote buffer flush list `FLUSH_RBP` for final persistence (⑦). Again, similar to *cNode*, we also employ a background daemon thread to perform bulk flushing. However, unlike the tier-1 flushing to memory, `HFT Dameon` writes a batch of modified pages gathered from `FLUSH_RBP` from the remote buffer pool to the data files on *pStorage* at once (⑨-⑩). Upon completion, `HFT Daemon` creates a tier-2 *checkpoint* on persistent storage.

It is worth mentioning that when a tier-1 checkpoint $chpt_{t1}$ is created, we cannot delete log entries whose LSN numbers are older than $chpt_{t1}$'s *LSN* in `WAL`, since the corresponding pages are just in remote buffer and not yet persisted to underlying *pStorage*. We delay this until a tier-2 checkpoint is built. At that moment, we report its *LSN* back to *cNode* (⑫). In the end, `LFT Daemon` can safely purge log entries prior to that *LSN* on *pStorage* (⑬).

直接说单调性更明了，不得不说这篇paper语言叙述方面有很大改进空间，和ipads逻辑的清晰程度有层次的差距

### 5.2 Consistent and Fast Flushes

There are two challenges LegoBase faces when considering to enable fast flushes initiated by *cNode* and *gmCluster*, respectively. First, potential conflicts exist when *cNode* and *gmCluster* flush the same dirty page simultaneously. A naive solution would use distributed locks to coordinate the two types of activities. However, it would introduce performance loss and complexities to handle lock coherence. Second, we face a partial remote memory write problem when *cNode* crashes in the middle of flushing. We address the two problems by not directly applying changes to pages in the remote buffer, instead, accommodating writes from *cNode* into a separated memory buffer on *gmCluster*. Here, we use the RDMA connection buffer. Once the full contents of pages are received, we notify *gmCluster* to copy to their memory locations. This design reduces the distributed locks into local locks at the remote memory side to coordinate local reads and writes by *gmCluster*. Meanwhile, it precludes partial page writes since the connection buffer will be discarded when *cNode* crashes during flushing.

We further optimize the above design by making *gmCluster* actively pull dirty pages from *cNode* via RDMA read operations. Compared to the naive design, where *cNode* performs RDMA writes to and explicitly notifies *gmCluster*, the optimized design is more efficient. Furthermore, the limited and precious RDMA connection resources may be wasted when the corresponding dirty pages are copied from *cNode* but the write locks on remote pages cannot be obtained due to background flushing. In such case, we will introduce an extra memory copy, which loads content in the RDMA connection buffer to another memory buffer, and release RDMA connections immediately. Finally, once the dirty pages from the additional memory space are written back to the remote buffer pool, we can clear them up at once for accommodating subsequent flush activities from *cNode* to *gmCluster*.

## 5.3 Failure Handling and State Recovery

After disaggregation, *cNode*, memory nodes of *gmCluster* and *pStorage* can fail independently. As the underlying persistent storage often employs fault tolerance protocols such as 3-way replication [44] or erasure coding [23, 37, 39], we omit the failure handling for data on *pStorage*. Therefore, here, we focus on recovering memory state from the following two crash scenarios.

First, we consider the case where *cNode* crashes but its remote memory buffer is still available. The recovery is rather straightforward. When rebooting from such a crash, *cNode* first attaches its remote buffer via initiating an RDMA connection to *gmCluster* with IP address presented in its configuration file. Next, `LFT Daemon` reads the latest *tier-1 checkpoint* created before crash in its remote buffer. Then, it continues to traverse the `WAL` on *pStorage* and apply changes with higher *LSN* than the latest *tier-1 checkpoint* to its local memory. This will re-construct `LRU_LBP`. However, this recovery can be done quickly, since we allow a high flushing rate for *cNode* and the changes contained in *checkpoint* are few.

Second, we consider a more complex case where *cNode* and the memory node hosting the remote buffer of *cNode* both crash. When this happens, we reboot *cNode* and allocate a new remote buffer for it. Next, we resort to the traditional recovery process identical to the one used for recovering applications in a monolithic server. In short, *cNode* reads *tier-2 checkpoint* from *pStorage* and use its *LSN* to determine relevant changes, which will be then applied. This recovery would result in many misses in the remote buffer. Considering the remote buffer is much larger than the local one, a long cold start would be expected. To mitigate this problem, we could leverage memory replication when monetary budget permits, i.e., allocating a backup remote buffer for *cNode* on another physical memory node in *gmCluster*, to mask the remote memory failure, as LegoOS did [40].

**Optimization for elastic deployment.** Datacenter applications expect cloud-native databases to offer the elasticity feature, which provisions and de-provisions resources such as CPU and memory, adapting workload changes. In the context of memory disaggregation, with all the above mechanisms, we can quickly migrate a hardware setup to another for meeting the elastic goal. Scaling up and down the remote memory allocation by *gmCluster* is rather straightforward. However, changing the CPU resource allocation on *cNode* would result in a planned shutdown and reboot. To optimize this, before stopping it, we make *cNode* flush all its dirty pages to *gmCluster* and create a tier-1 checkpoint. When rebooting from another physical server with new hardware configurations, *cNode* easily attaches its remote buffer and bypasses the crash recovery process. This can be done very efficiently, and performance evaluation numbers are presented in Section 7.6.

## 6 IMPLEMENTATION DETAILS

We implemented LegoBase on top of MySQL, with around 10000 lines of C++ code for its core functionality. The underlying storage layer is PolarFS [10], a distributed file system at Alibaba.

## 6.1 RDMA Library

We build our RDMA library on IB Verbs and RDMA CM API. We first use RDMA CM API to initialize the RDMA device context. Then we use the Verbs API to perform the memory buffer registration between two connected parties (i.e., *cNode* and memory node in *gmCluster*), and to post one-sided RDMA requests to target nodes via a RDMA connection. However, the establishment of RDMA connections between *gmCluster* and *cNode* can be expensive when facing a large number of concurrent requests due to the multiple round-trips required by the protocol and the memory buffer allocation and registration for performing remote writes. To eliminate this potential bottleneck, we maintain a RDMA connection pool shared between *gmCluster* and *cNode* for connection re-usage. For each remote memory access, *cNode* will grab an available connection and specify the remote memory address and the unique identifier of the target pages. For the sake of performance, we further align the registered memory address and let dedicated CPU cores poll the specific tag to inspect the status of ongoing remote requests.

## 6.2 Modifications to MySQL

*cNode* runs a modified MySQL database with the following changes.
**Memory management module.** We introduce a two-tier LRU implementation, where the first tier is the ordinary LRU list of MySQL, while the second tier is used to manage the remote pages' metadata. In addition, we extend the page metadata structure to include two fields, namely, remote memory address and LRU position. The LRU position tells if a page is cached locally. These two fields are used to implement different data access paths presented in Section 4.2. For a local hit, the target page can be directly served. However, when a page is stored remotely, a RDMA request will be issued to read it from *gmCluster*, taking the remote memory address field as input.
**ARIES protocol.** We remain the WAL writing path unchanged. Unlike this, for the dirty page and checkpoint flushing requests, we

intercept the original calls to file system, and redirect the requests to the remote buffer via RDMA operations. For state recovery, we add an additional fast recovery path to read checkpoints from *gmCluster* and replay WAL entries between them and the crashed point.

## 6.3 Limitations and Discussions

LegoBase organizes the memory space in a hierarchy rather than a flat structure, where LBP is the cache of RBP. This design choice stems from the need for fast scaling up/down the CPU resources. With this setup, LegoBase will bring significant performance gains for workloads with large working sets, whose sizes are far beyond the memory capacity of the physical server that hosts the corresponding *cNode*. In the future, we will explore the chances to apply optimizations such as page prefetching [2] to balance the trade-off between local buffer sizes and overall performance.

Though the current LegoBase implementation is bound to MySQL, we believe it is not difficult to make LegoBase support other ARIES databases such as PostgreSQL [21], PolarDB [9, 10], etc. The integration requires modifying the codebase of these databases to add an additional data path to fetch remote pages and to adapt their ARIES-based fault tolerance protocols to be aware of the checkpoints stored in *gmCluster* (Section 6.2).

To prototype LegoBase, we only support the single-instance deployment of MySQL. This says that LegoBase offers data availability and crash consistency in the presence of failures via the joint work of the two-tier ARIES protocol and the replication carried out by the underlying storage (i.e., PolarFS). To enable the database service failover, we need to employ database redundancy, e.g., deploying a stand-by MySQL instance in the background to catch up with the in-memory state mutations of the foreground active instance. We leave this exploration as future work.

## 7 EVALUATION

## 7.1 Experimental Setup

**Platform.** We run all experiments on three physical machines, each with two Intel Xeon CPU E5-2682 v4 processors, 512GB DDR4 DRAM, and one 25Gbps Mellanox ConnectX-4 network adapter; The Linux version we use is AliOS7.

**Baselines.** We compare the performance of LegoBase to the best performing monolithic setup, in which a MySQL instance running in a Docker container does not have remote memory allocation, denoted by "MySQL". We also choose MySQL atop Inifiniswap as the natural memory disaggregation baseline, which leverages a remote paging system via Linux kernel, denoted by "Infiniswap". And PolarFS is used as underlying storage for "MySQL" and "Infiniswap". We use two machines to host LegoBase's *gmCluster* and Infiniswap's remote paging sub-system with a total of 200GB memory space, respectively. Additionally, we use another machine to host the MySQL data compute instances (*cNode*). Note that all machines are used in a shared mode with other workloads.

We focus on the single-instance performance, and use Docker[25] to control the resource allocation of *cNode*. We allocate remote memory using the APIs offered by Infiniswap's paging system and LegoBase. Unless stated otherwise, we consider three memory configurations, namely 100%, 75%, and 50%. The 100% configuration corresponds to the monolithic MySQL setup, where we create a

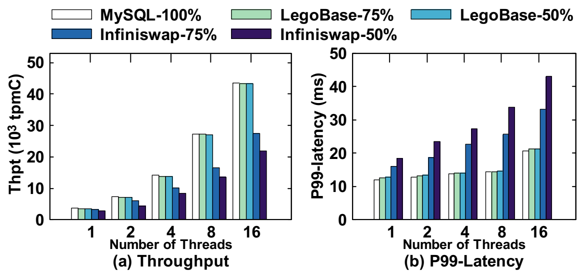

**Figure 8: Performance comparison of the TPC-C benchmark between LegoBase and Infiniswap with varied memory allocation configurations**

Docker with enough local memory to fit the entire working set in memory for a target workload. We measure the peak memory usage of the monolithic MySQL configuration, and run 75% and 50% configurations of LegoBase and Infiniswap by creating Dockers with enough local memory to fit the corresponding fractions of their peak usage and allocating remote memory for the remaining working sets. We do not further decrease the local memory ratio for Infiniswap since it fails to run when this ratio drops below 50%. However, LegoBase does not have such limitation and we even test it with a considerably low local memory ratio, e.g., 10%. By default, we allocate 8 cores for each containerized instance.

**Workloads and datasets.** We use two widely used benchmarks TPC-H [6] and TPC-C [5], which represent an OLAP and OLTP workload, respectively. We populate the database with 20GB records for TPC-C, and set the *warehouse* parameter to 200. Regarding TPC-H, we set its *scale factor* to 30, and use a 40GB database. Additionally, we use an Alibaba production workload, with a profile of 3:2:5 insert:update:select ratio. We also use Sysbench [29], a popular benchmark stressing DBMS systems, as a workload to evaluate the fast recovery and elasticity feature of LegoBase.

## 7.2 Overall Performance

**TPC-C results.** Figure 8 presents the throughput (measured as tpmC, the transactions per minute) and P99 latency comparison among three systems for the TPC-C. Here, 20GB memory space is sufficient to fit the entire working set. We also vary the number of concurrent workload threads to increase the workload density level. Clearly, across all test cases, the throughput and P99 latency of Infiniswap worsen by up to 2.01 × and 2.35×, respectively, compared to the best-performed monolithic MySQL setup. Furthermore, decreasing the local memory size plays a strongly negative impact on the overall performance of Infiniswap. For example, with 16 threads, Infiniswap-50% introduces a 20.52% drop in throughput and a 29.93% increase in P99 latency, compared to Infiniswap-75%. The reasons for this performance loss can be found in Section 2.

Unlike Infiniswap, LegoBase delivers comparable performance as MySQL across all cases, even with 50% memory placed remotely. For instance, LegoBase-75% lowers (increases) MySQL-100%'s throughput (P99 latency) by only up to 3.82% (4.42%). Surprisingly, LegoBase with smaller local buffer size has moderate performance loss, e.g., we observe a 0.74% throughput drop and a 1.51% P99 latency increase with 8 threads, when switching from LegoBase-75% to LegoBase-50%. In the end, LegoBase significantly outperforms the

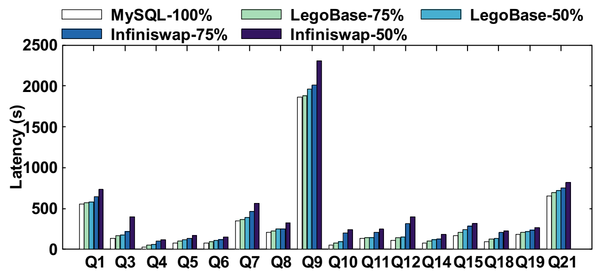

**Figure 9: Query latency comparison of the TPC-H benchmark between LegoBase and Infiniswap with varied memory allocation configurations**

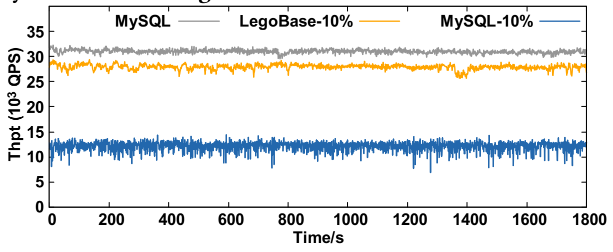

**Figure 10: The system throughput comparison of an Alibaba production workload as time goes.**

general database-oblivious Infiniswap by up to 1.99× and 2.33× in terms of throughput and P99 latency, respectively.

**TPC-H results.** Figure 9 shows the latency comparison of selected TPC-H queries among the three systems. The peak memory usage of TPC-H queries is 40GB. We exclude Q2, Q13, Q16, Q17, Q20 and Q22 queries from this figure for clarity since they behave similarly but run incredibly shorter or longer than others. We can draw similar performance conclusions for TPC-H as TPC-C. Overall, MySQL-100% delivers the lowest query latency, while Infiniswap achieves the highest ones. Across all queries, LegoBase-75%'s latencies are closer to the ones of MySQL-100%, while LegoBase-50% introduces a slight increase in query latency. However, regardless of the local buffer size, LegoBase significantly performs better than the two variants of Infiniswap. Take the 11th query as an example, LegoBase-50% runs only 4.54% slower than MySQL-100%, but 45.4% and 75.7% faster than Infiniswap-75% and Infiniswap-50%, respectively.

**Alibaba production workload result.** In addition to synthetic workloads, we evaluate LegoBase with a trading service workload at Alibaba, which is very memory-intensive and contains a significant number of update transactions. The peak memory usage of this workload is 256GB. We further stress LegoBase with a much smaller local buffer size. Here, we set it to 10%. Figure 10 shows the performance comparison between MySQL and LegoBase. At a glance, LegoBase achieves a sustained high throughput (measured as queries per second) and its average throughput is 27904.8, which is just 9.96% lower than MySQL. We also deployed a monolithic MySQL variant with only 10% local memory space, denoted by "MySQL-10%". Surprisingly, its performance is only around 39% of that of the best performing MySQL. The difference between MySQL-10% and LegoBase-10% implies that the amount of most used pages already exceeds the local memory space, and the remote memory in LegoBase is heavily used and important for performance enhancement, while MySQL-10% experiences IO inefficiencies when reading

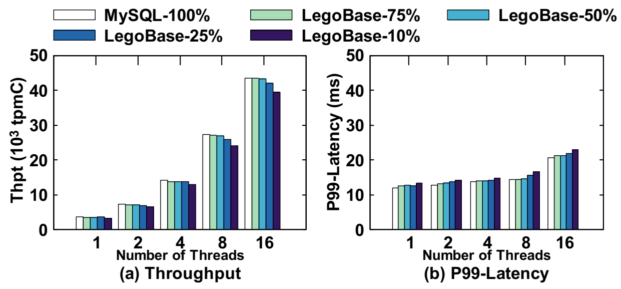

**Figure 11: Performance of TPC-C running over MySQL and LegoBase with different local memory buffer sizes.**

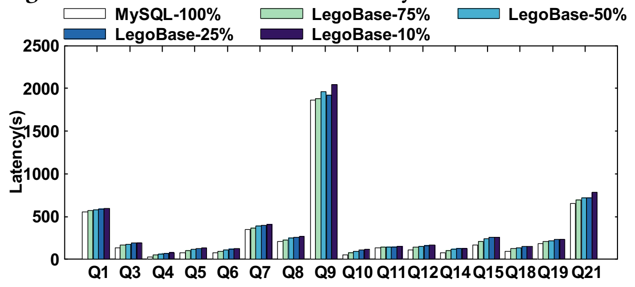

**Figure 12: Performance of TPC-H running over MySQL and LegoBase with different local memory buffer sizes.**

data from the persistent storage. Thus, LegoBase is well-suited the cloud-native environments and can offer comparable performance even with large memory space placed remotely.

## 7.3 Impact of Local Memory Buffer Size

Next, we focus on the influence of local memory buffer sizes on the performance of LegoBase. Here, we continue to use TPC-C and TPC-H as benchmarks. The experiment setups are the same as the above ones. We change the local memory buffer sizes from 10% to 100% of the peak memory usage.

Figure 11 summarizes the system throughput and P99 latency numbers performance of TPC-C with LegoBase w.r.t different local memory buffer sizes, and we compare these numbers against the best performing MySQL. We do not compare with Infiniswap since it does not accept less than 50% local memory ratios. With 1 to 2 threads, LegoBase achieves similar performance as MySQL, across all memory configurations. This is because that the system is lightly loaded and almost the entire working set can fit into the local buffer. However, with the increasing number of threads, smaller local buffer sizes have visible but moderate performance loss. For instance, even only having 10% of data available in local memory, LegoBase still achieves up to 91% of MySQL's throughput and its P99 latency by up to 10%, under 16 threads. The performance loss is a direct consequence of the number of unavoidable remote memory accesses to pages that are not cached in the local buffer.

Figure 12 shows the measured latency of TPC-H queries with varied local buffer ratios ranging from 10% to 100%. For all selected queries, decreasing the local buffer ratio would result in different degrees of performance degradation. For most queries except Q4 and Q10, LegoBase increases their latency numbers by 7.61%-62.7% across all memory configurations. However, we observe up to 2× worse performance achieved by LegoBase for Q4 and Q10, compared to MySQL. This is because the two queries are

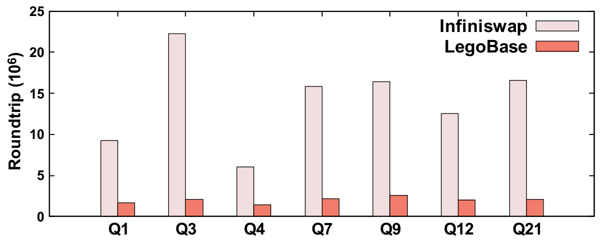

**Figure 13: The number of remote accesses of TPC-H Queries**

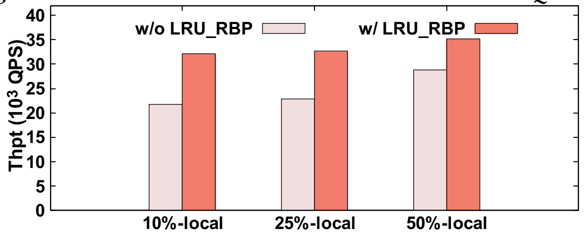

**Figure 14: Throughput comparison with `LRU_RBP` switched off/on, using LegoBase with Sysbench oltp_read_write.**

more memory-intensive than the rest with less temporal locality. Thus, we observe that a large number of remote memory accesses were introduced when the local buffer size is limited for the two queries. This set of results points out there exists a trade-off among query patterns, performance, and local buffer sizes. We leave this exploration as future work. Finally, in spite of the performance gap between LegoBase and the best performing MySQL, we observe that LegoBase significantly outperforms the worst-performing MySQL with a limited local buffer, whose size is a fraction of its peak memory consumption. For instance, LegoBase-10% introduces a 1.48× and 1.41× latency speedup than MySQL-10% for Q4 and Q10, respectively. This is because loading data from a remote buffer is still much faster than from the underlying storage.

## 7.4 Breakdown Analysis

**Reduction in communication steps.** Even with RDMA, a remote page access is still much longer than a local one. Therefore, the primary optimization enabled by our solution is to reduce the number of communication steps. To understand this, we plot the total number of round-trips of executing a set of selected TPC-H queries with Infiniswap and LegoBase in Figure 13. Here, the system configurations are looking identical to Figure 9. Across all queries, Infiniswap introduces significantly more network activities than LegoBase. For example, the number of round-trips of Infiniswap is 5.5×, 10.8×, and 4.11× of LegoBase's, for Q1,in Q3 and Q4, respectively. The superior performance of LegoBase is because of its co-design of memory disaggregation and database engine.

**Impact of `LRU_RBP`.** We use `LRU_RBP` as a remote metadata cache to reduce the number of remote address lookups. To understand this, we evaluate the impact of this optimization w.r.t different memory configurations. Here, we switch on and off the `LRU_RBP` and consider the following memory configurations, namely, 10%, 25% to 50%. We use the sysbench oltp_read_write workload with uniform distribution. Figure 14 reports the throughput number comparison. As expected, switching off `LRU_RBP` would result in significant performance loss. For example, the optimized LegoBase outperforms the counterpart without `LRU_RBP` by 47%, 43% and 21.6%, when the local memory buffer ratio is 10%, 25% and 50% of the peak memory usage, respectively. This is because with `LRU_RBP`, we can directly obtain the locally missed pages' remote memory address without network operations. Furthermore, smaller memory buffer sizes observe better performance improvement, since they result in more cache misses than other memory configurations.

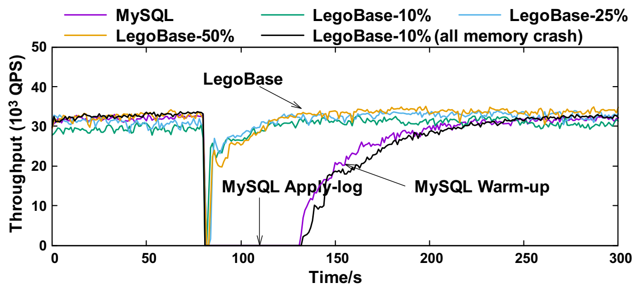

**Figure 15: The comparison of the time to recover from crash failures and warm up the rebooted compute instances.**

## 7.5 Fast State Recovery

To investigate LegoBase's recovery performance, we run the sysbench oltp_read_write workload with both MySQL and LegoBase. For LegoBase, we vary its memory configurations from 10% to 50%. We inject a crash failure to bring the database compute instance down at 80 second, and observe the recovery process. We first measure the recovery time, which corresponds to the time spent between rebooting a new instance after a crash and bringing its states to be consistent again. We then measure the warm-up time, which corresponds to the time from being able to serve requests to the point the peak throughput reaches.

In Figure 15, MySQL takes 50s to recover crash states and is able to serve requests again at 130 second. It also takes another 70s to populate all its memory space to reach its 90% of highest performance again. In contrast to MySQL, the downtime of LegoBase is very short. For instance, it takes only 2s to recover states for LegoBase-10% and LegoBase-25%, and 3s for LegoBase-50%. This is because the first-tier ARIES protocol enables a quick recovery of the compute node by reading checkpoints from *gmCluster*. Note that we only require a limited amount of computational resources in *gmCluster* since the recovery process still runs on *cNode*, while *gmCluster* just needs to buffer dirty pages and checkpoints, and flush them to storage.Furthermore, LegoBase takes up to 16s to achieve 90% of its peak throughput across all three memory configurations, and its warm-up time is up to 4.38× shorter than that of MySQL. This is because the vast majority of memory pages cached in the large remote buffer do not need to be recovered and loaded, instead, they are re-used by LegoBase's compute node.

Additionally, we inject another failure to bring a remote node in *gmCluster* down at the 80s, and measure the state recovery time of the victim *cNode*. As illustrated by the LegoBase-10%(all memory crash) curve, its recovery behavior looks similar to that of MySQL, and experienced 52-second downtime. This is because when the remote buffer crashes, we allocate a new remote buffer for *cNode*, and then resort to the original MySQL recovery procedure to populate both local and remote buffer again by loading the newest persisted checkpoint and replaying WAL records from the storage layer.

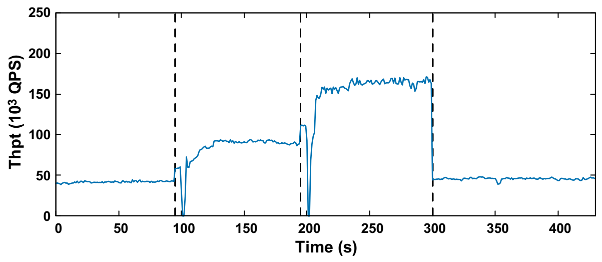

**Figure 16: Performance of LegoBase when adapting CPU allocations to three workload changes (vertical dashed lines)**

这里感觉才是这个系统设计的最亮点之处，但是为什么放到了后面

## 7.6 Elasticity and Cost-Effectiveness

Finally, to evaluate LegoBase's quick elastic deployment feature, we again use a multi-phase sysbench oltp_read_write to stimulate time-varying user behaviors in the production environment. From 0-95s, we use 6 workload generation threads, double the thread number between 96-195s, and use 24 threads in the 196-300s time interval. Finally, we reduce the number of threads from 24 to 6 in the final time interval (300-430s). When a sudden increase in traffic is detected, LegoBase will accordingly increase the CPU resource provision by migrating the target application to another container with more CPU cores. Figure 16 portraits the request throughput.

During phase 1, LegoBase achieves a stable throughput, which is about 43000 queries per second. At the 100 second, we observe that LegoBase's throughput drops to zero but the downtime is just 2 seconds. This connects to the container migration that responses to the first workload change. Because LegoBase flushes all dirty pages to the remote buffer before migration, the state recovery is very fast. During phase 2, the new LegoBase obtains 16 CPU cores and thus brings its peak throughput to 93188.84. Again, at the 200 second, to react to the second workload change, migration takes place again and introduces only a 2-second downtime. During phase 3, due to plenty of CPU cores, the LegoBase's performance reaches 167953.26 and is nearly 4.0× of the one achieved in phase 1. The phase 4 starts at the 300 second, where we shut down most of sysbench threads and only use 6 threads to generate workload. When detecting this change, we do not immediately de-provision LegoBase's CPU cores. Instead, we introduce a graceful time period of 30 seconds. This says that at the 330 second, we start to remove 2 CPU cores from the running container (rather than migrating to a new one) every two seconds until only 8 cores left. We observe that LegoBase's throughput drops to the level of the phase 1 right after the workload change, and then remains unchanged.

**Cost-effectiveness.** In the end, we try to calculate the cost saving with the elastic deployment. With regard to experiments in Figure 16, thank to the support of LegoBase's elasticity feature, we use 7800 (core × seconds) in total. However, using the traditional resource provision method to avoid server overloaded, one has to allocate 32 CPU cores all the time during four phases, which results in a total number of 13760 (core × seconds). Therefore, LegoBase reduces the monetary cost by 44% for the end-users.

## 8 RELATED WORK

**General Resource Disaggregation.** The recent proposals falling into the resource disaggregation category include LegoOS[40], dRedBox[27], Firebox[4], HP "The Machine"[13, 18], IBM system[12], Facebook Disaggregated Rack[17], Intel Rack Scale Architecture[26], etc. Among these proposals, LegoOS is most relevant to LegoBase, but they significantly differ in remote memory management. LegoBase manages remote memory mostly by the database kernel in the compute instance. This design choice brings benefits of by-passing OS kernel, re-using database-specific optimizations like sophisticated LRU algorithms, and leveraging data access patterns to derive new optimizations such as variable-sized RDMA operations.

Infiniswap[22] is an RDMA memory paging system and exposes unused memory in remote servers to applications. Leap[2] further explores the prefetching strategy to improve the cache efficiency. Remote Region[1] proposes a file-like new abstraction for memory in the remote server. However, our study in Section 2 plus the recent study [45] identifies the significant performance loss when directly applying resource disaggregation techniques to the context of cloud-native databases, because these techniques are unaware of unique characteristics of databases. Therefore, we advocate the need for co-designing database and disaggregation architectures for fully exploiting the benefits of resource disaggregation.

**Disaggregated Databases.** Amazon Aurora moves logging and storage to distributed storage systems [42]. Alibaba PolarDB decouples the compute and storage resources [31]. TiDB [24] separates SQL layer from the underlying key-value storage engine. Furthermore, HailStorm[7] disaggregates and scales independently storage and computation for distributed LSM-tree-based databases. Though promising, they only consider the resource decoupling between the compute and persistent storage resources and neglect memory, which we find more important than other resources for database applications. Therefore, we propose a novel memory-disaggregated cloud-native relational database architecture and offer a holistic system approach to optimize its performance.

**Distributed Shared Memory.** Existing works [8, 16, 35] provide a global shared memory model abstracted from the distributed memory interconnected with RDMA network. However, most of these approaches either expose inappropriate interfaces for or require substantial rewriting of our targeted cloud-native databases. In contrast, LegoBase includes distributed shared memory as one of its key components, but encompasses many new database-oriented optimizations such as adopting page abstraction, two-tier LRU caching, and two-tier ARIES protocol.

## 9 CONCLUSION

LegoBase is a novel memory-disaggregated cloud-native database architecture. Experimental results with various workloads demonstrate that LegoBase is able to scale CPU and memory capacities independently with comparable performance as the monolithic setup without using remote memory, and achieves faster state recovery and is more cost-effective than state-of-the-art baselines.

## ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their insightful feedback. This work was supported in part by National Nature Science Foundation of China (61802358, 61832011), USTC Research Funds of the Double First-Class Initiative (YD2150002006), and Alibaba Group through Alibaba Innovative Research (AIR) Program.

## REFERENCES

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, 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)*. USENIX Association, Boston, MA, 775–787.

[2] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.

[3] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association.

[4] Krste Asanović. 2014. Firebox: A hardware building block for 2020 warehouse-scale computers. (2014).

[5] TPC Benchmark. 2020. TPC-C. http://www.tpc.org/tpcc/. "[accessed-Dec-2020]".

[6] TPC Benchmark. 2020. TPC-H. http://www.tpc.org/tpch/. "[accessed-Dec-2020]".

[7] 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.

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

[9] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 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.

[10] 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.

[11] Yue Cheng, Ali Anwar, and Xuejing Duan. 2018. Analyzing alibaba's co-located datacenter workloads. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 292–297.

[12] I-Hsin Chung, Bulent Abali, and Paul Crumley. 2018. Towards a composable computer system. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 137–147.

[13] HP Development Company. 2020. The Machine: A New Kind of Computer. https://www.hpl.hp.com/research/systems-research/themachine/. "[accessed-Oct-2020]".

[14] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[15] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically indexing millions of databases in microsoft azure sql database. In *Proceedings of the 2019 International Conference on Management of Data*. 666–679.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[17] Facebook. 2013. Future rack technology. https://newsroom.intel.com/news-releases/intel-facebook-collaborate-on-future-data-center-rack-technologies/. "[accessed-Oct-2020]".

[18] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[19] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 249–264.

[20] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 455–466.

[21] The PostgreSQL Global Development Group. 2021. PostgreSQL. https://www.postgresql.org/. "[accessed-April-2021]".

[22] 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.

[23] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 15–26.

[24] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[25] Docker Inc. 2020. Docker. https://www.docker.com/. "[accessed-Dec-2020]".

[26] Intel. 2020. Rack Scale Architecture. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html. "[accessed-Oct-2020]".

[27] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, K Hasharoni, Daniel Raho, Christian Pinto, F Espina, et al. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 690–695.

[28] Kenneth C Knowlton. 1965. A fast storage allocator. *Commun. ACM* 8, 10 (1965), 623–624.

[29] Alexey Kopytov. 2012. Sysbench manual. *MySQL AB* (2012), 2–3.

[30] Willis Lang, Frank Bertsch, David J DeWitt, and Nigel Ellis. 2015. Microsoft azure SQL database telemetry. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 189–194.

[31] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[32] Huan Liu. 2011. A measurement study of server utilization in public clouds. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 435–442.

[33] 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.

[34] MySQL. 2015. MySQL Buffer Pool LRU Algorithm. https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool.html. "[accessed-Dec-2020]".

[35] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 291–305.

[36] Joe Novak, Sneha Kumar Kasera, and Ryan Stutsman. 2020. Auto-Scaling Cloud-Based Memory-Intensive Applications. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 229–237.

[37] James S Plank. 1997. A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems. *Software: Practice and Experience* 27, 9 (1997), 995–1012.

[38] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. 2018. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–33.

[39] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[40] 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)*. USENIX Association, 69–87.

[41] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, et al. 2019. Native store extension for SAP HANA. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2047–2058.

[42] 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.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[44] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 307–320.

[45] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the effect of data center resource disaggregation on production DBMSs. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1568–1581.